Qualcomm Technologies, Inc.

Qualcomm® Hexagon™ V75

Programmer's Reference Manual

80-N2040-57 Rev. AB

January 29, 2024

All Qualcomm products mentioned herein are products of Qualcomm Technologies, Inc. and/or its subsidiaries.

Qualcomm and Hexagon are trademarks or registered trademarks of Qualcomm Incorporated. Other product and brand names may be trademarks or registered trademarks of their respective owners.

This technical data may be subject to U.S. and international export, re-export, or transfer ("export") laws. Diversion contrary to U.S. and international law is strictly prohibited.

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

© 2023-2024 Qualcomm Technologies, Inc. and/or its subsidiaries. All rights reserved.

Contents

Figures 14
Tables 15

1 Introduction 17

1.1 Hexagon V75 processor architecture 17
1.1.1 Memory 17
1.1.1.1 Cache memory 17
1.1.1.2 Virtual memory 17
1.1.2 Registers 17
1.1.3 Instruction sequencer 18
1.1.4 Execution units 19
1.1.5 Load/store units 19
1.2 Instruction set 19
1.2.1 Addressing modes 20
1.2.2 Program flow 20
1.2.3 Instruction pipeline 21
1.3 Technical assistance 21

2 Registers 22

2.1 General registers 22
2.2 Control registers 24
2.2.1 Program counter 26
2.2.2 Loop registers 26
2.2.3 User status register 27
2.2.4 Modifier registers 30
2.2.5 Predicate registers 31
2.2.6 Circular start registers 31
2.2.7 User general pointer register 31
2.2.8 Global pointer 32
2.2.9 Cycle count registers 32
2.2.10 Frame limit register 32
2.2.11 Frame key register 33
2.2.12 Packet count registers 33
2.2.13 QTimer registers 34

3 Instructions 35

3.1 Hexagon processor instruction syntax 35
3.1.1 Numeric operands 36
3.1.2 Terminology 37
3.1.3 Register operands 38
3.2 Instruction classes 40
3.3 Instruction packets 41
3.3.1 Packet execution semantics 42
3.3.2 Sequencing semantics 42
3.3.3 Resource constraints 43
3.3.4 Grouping constraints 44
3.3.5 Dependency constraints 45
3.3.6 Ordering constraints 45
3.3.7 Alignment constraints 46
3.4 Instruction intrinsics 46
3.5 Compound instructions 47
3.6 Duplex instructions 47

4 Data processing 48

4.1 Data types 48
4.1.1 Fixed-point data 48
4.1.1.1 Scalar operations 48
4.1.1.2 Vector operations 49
4.1.2 Floating-point data 50
4.1.2.1 Floating-point operations 50
4.1.3 Complex data 50
4.1.4 Vector data 50
4.2 Instruction options 52
4.2.1 Fractional scaling 52
4.2.2 Saturation 52
4.2.3 Arithmetic rounding 52
4.2.4 Convergent rounding 53
4.2.5 Scaling for divide and square-root 53
4.3 XTYPE operations 54
4.3.1 ALU 54
4.3.2 Bit manipulation 54
4.3.3 Complex 55
4.3.4 Floating point 55
4.3.5 Multiply 56
4.3.6 Permute 58
4.3.7 Predicate 58
4.3.8 Shift 59

4.4 ALU32 operations 60
4.5 Vector operations 60
4.6 CR operations 62
4.7 Compound operations 62
4.8 Special operations 63
4.8.1 H.264 CABAC processing 63
4.8.1.1 CABAC implementation 64
4.8.1.2 Code example 65
4.8.2 IP Internet checksum 66
4.8.2.1 Code example 67
4.8.3 Software-defined radio 68
4.8.3.1 Rake despreading 68
4.8.3.2 Polynomial operations 69

5 Memory 70

5.1 Hexagon processor memory model 70
5.1.1 Address space 70
5.1.2 Byte order 70
5.1.3 Alignment 71
5.2 Memory loads 72
5.3 Memory stores 73
5.4 Dual stores 73
5.5 Slot 1 store with slot 0 load 74
5.6 New-value stores 74
5.7 Mem-ops 75
5.8 Addressing modes 75
5.8.1 Absolute 76
5.8.2 Absolute-set 76
5.8.3 Absolute with register offset 76
5.8.4 Global pointer relative 77
5.8.5 Indirect 78
5.8.6 Indirect with offset 78
5.8.7 Indirect with register offset 78
5.8.8 Indirect with auto-increment immediate 79
5.8.9 Indirect with auto-increment register 79
5.8.10 Circular with auto-increment immediate 80
5.8.11 Circular with auto-increment register 81
5.8.12 Bit-reversed with auto-increment register 82
5.9 Conditional load/stores 83
5.10 Cache memory 85
5.10.1 Uncached memory 85
5.10.2 Tightly coupled memory 86
5.10.3 Cache maintenance operations 86

5.10.4 L2 cache operations 87
5.10.5 Cache line zero 87
5.10.6 Cache prefetch types supported by the Hexagon processor 87
5.10.6.1 Hardware-based instruction cache prefetching 87
5.10.6.2 Software-based data cache prefetching 87
5.10.6.3 Software-based L2fetch 88
5.10.6.4 Hardware-based data cache prefetching 89
5.11 Memory ordering 90
5.12 Atomic operations 91

6 Conditional execution 93

6.1 Scalar predicates 93
6.1.1 Generating scalar predicates 93
6.1.2 Consuming scalar predicates 95
6.1.3 Auto-AND predicates 96
6.1.4 Dot-new predicates 97
6.1.5 Dependency constraints 98
6.2 Vector predicates 98
6.2.1 Vector compare 98
6.2.2 Vector mux instruction 100
6.2.3 Using vector conditionals 100
6.3 Predicate operations 101

7 Software stack 102

7.1 Stack structure 102
7.2 Stack frames 103
7.3 Stack protection 103
7.3.1 Stack bounds checking 103
7.3.2 Stack smashing protection 104
7.4 Stack registers 104
7.5 Stack instructions 105

8 Program flow 106

8.1 Conditional instructions 106
8.2 Hardware loops 106
8.2.1 Loop setup 108
8.2.2 Loop end 109
8.2.3 Loop execution 110
8.2.4 Pipelined hardware loops 110
8.2.5 Loop restrictions 113
8.3 Software branches 113
8.3.1 Jumps 114

8.3.2 Calls 114
8.3.3 Returns 115
8.3.4 Extended branches 116
8.3.5 Branches to and from packets 116
8.4 Speculative jumps 117
8.5 Compare jumps 118
8.5.1 New-value compare jumps 118
8.6 Register transfer jumps 120
8.7 Dual jumps 121
8.8 Hint indirect jump target 121
8.9 Pauses 122
8.10 Exceptions 122

9 PMU events 125

9.1 V75 processor event symbols 125

10 Instruction encoding 142

10.1 Instructions 142
10.2 Sub-instructions 144
10.3 Duplexes 146
10.4 Instruction classes 148
10.5 Instruction packets 149
10.6 Loop packets 150
10.7 Immediate values 151
10.8 Scaled immediate values 151
10.9 Constant extenders 152
10.10 New-value operands 155
10.11 Instruction mapping 155

11 Instruction set 156

11.1 ALU32 157
11.1.1 ALU32 ALU 157
Logical operations 159
Negate 161
NOP 162
Subtract 163
Sign extend 164
Transfer immediate 165
Transfer register 167
Vector add halfwords 168
Vector average halfwords 169
Vector subtract halfwords 170

Zero extend 171
11.1.2 ALU32 PERM 172
Mux 174
Shift word by 16 176
Pack high and low halfwords 177
11.1.3 ALU32 PRED 178
Conditional shift halfword 180
Conditional combine 182
Conditional logical operations 183
Conditional subtract 185
Conditional sign extend 186
Conditional transfer 188
Conditional zero extend 189
Compare 190
Compare to general register 192
11.2 CR 193
Corner detection acceleration 194
Logical reductions on predicates 195
Looping instructions 196
Add to PC 198
Pipelined loop instructions 199
Logical operations on predicates 201
User control register transfer 203
11.3 JR 204
Hinted call subroutine from register 205
Hint an indirect jump address 206
Jump to address from register 207
Hinted jump to address from register 208
11.4 J 209
Compare and jump 211
Jump to address 215
Jump to address conditioned on new predicate 216
Jump to address condition on register value 217
Transfer and jump 219
11.5 LD 220
Load-acquire doubleword 222
Load doubleword conditionally 223
Load byte 225
Load byte conditionally 227
Load byte into shifted vector 229
Load half into shifted vector 231
Load halfword 234
Load halfword conditionally 236

Load unsigned byte 238
Load unsigned byte conditionally....240
Load unsigned halfword 242
Load unsigned halfword conditionally 244
Load word....246
Load-acquire word 248
Load word conditionally....249
Deallocate stack frame....251
Deallocate frame and return....253
Load and unpack bytes to halfwords 255
11.6 MEMOP 263
Operation on memory halfword 264
Operation on memory word 265
11.7 NV....266
11.7.1 NV J 266
11.7.2 NV ST....270
Store new-value byte conditionally....272
Store new-value halfword 274
Store new-value halfword conditionally 276
Store new-value word 278
Store new-value word conditionally 280
11.8 ST....282
Store-release doubleword....284
Store doubleword conditionally....285
Store byte....287
Store byte conditionally 289
Store halfword....291
Store halfword conditionally....294
Release 297
Store word....298
Store-release word....300
Store word conditionally 301
Allocate stack frame....303
11.9 System 305
11.9.1 System user....305
Store conditional 306
Zero a cache line 307
Memory barrier....308
Breakpoint....309
Data cache prefetch 310
Data cache multi prefetch....311
Data cache maintenance user operations....312
Send value to DIAG trace 314

Instruction cache maintenance user operations....315
Instruction synchronization....316
L2 cache prefetch....317
Pause....320
Memory thread synchronization....321
Send value to ETM trace....322
Trap....323
Unpause....324
11.10 XTYPE....325
11.10.1 XTYPE ALU....325
Absolute value word....326
Add and accumulate....327
Add doublewords....329
Add halfword....331
Add or subtract doublewords with carry....333
Clip to unsigned....334
Logical doublewords....335
Logical-logical doublewords....336
Logical-logical words....337
Maximum words....339
Maximum doublewords....340
Minimum words....341
Minimum doublewords....342
Modulo wrap....343
Negate....344
Round....345
Subtract doublewords....348
Subtract and accumulate words....349
Subtract halfword....350
Sign extend word to doubleword....352
Vector absolute value halfwords....353
Vector absolute value words....354
Vector absolute difference bytes....355
Vector absolute difference halfwords....356
Vector absolute difference words....357
Vector add compare and select maximum bytes....358
Vector add compare and select maximum halfwords....359
Vector add halfwords....361
Vector add halfwords with saturate and pack to unsigned bytes....362
Vector reduce add unsigned bytes....363
Vector reduce add halfwords....365
Vector add bytes....367
Vector add words....368

Vector average halfwords 369
Vector average unsigned bytes 371
Vector average words 372
Vector clip to unsigned 374
Vector conditional negate 375
Vector maximum bytes 376
Vector maximum halfwords 377
Vector reduce maximum halfwords 378
Vector reduce maximum words 380
Vector maximum words 382
Vector minimum bytes 383
Vector minimum halfwords 384
Vector reduce minimum halfwords 385
Vector reduce minimum words 386
Vector minimum words 388
Vector sum of absolute differences unsigned bytes 389
Vector subtract halfwords 391
Vector subtract bytes 393
Vector subtract words 394
11.10.2 XTYPE bit 395
Count population 397
Count trailing 398
Extract bit field 399
Insert bit field 402
Interleave/deinterleave 404
Linear feedback-shift iteration 405
Masked parity 406
Bit reverse 407
Set/clear/toggle bit 408
Split bit field 410
Table index 412
11.10.3 XTYPE complex 414
Complex add/sub words 416
Complex multiply 418
Complex multiply real or imaginary 421
Complex multiply with round and pack 423
Complex multiply 32x16 425
Complex multiply real or imaginary 32-bit 427
Vector complex multiply real or imaginary 431
Vector complex conjugate 433
Vector complex rotate 434
Vector reduce complex multiply by scalar 436
Vector reduce complex multiply by scalar with round and pack 439

Vector reduce complex rotate 441
11.10.4 XTYPE FP 444
Classify floating-point value 445
Compare floating-point value 446
Convert floating-point value to other format 448
Convert integer to floating-point value 449
Convert floating-point value to integer 451
Floating point extreme value assistance 453
Floating point fused multiply-add 454
Floating point fused multiply-add with scaling 455
Floating point reciprocal square root approximation 456
Floating point fused multiply-add for library routines 457
Create floating-point constant 459
Floating point maximum 460
Floating point minimum 461
Floating point multiply 462
Floating point reciprocal approximation 463
Floating point subtraction 464
11.10.5 XTYPE MPY 465
Vector multiply word by signed half (32×16) 468
Vector multiply word by unsigned half (32×16) 471
Multiply signed halfwords 475
Multiply unsigned halfwords 482
Polynomial multiply words 487
Vector reduce multiply word by signed half (32×16) 489
Multiply and use upper result 491
Multiply and use full result 493
Vector dual multiply 495
Vector dual multiply with round and pack 497
Vector reduce multiply bytes 499
Vector dual multiply signed by unsigned bytes 501
Vector multiply even halfwords 503
Vector multiply halfwords 505
Vector multiply halfwords with round and pack 507
Vector multiply halfwords, signed by unsigned 509
Vector reduce multiply halfwords 510
Vector multiply bytes 512
Vector polynomial multiply halfwords 514
11.10.6 XTYPE perm 516
Saturate 518
Swizzle bytes 520
Vector align 521
Vector round and pack 522

Vector saturate and pack 524
Vector saturate without pack 526
Vector shuffle 528
Vector splat bytes 530
Vector splat halfwords 531
Vector splice 532
Vector sign extend 533
Vector truncate 534
Vector zero extend 536
11.10.7 XTYPE PRED 537
Compare byte 538
Compare half 539
Compare doublewords 541
Compare bit mask 542
Mask generate from predicate 543
Check for TLB match 544
Predicate transfer 545
Test bit 546
Vector compare halfwords 547
Vector compare bytes for any match 549
Vector compare bytes 550
Vector compare words 552
Viterbi pack even and odd predicate bits 554
Vector mux 555
11.10.8 XTYPE SHIFT 556
Shift by immediate 557
Shift by immediate and accumulate 559
Shift by immediate and add 562
Shift by immediate and logical 563
Shift right by immediate with rounding 567
Shift left by immediate with saturation 569
Shift by register 570
Shift by register and accumulate 573
Shift by register and logical 576
Shift by register with saturation 580
Vector shift halfwords by immediate 581
Vector arithmetic shift halfwords with round 582
Vector arithmetic shift halfwords with saturate and pack 583
Vector shift halfwords by register 585
Vector shift words by immediate 587
Vector shift words by register 588
Vector shift words with truncate and pack 590
Instruction Index 592

Intrinsics Index 620

Figures

Figure 1-1 Hexagon V75 processor architecture 19
Figure 1-2 Vector instruction example 23
Figure 1-3 Instruction classes and combinations 26
Figure 1-4 Register field symbols 29
Figure 2-1 General registers 32
Figure 2-2 Control registers 35
Figure 3-1 Packet grouping combinations 51
Figure 4-1 Vector byte operation 58
Figure 4-2 Vector halfword operation 58
Figure 4-3 Vector word operation 58
Figure 4-4 64-bit shift and add/sub/logical 67
Figure 4-5 Vector halfword shift right 70
Figure 5-1 Hexagon processor byte order 80
Figure 5-2 L2fetch instruction 99
Figure 6-1 Vector byte compare 110
Figure 6-2 Vector halfword compare 110
Figure 6-3 Vector mux instruction 111
Figure 7-1 Stack structure 115
Figure 10-1 Instruction packet encoding 158

Tables

Table 1-1 Register symbols . . . 28
Table 1-2 Register bit field symbols . . . 29
Table 1-3 Instruction operands . . . 30
Table 1-4 Data symbols . . . 31
Table 2-1 General register aliases . . . 34
Table 2-2 General register pairs . . . 34
Table 2-3 Aliased control registers . . . 36
Table 2-4 Control register pairs . . . 37
Table 2-5 Loop registers . . . 38
Table 2-6 User status register . . . 39
Table 2-7 Modifier registers (indirect auto-increment addressing) . . . 41
Table 2-8 Modifier registers (circular addressing) . . . 41
Table 2-9 Modifier registers (bit-reversed addressing) . . . 42
Table 2-10 Predicate registers . . . 42
Table 2-11 Circular start registers . . . 43
Table 2-12 User general pointer register . . . 43
Table 2-13 Global pointer register . . . 43
Table 2-14 Cycle count registers . . . 44
Table 2-15 Frame limit register . . . 44
Table 2-16 Frame key register . . . 45
Table 2-17 Packet count registers . . . 45
Table 2-18 Qtimer registers . . . 46
Table 3-1 Instruction symbols . . . 47
Table 3-2 Instruction classes . . . 48
Table 4-1 Single-precision multiply options . . . 65
Table 4-2 Double precision multiply options . . . 65
Table 4-3 Control register transfer instructions . . . 71
Table 5-1 Memory alignment restrictions . . . 81
Table 5-2 Load instructions . . . 81
Table 5-3 Store instructions . . . 82
Table 5-4 Mem-ops . . . 84
Table 5-5 Addressing modes . . . 84
Table 5-6 Offset ranges (global pointer relative) . . . 86
Table 5-7 Offset ranges (indirect with offset) . . . 87
Table 5-8 Increment ranges (indirect with auto-inc immediate) . . . 88
Table 5-9 Increment ranges (circular with auto-inc immediate) . . . 89
Table 5-10 Increment ranges (circular with auto-inc register) . . . 91
Table 5-11 Addressing modes (conditional load/store) . . . 93
Table 5-12 Conditional offset ranges (indirect with offset) . . . 94
Table 5-13 Cache instructions (user-level) . . . 96
Table 5-14 Memory ordering instructions . . . 100

Table 5-15 Atomic instructions 101
Table 6-1 Scalar predicate-generating instructions 104
Table 6-2 Vector mux instruction. 111
Table 6-3 Predicate register instructions 113
Table 7-1 Stack registers. 117
Table 7-2 Stack instructions 118
Table 8-1 Loop instructions 121
Table 8-2 Software pipelined loop. 125
Table 8-3 Software pipelined loop (using spNloop0) 126
Table 8-4 Software branch instructions 127
Table 8-5 Jump instructions 128
Table 8-6 Call instructions 128
Table 8-7 Return instructions. 129
Table 8-8 Speculative jump instructions 131
Table 8-9 Compare jump instructions 133
Table 8-10 New-value compare jump instructions. 134
Table 8-11 Register transfer jump instructions 135
Table 8-12 Dual jump instructions 135
Table 8-13 Jump hint instruction 136
Table 8-14 Pause instruction 137
Table 8-15 V75 exceptions 138
Table 9-1 V75 processor events symbols 141
Table 10-1 Instruction fields. 151
Table 10-2 Sub-instructions 153
Table 10-3 Sub-instruction registers 154
Table 10-4 Duplex instruction 155
Table 10-5 Duplex ICLASS field. 155
Table 10-6 Instruction class encoding 157
Table 10-7 Loop packet encoding 159
Table 10-8 Scaled immediate encoding (indirect offsets) 160
Table 10-9 Constant extender encoding 161
Table 10-10 Constant extender instructions. 162
Table 10-11 Instruction mapping 165
Table 11-1 Instruction operand symbols. 166
Table 11-2 Instruction behavior symbols 167

1 Introduction

The Qualcomm Hexagon™ processor is a general-purpose digital signal processor designed for high performance and low power across a wide variety of multimedia and modem applications. V75 is a member of the sixth generation of the Hexagon processor architecture.

1.1 Hexagon V75 processor architecture

1.1.1 Memory

The Hexagon processor features a unified byte-addressable memory. This memory has a single 32-bit virtual address space, which holds both instructions and data. It operates in little endian mode.

The load/store architecture supports a complete set of addressing modes for both compiler code generation and DSP application programming.

1.1.1.1 Cache memory

Memory accesses are cached or uncached. Separate L1 instruction and data caches exist for program code and data. A unified L2 cache is partly or wholly configured as tightly coupled memory (TCM).

1.1.1.2 Virtual memory

Memory is addressed virtually, with virtual-to-physical memory mapping handled by a resident OS. Virtual memory supports the implementation of memory management and memory protection in a hardware-independent manner.

1.1.2 Registers

The Hexagon processor has two sets of registers: General registers and Control registers.

The general registers include thirty-two 32-bit registers (named R0 through R31), which are accessed either as single registers or as aligned 64-bit register pairs. The general registers contain all data, including pointer, scalar, vector, and accumulator data.

The control registers include special-purpose registers such as program counter, status register, loop registers, and so on.

1.1.3 Instruction sequencer

The instruction sequencer processes packets of one to four instructions in each cycle. If a packet contains more than one instruction, the instructions execute in parallel.

The instruction combinations allowed in a packet are limited to the instruction types that can execute in parallel in the four execution units (shown in Figure 1-1).

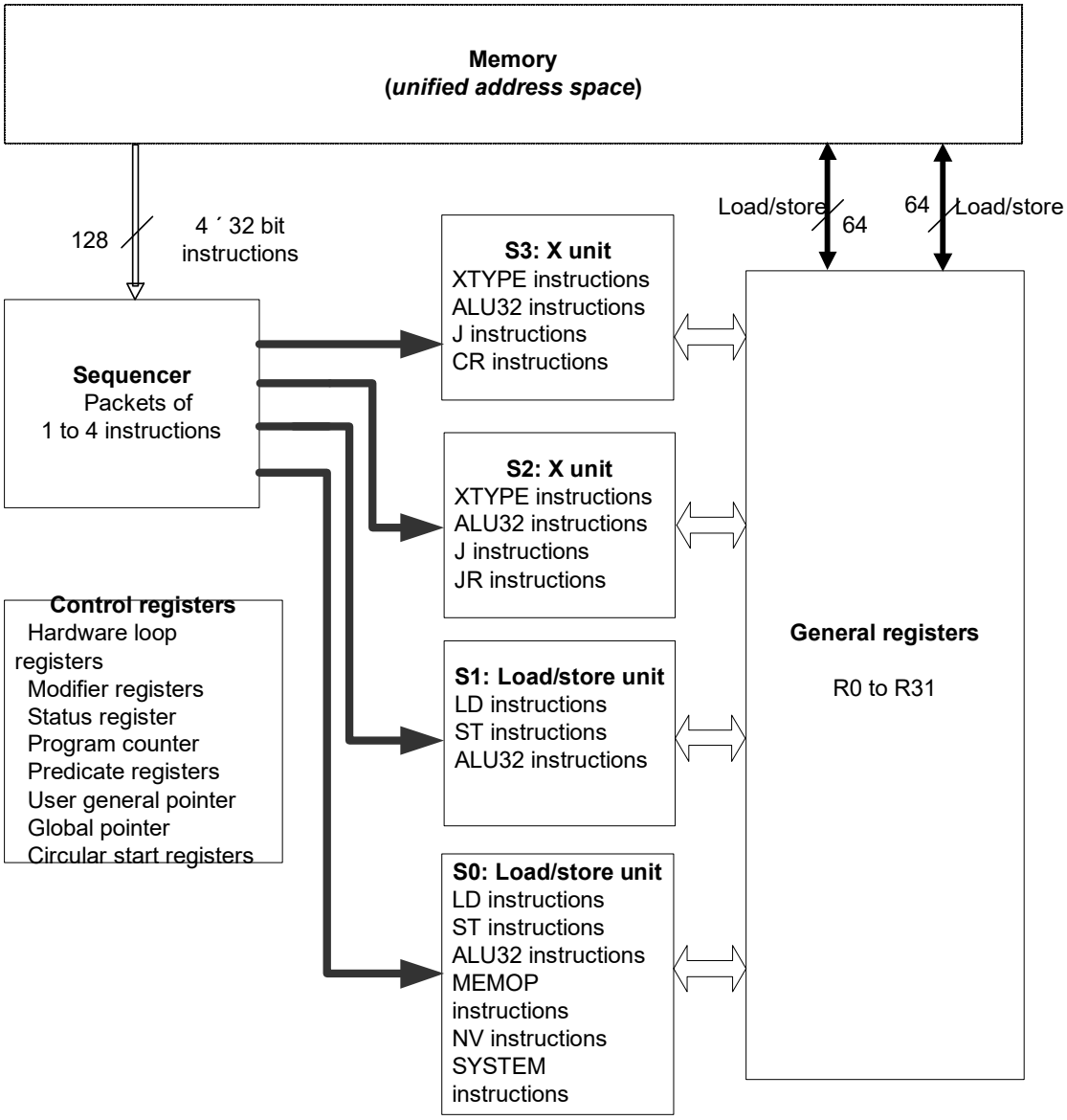


Figure 1-1 Hexagon V75 processor architecture

1.1.4 Execution units

The dual execution units are identical: each includes a 64-bit shifter and a vector multiply/accumulate unit with four 16 × 16 multipliers to support both scalar and vector instructions.

These units also perform 32-bit and 64-bit ALU instructions, and jump and loop instructions.

NOTE: Each execution unit supports floating-point instructions.

1.1.5 Load/store units

The two load/store units can operate on signed or unsigned bytes, halfwords (16-bit), words (32-bit), or double words (64-bit).

To increase the number of instruction combinations allowed in packets, the load units also support 32-bit ALU instructions.

1.2 Instruction set

For the Hexagon processor to achieve large amounts of work per cycle, the instruction set is designed with the following properties:

- Static grouping (VLIW) architecture
- Static fusing of simple dependent instructions
- Extensive compound instructions
- A large set of SIMD and application-specific instructions

To support efficient compilation, the instruction set is orthogonal with respect to registers, addressing modes, and load/store access size.

1.2.1 Addressing modes

The Hexagon processor supports the following memory addressing modes:

- 32-bit absolute
- 32-bit absolute-set
- Absolute with register offset
- Global pointer relative
- Indirect
- Indirect with offset
- Indirect with register offset
- Indirect with auto-increment (immediate or register)
- Circular with auto-increment (immediate or register)
- Bit-reversed with auto-increment register

```
For example:
R2 = memw(##myvariable)
R2 = memw(R3=##myvariable)
R2 = memw(R4<<#3+##myvariable)
R2 = memw(GP+#200)
R2 = memw(R1)
R2 = memw(R3+#100)
R2 = memw(R3+R4<<#2)
R2 = memw(R3++#4)
R2 = memw(R0++M1)
R0 = memw(R2++#8:circ(M0))
R0 = memw(R2++I:circ(M0))
R2 = memw(R0++M1:brev)
```

Auto-increment with register addressing uses one of the two dedicated address-modify registers M0 and M1 (which are part of the control registers).

NOTE: Atomic memory operations (load locked/store conditional) are supported to implement multithread synchronization.

1.2.2 Program flow

The Hexagon processor supports zero-overhead hardware loops. For example:

```
  loop0(start,#3)      // loop 3 times
start:
    { R0 = mpyi(R0,R0) } :endloop0
```

The loop instructions support nestable loops, with few restrictions on their use.

Software branches use a predicated branch mechanism. Explicit compare instructions generate a predicate bit, which conditional branch instructions then test. For example:

```
P1 = cmp.eq(R2, R3)
if (P1) jump end
```

Jumps and subroutine calls are conditional or unconditional, and support both PC-relative and register indirect addressing modes. For example:

```
jump end
jumpr R1
call function
callr R2
```

The subroutine call instructions store the return address in register R31. Subroutine returns are performed using a jump indirect instruction through this register. For example:

```
jumpr R31     // Subroutine return
```

Two program flow instructions can be grouped into one packet.

1.2.3 Instruction pipeline

Hardware resolves pipeline hazards: pipeline restrictions do not constrain instruction scheduling.

1.3 Technical assistance

For assistance or clarification on information in this document, open a technical support case at https://support.qualcomm.com/.

You will need to register for a Qualcomm ID account and your company must have support enabled to access our Case system.

Other systems and support resources are listed on https://qualcomm.com/support.

If you need further assistance, you can send an email to qualcomm.support@qti.qualcomm.com.

2 Registers

General registers are used for general-purpose computation, including address generation, and scalar and vector arithmetic.

Control registers support special-purpose processor features such as hardware loops and predicates.

2.1 General registers

The Hexagon processor has thirty-two 32-bit general-purpose registers (named R0 through R31). These registers store operands in the instructions:

- Memory addresses for load/store instructions
- Data operands for arithmetic/logic instructions
- Vector operands for vector instructions

For example:

```
R1 = memh(R0)              // Load from address R0
R4 = add(R2,R3)            // Add
R28 = vaddh(R11,R10)       // Vector add halfword
```

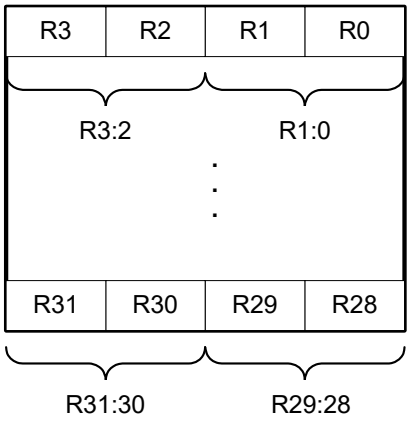


Figure 2-1 General registers

Aliased registers

Three of the general registers – R29 through R31 – support subroutines (Section 8.3.2) and the Software stack. The subroutine and stack instructions implicitly modify the registers. They have symbol aliases that indicate when these registers are accessed as subroutine and stack registers.

For example:

```
SP = add(SP, #-8)    // SP is an alias of R29
allocframe           // Modifies SP (R29) and FP (R30)
call init            // Modifies LR (R31)
```

Table 2-1 General register aliases

Register	Alias	Name	Description
R29	SP	Stack pointer	Points to the topmost element of the stack in memory.
R30	FP	Frame pointer	Points to the current procedure frame on the stack. Used by external debuggers to examine the stack and determine call sequence, parameters, local variables, and so on.
R31	LR	Link register	Stores the return address of a subroutine call.

Register pairs

The general registers can be specified as register pairs that represent a single 64-bit register. For example:

```
R1:0 = memd(R3)                    // Load doubleword
R7:6 = valignb(R9:8,R7:6, #2)      // Vector align
```

NOTE: The first register in a register pair must always be odd-numbered, and the second must be the next lower register.

Table 2-1 General register pairs

Register	Register pair
R0	R1:0
R1	
R2	R3:2
R3	
R4	R5:4
R5	
R6	R7:6
R7	
...	
R24	R25:24
R25	
R26	R27:26
R27	
R28	R29:28
R29 (SP)	
R30 (FP)	R31:30 (LR:FP)
R31 (LR)	

2.2 Control registers

The Hexagon processor includes a set of 32-bit control registers that provide access to processor features such as the program counter, hardware loops, and vector predicates.

Unlike general registers, control registers are used as instruction operands only in the following cases:

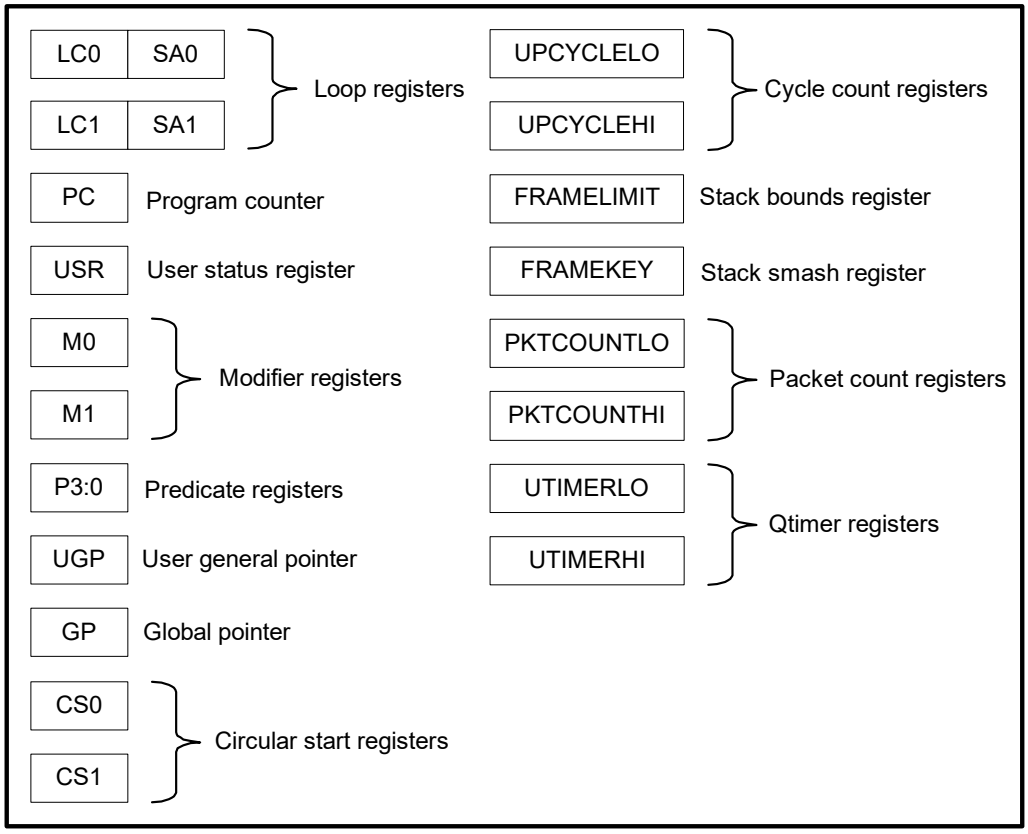
- Instructions that require a specific control register as an operand
- Register transfer instructions

For example:

```
R2 = memw(R0++M1)    // Auto-increment addressing mode (M1)
R9 = PC              // Get program counter (PC)
LC1 = R3             // Set hardware loop count (LC1)
```

NOTE: When a control register is used in a register transfer, the other operand must be a general register.

Figure 2-2 Control registers



Aliased registers

The control registers have numeric aliases (C0 through C31).

Table 2-2 Aliased control registers

Register	Alias	Name
SA0	C0	Loop start address register 0
LC0	C1	Loop count register 0
SA1	C2	Loop start address register 1
LC1	C3	Loop count register 1
P3:0	C4	Predicate registers 3:0
reserved	C5	–
M0	C6	Modifier register 0
M1	C7	Modifier register 1
USR	C8	User status register
PC	C9	Program counter
UGP	C10	User general pointer
GP	C11	Global pointer
CS0	C12	Circular start register 0
CS1	C13	Circular start register 1
UPCYCLELO	C14	Cycle count register (low)
UPCYCLEHI	C15	Cycle count register (high)
UPCYCLE	C15:14	Cycle count register
FRAMELIMIT	C16	Frame limit register
FRAMEKEY	C17	Frame key register
PKTCOUNTLO	C18	Packet count register (low)
PKTCOUNTHI	C19	Packet count register (high)
PKTCOUNT	C19:18	Packet count register
reserved	C20-29	–
UTIMERLO	C30	QTimer register (low)
UTIMERHI	C31	QTimer register (high)
UTIMER	C31:30	QTimer register

NOTE: The control register numbers (0 through 31) specify the control registers in Instruction encodings.

Register pairs

The control registers can be specified as register pairs that represent a single 64-bit register. Control registers specified as pairs must use their numeric aliases. For example:

```
C1:0 = R5:4     // C1:0 specifies the LC0/SA0 register pair
```

NOTE: The first register in a control register pair must always be odd-numbered, and the second must be the next lower register.

Table 2-2 Control register pairs

Register	Register pair
C0	C1:0
C1	
C2	C3:2
C3	
C4	C5:4
C5	
C6	C7:6
C7	
...	
C30	C31:30
C31	

2.2.1 Program counter

The program counter (PC) register points to the next instruction packet to execute (Section 3.3). Instruction execution implicitly modifies the PC register, but PC an be read directly. For example:

```
R7 = PC      // Get program counter
```

NOTE: The PC register is read only: writing to it has no effect.

2.2.2 Loop registers

The Hexagon processor includes two sets of loop registers to support nested Hardware loops. Each hardware loop is implemented with a pair of registers containing the loop count and loop start address. The loop instruction implicitly modifies the loop registers, but the loop instructions can also be accessed directly. For example:

```
loop0(start, R4)  // Modifies LC0 and SA0 (LC0 = R4, SA0 = &start)
LC1 = R22         // Set loop1 count
R9 = SA1          // Get loop1 start address
```

Table 2-3 Loop registers

Register	Name	Description
LC0, LC1	Loop count	Number of loop iterations to execute.
SA0, SA1	Loop start address	Address of first instruction in loop.

2.2.3 User status register

The user status register (USR) stores processor status and control bits that user programs can access. The status bits contain the status results of certain instructions, while the control bits contain user-settable processor modes for hardware prefetching. For example:

```
R9:8 = vaddw(R9:8, R3:2):sat      // Vector add words
R6 = USR                          // Get saturation status
```

USR stores the following status and control values:

- Cache prefetch types supported by the Hexagon processor enable
- Cache prefetch status
- Floating point modes
- Floating point status
- Hardware loop configuration (Section 8.2)
- Sticky Saturation overflow

NOTE: A user control register transfer to USR cannot be grouped in an instruction packet with a Floating point instruction.

When a transfer to USR changes the enable trap bits [29:25], an isync instruction (Section 5.11) must execute before the new exception programming can take effect.

Table 2-4 User status register

Name	RW	Bits	Field	Description
USR		32		User status register
	R	31	PFA	L2 prefetch active. 1: l2fetch instruction in progress 0: l2fetch finished (or inactive) Set when the nonblocking l2fetch instruction prefetches therequested data. Remains set until thel2fetch prefetch operation completes (or is inactive).
	R	30	reserved	Return 0 if read. Reserved for future expansion. To remain compatible with future processor versions, software should always write this field with the same value read from the field.
	RW	29	FPINEE	Enable trap on IEEE inexact.
	RW	28	FPUNFE	Enable trap on IEEE underflow.
	RW	27	FPOVFE	Enable trap on IEEE overflow.
	RW	26	FPDBZE	Enable trap on IEEE divide-by-zero.
	RW	25	FPINVE	Enable trap on IEEE invalid.
	R	24	reserved	Reserved
	RW	23:22	FPRND	Rounding mode for floating point instructions. 00: Round to nearest, ties to even (default) 01: Toward zero 10: Downward (toward negative infinity) 11: Upward (toward positive infinity)
	R	21:20	reserved	Return 0 if read. Reserved for future expansion. To remain compatible with future processor versions, software should always write this field with the same value read from the field.
	R	19:18	reserved	Reserved
	R	17	reserved	Return 0 if read. Reserved for future expansion. To remain compatible with future processor versions, software should always write this field with the same value read from the field.
	RW	16:15	HFI	L1 instruction prefetch. 00: Disable 01: Enable (1 line) 10: Enable (2 lines)

Table 2-4 User status register (cont.)

Name	RW	Bits	Field	Description
	RW	14:13	HFD	L1 data cache prefetch. Four levels are defined from disabled to aggressive. Implementation defines how to interpret these levels. 00: disable 01: conservative 10: moderate 11: aggressive
	RW	12	PCMME	Enable packet counting in Monitor mode.
	RW	11	PCGME	Enable packet counting in Guest mode.
	RW	10	PCUME	Enable packet counting in User mode.
	RW	9:8	LPCFGE	Hardware loop configuration. Number of loop iterations (0 to 3) that remain before the pipeline predicate should be set.
	R	7:6	reserved	Return 0 if read. Reserved for future expansion. To remain compatible with future processor versions, software should always write this field with the same value read from the field.
	RW	5	FPINPF	Floating-point IEEE inexact sticky fag.
	RW	4	FPUNFF	Floating-point IEEE underflow sticky flag.
	RW	3	FPOVFF	Floating-point IEEE overflow sticky flag.
	RW	2	FPDBZF	Floating-point IEEE divide-by-zero sticky flag.
	RW	1	FPINVF	Floating-point IEEE invalid sticky flag.
	RW	0	OVF	Sticky saturation overflow. 1: saturation occurred 0: no saturation Set when saturation occurs while executing an instruction that specifies optional saturation. Remains set until explicitly cleared by a USR = Rs instruction.

2.2.4 Modifier registers

The modifier registers (M0 to M1) are used in the following addressing modes.

Indirect auto-increment

In Indirect with auto-increment register addressing, the modifier registers store a signed 32-bit value that specifies the increment (or decrement) value.

Table 2-5 Modifier registers used in indirect auto-increment addressing

Register	Name	Description
M0, M1	Increment	Signed auto-increment value.

Circular

In circular addressing (Section 5.8.10), the modifier registers store the circular buffer length and related "I" values.

Table 2-6 Modifier registers as used in circular addressing

Name	RW	Bits	Field	Description
M0, M1		32		Circular buffer specifier.
	RW	31:28	I[10:7]	I value (MSB - see Section 5.8.11)
	RW	27:24		0x0
	RW	23:17	I[6:0]	I value (LSB)
	RW	16:0	Length	Circular buffer length

Bit-reversed

In bit-reversed addressing (Section 5.8.12), the modifier registers store a signed 32-bit value that specifies the increment (or decrement) value.

Table 2-7 Modifier registers as used in bit-reversed addressing

Register	Name	Description
M0, M1	Increment	Signed auto-increment value.

2.2.5 Predicate registers

The predicate registers (P0 through P3) store the status results of the scalar and vector compare instructions (Chapter 6). For example:

```
P1 = cmp.eq(R2, R3)        // Scalar compare
if (P1) jump end           // Jump to address (conditional)
R8 = P1                    // Get compare status (P1 only)
P3:0 = R4                  // Set compare status (P0 through P3)
```

The four predicate registers can be specified as a register quadruple (P3:0) that represents a single 32-bit register.

NOTE: Unlike the other control registers, the predicate registers are only eight bits wide because vector compares return a maximum of eight status results.

Table 2-3 Predicate registers

Register	Bits	Description
P0, P1, P2, P3	8	Compare status results.
P3:0	32	Compare status results.
	31:24	P3 register
	23:16	P2 register
	15:8	P1 register
	7:0	P0 register

2.2.6 Circular start registers

The circular start registers (CS0 through CS1) store the start address of a circular buffer in circular addressing (Section 5.8.10). For example:

```
CS0 = R5                      // Set circ start register
M0 = R7                       // Set modifier register
R0 = memb(R2++#4:circ(M0))    // Load from circ buffer pointed
                              // to by CS0 with size/K vals in M0
```

Table 2-4 Circular start registers

Register	Name	Description
CS0, CS1	Circular start	Circular buffer start address.

2.2.7 User general pointer register

The user general pointer (UGP) register is a general-purpose control register. For example:

```
R9 = UGP      // Get UGP
UGP = R3      // Set UGP
```

NOTE: UGP typically stores the address of thread local storage.

Table 2-5 User general pointer register

Register	Name	Description
UGP	User general pointer	General-purpose control register.

2.2.8 Global pointer

The global pointer (GP) is used in GP-relative addressing. For example:

```
GP = R7 // Set GP
R2 = memw(GP+#200)     // GP-relative load
```

Table 2-8 Global pointer register

Name	R/W	Bits	Field	Description
GP		32		Global pointer register
	R/W	31:6	GDP	Global data pointer (Section 5.8.4).
	R	5:0	reserved	Return 0 if read. Reserved for future expansion. To remain forward-compatible with future processor versions, software should always write this field with the same value read from the field.

2.2.9 Cycle count registers

The cycle count registers (UPCYCLELO to UPCYCLEHI) store a 64-bit value containing the current number of processor cycles executed since the Hexagon processor was last reset. For example:

```
R5 = UPCYCLEHI     // Get cycle count (high)
R4 = UPCYCLELO     // Get cycle count (low)
R5:4 = UPCYCLE     // Get cycle count
```

NOTE: The RTOS must grant permission to access these registers. Without this permission, reading these registers from user code returns zero.

Table 2-6 Cycle count registers

Register	Name	Description
UPCYCLELO	Cycle count (low)	Processor cycle count (low 32 bits)
UPCYCLEHI	Cycle count (high)	Processor cycle count (high 32 bits)
UPCYCLE	Cycle count	Processor cycle count (64 bits)

2.2.10 Frame limit register

The frame limit register (FRAMELIMIT) stores the low address of the memory area reserved for the software stack (Section 7.3.1). For example:

```
R9 = FRAMELIMIT      // Get frame limit register
FRAMELIMIT = R3      // Set frame limit register
```

Table 2-9 Frame limit register

Register	Name	Description
FRAMELIMIT	Frame limit	Low address of software stack area.

2.2.11 Frame key register

The frame key register (FRAMEKEY) stores the key value that XOR-scrambles return addresses when they are stored on the software stack (Section 7.3.2). For example:

```
R2 = FRAMEKEY       // Get frame key register
FRAMEKEY = R1       // Set frame key register
```

Table 2-10 Frame key register

Register	Name	Description
FRAMEKEY	Frame key	Key to scramble return addresses stored on the software stack.

2.2.12 Packet count registers

The packet count registers (PKTCOUNTLO to PKTCOUNTHI) store a 64-bit value containing the current number of instruction packets executed since a PKTCOUNT register was last written to. For example:

```
R9 = PKTCOUNTHI      // Get packet count (high)
R8 = PKTCOUNTLO      // Get packet count (low)
R9:8 = PKTCOUNT      // Get packet count
```

Packet counting can be configured to operate only in specific sets of processor modes (for example, User mode only, or Guest and Monitor modes only). Bits [12:10] in the User status register control the configuration for each mode.

Packets with exceptions are not counted as committed packets.

NOTE: Each hardware thread has its own set of packet count registers.

The RTOS must grant permission to access these registers. Without this permission, reading these registers from user code returns zero.

When a value is written to a PKTCOUNT register, the 64-bit packet count value is incremented before the value is stored in the register.

Table 2-11 Packet count registers

Register	Name	Description
PKTCOUNTLO	Packet count (low)	Processor packet count (low 32 bits)
PKTCOUNTHI	Packet count (high)	Processor packet count (high 32 bits)
PKTCOUNT	Cycle count	Processor packet count (64 bits)

2.2.13 QTimer registers

The QTimer registers (UTIMERLO to UTIMERHI) provide access to the QTimer global reference count value. QTimer registers enable Hexagon software to read the 64-bit time value without having to perform an expensive advanced high-performance bus (AHB) load. For example:

```
R5 = UTIMERHI     // Get QTimer reference count (high)
R4 = UTIMERLO     // Get QTimer reference count (low)
R5:4 = UTIMER     // Get QTimer reference count
```

These registers are read only – hardware automatically updates these registers to contain the current QTimer value.

NOTE: The RTOS must grant permission to access these registers. Without this permission, reading these registers from user code returns zero.

Table 2-12 QTimer registers

Register	Name	Description
UTIMERLO	QTimer (low)	QTimer global reference count (low 32 bits)
UTIMERHI	QTimer (high)	QTimer global reference count (high 32 bits)
UTIMER	QTimer	QTimer global reference count (64 bits)

3 Instructions

Instruction encoding is described in Chapter 10. For detailed descriptions of the Hexagon processor instructions, see Chapter 11.

3.1 Hexagon processor instruction syntax

NOTE: The notation described here does not appear in actual assembly language instructions. It is used only to specify the instruction syntax and behavior.

Most Hexagon processor instructions have the following syntax:

```
dest = instr_name(source1,source2,...)[:option1][:option2]...
```

The item specified on the left-hand side (LHS) of the equation is assigned the value specified by the right-hand side (RHS). For example:

```
R2 = add(R3,R1)    // Add R3 and R1, assign result to R2
```

- `Courier New font` is used for instructions
- Square brackets enclose optional items, for example, [`:sat`] means that saturation is optional
- Braces indicate a choice of items, for example, `{Rs,#s16}` means that either Rs or a signed 16-bit immediate can be used

Table 3-1 Instruction symbols

Symbol	Example	Meaning
`=`	`R2 = R3`	Assignment of RHS to LHS.
`#`	`R1 = #1`	Immediate constant value.
`##`	`##2147483647`	32-bit immediate constant value.
`0x`	`0xBABE`	Hexadecimal number prefix.
`MEMxx`	`R2 = MEMxx(R3)`	Access memory. xx specifies the size and type of access.
`;`	`R2 = R3; R4 = R5;`	Instruction delimiter, or end of the instruction.
`{ … }`	`{R2 = R3; R5 = R6}`	Instruction packet delimiter; indicates a group of parallel instructions.
`( … )`	`R2 = memw(R0 + #100)`	Source list delimiter.
`:endloop`*X*	`:endloop0`	Loop end. `X` specifies a loop instruction (0 or 1).
`:t`	`if (P0.new) jump:t target`	Direction hint (jump taken).
`:nt`	`if (!P1.new) jump:nt target`	Direction hint (jump not taken).
`:sat`	`R2 = add(R1,R2):sat`	Saturate the result.
`:rnd`	`R2 = mpy(R1.H,R2.H):rnd`	Round theresult.

Table 3-1 Instruction symbols (cont.)

Symbol	Example	Meaning
`:carry`	`R5:4=add(R1:0,R3:2,P1):carry`	Predicate used as carry input and output.
`:<<16`	`R2 = add(R1.L,R2.L):<<16`	Shift the result left by a halfword.
`:mem_noshuf`	`{memw(R5) = R2;` `R3 = memh(R6)}:mem_noshuf`	Inhibit load/store reordering (Section 5.5).

3.1.1 Numeric operands

Table 3-2 lists the notation that describes numeric operands in the syntax and behavior of instructions.

Table 3-2 Instruction operands

Symbol	Meaning	Min	Max	Example
`#uN`	Unsigned N-bit immediate value	0	2^N-1	`R2 = #u16`
`#sN`	Signed N-bit immediate value	-2^{N-1}	2^{N-1}-1	`R2 = add(R3,#s16)`
`#mN`	Signed N-bit immediate value	$-(2^{N-1}-1)$	2^{N-1}-1	`Rd = mpyi(Rs,#m9)`
`#uN:S`	Unsigned N-bit immediate value representing integral multiples of 2^S in specified range	0	$(2^N-1) \times 2^S$	`R2 = memh(#u16:1)`
`#sN:S`	Signed N-bit immediate value representing integral multiples of 2^S in specified range	$(-2^{N-1}) \times 2^S$	$(2^{N-1}-1) \times 2^S$	`Rd = memw(Rs++#s4:2)`
`#rN:S`	Same as #sN:S, but value is offset from PC of current packet	$(-2^{N-1}) \times 2^S$	$(2^{N-1}-1) \times 2^S$	`call #r22:2`
$usat_N$	Saturate value to unsigned N-bit number	0	2^N-1	$usat_{16}$`(Rs)`
sat_N	Saturate value to signed N-bit number	-2^{N-1}	2^{N-1}-1	sat_{16}`(Rs)`
`sxt x->y`	Sign-extend value from x to y bits	–	–	`sxt32->64(Rs)`
`zxt x->y`	Zero-extend value from x to y bits	–	–	`zxt32->64(Rs)`
`>>>`	Logical right shift	–	–	`Rss >>> offset`

The `#uN`, `#sN`, and `#mN` symbols specify immediate operands in instructions. The `#` symbol appears in the actual instruction to indicate the immediate operand.

The `#rN` symbol specifies loop and branch destinations in instructions. The `#` symbol does not appear in the instruction; instead, the entire `#rN` symbol (including the `:S` suffix) is expressed as a loop or branch symbol whose numeric value is determined by the assembler and linker.

For example:

```
call my_proc            // instruction example
```

The `:S` suffix indicates that the `S` least-significant bits in a value are implied zero bits and therefore not encoded in the instruction. The implied zero bits are called scale bits. For example, `#s4:2` denotes a signed immediate operand represented by four bits encoded in the instruction, and two scale bits. The possible values for this operand are -32, -28, -24, -20, -16, -12, -8, -4, 0, 4, 8, 12, 16, 20, 24, and 28.

The `##` symbol specifies a 32-bit immediate operand in an instruction (including a loop or branch destination). The `##` symbol indicates the operand in the actual instruction.

Examples of operand symbols:

```
Rd = add(Rs,#s16)      // #s16   -> signed 16-bit imm value
Rd = memw(Rs++#s4:2)   // #s4:2  -> scaled signed 4-bit imm value
call #r22:2            // #r22:2 -> scaled 22-bit PC-rel addr value

Rd = ##u32             // ##u32  -> unsigned 32-bit imm value
```

NOTE: When an instruction contains more than one immediate operand, the operand symbols are specified in upper and lowercase (for example, `#uN` and `#UN`) to indicate where they appear in the instruction encodings

3.1.2 Terminology

Table 3-3 lists the symbols Hexagon processor instruction names use to specify the supported data types.

Table 3-3 Data symbols

Size	Symbol	Type
8-bit	B	Byte
8-bit	UB	Unsigned byte
16-bit	H	Half word
16-bit	UH	Unsigned half word
32-bit	W	Word
32-bit	UW	Unsigned word
64-bit	D	Double word

3.1.3 Register operands

The following notation describes register operands in the syntax and behavior of instructions:

```
Rds[.elst]
```

The `ds` field indicates the register operand type and bit size (as defined in Table 3-4).

Table 3-4 Register symbols

Symbol	Operand type	Size (in bits)
d	Destination	32
dd		64
s	First source	32
ss		64
t	Second source	32
tt		64
u	Third source	32
uu		64
x	Source *and* destination	32
xx		64

Examples of the `ds` field that describe instruction syntax:

```
Rd = neg(Rs)            // Rd -> 32-bit dest, Rs 32-bit source
Rd = xor(Rs,Rt)         // Rt -> 32-bit second source
Rx = insert(Rs,Rtt)     // Rx -> both source and dest
```

Examples of the `ds` field that describe instruction behavior:

```
Rdd = Rss + Rtt         // Rdd, Rss, Rtt -> 64-bit registers
```

The optional `elst` (element size and type) field specifies parts of a register when the register is used as a vector. It can specify the following values:

- A signed or unsigned byte, halfword, or word within the register (as defined in Figure 3-1)
- A bit field within the register (as defined in Table 3-5)

Examples of the `elst` field:

```
EA = Rt.h[1]              // .h[1] -> bit field 31:16 in Rt
Pd = (Rss.u64 > Rtt.u64)  // .u64  -> unsigned 64-bit value
Rd = mpyu(Rs.L,Rt.H)      // .L/.H -> low/high 16-bit fields
```

NOTE: The control and predicate registers use the same notation as general registers, but are written as Cx and Px (respectively) instead of Rx.

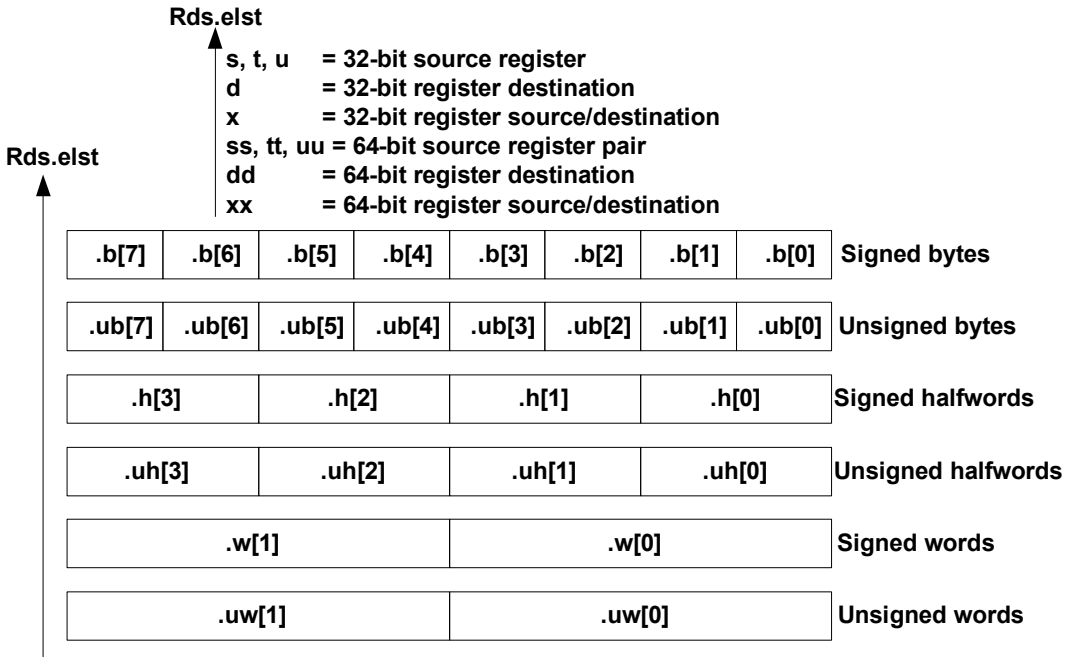


Figure 3-1 Register field symbols

Table 3-5 Register bit field symbols

Symbol	Meaning
.sN	Bits [N-1:0] are treated as a N-bit signed number. For example, R0.s16 means that the least significant 16-bits of R0 are treated as a 16-bit signed number.
.uN	Bits [N-1:0] are treated as a N-bit unsigned number.
.H	Most-significant 16 bits of a 32-bit register.
.L	Least-significant 16 bits of a 32-bit register.

3.2 Instruction classes

The Hexagon processor instructions are assigned to specific instruction classes. Classes determine the combinations of instructions that can be written in parallel (Section 3.3). Table 3-6 lists the instruction classes and how they can be grouped.

Instruction classes logically correspond with instruction types, so they serve as mnemonics for looking up specific instructions. For instance, the ALU32 class contains ALU instructions that operate on 32-bit operands.

Table 3-6 Instruction classes and subclasses

Class	Subclass	Description
ALU32	–	32-bit ALU operations
	ALU32 ALU	Arithmetic and logical
	ALU32 PERM	Permute
	ALU32 PRED	Predicate operations
CR	–	Control register access and loops
JR	–	Jumps (register indirect addressing mode)
J	–	Jumps (PC-relative addressing mode)
LD	–	Memory load operations
MEMOP	–	Memory operations
NV	–	New-value operations
	NV J	New-value jumps
	NV ST	New-value stores
ST	–	Memory store operations; allocate stack frame
System	–	Operating system access
	System user	Application-level access

Table 3-6 Instruction classes and subclasses

Class	Subclass	Description
XTYPE	–	Assorted operations
	XTYPE ALU	64-bit ALU operations
	XTYPE bit	Bit operations
	XTYPE complex	Complex math (using real and imaginary numbers)
	XTYPE FP	Floating point operations
	XTYPE MPY	Multiply operations
	XTYPE perm	Vector permute and format conversion (pack, splat, swizzle)
	XTYPE PRED	Predicate operations
	XTYPE SHIFT	Shift operations (with optional ALU operations)

3.3 Instruction packets

Instructions can be grouped into very long instruction word (VLIW) packets for parallel execution, with each packet contains one to four instructions. Packets of varying length can be freely mixed in a program.

Vector instructions operate on single instruction multiple data (SIMD) vectors.

Software must explicitly specify instruction packets. Instruction packets are expressed in assembly language by enclosing groups of instructions in curly braces.

For example, two instructions grouped in a packet:

```
{ R0 = R1; R2 = R3 }
```

Four instructions grouped in a packet:

```
{ R8 = memh(R3++#2);
  R12 = memw(R1++#4);
  R = mpy(R10,R6):<<1:sat;
  R7 = add(R9,#2);}
```

Packets have restrictions on instruction combinations. The instruction class of the instructions in a packet determines the primary restriction. Packet formation is subject to the following constraints:

- Resource constraints determine how many instructions of a specific type can appear in a packet. The Hexagon processor has a fixed number of execution units: each instruction executes on a particular type of unit, and each unit can process at most one instruction at a time. Thus, for example, because the Hexagon processor contains only two load units, an instruction packet with three load instructions is invalid.
- Grouping constraints are a small set of rules that apply above and beyond the resource constraints.
- Dependency constraints ensure that no write-after-write hazards exist in a packet.

- Ordering constraints dictate the ordering of instructions within a packet.
- Alignment constraints dictate the placement of packets in memory.

NOTE: The Hexagon processor executes individual instructions (that are not explicitly grouped in packets) as packets that contain a single instruction.

3.3.1 Packet execution semantics

Packets are defined to have parallel execution semantics. The execution behavior of a packet is defined as follows:

- First, instructions in the packet read their source registers in parallel.
- Next, instructions in the packet execute.
- Finally, instructions in the packet write their destination registers in parallel.

For example, consider the following packet:

```
{ R2 = R3; R3 = R2; }
```

In the first phase, registers R3 and R2 are read from the register file. After execution, R2 is written with the old value of R3 and R3 is written with the old value of R2. The result of this packet is the swap of the values of R2 and R3.

NOTE: Dual stores, Dual jumps, New-value stores, New-value compare jumps, and Dot-new predicates have non-parallel execution semantics.

3.3.2 Sequencing semantics

Packets of any length can freely mix in code. A packet is considered an atomic unit: in essence, a single large instruction. From the program perspective, a packet either executes to completion or not at all; it never partially executes. For example, if a packet causes a memory exception, the exception point is established before the packet.

A packet containing multiple load/store instructions can require service from the external system. For instance, consider a packet that performs two load operations that both miss in the cache. The packet requires the memory system to supply the data:

- From the memory system perspective, the two resulting load requests are processed serially.
- From the program perspective, however, both load operations must complete before the packet can complete.

Thus, the packet is atomic from the program perspective.

Packets have a single PC address, which is the address of the start of the packet. Branches cannot be performed into the middle of a packet.

Architecturally, packets execute to completion – including updating all registers and memory – before the next packet begins. As a result, application programs are not exposed to any pipeline artifacts.

3.3.3 Resource constraints

A packet cannot use more hardware resources than are physically available on the processor. For instance, because the Hexagon processor has only two load units, a packet with three load instructions is invalid. The behavior of such a packet is undefined. The assembler automatically rejects packets that oversubscribe the hardware resources.

The processor supports up to four parallel instructions. The instructions execute in four parallel pipelines, which are referred to as slots. The four slots are named Slot 0, Slot 1, Slot 2, and Slot 3.

NOTE: The `endloopN` instructions (Section 8.2.2) do not use any slots.

Each instruction belongs to specific Instruction classes. For example, jumps belong to instruction class J and loads belong to instruction class LD. The class of an instruction determines the slot in which the instruction can execute.

Figure 3-2 shows which instruction classes can be assigned to each of the four slots.

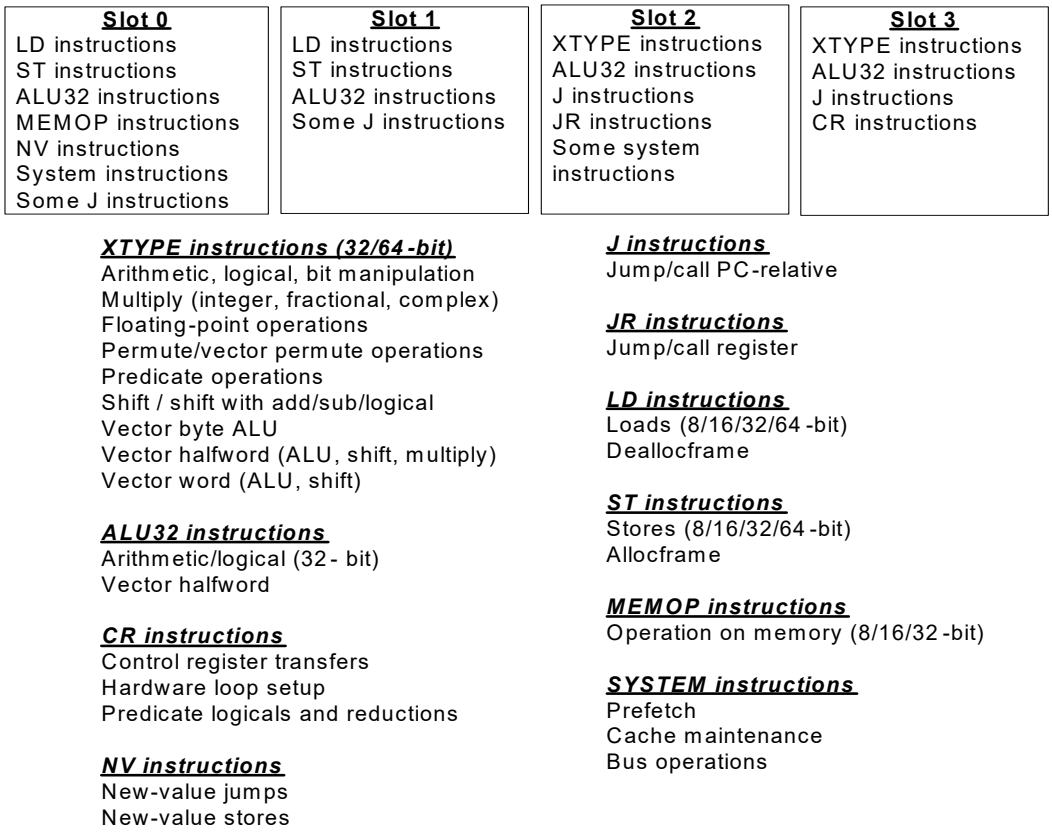


Figure 3-2 Packet grouping combinations

3.3.4 Grouping constraints

A few restrictions determine what constitutes a valid packet. The assembler ensures that packets follow valid grouping rules. If a packet executes that violates a grouping rule, the behavior is undefined. The following rules must be followed:

- Dot-new conditional instructions (Section 6.1.4) must be grouped in a packet with an instruction that generates dot-new predicates.
- ST-class instructions can be placed in Slot 1. In this case, Slot 0 normally must contain a second ST-class instruction (Section 5.4).
- J-class instructions can be placed in Slots 2 or 3. However, only certain combinations of program flow instructions (J or JR) can be grouped in a packet (Section 8.7). Otherwise, at most one program flow instruction is allowed in a packet. Some Jump and compare-Jump instructions can execute on slots 0 or 1, excluding calls, such as the following:
  - Instructions of the form "`Pd=cmp.xx(); if(Pd.new)jump:hint <target>`"
  - Instructions of the form "`If(Pd[.new]) jump[:hint] <target>`"
  - The "`jump<target>`" instruction
- JR-class instructions can be placed in Slot 2. However, when encoded in a duplex `jumpr` instruction, R31 can be placed in Slot 0 (Section 10.3).

- Restrictions limit the instructions that can appear in a packet at the setup or end of a hardware loop (Section 8.2.4).
- A user control register transfer to the control register `USR` cannot be grouped with a floating point instruction (Section 2.2.3).
- The SYSTEM-class instructions include prefetch, cache operations, bus operations, load locked, and store conditional instructions (Section 5.10). These instructions have the following grouping rules:
  - The brkpt, trap, pause, icinva, isync, and syncht instructions are solo instructions. They must not be grouped with other instructions in a packet.
  - The memw_locked, memd_locked, l2fetch, and trace instructions must execute on Slot 0. They must be grouped only with ALU32 or (non-FP) XTYPE instructions.
  - The dccleana, dcinva, dccleaninva, and dczeroa instructions must execute on Slot 0. Slot 1 must be empty or an ALU32 instruction.

3.3.5 Dependency constraints

Instructions in a packet cannot write to the same destination register. The assembler automatically flags such packets as invalid. If the processor executes a packet with two writes to the same general register, an error exception is raised.

If the processor executes a packet that performs multiple writes to the same predicate or control register, the behavior is undefined. Three special cases exist for this rule:

- Conditional writes are only allowed to target the same destination register when at most one of the writes is performed (Section 6.1.5).
- The overflow flag in the status register has defined behavior when multiple instructions write to it (Section 2.2.3). Do not group instructions that write to the entire user status register (for example, USR = R2) in a packet with any instruction that writes to a bit in the user status register.
- Multiple compare instructions are allowed to target the same predicate register to perform a logical AND of the results (Section 6.1.3).

3.3.6 Ordering constraints

In assembly code, instructions can appear in a packet in any order (except for Dual jumps). The assembler automatically encodes instructions in the packet in the proper order.

In the binary encoding of a packet, the instructions must be ordered from Slot 3 down to Slot 0. If the packet contains fewer than four instructions, any unused slot is skipped – a NOP is unnecessary as the hardware handles the proper spacing of the instructions.

In memory, instructions in a packet must appear in strictly decreasing slot order. Additionally, if an instruction can go in a higher-numbered slot, and that slot is empty, it must move into the higher-numbered slot.

For example, if a packet contains three instructions and slot 1 is not used, encode the instructions in the packet as follows:

- Slot 3 instruction at the lowest address
- Slot 2 instruction follows the Slot 3 instruction
- Slot 0 instructions at the last (highest) address

If a packet contains a single load or store instruction, that instruction must go in Slot 0, which is the highest address. As an example, a packet that contains both LD and ALU32 instructions must be ordered so the LD is in Slot 0 and the ALU32 in another slot.

3.3.7 Alignment constraints

Packets have the following constraints on their placement or alignment in memory:

- Packets must be word-aligned (32-bit). If the processor executes an improperly aligned packet, it raises an error exception (Section 8.10).
- Packets should not wrap the 4 GB address space. If address wraparound occurs, the processor behavior is undefined.

No other core-based restrictions exist for code placement or alignment.

If the processor branches to a packet that crosses a 16-byte address boundary, the resulting instruction fetch stalls for one cycle. Packets that are jump targets or loop body entries can be explicitly aligned to ensure that this stall does not occur (Section 8.3.5).

3.4 Instruction intrinsics

To support efficient coding of the time-critical sections of a program (without resorting to assembly language), the C compilers support intrinsics that directly express Hexagon processor instructions from within C code.

For example:

```
int main()
{
    long long v1 = 0xFFFF0000FFFF0000LL;
    long long v2 = 0x0000FFFF0000FFFFLL;
    long long result;

    // Find the minimum for each half-word in a 64-bit vector
    result = Q6_P_vminh_PP(v1,v2);
}
```

Intrinsics are defined for most of the Hexagon processor instructions.

3.5 Compound instructions

The Hexagon processor supports compound instructions, which encode pairs of common operations in a single instruction. For example, each of the following is a single compound instruction:

```
dealloc_return                  // Deallocate frame and return
R2 &= and(R1, R0)               // And and and
R7 = add(R4, sub(#15, R3))      // Subtract and add
R3 = sub(#20, asl(R3, #16))     // Shift and subtract
R5 = add(R2, mpyi(#8, R4))      // Multiply and add
{                               // Compare and jump
   P0 = cmp.eq (R2, R5)
   if (P0.new) jump:nt target
}
{                               // Register transfer and jump
      R2 = #15
      jump target
}
```

Compound instructions reduce code size and improve code performance.

NOTE: Compound instructions (except for X-and-jump as shown in Section 3.5) have distinct assembly syntax from the instructions they are composed of.

3.6 Duplex instructions

To reduce code size, the Hexagon processor supports duplex instructions, which encode pairs of common instructions in a 32-bit instruction container.

Unlike Compound instructions, duplex instructions do not have distinctive syntax – in assembly code they appear identical to the instructions they are composed of. The assembler is responsible for recognizing when a pair of instructions can be encoded as a single duplex rather than a pair of regular instruction words.

To fit two instructions into a single 32-bit word, Duplexes are limited to a subset of the most common instructions (load, store, branch, ALU), and the most common register operands.

4 Data processing

The Hexagon processor provides a rich set of operations for processing scalar and vector data. Instructions can perform a wide variety of operations on fixed-point or floating-point data. The fixed-point operations support scalar and vector data in a variety of sizes. The floating-point operations support single-precision data.

This chapter presents an overview of the operations provided by the following Hexagon processor instruction classes:

- XTYPE – General-purpose data operations
- ALU32 – Arithmetic/logical operations on 32-bit data

4.1 Data types

The Hexagon processor provides operations for processing the following data types.

4.1.1 Fixed-point data

The Hexagon processor provides operations to process 8-, 16-, 32-, or 64-bit fixed-point data. The data is either integer or fractional, and in signed or unsigned format.

4.1.1.1 Scalar operations

The Hexagon processor includes the following scalar operations on fixed-point data:

- Multiplication of 16-bit, 32-bit, and complex data
- Addition and subtraction of 16-, 32-, and 64-bit data (with and without saturation)
- Logical operations on 32-bit and 64-bit data (AND, OR, XOR, NOT)
- Shifts on 32-bit and 64-bit data (arithmetic and logical)
- Min/max, negation, absolute value, parity, norm, swizzle
- Compares of 8-, 16-, 32-, and 64-bit data
- Sign and zero extension (8- and 16-bit to 32-bit, 32-bit to 64-bit)
- Bit manipulation
- Predicate operations

4.1.1.2 Vector operations

The Hexagon processor includes the following vector operations on fixed-point data:

- Multiplication (halfwords, word by half, vector reduce, dual multiply)
- Addition and subtraction of word and halfword data
- Shifts on word and halfword data (arithmetic and logical)
- Min/max, average, negative average, absolute difference, absolute value
- Compares of word, halfword, and byte data
- Reduce, sum of absolute differences on unsigned bytes
- Special-purpose data arrangement (such as pack, splat, shuffle, align, saturate, splice, truncate, complex conjugate, complex rotate, zero extend)

NOTE: Certain vector operations support automatic scaling, saturation, and rounding.

For example, the following instruction performs a vector operation:

```
R1:0 += vrmpyh(R3:2,R5:4)
```

It is defined to perform the following operations in one cycle:

```
R1:0 += ((R2.L * R4.L) +
        (R2.H * R4.H) +
        (R3.L * R5.L) +
        (R3.H * R5.H)
          )
```

Figure 4-1 shows a schematic of this instruction type.

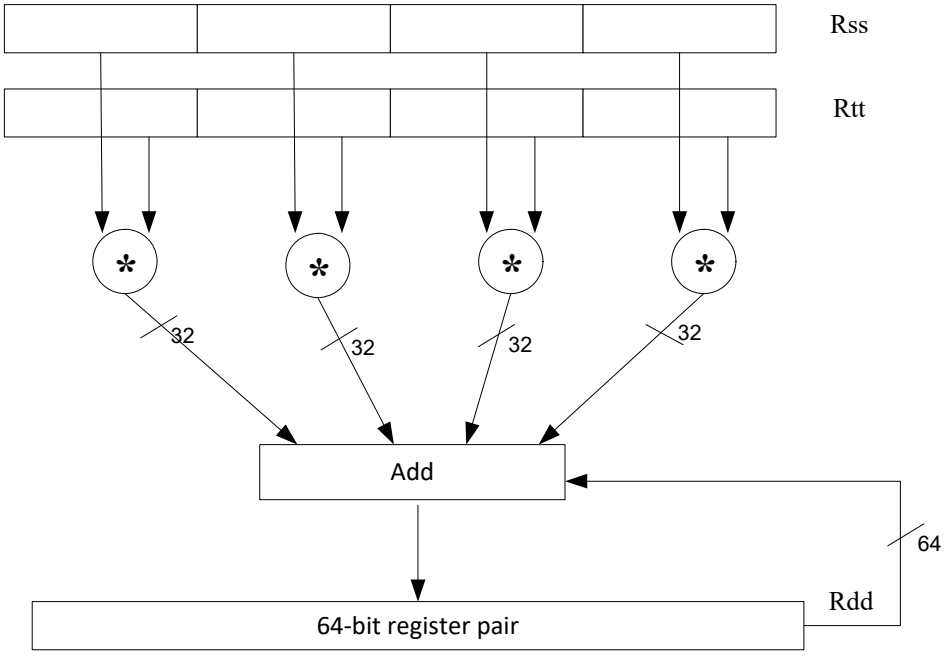


Figure 4-1 Vector instruction example

4.1.2 Floating-point data

The Hexagon processor provides operations to process 32-bit floating-point numbers. The numbers are stored in IEEE single-precision floating-point format.

Per the IEEE standard, certain floating-point values are defined to represent positive or negative infinity, as well as Not-a-Number (NaN), which represents values that have no mathematical meaning.

Floating-point numbers can be held in a general register.

4.1.2.1 Floating-point operations

The Hexagon processor includes the following operations on floating-point data:

- Addition and subtraction
- Multiplication (with optional scaling)
- Min/max/compare
- Reciprocal/square root approximation
- Format conversion

4.1.3 Complex data

The Hexagon processor provides operations to process 32-bit or 64-bit complex data.

Complex numbers include a signed real portion and a signed imaginary portion. Given two complex numbers (a + bi) and (c + di), the complex multiply operations compute both the real portion (ac - bd) and the imaginary portion (ad + bc) in a single instruction.

Complex numbers can be packed in a general register or register pair. When packed, the imaginary portion occupies the most-significant portion of the register or register pair.

4.1.4 Vector data

The Hexagon processor provides operations to process 64-bit vector data.

Vector data types pack multiple data items – bytes, halfwords, or words – into 64-bit registers. Vector data operations are common in video and image processing.

Eight 8-bit bytes can be packed into a 64-bit register.

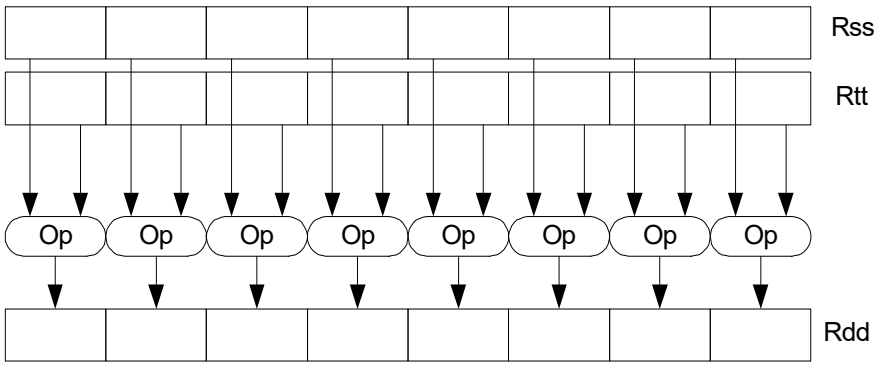


Figure 4-1 Vector byte operation example

Four 16-bit halfword values can be packed in a single 64-bit register pair.

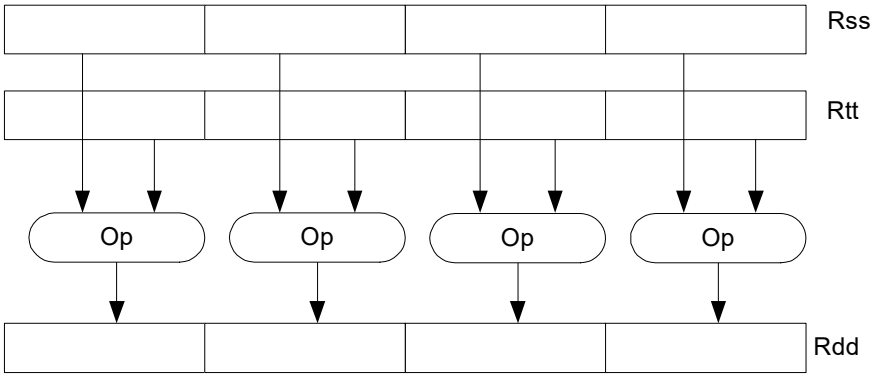


Figure 4-2 Vector halfword operation example

Two 32-bit word values can be packed in a single 64-bit register pair.

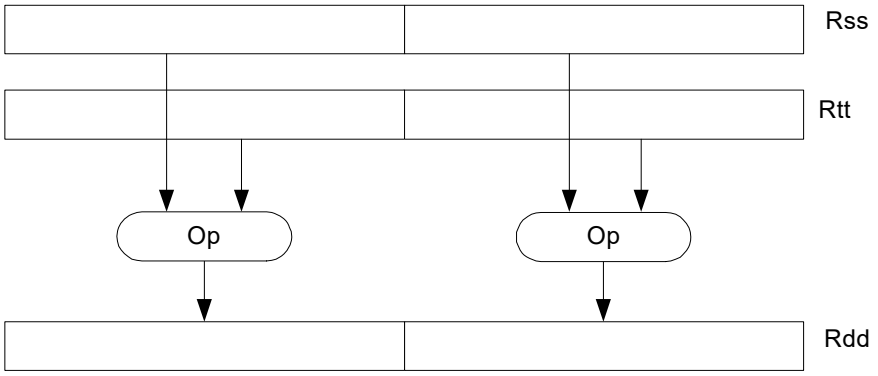


Figure 4-3 Vector word operation example

4.2 Instruction options

Some instructions support optional scaling, saturation, and rounding. There are no mode bits controlling these options – instead, they are explicitly specified as part of the instruction name. The options are described in this section.

4.2.1 Fractional scaling

In fractional data format, data is treated as fixed-point fractional values whose range is determined by the word length and radix point position.

Fractional scaling is specified in an instruction by adding the `:<<1` specifier. For example:

```
R3:2 = cmpy(R0,R1):<<1:sat
```

When two fractional numbers are multiplied, the product must be scaled to restore the original fractional data format. The Hexagon processor allows specification of the fractional scaling of the product in the instruction for shifts of 0 and 1. Perform a shift of 1 for Q1.15 numbers. Perform a shift of 0 for integer multiplication.

4.2.2 Saturation

Certain instructions are available in saturating form. If a saturating arithmetic instruction has a result that is smaller than the minimum value, the result is set to the minimum value. Similarly, if the operation has a result that is greater than the maximum value, the result is set to the maximum value.

Saturation is specified in an instruction by adding the `:sat` specifier. For example:

```
R2 = abs(R1):sat
```

The open virtualization format (OVF) bit in the User status register is set whenever a saturating operation saturates to the maximum or minimum value. It remains set until explicitly cleared by a control register transfer to USR. For vector-type saturating operations, if any of the individual elements of the vector saturate, OVF is set.

4.2.3 Arithmetic rounding

Certain signed multiply instructions support optional arithmetic rounding (also known as biased rounding). The arithmetic rounding operation takes a double precision fractional value and adds 0x8000 to the low 16-bits (least significant 16-bit halfword).

Rounding is specified in an instruction by adding the `:rnd` specifier. For example:

```
R2 = mpy(R1.h,R2.h):rnd
```

NOTE: Arithmetic rounding can accumulate numerical errors, especially when the number to round is exactly 0.5. This occurs most frequently when dividing by two or averaging.

4.2.4 Convergent rounding

To address the problem of error accumulation in Convergent rounding, the Hexagon processor includes four instructions that support positive and negative averaging with a convergent rounding option.

These instructions work as follows:

1. Compute (A + B) or (A - B) for AVG and NAVG respectively.
2. Based on the two least-significant bits of the result, add a rounding constant as follows:
   - If the two LSBs are 00, add 0
   - If the two LSBs are 01, add 0
   - If the two LSBs are 10, add 0
   - If the two LSBs are 11, add 1
3. Shift the result right by one bit.

4.2.5 Scaling for divide and square-root

On the Hexagon processor, floating point divide and square-root operations are implemented in software using library functions. To enable the efficient implementation of these operations, the processor supports special variants of the multiply-accumulate instruction, named scale fused multiply add (FMA).

Scale FMA supports optional scaling of the product generated by the floating-point fused multiply-add instruction.

Scaling is specified in the instruction by adding the `:scale` specifier and a predicate register operand. For example:

```
R3 += sfmpy(R0,R1,P2):scale
```

For single precision, the scaling factor is two raised to the power specified by the contents of the predicate register (which is treated as an 8-bit two's complement value). For double precision, the predicate register value is doubled before use as a power of two.

NOTE: Do not use scale FMA instructions outside of divide and square-root library routines. No guarantee is provided that future versions of the Hexagon processor implements these instructions using the same semantics. Future versions assume only that compatibility for scale FMA is limited to the needs of divide and square-root library routines.

4.3 XTYPE operations

The XTYPE instruction class includes most of the data-processing operations performed by the Hexagon processor. These operations are categorized by their operation type.

4.3.1 ALU

XTYPE ALU operations modify 8-, 16-, 32-, and 64-bit data. These operations include:

- Add and subtract with and without saturation
- Add and subtract with accumulate
- Absolute value
- Logical operations
- Min, max, negate instructions
- Register transfers of 64-bit data
- Word to doubleword sign extension
- Comparisons

4.3.2 Bit manipulation

XTYPE bit manipulation operations modify bit fields in a register or register pair. These operations include:

- Bit field insert
- Bit field signed and unsigned extract
- Count leading and trailing bits
- Compare bitmasks
- Set/clear/toggle bit
- Test bit operation
- Interleave/deinterleave bits
- Bit reverse
- Split bit field
- Masked parity and linear feedback shift
- Table index formation

4.3.3 Complex

XTYPE complex operations manipulate complex numbers. These operations include:

- Complex add and subtract
- Complex multiply with optional round and pack
- Vector complex multiply
- Vector complex conjugate
- Vector complex rotate
- Vector reduce complex multiply real or imaginary

4.3.4 Floating point

XTYPE FP operations manipulate single-precision floating point numbers. These operations include:

- Addition and subtraction
- Multiplication (with optional scaling)
- Min/max/compare
- Format conversion

The Hexagon floating-point operations are defined to support the IEEE floating-point standard. However, certain IEEE-required operations – such as divide and square root – are not supported directly. Instead, special instructions are defined to support the implementation of the required operations as library routines. These instructions include:

- A special version of the fused multiply-add instruction (designed specifically for use in library routines)
- Reciprocal/square root approximations (which compute the approximate initial values used in reciprocal and reciprocal-square-root routines)
- Extreme value assistance (which adjusts input values if they cannot produce correct results using convergence algorithms)

NOTE: The special floating-point instructions are not intended for use directly in user code – use the special floating-point instructions only in the floating point library.

Format conversion

The floating-point conversion instructions `sfmake` and `dfmake` convert an unsigned 10-bit immediate value into the corresponding floating-point value.

The immediate value must be encoded so that bits [5:0] contain the significand, and bits [9:6] the exponent. The exponent value is added to the initial exponent value (`bias - 6`).

For example, to generate the single-precision floating point value 2.0, bits [5:0] must be set to 0, and bits [9:6] set to 7. Performing the sfmake operation on this immediate value yields the floating point value `0x40000000`, which is 2.0.

NOTE: The conversion instructions are designed to handle common floating point values, including most integers and many basic fractions (1/2, 3/4, and so on).

Rounding

The Hexagon User status register includes the FPRND field, which specifies the IEEE-defined floating-point rounding mode.

Exceptions

The Hexagon user status register includes five status fields, which work as sticky flags for the five IEEE-defined exception conditions: inexact, overflow, underflow, divide by zero, and invalid. A sticky flag is set when the corresponding exception occurs, and remains set until explicitly cleared.

The user status register also includes five mode fields that specify whether to perform an operating-system trap when one of the floating-point exceptions occur. For every instruction packet containing a floating-point operation, if a floating-point sticky flag and the corresponding trap-enable bit are both set, a floating-point trap is generated. After the packet commits, the Hexagon processor then automatically traps to the operating system.

NOTE: Non-floating-point instructions never generate a floating-point trap, regardless of the state of the sticky flag and trap-enable bits.

4.3.5 Multiply

Multiply operations support fixed-point multiplication, including both single- and double-precision multiplication, and polynomial multiplication.

Single precision

In single-precision arithmetic, a 16-bit value is multiplied by another 16-bit value. These operands can come from the high portion or low portion of any register. Depending on the instruction, the result of the 16×16 operation can optionally be accumulated, saturated, rounded, or shifted left by 0 bits to 1 bits.

The instruction set supports operations on signed $\times$ signed, unsigned $\times$ unsigned, and signed $\times$ unsigned data.

Table 4-1 summarizes the options available for 16 × 16 single precision multiplications. The symbols used in the table are as follows:

- SS: Perform signed × signed multiply
- UU: Perform unsigned × unsigned multiply
- SU: Perform signed × unsigned multiply
- A+: Result added to accumulator
- A-: Result subtracted from accumulator
- 0: Result not added to accumulator

Table 4-1 Single-precision multiply options

Multiply	Result	Sign	Accumulate	Sat	Rnd	Scale
16 × 16	32	SS	A+, A-	Yes	No	0-1
16 × 16	32	SS	0	Yes	Yes	0-1
16 × 16	64	SS	A+, A-	No	No	0-1
16 × 16	64	SS	0	No	Yes	0-1
16 × 16	32	UU	A+, A-, 0	No	No	0-1
16 × 16	64	UU	A+, A-, 0	No	No	0-1
16 × 16	32	SU	A+, 0	Yes	No	0-1

Double precision

Double precision instructions are available for both 32 × 32 and 32 × 16 multiplication:

- For 32 × 32 multiplication the result is either 64 bits or 32 bits. The 32-bit result is either the high or low portion of the 64-bit product.
- For 32 × 16 multiplication the result is always taken as the upper 32 bits.

The operands are either signed or unsigned.

Table 4-2 Double precision multiply options

Multiply	Result	Sign	Accumulate	Sat	Rnd	Scale
32 × 32	64	SS, UU	A+, A-, 0	No	No	0
32 × 32	32 (upper)	SS, UU	0	No	Yes	0
32 × 32	32 (low)	SS, UU	A+, 0	No	No	0
32 × 16	32 (upper)	SS, UU	A+, 0	Yes	Yes	0-1
32 × 32	32 (upper)	SU	0	No	No	0

Polynomial

Polynomial XTYPE MPY instructions are available for both words and vector halfwords.

These instructions are useful for many algorithms including scramble code generation, cryptographic algorithms, convolutional, and Reed Solomon code.

4.3.6 Permute

XTYPE perm operations perform operations on vector data that includes arithmetic, format conversion, and rearrangement of vector elements. Many types of conversions are supported:

- Swizzle bytes
- Vector shuffle
- Vector align
- Vector saturate and pack
- Vector splat bytes
- Vector splice
- Vector sign extend halfwords
- Vector zero extend bytes
- Vector zero extend halfwords
- Scalar saturate to byte, halfword, word
- Vector pack high and low halfwords
- Vector round and pack
- Vector splat halfwords

4.3.7 Predicate

XTYPE PRED operations modify predicate source data. Available categories of instructions include:

- Vector mask generation
- Predicate transfers
- Viterbi packing

4.3.8 Shift

Scalar XTYPE SHIFT operations perform 32 and 64-bit shifts followed by an optional addition/subtraction or logical operation. Figure 4-4 shows the general operation.

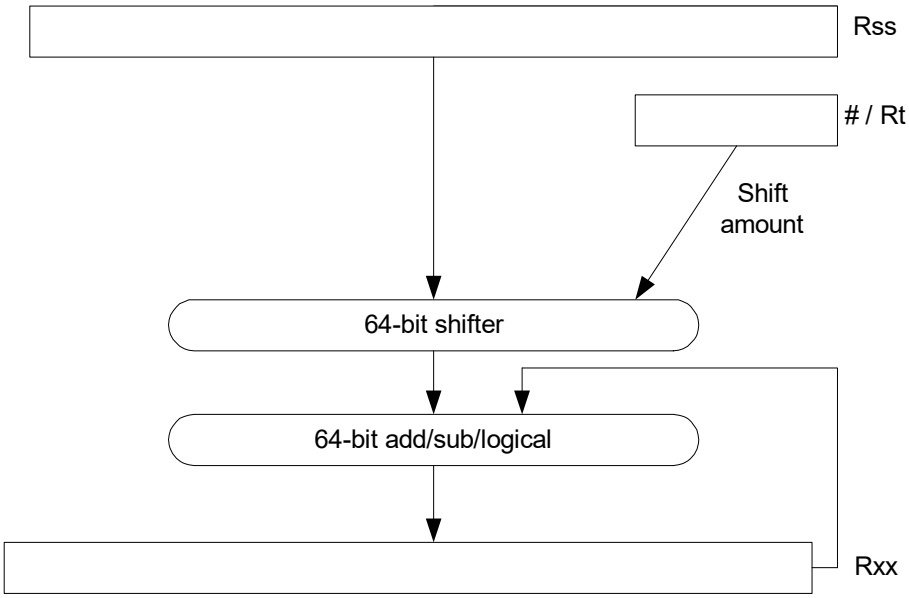


Figure 4-4 64-bit shift and add/sub/logical

Four shift types are supported:

- ASR: Arithmetic shift right
- ASL: Arithmetic shift left
- LSR: Logical shift right
- LSL: Logical shift left

In register-based shifts, the Rt register is a signed two's complement number. If this value is positive, the instruction opcode tells the direction of shift (right or left). If this value is negative, the shift direction indicated by the opcode is reversed.

When arithmetic right shifts are performed, the sign bit is shifted in, whereas logical right shifts shift in zeros. Left shifts always shift in zeros.

Some shifts are available with saturation and rounding options.

4.4 ALU32 operations

The ALU32 instruction class includes general arithmetic/logical operations on 32-bit data:

- Add, subtract, negate without saturation on 32-bit data
- Logical operations such as AND, OR, XOR, AND with immediate, and OR with immediate
- Scalar 32-bit compares
- Combine halfwords, combine words, combine with immediates, shift halfwords, and Mux
- Conditional add, combine, logical, subtract, and transfer.
- NOP
- Sign and zero-extend bytes and halfwords
- Transfer immediates and registers
- Vector add, subtract, and average halfwords

NOTE: ALU32 instructions can execute on any slot (Section 3.3.3).

Chapter 6 describes the conditional execution and compare instructions.

4.5 Vector operations

Vector operations support arithmetic operations on vectors of bytes, halfwords, and words.

The vector operations belong to the XTYPE instruction class (except for vector add, subtract, and average halfwords, which are ALU32).

Vector byte operations

The vector byte operations process packed vectors of signed or unsigned bytes. They include the following operations:

- Vector add and subtract signed or unsigned bytes
- Vector min and max signed or unsigned bytes
- Vector compare signed or unsigned bytes
- Vector average unsigned bytes
- Vector reduce add unsigned bytes
- Vector sum of absolute differences unsigned bytes

Vector halfword operations

The vector halfword operations process packed 16-bit halfwords. They include the following operations:

- Vector add and subtract halfwords
- Vector average halfwords
- Vector compare halfwords
- Vector min and max halfwords
- Vector shift halfwords
- Vector dual multiply
- Vector dual multiply with round and pack
- Vector multiply even halfwords with an optional round and pack
- Vector multiply halfwords
- Vector reduce multiply halfwords

For example, Figure 4-5 shows the operation of the vector arithmetic shift right halfword (vasrh) instruction. In this instruction, each 16-bit half-word is shifted right by the same amount that is specified in a register or with an immediate value. Because the shift is arithmetic, the bits shifted in are copies of the sign bit.

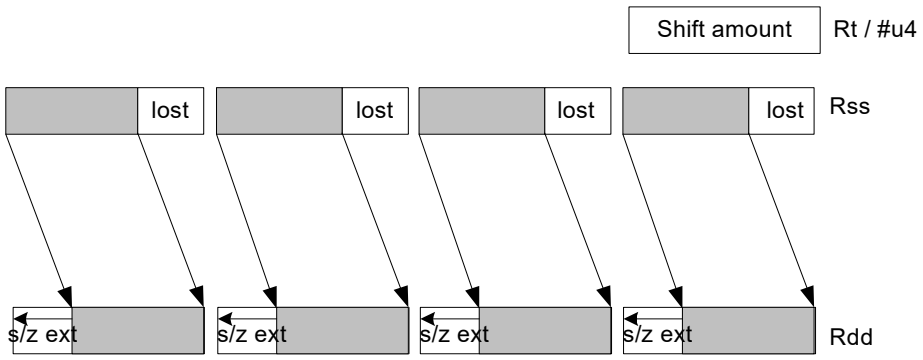


Figure 4-5 Vector halfword shift right

Vector word operations

The vector word operations process packed vectors of two words. They include the following operations:

- Vector add and subtract words
- Vector average words
- Vector compare words
- Vector min and max words
- Vector shift words with optional truncate and pack

For more information on vector operations, see Section 11.1.1 and Section 11.10.1.

4.6 CR operations

The CR instruction class includes operations that access the Control registers.

Table 4-3 Control register transfer instructions

Syntax	Operation
Rd = Cs Cd = Rs	Move control register to / from a general register. NOTE: PC is not a valid destination register.
Rdd = Css Cdd = Rss	Move control register pair to / from a general register pair. NOTE: PC is not a valid destination register.

NOTE: In register-pair transfers, control registers must be specified using their numeric alias names – see Section 2.2 for details.

4.7 Compound operations

The instruction set includes a number of instructions that perform multiple logical or arithmetic operations in a single instruction. They include the following operations:

- AND/OR with inverted input
- Compound logical register
- Compound logical predicate
- Compound add-subtract with immediates
- Compound shift-operation with immediates (arithmetic or logical)
- Multiply-add with immediates

For more information, see Section 11.10.1.

4.8 Special operations

The instruction set includes a number of special-purpose instructions to support specific applications.

4.8.1 H.264 CABAC processing

H.264/AVC is adopted in a diverse range of multimedia applications:

- HD-DVDs
- HDTV broadcasting
- Internet video streaming

Context Adaptive Binary Arithmetic Coding (CABAC) is one of the two alternative entropy coding methods specified in the H.264 main profile. CABAC offers superior coding efficiency at the expense of greater computational complexity. The Hexagon processor includes a dedicated instruction (decbin) to support CABAC decoding.

Binary arithmetic coding is based on the principle of recursive interval subdivision, and two quantities characterize its state:

- The current interval range
- The current offset in the current code interval

The offset is read from the encoded bit stream. When decoding a bin, the interval range is subdivided in two intervals based on the estimation of the probability $_{pLPS}$ of the least probable symbol (LPS): one interval with width of rLPS = range x pLPS, and another with width of rMPS = range x pMPS = range -rLPS, where MPS stands for most probable symbol.

Depending on which subinterval the offset falls into, the decoder decides whether the bin is decoded as MPS or LPS, after which the two quantities are iteratively updated, as shown in Figure 4-1.

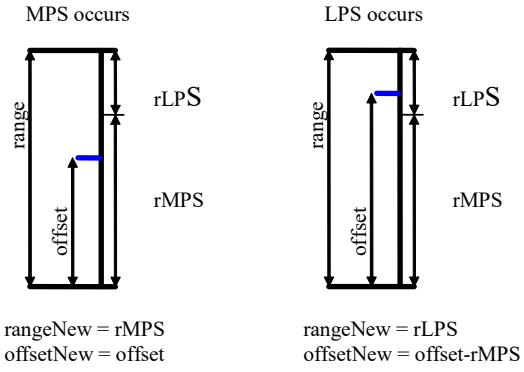


Figure 4-1 Arithmetic decoding for one bin

4.8.1.1 CABAC implementation

In H.264 video coding, range is a 9-bit quantity, and offset is 9 bits in regular mode and 10 bits in bypass mode during the whole decoding process. The calculation of rLPS is approximated by a 64 × 4 table of 256 bytes, where the range and the context state (selected for the bin to decode) address the lookup table. To maintain the precision of the whole decoding process, the new range must be renormalized to ensure that the most significant bit is always 1, and that the offset is synchronously refilled from the bit stream.

To simplify the renormalization/refilling process, the decoding scheme shown in Figure 4-2 significantly reduces the frequency of renormalization and refilling bits from the bit stream, while also being suitable for DSP implementation.

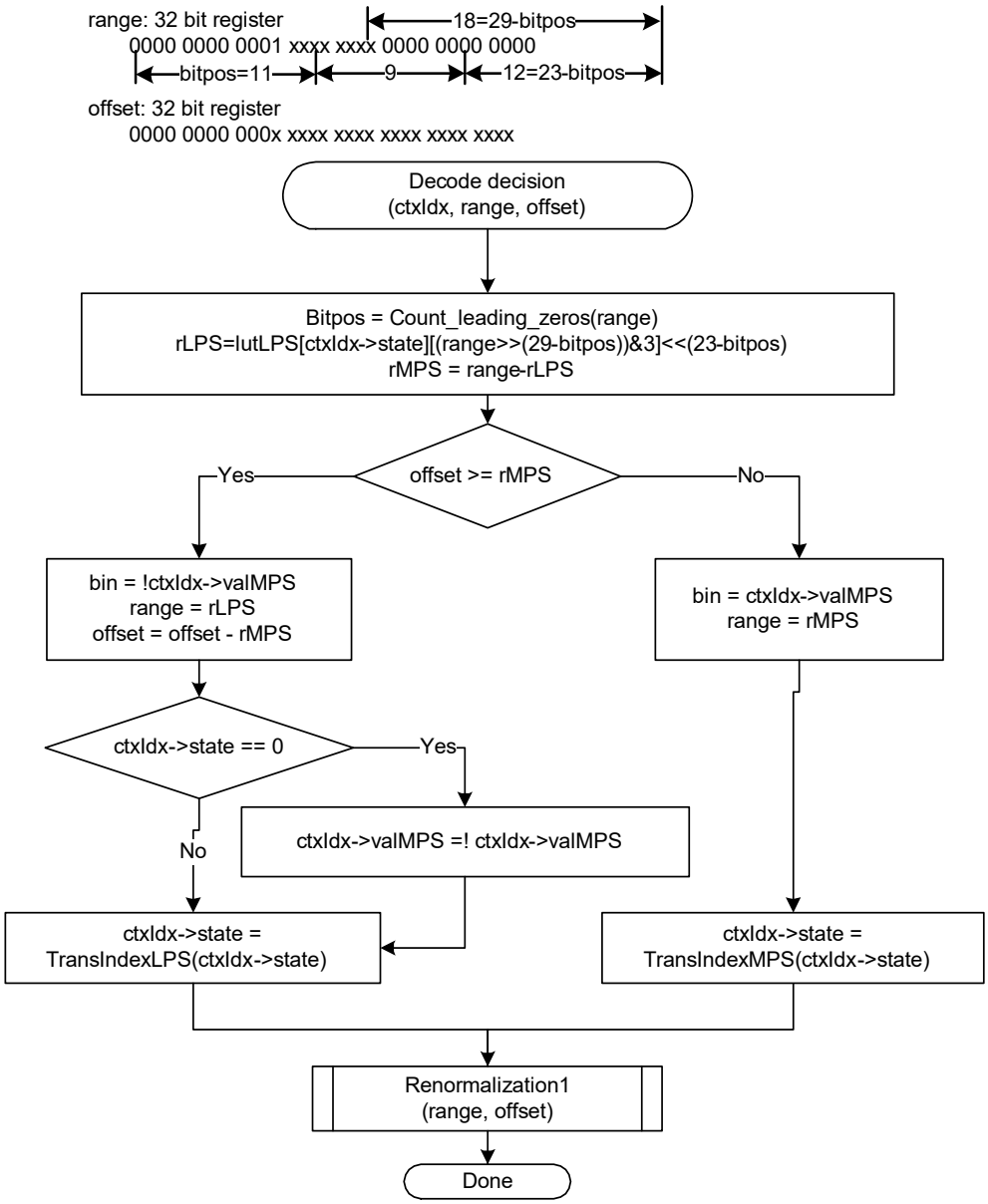


Figure 4-2 CABAC decoding engine for regular bin

The Hexagon processor can use the CABAC decode bin instruction to decode one regular bin in two cycles (not counting the bin refilling process).

For example:

```
Rdd = decbin(Rss,Rtt)

INPUT: Rss and Rtt register pairs as:
Rtt.w1[5:0] = state
Rtt.w1[8] = valMPS
Rtt.w0[4:0] =  bitpos
Rss.w0 = range
Rss.w1 = offset

OUTPUT: Rdd register pair is packed as
Rdd.w0[5:0] = state
Rdd.w0[8] = valMPS
Rdd.w0[31:23] = range
Rdd.w0[22:16] = '0'
Rdd.w1 = offset (normalized)

OUTPUT: P0
P0 = (bin)
```

4.8.1.2 Code example

```
H264CabacGetBinNC:
/*************************************************************
* Non-conventional call:
* Input: R1:0 = offset : range,  R2 = dep,   R3 = ctxIdx,
*      R4 = (*ctxIdx), R5 = bitpos
*
* Return:
*       R1: 0 - offset : range
*       P0 - (bin)
*************************************************************/

// Cycle #1
{ R1:0= decbin(R1:0,R5:4)   // Decode one bin
  R6 = asl(R22,R5)          // Where R22 = 0x100
}

// Cycle #2
{ memb(R3) = R0             // Save context to *ctxIdx
   R1:0 = vlsrw(R1:0,R5)    // Re-align range and offset
   P1 = cmp.gtu(R6,R1)      // Need refill? P1=(range < 0x100)
   IF (!P1.new) jumpr:t LR  // Return
}
RENORM_REFILL:
...
```

4.8.2 IP Internet checksum

Key features of the Internet checksum[1] include:

- The checksum can be summed in any order
- Carries can be accumulated using an accumulator larger than size being added, and added back in at any time

Using standard data-processing instructions, the Internet checksum can be computed at eight bytes per cycle in the main loop, by loading words and accumulating into doublewords. After the loop, the upper word is added to the lower word; the upper halfword is added to the lower halfword, and any carries are added back in.

The Hexagon processor supports a dedicated Vector reduce add halfwords (vradduh) instruction that computes the Internet checksum at a rate of 16 bytes per cycle.

The vradduh instruction accepts the halfwords of the two input vectors, adds them together, and places the result in a 32-bit destination register. This operation can both compute the sum of 16 bytes of input while preserving the carries, and accumulate carries at the end of computation.

NOTE: This operation uses the maximum load bandwidth available in the Hexagon processor.

[1] See RFC 1071 (http://www.faqs.org/rfcs/rfc1071.html)

4.8.2.1 Code example

```
    .text
    .global fast_ip_check
    // Assumes that data is 8-byte aligned
    // Assumes thatdata is padded at least 16 bytes after words with 0's.
    // Input R0 points to data
    // Input R1 is thelength of data
    // Returns IP checksum in R0

fast_ip_check:
    {
       R1 = lsr(R1,#4)          // 16-byte chunks, rounded down, +1
       R9:8 = combine(#0,#0)
       R3:2 = combine(#0,#0)
    }
    {
       loop0(1f,R1)
       R7:6 = memd(R0+#8)
       R5:4 = memd(R0++#16)
    }
    .falign
1:
    {
       R7:6 = memd(R0+#8)
       R5:4 = memd(R0++#16)
       R2 = vradduh(R5:4,R7:6)    // Accumulate 8 halfwords
       R8 = vradduh(R3:2,R9:8)    // Accumulate carries
    }:endloop0
    // Drain pipeline
    {
       R2 = vradduh(R5:4,R7:6)
       R8 = vradduh(R3:2,R9:8)
       R5:4 = combine(#0,#0)
    }
    {
       R8 = vradduh(R3:2,R9:8)
       R1 = #0
    }
    // Might have some carries to add back in
    {
       R0 = vradduh(R5:4,R9:8)
    }
    // Possible for one more to pop out
    {
       R0 = vradduh(R5:4,R1:0)
    }
    {
       R0 = not(R0)
       jumpr LR
    }
```

4.8.3 Software-defined radio

The Hexagon processor includes six special-purpose instructions that support the implementation of software-defined radio. The instructions greatly accelerate the following algorithms.

4.8.3.1 Rake despreading

A fundamental operation in despreading is the PN multiply operation. In this operation, the received complex chips are compared against a pseudorandom sequence of Quadrature Amplitude Modulation (QAM) constellation points and accumulated.

Figure 4-3 shows the `vrcrotate` instruction that performs this operation. The products are summed to form a soft 32-bit complex symbol. The instruction has both accumulating and non-accumulating versions.

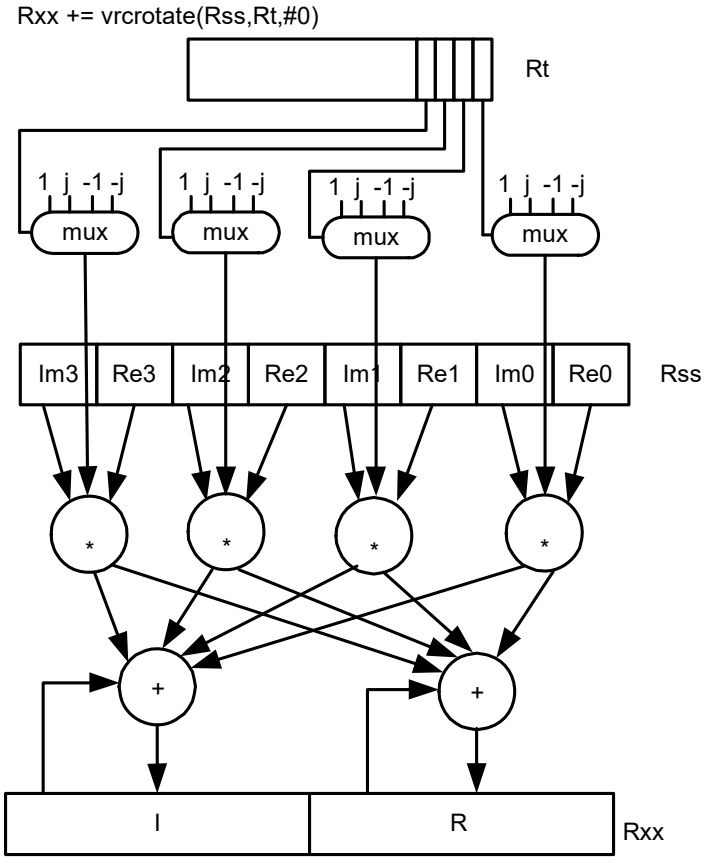


Figure 4-3 Vector reduce complex rotate

For more information on the `vrcrotate` instruction, see Vector reduce complex rotate.

NOTE: Using this instruction, the Hexagon processor can process 5.3 chips per cycle, and a 12-finger WCDMA user requires only 15 MHz.

4.8.3.2 Polynomial operations

The polynomial multiply instructions support the following operations:

- Scramble code generation (at a rate of eight symbols per cycle for WCDMA)
- Cryptographic algorithms (such as elliptic curve)
- CRCs (at a rate of 21 bits per cycle)
- Convolutional encoding
- Reed-Solomon codes

The four versions of this instruction support 32 × 32 and vector 16 × 16 multiplication both with and without accumulation, as shown in Figure 4-4.

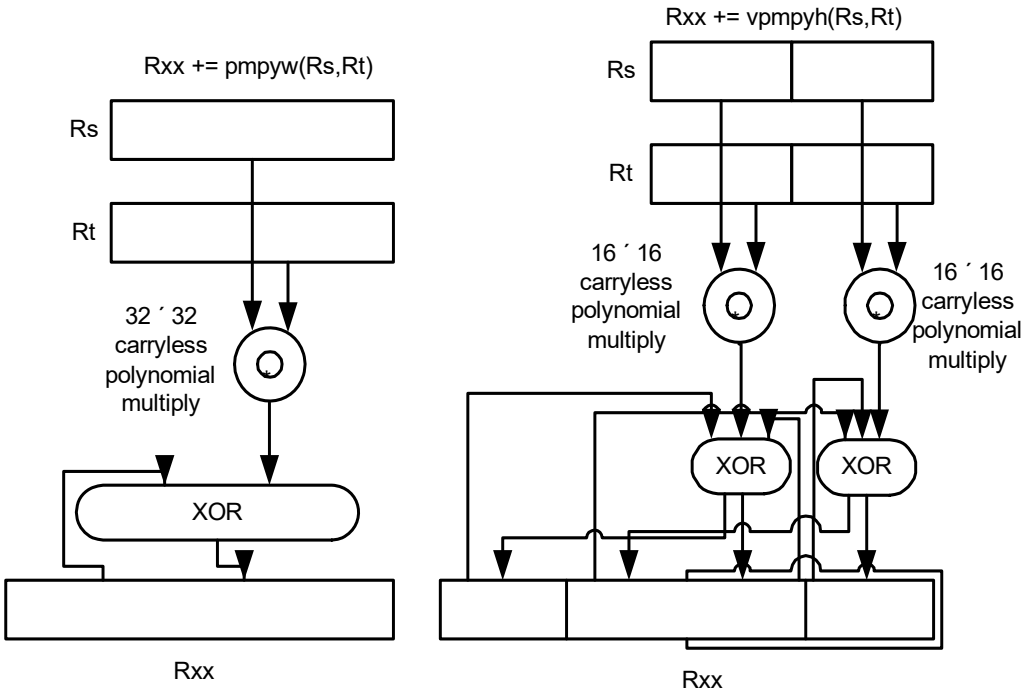


Figure 4-4 Polynomial multiply

For more information on the `pmpy` instructions, see Polynomial multiply words.

5 Memory

The Hexagon processor features a load/store architecture, where numeric and logical instructions operate on registers. Explicit load instructions move operands from memory to registers, while store instructions move operands from registers to memory. A few instructions (mem-ops) perform numeric and logical operations directly on memory.

The address space is unified: all accesses target the same linear address space, which contains both instructions and data.

5.1 Hexagon processor memory model

5.1.1 Address space

The Hexagon processor has a 32-bit byte-addressable memory address space. The entire 4G linear address space is addressable by the user application. A virtual-to-physical address translation mechanism is provided.

5.1.2 Byte order

The Hexagon processor is a little endian machine: the lowest address byte in memory is held in the least significant byte of a register, as shown in Figure 5-1.

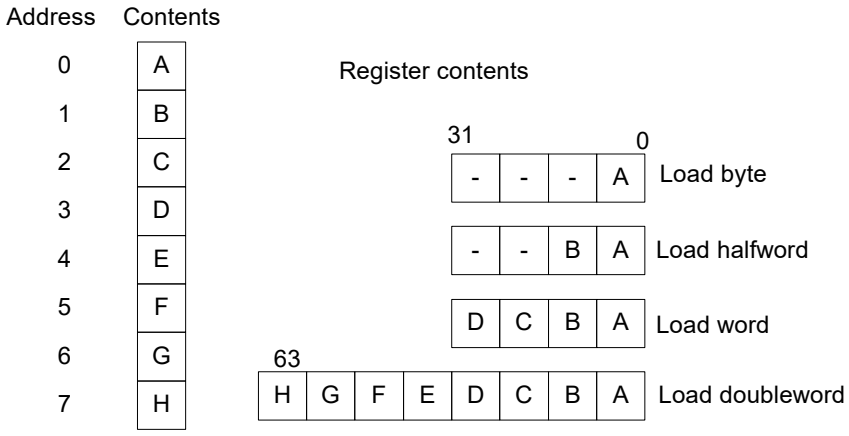


Figure 5-1 Hexagon processor byte order

5.1.3 Alignment

Even though the Hexagon processor memory is byte-addressable, instructions and data must be aligned in memory on specific address boundaries:

- Instructions and instruction packets must be 32-bit aligned
- Data must be aligned to its native access size.

Any unaligned memory access causes a memory-alignment exception.

Use the Permute instructions in applications that must reference unaligned vector data. The loads and stores still must be memory-aligned; however, the permute instructions enable easy rearrangement of the data in registers.

Table 5-1 Memory alignment restrictions

Data type	Size (bits)	Exception when
Byte Unsigned byte	8	Never
Halfword Unsigned halfword	16	LSB[0] != 0 [1]
Word Unsigned word	32	LSB[1:0] != 00
Doubleword	64	LSB[2:0] != 000
Instruction Instruction packet	32	LSB[1:0] != 00

[1] LSB = Least significant bits of address

5.2 Memory loads

Memory can be loaded in byte, halfword, word, or doubleword sizes. The data types supported are signed or unsigned. The syntax used is `memXX`, where `XX` denotes the data type.

Table 5-2 Load instructions

Syntax	Source size (bits)	Destination size (bits)	Data placement	Comment
`Rd = memub(Rs)`	8	32	Low 8 bits	Zero-extend 8 to 32 bits
`Rd = memb(Rs)`	8	32	Low 8 bits	Sign-extend 8 to 32 bits
`Rd = memuh(Rs)`	16	32	Low 16 bits	Zero-extend 16 to 32 bits
`Rd = memh(Rs)`	16	32	Low 16 bits	Sign-extend 16 to 32 bits
`Rd = memubh(Rs)`	16	32	Bytes 0 and 2	Bytes 1 and 3 zeroed [1]
`Rd = membh(Rs)`	16	32	Bytes 0 and 2	Bytes 1 and 3 sign-extended
`Rd = memw(Rs)`	32	32	All 32 bits	Load word
`Rdd = memubh(Rs)`	32	64	Bytes 0,2,4,6	Bytes 1, 3, 5, 7 zeroed
`Rdd = membh(Rs)`	32	64	Bytes 0,2,4,6	Bytes 1, 3, 5, 7 sign-extended
`Rdd = memd(Rs)`	64	64	All 64 bits	Load doubleword
`Ryy = memh_fifo(Rs)`	16	64	High 16 bits	Shift vector and load halfword
`deallocframe`	64	64	All 64 bits	See Chapter 7
`dealloc_return`	64	64	All 64 bits	See Chapter 7

[1] The memubh and membh instructions load contiguous bytes from memory (either 2 or 4 bytes) and unpack these bytes into a vector of halfwords. The instructions are useful when bytes are used as input into halfword vector operations, which is common in video and image processing..

NOTE: The memory load instructions belong to instruction class LD, and can execute only in slots 0 or 1.

5.3 Memory stores

Memory can be stored in byte, halfword, word, or doubleword sizes. The syntax used is `memX`, where `X` denotes the data type.

Table 5-3 Store instructions

Syntax	Source size (bits)	Destination size (bits)	Comment
`memb(Rs) = Rt`	32	8	Store byte (bits 7:0)
`memb(Rs) = #s8`	8	8	Store byte
`memh(Rs) = Rt`	32	16	Store lower half (bits 15:0)
`memh(Rs) = Rt.H`	32	16	Store upper half (bits 31:16)
`memh(Rs) = #s8`	8	16	Sign-extend 8 bits to 16 bits
`memw(Rs) = Rt`	32	32	Store word
`memw(Rs) = #s8`	8	32	Sign-extend 8 bits to 32 bits
`memd(Rs) = Rtt`	64	64	Store doubleword
`allocframe(#u11)`	64	64	See Chapter 7

NOTE: The memory store instructions belong to instruction class ST, and can execute only in slot 0 or – when part of a dual store – slot 1.

5.4 Dual stores

Two memory store instructions can appear in the same instruction packet. The resulting operation is considered a dual store. For example:

```
{
   memw(R5) = R2       // Dual store
   memh(R6) = R3
}
```

Unlike most packetized operations, dual stores do not execute in parallel (Section 3.3.1). Instead, the store instruction in Slot 1 effectively executes first, followed by the store instruction in Slot 0.

NOTE: The store instructions in a dual store must belong to instruction class ST, and can execute only in Slots 0 and 1.

5.5 Slot 1 store with slot 0 load

A slot 1 store operation with a slot 0 load operation can appear in a packet. The packet attribute :mem_noshuf inhibits the instruction reordering that is otherwise done by the assembler. For example:

```
{
       memw(R5) = R2   // Slot 1 store
       R3 = memh(R6)   // Slot 0 load
}:mem_noshuf
```

Unlike most packetized operations, these memory operations do not execute in parallel (Section 3.3.1). Instead, the store instruction in Slot 1 effectively executes first, followed by the load instruction in Slot 0. If the addresses of the two operations are overlapping, the load receives the newly stored data.

5.6 New-value stores

A memory store instruction can store a register that is assigned a new value in the same instruction packet (Section 3.3). This feature is expressed in assembly language by appending the suffix "**.**new" to the source register. For example:

```
{
   R2 = memh(R4+#8)        // load halfword
   memw(R5) = R2.new       // store newly-loaded value
}
```

New-value store instructions have the following restrictions:

- If an instruction uses auto-increment or absolute-set addressing mode (Section 5.8), its address register cannot be used as the new-value register.
- If an instruction produces a 64-bit result, its result registers cannot be used as the new-value register.
- If the instruction that sets a new-value register is conditional (Section 6.1.2), it must always execute.

NOTE: The new-value store instructions belong to instruction class NV, and can execute only in Slot 0.

5.7 Mem-ops

Mem-ops perform basic arithmetic, logical, and bit operations directly on memory operands, without the need for a separate load or store. Mem-ops can be performed on byte, halfword, or word sizes.

Table 5-4 Mem-ops

Syntax	Operation
`memXX(Rs+#u6) [+-|&] = Rt`	Arithmetic/logical on memory
`memXX(Rs+#u6) [+-] = #u5`	Arithmetic on memory
`memXX(Rs+#u6) = clrbit(#u5)`	Clear bit in memory
`memXX(Rs+#u6) = setbit(#u5)`	Set bit in memory

NOTE: The mem-op instructions belong to instruction class MEMOP, and can execute only in slot 0.

5.8 Addressing modes

Table 5-5 Addressing modes

Mode	Syntax	Operation [1]
Absolute	`memXX(##address)`	`EA = address`
Absolute-set	`memXX(Re=##address)`	`EA = address` `Re = address`
Absolute with register offset	`memXX(Ru<<#u2+##U32)`	`EA = imm + (Ru << #u2)`
Global pointer relative	`memXX(GP+#immediate)` `memXX(#immediate)`	`EA = GP + immediate`
Indirect	`memXX(Rs)`	`EA = Rs`
Indirect with offset	`memXX(Rs+#s11)`	`EA = Rs + imm`
Indirect with register offset	`memXX(Rs+Ru<<#u2)`	`EA = Rs + (Ru << #u2)`
Indirect with auto-increment immediate	`memXX(Rx++#s4)`	`EA = Rx;` `Rx += (imm)`
Indirect with auto-increment register	`memXX(Rx++Mu)`	`EA = Rx;` `Rx += Mu`
Circular with auto-increment immediate	`memXX(Rx++#s4:circ(Mu))`	`EA = Rx;` `Rx = circ_add(Rx,imm,Mu)`
Circular with auto-increment register	`memXX(Rx++I:circ(Mu))`	`EA = Rx;` `Rx = circ_add(Rx,I,Mu)`
Bit-reversed with auto-increment register	`memXX(Rx++Mu:brev)`	`EA = Rx.H + bit_reverse(Rx.L)` `Rx += Mu`

[1] EA (effective address) is equivalent to VA (virtual address).

5.8.1 Absolute

The absolute addressing mode uses a 32-bit constant value as the effective memory address. For example:

```
R2 = memw(##100000)    // Load R2 with word from addr 100000
memw(##200000) = R4    // Store R4 to word at addr 200000
```

5.8.2 Absolute-set

The absolute-set addressing mode assigns a 32-bit constant value to the specified general register, then uses the assigned value as the effective memory address. For example:

```
R2 = memw(R1=##400000)    // Load R2 with word from addr 400000
                          //  and load R1 with value 400000
memw(R3=##600000) = R4    // Store R4 to word at addr 600000
                          //  and load R3 with value 600000
```

5.8.3 Absolute with register offset

The absolute with register offset addressing mode performs an arithmetic left shift of a 32-bit general register value by the amount specified in a 2-bit unsigned immediate value, and then adds the shifted result to an unsigned 32-bit constant value to create the 32-bit effective memory address. For example:

```
R2 = memh(R3 << #3 + ##100000) // load R2 with signed halfword
                               // from addr [100000 + (R3 << 3)]
```

The 32-bit constant value is the base address, and the shifted result is the byte offset.

NOTE: This addressing mode is useful for loading an element from a global table, where the immediate value is the name of the table, and the register holds the index of the element.

5.8.4 Global pointer relative

The global pointer relative addressing mode adds an unsigned offset value to the Hexagon processor global data pointer GP to create the 32-bit effective memory address. This addressing mode accesses global and static data in C.

Global pointer relative addresses can be expressed two ways in assembly language:

- By explicitly adding an unsigned offset value to register GP
- By specifying only an immediate value as the instruction operand

For example:

```
R2 = memh(GP+#100)      // Load R2 with signed halfword
                        // from [GP + 100 bytes]

R3 = memh(#2000)        // Load R3 with signed halfword
                        // from [GP + #2000 - _SDA_BASE]
```

Specifying only an immediate value causes the assembler and linker to automatically subtract the value of the special symbol `_SDA_BASE_` from the immediate value, and use the result as the effective offset from GP.

The global data pointer is programmed in the GDP field of register GP (Section 2.2.8). This field contains an unsigned 26-bit value that specifies the most significant 26 bits of the 32-bit global data pointer. The least significant 6 bits of the pointer are always defined as zero.

The memory area referenced by the global data pointer is known as the global data area. It can be up to 512 KB in length, and – because of the way the global data pointer is defined – must be aligned to a 64-byte boundary in virtual memory.

When expressed in assembly language, the offset values used in global pointer relative addressing always specify byte offsets from the global data pointer. The offsets must be integral multiples of the size of the instruction data type.

Table 5-6 Offset ranges (global pointer relative)

Data type	Offset range	Offset must be multiple of
doubleword	0 ... 524280	8
word	0 ... 262140	4
halfword	0 ... 131070	2
byte	0 ... 65535	1

NOTE: When using global pointer relative addressing, the immediate operand should be a symbol in the `.sdata` or `.sbss` section to ensure that the offset is valid.

5.8.5 Indirect

The indirect addressing mode uses a 32-bit value stored in a general register as the effective memory address. For example:

```
R2 = memub(R1)   // load R2 with unsigned byte from addr R1
```

5.8.6 Indirect with offset

The indirect with offset addressing mode adds a signed offset value to a general register value to create the 32-bit effective memory address. For example:

```
R2 = memh(R3 + #100)   // load R2 with signed halfword
                       // from [R3 + 100 bytes]
```

When expressed in assembly language, the offset values always specify byte offsets from the general register value. The offsets must be integral multiples of the size of the instruction data type.

Table 5-7 Offset ranges (indirect with offset)

Data type	Offset range	Offset must be multiple of
doubleword	-8192 ... 8184	8
word	-4096 ... 4092	4
halfword	-2048 ... 2046	2
byte	-1024 ... 1023	1

NOTE: The offset range is smaller for conditional instructions (Section 5.9).

5.8.7 Indirect with register offset

The indirect with register offset addressing mode adds a 32-bit general register value to the result created by performing an arithmetic left shift of a second 32-bit general register value by the amount specified in a 2-bit unsigned immediate value, forming the 32-bit effective memory address. For example:

```
R2 = memh(R3+R4<<#1)   // load R2 with signed halfword
                       // from [R3 + (R4 << 1)]
```

The register values always specify byte addresses.

5.8.8 Indirect with auto-increment immediate

The indirect with auto-increment immediate addressing mode uses a 32-bit value stored in a general register to specify the effective memory address. However, after the address is accessed, a signed value (known as the increment) is added to the register so it specifies a different memory address (which is accessed in a subsequent instruction). For example:

```
R2 = memw(R3++#4)    // R3 contains the effective address
                     // R3 is then incremented by 4
```

When expressed in assembly language, the increment values always specify byte offsets from the general register value. The offsets must be integral multiples of the size of the instruction data type.

Table 5-8 Increment ranges (indirect with auto-increment immediate)

Data type	Increment range	Increment must be multiple of
doubleword	-64 ... 56	8
word	-32 ... 28	4
halfword	-16 ... 14	2
byte	-8 ... 7	1

5.8.9 Indirect with auto-increment register

The indirect with auto-increment register addressing mode is functionally equivalent to indirect with auto-increment immediate, but uses a modifier register Mx (Section 2.2.4) instead of an immediate value to hold the increment. For example:

```
R2 = memw(R0++M1)    // The effective addr is the value of R0.
                     // Next, M1 is added to R0 and the result
                     // is stored in R0.
```

When auto-incrementing with a modifier register, the increment is a signed 32-bit value that is added to the general register. This offers two advantages over auto-increment immediate:

- A larger increment range
- Variable increments (because the modifier register can be programmed at runtime)

The increment value always specifies a byte offset from the general register value.

NOTE: The signed 32-bit increment range is identical for all instruction data types (doubleword, word, halfword, byte).

5.8.10 Circular with auto-increment immediate

The circular with auto-increment immediate addressing mode is a variant of indirect with auto-increment addressing – it accesses data buffers in a modulo wrap around fashion. Circular addressing is commonly used in data stream processing.

Circular addressing is expressed in assembly language with the address modifier "`:circ(Mx)`", where Mx specifies a modifier register that is programmed to specify the circular buffer (Section 2.2.4). For example:

```
R0 = memb(R2++#4:circ(M0))    // Load from R2 in circ buf specified
                              //  by M0
memw(R2++#8:circ(M1)) = R0    // Store to R2 in circ buf specified
                              //  by M1
```

Circular addressing is set up by programming the following elements:

- The Length field of the `Mx` register is set to the length (in bytes) of the circular buffer to access. A circular buffer can be from 4 to (128K-1) bytes long.
- Bits 27:24 of the Mx register are always set to 0.
- The circular start register CSx that corresponds to Mx (CS0 for M0, CS1 for M1) is set to the start address of the circular buffer.

In circular addressing, after memory is accessed at the address specified in the general register, the general register is incremented by the immediate increment value and then modulo'd by the circular buffer length to implement wraparound access of the buffer.

When expressed in assembly language, the increment values always specify byte offsets from the general register value. The offsets must be integral multiples of the size of the instruction data type.

Table 5-9 Increment ranges (circular with auto-increment immediate addressing)

Data type	Increment range	Increment must be multiple of
doubleword	-64 ... 56	8
word	-32 ... 28	4
halfword	-16 ... 14	2
byte	-8 ... 7	1

When programming a circular buffer, the following rules apply:

- The start address must be aligned to the native access size of the buffer elements.
- ABS(Increment) < Length. The absolute value of the increment must be less than the buffer length.
- Access size < (Length-1). The memory access size (1 for byte, 2 for halfword, 4 for word, 8 for doubleword) must be less than (Length-1).
- Buffers must not wrap around in the 32-bit address space.

NOTE: If any of these rules are not followed, the execution result is undefined.

The following example sets up and accesses a 150-byte circular buffer:

```
R4.H = #0                       // M0[27:24]= 0x0
R4.L = #150                     // length = 150
M0 = R4
R2 = ##cbuf                     // start addr = cbuf
CS0 = R2
R0 = memb(R2++#4:circ(M0))      // Load byte from circ buf
                                // specified by M0/CS0
                                // inc R2 by 4 after load
                                // wrap R2 around if >= 150
```

The following C function describes the behavior of the circular add function:

```
unsigned int
fcircadd(unsigned int pointer, int offset,
    unsigned int M_reg, unsigned int CS_reg)
{
   unsigned int length;
   int new_pointer, start_addr, end_addr;

   length = (M_reg&0x01ffff);  // Lower 17 bits gives buffer size
   new_pointer = pointer+offset;
   start_addr = CS_reg;
   end_addr = CS_reg + lenth;
   if (new_pointer >= end_addr) {
      new_pointer -= length;
}  else if (new_pointer < start_addr) {
      new_pointer += length;
}
   return (new_pointer);
}
```

5.8.11 Circular with auto-increment register

The circular with auto-increment register addressing mode is functionally equivalent to circular with auto-increment immediate, but uses a register instead of an immediate value to hold the increment.

Register increments are specified in circular addressing instructions by using the symbol `I` as the increment (instead of an immediate value). For example:

```
R0 = memw(R2++I:circ(M1))       // Load byte with incr of I*4 from
                                // circ buf specified by M1/CS1
```

When auto-incrementing with a register, the increment is a signed 11-bit value that is added to the general register. This offers two advantages over circular addressing with immediate increments:

- Larger increment ranges
- Variable increments (because the increment register can be programmed at runtime)

The circular register increment value is programmed in the `I` field of the modifier register Mx (Section 2.2.4) as part of setting up the circular data access. This register field holds the signed 11-bit increment value.

Increment values are expressed in units of the buffer element data type, and are automatically scaled at runtime to the proper data access size.

Table 5-10 Increment ranges (circular with auto-increment register addressing)

Data type	Increment range	Increment must be multiple of
doubleword	-8192 ... 8184	8
word	-4096 ... 4092	4
halfword	-2048 ... 2046	2
byte	-1024 ... 1023	1

When programming a circular buffer (with either a register or immediate increment), all the rules that apply to circular addressing must be followed – for details see Section 5.8.10.

NOTE: If any of these rules are not followed, the execution result is undefined.

5.8.12 Bit-reversed with auto-increment register

The bit-reversed with auto-increment register addressing mode is a variant of indirect with auto-increment addressing – it accesses data buffers using an address value that is the bit-wise reversal of the value stored in the general register. Bit-reversed addressing is used in fast Fourier transforms (FFT) and Viterbi encoding.

The bit-wise reversal of a 32-bit address value is defined as follows:

- The lower 16 bits are transformed by exchanging bit 0 with bit 15, bit 1 with bit 14, and so on.
- The upper 16 bits remain unchanged.

Bit-reversed addressing is expressed in assembly language with the address modifier "`:brev`". For example:

```
R2 = memub(R0++M1:brev)   // The address is (R0.H | bitrev(R0.L))
                           // The original R0 (not reversed) is added
                          // to M1 and written back to R0
```

The initial values for the address and increment must be set in bit-reversed form, with the hardware bit-reversing the bit-reversed address value to form the effective address.

The buffer length for a bit-reversed buffer must be an integral power of 2, with a maximum length of 64K bytes.

To support bit-reversed addressing, buffers must be properly aligned in memory. A bit-reversed buffer is properly aligned when its starting byte address is aligned to a power of 2 greater than or equal to the buffer size (in bytes). For example:

```
int bitrev_buf[256] __attribute__((aligned(1024)));
```

The bit-reversed buffer declared above is aligned to 1024 bytes because the buffer size is 1024 bytes (256 integer words × 4 bytes), and 1024 is an integral power of 2.

The buffer location pointer for a bit-reversed buffer must be initialized so thatthe least-significant 16 bits of the address value are bit-reversed.

The increment value must be initialized to the following value:

```
bitreverse(buffer_size_in_bytes / 2)
```

...where `bitreverse` is defined as bit-reversing the least-significant 16 bits while leaving the remaining bits unchanged.

NOTE: To simplify the initialization of the bit-reversed pointer, bit-reversed buffers can be aligned to a 64K byte boundary. This initializes the bit-reversed pointer to the base address of the bit-reversed buffer, with no bit-reversing required for the least-significant 16 bits of the pointer value (which are set to 0 by the 64K alignment).

Because buffers allocated on the stack only have an alignment of 8 bytes or less, in most cases bit-reversed buffers should not be declared on the stack.

After a bit-reversed memory access is complete, the general register is incremented by the register increment value. The value in the general register is never affected by the bit-reversal that is performed as part of the memory access.

NOTE: The Hexagon processor supports only register increments for bit-reversed addressing – it does not support immediate increments.

5.9 Conditional load/stores

Some load and store instructions can be executed conditionally based on predicate values that were set in a previous instruction. The compiler generates conditional loads and stores to increase instruction-level parallelism.

Conditional loads and stores are expressed in assembly language with the instruction prefix "`if` (`pred_expr`)", where `pred_expr` specifies a predicate register expression (Section 6.1). For example:

```
if (P0) R0 = memw(R2)              // Conditional load
if (!P2) memh(R3 + #100) = R1      // Conditional store
if (P1.new) R3 = memw(R3++#4)      // Conditional load
```

Not all addressing modes are supported in conditional loads and stores. Table 5-11 shows which modes are supported.

Table 5-11 Addressing modes (conditional load/store)

Addressing mode	Conditional
Absolute	Yes
Absolute-set	No
Absolute with register offset	No
Global pointer relative	No
Indirect	Yes
Indirect with offset	Yes
Indirect with register offset	Yes
Indirect with auto-increment immediate	Yes
Indirect with auto-increment register	No
Circular with auto-increment immediate	No
Circular with auto-increment register	No
Bit-reversed with auto-increment register	No

When a conditional load or store instruction uses Indirect with offset addressing mode, the offset range is smaller than the range normally defined for indirect-with-offset addressing.

Table 5-12 Conditional and normal offset ranges (indirect with offset addressing)

Data type	Offset range (conditional)	Offset range (normal)	Offset must be multiple of
doubleword	0 ... 504	-8192 ... 8184	8
word	0 ... 252	-4096 ... 4092	4
halfword	0 ... 126	-2048 ... 2046	2
byte	0 ... 63	-1024 ... 1023	1

NOTE: For more information on conditional execution, see Chapter 6.

5.10 Cache memory

The Hexagon processor has a cache-based memory architecture:

- A level 1 instruction cache holds recently fetched instructions.
- A level 1 data cache holds recently accessed data memory.

Load/store operations that access memory through the level 1 caches are referred to as cached accesses.

Load/stores that bypass the level 1 caches are referred to as uncached accesses.

The memory management unit (MMU) of the Hexagon processor can configure specific memory areas to perform cached or uncached accesses. The operating system is responsible for programming the MMU.

Two types of caching are supported (as cache modes):

- Write-through caching keep the cache data consistent with external memory by always writing to the memory any data that is stored in the cache.
- Writeback caching stores data in the cache without being immediately written to external memory. Cached data that is inconsistent with external memory is referred to as dirty.

The Hexagon processor includes dedicated cache maintenance instructions that push dirty data out to external memory.

5.10.1 Uncached memory

In some cases, load/store operations must bypass the cache memories and be serviced externally (for example, when accessing memory-mapped I/O, registers, and peripheral devices, or other system defined entities). The operating system is responsible for configuring the MMU to generate uncached memory accesses.

Uncached memory is categorized into two distinct types:

- Device-type is for accessing memory that has side-effects (such as a memory-mapped FIFO peripheral). The hardware ensures that interrupts do not cancel a pending device access. The hardware does not reorder device accesses. Mark peripheral control registers as device-type.
- Uncached-type is for memory-like memory. No side effects are associated with an access. The hardware can load from uncached memory multiple times. The hardware can reorder uncached accesses.

For instruction accesses, device-type memory is functionally identical to uncached-type memory. For data accesses, they are different.

Code can execute directly from the L2 cache, bypassing the L1 cache.

5.10.2 Tightly coupled memory

The Hexagon processor supports tightly coupled instruction memory at Level 1, which is defined as memory with similar access properties to the instruction cache.

Tightly coupled memory is also supported at level 2, which is defined as backing store to the primary caches.

5.10.3 Cache maintenance operations

The Hexagon processor includes dedicated cache maintenance instructions that invalidate cache data or push dirty data out to external memory.

The cache maintenance instructions operate on specific memory addresses. If the instruction causes an address error (due to a privilege violation), the processor raises an exception.

NOTE: The exception to this rule is the dcfetch operation, which never causes a processor exception.

Whenever maintenance operations are performed on the instruction cache, the isync instruction (Section 5.11) must execute immediately after. This instruction ensures that subsequent instructions observe the maintenance operations.

Table 5-13 Cache instructions (user-level)

Syntax	Permitted in packet	Operation
`icinva(Rs)`	Solo [1]	Instruction cache invalidate. Look up instruction cache at address Rs. If the address is in the cache, invalidate it.
`dccleaninva(Rs)`	Slot 1 empty or ALU32 only	Data cache clean and invalidate. Look up data cache at address Rs. If the address is in the cache and has dirty data, flush that data out to memory. The cache line is then invalidated, whether or not dirty data was written.
`dccleana(Rs)`	Slot 1 empty or ALU32 only	Data cache clean. Look up data cache at address Rs. If the address is in the cache and has dirty data, flush that data out to memory.
`dcinva(Rs)`	Slot 1 empty or ALU32 only	Equivalent to `dccleaninva(Rs)`.
`dcfetch(Rs)`	Normal [2]	Data cache prefetch. Prefetch data at address Rs into the data cache. **NOTE:** This instruction does not cause an exception.
`l2fetch(Rs,Rt)`	ALU32 or XTYPE only	L2 cache prefetch. Prefetch data from memory specified by Rs and Rt into L2 cache.

[1] *Solo* means that the instruction must not be grouped with other instructions in a packet.

[2] *Normal* means that the normal instruction-grouping constraints apply.

5.10.4 L2 cache operations

Cache maintenance operations operate on both the L1 and L2 caches.

The data cache coherency operations (including clean, invalidate, and clean and invalidate) affect both the L1 and L2 caches, and ensure that the memory hierarchy remains coherent.

However, the instruction cache invalidate operation affects only the L1 cache. Therefore, invalidating instructions that might be in the L1 or L2 caches requires a two-step procedure:

1. Use `icinva` to invalidate instructions from the L1 cache.
2. Use `dcinva` separately to invalidate instructions from the L2 cache.

5.10.5 Cache line zero

The Hexagon processor includes the instruction dczeroa. This instruction allocates a line in the L1 data cache and clears it (by storing all zeros). The behavior is as follows:

- The Rs register value must be 32-byte aligned. If it is unaligned, the processor raises an unaligned error exception.
- For a cache hit, the specified cache line is cleared (written with all zeros) and made dirty.
- For a cache miss, the specified cache line is not fetched from external memory. Instead, the line is allocated in the data cache, cleared, and made dirty.

This instruction is useful in optimizing write-only data. It allows for the use of writeback pages – which are the most power and performance efficient – without the need to initially fetch the line to write. This removes unnecessary read bandwidth and latency.

NOTE: The `dczeroa` operation has the same exception behavior as writeback stores.

A packet with `dczeroa` must have slot 1 either empty or containing an ALU32 instruction.

5.10.6 Cache prefetch types supported by the Hexagon processor

5.10.6.1 Hardware-based instruction cache prefetching

L1 and L2 instruction cache prefetching can be enabled or disabled on a per-thread basis by setting the HFI field in the User status register.

5.10.6.2 Software-based data cache prefetching

The Hexagon processor includes the dcfetch instruction, which queries the L1 data cache based on the address specified in the instruction:

- If the address is present in the cache, no action is taken.
- If the cache line for the address is missing, the processor attempts to fill the cache line from the next level of memory. The thread does not stall, but rather continues executing while the cache line fill occurs in the background.

- If the address is invalid, no exception is generated and the dcfetch instruction is treated as a NOP.

5.10.6.3 Software-based L2fetch

More powerful L2 prefetching – of data or instructions – is provided by the l2fetch instruction, which specifies an area of memory that is prefetched by the Hexagon processor's hardware prefetch engine. l2fetch specifies two registers (Rs and Rt) as operands. Rs contains the 32-bit virtual start address of the memory area to prefetch. Rt contains three bit fields that further specify the memory area:

- Rt[15:8] – `Width`, specifies the width (in bytes) of a block of memory to fetch.
- Rt[7:0] – `Height`, specifies the number of `Width`-sized blocks to fetch.
- Rt[31:16] – `Stride`, specifies an unsigned byte offset that increments the pointer after each `Width`-sized block is fetched.

The l2fetch instruction is nonblocking: it initiates a prefetch operation that is performed in the background by the prefetch engine while the thread continues to execute Hexagon processor instructions.

The prefetch engine requests all lines in the specified memory area. If the line(s) of interest are already resident in the L2 cache, the prefetch engine performs no action. If the lines are not in the L2 cache, the prefetch engine attempts to fetch them.

The prefetch engine makes a best effort to prefetch the requested data, and attempts to perform prefetching at a lower priority than demand fetches. This prevents the prefetch engine from adding bus traffic when the system is under a heavy load.

If a program executes an `l2fetch` instruction while the prefetch operation from a previous l2fetch is still active, the prefetch engine halts the current prefetch operation.

NOTE: Executing l2fetch with any bit field operand programmed to zero cancels prefetch activity.

The status of the prefetch operation is maintained in the PFA field of the user status register. This field can determine whether a prefetch operation has completed.

With respect to MMU permissions and error checking, the `l2fetch` instruction behaves similarly to a load instruction. If the virtual address causes a processor exception, the exception is taken. This differs from the `dcfetch` instruction, which is treated as a NOP in the presence of a translation or protection error.

NOTE: Prefetches are dropped when the generated prefetch address resides on a different page than the start address. The programmer must use sufficiently large pages to ensure this does not occur.

Figure 5-2 shows two examples of using the l2fetch instruction. The first shows a box prefetch, where a 2D range of memory is defined within a larger frame. The second example shows a prefetch for a large linear memory area of size (`Lines` * 128).

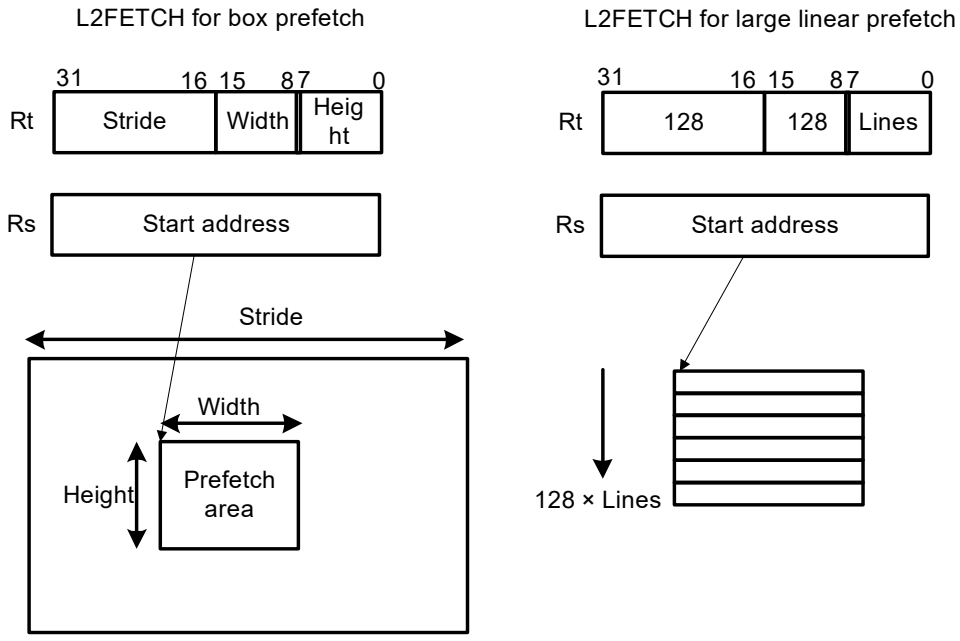


Figure 5-2 L2fetch instruction

5.10.6.4 Hardware-based data cache prefetching

L1 data cache prefetching is enabled or disabled on a per-thread basis by setting the HFD field in the User status register.

When data cache prefetching is enabled, the Hexagon processor observes patterns of data cache misses, and attempts to predict future misses based on any recurring patterns of misses where the addresses are separated by a constant stride. If such patterns are found, the processor attempts to automatically prefetch future cache lines.

Data cache prefetching is user-enabled at four levels of aggressiveness:

- HFD = 00: No prefetching
- HFD = 01: Prefetch up to four lines for misses originating from a load, with a post-update addressing mode that occurs within a hardware loop
- HFD = 10: Prefetch up to four lines for misses originating from loads that occur within a hardware loop
- HFD = 11: Prefetch up to eight lines for misses originating from loads

5.11 Memory ordering

Some devices might require synchronization of stores and loads when they are accessed. In this case, a set of processor instructions enable programmer control of the synchronization and ordering of memory accesses.

Table 5-14 Memory ordering instructions

Syntax	Operation
`isync`	Instruction synchronize. This instruction should execute after any instruction cache maintenance operation.
`syncht`	Synchronize transactions. Perform heavyweight synchronization. Ensure that previous program transactions (for example, memw_locked, cached and uncached load/store) are complete before execution resumes past this instruction. The syncht instruction ensures that outstanding memory operations from all threads are complete before the syncht instruction is committed.
`barrier`	Set memory barrier. Ensure proper ordering between the program accesses performed before the instruction and those performed after the instruction. All accesses before the barrier are globally observable before any access occurring after the barrier can be observed. The barrier instruction ensures that all outstanding memory operations from the thread executing the barrier are complete before the instruction is committed.

Data memory accesses and program memory accesses are treated separately and held in separate caches. Software should ensure coherency between data and program code if necessary.

For example, with generated or self-modified code, the modified code is placed in the data cache and can be inconsistent with program cache. The software must explicitly force modified data cache lines to memory (either by using a write-through policy, or through explicit cache clean instructions). Use a barrier instruction to ensure completion of the stores. Finally, invalidate relevant instruction cache contents so the new instructions can be refetched.

Here is the recommended code sequence to change and execute an instruction:

```
ICINVA(R1)         // Clear code from instruction cache
ISYNC              // Ensure that ICINVA is finished
MEMW(R1)=R0        // Write the new instruction
DCCLEANINVA(R1)    // Force data out of data cache
SYNCHT             // Ensure that it is in memory
JUMPR R1           // Can now execute code at R1
```

NOTE: The memory-ordering instructions must not be grouped with other instructions in a packet, otherwise the behavior is undefined.

This code sequence differs from the one used in previous processor versions.

5.12 Atomic operations

The Hexagon processor includes a load locked / store conditional (LL/SC) mechanism to provide the atomic read-modify-write operation that is necessary to implement synchronization primitives such as semaphores and mutexes.

These primitives synchronize the execution of different software programs running concurrently on the Hexagon processor. They can also provide atomic memory support between the Hexagon processor and external blocks.

Table 5-15 Atomic instructions

Syntax	Description
`Rd = memw_locked(Rs)`	Load locked word. Reserve lock on word at address Rs.
`memw_locked(Rs,Pd) = Rt`	Store conditional word. If no other atomic operation has been performed at the address (that is, atomicity is ensured), perform the store to the word at address Rs and return TRUE in Pd; otherwise return FALSE. TRUE indicates that the LL and SC operations were performed atomically.
`Rdd = memd_locked(Rs)`	Load locked doubleword. Reserve lock on doubleword at address Rs.
`memd_locked(Rs,Pd) = Rtt`	Store conditional doubleword. If no other atomic operation has been performed at the address (that is, atomicity is ensured), perform the store to the doubleword at address Rs and return TRUE in Pd; otherwise return FALSE. TRUE indicates that the LL and SC operations have been performed atomically.

Here is the recommended code sequence to acquire a mutex:

```
// Assume mutex address is held in R0
// Assume R1,R3,P0,P1 are scratch

lockMutex:
   R3 = #1
lock_test_spin:
   R1 = memw_locked(R0)          // Do normal test to wait
   P1 = cmp.eq(R1,#0)            // for lock to be available
   if (!P1) jump lock_test_spin
   memw_locked(R0,P0) = r3       // Do store conditional (SC)
   if (!P0) jump lock_test_spin  // was LL and SC done atomically?
```

Here is the recommended code sequence to release a mutex:

```
// Assume mutex address is held in R0
// Assume R1 is scratch

R1 = #0
memw(R0) = R1
```

Atomic `memX_locked` operations are supported for external accesses that use the advanced extensible interface (AXI) bus and support atomic operations. To perform load-locked operations with external memory, the operating system must define the memory page as uncacheable, otherwise the processor behavior is undefined.

If a load locked operation is performed on an address that does not support atomic operations, the behavior is undefined.

For atomic operations on cacheable memory, the page attributes must be set to cacheable and write-back, otherwise the behavior is undefined. Cacheable memory must be used when threads must synchronize with each other.

NOTE: External `memX_locked` operations are not supported on the AHB. If they are performed on the AHB, the behavior is undefined.

6 Conditional execution

The Hexagon processor uses a conditional execution model based on compare instructions that set predicate bits in one of four 8-bit predicate registers (P0 through P3). These predicate bits can conditionally execute certain instructions.

Conditional scalar operations examine only the least-significant bit in a predicate register, while conditional vector operations examine multiple bits in the register.

Branch instructions are the main consumers of the predicate registers.

6.1 Scalar predicates

Scalar predicates are 8-bit values in conditional instructions to represent truth values:

- 0xFF represents true
- 0x00 represents false

The Hexagon processor provides the four 8-bit predicate registers P0 through P3 to hold scalar predicates (Section 2.2.5). These registers are assigned values by the predicate-generating instructions, and examined by the predicate-consuming instructions.

6.1.1 Generating scalar predicates

The following instructions generate scalar predicates:

- Compare byte, halfword, word, doubleword
- Compare single- and double-precision floating point
- Classify floating-point value
- Compare bitmask
- Bounds check
- TLB match
- Store conditional

Table 6-1 Scalar predicate-generating instructions

Syntax	Operation

`Pd = cmpb.eq(Rs,{Rt,#u8})`
`Pd = cmph.eq(Rs,{Rt,#s8})`
`Pd = [!]cmp.eq(Rs,{Rt,#s10})`
`Pd = cmp.eq(Rss,Rtt)`
`Pd = sfcmp.eq(Rs,Rt)`
`Pd = dfcmp.eq(Rss,Rtt)`
	Equal (signed).
	Compare register Rs to Rt or a signed immediate for equality. Assign Pd the resulting truth value.

`Pd = cmpb.gt(Rs,{Rt,#s8}`
`Pd = cmph.gt(Rs,{Rt,#s8})`
`Pd = [!]cmp.gt(Rs,{Rt,#s10})`
`Pd = cmp.gt(Rss,Rtt)`
`Pd = sfcmp.gt(Rs,Rt)`
`Pd = dfcmp.gt(Rss,Rtt)`
	Greater than (signed).
	Compare register Rs to Rt or a signed immediate for signed greater than. Assign Pd the resulting truth value.

`Pd = cmpb.gtu(Rs,{Rt,#u7})`
`Pd = cmph.gtu(Rs,{Rt,#u7})`
`Pd = [!]cmp.gtu(Rs,{Rt,#u9})`
`Pd = cmp.gtu(Rss,Rtt)`
	Greater than (unsigned).
	Compare register Rs to Rt or an unsigned immediate for unsigned greater than. Assign Pd the resulting truth value.

`Pd = cmp.ge(Rs,#s8)`
`Pd = sfcmp.ge(Rs,Rt)`
`Pd = dfcmp.ge(Rss,Rtt)`
	Greater than or equal (signed).
	Compare register Rs to Rt or a signed immediate for signed greater than or equal. Assign Pd the resulting truth value.

`Pd = cmp.geu(Rs,#u8)`	Greater than or equal (unsigned). Compare register Rs to an unsigned immediate for unsigned greater than or equal. Assign Pd the resulting truth value.
`Pd = cmp.lt(Rs,Rt)`	Less than (signed). Compare register Rs to Rt for signed less than. Assign Pd the resulting truth value.
`Pd = cmp.ltu(Rs,Rt)`	Less than (unsigned). Compare register Rs to Rt for unsigned less than. Assign Pd the resulting truth value.
`Pd = sfcmp.uo(Rs,Rt)` `Pd = dfcmp.uo(Rss,Rtt)`	Unordered (signed). Determine if register Rs or Rt is set to the value NaN. Assign Pd the resulting truth value.
`Pd=sfclass(Rs,#u5)` `Pd=dfclass(Rss,#u5)`	Classify value (signed). Determine if register Rs is set to any of the specified classes. Assign Pd the resulting truth value.
`Pd = [!]tstbit(Rs,{Rt,#u5})`	Test if bit set. Rt or an unsigned immediate specifies a bit position. Test if the bit in Rs that is specified by the bit position is set. Assign Pd the resulting truth value.
`Pd = [!]bitsclr(Rs,{Rt,#u6})`	Test if bits clear. Rt or an unsigned immediate specifies a bitmask. Test if the bits in Rs that are specified by the bitmask are all clear. Assign Pd the resulting truth value.
`Pd = [!]bitsset(Rs,Rt)`	Test if bits set. Rt specifies a bitmask. Test if the bits in Rs that are specified by the bitmask are all set. Assign Pd the resulting truth value.

Table 6-1 Scalar predicate-generating instructions (cont.)

`memw_locked(Rs,Pd) = Rt` `memd_locked(Rs,Pd) = Rtt`	Store conditional. If no other atomic operation has been performed at the address (atomicity is ensured), perform the store to the word at address Rs. Assign Pd the resulting truth value.
`Pd = boundscheck(Rs,Rtt)`	Bounds check. Determine if Rs falls in the numeric range defined by Rtt. Assign Pd the resulting truth value.
`Pd = tlbmatch(Rss,Rt)`	Determine if TLB entry in Rss matches the ASID:PPN specified in Rt. Assign Pd the resulting truth value.

NOTE: One of the compare instructions (cmp.eq) includes a variant that stores a binary predicate value (0 or 1) in a general register not a predicate register.

6.1.2 Consuming scalar predicates

Certain instructions can be conditionally executed based on the value of a scalar predicate (or alternatively specify a scalar predicate as an input to their operation).

The conditional instructions that consume scalar predicates examine only the least-significant bit of the predicate value. In the simplest case, this bit value directly determines whether the instruction executes:

- 1 indicates that the instruction executes
- 0 indicates that the instruction does not execute

If a conditional instruction includes the operator `!` in its predicate expression, the logical negation of the bit value determines whether the instruction is executed.

Conditional instructions are expressed in assembly language with the instruction prefix `if (`*`pred_expr`*`)`, where *`pred_expr`* specifies the predicate expression. For example:

```
if (P0) jump target          // Jump if P0 is true
if (!P2) R2 = R5             // Assign register if !P2 is true
if (P1) R0 = sub(R2,R3)      // Conditionally subtract if P1
if (P2) R0 = memw(R2)        // Conditionally load word if P2
```

The following instructions can be used as conditional instructions:

- Jumps and calls (Section 8.3)
- Many load and store instructions (Section 5.9)
- Logical instructions (including AND/OR/XOR)
- Shift halfword
- 32-bit add/subtract by register or short immediate
- Sign and zero extend
- 32-bit register transfer and 64-bit combine word
- Register transfer immediate
- Deallocate frame and return

When a conditional load or store executes and the predicate expression is false, the instruction is canceled (including any exceptions that might occur). For example, if a conditional load uses an address with a memory permission violation, and the predicate expression is false, the load does not execute and the exception is not raised.

The mux instruction accepts a predicate as one of its basic operands:

```
Rd = mux(Ps,Rs,Rt)
```

The mux instruction selects either Rs or Rt based on the least significant bit in Ps. If the least-significant bit in Ps is a 1, Rd is set to Rs, otherwise it is set to Rt.

6.1.3 Auto-AND predicates

If multiple compare instructions in a packet write to the same predicate register, the result is the logical AND of the individual compare results. For example:

```
{
   P0 = cmp(A)                        // If A && B,  jump
   P0 = cmp(B)
   if (P0.new) jump:T taken_path
}
```

To perform the corresponding OR operation, the following instructions can compute the negation of an existing compare (using De Morgan's law):

- `Pd = !cmp.{eq,gt}(Rs, {#s10,Rt} )`
- `Pd = !cmp.gtu(Rs, {#u9,Rt} )`
- `Pd = !tstbit(Rs, {#u5,Rt} )`
- `Pd = !bitsclr(Rs, {#u6,Rt} )`
- `Pd = !bitsset(Rs,Rt)`

Auto-AND predicates have the following restrictions:

- If a packet contains `endloopN`, it cannot perform an auto-AND with predicate register P3.
- If a packet contains a register transfer from a general register to a predicate register, no other instruction in the packet can write to the same predicate register. As a result, a register transfer to P3:0 or C5:4 cannot be grouped with any other predicate-writing instruction.
- The instructions `spNloop0`, `decbin`, `tlbmatch`, `memw_locked`, `memd_locked`, `a`, `sub:carry`, `sfcmp`, and `dfcmp` cannot be grouped with another instruction that sets the same predicate register.

NOTE: A register transfer from a predicate register to a predicate register has the same auto-AND behavior as a compare instruction.

6.1.4 Dot-new predicates

The Hexagon processor can generate and use a scalar predicate in the same instruction packet (Section 3.3). This feature is expressed in assembly language by appending the suffix ".new" to the specified predicate register. For example:

```
if (P0.new) R3 = memw(R4)
```

The following C statement and the corresponding assembly code that is generated from it by the compiler is an example of how to use dot-new predicates.

C statement

```
if (R2 == 4)
   R3 = *R4;
else
   R5 = 5;
```

Assembly code

```
{
   P0 = cmp.eq(R2,#4)
   if (P0.new) R3 = memw(R4)
   if (!P0.new) R5 = #5
}
```

In the assembly code, a scalar predicate is generated and then consumed twice within the same instruction packet.

The following conditions apply to using dot-new predicates:

- The predicate must be generated by an instruction in the same packet. The assembler normally enforces this restriction, but if the processor executes a packet that violates this restriction, the execution result is undefined.
- A single packet can contain both the dot-new and normal forms of predicates. The normal form examines the old value in the predicate register, rather than the newly-generated value. For example:

```
{
  P0 = cmp.eq(R2,#4)
  if (P0.new) R3 = memw(R4)  // Use newly-generated P0 value
  if (P0) R5 = #5            // Use previous P0 value
}
```

6.1.5 Dependency constraints

Two instructions in an instruction packet should not write to the same destination register (Section 3.3.5). An exception to this rule is when the two instructions are conditional, and only one of them ever has the predicate expression value true when the packet executes.

For example, the following packet is valid as long as P2 and P3 never both evaluate to true when the packet is executed:

```
{
   if (P2) R3 = #4       // P2, P3, or both must be false
   if (P3) R3 = #7
}
```

Because predicate values change at runtime, the programmer is responsible for ensuring that such packets are always valid during program execution. If they are invalid, the processor takes the following actions:

- When writing to general registers, an error exception is raised.
- When writing to predicate or control registers, the result is undefined.

6.2 Vector predicates

The predicate registers are also used for conditional vector operations. Unlike scalar predicates, vector predicates contain multiple truth values which are generated by vector predicate-generating operations.

For example, a vector compare instruction compares each element of a vector and assigns the compare results to a predicate register. Each bit in the predicate vector contains a truth value indicating the outcome of a separate compare performed by the vector instruction.

The vector mux instruction uses a vector predicate to selectively merge elements from two separate vectors into a single destination vector. This operation is useful for enabling the vectorization of loops with control flow (branches).

The vector instructions that use predicates are described in the following sections.

6.2.1 Vector compare

A vector compare instruction inputs two 64-bit vectors, performs separate compares for each pair of vector elements, and generates a predicate value which contains a bit vector of truth values.

In Figure 6-1 two 64-bit vectors of bytes (contained in Rss and Rtt) are being compared. The result is assigned as a vector predicate to the destination register Pd.

In the example vector predicate shown in Figure 6-1, every other compare result in the predicate is true (for example, 1).

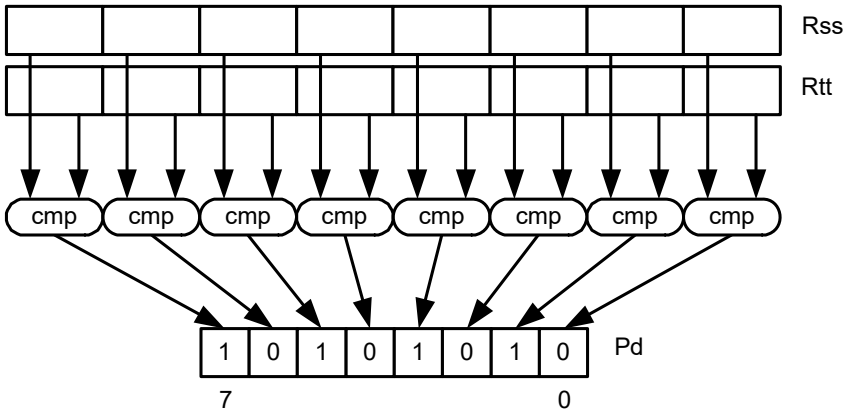


Figure 6-1 Vector byte compare

Figure 6-2 shows how a vector halfword compare generates a vector predicate. Two 64-bit vectors of halfwords are being compared. The result is assigned as a vector predicate to the destination register Pd.

Because a vector halfword compare yields only four truth values, each truth value is encoded as two bits in the generated vector predicate.

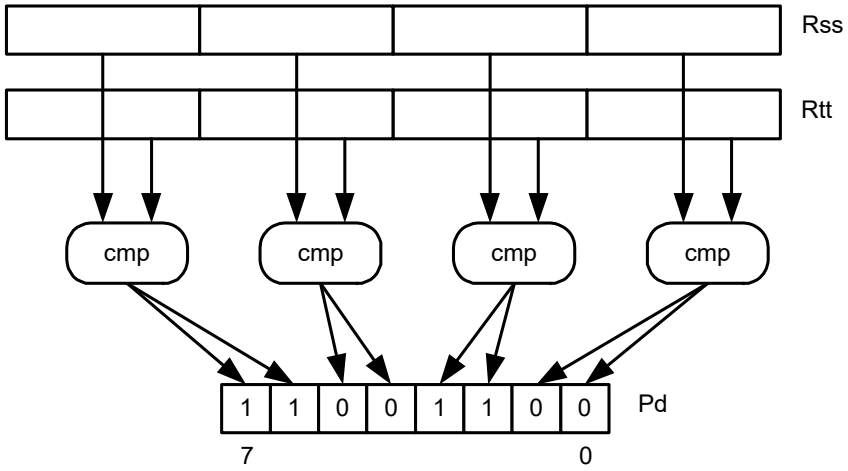


Figure 6-2 Vector halfword compare

6.2.2 Vector mux instruction

A vector mux instruction conditionally selects the elements from two vectors. The instruction takes as input two source vectors and a predicate register. For each byte in the vector, the corresponding bit in the predicate register is used to choose from one of the two input vectors. The combined result is written to the destination register.

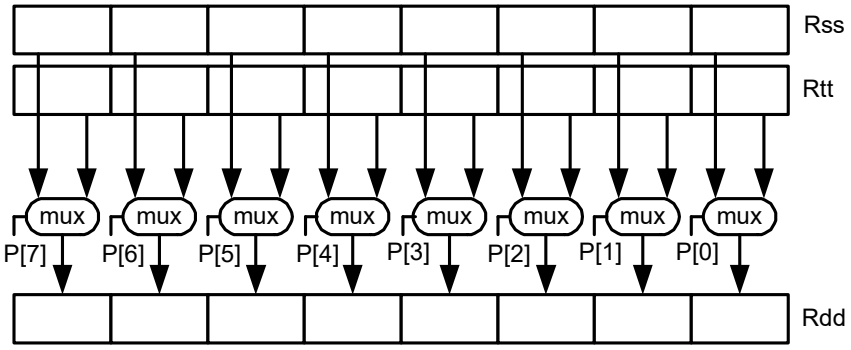


Figure 6-3 Vector mux instruction

Table 6-2 Vector mux instruction

Syntax	Operation
`Rdd = vmux(Ps,Rss,Rtt)`	Select bytes from Rss and Rtt

Changing the order of the source operands in a mux instruction enables formation of both senses of the result. For example:

```
R1:0 = vmux(P0,R3:2,R5:4)     // Choose bytes from R3:2 if true
R1:0 = vmux(P0,R5:4,R3:2)     // Choose bytes from R3:2 if false
```

NOTE: By replicating the predicate bits generated by word or halfword compares, the vector mux instruction can select words or halfwords.

6.2.3 Using vector conditionals

Vector conditional support is used to vectorize loops with conditional statements.

Consider the following C statement:

```
for (i=0; i<8; i++) {
   if (A[i]) {
      B[i] = C[i];
   }
}
```

Assuming arrays of bytes, this code can be vectorized as follows:

```
R1:0 = memd(R_A)             // R1:0 holds A[7]-A[0]
R3 = #0                     // Clear R3:2
R2 = #0
P0 = vcmpb.eq(R1:0,R3:2)     // Compare bytes in A to zero
```

```
R5:4 = memd(R_B)             // R5:4 holds B[7]-B[0]
R7:6 = memd(R_C)             // R7:6 holds C[7]-C[0]
R3:2 = vmux(P0,R7:6,R5:4)    // if (A[i]) B[i]=C[i]
memd(R_B) = R3:2             // store B[7]-B[0]
```

6.3 Predicate operations

The Hexagon processor provides a set of operations for manipulating and moving predicate registers.

Table 6-3 Predicate register instructions

Syntax	Operation
`Pd = Ps`	Transfer predicate Ps to Pd
`Pd = Rs`	Transfer register Rs to predicate Pd
`Rd = Ps`	Transfer predicate Ps to register Rd
`Pd = and(Ps,[!]Pt)`	Set Pd to bitwise AND of Ps and [NOT] Pt
`Pd = or(Ps,[!]Pt)`	Set Pd to bitwise OR of Ps and [NOT] Pt
`Pd = and(Ps, and(Pt,[!]Pu)`	Set Pd to AND of Ps and (AND of Pt and [NOT] Pu)
`Pd = and(Ps, or(Pt,[!]Pu)`	Set Pd to AND of Ps and (OR of Pt and [NOT] Pu)
`Pd = or(Ps, and(Pt,[!]Pu)`	Set Pd to OR of Ps and (AND of Pt and [NOT] Pu)
`Pd = or(Ps, or(Pt,[!]Pu)`	Set Pd to OR of Ps and (OR of Pt and [NOT] Pu)
`Pd = not(Ps)`	Set Pd to bitwise inversion of Ps
`Pd = xor(Ps,Pt)`	Set Pd to bitwise exclusive OR of Ps and Pt
`Pd = any8(Ps)`	Set Pd to 0xFF if any bit in Ps is 1, 0x00 otherwise
`Pd = all8(Ps)`	Set Pd to 0x00 if any bit in Ps is 0, 0xFF otherwise

NOTE: These instructions belong to instruction class CR.

Predicate registers can be transferred to and from the general registers either individually or as register quadruples (Section 2.2.5).

7 Software stack

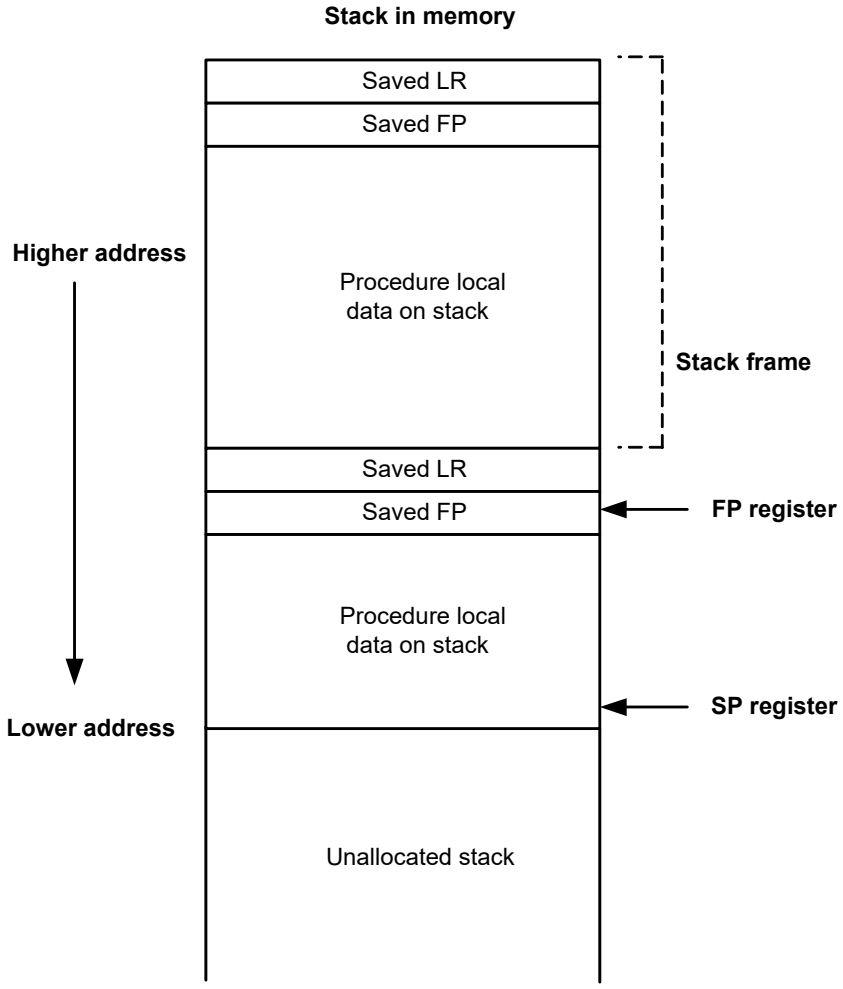
The Hexagon processor includes dedicated registers and instructions to support a call stack for subroutine execution.

The stack structure follows standard C conventions.

7.1 Stack structure

The stack is defined to grow from high addresses to low addresses. The stack pointer register SP points to the data element that is on the top of the stack.

Figure 7-1 Stack structure



NOTE: The Hexagon processor supports three dedicated stack instructions: allocframe, deallocframe, and dealloc_return (Section 7.5).

The SP address must always remain 8-byte aligned for the stack instructions to work properly.

7.2 Stack frames

The stack stores stack frames, which are data structures that store state information on the active subroutines in a program (for example, those that were called but have not yet returned). Each stack frame corresponds to an active subroutine in the program.

A stack frame contains the following elements:

- The local variables and data used by the subroutine
- The return address for the subroutine call (pushed from the link register LR)
- The address of the previous stack frame allocated on the stack (pushed from the frame pointer register FP)

The frame pointer register FP always contains the address of the saved frame pointer in the current stack frame. It facilitates debugging by enabling a debugger to examine the stack in memory and easily determine the call sequence, function parameters, and so on.

NOTE: For leaf functions, it is often unnecessary to save FP and LR. In this case, FP contains the frame pointer of the calling function, not the current function.

7.3 Stack protection

The Hexagon processor supports the following features to protect the integrity of the software stack.

7.3.1 Stack bounds checking

Stack bounds checking prevents a stack frame from being allocated past the lower boundary of the software stack.

FRAMELIMIT is a 32-bit control register that stores a memory address that specifies the lower bound of the memory area reserved for the software stack. When the allocframe instruction allocates a new stack frame, it compares the new stack pointer value in SP with the stack bound value in FRAMELIMIT. If SP is less than FRAMELIMIT, the Hexagon processor raises exception 0x27 (Section 8.10).

NOTE: Stack bounds checking is performed when the processor is in User and Guest modes but not in Monitor mode.

7.3.2 Stack smashing protection

Stack smashing is a technique malicious code uses to gain control over an executing program. Malicious code causes buffer overflows to occur in the local data of a procedure, with the goal of modifying the subroutine return address stored in a stack frame so it points to the malicious code instead of the intended return code.

Stack smashing protection prevents this from occurring by scrambling the subroutine return address when a new stack frame is allocated, and then unscrambling the return address when the frame is deallocated. Because the value in FRAMEKEY changes regularly and varies from device to device, it becomes difficult to precalculate a malicious return address.

FRAMEKEY is a 32-bit control register that scrambles return addresses stored on the stack:

- In the allocframe instruction, the 32-bit return address in link register LR is XOR-scrambled with the value in FRAMEKEY before it is stored in the new stack frame.
- In deallocframe and dealloc_return, the return address loaded from the stack frame is unscrambled with the value in FRAMEKEY before it is stored in LR.

After a processor reset, the default value of FRAMEKEY is 0. If this value is not changed, stack smashing protection is effectively disabled.

NOTE: Each hardware thread has its own instance of the FRAMEKEY register.

7.4 Stack registers

Table 7-1 Stack registers

Register	Name	Description	Alias
SP	Stack pointer	Points to topmost stack element in memory	R29
FP	Frame pointer	Points to previous stack frame on stack	R30
LR	Link register	Contains return address of subroutine call	R31
FRAMELIMIT	Frame limit register	Contains lowest address of stack area	C16
FRAMEKEY	Frame key register	Contains scrambling key for return addresses	C17

NOTE: SP, FP, and LR are aliases of three General registers. These general registers are conventionally dedicated for use as stack registers.

7.5 Stack instructions

The Hexagon processor includes the allocframe and deallocframe instructions to efficiently allocate and deallocate stack frames on the call stack.

Table 7-2 Stack instructions

Syntax	Operation
`allocframe(#u11:3)`	Allocate stack frame. This instruction is used after a call. It first XORs the values in LR and FRAMEKEY, and pushes the resulting scrambled return address and FP to the top of the stack. Next, it subtracts an unsigned immediate from SP to allocate room for local variables. If the resulting SP is less than FRAMELIMIT, the processor raises exception 0x27. Otherwise, SP is set to the new value, and FP is set to the address of the old frame pointer on the stack. The immediate operand as expressed in assembly syntax specifies the byte offset. This value must be 8-byte aligned. The valid range is from 0 to 16 KB.
`deallocframe`	Deallocate stack frame. Use this instruction before a return to free a stack frame. It first loads the saved FP and LR values from the address at FP, and XORs the restored LR with the value in FRAMEKEY to unscramble the return address. SP is pointed back to the previous frame.
`dealloc_return`	Subroutine return with stack frame deallocate. Perform the deallocframe operation, and then perform subroutine return (Section 8.3.3) to the target address loaded from LR by deallocframe.

NOTE: The allocframe and deallocframe instructions load and store the LR and FP registers on the stack as a single aligned 64-bit register pair (LR:FP).

8 Program flow

The Hexagon processor supports the following program flow facilities.

8.1 Conditional instructions

Many Hexagon processor instructions can conditionally execute. For example:

```
if (P0) R0 = memw(R2)     // Conditionally load word if P0
if (!P1) jump label       // Conditionally jump if not P1
```

The following instructions can be specified as conditional:

- Jumps and calls
- Many load and store instructions
- Logical instructions (including AND/OR/XOR)
- Shift halfword
- 32-bit add/subtract by register or short immediate
- Sign and zero extend
- 32-bit register transfer and 64-bit combine word
- Register transfer immediate
- Deallocate frame and return

For more information, see Section 5.9 and Chapter 6.

8.2 Hardware loops

The Hexagon processor includes hardware loop instructions that perform loop branches with zero overhead. For example:

```
  loop0(start,#3)           // loop 3 times
start:
    { R0 = mpyi(R0,R0) } :endloop0
```

Two sets of hardware loop instructions are provided – loop0 and loop1 – to nest hardware loops one level deep. For example:

```
// Sum the rows of a 100x200 matrix.

  loop1(outer_start,#100)
outer_start:
```

```
    R0 = #0
    loop0(inner_start,#200)
inner_start:
      R3 = memw(R1++#4)
      { R0 = add(R0,R3) }:endloop0
    { memw(R2++#4) = R0 }:endloop1
```

Use the hardware loop instructions as follows:

- For non-nested loops, loop0 is used.
- For nested loops, loop0 is used for the inner loop, and loop1 for the outer loop.

NOTE: If a program must create loops nested more than one level deep, the two innermost loops can be implemented as hardware loops, with the remaining outer loops implemented as software branches.

Each hardware loop is associated with a pair of dedicated loop registers:

- The loop start address register SAn is set to the address of the first instruction in the loop (which is typically expressed in assembly language as a label).
- The loop count register LCn is set to a 32-bit unsigned value that specifies the number of loop iterations to perform. When the PC reaches the end of the loop, LCn is examined to determine whether to repeat or exit the loop.

The hardware loop setup instruction sets both of these registers at the same time – typically there is no need to set them individually. However, because the loop registers completely specify the hardware loop state, saving and restoring the registers (either automatically by a processor interrupt or manually by the programmer) enables a suspended hardware loop to resume normally when its loop registers are reloaded with the saved values.

The Hexagon processor provides two sets of loop registers for the two hardware loops:

- SA0 and LC0 are used by loop0
- SA1 and LC1 are used by loop1

Table 8-1 Hardware loop instructions

Syntax	Description
`loopN(start, Rs)`	Hardware loop with register loop count. Set registers SAn and LCn for hardware loop N: ■ SAn is assigned the specified start address of the loop. ■ LCn is assigned the value of general register Rs. **NOTE:** The loop start operand is encoded as a PC-relative immediate value.
`loopN(start, #count)`	Hardware loop with immediate loop count. Set registers SAn and LCn for hardware loop N: ■ SAn is assigned the specified start address of the loop. ■ LCn is assigned the specified immediate value (0 to 1023). **NOTE:** The loop start operand is encoded as a PC-relative immediate value.
`:endloopN`	Hardware loop end instruction. Performs the following operation: `if (LCn > 1) {PC = SAn; LCn = LCn-1}` **NOTE:** This instruction appears in assembly as a suffix appended to the last packet in the loop. It is encoded in the last packet.
`SAn = Rs`	Set loop start address to general register Rs
`LCn = Rs`	Set loop count to general register Rs

NOTE: The loop instructions are assigned to instruction class CR.

8.2.1 Loop setup

To set up a hardware loop, the loop registers SAn and LCn must be set to the proper values. This is done in two ways:

- A loopN instruction
- Register transfers to SAn and LCn

The loopN instruction performs all the work of setting SAn and LCn. For example:

```
  loop0(start,#3)           // SA0=&start, LC0=3
start:
    { R0 = mpyi(R0,R0) } :endloop0
```

In this example, the hardware loop (consisting of a single multiply instruction) executes three times. The loop0 instruction sets register SA0 to the address value of label start, and LC0 to 3.

Loop counts are limited to the range 0 to1023 when they are expressed as immediate values in loopN. If the intended loop count exceeds this range, it must be specified as a register value. For example:

Using loopN:

```
  R1 = #20000;
  loop0(start,R1)          // LC0=20000, SA0=&start
```

```
start:
    { R0 = mpyi(R0,R0) } :endloop0
```

Using register transfers:

```
  R1 = #20000
  LC0 = R1                  // LC0=20000
  R1 = #start
  SA0 = R1                  // SA0=&start
start:
    { R0 = mpyi(R0,R0) } :endloop0
```

If a loopN instruction is located too far from its loop start address, the PC-relative offset value that specifies the start address can exceed the maximum range of the start address operand of the instruction . If this occurs, either move the loopN instruction closer to the loop start, or specify the loop start address as a 32-bit constant (Section 10.9).

For example, using 32-bit constants:

```
  R1 = #20000;
  loop0(##start,R1)         // LC0=20000, SA0=&start
  ...
```

8.2.2 Loop end

The loop end instruction indicates the last packet in a hardware loop. It is expressed in assembly language by appending the packet with the symbol `:endloopN`, where `N` specifies the hardware loop (0 or 1). For example:

```
  loop0(start,#3)
start:
    { R0 = mpyi(R0,R0) } :endloop0  // last packet in loop
```

The last instruction in the loop must always be expressed in assembly language as a packet (using curly braces), even if it is the only instruction in the packet.

Nested hardware loops can specify the same instruction as the end of both the inner and outer loops. For example:

```
// Sum the rows of a 100x200 matrix.
// Software pipeline the outer loop.

     p0 = cmp.gt(R0,R0)              // p0 = false
     loop1(outer_start,#100)
outer_start:
     { if (p0) memw(R2++#4) = R0
       p0 = cmp.eq(R0,R0)            // p0 = true
       R0 = #0
       loop0(inner_start,#200) }
inner_start:
     R3 = memw(R1++#4)
     { R0 = add(R0,R3) }:endloop0:endloop1
     memw(R2++#4) = R0
```

Though `endloopN` behaves like a regular instruction (by implementing the loop test and branch), it does not execute in any instruction slot, and does not count as an instruction in the packet. Therefore a single instruction packet that is marked as a loop end can perform up to six operations:

- Four regular instructions (the normal limit for an instruction packet)
- The `endloop0` test and branch
- The `endloop1` test and branch

NOTE: The `endloopN` instruction is encoded in the instruction packet (Section 10.6).

8.2.3 Loop execution

After a hardware loop is set up, the loop body always executes at least one time regardless of the specified loop count (because the loop count is not examined until the last instruction in the loop). Therefore, if a loop must optionally execute zero times, it must be preceded with an explicit conditional branch. For example:

```
  loop0(start,R1)
  P0 = cmp.eq(R1,#0)
  if (P0) jump skip
start:
    { R0 = mpyi(R0,R0) } :endloop0
skip:
```

In this example, a hardware loop is set up with the loop count in R1, but if the value in R1 is zero a software branch skips over the loop body.

After the loop end instruction of a hardware loop executes, the Hexagon processor examines the value in the corresponding loop count register:

- If the value is greater than 1, the processor decrements the loop count register and performs a zero-cycle branch to the loop start address.
- If the value is less than or equal to 1, the processor resumes program execution at the instruction immediately following the loop end instruction.

NOTE: Because nested hardware loops can share loop end instruction, the processor can examine both loop count registers in a single operation.

8.2.4 Pipelined hardware loops

Software pipelined loops are common for VLIW architectures such as the Hexagon processor. They offer increased code performance in loops by overlapping multiple loop iterations.

A software pipeline has three sections:

- A prologue in which the loop is primed
- A kernel (or steady-state) portion
- An epilogue that drains the pipeline

Table 8-2 shows an example where the kernel section of the pipelined loop performs three iterations of the loop in parallel:

- The load for iteration N+2
- The multiply for iteration N+1
- The store for iteration N

Table 8-2 Software pipelined loop

```
int foo(int *A, int *result)
{
    int i;
    for (i=0;i<100;i++) {
        result[i]= A[i]*A[i];
    }
}
```

```
foo:
{       R3 = R1
        loop0(.kernel,#98)          // Decrease loop count by 2
}
        R1 = memw(R0++#4)           // First prologue stage
{       R1 = memw(R0++#4)           // Second prologue stage
        R2 = mpyi(R1,R1)
}
        .falign
.kernel:
{       R1 = memw(R0++#4)           // Kernel
        R2 = mpyi(R1,R1)
        memw(R3++#4) = R2
}:endloop0
{       R2 = mpyi(R1,R1)            // First epilogue stage
        memw(R3++#4) = R2
}
        memw(R3++#4) = R2           // Second epilogue stage
        jumpr lr
```

A drawback to software pipelining is the extra code necessary for the prologue and epilogue sections of a pipelined loop.

To address this issue, the Hexagon processor provides the spNloop0 instruction, where the "N" in the instruction name indicates a digit in the range 1 to 3. For example:

```
P3 = sp2loop0(start,#10)      // Set up pipelined loop
```

The spNloop0 instruction is a variant of the loop0 instruction: it sets up a normal hardware loop using SA0 and LC0, but also performs the following additional operations:

- When the spNloop0 instruction executes, it assigns the truth value false to the predicate register P3.
- After the associated loop executes N times, P3 is automatically set to true.

This feature (known as automatic predicate control) enables the store instructions in the kernel section of a pipelined loop to conditionally execute by P3 and thus – because of the way spNloop0 controls P3 – not execute during the pipeline warm-up. This can reduce the code size of software pipelined loops by eliminating the need for prologue code.

The spNloop0 instruction cannot be used to eliminate the epilogue code from a pipelined loop; however, in some cases it is possible to do this through the use of programming techniques.

Typically, the issue affecting the removal of epilogue code is load safety. If the kernel section of a pipelined loop can safely access past the end of its arrays – either because it is known as safe, or because the arrays have been padded at the end – epilogue code is unnecessary. However, if load safety cannot be ensured, explicit epilogue code is required to drain the software pipeline.

Table 8-3 shows how spNloop0 and load safety simplify the code shown in Table 8-2.

Table 8-3 Software pipelined loop (using spNloop0)

```
int foo(int *A, int *result)
{
    int i;
    for (i=0;i<100;i++) {
        result[i]= A[i]*A[i];
    }
}
```

```
foo:
{ // load safety assumed
        P3 = sp2loop0(.kernel,#102)     // Set up pipelined loop
        R3 = R1
}
.falign
.kernel:
{       R1 = memw(R0++#4)               // Kernel
        R2 = mpyi(R1,R1)
        if (P3) memw(R3++#4) = R2
}:endloop0

        jumpr lr
```

NOTE: The count value that `spNloop0` uses to control the P3 setting is stored in the user status register USR.LPCFG.

8.2.5 Loop restrictions

Hardware loops have the following restrictions:

- The loop setup packet in loopN or spNloop0 (Section 8.2.4) cannot contain a speculative indirect jump, new-value compare jump, or dealloc_return.
- The last packet in a hardware loop cannot contain any program flow instructions (including jumps or calls).
- The loop end packet in loop0 cannot contain any instruction that changes SA0 or LC0. Similarly, the loop end packet in loop1 cannot contain any instruction that changes SA1 or LC1.
- The loop end packet in spNloop0 cannot contain any instruction that changes P3.

NOTE: SA1 and LC1 can be changed at the end of loop0 while SA0 and LC0 can be changed at the end of loop1.

8.3 Software branches

Unlike hardware loops, software branches use an explicit instruction to perform a branch operation. Software branches include jumps, calls, and returns.

The target address for branch instructions is specified as register indirect or PC-relative offsets. PC-relative offsets are normally less than 32 bits, but can be specified as 32 bits by using the appropriate syntax in the target operand (Section 8.3.4).

Branch instructions are unconditional or conditional, with the execution of conditional instructions controlled by a predicate expression.

Table 8-4 Software branch instructions

Syntax	Operation
`[if (pred_expr)] jump label` `[if (pred_expr)] jumpr Rs`	Branch to address specified by register Rs or PC-relative offset. Can be conditionally executed.
`[if (pred_expr)] call label` `[if (pred_expr)] callr Rs`	Branch to address specified by register Rs or PC-relative offset. Store subroutine return address in link register LR. Can be conditionally executed.
`[if (pred_expr)] jumpr LR`	Branch to subroutine return address contained in link register LR. Can be conditionally executed.

8.3.1 Jumps

Jump instructions change the program flow to a target address, which are specified by either a register or a PC-relative immediate value. Jump instructions can be conditional based on the value of a predicate expression.

Table 8-5 Jump instructions

Syntax	Operation
`jump label`	Direct jump. Branch to address specified by label. Label is encoded as PC-relative signed immediate value.
`jumpr Rs`	Indirect jump. Branch to address contained in general register Rs.
`if ([!]Ps) jump label` `if ([!]Ps) jumpr Rs`	Conditional jump. Perform jump if predicate expression evaluates to true.

NOTE: Conditional jumps can be specified as speculative (Section 8.4).

8.3.2 Calls

Call instructions jump to subroutines. The instruction performs a jump to the target address and also stores the return address in the link register LR.

The forms of call are functionally similar to jump instructions and include both PC-relative and register indirect in both unconditional and conditional forms.

Table 8-6 Call instructions

Syntax	Operation
`call label`	Direct subroutine call. Branch to address specified by label, and store return address in register LR. Label is encoded as PC-relative signed immediate value.
`callr Rs`	Indirect subroutine call. Branch to address contained in general register Rs, and store return address in register LR.
`if ([!]Ps) call label` `if ([!]Ps) callr Rs`	Conditional call. If predicate expression evaluates to true, perform subroutine call to specified target address.

8.3.3 Returns

Return instructions return from a subroutine. The instruction performs an indirect jump to the subroutine return address stored in link register LR.

Returns are implemented as jump register indirect instructions, and support both unconditional and conditional forms.

Table 8-7 Return instructions

Syntax	Operation
`jumpr LR`	Subroutine return. Branch to subroutine return address contained in link register LR.
`if ([!]Ps) jumpr LR`	Conditional subroutine return. If predicate expression evaluates to true, perform subroutine return to specified target address.
`dealloc_return`	Subroutine return with stack frame deallocate. Perform deallocframe operation (Section 7.5) and then perform subroutine return to the target address loaded by deallocframe from the link register.
`if ([!]Ps) dealloc_return`	Conditional subroutine return with stack frame deallocate. If predicate expression evaluates to true, perform deallocframe and subroutine return to the target address loaded by deallocframe from the link register.

NOTE: The link register LR is an alias of general register R31. Therefore subroutine returns can be performed with the instruction jumpr R31.

The conditional subroutine returns (including dealloc_return) can be specified as speculative (Section 8.4).

8.3.4 Extended branches

When a jump or call instruction specifies a PC-relative offset as the branch target, the offset value is normally encoded in significantly less than 32 bits. This can limit the ability for programs to specify "long" branches, which span a large range of the memory address space of the processor.

To support long branches, the jump and call instructions have special versions that encode a full 32-bit value as the PC-relative offset.

NOTE: Such instructions use an extra word to store the 32-bit offset (Section 10.9).

The size of a PC-relative branch offset is expressed in assembly language by optionally prefixing the target label with the symbol "##" or "#":

- "##" specifies that the assembler *must* use a 32-bit offset
- "#" specifies that the assembler must *not* use a 32-bit offset
- No "#" specifies that the assembler use a 32-bit offset only if necessary

For example:

```
jump ##label   // 32-bit offset
call #label    // Non 32-bit offset
jump label     // Offset size determined by assembler
```

8.3.5 Branches to and from packets

Instruction packets are atomic: even if they contain multiple instructions, they are referenced only by the address of the first instruction in the packet. Therefore, branches to a packet can target only the first instruction of the packet.

Packets can contain up to two branches (Section 8.7). The branch destination can target the current packet or the beginning of another packet.

A branch does not interrupt the execution of the current packet: all of the instructions in the packet execute, even if they appear in the assembly source after the branch instruction.

If a packet is at the end of a hardware loop, it cannot contain a branch instruction.

8.4 Speculative jumps

Conditional instructions normally depend on predicates that are generated in a previous instruction packet. However, Dot-new predicates enable conditional instructions to use a predicate generated in the same packet that contains the conditional instruction.

When dot-new predicates are used with a conditional jump, the resulting instruction is called a speculative jump. For example:

```
{
   P0 = cmp.eq(R9,#16)         // single-packet compare-and-jump
   IF (P0.new) jumpr:t R11     // ... enabled by use of P0.new
}
```

Speculative jumps require the programmer to specify a direction hint in the jump instruction, indicating whether the conditional jump is expected.

The hint initializes the dynamic branch predictor of the Hexagon processor. Whenever the predictor is wrong, the speculative jump instruction takes two cycles to execute instead of one (due to a pipeline stall).

Hints can improve program performance by indicating how speculative jumps are expected to execute over the course of a program: the more often the specified hint indicates how the instruction actually executes, the better the performance.

Hints are expressed in assembly language by appending the suffix "`:t`" or "`:nt`" to the jump instruction symbol. For example:

- jump:t – The jump instruction is most often taken
- jump:nt – The jump instruction is most often not taken

In addition to dot-new predicates, speculative jumps also accept conditional arithmetic expressions (=0, !=0, >=0, <=0) involving the general register Rs.

Table 8-8 Speculative jump instructions

Syntax	Operation
`if ([!]Ps.new) jump:t label` `if ([!]Ps.new) jump:nt label`	Speculative direct jump. If predicate expression evaluates to true, jump to address specified by label.
`if ([!]Ps.new) jumpr:t Rs` `if ([!]Ps.new) jumpr:nt Rs`	Speculative indirect jump. If predicate expression evaluates to true, jump to address in register Rs.
`if (Rs == #0) jump:t label` `if (Rs == #0) jump:nt label`	Speculative direct jump. If predicate Rs = 0 is true, jump to address specified by label.
`if (Rs != #0) jump:t label` `if (Rs != #0) jump:nt label`	Speculative direct jump. If predicate Rs != 0 is true, jump to address specified by label.
`if (Rs >= #0) jump:t label` `if (Rs >= #0) jump:nt label`	Speculative direct jump. If predicate Rs >= 0 is true, jump to address specified by label.
`if (Rs <= #0) jump:t label` `if (Rs <= #0) jump:nt label`	Speculative direct jump. If predicate Rs <= 0 is true, jump to address specified by label.

NOTE: The hints `:t` and `:nt` interact with the predicate value to determine the instruction cycle count.

Speculative indirect jumps are not supported with register Rs predicates.

8.5 Compare jumps

To reduce code size, the Hexagon processor supports a compound instruction that combines a compare with a speculative jump in a single 32-bit instruction.

For example:

```
{
   p0 = cmp.eq (R2,R5)           // Single-instr compare-and-jump
   if (p0.new) jump:nt target    // Enabled by compound instr
}
```

The register operands used in a compare jump are limited to R0 through R7 or R16 through R23 (Table 10-3).

The compare and jump instructions that are used in a compare jump are limited to the instructions listed in Table 8-9. The compare can use predicate P0 or P1, while the jump must specify the same predicate that is set in the compare.

A compare jump instruction is expressed in assembly source as two independent compare and jump instructions in a packet. The assembler translates the two instructions into a single compound instruction.

Table 8-9 Compare jump instructions

Compare Instruction	Jump Instruction
`Pd = cmp.eq (Rs, Rt)`	`IF (Pd.new) jump:t label`
`Pd = cmp.gt (Rs, Rt)`	`IF (Pd.new) jump:nt label`
`Pd = cmp.gtu (Rs, Rt)`	`IF (!Pd.new) jump:t label`
`Pd = cmp.eq (Rs,#U5)`	`IF (!Pd.new) jump:nt label`
`Pd = cmp.gt (Rs,#U5)`	
`Pd = cmp.gtu (Rs,#U5)`	
`Pd = cmp.eq (Rs,#-1)`	
`Pd = cmp.gt (Rs,#-1)`	
`Pd = tstbit (Rs, #0)`	

8.5.1 New-value compare jumps

A compare jump instruction can access a register that is assigned a new value in the same instruction packet (Section 3.3). This feature is expressed in assembly language by the following changes:

- Appending the suffix "`.new`" to the new-value register in the compare
- Rewriting the compare jump so its constituent compare and jump operations appear as a single conditional instruction

For example:

```
// load-compare-and-jump packet enabled by new-value compare jump
```

```
{
   R0 = memw(R2+#8)
   if (cmp.eq(R0.new,#0)) jump:nt target
}
```

New-value compare jump instructions have the following restrictions:

- They are limited to the instruction forms listed in Table 8-10.
- They cannot be combined with another jump instruction in the same packet.
- If an instruction produces a 64-bit result or performs a floating-point operation, its result registers cannot be used as the new-value register.
- If an instruction uses auto-increment or absolute-set addressing mode (Section 5.8), its address register cannot be used as the new-value register.
- If the instruction that sets a new-value register is conditional (Section 6.1.2), it must always execute.

If the specified jump direction hint is wrong (Section 8.4), a new-value compare jump takes three cycles to execute instead of one. While this penalty is one cycle longer than in a regular speculative jump, the overall performance is still better than using a regular speculative jump (which must execute an extra packet in all cases).

NOTE: New-value compare jump instructions are assigned to instruction class NV, which can execute only in Slot 0. The instruction that assigns the new value must execute in Slot 1, 2, or 3.

Table 8-10 New-value compare jump instructions

`if ([!]cmp.eq (Rs.new, Rt)) jump:[hint] label`
`if ([!]cmp.gt (Rs.new, Rt)) jump:[hint] label`
`if ([!]cmp.gtu (Rs.new, Rt)) jump:[hint] label`
 	
`if ([!]cmp.gt (Rs, Rt.new)) jump:[hint] label`
`if ([!]cmp.gtu (Rs, Rt.new)) jump:[hint] label`

`if ([!]cmp.eq (Rs.new, #u5)) jump:[hint] label`
`if ([!]cmp.gt (Rs.new, #u5)) jump:[hint] label`
`if ([!]cmp.gtu (Rs.new ,#u5)) jump:[hint] label`
 	
`if ([!]cmp.eq (Rs.new, #-1)) jump:[hint] label`
`if ([!]cmp.gt (Rs.new, #-1)) jump:[hint] label`
 	
`if ([!]tstbit (Rs.new, #0)) jump:[hint] label`

8.6 Register transfer jumps

To reduce code size, the Hexagon processor supports a compound instruction that combines a register transfer with an unconditional jump in a single 32-bit instruction.

For example:

```
{
   jump target     // Jump to label "target"
   R1 = R2         // Assign contents of reg R2 to R1
}
```

The source and target register operands in the register transfer are limited to R0 through R7 or R16 through R23 (Table 10-3).

The target address in the jump is a scaled 9-bit PC-relative address value (as opposed to the 22-bit value in the regular unconditional jump instruction).

A register transfer jump instruction is expressed in assembly source as two independent instructions in a packet. The assembler translates the instructions into a single compound instruction.

Table 8-11 Register transfer jump instructions

Syntax	Operation
`jump label;` `Rd=Rs`	Register transfer jump. Perform register transfer and branch to address specified by label. Label is encoded as PC-relative 9-bit signed immediate value.
`jump label;` `Rd=#u6`	Register transfer immediate jump. Perform register transfer (of 6-bit unsigned immediate value) and branch to address specified by label. Label is encoded as PC-relative 9-bit signed immediate value.

8.7 Dual jumps

Two software branch instructions (referred to here as "jumps") can appear in the same instruction packet, under the conditions listed in Table 8-12.

The first jump is defined as the jump instruction at the lower address, and the second jump as the jump instruction at the higher address.

Unlike most packetized operations, dual jumps are not executed in parallel (Section 3.3.1). Instead, the two jumps are processed in a well-defined order in a packet:

1. The predicate in the first jump is evaluated.
2. If the first jump is taken, the second jump is ignored.
3. If the first jump is not taken, the second jump is performed.

Table 8-12 Dual jump instructions

Instruction	Description	First jump in packet?	Second jump in packet?
`jump`	Direct jump	No	Yes
`if ([!]Ps[.new]) jump`	Conditional jump	Yes	Yes
`call` `if ([!]Ps) call`	Direct calls	No	Yes
`Pd=cmp.xx ; if ([!]Pd.new) jump`	Compare jump	Yes	Yes
`if ([!]cmp.xx(Rs.new, Rt)) jump`	New-value compare jump	No	No

`jumpr`
`if ([!]Ps[.new]) jumpr`
`callr`
`if ([!]Ps) callr`
`dealloc_return`
`if ([!]Ps[.new]) dealloc_return`
	Indirect jumps
	Indirect calls
	dealloc_return
		No	No

`endloopN`	Hardware loop end	No	No

NOTE: If a call is ignored in a dual jump, the link register LR is not changed.

8.8 Hint indirect jump target

Because it obtains the jump target address from a register, the jumpr instruction (Section 8.3.1) normally causes the processor to stall for one cycle.

To avoid the stall penalty caused by a jumpr instruction, the Hexagon processor supports the hintjr jump hint instruction, which can be specified before the jumpr instruction.

The hintjr instruction indicates that the program is about to execute a jumpr to the address contained in the specified register.

Table 8-13 Speculative jump hint instruction

Syntax	Operation
`hintjr(Rs)`	Informs the processor that the jumpr (Rs) instruction is about to be performed.

NOTE: To prevent a stall, the hintjr instruction must execute at least 2 packets before the corresponding jumpr instruction.

The hintjr instruction is not needed for jumpr instructions used as returns (Section 8.3.3), because in this case the Hexagon processor automatically predicts the jump targets based on the most recent nested call instructions.

8.9 Pauses

Pauses suspend the execution of a program for a period of time, and put it into low-power mode. The program remains suspended for the duration specified in the instruction.

The pause instruction accepts an unsigned 8-bit immediate operand that specifies the pause duration in terms of cycles. The maximum possible duration is 263 cycles (255+8).

Hexagon processor interrupts cause a program to exit the paused state before its specified duration has elapsed.

The pause instruction is useful for implementing user-level low-power synchronization operations (such as spin locks).

Table 8-14 Pause instruction

Syntax	Operation
`pause(#u8)`	Suspend program in low-power mode for specified cycle duration.

8.10 Exceptions

Exceptions are internally-generated disruptions to the program flow.

The Hexagon processor OS handles fatal exceptions by terminating the execution of the application system. The user is responsible for fixing the problem and recompiling their applications.

The error messages generated by exceptions include the following information to help locate the problem:

- Cause code: Hexadecimal value indicating the type of exception
- User IP: PC value indicating the instruction executed when the exception occurred
- Bad VA: Virtual address indicating the data accessed when the exception occurred

NOTE: The cause code, user IP, and Bad VA values are stored in the Hexagon processor system control registers SSR[7:0], ELR, and BADVA respectively.

If multiple exceptions occur simultaneously, the exception with the lowest error code value has the highest exception priority.

If a packet contains multiple loads, or a load and a store, and both operations have an exception of any type, all slot 1 exceptions process before any slot 0 exception is processed.

Table 8-15 V75 exceptions

Cause code	Event type	Event description	Notes
0x0	Reset	Software thread reset.	Non-maskable, highest priority
0x01	Precise, unrecoverable	Unrecoverable BIU error (bus error, timeout, L2 parity error, and so on).	Non-maskable
0x03	Precise, unrecoverable	Double exception (exception occurs while SSR[EX]=1).	Non-maskable
0x11	Precise	Privilege violation: User/Guest mode execute to page with no execute permissions (X=0).	Non-maskable
0x12	Precise	Privilege violation: User mode execute to a page with no user permissions (X=1, U=0).	Non-maskable
0x15	Precise	Invalid packet.	Non-maskable
0x16	Precise	Illegal execution of coprocessor instruction.	Non-maskable
0x17	Precise	Instruction cache error.	Non-maskable
0x1A	Precise	Privilege violation: Executing a guest mode instruction in user mode.	Non-maskable
0x1B	Precise	Privilege violation: Executing a supervisor instruction in User/Guest mode.	Non-maskable
0x1D	Precise, unrecoverable	Packet with multiple writes to the same destination register.	Non-maskable
0x1E	Precise, unrecoverable	Program counter values that are not properly aligned.	Non-maskable
0x20	Precise	Load to misaligned address.	Non-maskable
0x21	Precise	Store to misaligned address.	Non-maskable
0x22	Precise	Privilege violation: User/Guest mode read to page with no read permission (R=0).	Non-maskable
0x23	Precise	Privilege violation: User/Guest mode write to page with no write permissions (W=0).	Non-maskable
0x24	Precise	Privilege violation: User mode read to page with no user permission (R=1, U=0).	Non-maskable
0x25	Precise	Privilege violation: User mode write to page with no user permissions (W=1, U=0).	Non-maskable
0x26	Precise	Coprocessor VMEM address error.	Non-maskable
0x27	Precise	Stack overflow: Allocframe instruction exceeded FRAMELIMIT.	Non-maskable,

Table 8-15 V75 exceptions (cont.)

Cause code	Event type	Event description	Notes
0x42	Imprecise	Data abort.	Non-maskable
0x43	Imprecise	NMI	Non-maskable
0x44	Imprecise	Multiple TLB match.	Non-maskable
0x45	Imprecise	Livelock exception.	Non-maskable
0x60	TLB miss-X	Missing fetch address on PC-page.	Non-maskable
0x61	TLB miss-X	Missing fetch on second page from packet that spans pages.	Non-maskable
0x62	TLB miss-X	icinva.	Non-maskable
	Reserved		
0x70	TLB miss-RW	Memory read.	Non-maskable
0x71	TLB miss-RW	Memory write.	Non-maskable
	Reserved		
#u8	Trap0	Software Trap0 instruction.	Non-maskable
#u8	Trap1	Software Trap1 instruction.	Non-maskable
	Reserved		
0x80	Debug	Single-step debug exception.	
	Reserved		
0xBF	Floating-point	Execution of a floating-point instruction resulted in an exception.	Non-maskable
0xC0	Interrupt0	General external interrupt.	Maskable, highest priority general interrupt
0xC1	Interrupt 1	General external interrupt	Maskable
0xC2	Interrupt 2	General external interrupt	VIC0 interface
0xC3	Interrupt 3	General external interrupt	VIC1 interface
0xC4	Interrupt 4	General external interrupt	VIC2 interface
0xC5	Interrupt 5	General external interrupt	VIC3 interface
0xC6	Interrupt 6	General external interrupt	
0xC7	Interrupt 7	General external interrupt	Lowest-priority interrupt

9 PMU events

The Hexagon processor can collect execution statistics on the applications it executes. The statistics summarize the types of Hexagon processor events that occur while the application runs.

Execution statistics are collected in hardware or software:

- Statistics are collected in hardware with the performance monitor unit (PMU), which is defined as part of the Hexagon processor architecture.
- Statistics are collected in software that uses the Hexagon simulator. The simulator statistics are presented in the same format used by the PMU.

Execution statistics are expressed in terms of processor events. This chapter defines the event symbols, along with their associated numeric codes.

NOTE: Because the types of execution events vary across the Hexagon processor versions, different types of statistics are collected for each version. This chapter lists the event symbols defined for version V75.

9.1 V75 processor event symbols

Table 9-1 defines the symbols that represent processor events for the V75 Hexagon processor.

Table 9-1 V75 processor event symbols

Event	Symbol	Definition	Maskable
0x1	COUNTER0_OVERFLOW	Can be used as the event detected by counter1 to build an effective 64-bit counter.	Maskable
0x2	COUNTER2_OVERFLOW	Can be used as the event detected by counter3 to build an effective 64-bit counter.	Maskable
0x3	COMMITTED_PKT_ANY	Number of packets that are committed by any thread. Packets are executed.	Maskable
0x4	COMMITTED_PKT_BSB	Number of packets that are committed two cycles after an earlier packet in the same thread.	Not maskable
0x5	COUNTER4_OVERFLOW	Can be used as the event detected by counter5 to build an effective 64-bit counter.	Maskable
0x6	COUNTER6_OVERFLOW	Can be used as the event detected by counter7 to build an effective 64-bit counter.	Maskable
0x7	COMMITTED_PKT_B2B	Number of packets that are committed one cycle after the earlier packet in the same thread.	Not maskable
0x8	COMMITTED_PKT_SMT	Number of packets that are committed on the SMT threads. Includes the second, third, and fourth packets that are committed in one cycle.	Not maskable

Table 9-1 V75 processor event symbols

Event	Symbol	Definition	Maskable
0xa	CYCLES_5_THREAD_RUNNING	Processor cycles that exactly five threads are running. Running means the threads are not in the Wait or Stop state.	Not maskable
0xb	CYCLES_6_THREAD_RUNNING	Processor cycles that exactly six threads are running. Running means the threads are not in the Wait or Stop state.	Not maskable
0xc	COMMITTED_PKT_T0	Number of packets that are committed by thread 0. Packets are executed.	Not maskable
0xd	COMMITTED_PKT_T1	Number of packets that are committed by thread 1. Packets are executed.	Not maskable
0xe	COMMITTED_PKT_T2	Number of packets that are committed by thread 2. Packets are executed.	Not maskable
0xf	COMMITTED_PKT_T3	Number of packets that are committed by thread 3. Packets are executed.	Not maskable
0x10	COMMITTED_PKT_T4	Number of packets that are committed by thread 4. Packets are executed.	Not maskable
0x11	COMMITTED_PKT_T5	Number of packets that are committed by thread 5. Packets are executed.	Not maskable
0x12	ICACHE_DEMAND_MISS	Number of I-cache cacheable demand primary or secondary misses. Includes secondary misses.	Not maskable
0x13	DCACHE_DEMAND_MISS	Number of D-cache cacheable demand primary or secondary misses. Includes dczero stalls. Excludes uncacheables, prefetches, and no-allocate store misses.	Not maskable
0x14	DCACHE_STORE_MISS	Number of D-cache cacheable store misses.	Maskable
0x15	COMMITTED_PKT_T6	Thread 6 committed a packet. Packets are executed.	Not maskable
0x16	COMMITTED_PKT_T7	Thread 7 committed a packet. Packets are executed.	Not maskable
0x17	CU_PKT_READY_NOT_DISPATCHED	Packets were ready at the CU scheduler but were not scheduled because either the scheduler's thread was not picked or there was an inter-cluster resource conflict.	Not maskable
0x18	COMMITTED_PKT_5_THREAD_-RUNNING	Number of committed packets with five threads running. Running means the threads are not in Wait or Stop mode.	Maskable
0x19	COMMITTED_PKT_6_THREAD_-RUNNING	Number of committed packets with six threads running. Running means the threads are not in Wait or Stop mode.	Maskable
0x1a	COMMITTED_PKT_7_THREAD_-RUNNING	Number of committed packets with seven threads running. Running means the threads are not in Wait or Stop mode.	Maskable
0x1b	COMMITTED_PKT_8_THREAD_-RUNNING	Number of committed packets with eight threads running. Running means the threads are not in Wait or Stop mode.	Maskable

Table 9-1 V75 processor event symbols

Event	Symbol	Definition	Maskable
0x1c	IU_L1S_ACCESS	Number of IU L1S loads. Includes demands or prefetches.	Not maskable
0x1d	IU_L1S_PREFETCH	Number of IU L1S prefetches.	Not maskable
0x1e	IU_L1S_AXIS_STALL	Number of IU L1S stalls due to an AXI slave.	Not maskable
0x1f	IU_L1S_NO_GRANT	IU request to L1S, and no grant from the vector unit.	Not maskable
0x20	ANY_IU_REPLAY	Any IU stall other than an I-cache miss. Includes a jump register stall, fetchcross stall, ITLB miss stall, and so on. Excludes a CU replay.	Not maskable
0x21	ANY_DU_REPLAY	Any DU replay. Includes a bank conflict, store buffer full, and so on. Excludes a stall due to a cache miss.	Not maskable
0x22	CYCLES_7_THREAD_RUNNING	Processor cycles that exactly seven threads are running. Running means the threads are not in Wait or Stop mode.	Not maskable
0x23	ISSUED_PACKETS	Speculatively issued packets were delivered from an IU.	Not maskable
0x24	LOOPCACHE_PACKETS	Committed packets were cloned from the packet queue during a pinned hardware loop.	Maskable
0x25	COMMITTED_PKT_1_THREAD_-RUNNING	Number of committed packets with one thread running. Running means the thread is not in Wait or Stop mode.	Maskable
0x26	COMMITTED_PKT_2_THREAD_-RUNNING	Number of committed packets with two threads running. Running means the threads are not in Wait or Stop mode.	Maskable
0x27	COMMITTED_PKT_3_THREAD_-RUNNING	Number of committed packets with three threads running. Running means the threads are not in Wait or Stop mode.	Maskable
0x28	THREAD_LMH_THROTTLE	For a specific thread, the sustained power exceeds the limits management threshold and limits budget threshold. Results in throttling that is based on thread priority.	Not maskable
0x29	LMH_THROTTLE	Throttling is based on the value of the peak current that is over the current limits of LMH.	Not maskable
0x2a	COMMITTED_INSTS	Number of committed instructions. Increments by up to eight per cycle. Duplex of two instructions counts as two instructions. Does not include end loops.	Maskable
0x2b	COMMITTED_TC1_INSTS	Number of committed TC1 class instructions. Increments by up to eight per cycle. Duplex of two TC1 instructions counts as two separate TC1 instructions. Does not include NOPs.	Maskable

Table 9-1 V75 processor event symbols

Event	Symbol	Definition	Maskable
0x2c	COMMITTED_PRIVATE_INSTS	Number of committed instructions that have per-cluster (private) execution resources. Increments by up to eight per cycle. Duplex of two private instructions counts as two private instructions.	Maskable
0x2d	GLOBAL_POWERLIMITS_OVER	Sustained global power that exceeds the overall global limits management threshold and limits the budget threshold. Causes the thread-specific LMH to engage.	Not maskable
0x2e	CYCLES_8_THREAD_RUNNING	Processor cycles that exactly eight threads are running. Running means the threads are not in Wait or Stop mode.	Not maskable
0x2f	COMMITTED_PKT_4_THREAD_-RUNNING	Number of committed packets with four threads running. Running means the threads are not in Stop or Wait mode.	Maskable
0x30	COMMITTED_LOADS	Number of committed load instructions. Includes cached and uncached. Increments by two for dual loads. Excludes prefetches, memory operations, and coprocessor loads.	Maskable
0x31	COMMITTED_STORES	Number of committed store instructions. Includes cached and uncached. Increments by two for dual stores. Excludes memory operations and coprocessor stores.	Maskable
0x32	COMMITTED_MEMOPS	Number of committed memory operations instructions. Cached or uncached.	Maskable
0x33	COMMITTED_NOPS	Number of committed NOPs.	Maskable
0x34	ISSUED_INSTS	Speculatively issued instructions delivered from the IU.	Maskable
0x35	DISPATCHED_PACKETS	Number of packets that the CU dispatched. NOPs instructions are squashed.	Not maskable
0x36	DISPATCHED_INSTS	Number of instructions that the CU dispatched.	Maskable
0x37	COMMITTED_PROGRAM_FLOW_I-NSTS	Number of committed packets that contain a program flow instruction. Includes CR jumps, endloop, J, JR, dealloc_return, system/trap, superset of event 56. Dual jumps count as two jumps.	Maskable
0x38	COMMITTED_PKT_CHANGED_FLOW	Number of committed packets that resulted in a change of flow. Any taken jump. Includes endloop and dealloc_return.	Maskable
0x39	COMMITTED_PKT_ENDLOOP	Number of committed packets containing an end loop that was taken.	Maskable
0x3a	PST_USED_P0P1BUSY	Number of times a store port was used when p0 and p1 were both occupied. Only increments when store port is present.	Maskable
0x3b	CYCLES_1_THREAD_RUNNING	Processor cycles that exactly one thread is running. Running means the thread is not in Wait or Stop mode.	Not maskable

Table 9-1 V75 processor event symbols

Event	Symbol	Definition	Maskable
0x3c	CYCLES_2_THREAD_RUNNING	Processor cycles that exactly two threads are running. Running means the threads are not in Wait or Stop mode.	Not maskable
0x3d	CYCLES_3_THREAD_RUNNING	Processor cycles that exactly three threads are running. Running means the threads are not in Wait or Stop mode.	Not maskable
0x3e	CYCLES_4_THREAD_RUNNING	Processor cycles that exactly four threads are running. Running means the threads are not in Wait or Stop mode.	Not maskable
0x3f	AXI_LINE128_READ_REQUEST	Number of 128-byte line read requests issued by the primary AXI master. Includes all interleaved requests.	Not maskable
0x40	AXI_READ_REQUEST	All read requests issued by the primary AXI master. Includes full lines, partial lines, and all interleaved requests.	Not maskable
0x41	AXI_LINE32_READ_REQUEST	Number of 32-byte line read requests issued by the primary AXI master. Includes all interleaved requests.	Not maskable
0x42	AXI_WRITE_REQUEST	All write requests issued by the primary AXI master. Includes full lines, partial lines, and all interleaved requests.	Not maskable
0x43	AXI_LINE32_WRITE_REQUEST	Number of 32-byte line write requests issued by the primary AXI master. Includes all interleaved requests. All bytes are valid.	Not maskable
0x44	AHB_READ_REQUEST	Number of read requests issued by the AHB master.	Not maskable
0x45	AHB_WRITE_REQUEST	Number of write requests issued by the AHB master.	Not maskable
0x46	AXI_LINE128_WRITE_REQUEST	Number of 128-byte line write requests issued by the primary AXI master. Includes all interleaved requests. All bytes are valid.	Not maskable
0x47	AXI_SLAVE_MULTI_BEAT_ACCESS	Number of AXI slave multi-beat accesses.	Not maskable
0x48	AXI_SLAVE_SINGLE_BEAT_ACCESS	Number of AXI slave single-beat accesses.	Not maskable
0x49	AXI2_READ_REQUEST	All read requests issued by the secondary AXI master. Includes full lines and partial lines.	Not maskable
0x4a	AXI2_LINE32_READ_REQUEST	Number of 32-byte line read requests issued by the secondary AXI master.	Not maskable
0x4b	AXI2_WRITE_REQUEST	All write requests issued by the secondary AXI master. Includes full lines and partial lines.	Not maskable
0x4c	AXI2_LINE32_WRITE_REQUEST	Number of 32-byte line write requests issued by the secondary AXI master.	Not maskable
0x4d	AXI2_CONGESTION	Secondary AXI command or data queue is full. An operation is stuck at the head of the secondary AXI master command queue.	Not maskable

Table 9-1 V75 processor event symbols

Event	Symbol	Definition	Maskable
0x50	COMMITTED_FPS	Number of committed floating point instructions. Increments by two for dual floating-point operations. Excludes conversions.	Maskable
0x51	REDIRECT_BIMODAL_MISPREDICT	Mispredicted bimodal branch direction caused a control flow redirect.	Not maskable
0x52	REDIRECT_TARGET_MISPREDICT	Mispredicted branch target caused a control flow redirect. Includes an RAS mispredict, and HintJR mispredict. Excludes indirect jumps and calls other than JUMPR R31 returns. Excludes direction mispredicts.	Not maskable
0x53	REDIRECT_LOOP_MISPREDICT	Mispredicted hardware loop end caused a control flow redirect. Can only happen when the loop has few packets and the loop count is 2 or less.	Not maskable
0x54	REDIRECT_MISC	Control flow is redirected for a reason other than events 81, 82, and 83. Includes exceptions, traps, interrupts, non-R31 jumps, multiple initialization loops in flight, and so on.	Not maskable
0x55	AXI_LINE256_WRITE_REQUEST	Number of 256-byte line write requests issued by the AXI master. All bytes are valid.	Not maskable
0x56	NUM_PACKET_CRACKED	Number of packets that cracked.	Maskable
0x58	JTLB_MISS	Instruction or data address translation request was missed in the JTLB.	Maskable
0x5a	COMMITTED_PKT_RETURN	Number of committed return instructions. Includes canceled returns.	Maskable
0x5b	COMMITTED_PKT_INDIRECT_JUMP	Number of committed indirect jumps or call instructions. Includes canceled instructions. Does not include JUMPR R31 returns.	Maskable
0x5c	COMMITTED_BIMODAL_BRANCH_-INSTS	Number of committed bimodal branches. Includes *.old and *.new. Increments by two for dual jumps.	Maskable
0x5f	VTCM_FIFO_FULL_CYCLES	Cycles cluster can be issued if the VTCM FIFO queue is full.	Not maskable
0x61	DU_L1S_LOAD_ACCESS	Number of scalar load accesses to L1S.	Not maskable
0x62	ICACHE_ACCESS	Number of I-cacheline fetches.	Not maskable
0x63	BTB_HIT	Number of branch target buffer hits.	Not maskable
0x64	BTB_MISS	Number of branch target buffer misses.	Not maskable
0x65	IU_DEMAND_SECONDARY_MISS	Number of I-cache secondary misses.	Not maskable
0x67	FAST_FETCH_KILLED	Number of fast fetches that were killed (after an I-cache access).	Not maskable
0x69	FETCHED_PACKETS_DROPPED	Number of packets that are dropped because the IU cannot deliver them to the CU.	Not maskable

Table 9-1 V75 processor event symbols

Event	Symbol	Definition	Maskable
0x6b	IU_PREFETCHES_SENT_TO_L2	Number of IU prefetches sent to the L2 cache. Includes cachelines not dropped by the L2 cache. Excludes replayed prefetches and only counts prefetches the L2 accepts. Excludes IU prefetches that are sent to L2 ITCM.	Maskable
0x6c	ITLB_MISS	Number of ITLB misses that go to JTLB.	Maskable
0x72	FETCH_2_CYCLE	Number of two-cycle fetches in an IU (returns, loop end, fall through, BTB).	Not maskable
0x73	FETCH_3_CYCLE	Number of three-cycle fetches in an IU.	Not maskable
0x75	L2_IU_SECONDARY_MISS	Number of L2 secondary misses from an IU.	Maskable
0x76	L2_IU_ACCESS	Number of L2 cacheable access from an IU. Includes any access to the L2 cache that was the result of an IU command, either demand or L1 prefetch access. Excludes any prefetches generated in the L2 cache. Excludes L2fetch, TCM accesses, and uncacheables. Address must target the primary AXI master.	Maskable
0x77	L2_IU_MISS	Number of L2 misses from an IU. Of the events qualified by 0x76, the event that resulted in an L2 miss (demand miss or L1 prefetch miss). An L2 miss is any condition that prevents the immediate return of data to the IU, excluding pipeline conflicts.	Maskable
0x78	L2_IU_PREFETCH_ACCESS	Number of prefetches from an IU to the L2 cache. Any IU prefetch access sent to the L2 cache. Access must be L2 cacheable and target the primary AXI. Does not include L2 fetch-generated accesses.	Maskable
0x79	L2_IU_PREFETCH_MISS	Number of L2 misses that were IU prefetches. Of the events qualified by 0x78, the events that resulted in an L2 miss.	Maskable
0x7a	L2_IU_BRANCH_CACHE_WRITE_-REQUEST	Number of requests sent from the IU to the L2 cache to write the bimodal bits of instructions in L2. Includes all requests, regardless of the target.	Not maskable
0x7b	L2_IU_BRANCH_CACHE_WRITE	Number of writes to the bimodal bits of instructions in the L2 cache. This event is the number of event 122 requests that completed by updating memory in the L2 cache or TCM.	Not maskable
0x7c	L2_DU_READ_ACCESS	Number of L2 cacheable read accesses from a DU. Any read access from the DU that might cause a lookup in the L2 cache. Includes loads, L1 prefetches, dcfetches. Excludes the initial L2fetch command, uncacheables, TCM accesses, and coprocessor loads. Must target the primary AXI master.	Maskable

Table 9-1 V75 processor event symbols

Event	Symbol	Definition	Maskable
0x7d	L2_DU_READ_MISS	Number of L2 read misses from a DU. Of the events qualified by 0x7C, any event that results in an L2 miss (that is, the line was not previously allocated in the L2 cache and is fetched from the backing memory).	Maskable
0x7e	L2FETCH_ACCESS	Number of L2 fetch accesses from a DU. Any access to the L2 cache from the L2 prefetch engine initiated by programming the L2fetch engine.	Maskable
0x7f	L2FETCH_MISS	Number of L2 fetch misses from a programmed inquiry. Of the events qualified by 0x7E, the event that results in an L2 miss (that is, the line was not previously allocated in the L2 cache and is fetched from the backing memory).	Maskable
0x81	L2_ACCESS	All requests to the L2 cache. Does not include internally generated accesses like L2 fetch, however the programming of the L2Fetch engine is counted. All accesses to odd interleave or even interleave are counted. Can be L2 cacheable or TCM.	Maskable
0x82	L2_PIPE_CONFLICT_STALL	Request is not taken by the L2 cache due to a pipe conflict. The conflict can be a tag bank, data bank, or other pipeline conflict.	Maskable
0x83	L2_TAG_ARRAY_CONFLICT	Of the items in event 130, the items caused by a conflict with the tag array.	Maskable
0x87	TCM_DU_ACCESS	Number of TCM accesses from a DU. DU access to the L2 TCM space. Excludes HVX requests.	Maskable
0x88	TCM_DU_READ_ACCESS	Number of TCM read accesses from a DU. DU read access to the L2 TCM space. Includes HVX requests.	Maskable
0x89	TCM_IU_ACCESS	Number of TCM accesses from an IU. IU access to the L2 TCM space.	Maskable
0x8a	L2_CASTOUT	L2 cache evicts a dirty line due to an allocation. This event is not triggered on cache operations.	Not maskable
0x8b	L2_DU_STORE_ACCESS	Number of L2 cacheable store access from a DU. Any store access from the DU that might cause a lookup in the L2 cache. Excludes cache operations, uncacheables, TCM, and coprocessor stores. Must target the primary AXI master.	Maskable
0x8c	L2_DU_STORE_MISS	Number of L2 misses from a DU. Of the events qualified by 0x8B, the events that resulted in a miss. Specifically, the cases where the line is not in the cache or a coalesce buffer.	Maskable
0x8d	L2_DU_PREFETCH_ACCESS	Number of L2 prefetch accesses from a DU. Of the events qualified by 0x7C, the events that are dcfetch and dhwprefetch. These L2 cacheable events target the primary AXI master.	Maskable

Table 9-1 V75 processor event symbols

Event	Symbol	Definition	Maskable
0x8e	L2_DU_PREFETCH_MISS	Number of L2 prefetch misses from a DU. Of the events qualified by 0x8D, the events that missed the L2 cache.	Maskable
0x90	L2_DU_LOAD_SECONDARY_MISS	Number of L2 load secondary misses from a DU. Hit a busy line in the scoreboard, which prevented a return. A busy condition can include pipeline bubbles caused by back-to-back loads, like L1 UC loads.	Maskable
0x91	L2FETCH_COMMAND	Number of L2fetch commands. Excludes L2 fetch stop commands.	Not maskable
0x92	L2FETCH_COMMAND_KILLED	L2 fetch command was killed because a stop command was issued. Increments once for each L2 fetch command that is killed. If multiple commands are queued to the L2Fetch engine, the kill of each command is recorded.	Maskable
0x93	L2FETCH_COMMAND_OVERWRITE	L2 fetch command was overwritten. Kills an old L2 fetch command and replaces it with a new command.	Not maskable
0x94	L2FETCH_ACCESS_CREDIT_FAIL	L2 fetch access could not get a credit. L2 fetch was blocked due to a missing L2 fetch or L2 evict credit.	Not maskable
0x95	AXI_SLAVE_READ_BUSY	AXI slave read access hit a busy line.	Not maskable
0x96	AXI_SLAVE_WRITE_BUSY	AXI slave write access hit a busy line.	Not maskable
0x97	L2_ACCESS_EVEN	Of the events in 0x81, number of accesses made to the even L2 cache.	Maskable
0x98	CLADE_HIGH_PRIO_L2_ACCESS	Number of IU or DU requests for a high-priority CLADE region. Not counted for an L2 fetch.	Maskable
0x99	CLADE_LOW_PRIO_L2_ACCESS	Number of IU or DU requests for a low-priority CLADE region. Not counted for an L2 fetch.	Maskable
0x9a	CLADE_HIGH_PRIO_L2_MISS	Number of CLADE high-priority L2 accesses that missed in the L2 cache.	Maskable
0x9b	CLADE_LOW_PRIO_L2_MISS	Number of CLADE low-priority L2 accesses that missed in the L2 cache.	Maskable
0x9c	CLADE_HIGH_PRIO_EXCEPTION	CLADE high-priority decode that had an exception.	Not maskable
0x9d	CLADE_LOW_PRIO_EXCEPTION	CLADE low-priority decode that had an exception.	Not maskable
0x9e	AXI2_SLAVE_READ_BUSY	AXI secondary slave read access hit a busy line.	Not maskable
0x9f	AXI2_SLAVE_WRITE_BUSY	AXI secondary slave write access hit a busy line.	Not maskable
0xa0	ANY_DU_STALL	Any DU stall. Increments once when the thread has a DU stall (D-cache miss or DTLB miss).	Not maskable

Table 9-1 V75 processor event symbols

Event	Symbol	Definition	Maskable
0xa1	DU_BANK_CONFLICT_REPLAY	DU bank conflict replay. Dual memory access to same bank, but different lines.	Maskable
0xa2	DU_CREDIT_REPLAY	Number of times a packet took a replay because insufficient QoS DU credits were available.	Maskable
0xa3	L2_FIFO_FULL_REPLAY	Number of L2 even or odd FIFO full replays.	Maskable
0xa4	DU_STORE_BUFFER_FULL_REPLAY	Number of DU replays because a demand load access hit in the store buffer.	Maskable
0xa7	DU_SNOOP_REQUEST	Number of DU snoop requests that were accepted.	Not maskable
0xa8	DU_FILL_REPLAY	Fill has an index conflict with an instruction from the same thread in the pipeline. Fills and demands might be from different threads if there is a prefetch from the deferral queue, or if a fill has not be acknowledged for very long and forces itself into the pipeline.	Maskable
0xac	DU_READ_TO_L2	Number of DU reads to L2 cache. Total of everything that brings data from the L2 array. Includes prefetches (dcfetch and hwprefetch). Excludes coprocessor loads.	Not maskable
0xad	DU_WRITE_TO_L2	Number of DU writes to L2 cache. Total of everything that is written out of the DU to the L2 array. Includes dczeroa. Excludes dcclean, dccleaninv, tag writes, and coprocessor stores.	Not maskable
0xaf	DCZERO_COMMITTED	Dczeroa instruction was committed.	Maskable
0xb0	L2ITCM_IU_READ	Number of ITCM accesses from an IU. Includes IU demand fetches and IU prefetches. This event is not included in any other L2 events.	Not maskable
0xb1	L2ITCM_DU_READ	Number of L2 ITCM read accesses from a DU. Includes all demands and prefetches. This event is not included in any other L2 events.	Not maskable
0xb2	L2ITCM_DU_WRITE	Number of L2 ITCM write accesses from a DU. Includes stores and dczeroa events. Does not include any cache operations. This event is not included in any other L2 events.	Not maskable
0xb3	DTLB_MISS	DTLB miss that goes to JTLB. When both slots miss to different pages, increments by two. When both slots miss to the same page, only counts S1 because S1 goes first and fills for S0.	Maskable
0xb4	L2ITCM_BIMODAL_WRITES_-SUCCESS	Number of successful bimodal writes into L2 ITCM.	Not maskable
0xb6	STORE_BUFFER_HIT_REPLAY	Store buffer hit is replayed because a packet with two stores goes to the same bank but different cachelines, followed by a load from an address pushed into the store buffer.	Maskable

Table 9-1 V75 processor event symbols

Event	Symbol	Definition	Maskable
0xb7	STORE_BUFFER_FORCE_REPLAY	Store buffer must drain, forcing the current packet to replay. Typically occurs on a cache index match between the current packet and store buffer. Can also a store buffer timeout.	Maskable
0xb8	TAG_WRITE_CONFLICT_REPLAY	Number of inter-cluster tag write conflicts.	Maskable
0xb9	SMT_BANK_CONFLICT	Number of inter-thread SMT bank conflicts.	Not maskable
0xba	PORT_CONFLICT_REPLAY	Number of all port conflict replays, including the same cluster replays caused by high-priority fills and store buffer force drains. Includes inter-cluster replays.	Maskable
0xbb	L2ITCM_BIMODAL_WRITES_-DROPPED	Number of bimodal writes into L2 ITCM that were dropped.	Not maskable
0xbc	L2ITCM_IU_PREFETCH_READ	Number of ITCM accesses from IU prefetches. Includes only prefetches from an IU. This event is included in event 176. It is not included in any other L2 events.	Not maskable
0xbd	PAGE_CROSS_REPLAY	Page cross from a valid packet that caused a replay. Excludes pdkill packets. Counts twice if both slots cause a page cross.	Maskable
0xbe	PST_STORE_SENTON_OTHPORT	Number of times a slot 0 store was sent to the other store buffer because the other cluster had a slot 1 store or memop. Only increments when store port is present.	Maskable
0xbf	DU_DEMAND_SECONDARY_MISS	Number of DU demand secondary misses.	Maskable
0xc0	DU_MISC_REPLAY	All DU replays not counted by other replay events. This event counts every time ANY_DU_REPLAY counts and no other DU replay event counts.	Maskable
0xc2	DU_STATE_REPLAY	Number of times an access replayed because the access one cycle ahead of it allocates or invalidates a way in the same index.	Maskable
0xc3	DCFETCH_COMMITTED	Number of dcfetches committed. Includes hits and drops. Does not include convert-to-prefetches.	Maskable
0xc4	DCFETCH_HIT	Number of dcfetch hits in D-cache. Includes hitting valid or reserved lines.	Maskable
0xc5	DCFETCH_MISS	Number of dcfetches missed in L1 cache. Counts the dcfetches issued to L2 FIFO.	Maskable
0xc8	DU_LOAD_UNCACHEABLE	Load instructions with addresses uncacheable in the L1 cache.	Maskable
0xc9	DU_DUAL_LOAD_UNCACHEABLE	Packets where both loads have addresses uncacheable in the L1 cache	Maskable
0xca	DU_STORE_UNCACHEABLE	Store instructions with addresses uncacheable in the L1 cache.	Maskable
0xcb	DU_STORE_RELEASE_CREDIT_-STALL	Stall occurs because there are not enough credits from the store release.	Maskable

Table 9-1 V75 processor event symbols

Event	Symbol	Definition	Maskable
0xcd	AXI_LINE256_READ_REQUEST	Number of 256-byte line read requests issued by the AXI master. All bytes are valid.	Not maskable
0xce	AXI_LINE64_READ_REQUEST	Number of 64-byte line read requests issued by the primary AXI master. Includes all interleaved requests.	Not maskable
0xcf	AXI_LINE64_WRITE_REQUEST	Number of 64-byte line write requests issued by the primary AXI master. Includes all interleaved requests. All bytes are valid.	Not maskable
0xd1	AHB_8_READ_REQUEST	Number of 8-byte read requests issued by the AHB.	Not maskable
0xd3	L2FETCH_COMMAND_PAGE_-TERMINATION	L2fetch command terminated because it could not get a page translation from VA to PA. Includes terminations due to permission errors. That is, an address translation can fail because the VA to PA is not in the TLB, or the properties in the translation are not acceptable and the command terminates.	Not maskable
0xd5	L2_DU_STORE_COALESCE	Number of events from 139 that were coalesced	Maskable
0xd6	L2_STORE_LINK	Number of times a new store links to something else in the scoreboard.	Maskable
0xd7	L2_SCOREBOARD_70_PERCENT_-FULL	Increments by one for every cycle where the L2 scoreboard is at least 70% full. For a 32-entry scoreboard, 23 or more entries are consumed. This event continues to count even if the scoreboard is more than 80% full. For more than one interleave, this event considers only the scoreboard that has the most entries consumed.	Not maskable
0xd8	L2_SCOREBOARD_80_PERCENT_-FULL	Increments by one for every cycle where the L2 scoreboard is at least 80% full. For a 32-entry scoreboard, 26 or more entries are consumed. This event continues to count even if the scoreboard is more than 90% full. For more than one interleave, this event considers only the scoreboard that has the most entries consumed.	Not maskable
0xd9	L2_SCOREBOARD_90_PERCENT_-FULL	Increments by one for every cycle where the L2 scoreboard is at least 90% full. For a 32-entry scoreboard, 29 or more entries are consumed. For more than one interleave, this event considers only the scoreboard that has the most entries consumed.	Not maskable
0xda	L2_SCOREBOARD_FULL_REJECT	L2 scoreboard is too full to accept a selector request, and the selector has a request.	Not maskable
0xdc	L2_EVICTION_BUFFERS_FULL	Counts every cycle when all eviction buffers in any interleave are occupied.	Not maskable
0xdd	AHB_MULTI_BEAT_READ_REQUEST	Number of 32-byte multi-beat read requests issued by the AHB.	Not maskable
0xdf	L2_DU_LOAD_SECONDARY_MISS_ON_SW_PREFETCH	Of the events in 0x90, the events where the primary miss was a DC fetch or L2 fetch.	Maskable

Table 9-1 V75 processor event symbols

Event	Symbol	Definition	Maskable
0xe0	L2FETCH_DROP	L2 fetch data dropped because a previous eviction has not completed.	Not maskable
0xe5	THREAD_IDLE_PVIEW_CYCLES	Cycles cluster cannot commit because a thread is in the Off , Wait state or in pause.	Maskable
0xe6	ARCH_LOCK_PVIEW_CYCLES	Cycles cluster cannot commit due to a kernel lock or TLB lock.	Maskable
0xe7	REDIRECT_PVIEW_CYCLES	Cycles cluster cannot commit because of redirects such as branch mispredicts.	Maskable
0xe8	IU_NO_PKT_PVIEW_CYCLES	Cycles cluster cannot commit because the issue queue is empty.	Maskable
0xe9	DU_CACHE_MISS_PVIEW_CYCLES	Cycles cluster cannot commit due to a D-cache cacheable miss.	Maskable
0xea	DU_BUSY_OTHER_PVIEW_CYCLES	Cycles cluster cannot commit due to a DU replay, DU bubble, or DTLB miss.	Maskable
0xeb	CU_BUSY_PVIEW_CYCLES	Cycles cluster cannot commit due to a register interlock, register port conflict, bubbles due to a timing class such as tc_3stall, no B2B HVX, or HVX FIFO is full.	Maskable
0xec	DU_CONFLICT_PVIEW_CYCLES	Cycles cluster cannot commit due to a DU resource conflict.	Maskable
0xed	COPROC_BUSY_PVIEW_CYCLES	Cycles cluster cannot commit because the coprocessor is busy.	Maskable
0xee	DU_UNCACHED_PVIEW_CYCLES	Cycles cluster cannot commit due to a D-cache uncacheable access.	Maskable
0xef	SYSTEM_BUSY_PVIEW_CYCLES	Cycles cluster cannot commit due to system level stalls, including DMA synchronization, ETM is full, Qtimer read is not ready, AXI bus is busy, and global cache operations synchronization.	Maskable
0xf1	AXI_LINE128_READ_REQUEST_EVEN	Number of 128-byte line read requests issued by the even interleaved AXI master.	Not maskable
0xf2	AXI_READ_REQUEST_EVEN	All read requests issued by the even interleaved AXI master.	Not maskable
0xf3	AXI_LINE32_READ_REQUEST_EVEN	Number of 32-byte line read requests issued by the even-interleaved AXI master.	Not maskable
0xf4	AXI_WRITE_REQUEST_EVEN	All write requests issued by the even-interleaved AXI master.	Not maskable
0xf5	AXI_LINE32_WRITE_REQUEST_EVEN	Number of 32-byte line write requests issued by the even-interleaved AXI master. All bytes are valid.	Not maskable
0xf6	AXI_LINE128_WRITE_REQUEST_-EVEN	Number of 128-byte line write requests issued by the even-interleaved AXI master. All bytes are valid.	Not maskable
0xf8	AXI_LINE64_READ_REQUEST_EVEN	Number of 64-byte line read requests issued by the even-interleaved AXI master.	Not maskable

Table 9-1 V75 processor event symbols

Event	Symbol	Definition	Maskable
0xf9	AXI_LINE64_WRITE_REQUEST_EVEN	Number of 64-byte line write requests issued by the even-interleaved AXI master. All bytes are valid.	Not maskable
0xfa	AXI_WR_CONGESTION_EVEN	Even-interleaved AXI write command or data queue is full, and an operation is stuck at the head of the even interleaved AXI master command queue.	Not maskable
0xfb	AXI_INCOMPLETE_WRITE_REQUEST_EVEN	L2 line-sized write was made to the even-interleaved AXI master, but not all bytes were valid. Includes segmented writes. Excludes WT stores. This event captures the number of writes coalesced at a line level.	Not maskable
0xfc	AXI_LINE256_READ_REQUEST_EVEN	Number of 256-byte line read requests issued by an even-interleaved AXI master. All bytes are valid.	Not maskable
0xfd	AXI_LINE256_WRITE_REQUEST_-EVEN	Number of 256-byte line write requests issued by an even-interleaved AXI master. All bytes are valid.	Not maskable
0xfe	CYCLES_3_COPROC_THREADS_ONE_CLUSTER	Number of processor cycles during which a cluster has three threads in Run mode with the coprocessor bit (SSR.XE) enabled.	Maskable
0x2fa	L2_CLEAN_CASTOUT	Number of clean line evictions from L2 cache. Triggers when L2 cache evicts a line due to an allocation. Not triggered on cache operations.	Not maskable
0x2fb	AXI3_READ_REQUEST	All read requests issued by the tertiary AXI master. Includes full lines and partial lines.	Not maskable
0x2fc	AXI3_LINE32_READ_REQUEST	Number of 32-byte line read requests issued by the tertiary AXI master.	Not maskable
0x2fd	AXI3_WRITE_REQUEST	All write requests issued by the tertiary AXI master. Includes full lines and partial lines.	Not maskable
0x2fe	AXI3_LINE32_WRITE_REQUEST	Number of 32-byte line write requests issued by the tertiary AXI master.	Not maskable
0x2ff	AXI3_RD_CONGESTION	Tertiary AXI read command queue is full, and an operation is stuck at the head of the primary AXI master command queue. Includes all interleaved requests.	Not maskable
0x300	CYCLES_1_PACKET_COMMITTED	Number of cycles when one packet is committed.	Not maskable
0x301	CYCLES_2_PACKET_COMMITTED	Number of cycles when two packets are committed.	Not maskable
0x302	CYCLES_3_PACKET_COMMITTED	Number of cycles when three packets are committed.	Not maskable
0x303	CYCLES_4_PACKET_COMMITTED	Number of cycles when four packets are committed.	Not maskable
0x304	SMT_CLUSTER0	Number of cycles when more than one packet is committed in cluster 0.	Not maskable

Table 9-1 V75 processor event symbols

Event	Symbol	Definition	Maskable
0x305	SMT_CLUSTER1	Number of cycles when more than one packet is committed in cluster 1.	Not maskable
0x306	SMT_INTERCLUSTER	Number of cycles when packets are committed on both clusters.	Not maskable
0x307	SMT_CONFLICT_FOR_REG_READ_OR_CU_FWD	Number of cases when a packet is SMT-able without slot resource conflicts, but it cannot go due to s0/s1 register read/CU forwarding.	Not maskable
0x308	COMMITTED_PKT_2_THREAD_RUNNING_2T_PLUS_0T	Number of committed packets with two threads running on the same cluster. Running means the threads are not in the Wait or Stop state.	Not maskable
0x309	COMMITTED_PKT_2_THREAD_RUNNING_1T_PLUS_1T	Number of committed packets with two threads running on the different cluster. Running means the threads are not in the Wait or Stop state.	Not maskable
0x30a	COMMITTED_PKT_3_THREAD_RUNNING_3T_PLUS_0T	Number of committed packets with three threads running on the same cluster. Running means the threads are not in the Wait or Stop state.	Not maskable
0x30b	COMMITTED_PKT_3_THREAD_RUNNING_2T_PLUS_1T	Number of committed packets with three threads running, two threads on one cluster and 1 on another cluster. Running means the threads are not in the Wait or Stop state.	Not maskable
0x30c	COMMITTED_PKT_4_THREAD_RUNNING_4T_PLUS_0T	Number of committed packets with four threads running on same cluster. Running means the threads are not in the Wait or Stop state.	Not maskable
0x30d	COMMITTED_PKT_4_THREAD_RUNNING_3T_PLUS_1T	Number of committed packets with four threads running, three threads on one cluster and one thread running on another cluster. Running means the threads are not in the Wait or Stop state.	Not maskable
0x30e	COMMITTED_PKT_4_THREAD_RUNNING_2T_PLUS_2T	Number of committed packets with four threads running, two threads on one cluster and two threads running on another cluster. Running means the threads are not in the Wait or Stop state.	Not maskable
0x30f	COMMITTED_PKT_5_THREAD_RUNNING_4T_PLUS_1T	Number of committed packets with five threads running, four threads on one cluster and one thread running on another cluster. Running means the threads are not in the Wait or Stop state.	Not maskable
0x310	COMMITTED_PKT_5_THREAD_RUNNING_3T_PLUS_2T	Number of committed packets with five threads running, three threads on one cluster and two threads running on another cluster. Running means the threads are not in the Wait or Stop state.	Not maskable
0x311	COMMITTED_PKT_6_THREAD_RUNNING_4T_PLUS_2T	Number of committed packets with six threads running, four threads on one cluster and two threads running on another cluster. Running means the threads are not in the Wait or Stop state.	Not maskable

Table 9-1 V75 processor event symbols

Event	Symbol	Definition	Maskable
0x312	COMMITTED_PKT_6_THREAD_RUNNING_3T_PLUS_3T	Number of committed packets with six threads running, three threads on one cluster and three threads running on another cluster. Running means the threads are not in the Wait or Stop state.	Not maskable
0x313	ICACHE_DEMAND_MISS_PREFETCH_MISS	Number of iprfetches initiated on demand misses.	Not maskable
0x314	SIMPLE_PACKET	Number of committed simple packets, which can be dispatched on in-cluster SMT threads. Includes eligible packets that are committed on both primary and in-cluster SMT threads.	Not maskable
0x315	AXI3_LINE64_WRITE_REQUEST	Number of 64-byte line write requests issued by the primary AXI3 master. Includes all interleaved requests. All bytes are valid.	Not maskable
0x316	AXI3_LINE64_READ_REQUEST	Number of 64-byte line read requests issued by the primary AXI3 master. Includes all interleaved requests	Not maskable
0x317	AXI3_WR_CONGESTION	Tertiary AXI write command or data queue is full, and an operation is stuck at the head of the primary AXI3 master command queue. Includes all interleaved requests.	Not maskable
0x318	AXI3_INCOMPLETE_WRITE_REQUEST	L2 line-sized write was made to the AXI3 master, but not all bytes were valid. Includes segmented writes. Excludes WT stores.	Not maskable
0x319	ICACHE_DATA_REPLAY	Number of I-cache data replays due to incorrect way predictions.	Not maskable
0x31c	SMT_PKT_PICKED_BUT_NOT_COMMIT_PVIEW_CYCLES	In-cluster SMT thread is picked, but not committed.	Not maskable
0x31d	SMT_PKT_IQ_NO_PKT_PVIEW_CYCLES	In-cluster SMT thread is not picked because no packet is in IQ on the SMT thread.	Not maskable
0x31e	SMT_PKT_NOT_SIMPLE_PVIEW_CYCLES	In-cluster SMT thread is not picked because no simple packet is on the SMT thread.	Not maskable
0x31f	SMT_PKT_NOT_READY_PVIEW_CYCLES	In-cluster SMT thread is not picked because no simple packet is ready for dispatch on the SMT thread.	Not maskable
0x320	SMT_PKT_SLOT_CONFLICT_PVIEW_CYCLES	In-cluster SMT thread is not picked due to a slot conflict between the primary thread and SMT thread.	Not maskable
0x321	SMT_PKT_REG_FWD_BLOCK_PVIEW_CYCLES	In-cluster SMT thread is not picked due to a register and forward block on the SMT thread.	Not maskable
0x322	CLADE2_EB_FULL	CLADE2 can use up to two eviction buffer entries. Indicates that both entries are used and CLADE2 is congested.	Not maskable
0x323	CLADE2_RD_REQ	Number of L2 cache read requests in the CLADE2 region.	Not maskable

Table 9-1 V75 processor event symbols

Event	Symbol	Definition	Maskable
0x324	CLADE2_RDCACHE_MISS	Number of L2 cache read request misses in the CLADE2 region.	Not maskable
0x325	CLADE2_WR_REQ	Number of L2 cache write requests in the CLADE2 region.	Not maskable
0x326	CLADE2_WRCACHE_MISS	Number of L2 cache write request misses in the CLADE2 region.	Not maskable
0x327	AXI_EWD_REQUEST	L2 cache eviction of clean data to the AXI master bus.	Not maskable
0x328	AXI_EWD_REQUEST_EVEN	L2 cache eviction of clean data to the even interleaved AXI master bus.	Not maskable
0x329	AXI_CMO_REQUEST	Cache maintenance operation request from the QDSP6 core to AXIM.	Not maskable
0x32a	AXI_CMO_REQUEST_EVEN	Cache maintenance operation request from the QDSP6 core to AXIM Interleave0.	Not maskable
0x32d	TAGE_TABLE_ALLOC	Number of allocations in the TAGE branch predictor due to mispredicting branches.	Not maskable
0x32e	TAGE_TABLE_HIT	Number of hits in the TAGE branch predictor table.	Not maskable
0x32f	TAGE_BRANCH_OVERRIDE	Number of times the TAGE branch predictor overrode the bimodal prediction for a branch instruction.	Not maskable
0x333	DU_SPF_DTLBPGCROSS	Number of stopping prefetching due to page cross.	Not maskable
0x334	DU_SPF_DCACHE_HIT	Number of prefetch requests that hit in L1D$	Not maskable
0x335	DU_SPF_DCACHE_MISS	Number of prefetch requests missing in L1D$	Not maskable
0x336	DU_SPF_L2FIFOFULL_RETRY	Number of prefetch retry on L2FIFO queue full.	Not maskable
0x337	DU_SPF_L2BUFFULL_RETRY	Number of prefetch retry on L2 credits/AQoS busy.	Not maskable
0x338	DU_SPF_CONFLICT_RETRY	Number of cycles that prefetches losing arbitration.	Not maskable
0x340	DPM_AVG_COMPRESSED	Representation of relative DPM value (obtained using the DPM weights) produced every cycle. Computed absolute average power (mW) = (DPM_AVG_COMPRESSED count * max DPM value from DPM weights * scaling factor)/execution time (ns).	Not maskable

10 Instruction encoding

10.1 Instructions

Hexagon processor instructions are encoded in a 32-bit instruction word. The instruction word format varies according to the instruction type.

The instruction words contain two types of bit fields:

- Common fields appear in every processor instruction, and are defined the same in all instructions.
- Instruction-specific fields appear only in some instructions, or vary in definition across the instruction set.

Table 10-1 Instruction bit fields

Name	Description	Type
ICLASS	Instruction class	Common
Parse	Packet / loop bits	

Table 10-1 Instruction bit fields

Name	Description	Type
MajOp Maj	Major opcode	Instruction-specific
MinOp Min	Minor opcode	
RegType	Register type (32-bit, 64-bit)	
Type	Operand type (byte, halfword, and so on)	
Amode	Addressing mode	
d*n*	Destination register operand	
s*n*	Source register operand	
t*n*	Source register operand #2	
x*n*	Source and destination register operand	
u*n*	Predicate or modifier register operand	
sH	Source register bit field (Rs.H or Rs.L)	
tH	Source register #2 bit field (Rt.H or Rt.L)	
UN	Unsigned operand	
Rs	No source register read	
P	Predicate expression	
PS	Predicate sense (Pu or !Pu)	
DN	Dot-new predicate	
PT	Predict taken	
sm	Supervisor mode only	

NOTE: In some cases, instruction-specific fields encode instruction attributes other than the ones described for the fields in Table 10-1.

Reserved bits

Some instructions contain reserved bits that do not currently encode instruction attributes. Always set these bits to 0 to ensure compatibility with any future changes in the instruction encoding.

NOTE: Reserved bits appear as '–' characters in the instruction encoding tables.

10.2 Sub-instructions

To reduce code size, the Hexagon processor supports the encoding of certain pairs of instructions in a single 32-bit container. Instructions encoded this way are sub-instructions, and the containers are duplexes (Section 10.3).

Sub-instructions are limited to certain commonly-used instructions:

- Arithmetic and logical operations
- Register transfer
- Loads and stores
- Stack frame allocation/deallocation
- Subroutine return

Table 10-2 lists the sub-instructions along with the group identifiers that encode them in duplexes.

Sub-instructions can access only a subset of the general registers (R0 to R7, R16 to R23). Table 10-3 lists the sub-instruction register encodings.

NOTE: Certain sub-instructions implicitly access registers such as SP (R29).

Table 10-2 Sub-instructions

Group	Instruction	Description
L1	`Rd = memw(Rs+#u4:2)`	Word load
L1	`Rd = memub(Rs+#u4:0)`	Unsigned byte load
Group	**Instruction**	**Instruction**
L2	`Rd = memh/memuh(Rs+#u3:1)`	Halfword loads
L2	`Rd = memb(Rs+#u3:0)`	Signed byte load
L2	`Rd = memw(r29+#u5:2)`	Load word from stack
L2	`Rdd = memd(r29+#u5:3)`	Load pair from stack
L2	`deallocframe`	Deallocate stack frame
L2	`if ([!]P0) dealloc_return` `if ([!]P0.new) dealloc_return:nt`	Deallocate stack frame and return

L2	`jumpr R31`
	`if ([!]P0) jumpr R31`
	`if ([!]P0.new) jumpr:nt R31`
		Return

Group	**Instruction**	**Instruction**
S1	`memw(Rs+#u4:2) = Rt`	Store word
S1	`memb(Rs+#u4:0) = Rt`	Store byte
Group	**Instruction**	**Instruction**
S2	`memh(Rs+#u3:1) = Rt`	Store halfword
S2	`memw(r29+#u5:2) = Rt`	Store word to stack
S2	`memd(r29+#s6:3) = Rtt`	Store pair to stack
S2	`memw(Rs+#u4:2) = #U1`	Store immediate word #0 or #1

Table 10-2 Sub-instructions (cont.)

Group	Instruction	Description
S2	`memb(Rs+#u4) = #U1`	Store immediate byte #0 or #1
S2	`allocframe(#u5:3)`	Allocate stack frame
Group	**Instruction**	**Instruction**
A	`Rx = add(Rx,#s7)`	Add immediate
A	`Rd = Rs`	Transfer
A	`Rd = #u6`	Set to unsigned immediate
A	`Rd = #-1`	Set to -1
A	`if ([!]P0[.new]) Rd = #0`	Conditional clear
A	`Rd = add(r29,#u6:2)`	Add immediate to stack pointer
A	`Rx = add(Rx,Rs)`	Register add
A	`P0 = cmp.eq(Rs,#u2)`	Compare register equal immediate
A	`Rdd = combine(#0,Rs)`	Combine zero and register into pair
A	`Rdd = combine(Rs,#0)`	Combine register and zero into pair
A	`Rdd = combine(#u2,#U2)`	Combine immediates into pair
A	`Rd = add(Rs,#1)` `Rd = add(Rs,#-1)`	Add and subtract 1
A	`Rd = sxth/sxtb/zxtb/zxth(Rs)`	Sign- and zero-extends
A	`Rd = and(Rs,#1)`	And with 1

Table 10-3 Sub-instruction registers

Register	Encoding

`Rs,Rt,Rd,Rx`	`0000 = R0`
	`0001 = R1`
	`0010 = R2`
	`0011 = R3`
	`0100 = R4`
	`0101 = R5`
	`0110 = R6`
	`0111 = R7`
	`1000 = R16`
	`1001 = R17`
	`1010 = R18`
	`1011 = R19`
	`1100 = R20`
	`1101 = R21`
	`1110 = R22`
	`1111 = R23`

Table 10-3 Sub-instruction registers

Register	Encoding
Rdd,Rtt	000 = R1:0 001 = R3:2 010 = R5:4 011 = R7:6 100 = R17:16 101 = R19:18 110 = R21:20 111 = R23:22

10.3 Duplexes

A duplex is encoded as a 32-bit instruction with bits [15:14] set to 00. The sub-instructions that comprise a duplex are encoded as 13-bit fields in the duplex.

An instruction packet can contain one duplex and up to two other (non-duplex) instructions. The duplex must always appear as the last word in a packet.

The sub-instructions in a duplex always execute in slot 0 and slot 1.

Table 10-4 Duplex instruction encoding

Bits	Name	Description
15:14	Parse bits	00 = Duplex type, ends the packet and indicates that word contains two sub-instructions
12:0	Sub-instruction low	Encodes slot 0 sub-instruction
28:16	Sub-instruction high	Encodes slot 1 sub-instruction
31:29, 13	4-bit ICLASS	Indicates the group to which the low and high sub-instructions belong.

The duplex ICLASS field values that specify the group of each sub-instruction in a duplex are shown in Table 10-5.

Table 10-5 Duplex ICLASS field

ICLASS	Low slot 0 subinsn type	High slot 1 subinsn type
0x0	L1-type	L1-type
0x1	L2-type	L1-type
0x2	L2-type	L2-type
0x3	A-type	A-type
0x4	L1-type	A-type
0x5	L2-type	A-type
0x6	S1-type	A-type
0x7	S2-type	A-type
0x8	S1-type	L1-type
0x9	S1-type	L2-type
0xA	S1-type	S1-type

Table 10-5 Duplex ICLASS field

ICLASS	Low slot 0 subinsn type	High slot 1 subinsn type
0xB	S2-type	S1-type
0xC	S2-type	L1-type
0xD	S2-type	L2-type
0xE	S2-type	S2-type
0xF	Reserved	Reserved

Duplexes have the following grouping constraints:

- Constant extenders expand the range of the immediate operand of an instruction to 32 bits, and can expand the following sub-instructions:
  - `Rx = add(Rx,#s7)`
  - `Rd = #u6`

  A duplex can contain only one constant-extended instruction, and it must appear in the slot 1 position.
- When the sub-instructions are treated as 13-bit unsigned integer values for two instructions with the same sub-instruction group in a duplex, the instruction corresponding to the numerically smaller value must be encoded in the slot 1 position of the duplex.[1]
- Sub-instructions must conform to any slot assignment grouping rules that apply to the individual instructions, even if a duplex pattern exists that violates those assignments. One exception to this rule exists:
  - jumpr R31 must appear in the Slot 0 position

[1] The sub-instruction register and immediate fields are assumed to be 0 when performing this comparison.

10.4 Instruction classes

The instruction class (Section 3.2) is encoded in the four most-significant bits of the instruction word (31:28). These bits are referred to as the ICLASS field of the instruction. The Slots column in Table 10-6 indicates which slots can receive the instruction class.

Table 10-6 Instruction class encoding

Encoding	Instruction class	Slots
0000	Constant extender (Section 10.9)	–
0001	J	2, 3
0010	J	2, 3
0011	LD ST	0, 1
0100	LD ST (conditional or GP-relative)	0, 1
0101	J	2, 3
0110	CR	3
0111	ALU32	0, 1, 2, 3
1000	XTYPE	2, 3
1001	LD	0, 1
1010	ST	0
1011	ALU32	0, 1, 2, 3
1100	XTYPE	2, 3
1101	XTYPE	2, 3
1110	XTYPE	2, 3
1111	ALU32	0, 1, 2, 3

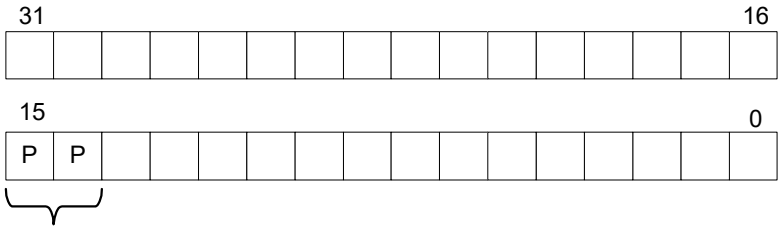
For details on encoding the individual class types, see Chapter 11.

10.5 Instruction packets

Instruction packets are encoded using two bits of the instruction word (15:14), which are referred to as the Parse field of the instruction word. The field values have the following definitions:

- '11' indicates that an instruction is the last instruction in a packet (the instruction word at the highest address).
- '01' or '10' indicate that an instruction is not the last instruction in a packet.
- '00'indicates a duplex.

If any sequence of four consecutive instructions occurs without one of them containing '11' or '00', the processor raises an error exception (illegal opcode).



Packet/loop parse bits:
01, 10 = not end of packet
11 = end of packet
00 = duplex

Figure 10-1 Parse field instruction packet encoding

The following examples show how to use the Parse field to encode instruction packets:

```
{ A ; B}
  01 11              // Parse fields of instructions A,B

{ A ; B ; C}
  01 01  11          // Parse fields of instructions A,B,C

{ A ; B ; C ; D}
  01 01  01  11      // Parse fields of instructions A,B,C,D
```

10.6 Loop packets

In addition to encoding the last instruction in a packet, the Parse field of the instruction word (Section 10.5) encodes the last packet in a hardware loop.

The Hexagon processor supports two Hardware loops that are labeled 0 and 1. The last packet in these loops is subject to the following restrictions:

- The last packet in a hardware loop 0 must contain two or more instruction words.
- The last packet in a hardware loop 1 must contain three or more instruction words.

If the last packet in a loop is expressed in assembly language with fewer than the required number of words, the assembler automatically adds one or two NOP instructions to the encoded packet so it contains the minimum required number of instruction words.

The Parse fields in the first and second instruction words (the words at the lowest addresses) of a packet encode whether the packet is the last packet in a hardware loop.

Table 10-7 Parse field loop packet encoding

Packet	Parse field in first Instruction	Parse field in second Instruction
Not last in loop	01 or 11	01 or 11 [1]
Last in loop 0	10	01 or 11
Last in loop 1	01	10
Last in loops 0 & 1	10	10

[1] Not applicable for single-instruction packets.

The following examples show how to use the Parse field to encode loop packets:

```
{ A   B}:endloop0
  10  11                          // Parse fields of instrs A,B

{ A   B   C}:endloop0
  10  01  11                      // Parse fields of instrs A,B,C

{ A   B   C   D}:endloop0
  10  01  01  11                  // Parse fields of instrs A,B,C,D

{ A   B   C}:endloop1
  01  10  11                      // Parse fields of instrs A,B,C

{ A   B   C   D}:endloop1
  01  10  01  11                  // Parse fields of instrs A,B,C,D

{ A   B   C}:endloop0:endloop1
  10  10  11                      // Parse fields of instrs A,B,C

{ A   B   C   D}:endloop0:endloop1
  10  10  01  11                  // Parse fields of instrs A,B,C,D
```

10.7 Immediate values

To conserve encoding space, the Hexagon processor often stores immediate values in instruction fields that are smaller (in bit size) than the values actually needed in the instruction operation.

When an instruction operates on one of its immediate operands, the processor automatically extends the immediate value to the bit size required by the operation:

- Signed immediate values are sign-extended
- Unsigned immediate values are zero-extended

10.8 Scaled immediate values

To minimize the number of bits in instruction words to store certain immediate values, the Hexagon processor stores the values as scaled immediate values. Use scaled immediate values when an immediate value must represent integral multiples of a power of 2 in a specific range.

For example, consider an instruction operand whose possible values are the following:

-32, -28, -24, -20, -16, -12, -8, -4, 0, 4, 8, 12, 16, 20, 24, 28

Encoding the full range of integers -32...28 normally requires 6 bits. However, if the operand is stored as a scaled immediate, it can first be shifted right by two bits, storing only the four remaining bits in the instruction word. When the operand is fetched from the instruction word, the processor automatically shifts the value left by two bits to recreate the original operand value.

NOTE: The scaled immediate value in the example above is represented notationally as `#s4:2`.

Scaled immediate values commonly encode address offsets that apply to data types of varying size. For example, Table 10-8 shows how to use the byte offsets in immediate-with-offset addressing mode that are stored as 11-bit scaled immediate values. This enables the offsets to span the same range of data elements regardless of the data type.

Table 10-8 Scaled immediate encoding (indirect offsets)

Data type	Offset size (stored)	Scale bits	Offset size (effective)	Offset range (bytes)	Offset range (elements)
byte	11	0	11	-1024 ... 1023	-1024 ... 1023
halfword	11	1	12	-2048 ... 2046	-1024 ... 1023
word	11	2	13	-4096 ... 4092	-1024 ... 1023
doubleword	11	3	14	-8192 ... 8184	-1024 ... 1023

10.9 Constant extenders

To support the use of 32-bit operands in a number of instructions, the Hexagon processor defines constant extenders, which are an instruction word that exists ony to extend the bit range of an immediate or address operand that is contained in an adjacent instruction in a packet.

For example, the absolute addressing mode specifies a 32-bit constant value as the effective address. Instructions using this addressing mode are encoded in a single packet that contains both the normal instruction word and a second word with a constant extender that increases the range of the normal constant operand of the instruction to a full 32 bits.

NOTE: Constant extended operands can encode symbols.

A constant extender is encoded as a 32-bit instruction with the 4-bit ICLASS field set to 0 and the 2-bit Parse field set to its usual value (Section 10.5). The remaining 26 bits in the instruction word store the data bits that are prepended to an operand as small as 6 bits to create a full 32-bit value.

Table 10-9 Constant extender encoding

Bits	Name	Description
31:28	ICLASS	Instruction class = 0000
27:16	Extender high	High 12 bits of 26-bit constant extension
15:14	Parse	Parse bits
13:0	Extender low	Low 14 bits of 26-bit constant extension

Within a packet, a constant extender must be positioned immediately before the instruction that it extends: in terms of memory addresses, the extender word must reside at address (<instr_address> - 4).

The constant extender effectively serves as a prefix for an instruction: it does not execute in a slot, nor does it consume any slot resources. All packets must contain four or fewer words, and the constant extender occupies one word.

If the instruction operand to extend is longer than 6 bits, the overlapping bits in the base instruction must be encoded as zeros. The value in the constant extender always supplies the upper 26 bits.

The Regclass field in Table 10-10 lists the values to set bits [27:24] to in the instruction word to identify the instruction as one that might include a constant extender.

NOTE: When the base instruction encodes two constant operands, the extended immediate is the one specified in the table.

Constant extenders appear in disassembly listings as Hexagon instructions with the name immext.

NOTE: If a constant extender is encoded in a packet for an instruction that does not accept a constant extender, the execution result is undefined. The assembler normally ensures that only valid constant extenders are generated.

Table 10-10 Constant extender instructions

ICLASS	Regclass	Instructions

LD	---1	`Rd = mem{b,ub,h,uh,w,d}(##U32)`
		`if ([!]Pt[.new]) Rd = mem{b,ub,h,uh,w,d} (Rs + ##U32)`
		`// predicated loads`

LD	----	`Rd = mem{b,ub,h,uh,w,d} (Rs + ##U32)`
		`Rd = mem{b,ub,h,uh,w,d} (Re=##U32)`
		`Rd = mem{b,ub,h,uh,w,d} (Rt<<#u2 + ##U32)`
		`if ([!]Pt[.new]) Rd = mem{b,ub,h,uh,w,d} (##U32)`

ST	---0	`mem{b,h,w,d}(##U32) = Rs[.new]  // GP-stores`
		`if ([!]Pt[.new]) mem{b,h,w,d}(Rs + ##U32) = Rt[.new]`
		`// predicated stores`

ST	----	`mem{b,h,w,d}(Rs + ##U32) = Rt[.new]`
		`mem{b,h,w,d}(Rd=##U32) = Rt[.new]`
		`mem{b,h,w,d}(Ru<<#u2 + ##U32) = Rt[.new]`
		`if ([!]Pt[.new]) mem{b,h,w,d}(##U32) = Rt[.new]`

MEMOP	----	`[if [!]Ps] memw(Rs + #u6) = ##U32  // constant store` `memw(Rs + Rt<<#u2) = ##U32  // constant store`
NV	----	`if (cmp.xx(Rs.new,##U32)) jump:hint target`

ALU32	----	`Rd = ##u32`
		`Rdd = combine(Rs,##u32)`
		`Rdd = combine(##u32,Rs)`
		`Rdd = combine(##u32,#s8)`
		`Rdd = combine(#s8,##u32)`
		`Rd = mux (Pu, Rs,##u32)`
		`Rd = mux (Pu, ##u32, Rs)`
		`Rd = mux(Pu,##u32,#s8)`
		`if ([!]Pu[.new]) Rd = add(Rs,##u32)`
		`if ([!]Pu[.new]) Rd = ##u32`
		`Pd = [!]cmp.eq (Rs,##u32)`
		`Pd = [!]cmp.gt (Rs,##u32)`
		`Pd = [!]cmp.gtu (Rs,##u32)`
		`Rd = [!]cmp.eq(Rs,##u32)`
		`Rd = and(Rs,##u32)`
		`Rd = or(Rs,##u32)`
		`Rd = sub(##u32,Rs)`

ALU32	----	`Rd = add(Rs,##s32)`

XTYPE	00--	`Rd = mpyi(Rs,##u32)`
		`Rd += mpyi(Rs,##u32)`
		`Rd -= mpyi(Rs,##u32)`
		`Rx += add(Rs,##u32)`
		`Rx -= add(Rs,##u32)`

ALU32	---- [1]	`Rd = ##u32` `Rd = add(Rs,##s32)`

J	1---	`jump (PC + ##s32)`
		`call (PC + ##s32)`
		`if ([!]Pu) call (PC + ##s32)`

CR	----	`Pd = spNloop0(PC+##s32,Rs/#U10)` `loop0/1 (PC+##s32,#Rs/#U10)`

Table 10-10 Constant extender instructions (cont.)

ICLASS	Regclass	Instructions

XTYPE	1---	`Rd = add(pc,##s32)`
		`Rd = add(##u32,mpyi(Rs,#u6))`
		`Rd = add(##u32,mpyi(Rs,Rt))`
		`Rd = add(Rs,add(Rt,##u32))`
		`Rd = add(Rs,sub(##u32,Rt))`
		`Rd = sub(##u32,add(Rs,Rt))`
		`Rd = or(Rs,and(Rt,##u32))`
		`Rx = add/sub/and/or (##u32,asl/asr/lsr(Rx,#U5))`
		`Rx = add/sub/and/or (##u32,asl/asr/lsr(Rs,Rx))`
		`Rx = add/sub/and/or (##u32,asl/asr/lsr(Rx,Rs))`
		`Pd = cmpb/h.{eq,gt,gtu} (Rs,##u32)`

[1] Constant extension is only for a slot 1 sub-instruction.

Encoding 32-bit address operands in load/stores

Two methods exist for encoding a 32-bit absolute address in a load or store instruction:

- For unconditional load/stores, the GP-relative load/store instruction is used. The assembler encodes the absolute 32-bit address as follows:
  - The upper 26 bits are encoded in a constant extender
  - The lower 6 bits are encoded in the 6 operand bits contained in the GP-relative instruction

  In this case the 32-bit value encoded must be a plain address, and the value stored in the GP register is ignored.

  NOTE: When a constant extender is explicitly specified with a GP-relative load/store, the processor ignores the value in GP and creates the effective address directly from the 32-bit constant value.

- For conditional load/store instructions that have their base address encoded only by a 6-bit immediate operand, a constant extender must be explicitly specified; otherwise, the execution result is undefined. The assembler ensures that these instructions always include a constant extender.

  This case applies also to instructions that use the absolute-set addressing mode or absolute-plus-register-offset addressing mode.

Encoding 32-bit immediate operands

The immediate operands of certain instructions use scaled immediates (Section 10.8) to increase their addressable range. When using constant extenders, scaled immediates are not scaled by the processor. Instead, the assembler must encode the full 32-bit unscaled value as follows:

- The upper 26 bits are encoded in the constant extender
- The lower six 6 bits are encoded in the base instruction in the least-significant bit positions of the immediate operand field.
- Any overlapping bits in the base instruction are encoded as zeros.

Encoding 32-bit jump/call target addresses

When a jump/call has a constant extender, the resulting target address is forced to a 32-bit alignment (bits 1:0 in the address are cleared by hardware). The resulting jump/call operation never causes an alignment violation.

10.10 New-value operands

Instructions that include a new-value register operand specify in their encodings which instruction in the packet has its destination register accessed as the new-value register.

New-value consumers include a 3-bit instruction field named Nt that specifies this information.

- Nt[0] is reserved and must always be encoded as zero. A nonzero value produces undefined results.
- Nt[2:1] encodes the distance (in instructions) from the producer to the consumer, as follows:
  - `Nt[2:1] = 00  // reserved`
  - `Nt[2:1] = 01  // producer is +1 instruction ahead of consumer`
  - `Nt[2:1] = 10  // producer is +2 instructions ahead of consumer`
  - `Nt[2:1] = 11  // producer is +3 instructions ahead of consumer`

"ahead" is defined here as the instruction encoded at a lower memory address than the consumer instruction, not counting empty slots or constant extenders. For example, the following producer/consumer relationship is encoded with `Nt[2:1]` set to 01.

```
...
<producer instruction word>
<consumer constant extender word>
<consumer instruction word>
...
```

NOTE: Instructions with 64-bit register pair destinations cannot produce new-values. The assembler flags this case with an error, as the result is undefined.

10.11 Instruction mapping

Some Hexagon processor instructions are encoded by the assembler as variants of other instructions. This is done for operations that are functionally equivalent to other instructions, but are still defined as separate instructions because of their programming utility as common operations.

Table 10-11 Instructions mapped to other instructions

Instruction	Mapping
`Rd = not(Rs)`	`Rd = sub(#-1,Rs)`
`Rd = neg(Rs)`	`Rd = sub(#0,Rs)`
`Rdd = Rss`	`Rdd = combine(Rss.H32, Rss.L32)`

11 Instruction set

This chapter describes the instruction set for version 6 of the Hexagon processor.

The following information is provided for each instruction:

- Instruction name
- A brief description of the instruction
- A high-level functional description (syntax and behavior) with all possible operand types
- Instruction class and slot information for grouping instructions in packets
- Notes on miscellaneous issues
- Any C intrinsic functions that provide access to the instruction
- Instruction encoding

11.1 ALU32

The ALU32 instruction class includes instructions that perform arithmetic and logical operations on 32-bit data. ALU32 instructions are executable on any slot.

11.1.1 ALU32 ALU

The ALU32 ALU instruction subclass includes instructions that perform arithmetic and logical operations on individual 32-bit items.

Add

Add a source register either to another source register or to a signed 16-bit immediate value. Store result in destination register. Source and destination registers are 32 bits. If the result overflows 32 bits, it wraps around. Optionally saturate result to a signed value between 0x80000000 and 0x7fffffff.

For 64-bit versions of this operation, see the XTYPE add instructions.

Syntax	Behavior
`Rd=add(Rs,#s16)`	`apply_extension(#s);` `Rd=Rs+#s;`
`Rd=add(Rs,Rt)`	`Rd=Rs+Rt;`
`Rd=add(Rs,Rt):sat`	$Rd=sat_{32}(Rs+Rt);$

Class: ALU32 (slots 0,1,2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=add(Rs,#s16)`	`Word32 Q6_R_add_RI(Word32 Rs, Word32 Is16)`
`Rd=add(Rs,Rt)`	`Word32 Q6_R_add_RR(Word32 Rs, Word32 Rt)`
`Rd=add(Rs,Rt):sat`	`Word32 Q6_R_add_RR_sat(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse											d5					
1	0	1	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=add(Rs,#s16)
ICLASS				P	MajOp			MinOp			s5					Parse			t5								d5					
1	1	1	1	0	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=add(Rs,Rt)
1	1	1	1	0	1	1	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=add(Rs,Rt):sat

Field name	Description
MajOp	Major opcode
MinOp	Minor opcode
P	Predicated
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Logical operations

Perform bitwise logical operations (AND, OR, XOR, NOT) either on two source registers or on a source register and a signed 10-bit immediate value. Store the result in the destination register. Source and destination registers are 32 bits.

For 64-bit versions of these operations, see the XTYPE logical instructions.

Syntax	Behavior
`Rd=and(Rs,#s10)`	`apply_extension(#s);` `Rd=Rs&#s;`
`Rd=and(Rs,Rt)`	`Rd=Rs&Rt;`
`Rd=and(Rt,~Rs)`	`Rd = (Rt & ~Rs);`
`Rd=not(Rs)`	`Assembler mapped to: "Rd=sub(#-1,Rs)"`
`Rd=or(Rs,#s10)`	`apply_extension(#s);` `Rd=Rs|#s;`
`Rd=or(Rs,Rt)`	`Rd=Rs|Rt;`
`Rd=or(Rt,~Rs)`	`Rd = (Rt | ~Rs);`
`Rd=xor(Rs,Rt)`	`Rd=Rs^Rt;`

Class: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=and(Rs,#s10)`	`Word32 Q6_R_and_RI(Word32 Rs, Word32 Is10)`
`Rd=and(Rs,Rt)`	`Word32 Q6_R_and_RR(Word32 Rs, Word32 Rt)`
`Rd=and(Rt,~Rs)`	`Word32 Q6_R_and_RnR(Word32 Rt, Word32 Rs)`
`Rd=not(Rs)`	`Word32 Q6_R_not_R(Word32 Rs)`
`Rd=or(Rs,#s10)`	`Word32 Q6_R_or_RI(Word32 Rs, Word32 Is10)`
`Rd=or(Rs,Rt)`	`Word32 Q6_R_or_RR(Word32 Rs, Word32 Rt)`
`Rd=or(Rt,~Rs)`	`Word32 Q6_R_or_RnR(Word32 Rt, Word32 Rs)`
`Rd=xor(Rs,Rt)`	`Word32 Q6_R_xor_RR(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			MinOp			s5					Parse											d5					
0	1	1	1	0	1	1	0	0	0	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=and(Rs,#s10)
0	1	1	1	0	1	1	0	1	0	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=or(Rs,#s10)
ICLASS				P	MajOp			MinOp			s5					Parse			t5								d5					
1	1	1	1	0	0	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=and(Rs,Rt)
1	1	1	1	0	0	0	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=or(Rs,Rt)
1	1	1	1	0	0	0	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=xor(Rs,Rt)
1	1	1	1	0	0	0	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=and(Rt,~Rs)
1	1	1	1	0	0	0	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=or(Rt,~Rs)

Field name	Description
MajOp	Major opcode
MinOp	Minor opcode
Rs	No Rs read
P	Predicated
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Negate

Perform arithmetic negation on a source register. Store result in destination register. Source and destination registers are 32 bits.

For 64-bit and saturating versions of this instruction, see the XTYPE-class negate instructions.

Syntax	Behavior
`Rd=neg(Rs)`	`Assembler mapped to: "Rd=sub(#0,Rs)"`

Class: N/A

Intrinsics

`Rd=neg(Rs)`	`Word32 Q6_R_neg_R(Word32 Rs)`

NOP

Perform no operation (NOP). This instruction is used for padding and alignment.

Within a packet, this instruction can be positioned in any slot 0 through slot 3.

Syntax	Behavior
`nop`	

Class: ALU32 (slots 0,1,2,3)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp											Parse																
0	1	1	1	1	1	1	1	-	-	-	-	-	-	-	-	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	nop

Field name	Description
`MajOp`	Major opcode
`Rs`	No Rs read
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits

Subtract

Subtract a source register from either another source register or from a signed 10-bit immediate value. Store the result in the destination register. Source and destination registers are 32 bits. If the result underflows 32 bits, it wraps around. Optionally saturate the result to a signed value between 0x8000_0000 and 0x7fff_ffff.

For 64-bit versions of this operation, see the XTYPE subtract instructions.

Syntax	Behavior
`Rd=sub(#s10,Rs)`	`apply_extension(#s);` `Rd=#s-Rs;`
`Rd=sub(Rt,Rs)`	`Rd=Rt-Rs;`
`Rd=sub(Rt,Rs):sat`	$Rd=sat_{32}(Rt - Rs);$

Class: ALU32 (slots 0,1,2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=sub(#s10,Rs)`	`Word32 Q6_R_sub_IR(Word32 Is10, Word32 Rs)`
`Rd=sub(Rt,Rs)`	`Word32 Q6_R_sub_RR(Word32 Rt, Word32 Rs)`
`Rd=sub(Rt,Rs):sat`	`Word32 Q6_R_sub_RR_sat(Word32 Rt, Word32 Rs)`

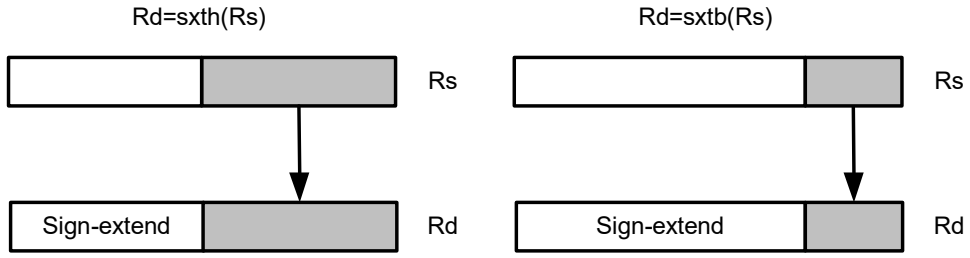
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			MinOp			s5					Parse												d5					
0	1	1	1	0	1	1	0	0	1	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=sub(#s10,Rs)
ICLASS				P	MajOp			MinOp			s5					Parse			t5								d5					
1	1	1	1	0	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt,Rs)
1	1	1	1	0	1	1	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt,Rs):sat

Field name	Description
`MajOp`	Major opcode
`MinOp`	Minor opcode
`Rs`	No Rs read
`P`	Predicated
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Sign extend

Sign-extend the least-significant byte or halfword from the source register and place the 32-bit result in the destination register.



Syntax	Behavior
`Rd=sxtb(Rs)`	`Rd = sxt8->32(Rs);`
`Rd=sxth(Rs)`	`Rd = sxt16->32(Rs);`

Class: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=sxtb(Rs)`	`Word32 Q6_R_sxtb_R(Word32 Rs)`
`Rd=sxth(Rs)`	`Word32 Q6_R_sxth_R(Word32 Rs)`

Encoding

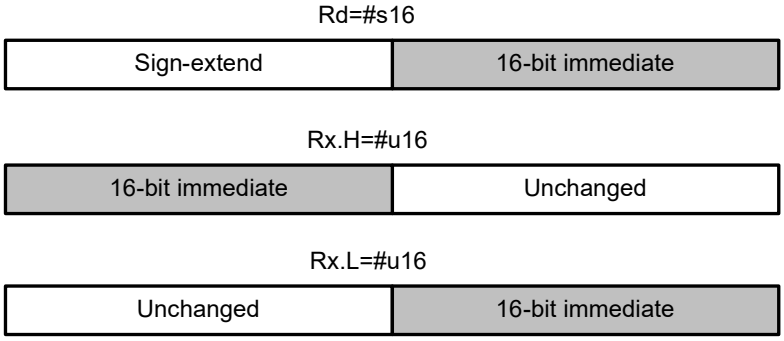
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			MinOp			s5					Parse		C									d5					
0	1	1	1	0	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=sxtb(Rs)
0	1	1	1	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=sxth(Rs)

Field name	Description
`MajOp`	Major opcode
`MinOp`	Minor opcode
`Rs`	No Rs read
`C`	Conditional
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s

Transfer immediate

Assign an immediate value to a 32-bit destination register.

Two types of assignment are supported. The first type of assignment sign-extends a 16-bit signed immediate value to 32 bits. The second type assigns a 16-bit unsigned immediate value to either the upper or lower 16 bits of the destination register, leaving the other 16 bits unchanged.



Syntax	Behavior
`Rd=#s16`	`apply_extension(#s);` `Rd=#s;`

`Rdd=#s8`	`if ("#s8<0") {`
	`  Assembler mapped to: "Rdd=combine(#-1,#s8)";`
	`} else {`
	`  Assembler mapped to: "Rdd=combine(#0,#s8)";`
	`}`

`Rx.[HL]=#u16`	`Rx.h[01]=#u;`

Class: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=#s16`	`Word32 Q6_R_equals_I(Word32 Is16)`
`Rdd=#s8`	`Word64 Q6_P_equals_I(Word32 Is8)`
`Rx.H=#u16`	`Word32 Q6_Rh_equals_I(Word32 Rx, Word32 Iu16)`
`Rx.L=#u16`	`Word32 Q6_Rl_equals_I(Word32 Rx, Word32 Iu16)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			MinOp			x5					Parse																
0	1	1	1	0	0	0	1	i	i	1	x	x	x	x	x	P	P	i	i	i	i	i	i	i	i	i	i	i	i	i	Rx.L=#u16
0	1	1	1	0	0	1	0	i	i	1	x	x	x	x	x	P	P	i	i	i	i	i	i	i	i	i	i	i	i	i	Rx.H=#u16
ICLASS				Rs	MajOp			MinOp								Parse											d5					
0	1	1	1	1	0	0	0	i	i	-	i	i	i	i	i	P	P	i	i	i	i	i	i	i	i	i	d	d	d	d	d	Rd=#s16

Field name	Description
MajOp	Major opcode
MinOp	Minor opcode
Rs	No Rs read
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
x5	Field to encode register x

Transfer register

Transfer a source register to a destination register. Source and destination registers are either 32 bits or 64 bits.

Syntax	Behavior
`Rd = Rs`	`Rd = Rs;`
`Rdd = Rss`	`Assembler mapped to: "Rdd=combine(Rss.H32,Rss.L32)"`

Class: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=Rs`	`Word32 Q6_R_equals_R(Word32 Rs)`
`Rdd=Rss`	`Word64 Q6_P_equals_P(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp				MinOp		s5					Parse		C									d5					
0	1	1	1	0	0	0	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=Rs

Field name	Description
`MajOp`	Major opcode
`MinOp`	Minor opcode
`Rs`	No Rs read
`C`	Conditional
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s

Vector add halfwords

Add the two 16-bit halfwords of Rs to the two 16-bit halfwords of Rt. The results are optionally saturated to signed or unsigned 16-bit values.

Syntax	Behavior

`Rd=vaddh(Rs,Rt)[:sat]`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=[sat16](Rs.h[i]+Rt.h[i]);`
	`}`

`Rd=vadduh(Rs,Rt):sat`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=usat16(Rs.uh[i]+Rt.uh[i]);`
	`}`

Class: ALU32 (slots 0,1,2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=vaddh(Rs,Rt)`	`Word32 Q6_R_vaddh_RR(Word32 Rs, Word32 Rt)`
`Rd=vaddh(Rs,Rt):sat`	`Word32 Q6_R_vaddh_RR_sat(Word32 Rs, Word32 Rt)`
`Rd=vadduh(Rs,Rt):sat`	`Word32 Q6_R_vadduh_RR_sat(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				P	MajOp			MinOp			s5					Parse			t5								d5						
1	1	1	1	0	1	1	0	0	0	0	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	d	d	d	d	d	Rd=vaddh(Rs,Rt)
1	1	1	1	0	1	1	0	0	0	1	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	d	d	d	d	d	Rd=vaddh(Rs,Rt):sat
1	1	1	1	0	1	1	0	0	1	1	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	d	d	d	d	d	Rd=vadduh(Rs,Rt):sat

Field name	Description
MajOp	Major opcode
MinOp	Minor opcode
P	Predicated
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector average halfwords

The `vavgh` instruction Vadds the two 16-bit halfwords of Rs to the two 16-bit halfwords of Rd, and shifts the result right by one bit. Optionally, a rounding constant is added before shifting.

The `vnavgh` instruction subtracts the two 16-bit halfwords of Rt from the two 16-bit halfwords of Rs, and shifts the result right by one bit. For vector negative average with rounding, see the XTYPE `vnavgh` (Vector average halfwords) instruction.

Syntax	Behavior

`Rd=vavgh(Rs,Rt)`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=((Rs.h[i]+Rt.h[i])>>1);`
	`}`

`Rd=vavgh(Rs,Rt):rnd`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=((Rs.h[i]+Rt.h[i]+1)>>1);`
	`}`

`Rd=vnavgh(Rt,Rs)`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=((Rt.h[i]-Rs.h[i])>>1);`
	`}`

Class: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=vavgh(Rs,Rt)`	`Word32 Q6_R_vavgh_RR(Word32 Rs, Word32 Rt)`
`Rd=vavgh(Rs,Rt):rnd`	`Word32 Q6_R_vavgh_RR_rnd(Word32 Rs, Word32 Rt)`
`Rd=vnavgh(Rt,Rs)`	`Word32 Q6_R_vnavgh_RR(Word32 Rt, Word32 Rs)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				P	MajOp			MinOp			s5					Parse			t5								d5					
1	1	1	1	0	1	1	1	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=vavgh(Rs,Rt)
1	1	1	1	0	1	1	1	-	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=vavgh(Rs,Rt):rnd
1	1	1	1	0	1	1	1	-	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=vnavgh(Rt,Rs)

Field name	Description
`MajOp`	Major opcode
`MinOp`	Minor opcode
`P`	Predicated
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector subtract halfwords

Subtract each of the two halfwords in 32-bit vector Rs from the corresponding halfword in vector Rt. Optionally saturate each 16-bit addition to either a signed or unsigned 16-bit value.

Applying saturation to the `vsubh` instruction clamps the result to the signed range 0x8000 to 0x7fff, whereas applying saturation to `vsubuh` ensures that the unsigned result is in the range 0 to 0xffff. When saturation is not needed, `vsubh` should be used.

Syntax	Behavior

`Rd=vsubh(Rt,Rs)[:sat]`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=[sat16](Rt.h[i]-Rs.h[i]);`
	`}`

`Rd=vsubuh(Rt,Rs):sat`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=usat16(Rt.uh[i]-Rs.uh[i]);`
	`}`

Class: ALU32 (slots 0,1,2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=vsubh(Rt,Rs)`	`Word32 Q6_R_vsubh_RR(Word32 Rt, Word32 Rs)`
`Rd=vsubh(Rt,Rs):sat`	`Word32 Q6_R_vsubh_RR_sat(Word32 Rt, Word32 Rs)`
`Rd=vsubuh(Rt,Rs):sat`	`Word32 Q6_R_vsubuh_RR_sat(Word32 Rt, Word32 Rs)`

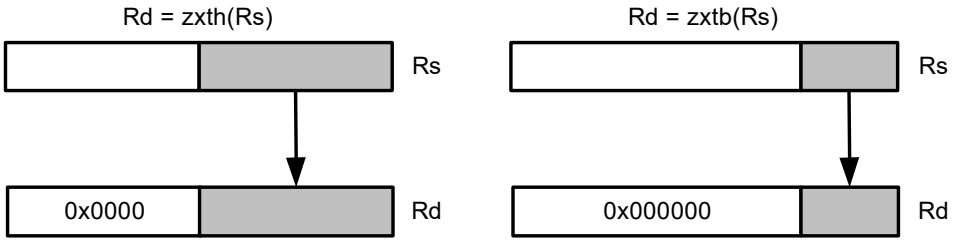
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				P	MajOp			MinOp			s5					Parse			t5								d5					
1	1	1	1	0	1	1	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=vsubh(Rt,Rs)
1	1	1	1	0	1	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=vsubh(Rt,Rs):sat
1	1	1	1	0	1	1	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=vsubuh(Rt,Rs):sat

Field name	Description
`MajOp`	Major opcode
`MinOp`	Minor opcode
`P`	Predicated
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Zero extend

Zero-extend the least significant byte or halfword from Rs and place the 32-bit result in Rd.



Syntax	Behavior
`Rd=zxtb(Rs)`	`Assembler mapped to: "Rd=and(Rs,#255)"`
`Rd=zxth(Rs)`	`Rd = zxt16->32(Rs);`

Class: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=zxtb(Rs)`	`Word32 Q6_R_zxtb_R(Word32 Rs)`
`Rd=zxth(Rs)`	`Word32 Q6_R_zxth_R(Word32 Rs)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			MinOp			s5					Parse		C									d5					
0	1	1	1	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=zxth(Rs)

Field name	Description
`MajOp`	Major opcode
`MinOp`	Minor opcode
`Rs`	No Rs read
`C`	Conditional
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s

11.1.2 ALU32 PERM

The ALU32 PERM instruction subclass includes instructions that rearrange or perform format conversion on vector data types.

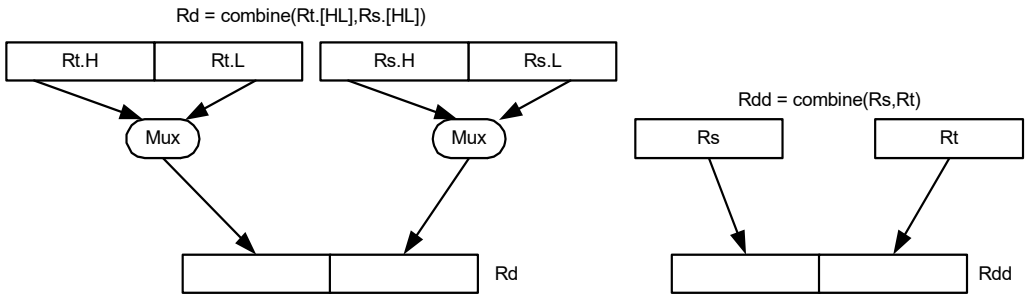
Combine words into doubleword

Combine halfwords or words into larger values.

In a halfword combine, either the high or low halfword of the first source register is transferred to the most-significant halfword of the destination register, while either the high or low halfword of the second source register is transferred to the least-significant halfword of the destination register. Source and destination registers are 32 bits.

In a word combine, the first source register is transferred to the most-significant word of the destination register, while the second source register is transferred to the least-significant word of the destination register. Source registers are 32 bits and the destination register is 64 bits.

In a variant of word combine, signed 8-bit immediate values (instead of registers) are transferred to the most- and least-significant words of the 64-bit destination register. Optionally one of the immediate values can be 32 bits.



Syntax	Behavior
`Rd=combine(Rt.[HL],Rs.[HL])`	`Rd = (Rt.uh[01]<<16) | Rs.uh[01];`

`Rdd=combine(#s8,#S8)`	`apply_extension(#s);`
	`Rdd.w[0]=#S;`
	`Rdd.w[1]=#s;`

`Rdd=combine(#s8,#U6)`	`apply_extension(#U);`
	`Rdd.w[0]=#U;`
	`Rdd.w[1]=#s;`

`Rdd=combine(#s8,Rs)`	`apply_extension(#s);`
	`Rdd.w[0]=Rs;`
	`Rdd.w[1]=#s;`

`Rdd=combine(Rs,#s8)`	`apply_extension(#s);`
	`Rdd.w[0]=#s;`
	`Rdd.w[1]=Rs;`

`Rdd=combine(Rs,Rt)`	`Rdd.w[0]=Rt;` `Rdd.w[1]=Rs;`

Class: ALU32 (slots 0,1,2,3)

Intrinsics

Rd=combine(Rt.H,Rs.H)	Word32 Q6_R_combine_RhRh(Word32 Rt, Word32 Rs)
Rd=combine(Rt.H,Rs.L)	Word32 Q6_R_combine_RhRl(Word32 Rt, Word32 Rs)
Rd=combine(Rt.L,Rs.H)	Word32 Q6_R_combine_RlRh(Word32 Rt, Word32 Rs)
Rd=combine(Rt.L,Rs.L)	Word32 Q6_R_combine_RlRl(Word32 Rt, Word32 Rs)
Rdd=combine(#s8,#S8)	Word64 Q6_P_combine_II(Word32 Is8, Word32 IS8)
Rdd=combine(#s8,Rs)	Word64 Q6_P_combine_IR(Word32 Is8, Word32 Rs)
Rdd=combine(Rs,#s8)	Word64 Q6_P_combine_RI(Word32 Rs, Word32 Is8)
Rdd=combine(Rs,Rt)	Word64 Q6_P_combine_RR(Word32 Rs, Word32 Rt)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			MinOp			s5					Parse											d5					
0	1	1	1	0	0	1	1	-	0	0	s	s	s	s	s	P	P	1	i	i	i	i	i	i	i	i	d	d	d	d	d	Rdd=combine(Rs,#s8)
0	1	1	1	0	0	1	1	-	0	1	s	s	s	s	s	P	P	1	i	i	i	i	i	i	i	i	d	d	d	d	d	Rdd=combine(#s8,Rs)
ICLASS				Rs	MajOp			MinOp								Parse											d5					
0	1	1	1	1	1	0	0	0	I	I	I	I	I	I	I	P	P	I	i	i	i	i	i	i	i	i	d	d	d	d	d	Rdd=combine(#s8,#S8)
0	1	1	1	1	1	0	0	1	-	-	I	I	I	I	I	P	P	I	i	i	i	i	i	i	i	i	d	d	d	d	d	Rdd=combine(#s8,#U6)
ICLASS				P	MajOp			MinOp			s5					Parse			t5								d5					
1	1	1	1	0	0	1	1	1	0	0	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	d	d	d	d	d	Rd=combine(Rt.H,Rs.H)
1	1	1	1	0	0	1	1	1	0	1	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	d	d	d	d	d	Rd=combine(Rt.H,Rs.L)
1	1	1	1	0	0	1	1	1	1	0	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	d	d	d	d	d	Rd=combine(Rt.L,Rs.H)
1	1	1	1	0	0	1	1	1	1	1	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	d	d	d	d	d	Rd=combine(Rt.L,Rs.L)
1	1	1	1	0	1	0	1	0	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	d	d	d	d	d	Rdd=combine(Rs,Rt)

Field name	Description
MajOp	Major opcode
MinOp	Minor opcode
Rs	No Rs read
P	Predicated
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Mux

Select between two source registers based on the LSB of a predicate register. If the bit is 1, transfer the first source register to the destination register; otherwise, transfer the second source register. Source and destination registers are 32 bits.

In a variant of mux, signed 8-bit immediate values can be used instead of registers for either or both source operands.

For 64-bit versions of this instruction, see the XTYPE vmux instruction.

Syntax	Behavior

`Rd=mux(Pu,#s8,#S8)`	`PREDUSE_TIMING;`
	`apply_extension(#s);`
	`Rd = (Pu[0] ? #s : #S);`

`Rd=mux(Pu,#s8,Rs)`	`PREDUSE_TIMING;`
	`apply_extension(#s);`
	`Rd = (Pu[0] ? #s : Rs);`

`Rd=mux(Pu,Rs,#s8)`	`PREDUSE_TIMING;`
	`apply_extension(#s);`
	`Rd = (Pu[0] ? Rs : #s);`

`Rd=mux(Pu,Rs,Rt)`	`PREDUSE_TIMING;` `Rd = (Pu[0] ? Rs : Rt);`

Class: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=mux(Pu,#s8,#S8)`	`Word32 Q6_R_mux_pII(Byte Pu, Word32 Is8, Word32 IS8)`
`Rd=mux(Pu,#s8,Rs)`	`Word32 Q6_R_mux_pIR(Byte Pu, Word32 Is8, Word32 Rs)`
`Rd=mux(Pu,Rs,#s8)`	`Word32 Q6_R_mux_pRI(Byte Pu, Word32 Rs, Word32 Is8)`
`Rd=mux(Pu,Rs,Rt)`	`Word32 Q6_R_mux_pRR(Byte Pu, Word32 Rs, Word32 Rt)`

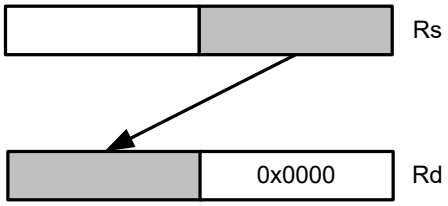
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp				u2		s5					Parse		d5														
0	1	1	1	0	0	1	1	0	u	u	s	s	s	s	s	P	P	0	i	i	i	i	i	i	i	i	d	d	d	d	d	Rd=mux(Pu,Rs,#s8)
0	1	1	1	0	0	1	1	1	u	u	s	s	s	s	s	P	P	0	i	i	i	i	i	i	i	i	d	d	d	d	d	Rd=mux(Pu,#s8,Rs)
ICLASS				Rs			u2									Parse		d5														
0	1	1	1	1	0	1	u	u	l	l	l	l	l	l	l	P	P	l	i	i	i	i	i	i	i	i	d	d	d	d	d	Rd=mux(Pu,#s8,#S8)
ICLASS				P	MajOp						s5					Parse			t5					u2			d5					
1	1	1	1	0	1	0	0	-	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	-	u	u	d	d	d	d	d	Rd=mux(Pu,Rs,Rt)

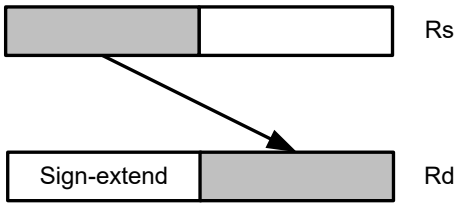
Field name	Description
`MajOp`	Major opcode
`MinOp`	Minor opcode
`Rs`	No Rs read
`P`	Predicated
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`u2`	Field to encode register u

Shift word by 16

The `aslh` instruction performs an arithmetic left shift of the 32-bit source register by 16 bits (one halfword). The lower 16 bits of the destination are zero-filled.



The `asrh` instruction performs an arithmetic right shift of the 32-bit source register by 16 bits (one halfword). The upper 16 bits of the destination are sign-extended.



Syntax	Behavior
`Rd=aslh(Rs)`	`Rd=Rs<<16;`
`Rd=asrh(Rs)`	`Rd=Rs>>16;`

Class: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=aslh(Rs)`	`Word32 Q6_R_aslh_R(Word32 Rs)`
`Rd=asrh(Rs)`	`Word32 Q6_R_asrh_R(Word32 Rs)`

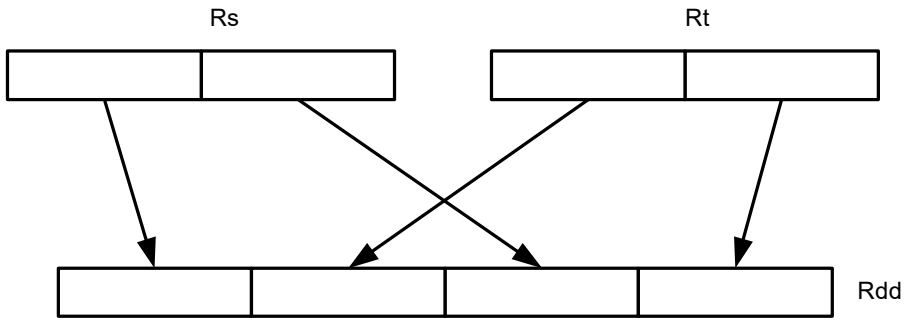
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp				MinOp		s5					Parse		C									d5					
0	1	1	1	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=aslh(Rs)
0	1	1	1	0	0	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=asrh(Rs)

Field name	Description
`MajOp`	Major opcode
`MinOp`	Minor opcode
`Rs`	No Rs read
`C`	Conditional
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s

Pack high and low halfwords

Pack together the most-significant halfwords from Rs and Rt into the most-significant word of register pair Rdd, and the least-significant halfwords from Rs and Rt into the least-significant halfword of Rdd.



Syntax	Behavior

`Rdd=packhl(Rs,Rt)`	`Rdd.h[0]=Rt.h[0];`
	`Rdd.h[1]=Rs.h[0];`
	`Rdd.h[2]=Rt.h[1];`
	`Rdd.h[3]=Rs.h[1];`

Class: ALU32 (slots 0,1,2,3)

Intrinsics

`Rdd=packhl(Rs,Rt)`	`Word64 Q6_P_packhl_RR(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				P	MajOp			MinOp			s5					Parse			t5								d5					
1	1	1	1	0	1	0	1	1	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	d	d	d	d	d	Rdd=packhl(Rs,Rt)

Field name	Description
MajOp	Major opcode
MinOp	Minor opcode
P	Predicated
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

11.1.3 ALU32 PRED

The ALU32 PRED instruction subclass includes instructions that perform conditional arithmetic and logical operations based on the values stored in a predicate register, and produce predicate results. They are executable on any slot.

Conditional add

If the least-significant bit of predicate Pu is set, add a 32-bit source register to either another register or an immediate value. The result is placed in 32-bit destination register. If the predicate is false, the instruction does nothing.

Syntax	Behavior
`if ([!]Pu[.new]) Rd=add(Rs,#s8)`	`if([!]Pu[.new][0]){ apply_extension(#s); Rd=Rs+#s; } else { NOP; }`
`if ([!]Pu[.new]) Rd=add(Rs,Rt)`	`if([!]Pu[.new][0]){ Rd=Rs+Rt; } else { NOP; }`

Class: ALU32 (slots 0,1,2,3)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			PS	u2		s5					Parse		DN									d5					
0	1	1	1	0	1	0	0	0	**u**	**u**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=add(Rs,#s8)
0	1	1	1	0	1	0	0	0	**u**	**u**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=add(Rs,#s8)
0	1	1	1	0	1	0	0	1	**u**	**u**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=add(Rs,#s8)
0	1	1	1	0	1	0	0	1	**u**	**u**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=add(Rs,#s8)
ICLASS				P	MajOp			MinOp			s5					Parse		DN	t5					PS	u2		d5					
1	1	1	1	1	0	1	1	0	-	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=add(Rs,Rt)
1	1	1	1	1	0	1	1	0	-	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=add(Rs,Rt)
1	1	1	1	1	0	1	1	0	-	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=add(Rs,Rt)
1	1	1	1	1	0	1	1	0	-	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=add(Rs,Rt)

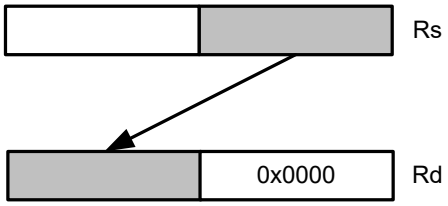
Field name	Description
MajOp	Major opcode
MinOp	Minor opcode
Rs	No Rs read
DN	Dot-new
PS	Predicate sense

Field name	Description
P	Predicated
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
u2	Field to encode register u

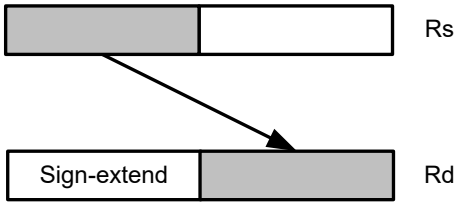
Conditional shift halfword

Conditionally shift a halfword.

The `aslh` instruction performs an arithmetic left shift of the 32-bit source register by 16 bits (one halfword). The lower 16 bits of the destination are zero-filled.



The `asrh` instruction performs an arithmetic right shift of the 32-bit source register by 16 bits (one halfword). The upper 16 bits of the destination are sign-extended.



Syntax	Behavior

`if ([!]Pu[.new]) Rd=aslh(Rs)`	`if([!]Pu[.new][0]){`
	`  Rd=Rs<<16;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pu[.new]) Rd=asrh(Rs)`	`if([!]Pu[.new][0]){`
	`  Rd=Rs>>16;`
	`} else {`
	`  NOP;`
	`}`

Class: ALU32 (slots 0,1,2,3)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			MinOp			s5					Parse		C		S	dn	u2					d5					
0	1	1	1	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**-**	0	0	**u**	**u**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=aslh(Rs)
0	1	1	1	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**-**	0	1	**u**	**u**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=aslh(Rs)
0	1	1	1	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**-**	1	0	**u**	**u**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=aslh(Rs)
0	1	1	1	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**-**	1	1	**u**	**u**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=aslh(Rs)
0	1	1	1	0	0	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**-**	0	0	**u**	**u**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=asrh(Rs)
0	1	1	1	0	0	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**-**	0	1	**u**	**u**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=asrh(Rs)
0	1	1	1	0	0	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**-**	1	0	**u**	**u**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=asrh(Rs)
0	1	1	1	0	0	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**-**	1	1	**u**	**u**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=asrh(Rs)

Field name	Description
MajOp	Major opcode
MinOp	Minor opcode
Rs	No Rs read
C	Conditional
S	Predicate sense
dn	Dot-new
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
u2	Field to encode register u

Conditional combine

If the least-significant bit of predicate Pu is set, the most-significant word of destination Rdd is taken from the first source register Rs, while the least-significant word is taken from the second source register Rt. If the predicate is false, this instruction does nothing.

Syntax

```
if ([!]Pu[.new])
Rdd=combine(Rs,Rt)
```

Behavior

```
if ([!]Pu[.new][0]) {
    Rdd.w[0]=Rt;
    Rdd.w[1]=Rs;
} else {
    NOP;
}
```

Class: ALU32 (slots 0,1,2,3)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				P	MajOp			MinOp			s5					Parse		DN	t5						PS	u2		d5					
1	1	1	1	1	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu) Rdd=combine(Rs,Rt)
1	1	1	1	1	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rdd=combine(Rs,Rt)
1	1	1	1	1	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rdd=combine(Rs,Rt)
1	1	1	1	1	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rdd=combine(Rs,Rt)

Field name	Description
DN	Dot-new
MajOp	Major opcode
MinOp	Minor opcode
P	Predicated
PS	Predicate sense
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
u2	Field to encode register u

Conditional logical operations

If the least-significant bit of predicate Pu is set, do a logical operation on the source values. The result is placed in 32-bit destination register. If the predicate is false, the instruction does nothing.

Syntax	Behavior
`if ([!]Pu[.new]) Rd=and(Rs,Rt)`	`if([!]Pu[.new][0]){ Rd=Rs&Rt; } else { NOP; }`
`if ([!]Pu[.new]) Rd=or(Rs,Rt)`	`if([!]Pu[.new][0]){ Rd=Rs|Rt; } else { NOP; }`
`if ([!]Pu[.new]) Rd=xor(Rs,Rt)`	`if([!]Pu[.new][0]){ Rd=Rs^Rt; } else { NOP; }`

Class: ALU32 (slots 0,1,2,3)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				P	MajOp			MinOp			s5					Parse		DN	t5						PS	u2		d5					
1	1	1	1	1	0	0	1	-	0	0	s	s	s	s	s	P	P	0	t	t	t	t	t	0	u	u	d	d	d	d	d	if (Pu) Rd=and(Rs,Rt)
1	1	1	1	1	0	0	1	-	0	0	s	s	s	s	s	P	P	0	t	t	t	t	t	1	u	u	d	d	d	d	d	if (!Pu) Rd=and(Rs,Rt)
1	1	1	1	1	0	0	1	-	0	0	s	s	s	s	s	P	P	1	t	t	t	t	t	0	u	u	d	d	d	d	d	if (Pu.new) Rd=and(Rs,Rt)
1	1	1	1	1	0	0	1	-	0	0	s	s	s	s	s	P	P	1	t	t	t	t	t	1	u	u	d	d	d	d	d	if (!Pu.new) Rd=and(Rs,Rt)
1	1	1	1	1	0	0	1	-	0	1	s	s	s	s	s	P	P	0	t	t	t	t	t	0	u	u	d	d	d	d	d	if (Pu) Rd=or(Rs,Rt)
1	1	1	1	1	0	0	1	-	0	1	s	s	s	s	s	P	P	0	t	t	t	t	t	1	u	u	d	d	d	d	d	if (!Pu) Rd=or(Rs,Rt)
1	1	1	1	1	0	0	1	-	0	1	s	s	s	s	s	P	P	1	t	t	t	t	t	0	u	u	d	d	d	d	d	if (Pu.new) Rd=or(Rs,Rt)
1	1	1	1	1	0	0	1	-	0	1	s	s	s	s	s	P	P	1	t	t	t	t	t	1	u	u	d	d	d	d	d	if (!Pu.new) Rd=or(Rs,Rt)
1	1	1	1	1	0	0	1	-	1	1	s	s	s	s	s	P	P	0	t	t	t	t	t	0	u	u	d	d	d	d	d	if (Pu) Rd=xor(Rs,Rt)
1	1	1	1	1	0	0	1	-	1	1	s	s	s	s	s	P	P	0	t	t	t	t	t	1	u	u	d	d	d	d	d	if (!Pu) Rd=xor(Rs,Rt)
1	1	1	1	1	0	0	1	-	1	1	s	s	s	s	s	P	P	1	t	t	t	t	t	0	u	u	d	d	d	d	d	if (Pu.new) Rd=xor(Rs,Rt)
1	1	1	1	1	0	0	1	-	1	1	s	s	s	s	s	P	P	1	t	t	t	t	t	1	u	u	d	d	d	d	d	if (!Pu.new) Rd=xor(Rs,Rt)

Field name	Description
DN	Dot-new
MajOp	Major opcode
MinOp	Minor opcode
P	Predicated
PS	Predicate sense
ICLASS	Instruction class
Parse	Packet/loop parse bits

Field name	Description
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
u2	Field to encode register u

Conditional subtract

If the least-significant bit of predicate Pu is set, subtract a 32-bit source register Rt from register Rs. The result is placed in a 32-bit destination register. If the predicate is false, the instruction does nothing.

Syntax

```
if ([!]Pu[.new])
Rd=sub(Rt,Rs)
```

Behavior

```
if([!]Pu[.new][0]){
    Rd=Rt-Rs;
} else {
    NOP;
}
```

Class: ALU32 (slots 0,1,2,3)

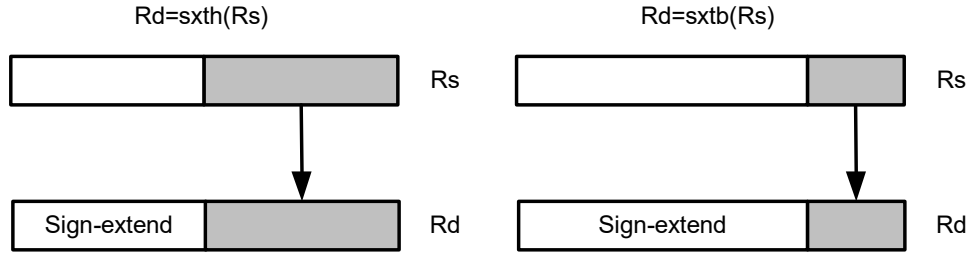
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				P	MajOp			MinOp			s5					Parse		DN	t5						PS	u2		d5					
1	1	1	1	1	0	1	1	0	-	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=sub(Rt,Rs)
1	1	1	1	1	0	1	1	0	-	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=sub(Rt,Rs)
1	1	1	1	1	0	1	1	0	-	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=sub(Rt,Rs)
1	1	1	1	1	0	1	1	0	-	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=sub(Rt,Rs)

Field name	Description
DN	Dot-new
MajOp	Major opcode
MinOp	Minor opcode
P	Predicated
PS	Predicate sense
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
u2	Field to encode register u

Conditional sign extend

Conditionally sign-extend the least-significant byte or halfword from Rs and put the 32-bit result in Rd.



Syntax	Behavior

`if ([!]Pu[.new]) Rd=sxtb(Rs)`	`if([!]Pu[.new][0]){`
	`  Rd=sxt8->32(Rs);`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pu[.new]) Rd=sxth(Rs)`	`if([!]Pu[.new][0]){`
	`  Rd=sxt16->32(Rs);`
	`} else {`
	`  NOP;`
	`}`

Class: ALU32 (slots 0,1,2,3)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			MinOp			s5					Parse		C		S	dn	u2					d5					
0	1	1	1	0	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	0	0	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=sxtb(Rs)
0	1	1	1	0	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	0	1	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=sxtb(Rs)
0	1	1	1	0	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	1	0	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=sxtb(Rs)
0	1	1	1	0	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	1	1	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=sxtb(Rs)
0	1	1	1	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	0	0	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=sxth(Rs)
0	1	1	1	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	0	1	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=sxth(Rs)
0	1	1	1	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	1	0	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=sxth(Rs)
0	1	1	1	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	1	1	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=sxth(Rs)

Field name	Description
MajOp	Major opcode
MinOp	Minor opcode
Rs	No Rs read
C	Conditional
S	Predicate sense
dn	Dot-new
ICLASS	Instruction class

Field name	Description
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
u2	Field to encode register u

Conditional transfer

If the LSB of predicate Pu is set, transfer register Rs or a signed immediate into destination Rd. If the predicate is false, this instruction does nothing.

Syntax	Behavior

`if ([!]Pu[.new]) Rd=#s12`	`apply_extension(#s);`
	`if ([!]Pu[.new][0]) Rd=#s;`
	`else NOP;`

`if ([!]Pu[.new]) Rd=Rs`	`Assembler mapped to: "if ([!]Pu[.new]) Rd=add(Rs,#0)"`
`if ([!]Pu[.new]) Rdd=Rss`	`Assembler mapped to: "if ([!]Pu[.new]) Rdd=combine(Rss.H32,Rss.L32)"`

Class: ALU32 (slots 0, 1, 2, 3)

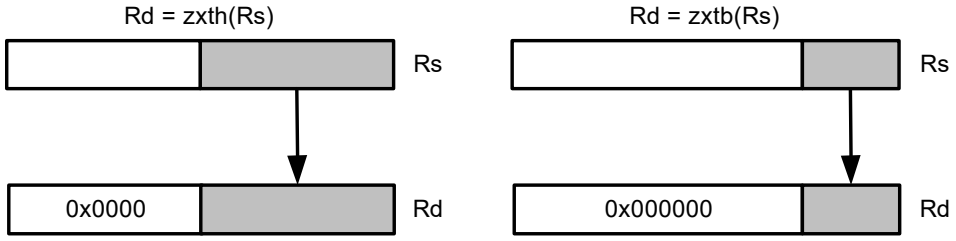
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			PS	u2							Parse		DN									d5					
0	1	1	1	1	1	1	0	0	**u**	**u**	0	**i**	**i**	**i**	**i**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=#s12
0	1	1	1	1	1	1	0	0	**u**	**u**	0	**i**	**i**	**i**	**i**	**P**	**P**	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=#s12
0	1	1	1	1	1	1	0	1	**u**	**u**	0	**i**	**i**	**i**	**i**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=#s12
0	1	1	1	1	1	1	0	1	**u**	**u**	0	**i**	**i**	**i**	**i**	**P**	**P**	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=#s12

Field name	Description
MajOp	Major opcode
MinOp	Minor opcode
Rs	No Rs read
DN	Dot-new
PS	Predicate sense
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
u2	Field to encode register u

Conditional zero extend

Conditionally zero-extend the least-significant byte or halfword from Rs and put the 32-bit result in Rd.



Syntax	Behavior

`if ([!]Pu[.new]) Rd=zxtb(Rs)`	`if([!]Pu[.new][0]){`
	`  Rd=zxt`$_{8->32}$`(Rs);`
	`} else {`
	`  NOP;}`

`if ([!]Pu[.new]) Rd=zxth(Rs)`	`if([!]Pu[.new][0]){`
	`  Rd=zxt`$_{16->32}$`(Rs);`
	`} else {`
	`  NOP;}`

Class: ALU32 (slots 0,1,2,3)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			MinOp			s5					Parse		C		S	dn	u2					d5					
0	1	1	1	0	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	0	0	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=zxtb(Rs)
0	1	1	1	0	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	0	1	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=zxtb(Rs)
0	1	1	1	0	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	1	0	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=zxtb(Rs)
0	1	1	1	0	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	1	1	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=zxtb(Rs)
0	1	1	1	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	0	0	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=zxth(Rs)
0	1	1	1	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	0	1	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=zxth(Rs)
0	1	1	1	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	1	0	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=zxth(Rs)
0	1	1	1	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	1	1	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=zxth(Rs)

Field name	Description
MajOp	Major opcode
MinOp	Minor opcode
Rs	No Rs read
C	Conditional
S	Predicate sense
dn	Dot-new
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
u2	Field to encode register u

Compare

The register form compares two 32-bit registers for unsigned greater than, greater than, or equal.

The immediate form compares a register against a signed or unsigned immediate value. The 8-bit predicate register Pd is set to all 1's or all 0's depending on the result. For 64-bit versions of this instruction, see the XTYPE compare instructions.

Syntax	Behavior
`Pd=[!]cmp.eq(Rs,#s10)`	`apply_extension(#s);` `Pd=Rs[!]=#s ? 0xff : 0x00;`
`Pd=[!]cmp.eq(Rs,Rt)`	`Pd=Rs[!]=Rt ? 0xff : 0x00;`
`Pd=[!]cmp.gt(Rs,#s10)`	`apply_extension(#s);` `Pd=Rs<=#s ? 0xff : 0x00;`
`Pd=[!]cmp.gt(Rs,Rt)`	`Pd=Rs<=Rt ? 0xff : 0x00;`
`Pd=[!]cmp.gtu(Rs,#u9)`	`apply_extension(#u);` `Pd=Rs.uw[0]<=#u.uw[0] ? 0xff : 0x00;`
`Pd=[!]cmp.gtu(Rs,Rt)`	`Pd=Rs.uw[0]<=Rt.uw[0] ? 0xff : 0x00;`
`Pd=cmp.ge(Rs,#s8)`	`Assembler mapped to: "Pd=cmp.gt(Rs,#s8-1)"`

`Pd=cmp.geu(Rs,#u8)`	`if ("#u8==0") {`
	`  Assembler mapped to: "Pd=cmp.eq(Rs,Rs)";`
	`} else {`
	`  Assembler mapped to: "Pd=cmp.gtu(Rs,#u8-1)";`
	`}`

`Pd=cmp.lt(Rs,Rt)`	`Assembler mapped to: "Pd=cmp.gt(Rt,Rs)"`
`Pd=cmp.ltu(Rs,Rt)`	`Assembler mapped to: "Pd=cmp.gtu(Rt,Rs)"`

Class: ALU32 (slots 0,1,2,3)

Intrinsics

`Pd=!cmp.eq(Rs,#s10)`	`Byte Q6_p_not_cmp_eq_RI(Word32 Rs, Word32 Is10)`
`Pd=!cmp.eq(Rs,Rt)`	`Byte Q6_p_not_cmp_eq_RR(Word32 Rs, Word32 Rt)`
`Pd=!cmp.gt(Rs,#s10)`	`Byte Q6_p_not_cmp_gt_RI(Word32 Rs, Word32 Is10)`
`Pd=!cmp.gt(Rs,Rt)`	`Byte Q6_p_not_cmp_gt_RR(Word32 Rs, Word32 Rt)`
`Pd=!cmp.gtu(Rs,#u9)`	`Byte Q6_p_not_cmp_gtu_RI(Word32 Rs, Word32 Iu9)`
`Pd=!cmp.gtu(Rs,Rt)`	`Byte Q6_p_not_cmp_gtu_RR(Word32 Rs, Word32 Rt)`
`Pd=cmp.eq(Rs,#s10)`	`Byte Q6_p_cmp_eq_RI(Word32 Rs, Word32 Is10)`
`Pd=cmp.eq(Rs,Rt)`	`Byte Q6_p_cmp_eq_RR(Word32 Rs, Word32 Rt)`
`Pd=cmp.ge(Rs,#s8)`	`Byte Q6_p_cmp_ge_RI(Word32 Rs, Word32 Is8)`
`Pd=cmp.geu(Rs,#u8)`	`Byte Q6_p_cmp_geu_RI(Word32 Rs, Word32 Iu8)`
`Pd=cmp.gt(Rs,#s10)`	`Byte Q6_p_cmp_gt_RI(Word32 Rs, Word32 Is10)`

`Pd=cmp.gt(Rs,Rt)`	`Byte Q6_p_cmp_gt_RR(Word32 Rs, Word32 Rt)`
`Pd=cmp.gtu(Rs,#u9)`	`Byte Q6_p_cmp_gtu_RI(Word32 Rs, Word32 Iu9)`
`Pd=cmp.gtu(Rs,Rt)`	`Byte Q6_p_cmp_gtu_RR(Word32 Rs, Word32 Rt)`
`Pd=cmp.lt(Rs,Rt)`	`Byte Q6_p_cmp_lt_RR(Word32 Rs, Word32 Rt)`
`Pd=cmp.ltu(Rs,Rt)`	`Byte Q6_p_cmp_ltu_RR(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			MinOp			s5					Parse															d2		
0	1	1	1	0	1	0	1	0	0	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	0	0	**d**	**d**	Pd=cmp.eq(Rs,#s10)
0	1	1	1	0	1	0	1	0	0	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	1	0	0	**d**	**d**	Pd=!cmp.eq(Rs,#s10)
0	1	1	1	0	1	0	1	0	1	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	0	0	**d**	**d**	Pd=cmp.gt(Rs,#s10)
0	1	1	1	0	1	0	1	0	1	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	1	0	0	**d**	**d**	Pd=!cmp.gt(Rs,#s10)
0	1	1	1	0	1	0	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	0	0	**d**	**d**	Pd=cmp.gtu(Rs,#u9)
0	1	1	1	0	1	0	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	1	0	0	**d**	**d**	Pd=!cmp.gtu(Rs,#u9)
ICLASS				P	MajOp			MinOp			s5					Parse			t5											d2		
1	1	1	1	0	0	1	0	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	0	0	0	**d**	**d**	Pd=cmp.eq(Rs,Rt)
1	1	1	1	0	0	1	0	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	1	0	0	**d**	**d**	Pd=!cmp.eq(Rs,Rt)
1	1	1	1	0	0	1	0	-	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	0	0	0	**d**	**d**	Pd=cmp.gt(Rs,Rt)
1	1	1	1	0	0	1	0	-	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	1	0	0	**d**	**d**	Pd=!cmp.gt(Rs,Rt)
1	1	1	1	0	0	1	0	-	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	0	0	0	**d**	**d**	Pd=cmp.gtu(Rs,Rt)
1	1	1	1	0	0	1	0	-	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	1	0	0	**d**	**d**	Pd=!cmp.gtu(Rs,Rt)

Field name	Description
MajOp	Major opcode
MinOp	Minor opcode
Rs	No Rs read
P	Predicated
ICLASS	Instruction class
Parse	Packet/loop parse bits
d2	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Compare to general register

The register form compares two 32-bit registers for unsigned greater than, greater than, or equal. The immediate form compares a register against a signed or unsigned immediate value. The resulting zero or one is placed in a general register.

Syntax	Behavior
`Rd=[!]cmp.eq(Rs,#s8)`	`apply_extension(#s);` `Rd=(Rs[!]=#s);`
`Rd=[!]cmp.eq(Rs,Rt)`	`Rd=(Rs[!]=Rt);`

Class: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=!cmp.eq(Rs,#s8)`	`Word32 Q6_R_not_cmp_eq_RI(Word32 Rs, Word32 Is8)`
`Rd=!cmp.eq(Rs,Rt)`	`Word32 Q6_R_not_cmp_eq_RR(Word32 Rs, Word32 Rt)`
`Rd=cmp.eq(Rs,#s8)`	`Word32 Q6_R_cmp_eq_RI(Word32 Rs, Word32 Is8)`
`Rd=cmp.eq(Rs,Rt)`	`Word32 Q6_R_cmp_eq_RR(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			MinOp			s5					Parse											d5					
0	1	1	1	0	0	1	1	-	1	0	s	s	s	s	s	P	P	1	i	i	i	i	i	i	i	i	d	d	d	d	d	Rd=cmp.eq(Rs,#s8)
0	1	1	1	0	0	1	1	-	1	1	s	s	s	s	s	P	P	1	i	i	i	i	i	i	i	i	d	d	d	d	d	Rd=!cmp.eq(Rs,#s8)
ICLASS				P	MajOp			MinOp			s5					Parse			t5								d5					
1	1	1	1	0	0	1	1	0	1	0	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	d	d	d	d	d	Rd=cmp.eq(Rs,Rt)
1	1	1	1	0	0	1	1	0	1	1	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	d	d	d	d	d	Rd=!cmp.eq(Rs,Rt)

Field name	Description
MajOp	Major opcode
MinOp	Minor opcode
Rs	No Rs read
P	Predicated
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

11.2 CR

The CR instruction class includes instructions that manage control registers, including hardware looping, modulo addressing, and status flags.

CR instructions are executable on slot 3.

End loop instructions

The endloop instructions mark the end of a hardware loop. If the loop count (LC) register indicates that a loop should continue to iterate, the LC register is decremented and the program flow changes to the address in the start address (SA) register.

The `endloopN` instruction is actually a pseudo-instruction encoded in bits 15:14 of each instruction. Therefore, no distinct 32-bit encoding exists for this instruction.

Syntax	Behavior

endloop0	if (USR.LPCFG) {
	  if (USR.LPCFG==1) {
	  P3=0xff;
	  }
	  USR.LPCFG=USR.LPCFG-1;}
	if (LC0>1) {
	  PC=SA0;
	  LC0=LC0-1;
	}

endloop01	if (USR.LPCFG) {
	  if (USR.LPCFG==1) {
	  P3=0xff;
	  }
	  USR.LPCFG=USR.LPCFG-1;}
	if (LC0>1) {
	  PC=SA0;
	  LC0=LC0-1;
	} else {
	  if (LC1>1) {
	  PC=SA1;
	  LC1=LC1-1;
	  }
	}

endloop1	if (LC1>1) {
	  PC=SA1;
	  LC1=LC1-1;}

Class: N/A

Notes

- This instruction cannot be grouped in a packet with any program flow instructions.
- The Next PC value is the address immediately following the last instruction in the packet containing this instruction.
- The PC value is the address of the start of the packet

Corner detection acceleration

The `fastcorner9` instruction takes the Ps and Pt values and treats them as a circular bit string. If any contiguous nine bits are set around the circle, the result is true, false otherwise. The sense can optionally be inverted. This instruction is used to accelerate FAST corner detection.

Syntax	Behavior

`Pd=[!]fastcorner9(Ps,Pt)`	`PREDUSE_TIMING;`
	`tmp.h[0]=(Ps<<8)|Pt;`
	`tmp.h[1]=(Ps<<8)|Pt;`
	`for (i = 1; i < 9; i++) {`
	`  tmp &= tmp >> 1;`
	`}`
	`Pd = tmp == 0 ? 0xff : 0x00;`

Class: CR (slot 2,3)

Notes

- This instruction can execute on either slot2 or slot3, even though it is a CR-type

Intrinsics

`Pd=!fastcorner9(Ps,Pt)`	`Byte Q6_p_not_fastcorner9_pp(Byte Ps, Byte Pt)`
`Pd=fastcorner9(Ps,Pt)`	`Byte Q6_p_fastcorner9_pp(Byte Ps, Byte Pt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					sm									s2		Parse						t2								d2		
0	1	1	0	1	0	1	1	0	0	0	0	-	-	**s**	**s**	**P**	**P**	1	-	-	-	**t**	**t**	1	-	-	1	-	-	**d**	**d**	Pd=fastcorner9(Ps,Pt)
0	1	1	0	1	0	1	1	0	0	0	1	-	-	**s**	**s**	**P**	**P**	1	-	-	-	**t**	**t**	1	-	-	1	-	-	**d**	**d**	Pd=!fastcorner9(Ps,Pt)

Field name	Description
`sm`	Supervisor mode only
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d2`	Field to encode register d
`s2`	Field to encode register s
`t2`	Field to encode register t

Logical reductions on predicates

The `any8` instruction sets a destination predicate register to 0xff if any of the low eight bits in source predicate register Ps are set. Otherwise, the predicate is set to 0x00.

The `all8` instruction sets a destination predicate register to 0xff if all of the low eight bits in the source predicate register Ps are set. Otherwise, the predicate is set to 0x00.

Syntax	Behavior
`Pd = all8(Ps)`	`PREDUSE_TIMING;` `Pd = (Ps == 0xff ? 0xff : 0x00);`
`Pd = any8(Ps)`	`PREDUSE_TIMING;` `Pd = (Ps ? 0xff : 0x00);`

Class: CR (slot 2,3)

Notes

- This instruction can execute on either slot2 or slot3, even though it is a CR-type

Intrinsics

`Pd=all8(Ps)`	`Byte Q6_p_all8_p(Byte Ps)`
`Pd=any8(Ps)`	`Byte Q6_p_any8_p(Byte Ps)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					sm									s2		Parse														d2		
0	1	1	0	1	0	1	1	1	0	0	0	-	-	**s**	**s**	**P**	**P**	0	-	-	-	-	-	-	-	-	-	-	-	**d**	**d**	Pd=any8(Ps)
0	1	1	0	1	0	1	1	1	0	1	0	-	-	**s**	**s**	**P**	**P**	0	-	-	-	-	-	-	-	-	-	-	-	**d**	**d**	Pd=all8(Ps)

Field name	Description
`sm`	Supervisor mode only
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d2`	Field to encode register d
`s2`	Field to encode register s

Looping instructions

loopN is a single instruction that sets up a hardware loop. The N in the instruction name indicates the set of loop registers to use. Loop0 is the innermost loop, while loop1 is the outer loop. The loopN instruction first sets the start address (SA) register based on a PC-relative immediate add. The relative immediate is added to the PC and stored in SA. The loop count (LC) register is set to either an unsigned immediate or to a register value.

Syntax	Behavior

`loop0(#r7:2,#U10)`	`apply_extension(#r);`
	`#r=#r & ~PCALIGN_MASK;`
	`SA0=PC+#r;`
	`LC0=#U;`
	`USR.LPCFG=0;`

`loop0(#r7:2,Rs)`	`apply_extension(#r);`
	`#r=#r & ~PCALIGN_MASK;`
	`SA0=PC+#r;`
	`LC0=Rs;`
	`USR.LPCFG=0;`

`loop1(#r7:2,#U10)`	`apply_extension(#r);`
	`#r=#r & ~PCALIGN_MASK;`
	`SA1=PC+#r;`
	`LC1=#U;`

`loop1(#r7:2,Rs)`	`apply_extension(#r);`
	`#r=#r & ~PCALIGN_MASK;`
	`SA1=PC+#r;`
	`LC1=Rs;`

Class: CR (slot 3)

Notes

- This instruction cannot execute in the last address of a hardware loop.
- The Next PC value is the address immediately following the last instruction in the packet containing this instruction.
- The PC value is the address of the start of the packet
- A PC-relative address is formed by taking the decoded immediate value and adding it to the current PC value.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					sm						s5					Parse																
0	1	1	0	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	-	-	-	**i**	**i**	-	-	-	loop0(#r7:2,Rs)
0	1	1	0	0	0	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	-	-	-	**i**	**i**	-	-	-	loop1(#r7:2,Rs)
ICLASS					sm											Parse																
0	1	1	0	1	0	0	1	0	0	0	**I**	**I**	**I**	**I**	**I**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	**I**	**I**	**I**	**i**	**i**	-	**I**	**I**	loop0(#r7:2,#U10)
0	1	1	0	1	0	0	1	0	0	1	**I**	**I**	**I**	**I**	**I**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	**I**	**I**	**I**	**i**	**i**	-	**I**	**I**	loop1(#r7:2,#U10)

Field name	Description
sm	Supervisor mode only
ICLASS	Instruction class
Parse	Packet/loop parse bits
s5	Field to encode register s

Add to PC

Add an immediate value to the program counter (PC) and place the result in a destination register. This instruction is typically used with a constant extender to add a 32-bit immediate value to PC.

Syntax	Behavior
`Rd=add(pc,#u6)`	`Rd=PC+apply_extension(#u);`

Class: CR (slot 3)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					sm											Parse											d5					
0	1	1	0	1	0	1	0	0	1	0	0	1	0	0	1	**P**	**P**	**-**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=add(pc,#u6)

Field name	Description
`sm`	Supervisor mode only
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d

Pipelined loop instructions

spNloop0 is a single instruction that sets up a hardware loop with automatic predicate control. This features saves code size by enabling generation of many software pipelined loops without prologue code. Upon executing this instruction, the P3 register is automatically cleared. After the loop has executed N times (where N is selectable from 1 through 3), the P3 register is set. Store instructions in the loop are predicated with P3 and thus not enabled during the pipeline warm-up.

In the spNloop0 instruction, the loop 0 (inner-loop) registers are used. This instruction sets the start address (SA0) register based on a PC-relative immediate add. The relative immediate is added to the PC and stored in SA0. The loop count (LC0) is set to either an unsigned immediate or to a register value. The predicate P3 is cleared. The USR.LPCFG bits are set based on the N value.

Syntax	Behavior

`p3=sp1loop0(#r7:2,#U10)`	`apply_extension(#r);`
	`#r=#r & ~PCALIGN_MASK;`
	`SA0=PC+#r;`
	`LC0=#U;`
	`USR.LPCFG=1;`
	`P3=0;`

`p3=sp1loop0(#r7:2,Rs)`	`apply_extension(#r);`
	`#r=#r & ~PCALIGN_MASK;`
	`SA0=PC+#r;`
	`LC0=Rs;`
	`USR.LPCFG=1;`
	`P3=0;`

`p3=sp2loop0(#r7:2,#U10)`	`apply_extension(#r);`
	`#r=#r & ~PCALIGN_MASK;`
	`SA0=PC+#r;`
	`LC0=#U;`
	`USR.LPCFG=2;`
	`P3=0;`

`p3=sp2loop0(#r7:2,Rs)`	`apply_extension(#r);`
	`#r=#r & ~PCALIGN_MASK;`
	`SA0=PC+#r;`
	`LC0=Rs;`
	`USR.LPCFG=2;`
	`P3=0;`

`p3=sp3loop0(#r7:2,#U10)`	`apply_extension(#r);`
	`#r=#r & ~PCALIGN_MASK;`
	`SA0=PC+#r;`
	`LC0=#U;`
	`USR.LPCFG=3;`
	`P3=0;`

`p3=sp3loop0(#r7:2,Rs)`	`apply_extension(#r);`
	`#r=#r & ~PCALIGN_MASK;`
	`SA0=PC+#r;`
	`LC0=Rs;`
	`USR.LPCFG=3;`
	`P3=0;`

Class: CR (slot 3)

Notes

- The predicate generated by this instruction cannot be used as a .new predicate, nor can it be automatically AND'd with another predicate.
- This instruction cannot execute in the last address of a hardware loop.
- The Next PC value is the address immediately following the last instruction in the packet containing this instruction.
- The PC value is the address of the start of the packet
- A PC-relative address is formed by taking the decoded immediate value and adding it to the current PC value.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					sm						s5					Parse																
0	1	1	0	0	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	-	-	-	**i**	**i**	-	-	-	p3=sp1loop0(#r7:2,Rs)
0	1	1	0	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	-	-	-	**i**	**i**	-	-	-	p3=sp2loop0(#r7:2,Rs)
0	1	1	0	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	-	-	-	**i**	**i**	-	-	-	p3=sp3loop0(#r7:2,Rs)
ICLASS					sm											Parse																
0	1	1	0	1	0	0	1	1	0	1	**I**	**I**	**I**	**I**	**I**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	**I**	**I**	**I**	**i**	**i**	-	**I**	**I**	p3=sp1loop0(#r7:2,#U10)
0	1	1	0	1	0	0	1	1	1	0	**I**	**I**	**I**	**I**	**I**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	**I**	**I**	**I**	**i**	**i**	-	**I**	**I**	p3=sp2loop0(#r7:2,#U10)
0	1	1	0	1	0	0	1	1	1	1	**I**	**I**	**I**	**I**	**I**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	**I**	**I**	**I**	**i**	**i**	-	**I**	**I**	p3=sp3loop0(#r7:2,#U10)

Field name	Description
sm	Supervisor mode only
ICLASS	Instruction class
Parse	Packet/loop parse bits
s5	Field to encode register s

Logical operations on predicates

Perform bitwise logical operations on predicate registers.

Syntax	Behavior
`Pd=Ps`	`Assembler mapped to: "Pd=or(Ps,Ps)"`
`Pd=and(Ps,and(Pt,[!]Pu))`	`PREDUSE_TIMING;` `Pd = Ps & Pt & (~Pu);`
`Pd=and(Ps,or(Pt,[!]Pu))`	`PREDUSE_TIMING;` `Pd = Ps & (Pt | (~Pu));`
`Pd=and(Pt,[!]Ps)`	`PREDUSE_TIMING;` `Pd=Pt & (~Ps);`
`Pd=not(Ps)`	`PREDUSE_TIMING;` `Pd=~Ps;`
`Pd=or(Ps,and(Pt,[!]Pu))`	`PREDUSE_TIMING;` `Pd = Ps | (Pt & (~Pu));`
`Pd=or(Ps,or(Pt,[!]Pu))`	`PREDUSE_TIMING;` `Pd = Ps | Pt | (~Pu);`
`Pd=or(Pt,[!]Ps)`	`PREDUSE_TIMING;` `Pd=Pt | (~Ps);`
`Pd=xor(Ps,Pt)`	`PREDUSE_TIMING;` `Pd=Ps ^ Pt;`

Class: CR (slot 2,3)

Notes

- This instruction can execute on either slot2 or slot3, even though it is a CR-type

Intrinsics

`Pd=Ps`	`Byte Q6_p_equals_p(Byte Ps)`
`Pd=and(Ps,and(Pt,!Pu))`	`Byte Q6_p_and_and_ppnp(Byte Ps, Byte Pt, Byte Pu)`
`Pd=and(Ps,and(Pt,Pu))`	`Byte Q6_p_and_and_ppp(Byte Ps, Byte Pt, Byte Pu)`
`Pd=and(Ps,or(Pt,!Pu))`	`Byte Q6_p_and_or_ppnp(Byte Ps, Byte Pt, Byte Pu)`
`Pd=and(Ps,or(Pt,Pu))`	`Byte Q6_p_and_or_ppp(Byte Ps, Byte Pt, Byte Pu)`
`Pd=and(Pt,!Ps)`	`Byte Q6_p_and_pnp(Byte Pt, Byte Ps)`
`Pd=and(Pt,Ps)`	`Byte Q6_p_and_pp(Byte Pt, Byte Ps)`
`Pd=not(Ps)`	`Byte Q6_p_not_p(Byte Ps)`
`Pd=or(Ps,and(Pt,!Pu))`	`Byte Q6_p_or_and_ppnp(Byte Ps, Byte Pt, Byte Pu)`
`Pd=or(Ps,and(Pt,Pu))`	`Byte Q6_p_or_and_ppp(Byte Ps, Byte Pt, Byte Pu)`

`Pd=or(Ps,or(Pt,!Pu))`	`Byte Q6_p_or_or_ppnp(Byte Ps, Byte Pt, Byte Pu)`
`Pd=or(Ps,or(Pt,Pu))`	`Byte Q6_p_or_or_ppp(Byte Ps, Byte Pt, Byte Pu)`
`Pd=or(Pt,!Ps)`	`Byte Q6_p_or_pnp(Byte Pt, Byte Ps)`
`Pd=or(Pt,Ps)`	`Byte Q6_p_or_pp(Byte Pt, Byte Ps)`
`Pd=xor(Ps,Pt)`	`Byte Q6_p_xor_pp(Byte Ps, Byte Pt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					sm									s2		Parse						t2								d2		
0	1	1	0	1	0	1	1	0	0	0	0	-	-	**s**	**s**	**P**	**P**	0	-	-	-	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=and(Pt,Ps)
ICLASS					sm									s2		Parse						t2		u2						d2		
0	1	1	0	1	0	1	1	0	0	0	1	-	-	**s**	**s**	**P**	**P**	0	-	-	-	**t**	**t**	**u**	**u**	-	-	-	-	**d**	**d**	Pd=and(Ps,and(Pt,Pu))
ICLASS					sm									s2		Parse						t2								d2		
0	1	1	0	1	0	1	1	0	0	1	0	-	-	**s**	**s**	**P**	**P**	0	-	-	-	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=or(Pt,Ps)
ICLASS					sm									s2		Parse						t2		u2						d2		
0	1	1	0	1	0	1	1	0	0	1	1	-	-	**s**	**s**	**P**	**P**	0	-	-	-	**t**	**t**	**u**	**u**	-	-	-	-	**d**	**d**	Pd=and(Ps,or(Pt,Pu))
ICLASS					sm									s2		Parse						t2								d2		
0	1	1	0	1	0	1	1	0	1	0	0	-	-	**s**	**s**	**P**	**P**	0	-	-	-	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=xor(Ps,Pt)
ICLASS					sm									s2		Parse						t2		u2						d2		
0	1	1	0	1	0	1	1	0	1	0	1	-	-	**s**	**s**	**P**	**P**	0	-	-	-	**t**	**t**	**u**	**u**	-	-	-	-	**d**	**d**	Pd=or(Ps,and(Pt,Pu))
ICLASS					sm									s2		Parse						t2								d2		
0	1	1	0	1	0	1	1	0	1	1	0	-	-	**s**	**s**	**P**	**P**	0	-	-	-	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=and(Pt,!Ps)
ICLASS					sm									s2		Parse						t2		u2						d2		
0	1	1	0	1	0	1	1	0	1	1	1	-	-	**s**	**s**	**P**	**P**	0	-	-	-	**t**	**t**	**u**	**u**	-	-	-	-	**d**	**d**	Pd=or(Ps,or(Pt,Pu))
0	1	1	0	1	0	1	1	1	0	0	1	-	-	**s**	**s**	**P**	**P**	0	-	-	-	**t**	**t**	**u**	**u**	-	-	-	-	**d**	**d**	Pd=and(Ps,and(Pt,!Pu))
0	1	1	0	1	0	1	1	1	0	1	1	-	-	**s**	**s**	**P**	**P**	0	-	-	-	**t**	**t**	**u**	**u**	-	-	-	-	**d**	**d**	Pd=and(Ps,or(Pt,!Pu))
ICLASS					sm									s2		Parse														d2		
0	1	1	0	1	0	1	1	1	1	0	0	-	-	**s**	**s**	**P**	**P**	0	-	-	-	-	-	-	-	-	-	-	-	**d**	**d**	Pd=not(Ps)
ICLASS					sm									s2		Parse						t2		u2						d2		
0	1	1	0	1	0	1	1	1	1	0	1	-	-	**s**	**s**	**P**	**P**	0	-	-	-	**t**	**t**	**u**	**u**	-	-	-	-	**d**	**d**	Pd=or(Ps,and(Pt,!Pu))
ICLASS					sm									s2		Parse						t2								d2		
0	1	1	0	1	0	1	1	1	1	1	0	-	-	**s**	**s**	**P**	**P**	0	-	-	-	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=or(Pt,!Ps)
ICLASS					sm									s2		Parse						t2		u2						d2		
0	1	1	0	1	0	1	1	1	1	1	1	-	-	**s**	**s**	**P**	**P**	0	-	-	-	**t**	**t**	**u**	**u**	-	-	-	-	**d**	**d**	Pd=or(Ps,or(Pt,!Pu))

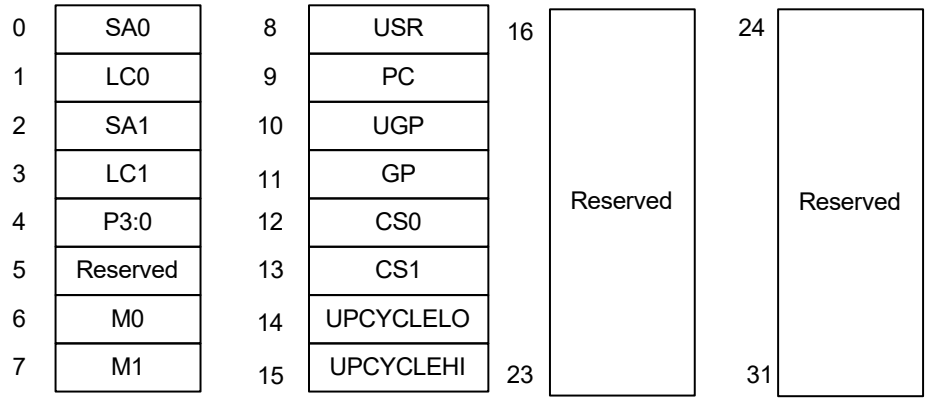
Field name	Description
`sm`	Supervisor mode only
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d2`	Field to encode register d
`s2`	Field to encode register s
`t2`	Field to encode register t
`u2`	Field to encode register u

User control register transfer

Move 32- or 64-bit values between a user control register and a general register. The user control registers include SA, LC, Predicates, M, USR, PC, UGP, GP, and CS, and UPCYCLE. The figure shows the user control registers and their register field encodings.

Registers can be moved as singles or as aligned 64-bit pairs.

Note that the PC register is not writable. A program flow instruction must be used to change the PC value.



Syntax	Behavior
`Cd=Rs`	`Cd=Rs;`
`Cdd=Rss`	`Cdd=Rss;`
`Rd=Cs`	`Rd=Cs;`
`Rdd=Css`	`Rdd=Css;`

Class: CR (slot 3)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					sm						s5					Parse											d5					
0	1	1	0	0	0	1	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Cd=Rs
0	1	1	0	0	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Cdd=Rss
0	1	1	0	1	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Rdd=Css
0	1	1	0	1	0	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=Cs

Field name	Description
`sm`	Supervisor mode only
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s

11.3 JR

The JR instruction class includes instructions to change the program flow to a new location contained in a register.

JR instructions are executable on slot 2.

Call subroutine from register

Change the program flow to a subroutine. This instruction first transfers the next program counter (NPC) value into the link register, and then jumps to a target address contained in a register.

This instruction can only appear in slot 2.

Syntax	Behavior
`callr Rs`	`LR=NPC;` `PC=Rs;`
`callrh Rs`	`LR=NPC;` `PC=Rs;`

`if ([!]Pu) callr Rs`	`if ([!]Pu[0]) {`
	`  LR=NPC;`
	`  PC=Rs;`
	`}`

Class: JR (slot 2)

Notes

- This instruction can be conditionally executed based on the value of a predicate register. If the instruction is preceded by 'if Pn', the instruction only executes if the LSB of the predicate register is 1. Similarly, if the instruction is preceded by 'if !Pn', the instruction executes only if the LSB of Pn is 0.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse																	
0	1	0	1	0	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	callr Rs
0	1	0	1	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	callrh Rs
ICLASS											s5					Parse						u2										
0	1	0	1	0	0	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	**u**	**u**	-	-	-	-	-	-	-	-	if (Pu) callr Rs
0	1	0	1	0	0	0	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	**u**	**u**	-	-	-	-	-	-	-	-	if (!Pu) callr Rs

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
s5	Field to encode register s
u2	Field to encode register u

Hinted call subroutine from register

Change the program flow to a subroutine. This instruction first transfers the next program counter (NPC) value into the link register, and then jumps to a target address contained in a register. This instruction is effective only when a preceding `hintjr` exists.

This instruction can only appear in slot 2.

Syntax	Behavior
`callrh Rs`	`LR = NPC;` `PC = Rs;`

Class: JR (slot 2)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse																	
0	1	0	1	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	callrh Rs

Field name	Description
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`s5`	Field to encode register s

Hint an indirect jump address

Provides a hint to indicate that there will soon be an indirect call to the address specified in Rs. The indirect call can be either `jumprh` or `callrh`.

This instruction can appear in either slot 2 or slot 3.

Syntax	Behavior
`hintjr(Rs)`	

Class: JR (slot 2,3)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse																
0	1	0	1	0	0	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	hintjr(Rs)

Field name	Description
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`s5`	Field to encode register s

Jump to address from register

This instruction changes the program counter to a target address contained in a register.

This instruction can appear only in slot 2.

Syntax	Behavior
`if ([!]Pu) jumpr Rs`	`Assembler mapped to: "if ([!]Pu) ""jumpr"":nt ""Rs"`
`if ([!]Pu[.new]) jumpr:<hint> Rs`	`if([!]Pu[.new][0]){ PC=Rs; }`
`jumpr Rs`	`PC = Rs;`
`jumprh Rs`	`PC = Rs;`

Class: JR (slot 2)

Notes

- This instruction can conditionally execute based on the value of a predicate register. If the instruction is preceded by 'if Pn', the instruction only executes if the LSB of the predicate register is 1. Similarly, if the instruction is preceded by 'if !Pn', the instruction executes only if the LSB of Pn is 0.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse																	
0	1	0	1	0	0	1	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	jumpr Rs
0	1	0	1	0	0	1	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	jumprh Rs
ICLASS											s5					Parse						u2										
0	1	0	1	0	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	0	0	-	**u**	**u**	-	-	-	-	-	-	-	-	if (Pu) jumpr:nt Rs
0	1	0	1	0	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	0	1	-	**u**	**u**	-	-	-	-	-	-	-	-	if (Pu.new) jumpr:nt Rs
0	1	0	1	0	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	1	0	-	**u**	**u**	-	-	-	-	-	-	-	-	if (Pu) jumpr:t Rs
0	1	0	1	0	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	1	1	-	**u**	**u**	-	-	-	-	-	-	-	-	if (Pu.new) jumpr:t Rs
0	1	0	1	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	0	0	-	**u**	**u**	-	-	-	-	-	-	-	-	if (!Pu) jumpr:nt Rs
0	1	0	1	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	0	1	-	**u**	**u**	-	-	-	-	-	-	-	-	if (!Pu.new) jumpr:nt Rs
0	1	0	1	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	1	0	-	**u**	**u**	-	-	-	-	-	-	-	-	if (!Pu) jumpr:t Rs
0	1	0	1	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	1	1	-	**u**	**u**	-	-	-	-	-	-	-	-	if (!Pu.new) jumpr:t Rs

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
s5	Field to encode register s
u2	Field to encode register u

Hinted jump to address from register

Change the program flow to a target address. This instruction changes the program counter to a target address contained in a register. This instruction is effective only when a preceding `hintjr` exists.

This instruction can appear only in slot 2.

Syntax	Behavior
`jumprh Rs`	`PC = Rs;`

Class: JR (slot 2)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse																	
0	1	0	1	0	0	1	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	jumprh Rs

Field name	Description
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`s5`	Field to encode register s

11.4 J

The J instruction class includes branch instructions (jumps and calls) that obtain the target address from a (PC-relative) immediate address value.

J instructions are executable on slot 2 and slot 3.

Call subroutine

Change the program flow to a subroutine. This instruction first transfers the next program counter (NPC) value into the link register, and then jumps to the target address.

This instruction can appear in slots 2 or 3.

Syntax	Behavior

`call #r22:2`	`apply_extension(#r);`
	`#r = #r & ~PCALIGN_MASK;`
	`LR = NPC;`
	`PC = PC + #r;`

`if ([!]Pu) call #r15:2`	`apply_extension(#r);`
	`#r = #r & ~PCALIGN_MASK;`
	`if ([!]Pu[0]) {`
	`  LR = NPC;`
	`  PC = PC + #r;`
	`}`

Class: J (slots 2,3)

Notes

- This instruction can conditionally execute based on the value of a predicate register. If the instruction is preceded by 'if Pn', the instruction only executes if the LSB of the predicate register is 1. Similarly, if the instruction is preceded by 'if !Pn', the instruction executes only if the LSB of Pn is 0.
- The Next PC value is the address immediately following the last instruction in the packet containing this instruction.
- The PC value is the address of the start of the packet
- A PC-relative address is formed by taking the decoded immediate value and adding it to the current PC value.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse																
0	1	0	1	1	0	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	call #r22:2
ICLASS																Parse				D N		u2										
0	1	0	1	1	1	0	1	**i**	**i**	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**-**	0	**-**	**u**	**u**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (Pu) call #r15:2
0	1	0	1	1	1	0	1	**i**	**i**	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**-**	0	**-**	**u**	**u**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!Pu) call #r15:2

Field name	Description
ICLASS	Instruction class
DN	Dot-new
Parse	Packet/loop parse bits
u2	Field to encode register u

Compare and jump

Compare two registers, or a register and immediate value, and write a predicate with the result. Then use the predicate result to conditionally jump to a PC-relative target address.

The registers available as operands are restricted to R0 through R7 and R16 through R23. The predicate destination is restricted to P0 and P1.

In assembly syntax, this instruction appears as two instructions in the packet: a compare and a separate conditional jump. The assembler can convert adjacent compare and jump instructions into compound compare-jump form.

Syntax	Behavior

`p[01]=cmp.eq(Rs,#-1); if ([!]p[01].new) jump:<hint> #r9:2`	`P[01]=(Rs==-1) ? 0xff : 0x00 if ([!]P[01].new[0]) {`
	`  apply_extension(#r);`
	`  #r = #r & ~PCALIGN_MASK;`
	`  PC = PC+#r;`
	`}`

`p[01]=cmp.eq(Rs,#U5); if ([!]p[01].new) jump:<hint> #r9:2`	`P[01]=(Rs==#U) ? 0xff : 0x00 if ([!]P[01].new[0]) {`
	`  apply_extension(#r);`
	`  #r =#r & ~PCALIGN_MASK;`
	`  PC =PC+#r;`
	`}`

`p[01]=cmp.eq(Rs,Rt); if ([!]p[01].new) jump:<hint> #r9:2`	`P[01]=(Rs==Rt) ? 0xff : 0x00 if ([!]P[01].new[0]) {`
	`  apply_extension(#r);`
	`  #r =#r & ~PCALIGN_MASK;`
	`  PC =PC+#r;`
	`}`

`p[01]=cmp.gt(Rs,#-1); if ([!]p[01].new) jump:<hint> #r9:2`	`P[01]=(Rs>-1) ? 0xff : 0x00 if ([!]P[01].new[0]) {`
	`  apply_extension(#r);`
	`  #r =#r & ~PCALIGN_MASK;`
	`  PC =PC+#r;`
	`}`

`p[01]=cmp.gt(Rs,#U5); if ([!]p[01].new) jump:<hint> #r9:2`	`P[01]=(Rs>#U) ? 0xff : 0x00 if ([!]P[01].new[0]) {`
	`  apply_extension(#r);`
	`  #r=#r & ~PCALIGN_MASK;`
	`  PC=PC+#r;`
	`}`

`p[01]=cmp.gt(Rs,Rt); if ([!]p[01].new) jump:<hint> #r9:2`	`P[01]=(Rs>Rt) ? 0xff : 0x00 if ([!]P[01].new[0]) {`
	`  apply_extension(#r);`
	`  #r=#r & ~PCALIGN_MASK;`
	`  PC=PC+#r;`
	`}`

Syntax	Behavior

`p[01]=cmp.gtu(Rs,#U5); if ([!]p[01].new) jump:<hint> #r9:2`	`P[01]=(Rs.uw[0]>#U) ? 0xff : 0x00 if ([!]P[01].new[0]) {`
	`  apply_extension(#r);`
	`  #r=#r & ~PCALIGN_MASK;`
	`  PC=PC+#r;`
	`}`

`p[01]=cmp.gtu(Rs,Rt); if ([!]p[01].new) jump:<hint> #r9:2`	`P[01]=(Rs.uw[0]>Rt) ? 0xff : 0x00 if ([!]P[01].new[0]) {`
	`  apply_extension(#r);`
	`  #r=#r & ~PCALIGN_MASK;`
	`  PC=PC+#r;`
	`}`

`p[01]=tstbit(Rs,#0); if ([!]p[01].new) jump:<hint> #r9:2`	`P[01]=(Rs & 1) ? 0xff : 0x00 if ([!]P[01].new[0]) {`
	`  apply_extension(#r);`
	`  #r=#r & ~PCALIGN_MASK;`
	`  PC=PC+#r;`
	`}`

Class: J (slots 0, 1, 2, 3)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS												s4				Parse																
0	0	0	1	0	0	0	0	0	0	i	i	s	s	s	s	P	P	0	I	I	I	I	I	i	i	i	i	i	i	i	-	p0=cmp.eq(Rs,#U5); if (p0.new) jump:nt #r9:2
0	0	0	1	0	0	0	0	0	0	i	i	s	s	s	s	P	P	1	I	I	I	I	I	i	i	i	i	i	i	i	-	p0=cmp.eq(Rs,#U5); if (p0.new) jump:t #r9:2
0	0	0	1	0	0	0	0	0	1	i	i	s	s	s	s	P	P	0	I	I	I	I	I	i	i	i	i	i	i	i	-	p0=cmp.eq(Rs,#U5); if (!p0.new) jump:nt #r9:2
0	0	0	1	0	0	0	0	0	1	i	i	s	s	s	s	P	P	1	I	I	I	I	I	i	i	i	i	i	i	i	-	p0=cmp.eq(Rs,#U5); if (!p0.new) jump:t #r9:2
0	0	0	1	0	0	0	0	1	0	i	i	s	s	s	s	P	P	0	I	I	I	I	I	i	i	i	i	i	i	i	-	p0=cmp.gt(Rs,#U5); if (p0.new) jump:nt #r9:2
0	0	0	1	0	0	0	0	1	0	i	i	s	s	s	s	P	P	1	I	I	I	I	I	i	i	i	i	i	i	i	-	p0=cmp.gt(Rs,#U5); if (p0.new) jump:t #r9:2
0	0	0	1	0	0	0	0	1	1	i	i	s	s	s	s	P	P	0	I	I	I	I	I	i	i	i	i	i	i	i	-	p0=cmp.gt(Rs,#U5); if (!p0.new) jump:nt #r9:2
0	0	0	1	0	0	0	0	1	1	i	i	s	s	s	s	P	P	1	I	I	I	I	I	i	i	i	i	i	i	i	-	p0=cmp.gt(Rs,#U5); if (!p0.new) jump:t #r9:2
0	0	0	1	0	0	0	1	0	0	i	i	s	s	s	s	P	P	0	I	I	I	I	I	i	i	i	i	i	i	i	-	p0=cmp.gtu(Rs,#U5); if (p0.new) jump:nt #r9:2
0	0	0	1	0	0	0	1	0	0	i	i	s	s	s	s	P	P	1	I	I	I	I	I	i	i	i	i	i	i	i	-	p0=cmp.gtu(Rs,#U5); if (p0.new) jump:t #r9:2
0	0	0	1	0	0	0	1	0	1	i	i	s	s	s	s	P	P	0	I	I	I	I	I	i	i	i	i	i	i	i	-	p0=cmp.gtu(Rs,#U5); if (!p0.new) jump:nt #r9:2
0	0	0	1	0	0	0	1	0	1	i	i	s	s	s	s	P	P	1	I	I	I	I	I	i	i	i	i	i	i	i	-	p0=cmp.gtu(Rs,#U5); if (!p0.new) jump:t #r9:2
0	0	0	1	0	0	0	1	1	0	i	i	s	s	s	s	P	P	0	-	-	-	0	0	i	i	i	i	i	i	i	-	p0=cmp.eq(Rs,#-1); if (p0.new) jump:nt #r9:2
0	0	0	1	0	0	0	1	1	0	i	i	s	s	s	s	P	P	0	-	-	-	0	1	i	i	i	i	i	i	i	-	p0=cmp.gt(Rs,#-1); if (p0.new) jump:nt #r9:2
0	0	0	1	0	0	0	1	1	0	i	i	s	s	s	s	P	P	0	-	-	-	1	1	i	i	i	i	i	i	i	-	p0=tstbit(Rs,#0); if (p0.new) jump:nt #r9:2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	1	0	0	0	1	1	0	i	i	s	s	s	s	P	P	1	-	-	-	0	0	i	i	i	i	i	i	i	-	p0=cmp.eq(Rs,#-1); if (p0.new) jump:t #r9:2
0	0	0	1	0	0	0	1	1	0	i	i	s	s	s	s	P	P	1	-	-	-	0	1	i	i	i	i	i	i	i	-	p0=cmp.gt(Rs,#-1); if (p0.new) jump:t #r9:2
0	0	0	1	0	0	0	1	1	0	i	i	s	s	s	s	P	P	1	-	-	-	1	1	i	i	i	i	i	i	i	-	p0=tstbit(Rs,#0); if (p0.new) jump:t #r9:2
0	0	0	1	0	0	0	1	1	1	i	i	s	s	s	s	P	P	0	-	-	-	0	0	i	i	i	i	i	i	i	-	p0=cmp.eq(Rs,#-1); if (!p0.new) jump:nt #r9:2
0	0	0	1	0	0	0	1	1	1	i	i	s	s	s	s	P	P	0	-	-	-	0	1	i	i	i	i	i	i	i	-	p0=cmp.gt(Rs,#-1); if (!p0.new) jump:nt #r9:2
0	0	0	1	0	0	0	1	1	1	i	i	s	s	s	s	P	P	0	-	-	-	1	1	i	i	i	i	i	i	i	-	p0=tstbit(Rs,#0); if (!p0.new) jump:nt #r9:2
0	0	0	1	0	0	0	1	1	1	i	i	s	s	s	s	P	P	1	-	-	-	0	0	i	i	i	i	i	i	i	-	p0=cmp.eq(Rs,#-1); if (!p0.new) jump:t #r9:2
0	0	0	1	0	0	0	1	1	1	i	i	s	s	s	s	P	P	1	-	-	-	0	1	i	i	i	i	i	i	i	-	p0=cmp.gt(Rs,#-1); if (!p0.new) jump:t #r9:2
0	0	0	1	0	0	0	1	1	1	i	i	s	s	s	s	P	P	1	-	-	-	1	1	i	i	i	i	i	i	i	-	p0=tstbit(Rs,#0); if (!p0.new) jump:t #r9:2
0	0	0	1	0	0	1	0	0	0	i	i	s	s	s	s	P	P	0	I	I	I	I	I	i	i	i	i	i	i	i	-	p1=cmp.eq(Rs,#U5); if (p1.new) jump:nt #r9:2
0	0	0	1	0	0	1	0	0	0	i	i	s	s	s	s	P	P	1	I	I	I	I	I	i	i	i	i	i	i	i	-	p1=cmp.eq(Rs,#U5); if (p1.new) jump:t #r9:2
0	0	0	1	0	0	1	0	0	1	i	i	s	s	s	s	P	P	0	I	I	I	I	I	i	i	i	i	i	i	i	-	p1=cmp.eq(Rs,#U5); if (!p1.new) jump:nt #r9:2
0	0	0	1	0	0	1	0	0	1	i	i	s	s	s	s	P	P	1	I	I	I	I	I	i	i	i	i	i	i	i	-	p1=cmp.eq(Rs,#U5); if (!p1.new) jump:t #r9:2
0	0	0	1	0	0	1	0	1	0	i	i	s	s	s	s	P	P	0	I	I	I	I	I	i	i	i	i	i	i	i	-	p1=cmp.gt(Rs,#U5); if (p1.new) jump:nt #r9:2
0	0	0	1	0	0	1	0	1	0	i	i	s	s	s	s	P	P	1	I	I	I	I	I	i	i	i	i	i	i	i	-	p1=cmp.gt(Rs,#U5); if (p1.new) jump:t #r9:2
0	0	0	1	0	0	1	0	1	1	i	i	s	s	s	s	P	P	0	I	I	I	I	I	i	i	i	i	i	i	i	-	p1=cmp.gt(Rs,#U5); if (!p1.new) jump:nt #r9:2
0	0	0	1	0	0	1	0	1	1	i	i	s	s	s	s	P	P	1	I	I	I	I	I	i	i	i	i	i	i	i	-	p1=cmp.gt(Rs,#U5); if (!p1.new) jump:t #r9:2
0	0	0	1	0	0	1	1	0	0	i	i	s	s	s	s	P	P	0	I	I	I	I	I	i	i	i	i	i	i	i	-	p1=cmp.gtu(Rs,#U5); if (p1.new) jump:nt #r9:2
0	0	0	1	0	0	1	1	0	0	i	i	s	s	s	s	P	P	1	I	I	I	I	I	i	i	i	i	i	i	i	-	p1=cmp.gtu(Rs,#U5); if (p1.new) jump:t #r9:2
0	0	0	1	0	0	1	1	0	1	i	i	s	s	s	s	P	P	0	I	I	I	I	I	i	i	i	i	i	i	i	-	p1=cmp.gtu(Rs,#U5); if (!p1.new) jump:nt #r9:2
0	0	0	1	0	0	1	1	0	1	i	i	s	s	s	s	P	P	1	I	I	I	I	I	i	i	i	i	i	i	i	-	p1=cmp.gtu(Rs,#U5); if (!p1.new) jump:t #r9:2
0	0	0	1	0	0	1	1	1	0	i	i	s	s	s	s	P	P	0	-	-	-	0	0	i	i	i	i	i	i	i	-	p1=cmp.eq(Rs,#-1); if (p1.new) jump:nt #r9:2
0	0	0	1	0	0	1	1	1	0	i	i	s	s	s	s	P	P	0	-	-	-	0	1	i	i	i	i	i	i	i	-	p1=cmp.gt(Rs,#-1); if (p1.new) jump:nt #r9:2
0	0	0	1	0	0	1	1	1	0	i	i	s	s	s	s	P	P	0	-	-	-	1	1	i	i	i	i	i	i	i	-	p1=tstbit(Rs,#0); if (p1.new) jump:nt #r9:2
0	0	0	1	0	0	1	1	1	0	i	i	s	s	s	s	P	P	1	-	-	-	0	0	i	i	i	i	i	i	i	-	p1=cmp.eq(Rs,#-1); if (p1.new) jump:t #r9:2
0	0	0	1	0	0	1	1	1	0	i	i	s	s	s	s	P	P	1	-	-	-	0	1	i	i	i	i	i	i	i	-	p1=cmp.gt(Rs,#-1); if (p1.new) jump:t #r9:2
0	0	0	1	0	0	1	1	1	0	i	i	s	s	s	s	P	P	1	-	-	-	1	1	i	i	i	i	i	i	i	-	p1=tstbit(Rs,#0); if (p1.new) jump:t #r9:2
0	0	0	1	0	0	1	1	1	1	i	i	s	s	s	s	P	P	0	-	-	-	0	0	i	i	i	i	i	i	i	-	p1=cmp.eq(Rs,#-1); if (!p1.new) jump:nt #r9:2
0	0	0	1	0	0	1	1	1	1	i	i	s	s	s	s	P	P	0	-	-	-	0	1	i	i	i	i	i	i	i	-	p1=cmp.gt(Rs,#-1); if (!p1.new) jump:nt #r9:2
0	0	0	1	0	0	1	1	1	1	i	i	s	s	s	s	P	P	0	-	-	-	1	1	i	i	i	i	i	i	i	-	p1=tstbit(Rs,#0); if (!p1.new) jump:nt #r9:2
0	0	0	1	0	0	1	1	1	1	i	i	s	s	s	s	P	P	1	-	-	-	0	0	i	i	i	i	i	i	i	-	p1=cmp.eq(Rs,#-1); if (!p1.new) jump:t #r9:2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	1	0	0	1	1	1	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	**-**	**-**	**-**	0	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	p1=cmp.gt(Rs,#-1); if (!p1.new) jump:t #r9:2
0	0	0	1	0	0	1	1	1	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	**-**	**-**	**-**	1	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	p1=tstbit(Rs,#0); if (!p1.new) jump:t #r9:2
ICLASS												s4				Parse				t4												
0	0	0	1	0	1	0	0	0	0	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	p0=cmp.eq(Rs,Rt); if (p0.new) jump:nt #r9:2
0	0	0	1	0	1	0	0	0	0	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	0	1	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	p1=cmp.eq(Rs,Rt); if (p1.new) jump:nt #r9:2
0	0	0	1	0	1	0	0	0	0	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	0	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	p0=cmp.eq(Rs,Rt); if (p0.new) jump:t #r9:2
0	0	0	1	0	1	0	0	0	0	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	1	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	p1=cmp.eq(Rs,Rt); if (p1.new) jump:t #r9:2
0	0	0	1	0	1	0	0	0	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	p0=cmp.eq(Rs,Rt); if (!p0.new) jump:nt #r9:2
0	0	0	1	0	1	0	0	0	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	0	1	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	p1=cmp.eq(Rs,Rt); if (!p1.new) jump:nt #r9:2
0	0	0	1	0	1	0	0	0	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	0	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	p0=cmp.eq(Rs,Rt); if (!p0.new) jump:t #r9:2
0	0	0	1	0	1	0	0	0	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	1	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	p1=cmp.eq(Rs,Rt); if (!p1.new) jump:t #r9:2
0	0	0	1	0	1	0	0	1	0	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	p0=cmp.gt(Rs,Rt); if (p0.new) jump:nt #r9:2
0	0	0	1	0	1	0	0	1	0	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	0	1	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	p1=cmp.gt(Rs,Rt); if (p1.new) jump:nt #r9:2
0	0	0	1	0	1	0	0	1	0	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	0	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	p0=cmp.gt(Rs,Rt); if (p0.new) jump:t #r9:2
0	0	0	1	0	1	0	0	1	0	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	1	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	p1=cmp.gt(Rs,Rt); if (p1.new) jump:t #r9:2
0	0	0	1	0	1	0	0	1	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	p0=cmp.gt(Rs,Rt); if (!p0.new) jump:nt #r9:2
0	0	0	1	0	1	0	0	1	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	0	1	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	p1=cmp.gt(Rs,Rt); if (!p1.new) jump:nt #r9:2
0	0	0	1	0	1	0	0	1	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	0	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	p0=cmp.gt(Rs,Rt); if (!p0.new) jump:t #r9:2
0	0	0	1	0	1	0	0	1	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	1	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	p1=cmp.gt(Rs,Rt); if (!p1.new) jump:t #r9:2
0	0	0	1	0	1	0	1	0	0	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	p0=cmp.gtu(Rs,Rt); if (p0.new) jump:nt #r9:2
0	0	0	1	0	1	0	1	0	0	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	0	1	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	p1=cmp.gtu(Rs,Rt); if (p1.new) jump:nt #r9:2
0	0	0	1	0	1	0	1	0	0	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	0	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	p0=cmp.gtu(Rs,Rt); if (p0.new) jump:t #r9:2
0	0	0	1	0	1	0	1	0	0	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	1	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	p1=cmp.gtu(Rs,Rt); if (p1.new) jump:t #r9:2
0	0	0	1	0	1	0	1	0	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	p0=cmp.gtu(Rs,Rt); if (!p0.new) jump:nt #r9:2
0	0	0	1	0	1	0	1	0	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	0	1	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	p1=cmp.gtu(Rs,Rt); if (!p1.new) jump:nt #r9:2
0	0	0	1	0	1	0	1	0	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	0	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	p0=cmp.gtu(Rs,Rt); if (!p0.new) jump:t #r9:2
0	0	0	1	0	1	0	1	0	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	1	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	p1=cmp.gtu(Rs,Rt); if (!p1.new) jump:t #r9:2

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
s4	Field to encode register s
t4	Field to encode register t

Jump to address

Change the program flow to a target address. This instruction changes the program counter to a target address that is relative to the PC address. The offset from the current PC address is contained in the instruction encoding.

A speculated jump instruction includes a hint ("taken" or "not taken") that specifies the expected value of the conditional expression. If the actual generated value of the predicate differs from this expected value, the jump instruction incurs a performance penalty.

This instruction can appear in slots 2 or 3.

Syntax	Behavior
`if ([!]Pu) jump #r15:2`	`Assembler mapped to: "if ([!]Pu) ""jump"":nt ""#r15:2"`

`if ([!]Pu) jump:<hint> #r15:2`	`if ([!]Pu[0]) {`
	`  apply_extension(#r);`
	`  #r = #r & ~PCALIGN_MASK;`
	`  PC = PC+#r;`
	`}`

`jump #r22:2`	`apply_extension(#r);`
	`#r = #r & ~PCALIGN_MASK;`
	`PC = PC + #r;`

Class: J (slots 0,1,2,3)

Notes

- This instruction can be conditionally executed based on the value of a predicate register. If the instruction is preceded by 'if Pn', the instruction only executes if the LSB of the predicate register is 1. Similarly, if the instruction is preceded by 'if !Pn', the instruction executes only if the LSB of Pn is 0.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse																
0	1	0	1	1	0	0	i	i	i	i	i	i	i	i	i	P	P	i	i	i	i	i	i	i	i	i	i	i	i	i	-	jump #r22:2
ICLASS																Parse			PT	DN		u2										
0	1	0	1	1	1	0	0	i	i	0	i	i	i	i	i	P	P	i	0	0	-	u	u	i	i	i	i	i	i	i	-	if (Pu) jump:nt #r15:2
0	1	0	1	1	1	0	0	i	i	0	i	i	i	i	i	P	P	i	1	0	-	u	u	i	i	i	i	i	i	i	-	if (Pu) jump:t #r15:2
0	1	0	1	1	1	0	0	i	i	1	i	i	i	i	i	P	P	i	0	0	-	u	u	i	i	i	i	i	i	i	-	if (!Pu) jump:nt #r15:2
0	1	0	1	1	1	0	0	i	i	1	i	i	i	i	i	P	P	i	1	0	-	u	u	i	i	i	i	i	i	i	-	if (!Pu) jump:t #r15:2

Field name	Description
ICLASS	Instruction class
DN	Dot-new
PT	Predict-taken
Parse	Packet/loop parse bits
u2	Field to encode register u

Jump to address conditioned on new predicate

Perform a speculated jump.

Jump if the LSB of the newly-generated predicate is true. The predicate must be generated in the same packet as the speculated jump instruction.

A speculated jump instruction includes a hint ("taken" or "not taken") that specifies the expected value of the conditional expression. If the actual generated value of the predicate differs from this expected value, the jump instruction incurs a performance penalty.

This instruction can appear in slots 2 or 3.

Syntax	Behavior
`if ([!]Pu.new) jump:<hint> #r15:2`	`if([!]Pu.new[0]){ apply_extension(#r); #r=#r & ~PCALIGN_MASK; PC=PC+#r; }`

Class: J (slots 0,1,2,3)

Notes

- This instruction can be conditionally executed based on the value of a predicate register. If the instruction is preceded by 'if Pn', the instruction only executes if the LSB of the predicate register is 1. Similarly, if the instruction is preceded by 'if !Pn', the instruction executes only if the LSB of Pn is 0.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse			PT	D N		u2										
0	1	0	1	1	1	0	0	i	i	0	i	i	i	i	i	P	P	i	0	1	-	u	u	i	i	i	i	i	i	i	-	if (Pu.new) jump:nt #r15:2
0	1	0	1	1	1	0	0	i	i	0	i	i	i	i	i	P	P	i	1	1	-	u	u	i	i	i	i	i	i	i	-	if (Pu.new) jump:t #r15:2
0	1	0	1	1	1	0	0	i	i	1	i	i	i	i	i	P	P	i	0	1	-	u	u	i	i	i	i	i	i	i	-	if (!Pu.new) jump:nt #r15:2
0	1	0	1	1	1	0	0	i	i	1	i	i	i	i	i	P	P	i	1	1	-	u	u	i	i	i	i	i	i	i	-	if (!Pu.new) jump:t #r15:2

Field name	Description
ICLASS	Instruction class
DN	Dot-new
PT	Predict-taken
Parse	Packet/loop parse bits
u2	Field to encode register u

Jump to address condition on register value

Perform register-conditional jump.

Jump if the specified register expression is true.

A register-conditional jump includes a hint ("taken" or "not taken") that specifies the expected value of the register expression. If the actual generated value of the expression differs from this expected value, the jump instruction incurs a performance penalty.

This instruction can appear only in slot 3.

Syntax	Behavior

`if (Rs!=#0) jump:nt #r13:2`	`if (Rs != 0) {`
	`  PC=PC+#r;`
	`}`

`if (Rs!=#0) jump:t #r13:2`	`if (Rs != 0) {`
	`  PC=PC+#r;`
	`}`

`if (Rs<=#0) jump:nt #r13:2`	`if (Rs<=0) {`
	`  PC=PC+#r;`
	`}`

`if (Rs<=#0) jump:t #r13:2`	`if (Rs<=0) {`
	`  PC=PC+#r;`
	`}`

`if (Rs==#0) jump:nt #r13:2`	`if (Rs == 0) {`
	`  PC=PC+#r;`
	`}`

`if (Rs==#0) jump:t #r13:2`	`if (Rs == 0) {`
	`  PC=PC+#r;`
	`}`

`if (Rs>=#0) jump:nt #r13:2`	`if (Rs>=0) {`
	`  PC=PC+#r;`
	`}`

`if (Rs>=#0) jump:t #r13:2`	`if (Rs>=0) {`
	`  PC=PC+#r;`
	`}`

Class: J (slot 3)

Notes

- This instruction will be deprecated in a future version.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					sm						s5					Parse																	
0	1	1	0	0	0	0	1	0	0	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (Rs!=#0) jump:nt #r13:2
0	1	1	0	0	0	0	1	0	0	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (Rs!=#0) jump:t #r13:2
0	1	1	0	0	0	0	1	0	1	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (Rs>=#0) jump:nt #r13:2
0	1	1	0	0	0	0	1	0	1	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (Rs>=#0) jump:t #r13:2
0	1	1	0	0	0	0	1	1	0	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (Rs==#0) jump:nt #r13:2
0	1	1	0	0	0	0	1	1	0	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (Rs==#0) jump:t #r13:2
0	1	1	0	0	0	0	1	1	1	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (Rs<=#0) jump:nt #r13:2
0	1	1	0	0	0	0	1	1	1	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (Rs<=#0) jump:t #r13:2

Field name	Description
`sm`	Supervisor mode only
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`s5`	Field to encode register s

Transfer and jump

Move an unsigned immediate or register value into a destination register and unconditionally jump. In assembly syntax, this instruction appears as two instructions in the packet, a transfer and a separate jump. The assembler may convert adjacent transfer and jump instructions into compound transfer-jump form.

Syntax	Behavior

`Rd = #U6 ; jump #r9:2`	`apply_extension(#r);`
	`#r = #r & ~PCALIGN_MASK;`
	`Rd = #U;`
	`PC = PC+#r;`

`Rd = Rs ; jump #r9:2`	`apply_extension(#r);`
	`#r = #r & ~PCALIGN_MASK;`
	`Rd = Rs;`
	`PC = PC+#r;`

Class: J (slots 2,3)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS												d4				Parse																
0	0	0	1	0	1	1	0	-	-	i	i	d	d	d	d	P	P	I	I	I	I	I	I	i	i	i	i	i	i	i	-	Rd=#U6 ; jump #r9:2
ICLASS												s4				Parse				d4												
0	0	0	1	0	1	1	1	-	-	i	i	s	s	s	s	P	P	-	-	d	d	d	d	i	i	i	i	i	i	i	-	Rd=Rs ; jump #r9:2

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d4	Field to encode register d
s4	Field to encode register s

11.5 LD

The LD instruction class includes load instructions, which are used to load values into registers.

LD instructions are executable on slot 0 and slot 1.

Load doubleword

Load a 64-bit doubleword from memory and place in a destination register pair.

Syntax	Behavior

`Rdd=memd(Re=#U6)`	`apply_extension(#U);`
	`EA =#U;`
	`Rdd = *EA;`
	`Re =#U;`

`Rdd=memd(Rs+#s11:3)`	`apply_extension(#s);`
	`EA =Rs+#s;`
	`Rdd = *EA;`

`Rdd=memd(Rs+Rt<<#u2)`	`EA =Rs+(Rt<<#u);` `Rdd = *EA;`

`Rdd=memd(Rt<<#u2+#U6)`	`apply_extension(#U);`
	`EA =#U+(Rt<<#u);`
	`Rdd = *EA;`

`Rdd=memd(Rx++#s4:3)`	`EA =Rx;`
	`Rx =Rx+#s;`
	`Rdd = *EA;`

`Rdd=memd(Rx++#s4:3:circ(Mu))`	`EA =Rx;`
	`Rx =Rx=circ_add(Rx,#s,MuV);`
	`Rdd = *EA;`

`Rdd=memd(Rx++I:circ(Mu))`	`EA =Rx;`
	`Rx =Rx=circ_add(Rx,I<<3,MuV);`
	`Rdd = *EA;`

`Rdd=memd(Rx++Mu)`	`EA =Rx;`
	`Rx =Rx+MuV;`
	`Rdd = *EA;`

`Rdd=memd(Rx++Mu:brev)`	`EA =Rx.h[1] | brev(Rx.h[0]);`
	`Rx =Rx+MuV;`
	`Rdd = *EA;`

`Rdd=memd(gp+#u16:3)`	`apply_extension(#u);`
	`EA =(Constant_extended ? (0) : GP)+#u;`
	`Rdd = *EA;`

Class: LD (slots 0,1)

Intrinsics

`Rdd=memd(Rx++#s4:3:circ(Mu))`	`Word32 Q6_R_memd_IM_circ(void** StartAddress, Word32 Is4_3, Word32 Mu, void* BaseAddress)`
`Rdd=memd(Rx++I:circ(Mu))`	`Word32 Q6_R_memd_M_circ(void** StartAddress, Word32 Mu, void* BaseAddress)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			t5								d5					
0	0	1	1	1	0	1	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rdd=memd(Rs+Rt<<#u2)
ICLASS								Type		U N						Parse											d5					
0	1	0	0	1	**i**	**i**	1	1	1	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rdd=memd(gp+#u16:3)
ICLASS				Amode			Type			U N	s5					Parse											d5					
1	0	0	1	0	**i**	**i**	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rdd=memd(Rs+#s11:3)
ICLASS				Amode			Type			U N	x5					Parse		u1									d5					
1	0	0	1	1	0	0	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	0	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rdd=memd(Rx++#s4:3:circ(Mu))
1	0	0	1	1	0	0	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	1	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rdd=memd(Rx++I:circ(Mu))
ICLASS				Amode			Type			U N	e5					Parse											d5					
1	0	0	1	1	0	1	1	1	1	0	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	1	**I**	**I**	**I**	**I**	**-**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rdd=memd(Re=#U6)
ICLASS				Amode			Type			U N	x5					Parse											d5					
1	0	0	1	1	0	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	0	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rdd=memd(Rx++#s4:3)
ICLASS				Amode			Type			U N	t5					Parse											d5					
1	0	0	1	1	1	0	1	1	1	0	**t**	**t**	**t**	**t**	**t**	**P**	**P**	**i**	1	**I**	**I**	**I**	**I**	**i**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rdd=memd(Rt<<#u2+#U6)
ICLASS				Amode			Type			U N	x5					Parse		u1									d5					
1	0	0	1	1	1	0	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rdd=memd(Rx++Mu)
1	0	0	1	1	1	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rdd=memd(Rx++Mu:brev)

Field name	Description
ICLASS	Instruction class
Amode	Amode
Type	Type
UN	Unsigned
Parse	Packet/loop parse bits
d5	Field to encode register d
e5	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
x5	Field to encode register x

Load-acquire doubleword

Load a 64-bit doubleword from memory and place in a destination register pair. The load-acquire memory operation is observed before any following memory operations (in program order) have been observed at the local point of serialization. A different order can be observed at the global point of serialization (see Memory ordering).

Syntax	Behavior
`Rdd=memd_aq(Rs)`	`EA = Rs;` `Rdd = *EA`

Class: LD (slots 0)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	s5					Parse											d5					
1	0	0	1	0	0	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	1	1	-	-	-	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=memd_aq(Rs)

Field name	Description
ICLASS	Instruction class
Amode	Amode
Type	Type
UN	Unsigned
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s

Load doubleword conditionally

Load a 64-bit doubleword from memory and place in a destination register pair.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior

`if ([!]Pt[.new]) Rdd=memd(#u6)`	`apply_extension(#u);`
	`EA=#u;`
	`if ([!]Pt[.new][0]) {`
	`  Rdd = *EA;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pt[.new]) Rdd=memd(Rs+#u6:3)`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`if ([!]Pt[.new][0]) {`
	`  Rdd = *EA;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pt[.new]) Rdd=memd(Rx++#s4:3)`	`EA=Rx;`
	`if([!]Pt[.new][0]){`
	`  Rx=Rx+#s;`
	`  Rdd = *EA;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new]) Rdd=memd(Rs+Rt<<#u2)`	`EA=Rs+(Rt<<#u);`
	`if ([!]Pv[.new][0]) {`
	`  Rdd = *EA;`
	`} else {`
	`  NOP;`
	`}`

Class: LD (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			t5								d5					
0	0	1	1	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (Pv) Rdd=memd(Rs+Rt<<#u2)
0	0	1	1	0	0	0	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (!Pv) Rdd=memd(Rs+Rt<<#u2)
0	0	1	1	0	0	1	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (Pv.new) Rdd=memd(Rs+Rt<<#u2)
0	0	1	1	0	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (!Pv.new) Rdd=memd(Rs+Rt<<#u2)
ICLASS					Sense	Pred New		Type		UN	s5					Parse			t2								d5					

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	0	0	0	0	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rdd=memd(Rs+#u6:3)
0	1	0	0	0	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rdd=memd(Rs+#u6:3)
0	1	0	0	0	1	0	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rdd=memd(Rs+#u6:3)
0	1	0	0	0	1	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rdd=memd(Rs+#u6:3)
ICLASS				Amode			Type			UN	x5					Parse					t2						d5					
1	0	0	1	1	0	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	0	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rdd=memd(Rx++#s4:3)
1	0	0	1	1	0	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	1	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rdd=memd(Rx++#s4:3)
1	0	0	1	1	0	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	0	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rdd=memd(Rx++#s4:3)
1	0	0	1	1	0	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	1	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rdd=memd(Rx++#s4:3)
ICLASS				Amode			Type			UN						Parse					t2						d5					
1	0	0	1	1	1	1	1	1	1	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	0	0	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rdd=memd(#u6)
1	0	0	1	1	1	1	1	1	1	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	0	1	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rdd=memd(#u6)
1	0	0	1	1	1	1	1	1	1	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	1	0	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rdd=memd(#u6)
1	0	0	1	1	1	1	1	1	1	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	1	1	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rdd=memd(#u6)

Field name	Description
ICLASS	Instruction class
Amode	Amode
Type	Type
UN	Unsigned
PredNew	PredNew
Sense	Sense
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t2	Field to encode register t
t5	Field to encode register t
v2	Field to encode register v
x5	Field to encode register x

Load byte

Load a signed byte from memory. The byte at the effective address in memory is placed in the least-significant eight bits of the destination register. The destination register is then sign-extended from 8 bits to 32.

Syntax	Behavior

`Rd=memb(Re=#U6)`	`apply_extension(#U);`
	`EA=#U;`
	`Rd = *EA;`
	`Re=#U;`

`Rd=memb(Rs+#s11:0)`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`Rd = *EA;`

`Rd=memb(Rs+Rt<<#u2)`	`EA=Rs+(Rt<<#u);` `Rd = *EA;`

`Rd=memb(Rt<<#u2+#U6)`	`apply_extension(#U);`
	`EA=#U+(Rt<<#u);`
	`Rd = *EA;`

`Rd=memb(Rx++#s4:0)`	`EA=Rx;`
	`Rx=Rx+#s;`
	`Rd = *EA;`

`Rd=memb(Rx++#s4:0:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`Rd = *EA;`

`Rd=memb(Rx++I:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<0,MuV);`
	`Rd = *EA;`

`Rd=memb(Rx++Mu)`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`Rd = *EA;`

`Rd=memb(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`Rd = *EA;`

`Rd=memb(gp+#u16:0)`	`apply_extension(#u);`
	`EA=(Constant_extended ? (0) : GP)+#u;`
	`Rd = *EA;`

Class: LD (slots 0,1)

Intrinsics

`Rd=memb(Rx++#s4:0:circ(Mu))`	`Word32 Q6_R_memb_IM_circ(void** StartAddress, Word32 Is4_0, Word32 Mu, void* BaseAddress)`
`Rd=memb(Rx++I:circ(Mu))`	`Word32 Q6_R_memb_M_circ(void** StartAddress, Word32 Mu, void* BaseAddress)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			t5								d5					
0	0	1	1	1	0	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memb(Rs+Rt<<#u2)
ICLASS								Type		UN						Parse											d5					
0	1	0	0	1	**i**	**i**	1	0	0	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memb(gp+#u16:0)
ICLASS				Amode			Type			UN	s5					Parse											d5					
1	0	0	1	0	**i**	**i**	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memb(Rs+#s11:0)
ICLASS				Amode			Type			UN	x5					Parse		u1									d5					
1	0	0	1	1	0	0	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	0	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memb(Rx++#s4:0:circ(Mu))
1	0	0	1	1	0	0	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	1	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memb(Rx++I:circ(Mu))
ICLASS				Amode			Type			UN	e5					Parse											d5					
1	0	0	1	1	0	1	1	0	0	0	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	1	**I**	**I**	**I**	**I**	**-**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rd=memb(Re=#U6)
ICLASS				Amode			Type			UN	x5					Parse											d5					
1	0	0	1	1	0	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memb(Rx++#s4:0)
ICLASS				Amode			Type			UN	t5					Parse											d5					
1	0	0	1	1	1	0	1	0	0	0	**t**	**t**	**t**	**t**	**t**	**P**	**P**	**i**	1	**I**	**I**	**I**	**I**	**i**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rd=memb(Rt<<#u2+#U6)
ICLASS				Amode			Type			UN	x5					Parse		u1									d5					
1	0	0	1	1	1	0	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memb(Rx++Mu)
1	0	0	1	1	1	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memb(Rx++Mu:brev)

Field name	Description
ICLASS	Instruction class
Amode	Amode
Type	Type
UN	Unsigned
Parse	Packet/loop parse bits
d5	Field to encode register d
e5	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
x5	Field to encode register x

Load byte conditionally

Load a signed byte from memory. The byte at the effective address in memory is placed in the least-significant 8 bits of the destination register. The destination register is then sign-extended from 8 bits to 32.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior

`if ([!]Pt[.new])`
`Rd=memb(#u6)`
	`apply_extension(#u);`
	`EA=#u;`
	`if ([!]Pt[.new][0]) {`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pt[.new])`
`Rd=memb(Rs+#u6:0)`
	`apply_extension(#u);`
	`EA=Rs+#u;`
	`if ([!]Pt[.new][0]) {`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pt[.new])`
`Rd=memb(Rx++#s4:0)`
	`EA=Rx;`
	`if([!]Pt[.new][0]){`
	`  Rx=Rx+#s;`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new])`
`Rd=memb(Rs+Rt<<#u2)`
	`EA=Rs+(Rt<<#u);`
	`if ([!]Pv[.new][0]) {`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`}`

Class: LD (slots 0,1)

Encoding

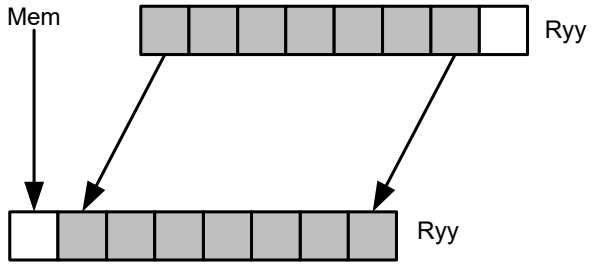
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			t5								d5					
0	0	1	1	0	0	0	0	0	0	0	s	s	s	s	s	P	P	i	t	t	t	t	t	i	v	v	d	d	d	d	d	if (Pv) Rd=memb(Rs+Rt<<#u2)
0	0	1	1	0	0	0	1	0	0	0	s	s	s	s	s	P	P	i	t	t	t	t	t	i	v	v	d	d	d	d	d	if (!Pv) Rd=memb(Rs+Rt<<#u2)
0	0	1	1	0	0	1	0	0	0	0	s	s	s	s	s	P	P	i	t	t	t	t	t	i	v	v	d	d	d	d	d	if (Pv.new) Rd=memb(Rs+Rt<<#u2)
0	0	1	1	0	0	1	1	0	0	0	s	s	s	s	s	P	P	i	t	t	t	t	t	i	v	v	d	d	d	d	d	if (!Pv.new) Rd=memb(Rs+Rt<<#u2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					Sense	PredNew		Type		UN	s5					Parse			t2								d5					
0	1	0	0	0	0	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memb(Rs+#u6:0)
0	1	0	0	0	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memb(Rs+#u6:0)
0	1	0	0	0	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memb(Rs+#u6:0)
0	1	0	0	0	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memb(Rs+#u6:0)
ICLASS				Amode			Type			UN	x5					Parse					t2						d5					
1	0	0	1	1	0	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	0	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memb(Rx++#s4:0)
1	0	0	1	1	0	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	1	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memb(Rx++#s4:0)
1	0	0	1	1	0	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	0	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memb(Rx++#s4:0)
1	0	0	1	1	0	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	1	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memb(Rx++#s4:0)
ICLASS				Amode			Type			UN						Parse					t2						d5					
1	0	0	1	1	1	1	1	0	0	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	0	0	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memb(#u6)
1	0	0	1	1	1	1	1	0	0	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	0	1	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memb(#u6)
1	0	0	1	1	1	1	1	0	0	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	1	0	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memb(#u6)
1	0	0	1	1	1	1	1	0	0	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	1	1	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memb(#u6)

Field name	Description
ICLASS	Instruction class
Amode	Amode
Type	Type
UN	Unsigned
PredNew	PredNew
Sense	Sense
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t2	Field to encode register t
t5	Field to encode register t
v2	Field to encode register v
x5	Field to encode register x

Load byte into shifted vector

Shift a 64-bit vector right by one byte. Insert a byte from memory into the vacated upper byte of the vector.



Syntax	Behavior

`Ryy=memb_fifo(Re=#U6)`	`apply_extension(#U);`
	`EA = #U;`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>8)|(tmpV<<56);`
	`}  Re = #U;`

`Ryy=memb_fifo(Rs)`	`Assembler mapped to: "Ryy = memb_fifo""(Rs + #0)"`

`Ryy=memb_fifo(Rs+#s11:0)`	`apply_extension(#s);`
	`EA = Rs +#s;`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>8)|(tmpV<<56);`
	`}`

`Ryy=memb_fifo(Rt<<#u2+#U6)`	`apply_extension(#U);`
	`EA = #U+(Rt<<#u);`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>8)|(tmpV<<56);`
	`}`

`Ryy=memb_fifo(Rx ++ #s4:0)`	`EA = Rx;`
	`Rx = Rx+#s;`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>8)|(tmpV<<56);`
	`}`

`Ryy=memb_fifo(Rx ++ #s4:0:circ(Mu))`	`EA = Rx;`
	`Rx = Rx = circ_add(Rx, #s,MuV);`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>8)|(tmpV<<56);`
	`}`

`Ryy=memb_fifo(Rx ++ I:circ(Mu))`	`EA = Rx;`
	`Rx = Rx=circ_add(Rx,I<<0,MuV);`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>8)|(tmpV<<56);`
	`}`

Syntax	Behavior

`Ryy=memb_fifo(Rx ++ Mu)`	`EA = Rx;`
	`Rx = Rx+MuV;`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>8)|(tmpV<<56);`
	`}`

`Ryy=memb_fifo(Rx ++ Mu:brev)`	`EA = Rx.h[1] | brev(Rx.h[0]);`
	`Rx = Rx+MuV;`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>8)|(tmpV<<56);`
	`}`

Class: LD (slots 0,1)

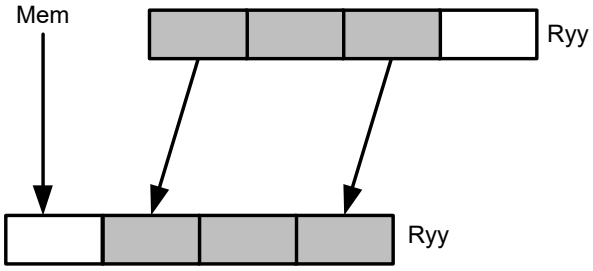
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	s5					Parse											y5					
1	0	0	1	0	**i**	**i**	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**y**	**y**	**y**	**y**	**y**	Ryy=memb_fifo(Rs+#s11:0)
ICLASS				Amode			Type			UN	x5					Parse		u1									y5					
1	0	0	1	1	0	0	0	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	0	**i**	**i**	**i**	**i**	**y**	**y**	**y**	**y**	**y**	Ryy=memb_fifo(Rx++#s4:0:circ(Mu))
1	0	0	1	1	0	0	0	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	1	**-**	0	**-**	**-**	**y**	**y**	**y**	**y**	**y**	Ryy=memb_fifo(Rx++I:circ(Mu))
ICLASS				Amode			Type			UN	e5					Parse											y5					
1	0	0	1	1	0	1	0	1	0	0	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	1	**I**	**I**	**I**	**I**	**-**	**I**	**I**	**y**	**y**	**y**	**y**	**y**	Ryy=memb_fifo(Re=#U6)
ICLASS				Amode			Type			UN	x5					Parse											y5					
1	0	0	1	1	0	1	0	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**y**	**y**	**y**	**y**	**y**	Ryy=memb_fifo(Rx++#s4:0)
ICLASS				Amode			Type			UN	t5					Parse											y5					
1	0	0	1	1	1	0	0	1	0	0	**t**	**t**	**t**	**t**	**t**	**P**	**P**	**i**	1	**I**	**I**	**I**	**I**	**i**	**I**	**I**	**y**	**y**	**y**	**y**	**y**	Ryy=memb_fifo(Rt<<#u2+#U6)
ICLASS				Amode			Type			UN	x5					Parse		u1									y5					
1	0	0	1	1	1	0	0	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**y**	**y**	**y**	**y**	**y**	Ryy=memb_fifo(Rx++Mu)
1	0	0	1	1	1	1	0	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**y**	**y**	**y**	**y**	**y**	Ryy=memb_fifo(Rx++Mu:brev)

Field name	Description
ICLASS	Instruction class
Amode	Amode
Type	Type
UN	Unsigned
Parse	Packet/loop parse bits
e5	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
x5	Field to encode register x
y5	Field to encode register y

Load half into shifted vector

Shift a 64-bit vector right by one halfword. Insert a halfword from memory into the vacated upper halfword of the vector.



Syntax	Behavior

`Ryy=memh_fifo(Re=#U6)`	`apply_extension(#U);`
	`EA =#U;`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>16)|(tmpV<<48);`
	`}`
	`Re = #U;`

`Ryy=memh_fifo(Rs)`	`Assembler mapped to: "Ryy=memh_fifo""(Rs + #0)"`

`Ryy=memh_fifo(Rs+#s11:1)`	`apply_extension(#s);`
	`EA =Rs+#s;`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>16)|(tmpV<<48);`
	`}`

`Ryy=memh_fifo(Rt<<#u2+#U6)`	`apply_extension(#U);`
	`EA =#U+(Rt<<#u);`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>16)|(tmpV<<48);`
	`}`

`Ryy=memh_fifo(Rx++#s4:1)`	`EA=Rx;`
	`Rx=Rx+#s;`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>16)|(tmpV<<48);`
	`}`

`Ryy=memh_fifo(Rx ++ #s4:1:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>16)|(tmpV<<48);`
	`}`

Syntax	Behavior

`Ryy=memh_fifo(Rx ++ I:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<1,MuV);`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>16)|(tmpV<<48);`
	`}`

`Ryy = memh_fifo(Rx++Mu)`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>16)|(tmpV<<48);`
	`}`

`Ryy = memh_fifo(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>16)|(tmpV<<48);`
	`}`

Class: LD (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	s5					Parse											y5					
1	0	0	1	0	**i**	**i**	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**y**	**y**	**y**	**y**	**y**	Ryy=memh_fifo(Rs+#s11:1)
ICLASS				Amode			Type			UN	x5					Parse		u1									y5					
1	0	0	1	1	0	0	0	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	0	**i**	**i**	**i**	**i**	**y**	**y**	**y**	**y**	**y**	Ryy=memh_fifo(Rx++#s4:1:circ(Mu))
1	0	0	1	1	0	0	0	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	1	**-**	0	**-**	**-**	**y**	**y**	**y**	**y**	**y**	Ryy=memh_fifo(Rx++I:circ(Mu))
ICLASS				Amode			Type			UN	e5					Parse											y5					
1	0	0	1	1	0	1	0	0	1	0	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	1	**I**	**I**	**I**	**I**	**-**	**I**	**I**	**y**	**y**	**y**	**y**	**y**	Ryy=memh_fifo(Re=#U6)
ICLASS				Amode			Type			UN	x5					Parse											y5					
1	0	0	1	1	0	1	0	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**y**	**y**	**y**	**y**	**y**	Ryy=memh_fifo(Rx++#s4:1)
ICLASS				Amode			Type			UN	t5					Parse											y5					
1	0	0	1	1	1	0	0	0	1	0	**t**	**t**	**t**	**t**	**t**	**P**	**P**	**i**	1	**I**	**I**	**I**	**I**	**i**	**I**	**I**	**y**	**y**	**y**	**y**	**y**	Ryy=memh_fifo(Rt<<#u2+#U6)
ICLASS				Amode			Type			UN	x5					Parse		u1									y5					
1	0	0	1	1	1	0	0	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**y**	**y**	**y**	**y**	**y**	Ryy=memh_fifo(Rx++Mu)
1	0	0	1	1	1	1	0	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**y**	**y**	**y**	**y**	**y**	Ryy=memh_fifo(Rx++Mu:brev)

Field name	Description
ICLASS	Instruction class
Amode	Amode
Type	Type
UN	Unsigned
Parse	Packet/loop parse bits
e5	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
x5	Field to encode register x
y5	Field to encode register y

Load halfword

Load a signed halfword from memory. The 16-bit halfword at the effective address in memory is placed in the least-significant 16 bits of the destination register. The destination register is then sign-extended from 16 bits to 32.

Syntax	Behavior

`Rd=memh(Re=#U6)`	`apply_extension(#U);`
	`EA=#U;`
	`Rd = *EA;`
	`Re=#U;`

`Rd=memh(Rs+#s11:1)`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`Rd = *EA;`

`Rd=memh(Rs+Rt<<#u2)`	`EA=Rs+(Rt<<#u);` `Rd = *EA;`

`Rd=memh(Rt<<#u2+#U6)`	`apply_extension(#U);`
	`EA=#U+(Rt<<#u);`
	`Rd = *EA;`

`Rd=memh(Rx++#s4:1)`	`EA=Rx;`
	`Rx=Rx+#s;`
	`Rd = *EA;`

`Rd=memh(Rx++#s4:1:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`Rd = *EA;`

`Rd=memh(Rx++I:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<1,MuV);`
	`Rd = *EA;`

`Rd=memh(Rx++Mu)`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`Rd = *EA;`

`Rd=memh(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`Rd = *EA;`

`Rd=memh(gp+#u16:1)`	`apply_extension(#u);`
	`EA=(Constant_extended ? (0) : GP)+#u;`
	`Rd = *EA;`

Class: LD (slots 0,1)

Intrinsics

`Rd=memh(Rx++#s4:1:circ(Mu))`	`Word32 Q6_R_memh_IM_circ(void** StartAddress, Word32 Is4_1, Word32 Mu, void* BaseAddress)`
`Rd=memh(Rx++I:circ(Mu))`	`Word32 Q6_R_memh_M_circ(void** StartAddress, Word32 Mu, void* BaseAddress)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			t5								d5					
0	0	1	1	1	0	1	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memh(Rs+Rt<<#u2)
ICLASS								Type		UN						Parse											d5					
0	1	0	0	1	**i**	**i**	1	0	1	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memh(gp+#u16:1)
ICLASS				Amode			Type			UN	s5					Parse											d5					
1	0	0	1	0	**i**	**i**	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memh(Rs+#s11:1)
ICLASS				Amode			Type			UN	x5					Parse		u1									d5					
1	0	0	1	1	0	0	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	0	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memh(Rx++#s4:1:circ(Mu))
1	0	0	1	1	0	0	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	1	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memh(Rx++I:circ(Mu))
ICLASS				Amode			Type			UN	e5					Parse											d5					
1	0	0	1	1	0	1	1	0	1	0	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	1	**I**	**I**	**I**	**I**	**-**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rd=memh(Re=#U6)
ICLASS				Amode			Type			UN	x5					Parse											d5					
1	0	0	1	1	0	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memh(Rx++#s4:1)
ICLASS				Amode			Type			UN	t5					Parse											d5					
1	0	0	1	1	1	0	1	0	1	0	**t**	**t**	**t**	**t**	**t**	**P**	**P**	**i**	1	**I**	**I**	**I**	**I**	**i**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rd=memh(Rt<<#u2+#U6)
ICLASS				Amode			Type			UN	x5					Parse		u1									d5					
1	0	0	1	1	1	0	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memh(Rx++Mu)
1	0	0	1	1	1	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memh(Rx++Mu:brev)

Field name	Description
ICLASS	Instruction class
Amode	Amode
Type	Type
UN	Unsigned
Parse	Packet/loop parse bits
d5	Field to encode register d
e5	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
x5	Field to encode register x

Load halfword conditionally

Load a signed halfword from memory. The 16-bit halfword at the effective address in memory is placed in the least-significant 16 bits of the destination register. The destination register is then sign-extended from 16 bits to 32.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior

`if ([!]Pt[.new])`
`Rd=memh(#u6)`
	`apply_extension(#u);`
	`EA=#u;`
	`if ([!]Pt[.new][0]) {`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pt[.new])`
`Rd=memh(Rs+#u6:1)`
	`apply_extension(#u);`
	`EA=Rs+#u;`
	`if ([!]Pt[.new][0]) {`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pt[.new])`
`Rd=memh(Rx++#s4:1)`
	`EA=Rx;`
	`if([!]Pt[.new][0]){`
	`  Rx=Rx+#s;`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new])`
`Rd=memh(Rs+Rt<<#u2)`
	`EA=Rs+(Rt<<#u);`
	`if ([!]Pv[.new][0]) {`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`}`

Class: LD (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			t5								d5						
0	0	1	1	0	0	0	0	0	1	0	s	s	s	s	s	P	P	i	t	t	t	t	t	i	v	v	d	d	d	d	d	if (Pv) Rd=memh(Rs+Rt<<#u2)
0	0	1	1	0	0	0	1	0	1	0	s	s	s	s	s	P	P	i	t	t	t	t	t	i	v	v	d	d	d	d	d	if (!Pv) Rd=memh(Rs+Rt<<#u2)
0	0	1	1	0	0	1	0	0	1	0	s	s	s	s	s	P	P	i	t	t	t	t	t	i	v	v	d	d	d	d	d	if (Pv.new) Rd=memh(Rs+Rt<<#u2)
0	0	1	1	0	0	1	1	0	1	0	s	s	s	s	s	P	P	i	t	t	t	t	t	i	v	v	d	d	d	d	d	if (!Pv.new) Rd=memh(Rs+Rt<<#u2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					Sense	PredNew		Type		UN	s5					Parse			t2								d5					
0	1	0	0	0	0	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memh(Rs+#u6:1)
0	1	0	0	0	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memh(Rs+#u6:1)
0	1	0	0	0	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memh(Rs+#u6:1)
0	1	0	0	0	1	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memh(Rs+#u6:1)
ICLASS				Amode			Type			UN	x5					Parse					t2						d5					
1	0	0	1	1	0	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	0	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memh(Rx++#s4:1)
1	0	0	1	1	0	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	1	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memh(Rx++#s4:1)
1	0	0	1	1	0	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	0	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memh(Rx++#s4:1)
1	0	0	1	1	0	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	1	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memh(Rx++#s4:1)
ICLASS				Amode			Type			UN						Parse					t2						d5					
1	0	0	1	1	1	1	1	0	1	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	0	0	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memh(#u6)
1	0	0	1	1	1	1	1	0	1	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	0	1	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memh(#u6)
1	0	0	1	1	1	1	1	0	1	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	1	0	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memh(#u6)
1	0	0	1	1	1	1	1	0	1	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	1	1	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memh(#u6)

Field name	Description
CLASS	Instruction class
Amode	Amode
Type	Type
UN	Unsigned
PredNew	PredNew
Sense	Sense
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t2	Field to encode register t
t5	Field to encode register t
v2	Field to encode register v
x5	Field to encode register x

Load unsigned byte

Load an unsigned byte from memory. The byte at the effective address in memory is placed in the least-significant 8 bits of the destination register. The destination register is then zero-extended from 8 bits to 32.

Syntax	Behavior

`Rd=memub(Re=#U6)`	`apply_extension(#U);`
	`EA=#U;`
	`Rd = *EA;`
	`Re=#U;`

`Rd=memub(Rs+#s11:0)`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`Rd = *EA;`

`Rd=memub(Rs+Rt<<#u2)`	`EA=Rs+(Rt<<#u);` `Rd = *EA;`

`Rd=memub(Rt<<#u2+#U6)`	`apply_extension(#U);`
	`EA=#U+(Rt<<#u);`
	`Rd = *EA;`

`Rd=memub(Rx++#s4:0)`	`EA=Rx;`
	`Rx=Rx+#s;`
	`Rd = *EA;`

`Rd=memub(Rx++#s4:0:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`Rd = *EA;`

`Rd=memub(Rx++I:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<0,MuV);`
	`Rd = *EA;`

`Rd=memub(Rx++Mu)`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`Rd = *EA;`

`Rd=memub(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`Rd = *EA;`

`Rd=memub(gp+#u16:0)`	`apply_extension(#u);`
	`EA=(Constant_extended ? (0) : GP)+#u;`
	`Rd = *EA;`

Class: LD (slots 0,1)

Intrinsics

`Rd=memub(Rx++#s4:0:circ(Mu))`	`Word32 Q6_R_memub_IM_circ(void** StartAddress, Word32 Is4_0, Word32 Mu, void* BaseAddress)`
`Rd=memub(Rx++I:circ(Mu))`	`Word32 Q6_R_memub_M_circ(void** StartAddress, Word32 Mu, void* BaseAddress)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			t5								d5					
0	0	1	1	1	0	1	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memub(Rs+Rt<<#u2)
ICLASS								Type		U N						Parse											d5					
0	1	0	0	1	**i**	**i**	1	0	0	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memub(gp+#u16:0)
ICLASS				Amode			Type			U N	s5					Parse											d5					
1	0	0	1	0	**i**	**i**	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memub(Rs+#s11:0)
ICLASS				Amode			Type			U N	x5					Parse		u1									d5					
1	0	0	1	1	0	0	1	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	0	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memub(Rx++#s4:0:circ (Mu))
1	0	0	1	1	0	0	1	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	1	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memub(Rx++I:circ(Mu))
ICLASS				Amode			Type			U N	e5					Parse											d5					
1	0	0	1	1	0	1	1	0	0	1	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	1	**I**	**I**	**I**	**I**	**-**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rd=memub(Re=#U6)
ICLASS				Amode			Type			U N	x5					Parse											d5					
1	0	0	1	1	0	1	1	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memub(Rx++#s4:0)
ICLASS				Amode			Type			U N	t5					Parse											d5					
1	0	0	1	1	1	0	1	0	0	1	**t**	**t**	**t**	**t**	**t**	**P**	**P**	**i**	1	**I**	**I**	**I**	**I**	**i**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rd=memub(Rt<<#u2+#U6)
ICLASS				Amode			Type			U N	x5					Parse		u1									d5					
1	0	0	1	1	1	0	1	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memub(Rx++Mu)
1	0	0	1	1	1	1	1	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memub(Rx++Mu:brev)

Field name	Description
CLASS	Instruction class
Amode	Amode
Type	Type
UN	Unsigned
Parse	Packet/loop parse bits
d5	Field to encode register d
e5	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
x5	Field to encode register x

Load unsigned byte conditionally

Load an unsigned byte from memory. The byte at the effective address in memory is placed in the least-significant 8 bits of the destination register. The destination register is then zero-extended from 8 bits to 32.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior

`if ([!]Pt[.new])`
`Rd=memub(#u6)`
	`apply_extension(#u);`
	`EA=#u;`
	`if ([!]Pt[.new][0]) {`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pt[.new])`
`Rd=memub(Rs+#u6:0)`
	`apply_extension(#u);`
	`EA=Rs+#u;`
	`if ([!]Pt[.new][0]) {`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pt[.new])`
`Rd=memub(Rx++#s4:0)`
	`EA=Rx;`
	`if([!]Pt[.new][0]){`
	`  Rx=Rx+#s;`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new])`
`Rd=memub(Rs+Rt<<#u2)`
	`EA=Rs+(Rt<<#u);`
	`if ([!]Pv[.new][0]) {`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`}`

Class: LD (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			t5								d5						
0	0	1	1	0	0	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (Pv) Rd=memub(Rs+Rt<<#u2)
0	0	1	1	0	0	0	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (!Pv) Rd=memub(Rs+Rt<<#u2)
0	0	1	1	0	0	1	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (Pv.new) Rd=memub(Rs+Rt<<#u2)
0	0	1	1	0	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (!Pv.new) Rd=memub(Rs+Rt<<#u2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					Sense	PredNew		Type		UN	s5					Parse			t2								d5					
0	1	0	0	0	0	0	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memub(Rs+#u6:0)
0	1	0	0	0	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memub(Rs+#u6:0)
0	1	0	0	0	1	0	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memub(Rs+#u6:0)
0	1	0	0	0	1	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memub(Rs+#u6:0)
ICLASS				Amode			Type			UN	x5					Parse					t2						d5					
1	0	0	1	1	0	1	1	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	0	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memub(Rx++#s4:0)
1	0	0	1	1	0	1	1	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	1	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memub(Rx++#s4:0)
1	0	0	1	1	0	1	1	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	0	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memub(Rx++#s4:0)
1	0	0	1	1	0	1	1	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	1	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memub(Rx++#s4:0)
ICLASS				Amode			Type			UN						Parse					t2						d5					
1	0	0	1	1	1	1	1	0	0	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	0	0	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memub(#u6)
1	0	0	1	1	1	1	1	0	0	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	0	1	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memub(#u6)
1	0	0	1	1	1	1	1	0	0	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	1	0	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memub(#u6)
1	0	0	1	1	1	1	1	0	0	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	1	1	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memub(#u6)

Field name	Description
ICLASS	Instruction class
Amode	Amode
Type	Type
UN	Unsigned
PredNew	PredNew
Sense	Sense
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t2	Field to encode register t
t5	Field to encode register t
v2	Field to encode register v
x5	Field to encode register x

Load unsigned halfword

Load an unsigned halfword from memory. The 16-bit halfword at the effective address in memory is placed in the least-significant 16 bits of the destination register. The destination register is zero-extended from 16 bits to 32.

Syntax	Behavior

`Rd=memuh(Re=#U6)`	`apply_extension(#U);`
	`EA=#U;`
	`Rd = *EA;`
	`Re=#U;`

`Rd=memuh(Rs+#s11:1)`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`Rd = *EA;`

`Rd=memuh(Rs+Rt<<#u2)`	`EA=Rs+(Rt<<#u);` `Rd = *EA;`

`Rd=memuh(Rt<<#u2+#U6)`	`apply_extension(#U);`
	`EA=#U+(Rt<<#u);`
	`Rd = *EA;`

`Rd=memuh(Rx++#s4:1)`	`EA=Rx;`
	`Rx=Rx+#s;`
	`Rd = *EA;`

`Rd=memuh(Rx++#s4:1:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`Rd = *EA;`

`Rd=memuh(Rx++I:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<1,MuV);`
	`Rd = *EA;`

`Rd=memuh(Rx++Mu)`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`Rd = *EA;`

`Rd=memuh(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`Rd = *EA;`

`Rd=memuh(gp+#u16:1)`	`apply_extension(#u);`
	`EA=(Constant_extended ? (0) : GP)+#u;`
	`Rd = *EA;`

Class: LD (slots 0,1)

Intrinsics

`Rd=memuh(Rx++#s4:1:circ(Mu))`	`Word32 Q6_R_memuh_IM_circ(void** StartAddress, Word32 Is4_1, Word32 Mu, void* BaseAddress)`
`Rd=memuh(Rx++I:circ(Mu))`	`Word32 Q6_R_memuh_M_circ(void** StartAddress, Word32 Mu, void* BaseAddress)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			t5								d5					
0	0	1	1	1	0	1	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memuh(Rs+Rt<<#u2)
ICLASS								Type		UN						Parse											d5					
0	1	0	0	1	**i**	**i**	1	0	1	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memuh(gp+#u16:1)
ICLASS				Amode			Type			UN	s5					Parse											d5					
1	0	0	1	0	**i**	**i**	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memuh(Rs+#s11:1)
ICLASS				Amode			Type			UN	x5					Parse		u1									d5					
1	0	0	1	1	0	0	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	0	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memuh(Rx++#s4:1:circ(Mu))
1	0	0	1	1	0	0	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	1	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memuh(Rx++I:circ(Mu))
ICLASS				Amode			Type			UN	e5					Parse											d5					
1	0	0	1	1	0	1	1	0	1	1	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	1	**I**	**I**	**I**	**I**	**-**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rd=memuh(Re=#U6)
ICLASS				Amode			Type			UN	x5					Parse											d5					
1	0	0	1	1	0	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memuh(Rx++#s4:1)
ICLASS				Amode			Type			UN	t5					Parse											d5					
1	0	0	1	1	1	0	1	0	1	1	**t**	**t**	**t**	**t**	**t**	**P**	**P**	**i**	1	**I**	**I**	**I**	**I**	**i**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rd=memuh(Rt<<#u2+#U6)
ICLASS				Amode			Type			UN	x5					Parse		u1									d5					
1	0	0	1	1	1	0	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memuh(Rx++Mu)
1	0	0	1	1	1	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memuh(Rx++Mu:brev)

Field name	Description
CLASS	Instruction class
Amode	Amode
Type	Type
UN	Unsigned
Parse	Packet/loop parse bits
d5	Field to encode register d
e5	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
x5	Field to encode register x

Load unsigned halfword conditionally

Load an unsigned halfword from memory. The 16-bit halfword at the effective address in memory is placed in the least-significant 16 bits of the destination register. The destination register is zero-extended from 16 bits to 32.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior

`if ([!]Pt[.new]) Rd=memuh(#u6)`	`apply_extension(#u);`
	`EA=#u;`
	`if ([!]Pt[.new][0]) {`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pt[.new]) Rd=memuh(Rs+#u6:1)`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`if ([!]Pt[.new][0]) {`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pt[.new]) Rd=memuh(Rx++#s4:1)`	`EA=Rx;`
	`if([!]Pt[.new][0]){`
	`  Rx=Rx+#s;`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new]) Rd=memuh(Rs+Rt<<#u2)`	`EA=Rs+(Rt<<#u);`
	`if ([!]Pv[.new][0]) {`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`}`

Class: LD (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			t5								d5					
0	0	1	1	0	0	0	0	0	1	1	s	s	s	s	s	P	P	i	t	t	t	t	t	i	v	v	d	d	d	d	d	if (Pv) Rd=memuh(Rs+Rt<<#u2)
0	0	1	1	0	0	0	1	0	1	1	s	s	s	s	s	P	P	i	t	t	t	t	t	i	v	v	d	d	d	d	d	if (!Pv) Rd=memuh(Rs+Rt<<#u2)
0	0	1	1	0	0	1	0	0	1	1	s	s	s	s	s	P	P	i	t	t	t	t	t	i	v	v	d	d	d	d	d	if (Pv.new) Rd=memuh(Rs+Rt<<#u2)
0	0	1	1	0	0	1	1	0	1	1	s	s	s	s	s	P	P	i	t	t	t	t	t	i	v	v	d	d	d	d	d	if (!Pv.new) Rd=memuh(Rs+Rt<<#u2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					Sense	PredNew		Type		UN	s5					Parse			t2								d5					
0	1	0	0	0	0	0	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memuh(Rs+#u6:1)
0	1	0	0	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memuh(Rs+#u6:1)
0	1	0	0	0	1	0	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memuh(Rs+#u6:1)
0	1	0	0	0	1	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memuh(Rs+#u6:1)
ICLASS				Amode			Type			UN	x5					Parse					t2						d5					
1	0	0	1	1	0	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	0	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memuh(Rx++#s4:1)
1	0	0	1	1	0	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	1	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memuh(Rx++#s4:1)
1	0	0	1	1	0	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	0	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memuh(Rx++#s4:1)
1	0	0	1	1	0	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	1	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memuh(Rx++#s4:1)
ICLASS				Amode			Type			UN						Parse					t2						d5					
1	0	0	1	1	1	1	1	0	1	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	0	0	**t**	**t**	**i**	1	-	-	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memuh(#u6)
1	0	0	1	1	1	1	1	0	1	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	0	1	**t**	**t**	**i**	1	-	-	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memuh(#u6)
1	0	0	1	1	1	1	1	0	1	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	1	0	**t**	**t**	**i**	1	-	-	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memuh(#u6)
1	0	0	1	1	1	1	1	0	1	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	1	1	**t**	**t**	**i**	1	-	-	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memuh(#u6)

Field name	Description
ICLASS	Instruction class
Amode	Amode
Type	Type
UN	Unsigned
PredNew	PredNew
Sense	Sense
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t2	Field to encode register t
t5	Field to encode register t
v2	Field to encode register v
x5	Field to encode register x

Load word

Load a 32-bit word from memory and place in a destination register.

Syntax	Behavior

`Rd=memw(Re=#U6)`	`apply_extension(#U);`
	`EA=#U;`
	`Rd = *EA;`
	`Re=#U;`

`Rd=memw(Rs+#s11:2)`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`Rd = *EA;`

`Rd=memw(Rs+Rt<<#u2)`	`EA=Rs+(Rt<<#u);` `Rd = *EA;`

`Rd=memw(Rt<<#u2+#U6)`	`apply_extension(#U);`
	`EA=#U+(Rt<<#u);`
	`Rd = *EA;`

`Rd=memw(Rx++#s4:2)`	`EA=Rx;`
	`Rx=Rx+#s;`
	`Rd = *EA;`

`Rd=memw(Rx++#s4:2:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`Rd = *EA;`

`Rd=memw(Rx++I:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<2,MuV);`
	`Rd = *EA;`

`Rd=memw(Rx++Mu)`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`Rd = *EA;`

`Rd=memw(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`Rd = *EA;`

`Rd=memw(gp+#u16:2)`	`apply_extension(#u);`
	`EA=(Constant_extended ? (0) : GP)+#u;`
	`Rd = *EA;`

Class: LD (slots 0,1)

Intrinsics

`Rd=memw(Rx++#s4:2:circ(Mu))`	`Word32 Q6_R_memw_IM_circ(void** StartAddress, Word32 Is4_2, Word32 Mu, void* BaseAddress)`
`Rd=memw(Rx++I:circ(Mu))`	`Word32 Q6_R_memw_M_circ(void** StartAddress, Word32 Mu, void* BaseAddress)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			t5								d5					
0	0	1	1	1	0	1	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memw(Rs+Rt<<#u2)
ICLASS								Type		UN						Parse											d5					
0	1	0	0	1	**i**	**i**	1	1	0	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memw(gp+#u16:2)
ICLASS				Amode			Type			UN	s5					Parse											d5					
1	0	0	1	0	**i**	**i**	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memw(Rs+#s11:2)
ICLASS				Amode			Type			UN	x5					Parse		u1									d5					
1	0	0	1	1	0	0	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	0	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memw(Rx++#s4:2:circ(Mu))
1	0	0	1	1	0	0	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	1	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memw(Rx++I:circ(Mu))
ICLASS				Amode			Type			UN	e5					Parse											d5					
1	0	0	1	1	0	1	1	1	0	0	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	1	**I**	**I**	**I**	**I**	**-**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rd=memw(Re=#U6)
ICLASS				Amode			Type			UN	x5					Parse											d5					
1	0	0	1	1	0	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memw(Rx++#s4:2)
ICLASS				Amode			Type			UN	t5					Parse											d5					
1	0	0	1	1	1	0	1	1	0	0	**t**	**t**	**t**	**t**	**t**	**P**	**P**	**i**	1	**I**	**I**	**I**	**I**	**i**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rd=memw(Rt<<#u2+#U6)
ICLASS				Amode			Type			UN	x5					Parse		u1									d5					
1	0	0	1	1	1	0	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memw(Rx++Mu)
1	0	0	1	1	1	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memw(Rx++Mu:brev)

Field name	Description
CLASS	Instruction class
Amode	Amode
Type	Type
UN	Unsigned
Parse	Packet/loop parse bits
d5	Field to encode register d
e5	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
x5	Field to encode register x

Load-acquire word

Load a 32-bit word from memory and place in a destination register. The load-acquire memory operation is observed before any following memory operations (in program order) have been observed at the local point of serialization. A different order may be observed at the global point of serialization (see Memory ordering).

Syntax	Behavior
`Rd=memw_aq(Rs)`	`EA=Rs;` `Rd = *EA`

Class: LD (slots 0)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	s5					Parse											d5					
1	0	0	1	0	0	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	1	-	-	-	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=memw_aq(Rs)

Field name	Description
CLASS	Instruction class
Amode	Amode
Type	Type
UN	Unsigned
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s

Load word conditionally

Load a 32-bit word from memory and place in a destination register.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior

`if ([!]Pt[.new]) Rd=memw(#u6)`	`apply_extension(#u);`
	`EA=#u;`
	`if ([!]Pt[.new][0]) {`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pt[.new]) Rd=memw(Rs+#u6:2)`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`if ([!]Pt[.new][0]) {`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pt[.new]) Rd=memw(Rx++#s4:2)`	`EA=Rx;`
	`if([!]Pt[.new][0]){`
	`  Rx=Rx+#s;`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new]) Rd=memw(Rs+Rt<<#u2)`	`EA=Rs+(Rt<<#u);`
	`if ([!]Pv[.new][0]) {`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`}`

Class: LD (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			t5								d5					
0	0	1	1	0	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (Pv) Rd=memw(Rs+Rt<<#u2)
0	0	1	1	0	0	0	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (!Pv) Rd=memw(Rs+Rt<<#u2)
0	0	1	1	0	0	1	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (Pv.new) Rd=memw(Rs+Rt<<#u2)
0	0	1	1	0	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (!Pv.new) Rd=memw(Rs+Rt<<#u2)
ICLASS					Sense	Pred New		Type		UN	s5					Parse			t2								d5					

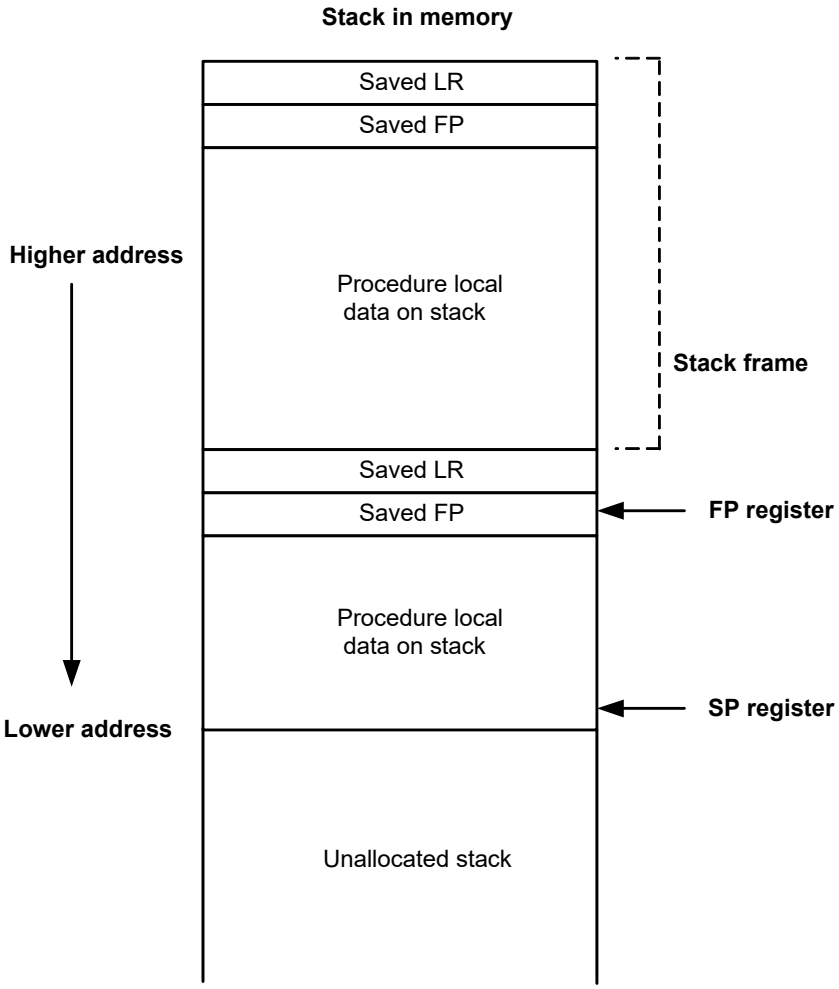
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	0	0	0	0	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memw(Rs+#u6:2)
0	1	0	0	0	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memw(Rs+#u6:2)
0	1	0	0	0	1	0	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memw(Rs+#u6:2)
0	1	0	0	0	1	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memw(Rs+#u6:2)
ICLASS				Amode			Type			UN	x5					Parse					t2							d5					
1	0	0	1	1	0	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	0	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memw(Rx++#s4:2)
1	0	0	1	1	0	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	1	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memw(Rx++#s4:2)
1	0	0	1	1	0	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	0	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memw(Rx++#s4:2)
1	0	0	1	1	0	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	1	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memw(Rx++#s4:2)
ICLASS				Amode			Type			UN						Parse					t2							d5					
1	0	0	1	1	1	1	1	1	0	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	0	0	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memw(#u6)
1	0	0	1	1	1	1	1	1	0	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	0	1	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memw(#u6)
1	0	0	1	1	1	1	1	1	0	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	1	0	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memw(#u6)
1	0	0	1	1	1	1	1	1	0	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	1	1	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memw(#u6)

Field name	Description
ICLASS	Instruction class
Amode	Amode
Type	Type
UN	Unsigned
PredNew	PredNew
Sense	Sense
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t2	Field to encode register t
t5	Field to encode register t
v2	Field to encode register v
x5	Field to encode register x

Deallocate stack frame

Deallocate a stack frame from the call stack. The instruction first loads the saved FP and saved LR values from the address at FP. It then points SP back to the previous frame.

The stack layout is seen in the following figure.



Syntax	Behavior

`Rdd=deallocframe(Rs):raw`	`EA=Rs;`
	`tmp = *EA;`
	`Rdd = frame_unscramble(tmp);`
	`SP=EA+8;`

`deallocframe`	`Assembler mapped to:` `"r31:30=deallocframe(r30):raw"`

Class: LD (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	s5					Parse											d5					
1	0	0	1	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Rdd=deallocframe(Rs):raw

Field name	Description
ICLASS	Instruction class
Amode	Amode
Type	Type
UN	Unsigned
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s

Deallocate frame and return

Return from a function with a stack frame. This instruction is equivalent to deallocframe followed by jumpr R31.

Syntax	Behavior

`Rdd=dealloc_return(Rs):raw`	`EA=Rs;`
	`tmp = *EA;`
	`Rdd = frame_unscramble(tmp);`
	`SP=EA+8;`
	`PC=Rdd.w[1];`

`dealloc_return`	`Assembler mapped to: "r31:30 = dealloc_return(r30):raw";`

`if ([!]Pv) Rdd = dealloc_return(Rs):raw`	`EA=Rs;`
	`if ([!]Pv[0]) {`
	`  tmp = *EA;`
	`  Rdd = frame_unscramble(tmp);`
	`  SP=EA+8;`
	`  PC=Rdd.w[1];`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv) dealloc_return`	`Assembler mapped to: "if ([!]Pv"") r31:30 = dealloc_return(r30)"":raw";`

`if ([!]Pv.new) Rdd = dealloc_return(Rs):nt:raw`	`EA=Rs;`
	`if ([!]Pv.new[0]) {`
	`  tmp = *EA;`
	`  Rdd = frame_unscramble(tmp);`
	`  SP =EA+8;`
	`  PC =Rdd.w[1];`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv.new) Rdd = dealloc_return(Rs):t:raw`	`EA=Rs;`
	`if ([!]Pv.new[0]) {`
	`  tmp = *EA;`
	`  Rdd = frame_unscramble(tmp);`
	`  SP=EA+8;`
	`  PC=Rdd.w[1];`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv.new) dealloc_return:nt`	`Assembler mapped to: "if ([!]Pv"".new"") r31:30=dealloc_return(r30)"":nt"":raw";`
`if ([!]Pv.new) dealloc_return:t`	`Assembler mapped to: "if ([!]Pv"".new"") r31:30=dealloc_return(r30)"":t"":raw";`

Class: LD (slots 0)

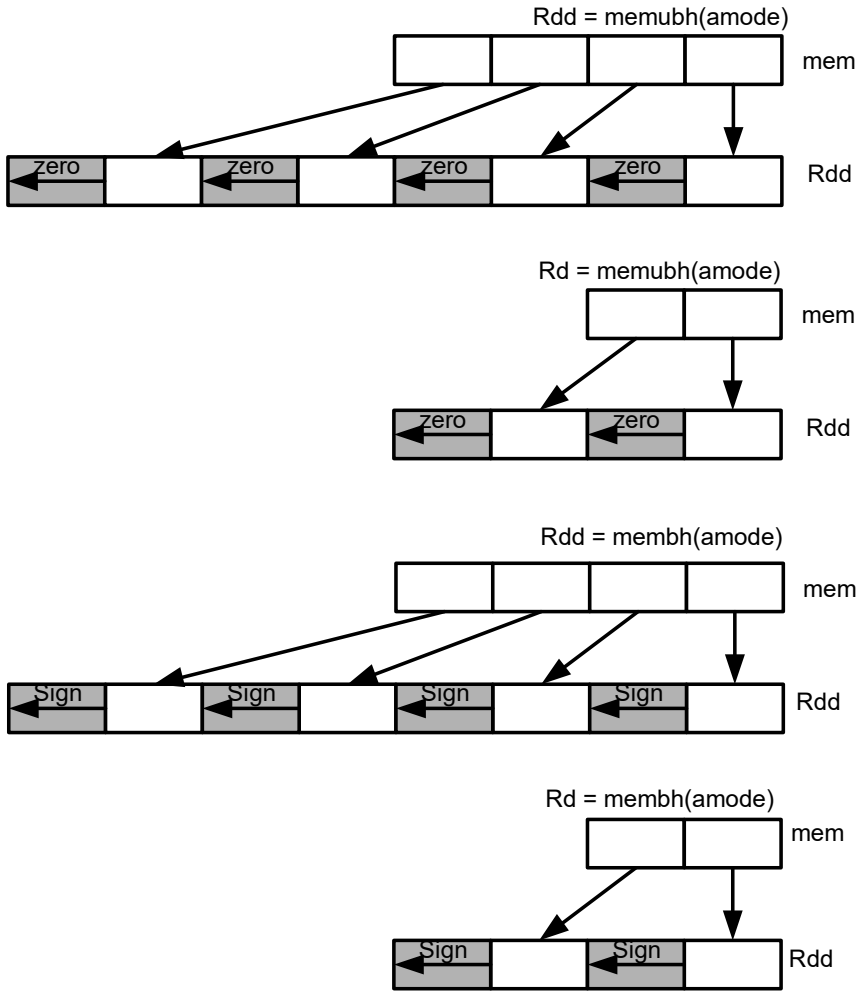
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	s5					Parse												d5					
1	0	0	1	0	1	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	0	0	**-**	**-**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rdd=dealloc_return(Rs):raw
1	0	0	1	0	1	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	1	0	**v**	**v**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (Pv.new) Rdd=dealloc_return(Rs):nt:raw
1	0	0	1	0	1	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	1	0	0	**v**	**v**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (Pv) Rdd=dealloc_return(Rs):raw
1	0	0	1	0	1	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	1	1	0	**v**	**v**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (Pv.new) Rdd=dealloc_return(Rs):t:raw
1	0	0	1	0	1	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	0	1	0	**v**	**v**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pv.new) Rdd=dealloc_return(Rs):nt:raw
1	0	0	1	0	1	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	1	0	0	**v**	**v**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pv) Rdd=dealloc_return(Rs):raw
1	0	0	1	0	1	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	1	1	0	**v**	**v**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pv.new) Rdd=dealloc_return(Rs):t:raw

Field name	Description
ICLASS	Instruction class
Amode	Amode
Type	Type
UN	Unsigned
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
v2	Field to encode register v

Load and unpack bytes to halfwords

Load contiguous bytes from memory and vector unpack them into halfwords.



Syntax	Behavior

`Rd=membh(Re=#U6)`	`apply_extension(#U);`
	`EA=#U;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.b[i];`
	`  }`
	`};`
	`Re=#U;`

`Rd=membh(Rs)`	`Assembler mapped to: "Rd=membh""(Rs+#0)"`

Syntax	Behavior

`Rd=membh(Rs+#s11:1)`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.b[i];`
	`  }`
	`};`

`Rd=membh(Rt<<#u2+#U6)`	`apply_extension(#U);`
	`EA=#U+(Rt<<#u);`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.b[i];`
	`  }`
	`};`

`Rd=membh(Rx++#s4:1)`	`EA=Rx;`
	`Rx=Rx+#s;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.b[i];`
	`  }`
	`};`

`Rd=membh(Rx++#s4:1:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.b[i];`
	`  }`
	`};`

`Rd=membh(Rx++I:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<1,MuV);`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.b[i];`
	`  }`
	`};`

`Rd=membh(Rx++Mu)`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.b[i];`
	`  }`
	`};`

Syntax	Behavior

`Rd=membh(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.b[i];`
	`  }`
	`};`

`Rd=memubh(Re=#U6)`	`apply_extension(#U);`
	`EA=#U;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.ub[i];`
	`  }`
	`};`
	`Re=#U;`

`Rd=memubh(Rs+#s11:1)`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.ub[i];`
	`  }`
	`};`

`Rd=memubh(Rt<<#u2+#U6)`	`apply_extension(#U);`
	`EA=#U+(Rt<<#u);`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.ub[i];`
	`  }`
	`};`

`Rd=memubh(Rx++#s4:1)`	`EA=Rx;`
	`Rx=Rx+#s;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.ub[i];`
	`  }`
	`};`

`Rd = memubh(Rx ++ #s4:1:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.ub[i];`
	`  }`
	`};`

Syntax	Behavior

`Rd=memubh(Rx++I:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<1,MuV);`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.ub[i];`
	`  }`
	`};`

`Rd=memubh(Rx++Mu)`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.ub[i];`
	`  }`
	`};`

`Rd=memubh(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.ub[i];`
	`  }`
	`};`

`Rdd=membh(Re=#U6)`	`apply_extension(#U);`
	`EA=#U;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.b[i];`
	`  }`
	`};`
	`Re=#U;`

`Rdd=membh(Rs)`	`Assembler mapped to: "Rdd=membh""(Rs+#0)"`

`Rdd=membh(Rs+#s11:2)`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.b[i];`
	`  }`
	`};`

`Rdd=membh(Rt<<#u2+#U6)`	`apply_extension(#U);`
	`EA=#U+(Rt<<#u);`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.b[i];`
	`  }`
	`};`

Syntax	Behavior

`Rdd=membh(Rx++#s4:2)`	`EA=Rx;`
	`Rx=Rx+#s;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.b[i];`
	`  }`
	`};`

`Rdd = membh(Rx ++ #s4:2:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.b[i];`
	`  }`
	`};`

`Rdd=membh(Rx++I:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<2,MuV);`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.b[i];`
	`  }`
	`};`

`Rdd=membh(Rx++Mu)`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.b[i];`
	`  }`
	`};`

`Rdd=membh(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.b[i];`
	`  }`
	`};`

`Rdd=memubh(Re=#U6)`	`apply_extension(#U);`
	`EA=#U;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.ub[i];`
	`  }`
	`};`
	`Re=#U;`

Syntax	Behavior

`Rdd=memubh(Rs+#s11:2)`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.ub[i];`
	`  }`
	`};`

`Rdd=memubh(Rt<<#u2+#U6)`	`apply_extension(#U);`
	`EA=#U+(Rt<<#u);`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.ub[i];`
	`  }`
	`};`

`Rdd=memubh(Rx++#s4:2)`	`EA=Rx;`
	`Rx=Rx+#s;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.ub[i];`
	`  }`
	`};`

`Rdd = memubh(Rx ++ #s4:2:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.ub[i];`
	`  }`
	`};`

`Rdd=memubh(Rx++I:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<2,MuV);`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.ub[i];`
	`  }`
	`};`

`Rdd=memubh(Rx++Mu)`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.ub[i];`
	`  }`
	`};`

Syntax	Behavior

`Rdd=memubh(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.ub[i];`
	`  }`
	`};`

Class: LD (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	s5					Parse												d5					
1	0	0	1	0	**i**	**i**	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=membh(Rs+#s11:1)
1	0	0	1	0	**i**	**i**	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memubh(Rs+#s11:1)
1	0	0	1	0	**i**	**i**	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rdd=memubh(Rs+#s11:2)
1	0	0	1	0	**i**	**i**	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rdd=membh(Rs+#s11:2)
ICLASS				Amode			Type			UN	x5					Parse		u1										d5					
1	0	0	1	1	0	0	0	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	0	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=membh(Rx++#s4:1:circ(Mu))
1	0	0	1	1	0	0	0	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	1	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=membh(Rx++I:circ(Mu))
1	0	0	1	1	0	0	0	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	0	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memubh(Rx++#s4:1:circ(Mu))
1	0	0	1	1	0	0	0	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	1	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memubh(Rx++I:circ(Mu))
1	0	0	1	1	0	0	0	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	0	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rdd=memubh(Rx++#s4:2:circ(Mu))
1	0	0	1	1	0	0	0	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	1	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rdd=memubh(Rx++I:circ(Mu))
1	0	0	1	1	0	0	0	1	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	0	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rdd=membh(Rx++#s4:2:circ(Mu))
1	0	0	1	1	0	0	0	1	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	1	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rdd=membh(Rx++I:circ(Mu))
ICLASS				Amode			Type			UN	e5					Parse												d5					
1	0	0	1	1	0	1	0	0	0	1	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	1	**I**	**I**	**I**	**I**	**-**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rd=membh(Re=#U6)
ICLASS				Amode			Type			UN	x5					Parse												d5					
1	0	0	1	1	0	1	0	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=membh(Rx++#s4:1)
ICLASS				Amode			Type			UN	e5					Parse												d5					
1	0	0	1	1	0	1	0	0	1	1	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	1	**I**	**I**	**I**	**I**	**-**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rd=memubh(Re=#U6)
ICLASS				Amode			Type			UN	x5					Parse												d5					
1	0	0	1	1	0	1	0	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memubh(Rx++#s4:1)
ICLASS				Amode			Type			UN	e5					Parse												d5					
1	0	0	1	1	0	1	0	1	0	1	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	1	**I**	**I**	**I**	**I**	**-**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rd=memubh(Re=#U6)
ICLASS				Amode			Type			UN	x5					Parse												d5					
1	0	0	1	1	0	1	0	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memubh(Rx++#s4:2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	e5					Parse												d5					
1	0	0	1	1	0	1	0	1	1	1	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	1	**l**	**l**	**l**	**l**	**-**	**l**	**l**	**d**	**d**	**d**	**d**	**d**	Rdd=membh(Re=#U6)
ICLASS				Amode			Type			UN	x5					Parse												d5					
1	0	0	1	1	0	1	0	1	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rdd=membh(Rx++#s4:2)
ICLASS				Amode			Type			UN	t5					Parse												d5					
1	0	0	1	1	1	0	0	0	0	1	**t**	**t**	**t**	**t**	**t**	**P**	**P**	**i**	1	**l**	**l**	**l**	**l**	**i**	**l**	**l**	**d**	**d**	**d**	**d**	**d**	Rd=membh(Rt<<#u2+#U6)
ICLASS				Amode			Type			UN	x5					Parse		u1										d5					
1	0	0	1	1	1	0	0	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=membh(Rx++Mu)
ICLASS				Amode			Type			UN	t5					Parse												d5					
1	0	0	1	1	1	0	0	0	1	1	**t**	**t**	**t**	**t**	**t**	**P**	**P**	**i**	1	**l**	**l**	**l**	**l**	**i**	**l**	**l**	**d**	**d**	**d**	**d**	**d**	Rd=memubh(Rt<<#u2+#U6)
ICLASS				Amode			Type			UN	x5					Parse		u1										d5					
1	0	0	1	1	1	0	0	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memubh(Rx++Mu)
ICLASS				Amode			Type			UN	t5					Parse												d5					
1	0	0	1	1	1	0	0	1	0	1	**t**	**t**	**t**	**t**	**t**	**P**	**P**	**i**	1	**l**	**l**	**l**	**l**	**i**	**l**	**l**	**d**	**d**	**d**	**d**	**d**	Rdd=memubh(Rt<<#u2+#U6)
ICLASS				Amode			Type			UN	x5					Parse		u1										d5					
1	0	0	1	1	1	0	0	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rdd=memubh(Rx++Mu)
ICLASS				Amode			Type			UN	t5					Parse												d5					
1	0	0	1	1	1	0	0	1	1	1	**t**	**t**	**t**	**t**	**t**	**P**	**P**	**i**	1	**l**	**l**	**l**	**l**	**i**	**l**	**l**	**d**	**d**	**d**	**d**	**d**	Rd=membh(Rt<<#u2+#U6)
ICLASS				Amode			Type			UN	x5					Parse		u1										d5					
1	0	0	1	1	1	0	0	1	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=membh(Rx++Mu)
1	0	0	1	1	1	1	0	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=membh(Rx++Mu:brev)
1	0	0	1	1	1	1	0	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memubh(Rx++Mu:brev)
1	0	0	1	1	1	1	0	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rdd=memubh(Rx++Mu:brev)
1	0	0	1	1	1	1	0	1	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rdd=membh(Rx++Mu:brev)

Field name	Description
ICLASS	Instruction class
Amode	Amode
Type	Type
UN	Unsigned
Parse	Packet/loop parse bits
d5	Field to encode register d
e5	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
x5	Field to encode register x

11.6 MEMOP

The MEMOP instruction class includes simple operations on values in memory. MEMOP instructions are executable on slot 0.

Operation on memory byte

Perform ALU or bit operation on the memory byte at the effective address.

Syntax	Behavior

`memb(Rs+#u6:0)=clrbit(#U5)`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`tmp = *EA;`
	`tmp &= (~(1<<#U));`
	`*EA = tmp;`

`memb(Rs+#u6:0)=setbit(#U5)`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`tmp = *EA;`
	`tmp |= (1<<#U);`
	`*EA = tmp;`

`memb(Rs+#u6:0)[+-]=#U5`	`apply_extension(#u);`
	`EA=Rs[+-]#u;`
	`tmp = *EA;`
	`tmp [+-]= #U;`
	`*EA = tmp;`

`memb(Rs+#u6:0)[+-|&]=Rt`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`tmp = *EA;`
	`tmp [+-|&]= Rt;`
	`*EA = tmp;`

Class: MEMOP (slots 0)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse											t5					
0	0	1	1	1	1	1	0	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	0	0	**t**	**t**	**t**	**t**	**t**	memb(Rs+#u6:0)+=Rt
0	0	1	1	1	1	1	0	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	0	1	**t**	**t**	**t**	**t**	**t**	memb(Rs+#u6:0)-=Rt
0	0	1	1	1	1	1	0	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	1	0	**t**	**t**	**t**	**t**	**t**	memb(Rs+#u6:0)&=Rt
0	0	1	1	1	1	1	0	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	1	1	**t**	**t**	**t**	**t**	**t**	memb(Rs+#u6:0)|=Rt
ICLASS											s5					Parse																
0	0	1	1	1	1	1	1	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	0	0	**I**	**I**	**I**	**I**	**I**	memb(Rs+#u6:0)+=#U5
0	0	1	1	1	1	1	1	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	0	1	**I**	**I**	**I**	**I**	**I**	memb(Rs+#u6:0)-=#U5
0	0	1	1	1	1	1	1	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	1	0	**I**	**I**	**I**	**I**	**I**	memb(Rs+#u6:0)=clrbit(#U5)
0	0	1	1	1	1	1	1	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	1	1	**I**	**I**	**I**	**I**	**I**	memb(Rs+#u6:0)=setbit(#U5)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
s5	Field to encode register s
t5	Field to encode register t

Operation on memory halfword

Perform ALU or bit operation on the memory halfword at the effective address.

Syntax	Behavior

`memh(Rs+#u6:1)=clrbit(#U5)`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`tmp = *EA;`
	`tmp &= (~(1<<#U));`
	`*EA = tmp;`

`memh(Rs+#u6:1)=setbit(#U5)`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`tmp = *EA;`
	`tmp |= (1<<#U);`
	`*EA = tmp;`

`memh(Rs+#u6:1)[+-]=#U5`	`apply_extension(#u);`
	`EA=Rs[+-]#u;`
	`tmp = *EA;`
	`tmp [+-]= #U;`
	`*EA = tmp;`

`memh(Rs+#u6:1)[+-|&]=Rt`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`tmp = *EA;`
	`tmp [+-|&]= Rt;`
	`*EA = tmp;`

Class: MEMOP (slots 0)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse											t5					
0	0	1	1	1	1	1	0	-	0	1	s	s	s	s	s	P	P	0	i	i	i	i	i	i	0	0	t	t	t	t	t	memh(Rs+#u6:1)+=Rt
0	0	1	1	1	1	1	0	-	0	1	s	s	s	s	s	P	P	0	i	i	i	i	i	i	0	1	t	t	t	t	t	memh(Rs+#u6:1)-=Rt
0	0	1	1	1	1	1	0	-	0	1	s	s	s	s	s	P	P	0	i	i	i	i	i	i	1	0	t	t	t	t	t	memh(Rs+#u6:1)&=Rt
0	0	1	1	1	1	1	0	-	0	1	s	s	s	s	s	P	P	0	i	i	i	i	i	i	1	1	t	t	t	t	t	memh(Rs+#u6:1)|=Rt
ICLASS											s5					Parse																
0	0	1	1	1	1	1	1	-	0	1	s	s	s	s	s	P	P	0	i	i	i	i	i	i	0	0	I	I	I	I	I	memh(Rs+#u6:1)+=#U5
0	0	1	1	1	1	1	1	-	0	1	s	s	s	s	s	P	P	0	i	i	i	i	i	i	0	1	I	I	I	I	I	memh(Rs+#u6:1)-=#U5
0	0	1	1	1	1	1	1	-	0	1	s	s	s	s	s	P	P	0	i	i	i	i	i	i	1	0	I	I	I	I	I	memh(Rs+#u6:1)=clrbit(#U5)
0	0	1	1	1	1	1	1	-	0	1	s	s	s	s	s	P	P	0	i	i	i	i	i	i	1	1	I	I	I	I	I	memh(Rs+#u6:1)=setbit(#U5)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
s5	Field to encode register s
t5	Field to encode register t

Operation on memory word

Perform ALU or bit operation on the memory word at the effective address.

Syntax	Behavior

`memw(Rs+#u6:2)=clrbit(#U5)`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`tmp = *EA;`
	`tmp &= (~(1<<#U));`
	`*EA = tmp;`

`memw(Rs+#u6:2)=setbit(#U5)`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`tmp = *EA;`
	`tmp |= (1<<#U);`
	`*EA = tmp;`

`memw(Rs+#u6:2)[+-]=#U5`	`apply_extension(#u);`
	`EA=Rs[+-]#u;`
	`tmp = *EA;`
	`tmp [+-]= #U;`
	`*EA = tmp;`

`memw(Rs+#u6:2)[+-|&]=Rt`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`tmp = *EA;`
	`tmp [+-|&]= Rt;`
	`*EA = tmp;`

Class: MEMOP (slots 0)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse											t5					
0	0	1	1	1	1	1	0	-	1	0	s	s	s	s	s	P	P	0	i	i	i	i	i	i	0	0	t	t	t	t	t	memw(Rs+#u6:2)+=Rt
0	0	1	1	1	1	1	0	-	1	0	s	s	s	s	s	P	P	0	i	i	i	i	i	i	0	1	t	t	t	t	t	memw(Rs+#u6:2)-=Rt
0	0	1	1	1	1	1	0	-	1	0	s	s	s	s	s	P	P	0	i	i	i	i	i	i	1	0	t	t	t	t	t	memw(Rs+#u6:2)&=Rt
0	0	1	1	1	1	1	0	-	1	0	s	s	s	s	s	P	P	0	i	i	i	i	i	i	1	1	t	t	t	t	t	memw(Rs+#u6:2)|=Rt
ICLASS											s5					Parse																
0	0	1	1	1	1	1	1	-	1	0	s	s	s	s	s	P	P	0	i	i	i	i	i	i	0	0	I	I	I	I	I	memw(Rs+#u6:2)+=#U5
0	0	1	1	1	1	1	1	-	1	0	s	s	s	s	s	P	P	0	i	i	i	i	i	i	0	1	I	I	I	I	I	memw(Rs+#u6:2)-=#U5
0	0	1	1	1	1	1	1	-	1	0	s	s	s	s	s	P	P	0	i	i	i	i	i	i	1	0	I	I	I	I	I	memw(Rs+#u6:2)=clrbit(#U5)
0	0	1	1	1	1	1	1	-	1	0	s	s	s	s	s	P	P	0	i	i	i	i	i	i	1	1	I	I	I	I	I	memw(Rs+#u6:2)=setbit(#U5)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
s5	Field to encode register s
t5	Field to encode register t

11.7 NV

The NV instruction class includes instructions that take the register source operand from another instruction in the same packet.

NV instructions are executable on slot 0.

11.7.1 NV J

The NV J instruction subclass includes jump instructions that take the register source operand from another instruction in the same packet.

Jump to address condition on new register value

Compare a register or constant against the value produced by a slot 1 instruction. If the comparison is true, the program counter is changed to a target address, relative to the current PC.

This instruction is executable only on slot 0.

Syntax	Behavior
`if ([!]cmp.eq(Ns.new,#-1)) jump:<hint> #r9:2`	`if ((Ns.new[!]=(-1))) { apply_extension(#r); #r=#r & ~PCALIGN_MASK; PC=PC+#r; }`
`if ([!]cmp.eq(Ns.new,#U5)) jump:<hint> #r9:2`	`if ((Ns.new[!]=(#U))) { apply_extension(#r); #r=#r & ~PCALIGN_MASK; PC=PC+#r; }`
`if ([!]cmp.eq(Ns.new,Rt)) jump:<hint> #r9:2`	`if ((Ns.new[!]=Rt)) { apply_extension(#r); #r=#r & ~PCALIGN_MASK; PC=PC+#r; }`
`if ([!]cmp.gt(Ns.new,#-1)) jump:<hint> #r9:2`	`if ([!](Ns.new>(-1))) { apply_extension(#r); #r=#r & ~PCALIGN_MASK; PC=PC+#r; }`
`if ([!]cmp.gt(Ns.new,#U5)) jump:<hint> #r9:2`	`if ([!](Ns.new>(#U))) { apply_extension(#r); #r=#r & ~PCALIGN_MASK; PC=PC+#r; }`
`if ([!]cmp.gt(Ns.new,Rt)) jump:<hint> #r9:2`	`if ([!](Ns.new>Rt)) { apply_extension(#r); #r=#r & ~PCALIGN_MASK; PC=PC+#r; }`

Syntax	Behavior

`if ([!]cmp.gt(Rt,Ns.new)) jump:<hint> #r9:2`	`if ([!](Rt>Ns.new)) {`
	`  apply_extension(#r);`
	`  #r=#r & ~PCALIGN_MASK;`
	`  PC=PC+#r;`
	`}`

`if ([!]cmp.gtu(Ns.new,#U5)) jump:<hint> #r9:2`	`if ([!](Ns.new.uw[0]>(#U))) {`
	`  apply_extension(#r);`
	`  #r=#r & ~PCALIGN_MASK;`
	`  PC=PC+#r;`
	`}`

`if ([!]cmp.gtu(Ns.new,Rt)) jump:<hint> #r9:2`	`if ([!](Ns.new.uw[0]>Rt.uw[0])) {`
	`  apply_extension(#r);`
	`  #r=#r & ~PCALIGN_MASK;`
	`  PC=PC+#r;`
	`}`

`if ([!]cmp.gtu(Rt,Ns.new)) jump:<hint> #r9:2`	`if ([!](Rt.uw[0]>Ns.new.uw[0])) {`
	`  apply_extension(#r);`
	`  #r=#r & ~PCALIGN_MASK;`
	`  PC=PC+#r;`
	`}`

`if ([!]tstbit(Ns.new,#0)) jump:<hint> #r9:2`	`if ([!]((Ns.new) & 1)) {`
	`  apply_extension(#r);`
	`  #r=#r & ~PCALIGN_MASK;`
	`  PC=PC+#r;`
	`}`

Class: NV (slots 0)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS													s3			Parse			t5														
0	0	1	0	0	0	0	0	0	0	i	i	-	s	s	s	P	P	0	t	t	t	t	t	i	i	i	i	i	i	i	-	if (cmp.eq(Ns.new,Rt)) jump:nt #r9:2
0	0	1	0	0	0	0	0	0	0	i	i	-	s	s	s	P	P	1	t	t	t	t	t	i	i	i	i	i	i	i	-	if (cmp.eq(Ns.new,Rt)) jump:t #r9:2
0	0	1	0	0	0	0	0	0	1	i	i	-	s	s	s	P	P	0	t	t	t	t	t	i	i	i	i	i	i	i	-	if (!cmp.eq(Ns.new,Rt)) jump:nt #r9:2
0	0	1	0	0	0	0	0	0	1	i	i	-	s	s	s	P	P	1	t	t	t	t	t	i	i	i	i	i	i	i	-	if (!cmp.eq(Ns.new,Rt)) jump:t #r9:2
0	0	1	0	0	0	0	0	1	0	i	i	-	s	s	s	P	P	0	t	t	t	t	t	i	i	i	i	i	i	i	-	if (cmp.gt(Ns.new,Rt)) jump:nt #r9:2
0	0	1	0	0	0	0	0	1	0	i	i	-	s	s	s	P	P	1	t	t	t	t	t	i	i	i	i	i	i	i	-	if (cmp.gt(Ns.new,Rt)) jump:t #r9:2
0	0	1	0	0	0	0	0	1	1	i	i	-	s	s	s	P	P	0	t	t	t	t	t	i	i	i	i	i	i	i	-	if (!cmp.gt(Ns.new,Rt)) jump:nt #r9:2
0	0	1	0	0	0	0	0	1	1	i	i	-	s	s	s	P	P	1	t	t	t	t	t	i	i	i	i	i	i	i	-	if (!cmp.gt(Ns.new,Rt)) jump:t #r9:2
0	0	1	0	0	0	0	1	0	0	i	i	-	s	s	s	P	P	0	t	t	t	t	t	i	i	i	i	i	i	i	-	if (cmp.gtu(Ns.new,Rt)) jump:nt #r9:2
0	0	1	0	0	0	0	1	0	0	i	i	-	s	s	s	P	P	1	t	t	t	t	t	i	i	i	i	i	i	i	-	if (cmp.gtu(Ns.new,Rt)) jump:t #r9:2
0	0	1	0	0	0	0	1	0	1	i	i	-	s	s	s	P	P	0	t	t	t	t	t	i	i	i	i	i	i	i	-	if (!cmp.gtu(Ns.new,Rt)) jump:nt #r9:2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	0	0	0	0	1	0	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.gtu(Ns.new,Rt)) jump:t #r9:2
0	0	1	0	0	0	0	1	1	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.gt(Rt,Ns.new)) jump:nt #r9:2
0	0	1	0	0	0	0	1	1	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.gt(Rt,Ns.new)) jump:t #r9:2
0	0	1	0	0	0	0	1	1	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.gt(Rt,Ns.new)) jump:nt #r9:2
0	0	1	0	0	0	0	1	1	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.gt(Rt,Ns.new)) jump:t #r9:2
0	0	1	0	0	0	1	0	0	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.gtu(Rt,Ns.new)) jump:nt #r9:2
0	0	1	0	0	0	1	0	0	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.gtu(Rt,Ns.new)) jump:t #r9:2
0	0	1	0	0	0	1	0	0	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.gtu(Rt,Ns.new)) jump:nt #r9:2
0	0	1	0	0	0	1	0	0	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.gtu(Rt,Ns.new)) jump:t #r9:2
ICLASS													s3			Parse																
0	0	1	0	0	1	0	0	0	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.eq(Ns.new,#U5)) jump:nt #r9:2
0	0	1	0	0	1	0	0	0	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.eq(Ns.new,#U5)) jump:t #r9:2
0	0	1	0	0	1	0	0	0	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.eq(Ns.new,#U5)) jump:nt #r9:2
0	0	1	0	0	1	0	0	0	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.eq(Ns.new,#U5)) jump:t #r9:2
0	0	1	0	0	1	0	0	1	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.gt(Ns.new,#U5)) jump:nt #r9:2
0	0	1	0	0	1	0	0	1	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.gt(Ns.new,#U5)) jump:t #r9:2
0	0	1	0	0	1	0	0	1	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.gt(Ns.new,#U5)) jump:nt #r9:2
0	0	1	0	0	1	0	0	1	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.gt(Ns.new,#U5)) jump:t #r9:2
0	0	1	0	0	1	0	1	0	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.gtu(Ns.new,#U5)) jump:nt #r9:2
0	0	1	0	0	1	0	1	0	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.gtu(Ns.new,#U5)) jump:t #r9:2
0	0	1	0	0	1	0	1	0	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.gtu(Ns.new,#U5)) jump:nt #r9:2
0	0	1	0	0	1	0	1	0	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.gtu(Ns.new,#U5)) jump:t #r9:2
0	0	1	0	0	1	0	1	1	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**-**	**-**	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (tstbit(Ns.new,#0)) jump:nt #r9:2
0	0	1	0	0	1	0	1	1	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**-**	**-**	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (tstbit(Ns.new,#0)) jump:t #r9:2
0	0	1	0	0	1	0	1	1	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**-**	**-**	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!tstbit(Ns.new,#0)) jump:nt #r9:2
0	0	1	0	0	1	0	1	1	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**-**	**-**	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!tstbit(Ns.new,#0)) jump:t #r9:2
0	0	1	0	0	1	1	0	0	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**-**	**-**	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.eq(Ns.new,#-1)) jump:nt #r9:2
0	0	1	0	0	1	1	0	0	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**-**	**-**	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.eq(Ns.new,#-1)) jump:t #r9:2
0	0	1	0	0	1	1	0	0	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**-**	**-**	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.eq(Ns.new,#-1)) jump:nt #r9:2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	0	0	1	1	0	0	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**-**	**-**	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.eq(Ns.new,#-1)) jump:t #r9:2
0	0	1	0	0	1	1	0	1	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**-**	**-**	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.gt(Ns.new,#-1)) jump:nt #r9:2
0	0	1	0	0	1	1	0	1	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**-**	**-**	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.gt(Ns.new,#-1)) jump:t #r9:2
0	0	1	0	0	1	1	0	1	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**-**	**-**	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.gt(Ns.new,#-1)) jump:nt #r9:2
0	0	1	0	0	1	1	0	1	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**-**	**-**	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.gt(Ns.new,#-1)) jump:t #r9:2

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
s3	Field to encode register s
t5	Field to encode register t

11.7.2 NV ST

The NV ST instruction subclass includes store instructions that take the register source operand from another instruction in the same packet.

Store new-value byte

Store the least-significant byte in a source register in memory at the effective address.

Syntax	Behavior

`memb(Re=#U6)=Nt.new`	`apply_extension(#U);`
	`EA=#U;`
	`*EA = Nt.new.b[0];`
	`Re=#U;`

`memb(Rs+#s11:0)=Nt.new`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`*EA = Nt.new.b[0];`

`memb(Rs+Ru<<#u2)=Nt.new`	`EA=Rs+(Ru<<#u);` `*EA = Nt.new.b[0];`

`memb(Ru<<#u2+#U6)=Nt.new`	`apply_extension(#U);`
	`EA=#U+(Ru<<#u);`
	`*EA = Nt.new.b[0];`

`memb(Rx++#s4:0)=Nt.new`	`EA=Rx;`
	`Rx=Rx+#s;`
	`*EA = Nt.new.b[0];`

`memb(Rx++#s4:0:circ(Mu))=Nt.new`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`*EA = Nt.new.b[0];`

`memb(Rx++I:circ(Mu))=Nt.new`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<0,MuV);`
	`*EA = Nt.new.b[0];`

`memb(Rx++Mu)=Nt.new`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`*EA = Nt.new.b[0];`

`memb(Rx++Mu:brev)=Nt.new`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`*EA = Nt.new.b[0];`

`memb(gp+#u16:0)=Nt.new`	`apply_extension(#u);`
	`EA=(Constant_extended ? (0) : GP)+#u;`
	`*EA = Nt.new.b[0];`

Class: NV (slots 0)

Notes

- Forms of this instruction that use a new-value operand produced in the packet must execute on slot 0.
- This instruction can execute only in slot 0, even though it is an ST instruction.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5									t3			
0	0	1	1	1	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**-**	**-**	0	0	**t**	**t**	**t**	memb(Rs+Ru<<#u2)=Nt.new
ICLASS								Type								Parse					t3											
0	1	0	0	1	**i**	**i**	0	1	0	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	0	0	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memb(gp+#u16:0)=Nt.new
ICLASS				Amode			Type			UN	s5					Parse					t3											
1	0	1	0	0	**i**	**i**	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	0	0	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memb(Rs+#s11:0)=Nt.new
ICLASS				Amode			Type			UN	x5					Parse		u1			t3											
1	0	1	0	1	0	0	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	0	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	1	**-**	memb(Rx++I:circ(Mu))=Nt.new
1	0	1	0	1	0	0	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	0	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	0	**-**	memb(Rx++#s4:0:circ(Mu))=Nt.new
ICLASS				Amode			Type			UN	e5					Parse					t3											
1	0	1	0	1	0	1	1	1	0	1	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	0	0	**t**	**t**	**t**	1	**-**	**I**	**I**	**I**	**I**	**I**	**I**	memb(Re=#U6)=Nt.new
ICLASS				Amode			Type			UN	x5					Parse					t3											
1	0	1	0	1	0	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	0	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	0	**-**	memb(Rx++#s4:0)=Nt.new
ICLASS				Amode			Type			UN	u5					Parse					t3											
1	0	1	0	1	1	0	1	1	0	1	**u**	**u**	**u**	**u**	**u**	**P**	**P**	**i**	0	0	**t**	**t**	**t**	1	**i**	**I**	**I**	**I**	**I**	**I**	**I**	memb(Ru<<#u2+#U6)=Nt.new
ICLASS				Amode			Type			UN	x5					Parse		u1			t3											
1	0	1	0	1	1	0	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	0	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memb(Rx++Mu)=Nt.new
1	0	1	0	1	1	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	0	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memb(Rx++Mu:brev)=Nt.new

Field name	Description
ICLASS	Instruction class
Type	Type
Parse	Packet/loop parse bits
e5	Field to encode register e
s5	Field to encode register s
t3	Field to encode register t
u1	Field to encode register u
u5	Field to encode register u
x5	Field to encode register x
Amode	Amode
UN	Unsigned

Store new-value byte conditionally

Store the least-significant byte in a source register in memory at the effective address.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior

`if ([!]Pv[.new]) memb(#u6) = Nt.new`	`apply_extension(#u);`
	`EA=#u;`
	`if ([!]Pv[.new][0]) {`
	`  *EA = Nt[.new].b[0];`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new]) memb(Rs + #u6:0) = Nt.new`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`if ([!]Pv[.new][0]) {`
	`  *EA = Nt[.new].b[0];`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new]) memb(Rs + Ru<<#u2) = Nt.new`	`EA=Rs+(Ru<<#u);`
	`if ([!]Pv[.new][0]) {`
	`  *EA = Nt[.new].b[0];`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new]) memb(Rx ++ #s4:0) =Nt.new`	`EA=Rx;`
	`if ([!]Pv[.new][0]){`
	`  Rx=Rx+#s;`
	`  *EA = Nt[.new].b[0];`
	`} else {`
	`  NOP;`
	`}`

Class: NV (slots 0)

Notes

- Forms of this instruction that use a new-value operand produced in the packet must execute on slot 0.
- This instruction can execute only in slot 0, even though it is an ST instruction.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5									t3			
0	0	1	1	0	1	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	0	0	**t**	**t**	**t**	if (Pv) memb(Rs+Ru<<#u2)=Nt.new
0	0	1	1	0	1	0	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	0	0	**t**	**t**	**t**	if (!Pv) memb(Rs+Ru<<#u2)=Nt.new
0	0	1	1	0	1	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	0	0	**t**	**t**	**t**	if (Pv.new) memb(Rs+Ru<<#u2)=Nt.new
0	0	1	1	0	1	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	0	0	**t**	**t**	**t**	if (!Pv.new) memb(Rs+Ru<<#u2)=Nt.new
ICLASS					Sense	PredNew		Type			s5					Parse					t3											
0	1	0	0	0	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	0	0	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memb(Rs+#u6:0)=Nt.new
0	1	0	0	0	0	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	0	0	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memb(Rs+#u6:0)=Nt.new
0	1	0	0	0	1	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	0	0	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv) memb(Rs+#u6:0)=Nt.new
0	1	0	0	0	1	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	0	0	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv.new) memb(Rs+#u6:0)=Nt.new
ICLASS				Amode			Type			UN	x5					Parse					t3											
1	0	1	0	1	0	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	0	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memb(Rx++#s4:0)=Nt.new
1	0	1	0	1	0	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	0	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memb(Rx++#s4:0)=Nt.new
1	0	1	0	1	0	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	0	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memb(Rx++#s4:0)=Nt.new
1	0	1	0	1	0	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	0	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memb(Rx++#s4:0)=Nt.new
ICLASS				Amode			Type			UN						Parse					t3											
1	0	1	0	1	1	1	1	1	0	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	0	0	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memb(#u6)=Nt.new
1	0	1	0	1	1	1	1	1	0	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	0	0	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memb(#u6)=Nt.new
1	0	1	0	1	1	1	1	1	0	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	0	0	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memb(#u6)=Nt.new
1	0	1	0	1	1	1	1	1	0	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	0	0	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memb(#u6)=Nt.new

Field name	Description
CLASS	Instruction class
Type	Type
PredNew	PredNew
Sense	Sense
Parse	Packet/loop parse bits
s5	Field to encode register s
t3	Field to encode register t
u5	Field to encode register u
v2	Field to encode register v
x5	Field to encode register x
Amode	Amode
UN	Unsigned

Store new-value halfword

Store the upper or lower 16-bits of a source register in memory at the effective address.

Syntax	Behavior

`memh(Re=#U6)=Nt.new`	`apply_extension(#U);`
	`EA=#U;`
	`*EA = Nt.new.h[0];`
	`Re=#U;`

`memh(Rs+#s11:1)=Nt.new`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`*EA = Nt.new.h[0];`

`memh(Rs+Ru<<#u2)=Nt.new`	`EA=Rs+(Ru<<#u);` `*EA = Nt.new.h[0];`

`memh(Ru<<#u2+#U6)=Nt.new`	`apply_extension(#U);`
	`EA=#U+(Ru<<#u);`
	`*EA = Nt.new.h[0];`

`memh(Rx++#s4:1)=Nt.new`	`EA=Rx;`
	`Rx=Rx+#s;`
	`*EA = Nt.new.h[0];`

`memh(Rx++#s4:1:circ(Mu)) = Nt.new`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`*EA = Nt.new.h[0];`

`memh(Rx++I:circ(Mu))=Nt.new`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<1,MuV);`
	`*EA = Nt.new.h[0];`

`memh(Rx++Mu)=Nt.new`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`*EA = Nt.new.h[0];`

`memh(Rx++Mu:brev)=Nt.new`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`*EA = Nt.new.h[0];`

`memh(gp+#u16:1)=Nt.new`	`apply_extension(#u);`
	`EA=(Constant_extended ? (0) : GP)+#u;`
	`*EA = Nt.new.h[0];`

Class: NV (slots 0)

Notes

- Forms of this instruction that use a new-value operand produced in the packet must execute on slot 0.
- This instruction can execute only in slot 0, even though it is an ST instruction.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5									t3			
0	0	1	1	1	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**-**	**-**	0	1	**t**	**t**	**t**	memh(Rs+Ru<<#u2)=Nt.new
ICLASS								Type								Parse					t3											
0	1	0	0	1	**i**	**i**	0	1	0	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	0	1	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memh(gp+#u16:1)=Nt.new
ICLASS				Amode			Type			UN	s5					Parse					t3											
1	0	1	0	0	**i**	**i**	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	0	1	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memh(Rs+#s11:1)=Nt.new
ICLASS				Amode			Type			UN	x5					Parse		u1			t3											
1	0	1	0	1	0	0	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	1	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	1	**-**	memh(Rx++I:circ(Mu))=Nt.new
1	0	1	0	1	0	0	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	1	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	0	**-**	memh(Rx++#s4:1:circ(Mu))=Nt.new
ICLASS				Amode			Type			UN	e5					Parse					t3											
1	0	1	0	1	0	1	1	1	0	1	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	0	1	**t**	**t**	**t**	1	**-**	**I**	**I**	**I**	**I**	**I**	**I**	memh(Re=#U6)=Nt.new
ICLASS				Amode			Type			UN	x5					Parse					t3											
1	0	1	0	1	0	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	1	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	0	**-**	memh(Rx++#s4:1)=Nt.new
ICLASS				Amode			Type			UN	u5					Parse					t3											
1	0	1	0	1	1	0	1	1	0	1	**u**	**u**	**u**	**u**	**u**	**P**	**P**	**i**	0	1	**t**	**t**	**t**	1	**i**	**I**	**I**	**I**	**I**	**I**	**I**	memh(Ru<<#u2+#U6)=Nt.new
ICLASS				Amode			Type			UN	x5					Parse		u1			t3											
1	0	1	0	1	1	0	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	1	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memh(Rx++Mu)=Nt.new
1	0	1	0	1	1	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	1	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memh(Rx++Mu:brev)=Nt.new

Field name	Description
ICLASS	Instruction class
Type	Type
Parse	Packet/loop parse bits
e5	Field to encode register e
s5	Field to encode register s
t3	Field to encode register t
u1	Field to encode register u
u5	Field to encode register u
x5	Field to encode register x
Amode	Amode
UN	Unsigned

Store new-value halfword conditionally

Store the upper or lower 16-bits of a source register in memory at the effective address.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior

`if ([!]Pv[.new]) memh(#u6)=Nt.new`	`apply_extension(#u);`
	`EA=#u;`
	`if ([!]Pv[.new][0]) {`
	`  *EA = Nt[.new].h[0];`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new]) memh(Rs+#u6:1)=Nt.new`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`if ([!]Pv[.new][0]) {`
	`  *EA = Nt[.new].h[0];`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new]) memh(Rs+Ru<<#u2)=Nt.new`	`EA=Rs+(Ru<<#u);`
	`if ([!]Pv[.new][0]) {`
	`  *EA = Nt[.new].h[0];`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new]) memh(Rx++#s4:1)=Nt.new`	`EA=Rx;`
	`if ([!]Pv[.new][0]){`
	`  Rx=Rx+#s;`
	`  *EA = Nt[.new].h[0];`
	`} else {`
	`  NOP;`
	`}`

Class: NV (slots 0)

Notes

- Forms of this instruction that use a new-value operand produced in the packet must execute on slot 0.
- This instruction can execute only in slot 0, even though it is an ST instruction.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5									t3			
0	0	1	1	0	1	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	0	1	**t**	**t**	**t**	if (Pv) memh(Rs+Ru<<#u2)=Nt.new
0	0	1	1	0	1	0	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	0	1	**t**	**t**	**t**	if (!Pv) memh(Rs+Ru<<#u2)=Nt.new
0	0	1	1	0	1	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	0	1	**t**	**t**	**t**	if (Pv.new) memh(Rs+Ru<<#u2)=Nt.new
0	0	1	1	0	1	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	0	1	**t**	**t**	**t**	if (!Pv.new) memh(Rs+Ru<<#u2)=Nt.new
ICLASS					Sense	PredNew		Type			s5					Parse					t3											
0	1	0	0	0	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	0	1	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memh(Rs+#u6:1)=Nt.new
0	1	0	0	0	0	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	0	1	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memh(Rs+#u6:1)=Nt.new
0	1	0	0	0	1	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	0	1	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv) memh(Rs+#u6:1)=Nt.new
0	1	0	0	0	1	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	0	1	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv.new) memh(Rs+#u6:1)=Nt.new
ICLASS				Amode			Type			UN	x5					Parse					t3											
1	0	1	0	1	0	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	1	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memh(Rx++#s4:1)=Nt.new
1	0	1	0	1	0	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	1	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memh(Rx++#s4:1)=Nt.new
1	0	1	0	1	0	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	1	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memh(Rx++#s4:1)=Nt.new
1	0	1	0	1	0	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	1	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memh(Rx++#s4:1)=Nt.new
ICLASS				Amode			Type			UN						Parse					t3											
1	0	1	0	1	1	1	1	1	0	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	0	1	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memh(#u6)=Nt.new
1	0	1	0	1	1	1	1	1	0	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	0	1	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memh(#u6)=Nt.new
1	0	1	0	1	1	1	1	1	0	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	0	1	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memh(#u6)=Nt.new
1	0	1	0	1	1	1	1	1	0	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	0	1	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memh(#u6)=Nt.new

Field name	Description
ICLASS	Instruction class
Type	Type
PredNew	PredNew
Sense	Sense
Parse	Packet/loop parse bits
s5	Field to encode register s
t3	Field to encode register t
u5	Field to encode register u
v2	Field to encode register v
x5	Field to encode register x
Amode	Amode
UN	Unsigned

Store new-value word

Store a 32-bit register in memory at the effective address.

Syntax	Behavior

`memw(Re=#U6)=Nt.new`	`apply_extension(#U);`
	`EA=#U;`
	`*EA = Nt.new;`
	`Re=#U;`

`memw(Rs+#s11:2)=Nt.new`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`*EA = Nt.new;`

`memw(Rs+Ru<<#u2)=Nt.new`	`EA=Rs+(Ru<<#u);` `*EA = Nt.new;`

`memw(Ru<<#u2+#U6)=Nt.new`	`apply_extension(#U);`
	`EA=#U+(Ru<<#u);`
	`*EA = Nt.new;`

`memw(Rx++#s4:2)=Nt.new`	`EA=Rx;`
	`Rx=Rx+#s;`
	`*EA = Nt.new;`

`memw(Rx ++ #s4:2:circ(Mu)) = Nt.new`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`*EA = Nt.new;`

`memw(Rx++I:circ(Mu))=Nt.new`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<2,MuV);`
	`*EA = Nt.new;`

`memw(Rx++Mu)=Nt.new`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`*EA = Nt.new;`

`memw(Rx++Mu:brev)=Nt.new`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`*EA = Nt.new;`

`memw(gp+#u16:2)=Nt.new`	`apply_extension(#u);`
	`EA=(Constant_extended ? (0) : GP)+#u;`
	`*EA = Nt.new;`

Class: NV (slots 0)

Notes

- Forms of this instruction that use a new-value operand produced in the packet must execute on slot 0.
- This instruction can execute only in slot 0, even though it is an ST instruction.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5									t3			
0	0	1	1	1	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**-**	**-**	1	0	**t**	**t**	**t**	memw(Rs+Ru<<#u2)=Nt.new
ICLASS								Type								Parse					t3											
0	1	0	0	1	**i**	**i**	0	1	0	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	1	0	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memw(gp+#u16:2)=Nt.new
ICLASS				Amode			Type			UN	s5					Parse					t3											
1	0	1	0	0	**i**	**i**	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	1	0	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memw(Rs+#s11:2)=Nt.new
ICLASS				Amode			Type			UN	x5					Parse		u1			t3											
1	0	1	0	1	0	0	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	1	0	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	1	**-**	memw(Rx++I:circ(Mu))=Nt.new
1	0	1	0	1	0	0	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	1	0	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	0	**-**	memw(Rx++#s4:2:circ(Mu))=Nt.new
ICLASS				Amode			Type			UN	e5					Parse					t3											
1	0	1	0	1	0	1	1	1	0	1	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	1	0	**t**	**t**	**t**	1	**-**	**I**	**I**	**I**	**I**	**I**	**I**	memw(Re=#U6)=Nt.new
ICLASS				Amode			Type			UN	x5					Parse					t3											
1	0	1	0	1	0	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	1	0	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	0	**-**	memw(Rx++#s4:2)=Nt.new
ICLASS				Amode			Type			UN	u5					Parse					t3											
1	0	1	0	1	1	0	1	1	0	1	**u**	**u**	**u**	**u**	**u**	**P**	**P**	**i**	1	0	**t**	**t**	**t**	1	**i**	**I**	**I**	**I**	**I**	**I**	**I**	memw(Ru<<#u2+#U6)=Nt.new
ICLASS				Amode			Type			UN	x5					Parse		u1			t3											
1	0	1	0	1	1	0	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	1	0	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memw(Rx++Mu)=Nt.new
1	0	1	0	1	1	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	1	0	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memw(Rx++Mu:brev)=Nt.new

Field name	Description
ICLASS	Instruction class
Type	Type
Parse	Packet/loop parse bits
e5	Field to encode register e
s5	Field to encode register s
t3	Field to encode register t
u1	Field to encode register u
u5	Field to encode register u
x5	Field to encode register x
Amode	Amode
UN	Unsigned

Store new-value word conditionally

Store a 32-bit register in memory at the effective address.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior

`if ([!]Pv[.new]) memw(#u6)=Nt.new`	`apply_extension(#u);`
	`EA=#u;`
	`if ([!]Pv[.new][0]) {`
	`  *EA = Nt[.new];`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new]) memw(Rs+#u6:2)=Nt.new`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`if ([!]Pv[.new][0]) {`
	`  *EA = Nt[.new];`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new]) memw(Rs+Ru<<#u2)=Nt.new`	`EA=Rs+(Ru<<#u);`
	`if ([!]Pv[.new][0]) {`
	`  *EA = Nt[.new];`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new]) memw(Rx++#s4:2)=Nt.new`	`EA=Rx;`
	`if ([!]Pv[.new][0]){`
	`  Rx=Rx+#s;`
	`  *EA = Nt[.new];`
	`} else {`
	`  NOP;`
	`}`

Class: NV (slots 0)

Notes

- Forms of this instruction that use a new-value operand produced in the packet must execute on slot 0.
- This instruction can execute only in slot 0, even though it is an ST instruction.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5									t3			
0	0	1	1	0	1	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	1	0	**t**	**t**	**t**	if (Pv) memw(Rs+Ru<<#u2)=Nt.new
0	0	1	1	0	1	0	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	1	0	**t**	**t**	**t**	if (!Pv) memw(Rs+Ru<<#u2)=Nt.new
0	0	1	1	0	1	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	1	0	**t**	**t**	**t**	if (Pv.new) memw(Rs+Ru<<#u2)=Nt.new
0	0	1	1	0	1	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	1	0	**t**	**t**	**t**	if (!Pv.new) memw(Rs+Ru<<#u2)=Nt.new
ICLASS					Sense	PredNew		Type			s5					Parse					t3											
0	1	0	0	0	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	1	0	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memw(Rs+#u6:2)=Nt.new
0	1	0	0	0	0	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	1	0	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memw(Rs+#u6:2)=Nt.new
0	1	0	0	0	1	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	1	0	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv) memw(Rs+#u6:2)=Nt.new
0	1	0	0	0	1	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	1	0	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv.new) memw(Rs+#u6:2)=Nt.new
ICLASS				Amode			Type			UN	x5					Parse					t3											
1	0	1	0	1	0	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	0	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memw(Rx++#s4:2)=Nt.new
1	0	1	0	1	0	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	0	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memw(Rx++#s4:2)=Nt.new
1	0	1	0	1	0	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	0	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memw(Rx++#s4:2)=Nt.new
1	0	1	0	1	0	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	0	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memw(Rx++#s4:2)=Nt.new
ICLASS				Amode			Type			UN						Parse					t3											
1	0	1	0	1	1	1	1	1	0	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	1	0	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memw(#u6)=Nt.new
1	0	1	0	1	1	1	1	1	0	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	1	0	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memw(#u6)=Nt.new
1	0	1	0	1	1	1	1	1	0	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	1	0	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memw(#u6)=Nt.new
1	0	1	0	1	1	1	1	1	0	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	1	0	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memw(#u6)=Nt.new

Field name	Description
ICLASS	Instruction class
Type	Type
PredNew	PredNew
Sense	Sense
Parse	Packet/loop parse bits
s5	Field to encode register s
t3	Field to encode register t
u5	Field to encode register u
v2	Field to encode register v
x5	Field to encode register x
Amode	Amode
UN	Unsigned

11.8 ST

The ST instruction class includes store instructions, used to store values in memory.

ST instructions are executable on slot 0 and slot 1.

Store doubleword

Store a 64-bit register pair in memory at the effective address.

Syntax	Behavior

`memd(Re=#U6)=Rtt`	`apply_extension(#U);`
	`EA=#U;`
	`*EA = Rtt;`
	`Re=#U;`

`memd(Rs+#s11:3)=Rtt`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`*EA = Rtt;`

`memd(Rs+Ru<<#u2)=Rtt`	`EA=Rs+(Ru<<#u);` `*EA = Rtt;`

`memd(Ru<<#u2+#U6)=Rtt`	`apply_extension(#U);`
	`EA=#U+(Ru<<#u);`
	`*EA = Rtt;`

`memd(Rx++#s4:3)=Rtt`	`EA=Rx;`
	`Rx=Rx+#s;`
	`*EA = Rtt;`

`memd(Rx++#s4:3:circ(Mu))=Rtt`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`*EA = Rtt;`

`memd(Rx++I:circ(Mu))=Rtt`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<3,MuV);`
	`*EA = Rtt;`

`memd(Rx++Mu)=Rtt`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`*EA = Rtt;`

`memd(Rx++Mu:brev)=Rtt`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`*EA = Rtt;`

`memd(gp+#u16:3)=Rtt`	`apply_extension(#u);`
	`EA=(Constant_extended ? (0) : GP)+#u;`
	`*EA = Rtt;`

Class: ST (slots 0,1)

Intrinsics

`memd(Rx++#s4:3:circ(Mu))=Rtt`	`void Q6_memd_IMP_circ(void** StartAddress, Word32 Is4_3, Word32 Mu, Word64 Rtt, void* BaseAddress)`
`memd(Rx++I:circ(Mu))=Rtt`	`void Q6_memd_MP_circ(void** StartAddress, Word32 Mu, Word64 Rtt, void* BaseAddress)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5								t5					
0	0	1	1	1	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**-**	**-**	**t**	**t**	**t**	**t**	**t**	memd(Rs+Ru<<#u2)=Rtt
ICLASS								Type								Parse			t5													
0	1	0	0	1	**i**	**i**	0	1	1	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memd(gp+#u16:3)=Rtt
ICLASS				Amode			Type			UN	s5					Parse			t5													
1	0	1	0	0	**i**	**i**	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memd(Rs+#s11:3)=Rtt
ICLASS				Amode			Type			UN	x5					Parse		u1	t5													
1	0	1	0	1	0	0	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	1	**-**	memd(Rx++I:circ(Mu))=Rtt
1	0	1	0	1	0	0	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	0	**-**	memd(Rx++#s4:3:circ(Mu))=Rtt
ICLASS				Amode			Type			UN	e5					Parse			t5													
1	0	1	0	1	0	1	1	1	1	0	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**-**	**I**	**I**	**I**	**I**	**I**	**I**	memd(Re=#U6)=Rtt
ICLASS				Amode			Type			UN	x5					Parse			t5													
1	0	1	0	1	0	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	0	0	memd(Rx++#s4:3)=Rtt
ICLASS				Amode			Type			UN	u5					Parse			t5													
1	0	1	0	1	1	0	1	1	1	0	**u**	**u**	**u**	**u**	**u**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	1	**i**	**I**	**I**	**I**	**I**	**I**	**I**	memd(Ru<<#u2+#U6)=Rtt
ICLASS				Amode			Type			UN	x5					Parse		u1	t5													
1	0	1	0	1	1	0	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memd(Rx++Mu)=Rtt
1	0	1	0	1	1	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memd(Rx++Mu:brev)=Rtt

Field name	Description
ICLASS	Instruction class
Type	Type
Parse	Packet/loop parse bits
e5	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
u5	Field to encode register u
x5	Field to encode register x
Amode	Amode
Type	Type
UN	Unsigned

Store-release doubleword

Store a 64-bit register pair in memory at the effective address. The store-release memory operation is observed after all preceding memory operations have been observed at the local point of serialization. A different order can be observed at the global point of serialization (see Memory ordering).

When the :st (same domain) option is specified, the preceding memory operations are those that were committed on any thread with the same Consistency Domain before this instruction was committed.

When the :at (all threads) option is specified, the preceding memory operations are those that were committed on any thread before this instruction was committed.

The Store release address is limited to certain memory regions. The following are excluded memory regions: AHB memory space, AXI M2 memory space, Hexagon memory cut-out is excluded with the exception of addressable TCM and VTCM memory, and memory with the CCCC types 2, 3, or 4 are excluded. The :st option does not apply to cache operation by index or global cache operation. The :st option does not apply a consistency domain to vector operations, but instead uses a per hardware thread ordering scope.

Syntax	Behavior
`memd_rl(Rs):at=Rtt`	`EA=Rs;` `*EA = Rtt`
`memd_rl(Rs):st=Rtt`	`EA=Rs;` `*EA = Rtt`

Class: ST (slots 0)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	s5					Parse			t5											d2		
1	0	1	0	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	-	-	0	0	1	0	**d**	**d**	memd_rl(Rs):at=Rtt
1	0	1	0	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	-	-	1	0	1	0	**d**	**d**	memd_rl(Rs):st=Rtt

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d2	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
Amode	Amode
Type	Type
UN	Unsigned

Store doubleword conditionally

Store a 64-bit register pair in memory at the effective address.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior

`if ([!]Pv[.new]) memd(#u6)=Rtt`	`apply_extension(#u);`
	`EA=#u;`
	`if ([!]Pv[.new][0]) {`
	`  *EA = Rtt;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new]) memd(Rs+#u6:3)=Rtt`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`if ([!]Pv[.new][0]) {`
	`  *EA = Rtt;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new]) memd(Rs+Ru<<#u2)=Rtt`	`EA=Rs+(Ru<<#u);`
	`if ([!]Pv[.new][0]) {`
	`  *EA = Rtt;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new]) memd(Rx++#s4:3)=Rtt`	`EA=Rx;`
	`if ([!]Pv[.new][0]){`
	`  Rx=Rx+#s;`
	`  *EA = Rtt;`
	`} else {`
	`  NOP;`
	`}`

Class: ST (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5								t5						
0	0	1	1	0	1	0	0	1	1	0	s	s	s	s	s	P	P	i	u	u	u	u	u	i	v	v	t	t	t	t	t	if (Pv) memd(Rs+Ru<<#u2)=Rtt
0	0	1	1	0	1	0	1	1	1	0	s	s	s	s	s	P	P	i	u	u	u	u	u	i	v	v	t	t	t	t	t	if (!Pv) memd(Rs+Ru<<#u2)=Rtt
0	0	1	1	0	1	1	0	1	1	0	s	s	s	s	s	P	P	i	u	u	u	u	u	i	v	v	t	t	t	t	t	if (Pv.new) memd(Rs+Ru<<#u2)=Rtt
0	0	1	1	0	1	1	1	1	1	0	s	s	s	s	s	P	P	i	u	u	u	u	u	i	v	v	t	t	t	t	t	if (!Pv.new) memd(Rs+Ru<<#u2)=Rtt
ICLASS					Sense	Pred New		Type			s5					Parse			t5													

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	0	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memd(Rs+#u6:3)=Rtt
0	1	0	0	0	0	1	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memd(Rs+#u6:3)=Rtt
0	1	0	0	0	1	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv) memd(Rs+#u6:3)=Rtt
0	1	0	0	0	1	1	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv.new) memd(Rs+#u6:3)=Rtt
ICLASS				Amode			Type			UN	x5					Parse			t5													
1	0	1	0	1	0	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memd(Rx++#s4:3)=Rtt
1	0	1	0	1	0	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memd(Rx++#s4:3)=Rtt
1	0	1	0	1	0	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memd(Rx++#s4:3)=Rtt
1	0	1	0	1	0	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memd(Rx++#s4:3)=Rtt
ICLASS				Amode			Type			UN						Parse			t5													
1	0	1	0	1	1	1	1	1	1	0	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memd(#u6)=Rtt
1	0	1	0	1	1	1	1	1	1	0	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memd(#u6)=Rtt
1	0	1	0	1	1	1	1	1	1	0	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memd(#u6)=Rtt
1	0	1	0	1	1	1	1	1	1	0	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memd(#u6)=Rtt

Field name	Description
ICLASS	Instruction class
Type	Type
PredNew	PredNew
Sense	Sense
Parse	Packet/loop parse bits
s5	Field to encode register s
t5	Field to encode register t
u5	Field to encode register u
v2	Field to encode register v
x5	Field to encode register x
Amode	Amode
UN	Unsigned

Store byte

Store the least-significant byte in a source register at the effective address.

Syntax	Behavior

`memb(Re=#U6)=Rt`	`apply_extension(#U);`
	`EA=#U;`
	`*EA = Rt.b[0];`
	`Re=#U;`

`memb(Rs+#s11:0)=Rt`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`*EA = Rt.b[0];`

`memb(Rs+#u6:0)=#S8`	`EA=Rs+#u;`
	`apply_extension(#S);`
	`*EA = #S;`

`memb(Rs+Ru<<#u2)=Rt`	`EA=Rs+(Ru<<#u);` `*EA = Rt.b[0];`

`memb(Ru<<#u2+#U6)=Rt`	`apply_extension(#U);`
	`EA=#U+(Ru<<#u);`
	`*EA = Rt.b[0];`

`memb(Rx++#s4:0)=Rt`	`EA=Rx;`
	`Rx=Rx+#s;`
	`*EA = Rt.b[0];`

`memb(Rx++#s4:0:circ(Mu))=Rt`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`*EA = Rt.b[0];`

`memb(Rx++I:circ(Mu))=Rt`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<0,MuV);`
	`*EA = Rt.b[0];`

`memb(Rx++Mu)=Rt`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`*EA = Rt.b[0];`

`memb(Rx++Mu:brev)=Rt`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`*EA = Rt.b[0];`

`memb(gp+#u16:0)=Rt`	`apply_extension(#u);`
	`EA=(Constant_extended ? (0) : GP)+#u;`
	`*EA = Rt.b[0];`

Class: ST (slots 0,1)

Intrinsics

`memb(Rx++#s4:0:circ(Mu))=Rt`	`void Q6_memb_IMR_circ(void** StartAddress, Word32 Is4_0, Word32 Mu, Word32 Rt, void* BaseAddress)`
`memb(Rx++I:circ(Mu))=Rt`	`void Q6_memb_MR_circ(void** StartAddress, Word32 Mu, Word32 Rt, void* BaseAddress)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5								t5					
0	0	1	1	1	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**-**	**-**	**t**	**t**	**t**	**t**	**t**	memb(Rs+Ru<<#u2)=Rt
ICLASS											s5					Parse																
0	0	1	1	1	1	0	**-**	**-**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**I**	**I**	**I**	**I**	**I**	**I**	**I**	memb(Rs+#u6:0)=#S8
ICLASS								Type								Parse			t5													
0	1	0	0	1	**i**	**i**	0	0	0	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memb(gp+#u16:0)=Rt
ICLASS				Amode			Type			U N	s5					Parse			t5													
1	0	1	0	0	**i**	**i**	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memb(Rs+#s11:0)=Rt
ICLASS				Amode			Type			U N	x5					Parse		u1	t5													
1	0	1	0	1	0	0	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	1	**-**	memb(Rx++I:circ(Mu))=Rt
1	0	1	0	1	0	0	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	0	**-**	memb(Rx++#s4:0:circ(Mu)) =Rt
ICLASS				Amode			Type			U N	e5					Parse			t5													
1	0	1	0	1	0	1	1	0	0	0	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**-**	**I**	**I**	**I**	**I**	**I**	**I**	memb(Re=#U6)=Rt
ICLASS				Amode			Type			U N	x5					Parse			t5													
1	0	1	0	1	0	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	0	**-**	memb(Rx++#s4:0)=Rt
ICLASS				Amode			Type			U N	u5					Parse			t5													
1	0	1	0	1	1	0	1	0	0	0	**u**	**u**	**u**	**u**	**u**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	1	**i**	**I**	**I**	**I**	**I**	**I**	**I**	memb(Ru<<#u2+#U6)=Rt
ICLASS				Amode			Type			U N	x5					Parse		u1	t5													
1	0	1	0	1	1	0	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memb(Rx++Mu)=Rt
1	0	1	0	1	1	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memb(Rx++Mu:brev)=Rt

Field name	Description
ICLASS	Instruction class
Type	Type
Parse	Packet/loop parse bits
e5	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
u5	Field to encode register u
x5	Field to encode register x
Amode	Amode
UN	Unsigned

Store byte conditionally

Store the least-significant byte in a source register at the effective address.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior

`if ([!]Pv[.new])`
`memb(#u6)=Rt`
	`apply_extension(#u);`
	`EA=#u;`
	`if ([!]Pv[.new][0]) {`
	`  *EA = Rt.b[0];`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new])`
`memb(Rs+#u6:0)=#S6`
	`EA=Rs+#u;`
	`if ([!]Pv[.new][0]){`
	`  apply_extension(#S);`
	`  *EA = #S;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new])`
`memb(Rs+#u6:0)=Rt`
	`apply_extension(#u);`
	`EA=Rs+#u;`
	`if ([!]Pv[.new][0]) {`
	`  *EA = Rt.b[0];`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new])`
`memb(Rs+Ru<<#u2)=Rt`
	`EA=Rs+(Ru<<#u);`
	`if ([!]Pv[.new][0]) {`
	`  *EA = Rt.b[0];`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new])`
`memb(Rx++#s4:0)=Rt`
	`EA=Rx;`
	`if ([!]Pv[.new][0]){`
	`  Rx=Rx+#s;`
	`  *EA = Rt.b[0];`
	`} else {`
	`  NOP;`
	`}`

Class: ST (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5								t5					
0	0	1	1	0	1	0	0	0	0	0	s	s	s	s	s	P	P	i	u	u	u	u	u	i	v	v	t	t	t	t	t	if (Pv) memb(Rs+Ru<<#u2)=Rt

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	1	0	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	**t**	**t**	**t**	**t**	**t**	if (!Pv) memb(Rs+Ru<<#u2)=Rt
0	0	1	1	0	1	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	**t**	**t**	**t**	**t**	**t**	if (Pv.new) memb(Rs+Ru<<#u2)=Rt
0	0	1	1	0	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	**t**	**t**	**t**	**t**	**t**	if (!Pv.new) memb(Rs+Ru<<#u2)=Rt
ICLASS											s5					Parse																
0	0	1	1	1	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**v**	**v**	**I**	**I**	**I**	**I**	**I**	if (Pv) memb(Rs+#u6:0)=#S6
0	0	1	1	1	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**v**	**v**	**I**	**I**	**I**	**I**	**I**	if (!Pv) memb(Rs+#u6:0)=#S6
0	0	1	1	1	0	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**v**	**v**	**I**	**I**	**I**	**I**	**I**	if (Pv.new) memb(Rs+#u6:0)=#S6
0	0	1	1	1	0	0	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**v**	**v**	**I**	**I**	**I**	**I**	**I**	if (!Pv.new) memb(Rs+#u6:0)=#S6
ICLASS					Sense	PredNew		Type			s5					Parse			t5													
0	1	0	0	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memb(Rs+#u6:0)=Rt
0	1	0	0	0	0	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memb(Rs+#u6:0)=Rt
0	1	0	0	0	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv) memb(Rs+#u6:0)=Rt
0	1	0	0	0	1	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv.new) memb(Rs+#u6:0)=Rt
ICLASS				Amode			Type			UN	x5					Parse			t5													
1	0	1	0	1	0	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memb(Rx++#s4:0)=Rt
1	0	1	0	1	0	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memb(Rx++#s4:0)=Rt
1	0	1	0	1	0	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memb(Rx++#s4:0)=Rt
1	0	1	0	1	0	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memb(Rx++#s4:0)=Rt
ICLASS				Amode			Type			UN						Parse			t5													
1	0	1	0	1	1	1	1	0	0	0	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memb(#u6)=Rt
1	0	1	0	1	1	1	1	0	0	0	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memb(#u6)=Rt
1	0	1	0	1	1	1	1	0	0	0	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memb(#u6)=Rt
1	0	1	0	1	1	1	1	0	0	0	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memb(#u6)=Rt

Field name	Description
ICLASS	Instruction class
Type	Type
PredNew	PredNew
Sense	Sense
Parse	Packet/loop parse bits
s5	Field to encode register s
t5	Field to encode register t
u5	Field to encode register u
v2	Field to encode register v
x5	Field to encode register x
Amode	Amode
UN	Unsigned

Store halfword

Store the upper or lower 16-bits of a source register at the effective address.

Syntax	Behavior

`memh(Re=#U6)=Rt.H`	`apply_extension(#U);`
	`EA=#U;`
	`*EA = Rt.h[1];`
	`Re=#U;`

`memh(Re=#U6)=Rt`	`apply_extension(#U);`
	`EA=#U;`
	`*EA = Rt.h[0];`
	`Re=#U;`

`memh(Rs+#s11:1)=Rt.H`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`*EA = Rt.h[1];`

`memh(Rs+#s11:1)=Rt`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`*EA = Rt.h[0];`

`memh(Rs+#u6:1)=#S8`	`EA=Rs+#u;`
	`apply_extension(#S);`
	`*EA = #S;`

`memh(Rs+Ru<<#u2)=Rt.H`	`EA=Rs+(Ru<<#u);` `*EA = Rt.h[1];`
`memh(Rs+Ru<<#u2)=Rt`	`EA=Rs+(Ru<<#u);` `*EA = Rt.h[0];`

`memh(Ru<<#u2+#U6)=Rt.H`	`apply_extension(#U);`
	`EA=#U+(Ru<<#u);`
	`*EA = Rt.h[1];`

`memh(Ru<<#u2+#U6)=Rt`	`apply_extension(#U);`
	`EA=#U+(Ru<<#u);`
	`*EA = Rt.h[0];`

`memh(Rx++#s4:1)=Rt.H`	`EA=Rx;`
	`Rx=Rx+#s;`
	`*EA = Rt.h[1];`

`memh(Rx++#s4:1)=Rt`	`EA=Rx;`
	`Rx=Rx+#s;`
	`*EA = Rt.h[0];`

`memh(Rx++#s4:1:circ(Mu)) = Rt.H`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`*EA = Rt.h[1];`

`memh(Rx++#s4:1:circ(Mu))=Rt`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`*EA = Rt.h[0];`

`memh(Rx++I:circ(Mu))=Rt.H`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<1,MuV);`
	`*EA = Rt.h[1];`

Syntax	Behavior

`memh(Rx++I:circ(Mu))=Rt`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<1,MuV);`
	`*EA = Rt.h[0];`

`memh(Rx++Mu)=Rt.H`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`*EA = Rt.h[1];`

`memh(Rx++Mu)=Rt`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`*EA = Rt.h[0];`

`memh(Rx++Mu:brev)=Rt.H`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`*EA = Rt.h[1];`

`memh(Rx++Mu:brev)=Rt`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`*EA = Rt.h[0];`

`memh(gp+#u16:1)=Rt.H`	`apply_extension(#u);`
	`EA=(Constant_extended ? (0) : GP)+#u;`
	`*EA = Rt.h[1];`

`memh(gp+#u16:1)=Rt`	`apply_extension(#u);`
	`EA=(Constant_extended ? (0) : GP)+#u;`
	`*EA = Rt.h[0];`

Class: ST (slots 0,1)

Intrinsics

`memh(Rx++#s4:1:circ(Mu)) = Rt.H`	`void Q6_memh_IMRh_circ(void** StartAddress, Word32 Is4_1, Word32 Mu, Word32 Rt, void* BaseAddress)`
`memh(Rx++#s4:1:circ(Mu))=Rt`	`void Q6_memh_IMR_circ(void** StartAddress, Word32 Is4_1, Word32 Mu, Word32 Rt, void* BaseAddress)`
`memh(Rx++I:circ(Mu))=Rt.H`	`void Q6_memh_MRh_circ(void** StartAddress, Word32 Mu, Word32 Rt, void* BaseAddress)`
`memh(Rx++I:circ(Mu))=Rt`	`void Q6_memh_MR_circ(void** StartAddress, Word32 Mu, Word32 Rt, void* BaseAddress)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5								t5					
0	0	1	1	1	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**-**	**-**	**t**	**t**	**t**	**t**	**t**	memh(Rs+Ru<<#u2)=Rt
0	0	1	1	1	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**-**	**-**	**t**	**t**	**t**	**t**	**t**	memh(Rs+Ru<<#u2)=Rt.H
ICLASS											s5					Parse																
0	0	1	1	1	1	0	**-**	**-**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**I**	**I**	**I**	**I**	**I**	**I**	**I**	memh(Rs+#u6:1)=#S8
ICLASS								Type								Parse			t5													

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	0	1	**i**	**i**	0	0	1	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memh(gp+#u16:1)=Rt
0	1	0	0	1	**i**	**i**	0	0	1	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memh(gp+#u16:1)=Rt.H
ICLASS				Amode			Type			UN	s5					Parse			t5													
1	0	1	0	0	**i**	**i**	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memh(Rs+#s11:1)=Rt
1	0	1	0	0	**i**	**i**	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memh(Rs+#s11:1)=Rt.H
ICLASS				Amode			Type			UN	x5					Parse		u1	t5													
1	0	1	0	1	0	0	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	1	**-**	memh(Rx++I:circ(Mu))=Rt
1	0	1	0	1	0	0	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	0	**-**	memh(Rx++#s4:1:circ(Mu))=Rt
1	0	1	0	1	0	0	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	1	**-**	memh(Rx++I:circ(Mu))=Rt.H
1	0	1	0	1	0	0	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	0	**-**	memh(Rx++#s4:1:circ(Mu))=Rt.H
ICLASS				Amode			Type			UN	e5					Parse			t5													
1	0	1	0	1	0	1	1	0	1	0	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**-**	**I**	**I**	**I**	**I**	**I**	**I**	memh(Re=#U6)=Rt
ICLASS				Amode			Type			UN	x5					Parse			t5													
1	0	1	0	1	0	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	0	**-**	memh(Rx++#s4:1)=Rt
ICLASS				Amode			Type			UN	e5					Parse			t5													
1	0	1	0	1	0	1	1	0	1	1	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**-**	**I**	**I**	**I**	**I**	**I**	**I**	memh(Re=#U6)=Rt.H
ICLASS				Amode			Type			UN	x5					Parse			t5													
1	0	1	0	1	0	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	0	**-**	memh(Rx++#s4:1)=Rt.H
ICLASS				Amode			Type			UN	u5					Parse			t5													
1	0	1	0	1	1	0	1	0	1	0	**u**	**u**	**u**	**u**	**u**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	1	**i**	**I**	**I**	**I**	**I**	**I**	**I**	memh(Ru<<#u2+#U6)=Rt
ICLASS				Amode			Type			UN	x5					Parse		u1	t5													
1	0	1	0	1	1	0	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memh(Rx++Mu)=Rt
ICLASS				Amode			Type			UN	u5					Parse			t5													
1	0	1	0	1	1	0	1	0	1	1	**u**	**u**	**u**	**u**	**u**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	1	**i**	**I**	**I**	**I**	**I**	**I**	**I**	memh(Ru<<#u2+#U6)=Rt.H
ICLASS				Amode			Type			UN	x5					Parse		u1	t5													
1	0	1	0	1	1	0	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memh(Rx++Mu)=Rt.H
1	0	1	0	1	1	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memh(Rx++Mu:brev)=Rt
1	0	1	0	1	1	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memh(Rx++Mu:brev)=Rt.H

Field name	Description
ICLASS	Instruction class
Type	Type
Parse	Packet/loop parse bits
e5	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
u5	Field to encode register u
x5	Field to encode register x
Amode	Amode
UN	Unsigned

Store halfword conditionally

Store the upper or lower 16-bits of a source register in memory at the effective address.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior

`if ([!]Pv[.new]) memh(#u6)=Rt.H`	`apply_extension(#u);`
	`EA=#u;`
	`if ([!]Pv[.new][0]) {`
	`  *EA = Rt.h[1];`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new]) memh(#u6)=Rt`	`apply_extension(#u);`
	`EA=#u;`
	`if ([!]Pv[.new][0]) {`
	`  *EA = Rt.h[0];`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new]) memh(Rs+#u6:1)=#S6`	`EA=Rs+#u;`
	`if ([!]Pv[.new][0]){`
	`  apply_extension(#S);`
	`  *EA = #S;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new]) memh(Rs+#u6:1)=Rt.H`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`if ([!]Pv[.new][0]) {`
	`  *EA = Rt.h[1];`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new]) memh(Rs+#u6:1)=Rt`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`if ([!]Pv[.new][0]) {`
	`  *EA = Rt.h[0];`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new]) memh(Rs+Ru<<#u2)=Rt.H`	`EA=Rs+(Ru<<#u);`
	`if ([!]Pv[.new][0]) {`
	`  *EA = Rt.h[1];`
	`} else {`
	`  NOP;`
	`}`

Syntax	Behavior

`if ([!]Pv[.new]) memh(Rs+Ru<<#u2)=Rt`	`EA=Rs+(Ru<<#u);`
	`if ([!]Pv[.new][0]) {`
	`  *EA = Rt.h[0];`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new]) memh(Rx++#s4:1)=Rt.H`	`EA=Rx;`
	`if ([!]Pv[.new][0]){`
	`  Rx=Rx+#s;`
	`  *EA = Rt.h[1];`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new]) memh(Rx++#s4:1)=Rt`	`EA=Rx;`
	`if ([!]Pv[.new][0]){`
	`  Rx=Rx+#s;`
	`  *EA = Rt.h[0];`
	`} else {`
	`  NOP;`
	`}`

Class: ST (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5								t5					
0	0	1	1	0	1	0	0	0	1	0	s	s	s	s	s	P	P	i	u	u	u	u	u	i	v	v	t	t	t	t	t	if (Pv) memh(Rs+Ru<<#u2)=Rt
0	0	1	1	0	1	0	0	0	1	1	s	s	s	s	s	P	P	i	u	u	u	u	u	i	v	v	t	t	t	t	t	if (Pv) memh(Rs+Ru<<#u2)=Rt.H
0	0	1	1	0	1	0	1	0	1	0	s	s	s	s	s	P	P	i	u	u	u	u	u	i	v	v	t	t	t	t	t	if (!Pv) memh(Rs+Ru<<#u2)=Rt
0	0	1	1	0	1	0	1	0	1	1	s	s	s	s	s	P	P	i	u	u	u	u	u	i	v	v	t	t	t	t	t	if (!Pv) memh(Rs+Ru<<#u2)=Rt.H
0	0	1	1	0	1	1	0	0	1	0	s	s	s	s	s	P	P	i	u	u	u	u	u	i	v	v	t	t	t	t	t	if (Pv.new) memh(Rs+Ru<<#u2)=Rt
0	0	1	1	0	1	1	0	0	1	1	s	s	s	s	s	P	P	i	u	u	u	u	u	i	v	v	t	t	t	t	t	if (Pv.new) memh(Rs+Ru<<#u2)=Rt.H
0	0	1	1	0	1	1	1	0	1	0	s	s	s	s	s	P	P	i	u	u	u	u	u	i	v	v	t	t	t	t	t	if (!Pv.new) memh(Rs+Ru<<#u2)=Rt
0	0	1	1	0	1	1	1	0	1	1	s	s	s	s	s	P	P	i	u	u	u	u	u	i	v	v	t	t	t	t	t	if (!Pv.new) memh(Rs+Ru<<#u2)=Rt.H
ICLASS											s5					Parse																
0	0	1	1	1	0	0	0	0	0	1	s	s	s	s	s	P	P	I	i	i	i	i	i	i	v	v	I	I	I	I	I	if (Pv) memh(Rs+#u6:1)=#S6
0	0	1	1	1	0	0	0	1	0	1	s	s	s	s	s	P	P	I	i	i	i	i	i	i	v	v	I	I	I	I	I	if (!Pv) memh(Rs+#u6:1)=#S6
0	0	1	1	1	0	0	1	0	0	1	s	s	s	s	s	P	P	I	i	i	i	i	i	i	v	v	I	I	I	I	I	if (Pv.new) memh(Rs+#u6:1)=#S6
0	0	1	1	1	0	0	1	1	0	1	s	s	s	s	s	P	P	I	i	i	i	i	i	i	v	v	I	I	I	I	I	if (!Pv.new) memh(Rs+#u6:1)=#S6
ICLASS					Sense	Pred New		Type			s5					Parse			t5													
0	1	0	0	0	0	0	0	0	1	0	s	s	s	s	s	P	P	i	t	t	t	t	t	i	i	i	i	i	0	v	v	if (Pv) memh(Rs+#u6:1)=Rt

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	0	0	0	0	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memh(Rs+#u6:1)=Rt.H
0	1	0	0	0	0	1	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memh(Rs+#u6:1)=Rt
0	1	0	0	0	0	1	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memh(Rs+#u6:1)=Rt.H
0	1	0	0	0	1	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv) memh(Rs+#u6:1)=Rt
0	1	0	0	0	1	0	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv) memh(Rs+#u6:1)=Rt.H
0	1	0	0	0	1	1	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv.new) memh(Rs+#u6:1)=Rt
0	1	0	0	0	1	1	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv.new) memh(Rs+#u6:1)=Rt.H
ICLASS				Amode			Type			U N	x5					Parse			t5													
1	0	1	0	1	0	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memh(Rx++#s4:1)=Rt
1	0	1	0	1	0	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memh(Rx++#s4:1)=Rt
1	0	1	0	1	0	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memh(Rx++#s4:1)=Rt
1	0	1	0	1	0	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memh(Rx++#s4:1)=Rt
1	0	1	0	1	0	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memh(Rx++#s4:1)=Rt.H
1	0	1	0	1	0	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memh(Rx++#s4:1)=Rt.H
1	0	1	0	1	0	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memh(Rx++#s4:1)=Rt.H
1	0	1	0	1	0	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memh(Rx++#s4:1)=Rt.H
ICLASS				Amode			Type			U N						Parse			t5													
1	0	1	0	1	1	1	1	0	1	0	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memh(#u6)=Rt
1	0	1	0	1	1	1	1	0	1	0	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memh(#u6)=Rt
1	0	1	0	1	1	1	1	0	1	0	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memh(#u6)=Rt
1	0	1	0	1	1	1	1	0	1	0	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memh(#u6)=Rt
1	0	1	0	1	1	1	1	0	1	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memh(#u6)=Rt.H
1	0	1	0	1	1	1	1	0	1	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memh(#u6)=Rt.H
1	0	1	0	1	1	1	1	0	1	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memh(#u6)=Rt.H
1	0	1	0	1	1	1	1	0	1	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memh(#u6)=Rt.H

Field name	Description
ICLASS	Instruction class
Type	Type
PredNew	PredNew
Sense	Sense
Parse	Packet/loop parse bits
s5	Field to encode register s
t5	Field to encode register t
u5	Field to encode register u
v2	Field to encode register v
x5	Field to encode register x
Amode	Amode
UN	Unsigned

Release

The release memory operation is observed after all preceding memory operations have been observed at the local point of serialization. A different order may be observed at the global point of serialization (see Memory ordering). No data is modified by this instruction.

When the :st (same domain) option is specified, the preceding memory operations are those that were committed on any thread with the same consistency domain before this instruction was committed.

When the :at (all threads) option is specified, the preceding memory operations are those that were committed on any thread before this instruction was committed.

The Store release address is limited to certain memory regions. The following are excluded memory regions: AHB memory space, AXI M2 memory space, Hexagon memory cut-out is excluded with the exception of addressable TCM and VTCM memory, and memory with the CCCC types 2, 3, or 4 are excluded. The :st option does not apply to cache operation by index or global cache operation. The :st option does not apply a consistency domain to vector operations, but instead uses a per hardware thread ordering scope.

NOTE: Observing the release operation means a later access to the target address by the issuing software thread.

Syntax	Behavior
`release(Rs):at`	`EA=Rs;` `*EA = Rs`
`release(Rs):st`	`EA=Rs;` `*EA = Rs`

Class: ST (slots 0)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	s5					Parse			t5											d2		
1	0	1	0	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	-	-	0	0	1	1	**d**	**d**	release(Rs):at
1	0	1	0	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	-	-	1	0	1	1	**d**	**d**	release(Rs):st

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d2	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
Amode	Amode
Type	Type
UN	Unsigned

Store word

Store a 32-bit register in memory at the effective address.

Syntax	Behavior

`memw(Re=#U6)=Rt`	`apply_extension(#U);`
	`EA=#U;`
	`*EA = Rt;`
	`Re=#U;`

`memw(Rs+#s11:2)=Rt`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`*EA = Rt;`

`memw(Rs+#u6:2)=#S8`	`EA=Rs+#u;`
	`apply_extension(#S);`
	`*EA = #S;`

`memw(Rs+Ru<<#u2)=Rt`	`EA=Rs+(Ru<<#u);` `*EA = Rt;`

`memw(Ru<<#u2+#U6)=Rt`	`apply_extension(#U);`
	`EA=#U+(Ru<<#u);`
	`*EA = Rt;`

`memw(Rx++#s4:2)=Rt`	`EA=Rx;`
	`Rx=Rx+#s;`
	`*EA = Rt;`

`memw(Rx++#s4:2:circ(Mu))=Rt`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`*EA = Rt;`

`memw(Rx++I:circ(Mu))=Rt`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<2,MuV);`
	`*EA = Rt;`

`memw(Rx++Mu)=Rt`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`*EA = Rt;`

`memw(Rx++Mu:brev)=Rt`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`*EA = Rt;`

`memw(gp+#u16:2)=Rt`	`apply_extension(#u);`
	`EA=(Constant_extended ? (0) : GP)+#u;`
	`*EA = Rt;`

Class: ST (slots 0,1)

Intrinsics

`memw(Rx++#s4:2:circ(Mu))=Rt`	`void Q6_memw_IMR_circ(void** StartAddress, Word32 Is4_2, Word32 Mu, Word32 Rt, void* BaseAddress)`
`memw(Rx++I:circ(Mu))=Rt`	`void Q6_memw_MR_circ(void** StartAddress, Word32 Mu, Word32 Rt, void* BaseAddress)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5								t5					
0	0	1	1	1	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**-**	**-**	**t**	**t**	**t**	**t**	**t**	memw(Rs+Ru<<#u2)=Rt
ICLASS											s5					Parse																
0	0	1	1	1	1	0	**-**	**-**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**I**	**I**	**I**	**I**	**I**	**I**	**I**	memw(Rs+#u6:2)=#S8
ICLASS								Type								Parse			t5													
0	1	0	0	1	**i**	**i**	0	1	0	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memw(gp+#u16:2)=Rt
ICLASS				Amode			Type			U N	s5					Parse			t5													
1	0	1	0	0	**i**	**i**	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memw(Rs+#s11:2)=Rt
ICLASS				Amode			Type			U N	x5					Parse		u1	t5													
1	0	1	0	1	0	0	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	1	**-**	memw(Rx++I:circ(Mu))=Rt
1	0	1	0	1	0	0	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	0	**-**	memw(Rx++#s4:2:circ(Mu))=Rt
ICLASS				Amode			Type			U N	e5					Parse			t5													
1	0	1	0	1	0	1	1	1	0	0	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**-**	**I**	**I**	**I**	**I**	**I**	**I**	memw(Re=#U6)=Rt
ICLASS				Amode			Type			U N	x5					Parse			t5													
1	0	1	0	1	0	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	0	**-**	memw(Rx++#s4:2)=Rt
ICLASS				Amode			Type			U N	u5					Parse			t5													
1	0	1	0	1	1	0	1	1	0	0	**u**	**u**	**u**	**u**	**u**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	1	**i**	**I**	**I**	**I**	**I**	**I**	**I**	memw(Ru<<#u2+#U6)=Rt
ICLASS				Amode			Type			U N	x5					Parse		u1	t5													
1	0	1	0	1	1	0	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memw(Rx++Mu)=Rt
1	0	1	0	1	1	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memw(Rx++Mu:brev)=Rt

Field name	Description
ICLASS	Instruction class
Type	Type
Parse	Packet/loop parse bits
e5	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
u5	Field to encode register u
x5	Field to encode register x
Amode	Amode
UN	Unsigned

Store-release word

Store a 32-bit register in memory at the effective address. The store-release memory operation is observed after all preceding memory operations have been observed at the local point of serialization. A different order may be observed at the global point of serialization (see Memory ordering).

When the :st (same domain) option is specified, the preceding memory operations are those that were committed on any thread with the same Consistency Domain before the release:st was committed.

When the :at (all threads) option is specified, the preceding memory operations are those that were committed on any thread before the release:at instruction was committed.

The Store release address is limited to certain memory regions. The following are excluded memory regions: AHB memory space, AXI M2 memory space, Hexagon memory cut-out is excluded with the exception of addressable TCM and VTCM memory, and memory with the CCCC types 2, 3, or 4 are excluded. The :st option does not apply to cache operation by index or global cache operation. The :st option does not apply a consistency domain to vector operations, but instead uses a per hardware thread ordering scope.

Syntax	Behavior
`memw_rl(Rs):at=Rt`	`EA=Rs;` `*EA = Rt`
`memw_rl(Rs):st=Rt`	`EA=Rs;` `*EA = Rt`

Class: ST (slots 0)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	s5					Parse			t5											d2		
1	0	1	0	0	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	0	0	1	0	**d**	**d**	memw_rl(Rs):at=Rt
1	0	1	0	0	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	1	0	1	0	**d**	**d**	memw_rl(Rs):st=Rt

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d2	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
Amode	Amode
Type	Type
UN	Unsigned

Store word conditionally

Store a 32-bit register in memory at the effective address.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior

`if ([!]Pv[.new]) memw(#u6)=Rt`	`apply_extension(#u);`
	`EA=#u;`
	`if ([!]Pv[.new][0]) {`
	`  *EA = Rt;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new]) memw(Rs+#u6:2)=#S6`	`EA=Rs+#u;`
	`if ([!]Pv[.new][0]){`
	`  apply_extension(#S);`
	`  *EA = #S;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new]) memw(Rs+#u6:2)=Rt`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`if ([!]Pv[.new][0]) {`
	`  *EA = Rt;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new]) memw(Rs+Ru<<#u2)=Rt`	`EA=Rs+(Ru<<#u);`
	`if ([!]Pv[.new][0]) {`
	`  *EA = Rt;`
	`} else {`
	`  NOP;`
	`}`

`if ([!]Pv[.new]) memw(Rx++#s4:2)=Rt`	`EA=Rx;`
	`if ([!]Pv[.new][0]){`
	`  Rx=Rx+#s;`
	`  *EA = Rt;`
	`} else {`
	`  NOP;`
	`}`

Class: ST (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5								t5					
0	0	1	1	0	1	0	0	1	0	0	s	s	s	s	s	P	P	i	u	u	u	u	u	i	v	v	t	t	t	t	t	if (Pv) memw(Rs+Ru<<#u2)=Rt
0	0	1	1	0	1	0	1	1	0	0	s	s	s	s	s	P	P	i	u	u	u	u	u	i	v	v	t	t	t	t	t	if (!Pv) memw(Rs+Ru<<#u2)=Rt

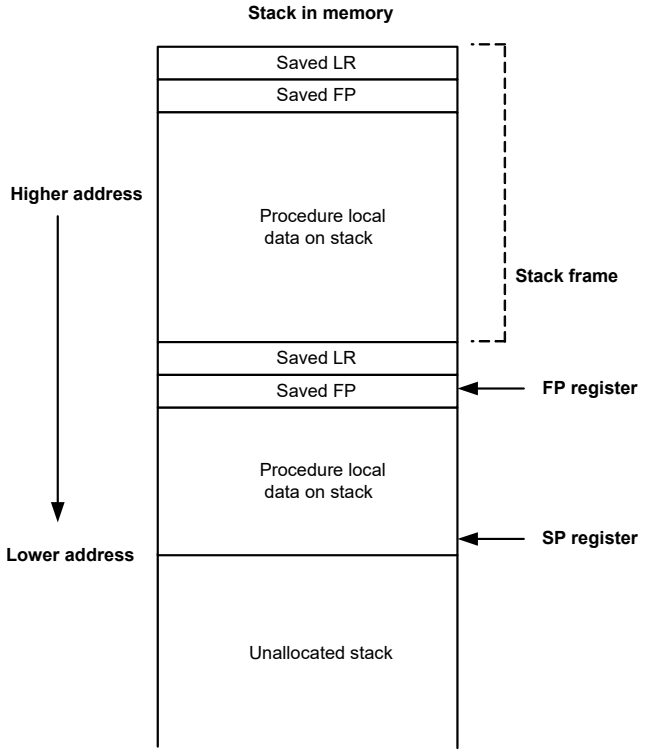
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	1	0	1	1	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	**t**	**t**	**t**	**t**	**t**	if (Pv.new) memw(Rs+Ru<<#u2)=Rt
0	0	1	1	0	1	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	**t**	**t**	**t**	**t**	**t**	if (!Pv.new) memw(Rs+Ru<<#u2)=Rt
ICLASS											s5					Parse																
0	0	1	1	1	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**v**	**v**	**I**	**I**	**I**	**I**	**I**	if (Pv) memw(Rs+#u6:2)=#S6
0	0	1	1	1	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**v**	**v**	**I**	**I**	**I**	**I**	**I**	if (!Pv) memw(Rs+#u6:2)=#S6
0	0	1	1	1	0	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**v**	**v**	**I**	**I**	**I**	**I**	**I**	if (Pv.new) memw(Rs+#u6:2)=#S6
0	0	1	1	1	0	0	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**v**	**v**	**I**	**I**	**I**	**I**	**I**	if (!Pv.new) memw(Rs+#u6:2)=#S6
ICLASS					Sense	PredNew		Type			s5					Parse			t5													
0	1	0	0	0	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memw(Rs+#u6:2)=Rt
0	1	0	0	0	0	1	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memw(Rs+#u6:2)=Rt
0	1	0	0	0	1	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv) memw(Rs+#u6:2)=Rt
0	1	0	0	0	1	1	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv.new) memw(Rs+#u6:2)=Rt
ICLASS				Amode			Type			UN	x5					Parse			t5													
1	0	1	0	1	0	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memw(Rx++#s4:2)=Rt
1	0	1	0	1	0	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memw(Rx++#s4:2)=Rt
1	0	1	0	1	0	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memw(Rx++#s4:2)=Rt
1	0	1	0	1	0	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memw(Rx++#s4:2)=Rt
ICLASS				Amode			Type			UN						Parse			t5													
1	0	1	0	1	1	1	1	1	0	0	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memw(#u6)=Rt
1	0	1	0	1	1	1	1	1	0	0	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memw(#u6)=Rt
1	0	1	0	1	1	1	1	1	0	0	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memw(#u6)=Rt
1	0	1	0	1	1	1	1	1	0	0	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memw(#u6)=Rt

Field name	Description
ICLASS	Instruction class
Type	Type
PredNew	PredNew
Sense	Sense
Parse	Packet/loop parse bits
s5	Field to encode register s
t5	Field to encode register t
u5	Field to encode register u
v2	Field to encode register v
x5	Field to encode register x
Amode	Amode
UN	Unsigned

Allocate stack frame

Allocate a stack frame on the call stack. This instruction first pushes LR and FP to the top of stack. It then subtracts an unsigned immediate from SP to allocate room for local variables. FP is set to the address of the old frame pointer on the stack.

The following figure shows the stack layout.



Syntax	Behavior
`allocframe(#u11:3)`	`Assembler mapped to: "allocframe(r29, #u11:3):raw"`

`allocframe(Rx,#u11:3):raw`	`EA = Rx+-8;`
	`*EA = frame_scramble((LR << 32) | FP);`
	`FP=EA;`
	`frame_check_limit(EA-#u);`
	`Rx = EA-#u;`

Class: ST (slots 0)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			U N	x5					Parse																	
1	0	1	0	0	0	0	0	1	0	0	x	x	x	x	x	P	P	0	0	0	i	i	i	i	i	i	i	i	i	i	allocframe(Rx,#u11:3):raw

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
x5	Field to encode register x
Amode	Amode
Type	Type
UN	Unsigned

11.9 System

The system instruction class includes instructions to manage system resources.

11.9.1 System user

The system user instruction subclass includes instructions that allow user access to system resources.

Load locked

This memory lock instruction performs a word or double-word locked load.

This instruction returns the contents of the memory at address Rs and also reserves a lock reservation at that address. For more information, see Atomic operations.

Syntax	Behavior
`Rd=memw_locked(Rs)`	`EA=Rs;` `Rd = *EA;`
`Rdd=memd_locked(Rs)`	`EA=Rs;` `Rdd = *EA;`

Class: SYSTEM (slots 0)

Notes

- This instruction can only be grouped with ALU32 or non-floating-point XTYPE instructions.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	s5					Parse											d5					
1	0	0	1	0	0	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	0	-	-	-	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=memw_locked(Rs)
1	0	0	1	0	0	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	1	0	-	-	-	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=memd_locked(Rs)

Field name	Description
ICLASS	Instruction class
Amode	Amode
Type	Type
UN	Unsigned
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s

Store conditional

This memory lock instruction performs a word or double-word conditional store operation.

If the address reservation is held by this thread and there are no intervening accesses to the memory location, the store is performed and the predicate is set to true. Otherwise, the store is not performed and the predicate returns false. For more information, see Atomic operations.

Syntax	Behavior

`memd_locked(Rs,Pd)=Rtt`	`EA=Rs;`
	`if (lock_valid) {`
	`  *EA = Rtt;`
	`  Pd = 0xff;`
	`  lock_valid = 0;`
	`} else {`
	`  Pd = 0;`
	`}`

`memw_locked(Rs,Pd)=Rt`	`EA=Rs;`
	`if (lock_valid) {`
	`  *EA = Rt;`
	`  Pd = 0xff;`
	`  lock_valid = 0;`
	`} else {`
	`  Pd = 0;`
	`}`

Class: SYSTEM (slots 0)

Notes

- This instruction can only be grouped with ALU32 or non-floating-point XTYPE instructions.
- The predicate generated by this instruction cannot be used as a .new predicate, nor can it be automatically AND'd with another predicate.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	s5					Parse			t5											d2		
1	0	1	0	0	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	-	0	0	**d**	**d**	memw_locked(Rs,Pd)=Rt
1	0	1	0	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	-	-	-	-	0	0	**d**	**d**	memd_locked(Rs,Pd)=Rtt

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d2	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
Amode	Amode
Type	Type
UN	Unsigned

Zero a cache line

The `dczeroa` instruction clears 32 bytes of memory.

If the memory is marked write-back cacheable, a cache line is allocated in the data cache and 32 bytes are cleared.

If the memory is write-through or write-back, 32 bytes of zeros are sent to memory.

This instruction is useful for efficiently handling write-only data by pre-allocating lines in the cache.

The address must be 32-byte aligned. If not, an unaligned error exception is raised.

If this instruction appears in a packet, slot 1 must be A-type or empty.

Syntax	Behavior
`dczeroa(Rs)`	`EA=Rs;` `dcache_zero_addr(EA);`

Class: SYSTEM (slots 0)

Notes

- A packet containing this instruction must have slot 1 either empty or executing an ALU32 instruction.

Intrinsics

`dczeroa(Rs)`	`void Q6_dczeroa_A(Address a)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	s5					Parse																	
1	0	1	0	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	-	-	-	-	-	-	-	-	dczeroa(Rs)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
s5	Field to encode register s
Amode	Amode
Type	Type
UN	Unsigned

Memory barrier

The `barrier` instruction establishes a memory barrier to ensure proper ordering between load/store accesses within a consistency domain before the barrier instruction and accesses after the barrier instruction.

All scalar loads, stores, and cache operation by address within a consistency domain before the barrier are globally observable before any access after the barrier can be observed.

The use of this instruction is system-dependent.

Syntax	Behavior
`barrier`	`memory_barrier;`

Class: SYSTEM (slots 0)

Notes

- This is a solo instruction. It must not be grouped with other instructions in a packet.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN						Parse																	
1	0	1	0	1	0	0	0	0	0	0	-	-	-	-	-	**P**	**P**	-	-	-	-	-	-	0	0	0	-	-	-	-	-	barrier

Field name	Description
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`Amode`	Amode
`Type`	Type
`UN`	Unsigned

Breakpoint

The `brkpt` operation causes the program to enter Debug mode if enabled by ISDB.

Execution control is handed to ISDB and the program does not proceed until directed by the debugger.

If ISDB is disabled, this instruction is treated as a NOP.

Syntax	Behavior
`brkpt`	`Enter Debug mode;`

Class: SYSTEM (slot 3)

Notes

- This is a solo instruction. It must not be grouped with other instructions in a packet.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					sm											Parse																
0	1	1	0	1	1	0	0	0	0	1	-	-	-	-	-	**P**	**P**	-	-	-	-	-	-	0	0	0	-	-	-	-	-	brkpt

Field name	Description
`sm`	Supervisor mode only
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits

Data cache prefetch

The `dcfetch` instruction prefetches the data at address Rs + unsigned immediate.

This instruction is a hint to the memory system, and is handled in an implementation-dependent manner.

Syntax	Behavior
`dcfetch(Rs)`	`Assembler mapped to: "dcfetch(Rs+#0)"`
`dcfetch(Rs+#u11:3)`	`EA=Rs+#u;` `dcache_fetch(EA);`

Class: SYSTEM (slots 0)

Intrinsics

`dcfetch(Rs)`	`void Q6_dcfetch_A(Address a)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	s5					Parse																	
1	0	0	1	0	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	-	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	dcfetch(Rs+#u11:3)

Field name	Description
`ICLASS`	Instruction class
`Amode`	Amode
`Type`	Type
`UN`	Unsigned
`Parse`	Packet/loop parse bits
`s5`	Field to encode register s

Data cache multi prefetch

The `dcfetch_multi`instruction prefetches the data at address Rs + unsigned immediate and also triggers multiple prefetches in the background.

This instruction is a hint to the memory system, and is handled in an implementation-dependent manner.

Syntax	Behavior
`dcfetch_multi(Rs, Rt)`	

Class: ST (slot 0)

Data cache maintenance user operations

Perform maintenance operations on the data cache.

The `dccleaninva` instruction looks up the data cache at address Rs. If this address is in the cache and has dirty data, the data is flushed out to memory and the line is then invalidated.

The `dccleana` instruction looks up the data cache at address Rs. If this address is in the cache and has dirty data, the data is flushed out to memory.

The `dcinva` instruction looks up the data cache at address Rs. If this address is in the cache, the line containing the data is invalidated.

If an instruction appears in a packet, slot 1 must be A-type or empty.

In implementations that support L2 cache, these instructions operate on both L1 data and L2 caches.

Syntax	Behavior
`dccleana(Rs)`	`EA=Rs;` `dcache_clean_addr(EA);`
`dccleaninva(Rs)`	`EA=Rs;` `dcache_cleaninv_addr(EA);`
`dcinva(Rs)`	`EA=Rs;` `dcache_cleaninv_addr(EA);`

Class: SYSTEM (slots 0)

Notes

- A packet containing this instruction must have slot 1 either empty or executing an ALU32 instruction.

Intrinsics

`dccleana(Rs)`	`void Q6_dccleana_A(Address a)`
`dccleaninva(Rs)`	`void Q6_dccleaninva_A(Address a)`
`dcinva(Rs)`	`void Q6_dcinva_A(Address a)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	s5					Parse																	
1	0	1	0	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	dccleana(Rs)
1	0	1	0	0	0	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	dcinva(Rs)
1	0	1	0	0	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	dccleaninva(Rs)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
s5	Field to encode register s
Amode	Amode
Type	Type
UN	Unsigned

Send value to DIAG trace

Send the sources to the external DIAG trace.

Syntax	Behavior
`diag(Rs)`	
`diag0(Rss,Rtt)`	
`diag1(Rss,Rtt)`	

Class: SYSTEM (slot 3)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					sm						s5					Parse																	
0	1	1	0	0	0	1	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	1	-	-	-	-	-	diag(Rs)
ICLASS					sm						s5					Parse			t5													
0	1	1	0	0	0	1	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	-	-	-	-	-	diag0(Rss,Rtt)
0	1	1	0	0	0	1	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	-	-	-	-	-	diag1(Rss,Rtt)

Field name	Description
`sm`	Supervisor mode only
`CLASS`	Instruction class
`Parse`	Packet/loop parse bits
`s5`	Field to encode register s
`t5`	Field to encode register t

Instruction cache maintenance user operations

The `icinva` operation looks up the address specified in Rs in the instruction cache. If a translation for Rs cannot be found, a TLB-miss-X exception with cause code 0x62 is indicated.

If a translation is found but the user does not have proper permissions to the page to invalidate, the instruction is converted to a NOP.

If a translation is found and the user has proper permissions, all ways of all 32 B segments matching Rs[11:5] in any instruction cache are invalidated.

Syntax	Behavior
`icinva(Rs)`	`EA=Rs;` `icache_inv_addr(EA);`

Class: SYSTEM (slot 2)

Notes

- This is a solo instruction. It must not be grouped with other instructions in a packet.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse																
0	1	0	1	0	1	1	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	0	-	-	-	-	-	-	-	-	-	-	-	icinva(Rs)

Field name	Description
`CLASS`	Instruction class
`Parse`	Packet/loop parse bits
`s5`	Field to encode register s

Instruction synchronization

The `isync` operation ensures that all previous instructions have committed before continuing to the next instruction.

This instruction should execute after the following events (when subsequent instructions must observe the results of the event):

- After modifying the TLB with a TLBW instruction
- After modifying the SSR register
- After modifying the SYSCFG register
- After any instruction cache maintenance operation
- After modifying the TID register

Syntax	Behavior
`isync`	`instruction_sync;`

Class: SYSTEM (slot 2)

Notes

- This is a solo instruction. It must not be grouped with other instructions in a packet.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse																
0	1	0	1	0	1	1	1	1	1	0	0	0	0	0	0	**P**	**P**	0	**-**	**-**	**-**	0	0	0	0	0	0	0	0	1	0	isync

Field name	Description
`CLASS`	Instruction class
`Parse`	Packet/loop parse bits

L2 cache prefetch

The `l2fetch` operation initiates background prefetching into the L2 cache.

Rs specifies the 32-bit virtual start address. There are two forms of this instruction.

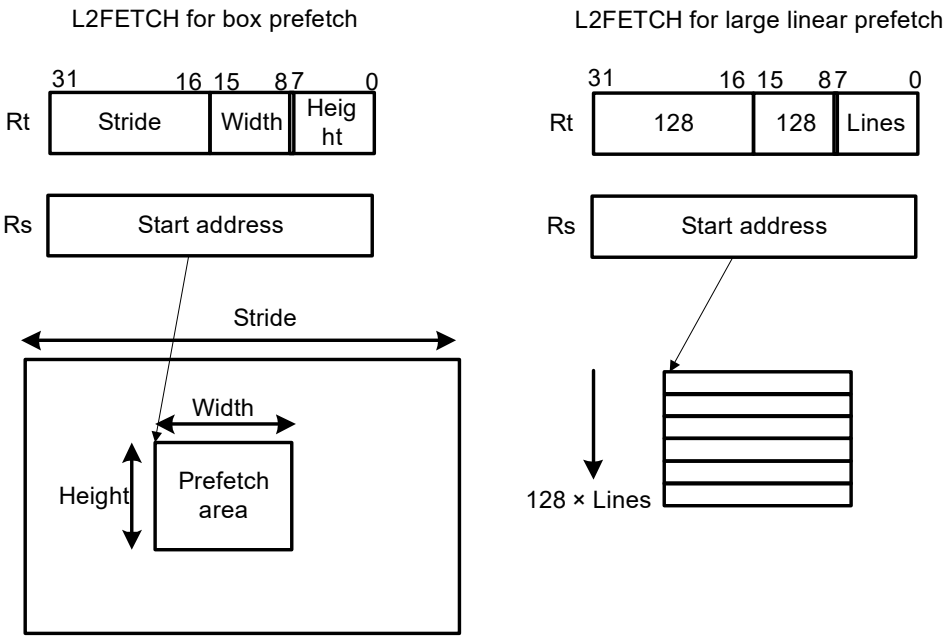
In the first form, the dimensions of the area to prefetch are encoded in source register Rt as follows:

- Rt[15:8] = Width of a fetch block in bytes.
- Rt[7:0] = Height: the number of Width-sized blocks to fetch.
- Rt[31:16] = Stride: an unsigned byte offset that is used to increment the pointer after each Width-sized block is fetched.

In the second form, the operands are encoded in register pair Rtt as follows:

- Rtt[31:16] = Width of a fetch block in bytes.
- Rtt[15:0] = Height: the number of Width-sized blocks to fetch.
- Rtt[47:32] = Stride: an unsigned byte offset that is used to increment the pointer after each Width-sized block is fetched.
- Rtt[48] = Direction. If clear, the prefetches should be performed in row major form, meaning all cache lines in a row should be fetched before proceeding to the next row. If the bit is set, prefetch should be done in column major form, meaning all cache lines in a column are fetched before proceeding to the next column.

The following figure shows two examples of using the L2FETCH instruction.



In the box prefetch, a 2D range of memory is defined within a larger frame. The second example shows prefetch for a large linear area of memory that has size Lines * 128.

`l2fetch` is non-blocking. After the instruction is initiated, the program continues to the next instruction while the prefetching is performed in the background. L2fetch can be used to bring in either code or data to the L2 cache. If the lines of interest are already in the L2, no action is performed. If the lines are missing from the L2$, the hardware attempts to fetch them from the system memory.

The hardware prefetch engine continues to request all lines in the programmed memory range. The prefetching hardware makes a best-effort to prefetch the requested data, and attempts to perform prefetching at a lower priority than demand fetches. This prevents prefetch from adding traffic while the system is under heavy load.

If a program initiates a new `l2fetch` while an older `l2fetch` operation is still pending, the new request is queued, up to three deep. If three `l2fetch` operations are already pending, the oldest request is dropped. During the time a L2 prefetch is active for a thread, the USR:PFA status bit is set to indicate that prefetches are in progress. This bit can be used by the programmer to decide whether or not to start a new L2FETCH before the previous one completes.

Executing an L2fetch with any subfield programmed as zero cancels all pending prefetches by the calling thread.

The implementation is free to drop prefetches when needed.

Syntax	Behavior
`l2fetch(Rs,Rt)`	`l2fetch(Rs,INFO);`
`l2fetch(Rs,Rtt)`	`l2fetch(Rs,INFO);`

Class: SYSTEM (slots 0)

Notes

- This instruction may only be grouped with ALU32 or non-floating-point XTYPE instructions.

Intrinsics

`l2fetch(Rs,Rt)`	`void Q6_l2fetch_AR(Address a, Word32 Rt)`
`l2fetch(Rs,Rtt)`	`void Q6_l2fetch_AP(Address a, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	s5					Parse			t5														
1	0	1	0	0	1	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	-	-	-	-	-	l2fetch(Rs,Rt)
1	0	1	0	0	1	1	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	-	-	-	-	-	l2fetch(Rs,Rtt)

Field name	Description
CLASS	Instruction class
Parse	Packet/loop parse bits
s5	Field to encode register s
t5	Field to encode register t
Amode	Amode
Type	Type
UN	Unsigned

Pause

The `pause` instruction pauses execution for a specified period of time.

During the pause duration, the program enters a low-power state and does not fetch and execute instructions. The instruction provides a short immediate that indicates the pause duration. The program pauses for at most the number of cycles specified in the immediate plus 8. The minimum pause is 0 cycles, and the maximum pause is implementation defined.

An interrupt to the program exits the paused state.

System events, such as hardware or DMA completion, can trigger exits from Pause mode.

An implementation is free to pause for durations shorter than (immediate + 8), but not longer.

This instruction is useful for implementing user-level low-power synchronization operations, such as spin locks or wait-for-event signaling.

Syntax	Behavior
`pause(#u10)`	`Pause for #u cycles;`

Class: SYSTEM (slot 2)

Notes

- This is a solo instruction. It must not be grouped with other instructions in a packet.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse																
0	1	0	1	0	1	0	0	0	1	-	-	-	-	i	i	P	P	-	i	i	i	i	i	-	-	-	i	i	i	-	-	pause(#u10)

Field name	Description
`CLASS`	Instruction class
`Parse`	Packet/loop parse bits

Memory thread synchronization

The `syncht` instruction synchronizes memory.

All outstanding memory operations, including cached and uncached loads and stores, complete before the processor continues to the next instruction. This ensures that certain memory operations are performed in the desired order (for example, when accessing IO devices).

After performing a `syncht` operation, the processor ceases fetching and executing instructions from the program until all outstanding memory operations of that program are complete.

In multi-threaded or multi-core environments, `syncht` is not concerned with other execution contexts.

The use of this instruction is system-dependent.

Syntax	Behavior
`Rd=dmsyncht`	`Rd = DM0;`
`syncht`	`memory_synch;`

Class: SYSTEM (slots 0)

Notes

- This is a solo instruction. It must not be grouped with other instructions in a packet.
- This is a monitor-level feature. If performed in User or Guest mode, a privilege error exception occurs.

Intrinsics

`Rd=dmsyncht`	`Word32 Q6_R_dmsyncht()`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN						Parse											d5					
1	0	1	0	1	0	0	0	0	0	0	-	-	-	-	-	**P**	**P**	-	-	-	-	-	0	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=dmsyncht
ICLASS				Amode			Type			UN						Parse																
1	0	1	0	1	0	0	0	0	1	0	-	-	-	-	-	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	syncht

Field name	Description
`CLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`Amode`	Amode
`Type`	Type
`UN`	Unsigned

Send value to ETM trace

The `trace` operation takes the value of register Rs and emits it to the ETM trace.

The ETM block must be enabled, and the thread must have permissions to perform tracing. The contents of Rs are user-defined.

Syntax	Behavior
`trace(Rs)`	`Send value to ETM trace;`

Class: SYSTEM (slot 3)

Notes

- This instruction may only be grouped with ALU32 or non-floating-point XTYPE instructions.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					sm						s5					Parse																	
0	1	1	0	0	0	1	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	0	-	-	-	-	-	trace(Rs)

Field name	Description
`sm`	Supervisor mode only
`CLASS`	Instruction class
`Parse`	Packet/loop parse bits
`s5`	Field to encode register s

Trap

Trap causes a precise exception.

Executing a trap instruction sets the EX bit in SSR to 1, which disables interrupts and enables Supervisor mode. The program then jumps to the vector location (either `trap0` or `trap1`. The instruction specifies an 8-bit immediate field. This field is copied into the system status register cause field.

Upon returning from the service routine with a RTE, execution resumes at the packet after the trap instruction.

These instructions are generally intended for user code to request services from the operating system. Two trap instructions are provided so the OS can optimize for fast service routines and slower service routines.

Syntax	Behavior
`trap0(#u8)`	`SSR.CAUSE = #u;` `TRAP "0";`
`trap1(#u8)`	`Assembler mapped to: "trap1(R0,#u8)"`

`trap1(Rx,#u8)`	`if (!can_handle_trap1_virtinsn(#u)) {`
	`  SSR.CAUSE = #u;`
	`  TRAP "1";`
	`} else if (#u == 1) {`
	`  VMRTE;`
	`} else if (#u == 3) {`
	`  VMSETIE;`
	`} else if (#u == 4) {`
	`  VMGETIE;`
	`} else if (#u == 6) {`
	`  VMSPSWAP;`

Class: SYSTEM (slot 2)

Notes

- This is a solo instruction. It must not be grouped with other instructions in a packet.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse																
0	1	0	1	0	1	0	0	0	0	-	-	-	-	-	-	P	P	-	i	i	i	i	i	-	-	-	i	i	i	-	-	trap0(#u8)
ICLASS											x5					Parse																
0	1	0	1	0	1	0	0	1	0	-	x	x	x	x	x	P	P	-	i	i	i	i	i	-	-	-	i	i	i	-	-	trap1(Rx,#u8)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
x5	Field to encode register x

Unpause

The `unpause` instruction resumes all threads whose execution has been stalled with a pause instruction.

Syntax	Behavior
`unpause`	`Unpause threads in pause state;`

Class: SYSTEM (slot 2)

Notes

- This is a solo instruction. It must not be grouped with other instructions in a packet.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse																
0	1	0	1	0	1	1	1	1	1	1	-	-	-	-	-	P	P	0	1	-	-	-	-	0	0	0	-	-	-	-	-	unpause

Field name	Description
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits

11.10 XTYPE

The XTYPE instruction class includes instructions that perform most of the data processing performed by the Hexagon processor.

XTYPE instructions are executable on slot 2 or slot 3.

11.10.1 XTYPE ALU

The XTYPE ALU instruction subclass includes instructions that perform arithmetic and logical operations.

Absolute value doubleword

Take the absolute value of the 64-bit source register and place it in the destination register.

Syntax	Behavior
`Rdd=abs(Rss)`	`Rdd = ABS(Rss);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=abs(Rss)`	`Word64 Q6_P_abs_P(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	0	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=abs(Rss)

Field name	Description
`CLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major opcode
`MinOp`	Minor opcode
`RegType`	Register type

Absolute value word

Take the absolute value of the source register and place it in the destination register.

The 32-bit absolute value is available with optional saturation. The single case of saturation is when the source register is equal to 0x8000_0000, the destination saturates to 0x7fff_ffff.

Syntax	Behavior
`Rd=abs(Rs)[:sat]`	`Rd = [sat32](ABS(sxt32->64(Rs)));`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=abs(Rs)`	`Word32 Q6_R_abs_R(Word32 Rs)`
`Rd=abs(Rs):sat`	`Word32 Q6_R_abs_R_sat(Word32 Rs)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	1	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=abs(Rs)
1	0	0	0	1	1	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	1	**d**	**d**	**d**	**d**	**d**	Rd=abs(Rs):sat

Field name	Description
CLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

Add and accumulate

Add Rs and Rt or a signed immediate, then add or subtract the resulting value. The result is saved in Rx.

Syntax	Behavior
`Rd = add(Rs,add(Ru,#s6))`	`Rd = Rs + Ru + apply_extension(#s);`
`Rd = add(Rs,sub(#s6,Ru))`	`Rd = Rs - Ru + apply_extension(#s);`
`Rx += add(Rs,#s8)`	`apply_extension(#s);` `Rx = Rx + Rs + #s;`
`Rx += add(Rs,Rt)`	`Rx = Rx + Rs + Rt;`
`Rx -= add(Rs,#s8)`	`apply_extension(#s);` `Rx = Rx - (Rs + #s);`
`Rx -= add(Rs,Rt)`	`Rx = R x - (Rs + Rt);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=add(Rs,add(Ru,#s6))`	`Word32 Q6_R_add_add_RRI(Word32 Rs, Word32 Ru, Word32 Is6)`
`Rd=add(Rs,sub(#s6,Ru))`	`Word32 Q6_R_add_sub_RIR(Word32 Rs, Word32 Is6, Word32 Ru)`
`Rx+=add(Rs,#s8)`	`Word32 Q6_R_addacc_RI(Word32 Rx, Word32 Rs, Word32 Is8)`
`Rx+=add(Rs,Rt)`	`Word32 Q6_R_addacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx-=add(Rs,#s8)`	`Word32 Q6_R_addnac_RI(Word32 Rx, Word32 Rs, Word32 Is8)`
`Rx-=add(Rs,Rt)`	`Word32 Q6_R_addnac_RR(Word32 Rx, Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			d5								u5					
1	1	0	1	1	0	1	1	0	i	i	s	s	s	s	s	P	P	i	d	d	d	d	d	i	i	i	u	u	u	u	u	Rd=add(Rs,add(Ru,#s6))
1	1	0	1	1	0	1	1	1	i	i	s	s	s	s	s	P	P	i	d	d	d	d	d	i	i	i	u	u	u	u	u	Rd=add(Rs,sub(#s6,Ru))
ICLASS				RegType				MajOp			s5					Parse								MinOp			x5					
1	1	1	0	0	0	1	0	0	-	-	s	s	s	s	s	P	P	0	i	i	i	i	i	i	i	i	x	x	x	x	x	Rx+=add(Rs,#s8)
1	1	1	0	0	0	1	0	1	-	-	s	s	s	s	s	P	P	0	i	i	i	i	i	i	i	i	x	x	x	x	x	Rx-=add(Rs,#s8)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	1	1	1	0	0	0	s	s	s	s	s	P	P	0	t	t	t	t	t	0	0	1	x	x	x	x	x	Rx+=add(Rs,Rt)
1	1	1	0	1	1	1	1	1	0	0	s	s	s	s	s	P	P	0	t	t	t	t	t	0	0	1	x	x	x	x	x	Rx-=add(Rs,Rt)

Field name	Description
RegType	Register type
CLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
u5	Field to encode register u
x5	Field to encode register x

Add doublewords

The first form of this instruction adds two 32-bit registers. If the result overflows 32 bits, the result is saturated to 0x7FFF_FFFF for a positive result, or 0x8000_0000 for a negative result. A 32-bit non-saturating register add is a ALU32-class instruction and can execute on any slot.

The second instruction form sign-extends a 32-bit register Rt to 64-bits and performs a 64-bit add with Rss. The result is stored in Rdd.

The third instruction form adds 64-bit registers Rss and Rtt and places the result in Rdd.

The final instruction form adds two 64-bit registers Rss and Rtt. If the result overflows 64 bits, it is saturated to 0x7fff_ffff_ffff_ffff for a positive result, or 0x8000_0000_0000_0000 for a negative result.

Syntax	Behavior
`Rd=add(Rs,Rt):sat:deprecated`	`Rd = sat32(Rs+Rt);`

`Rdd=add(Rs,Rtt)`	`if ("Rs & 1") {`
	`  Assembler mapped to: "Rdd = add(Rss, Rtt):raw:hi";`
	`} else {`
	`  Assembler mapped to: "Rdd = add(Rss, Rtt):raw:lo";`
	`}`

`Rdd=add(Rss,Rtt)`	`Rdd=Rss+Rtt;`
`Rdd=add(Rss,Rtt):raw:hi`	`Rdd=Rtt+sxt32->64(Rss.w[1]);`
`Rdd=add(Rss,Rtt):raw:lo`	`Rdd=Rtt+sxt32->64(Rss.w[0]);`
`Rdd=add(Rss,Rtt):sat`	`Rdd=sat64(Rss+Rtt);`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=add(Rs,Rtt)`	`Word64 Q6_P_add_RP(Word32 Rs, Word64 Rtt)`
`Rdd=add(Rss,Rtt)`	`Word64 Q6_P_add_PP(Word64 Rss, Word64 Rtt)`
`Rdd=add(Rss,Rtt):sat`	`Word64 Q6_P_add_PP_sat(Word64 Rss, Word64 Rtt)`

Encoding

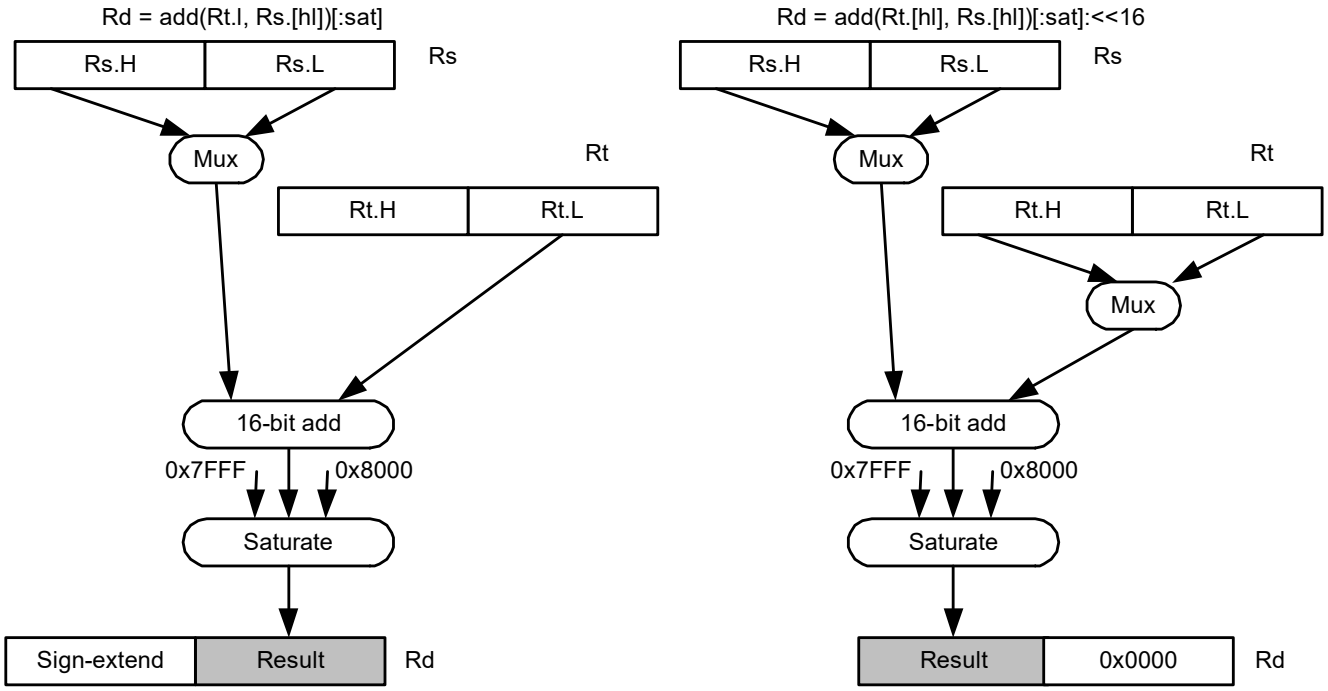
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5						
1	1	0	1	0	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=add(Rss,Rtt)
1	1	0	1	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=add(Rss,Rtt):sat
1	1	0	1	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=add(Rss,Rtt):raw:lo
1	1	0	1	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=add(Rss,Rtt):raw:hi
1	1	0	1	0	1	0	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	-	-	**d**	**d**	**d**	**d**	**d**	Rd=add(Rs,Rt):sat:deprecated

Field name	Description
RegType	Register type
MinOp	Minor opcode
CLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Add halfword

Perform a 16-bit add with optional saturation, and place the result in either the upper or lower half of a register. If the result goes in the upper half, the sources can be any high or low halfword of Rs and Rt. The lower 16 bits of the result are zeroed.

If the result is to be placed in the lower 16 bits of Rd, the Rs source can be either high or low, but the other source must be the low halfword of Rt. In this case, the upper halfword of Rd is the sign-extension of the low halfword.



Syntax	Behavior
`Rd = add(Rt.L,Rs.[HL])[:sat]`	`Rd = [sat16](Rt.h[0]+Rs.h[01]);`
`Rd = add(Rt.[HL], Rs.[HL])[:sat]: <<16`	`Rd = ([sat16](Rt.h[01]+Rs.h[01]))<<16;`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=add(Rt.H,Rs.H):<<16`	`Word32 Q6_R_add_RhRh_s16(Word32 Rt, Word32 Rs)`

`Rd=add(Rt.H,Rs.H):sat:<<16`	`Word32 Q6_R_add_RhRh_sat_s16(Word32 Rt, Word32 Rs)`
`Rd=add(Rt.H,Rs.L):<<16`	`Word32 Q6_R_add_RhRl_s16(Word32 Rt, Word32 Rs)`
`Rd=add(Rt.H,Rs.L):sat:<<16`	`Word32 Q6_R_add_RhRl_sat_s16(Word32 Rt, Word32 Rs)`
`Rd=add(Rt.L,Rs.H)`	`Word32 Q6_R_add_RlRh(Word32 Rt, Word32 Rs)`
`Rd=add(Rt.L,Rs.H):<<16`	`Word32 Q6_R_add_RlRh_s16(Word32 Rt, Word32 Rs)`
`Rd=add(Rt.L,Rs.H):sat`	`Word32 Q6_R_add_RlRh_sat(Word32 Rt, Word32 Rs)`
`Rd=add(Rt.L,Rs.H):sat:<<16`	`Word32 Q6_R_add_RlRh_sat_s16(Word32 Rt, Word32 Rs)`
`Rd=add(Rt.L,Rs.L)`	`Word32 Q6_R_add_RlRl(Word32 Rt, Word32 Rs)`
`Rd=add(Rt.L,Rs.L):<<16`	`Word32 Q6_R_add_RlRl_s16(Word32 Rt, Word32 Rs)`
`Rd=add(Rt.L,Rs.L):sat`	`Word32 Q6_R_add_RlRl_sat(Word32 Rt, Word32 Rs)`
`Rd=add(Rt.L,Rs.L):sat:<<16`	`Word32 Q6_R_add_RlRl_sat_s16(Word32 Rt, Word32 Rs)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.L,Rs.L)
1	1	0	1	0	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.L,Rs.H)
1	1	0	1	0	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.L,Rs.L):sat
1	1	0	1	0	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.L,Rs.H):sat
1	1	0	1	0	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.L,Rs.L):<<16
1	1	0	1	0	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.L,Rs.H):<<16
1	1	0	1	0	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.H,Rs.L):<<16
1	1	0	1	0	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.H,Rs.H):<<16
1	1	0	1	0	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.L,Rs.L):sat:<<16
1	1	0	1	0	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.L,Rs.H):sat:<<16
1	1	0	1	0	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.H,Rs.L):sat:<<16
1	1	0	1	0	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.H,Rs.H):sat:<<16

Field name	Description
`RegType`	Register type
`MinOp`	Minor opcode
`CLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Add or subtract doublewords with carry

Add or subtract with carry. Predicate register Px is used as an extra input and output.

For adds, the LSB of the predicate is added to the sum of the two input pairs.

For subtracts, the predicate is considered a not-borrow. The LSB of the predicate is added to the first source register and the logical complement of the second argument.

The carry-out from the sum is saved in predicate Px.

These instructions allow efficient addition or subtraction of numbers larger than 64 bits.

Syntax	Behavior

`Rdd=add(Rss,Rtt,Px):carry`	`PREDUSE_TIMING;`
	`Rdd = Rss + Rtt + Px[0];`
	`Px = carry_from_add(Rss, Rtt,Px[0]) ? 0xff : 0x00;`

`Rdd=sub(Rss,Rtt,Px):carry`	`PREDUSE_TIMING;`
	`Rdd = Rss + ~Rtt + Px[0];`
	`Px = carry_from_add(Rss, ~Rtt, Px[0]) ? 0xff : 0x00;`

Class: XTYPE (slots 2,3)

Notes

- The predicate generated by this instruction cannot be used as a .new predicate, nor can it be automatically AND'd with another predicate.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5						x2		d5						
1	1	0	0	0	0	1	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	**x**	**x**	**d**	**d**	**d**	**d**	**d**	Rdd=add(Rss,Rtt,Px):carry
1	1	0	0	0	0	1	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	**x**	**x**	**d**	**d**	**d**	**d**	**d**	Rdd=sub(Rss,Rtt,Px):carry

Field name	Description
CLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x2	Field to encode register x
Maj	Major opcode
Min	Minor opcode
RegType	Register type

Clip to unsigned

Clip input to unsigned integer.

Syntax	Behavior
`Rd = clip(Rs,#u5)`	`Rd=MIN((1<<#u)-1,MAX(Rs,-(1<<#u)));`

Class: XTYPE (slots 2,3)

Notes

- This instruction can only execute on a core with the Hexagon audio extensions

Intrinsics

`Rd=clip(Rs,#u5)`	`Word32 Q6_R_clip_RI(Word32 Rs, Word32 Iu5)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	1	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rd=clip(Rs,#u5)

Field name	Description
CLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

Logical doublewords

Perform bitwise logical AND, OR, XOR, and NOT operations. The source and destination registers are 64-bit.

For 32-bit logical operations, see the ALU32 logical instructions.

Syntax	Behavior
`Rdd=and(Rss,Rtt)`	`Rdd = Rss&Rtt;`
`Rdd=and(Rtt,~Rss)`	`Rdd = (Rtt & ~Rss);`
`Rdd=not(Rss)`	`Rdd = ~Rss;`
`Rdd=or(Rss,Rtt)`	`Rdd = Rss|Rtt;`
`Rdd=or(Rtt,~Rss)`	`Rdd = (Rtt | ~Rss);`
`Rdd=xor(Rss,Rtt)`	`Rdd = Rss^Rtt;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=and(Rss,Rtt)`	`Word64 Q6_P_and_PP(Word64 Rss, Word64 Rtt)`
`Rdd=and(Rtt,~Rss)`	`Word64 Q6_P_and_PnP(Word64 Rtt, Word64 Rss)`
`Rdd=not(Rss)`	`Word64 Q6_P_not_P(Word64 Rss)`
`Rdd=or(Rss,Rtt)`	`Word64 Q6_P_or_PP(Word64 Rss, Word64 Rtt)`
`Rdd=or(Rtt,~Rss)`	`Word64 Q6_P_or_PnP(Word64 Rtt, Word64 Rss)`
`Rdd=xor(Rss,Rtt)`	`Word64 Q6_P_xor_PP(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=not(Rss)
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=and(Rss,Rtt)
1	1	0	1	0	0	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=and(Rtt,~Rss)
1	1	0	1	0	0	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=or(Rss,Rtt)
1	1	0	1	0	0	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=or(Rtt,~Rss)
1	1	0	1	0	0	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=xor(Rss,Rtt)

Field name	Description
`RegType`	Register type
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`MajOp`	Major opcode
`MinOp`	Minor opcode

Logical-logical doublewords

Perform a logical operation of the two source operands, perform a second logical operation of the result with the destination register Rxx.

The source and destination registers are 64-bit.

Syntax	Behavior
`Rxx^=xor(Rss,Rtt)`	`Rxx^=Rss^Rtt;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rxx^=xor(Rss,Rtt)`	`Word64 Q6_P_xorxacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			x5					
1	1	0	0	1	0	1	0	1	0	-	s	s	s	s	s	P	P	0	t	t	t	t	t	0	0	0	x	x	x	x	x	Rxx^=xor(Rss,Rtt)

Field name	Description
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`s5`	Field to encode register s
`t5`	Field to encode register t
`x5`	Field to encode register x
`Maj`	Major opcode
`Min`	Minor opcode
`RegType`	Register type

Logical-logical words

Perform a logical operation of the two source operands, then perform a second logical operation of the result with the destination register Rx.

The source and destination registers are 32-bit.

Syntax	Behavior
`Rx=or(Ru,and(Rx,#s10))`	`Rx = Ru | (Rx & apply_extension(#s));`
`Rx[&|^]=and(Rs,Rt)`	`Rx [|&^]= (Rs [|&^] Rt);`
`Rx[&|^]=and(Rs,~Rt)`	`Rx [|&^]= (Rs [|&^] ~Rt);`
`Rx[&|^]=or(Rs,Rt)`	`Rx [|&^]= (Rs [|&^] Rt);`
`Rx[&|^]=xor(Rs,Rt)`	`Rx[|&^]=Rs[|&^]Rt;`
`Rx|=and(Rs,#s10)`	`Rx = Rx | (Rs & apply_extension(#s));`
`Rx|=or(Rs,#s10)`	`Rx = Rx | (Rs | apply_extension(#s));`

Class: XTYPE (slots 2,3)

Intrinsics

`Rx&=and(Rs,Rt)`	`Word32 Q6_R_andand_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx&=and(Rs,~Rt)`	`Word32 Q6_R_andand_RnR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx&=or(Rs,Rt)`	`Word32 Q6_R_orand_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx&=xor(Rs,Rt)`	`Word32 Q6_R_xorand_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx=or(Ru,and(Rx,#s10))`	`Word32 Q6_R_or_and_RRI(Word32 Ru, Word32 Rx, Word32 Is10)`
`Rx^=and(Rs,Rt)`	`Word32 Q6_R_andxacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx^=and(Rs,~Rt)`	`Word32 Q6_R_andxacc_RnR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx^=or(Rs,Rt)`	`Word32 Q6_R_orxacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx^=xor(Rs,Rt)`	`Word32 Q6_R_xorxacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx|=and(Rs,#s10)`	`Word32 Q6_R_andor_RI(Word32 Rx, Word32 Rs, Word32 Is10)`
`Rx|=and(Rs,Rt)`	`Word32 Q6_R_andor_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx|=and(Rs,~Rt)`	`Word32 Q6_R_andor_RnR(Word32 Rx, Word32 Rs, Word32 Rt)`

`Rx|=or(Rs,#s10)`	`Word32 Q6_R_oror_RI(Word32 Rx, Word32 Rs, Word32 Is10)`
`Rx|=or(Rs,Rt)`	`Word32 Q6_R_oror_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx|=xor(Rs,Rt)`	`Word32 Q6_R_xoror_RR(Word32 Rx, Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse											x5					
1	1	0	1	1	0	1	0	0	0	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**x**	**x**	**x**	**x**	**x**	Rx|=and(Rs,#s10)
ICLASS				RegType							x5					Parse											u5					
1	1	0	1	1	0	1	0	0	1	**i**	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**u**	**u**	**u**	**u**	**u**	Rx=or(Ru,and(Rx,#s10))
ICLASS				RegType							s5					Parse											x5					
1	1	0	1	1	0	1	0	1	0	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**x**	**x**	**x**	**x**	**x**	Rx|=or(Rs,#s10)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	1	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rx|=and(Rs,~Rt)
1	1	1	0	1	1	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rx&=and(Rs,~Rt)
1	1	1	0	1	1	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rx^=and(Rs,~Rt)
1	1	1	0	1	1	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rx&=and(Rs,Rt)
1	1	1	0	1	1	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rx&=or(Rs,Rt)
1	1	1	0	1	1	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rx&=xor(Rs,Rt)
1	1	1	0	1	1	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rx|=and(Rs,Rt)
1	1	1	0	1	1	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rx^=xor(Rs,Rt)
1	1	1	0	1	1	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rx|=or(Rs,Rt)
1	1	1	0	1	1	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rx|=xor(Rs,Rt)
1	1	1	0	1	1	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rx^=and(Rs,Rt)
1	1	1	0	1	1	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rx^=or(Rs,Rt)

Field name	Description
`RegType`	Register type
`ICLASS`	Instruction class
`MajOp`	Major opcode
`MinOp`	Minor opcode
`Parse`	Packet/loop parse bits
`s5`	Field to encode register s
`t5`	Field to encode register t
`u5`	Field to encode register u
`x5`	Field to encode register x

Maximum words

Select either the signed or unsigned maximum of two source registers and place in a destination register Rdd.

Syntax	Behavior
`Rd=max(Rs,Rt)`	`Rd = max(Rs,Rt);`
`Rd=maxu(Rs,Rt)`	`Rd = max(Rs.uw[0],Rt.uw[0]);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=max(Rs,Rt)`	`Word32 Q6_R_max_RR(Word32 Rs, Word32 Rt)`
`Rd=maxu(Rs,Rt)`	`UWord32 Q6_R_maxu_RR(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	1	0	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	-	-	**d**	**d**	**d**	**d**	**d**	Rd=max(Rs,Rt)
1	1	0	1	0	1	0	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	-	-	**d**	**d**	**d**	**d**	**d**	Rd=maxu(Rs,Rt)

Field name	Description
`RegType`	Register type
`MinOp`	Minor opcode
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Maximum doublewords

Select either the signed or unsigned maximum of two 64-bit source registers and place in a destination register.

Syntax	Behavior
`Rdd=max(Rss,Rtt)`	`Rdd = max(Rss,Rtt);`
`Rdd=maxu(Rss,Rtt)`	`Rdd = max(Rss.u64,Rtt.u64);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=max(Rss,Rtt)`	`Word64 Q6_P_max_PP(Word64 Rss, Word64 Rtt)`
`Rdd=maxu(Rss,Rtt)`	`UWord64 Q6_P_maxu_PP(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=max(Rss,Rtt)
1	1	0	1	0	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=maxu(Rss,Rtt)

Field name	Description
`RegType`	Register type
`MinOp`	Minor opcode
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Minimum words

Select either the signed or unsigned minimum of two source registers and place in destination register Rd.

Syntax	Behavior
`Rd=min(Rt,Rs)`	`Rd = min(Rt,Rs);`
`Rd=minu(Rt,Rs)`	`Rd = min(Rt.uw[0],Rs.uw[0]);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=min(Rt,Rs)`	`Word32 Q6_R_min_RR(Word32 Rt, Word32 Rs)`
`Rd=minu(Rt,Rs)`	`UWord32 Q6_R_minu_RR(Word32 Rt, Word32 Rs)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	1	0	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	-	-	**d**	**d**	**d**	**d**	**d**	Rd=min(Rt,Rs)
1	1	0	1	0	1	0	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	-	-	**d**	**d**	**d**	**d**	**d**	Rd=minu(Rt,Rs)

Field name	Description
`RegType`	Register type
`MinOp`	Minor opcode
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Minimum doublewords

Select either the signed or unsigned minimum of two 64-bit source registers and place in the destination register Rdd.

Syntax	Behavior
`Rdd=min(Rtt,Rss)`	`Rdd = min(Rtt,Rss);`
`Rdd=minu(Rtt,Rss)`	`Rdd = min(Rtt.u64,Rss.u64);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=min(Rtt,Rss)`	`Word64 Q6_P_min_PP(Word64 Rtt, Word64 Rss)`
`Rdd=minu(Rtt,Rss)`	`UWord64 Q6_P_minu_PP(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=min(Rtt,Rss)
1	1	0	1	0	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=minu(Rtt,Rss)

Field name	Description
`RegType`	Register type
`MinOp`	Minor opcode
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Modulo wrap

Wrap the Rs value into the modulo range from 0 to Rt.

If Rs is greater than or equal to Rt, wrap it to the bottom of the range by subtracting Rt.

If Rs is less than zero, wrap it to the top of the range by adding Rt.

Otherwise, when Rs fits within the range, no adjustment is necessary. The result is returned in register Rd.

Syntax	Behavior

`Rd=modwrap(Rs,Rt)`	`if (Rs < 0) {`
	`  Rd = Rs + Rt.uw[0];`
	`} else if (Rs.uw[0] >= Rt.uw[0]) {`
	`  Rd = Rs - Rt.uw[0];`
	`} else {`
	`  Rd = Rs;`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=modwrap(Rs,Rt)`	`Word32 Q6_R_modwrap_RR(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5						MinOp			d5					
1	1	0	1	0	0	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=modwrap(Rs,Rt)

Field name	Description
`RegType`	Register type
`MinOp`	Minor opcode
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Negate

The first form of this instruction performs a negate on a 32-bit register with saturation. If the input is 0x80000000, the result is saturated to 0x7fffffff. Note that the non-saturating 32-bit register negate is a ALU32-class instruction and can be executed on any slot.

The second form of this instruction negates a 64-bit source register and place the result in destination Rdd.

Syntax	Behavior
`Rd=neg(Rs):sat`	`Rd = sat32(-Rs.s64);`
`Rdd=neg(Rss)`	`Rdd = -Rss;`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=neg(Rs):sat`	`Word32 Q6_R_neg_R_sat(Word32 Rs)`
`Rdd=neg(Rss)`	`Word64 Q6_P_neg_P(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=neg(Rss)
1	0	0	0	1	1	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=neg(Rs):sat

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

Round

Perform either arithmetic (.5 is rounded up) or convergent (.5 is rounded towards even) rounding to any bit location.

Arithmetic rounding has optional saturation. In this version, the result is saturated to a 32-bit number after adding the rounding constant. After the rounding and saturation have been performed, the final result is right shifted using a sign-extending shift.

Syntax	Behavior
`Rd=cround(Rs,#u5)`	`Rd = (#u==0)?Rs:convround(Rs,2**(#u-1))>>#u;`
`Rd=cround(Rs,Rt)`	`Rd = (zxt5->32(Rt)==0)?Rs:convround(Rs,2**(zxt5->32(Rt)-1))>>zxt5->32(Rt);`
`Rd=round(Rs,#u5)[:sat]`	`Rd = ([sat32]((#u==0)?(Rs):round(Rs,2**(#u-1))))>>#u;`
`Rd=round(Rs,Rt)[:sat]`	`Rd = ([sat32]((zxt5->32(Rt)==0)?(Rs):round(Rs,2**(zxt5->32(Rt)-1))))>>zxt5->32(Rt);`
`Rd=round(Rss):sat`	`tmp=sat64(Rss+0x080000000ULL);` `Rd = tmp.w[1];`

`Rdd=cround(Rss,#u6)`	`if (#u == 0) {`
	`  Rdd = Rss;`
	`} else if ((Rss & (size8s_t)((1LL << (#u - 1)) - 1LL)) == 0) {`
	`  src_128 = sxt64->128(Rss);`
	`  rndbit_128 = sxt64->128(1LL);`
	`  rndbit_128 = (rndbit_128 << #u);`
	`  rndbit_128 = (rndbit_128 & src_128);`
	`  rndbit_128 = (size8s_t) (rndbit_128 >> 1);`
	`  tmp128 = src_128+rndbit_128;`
	`  tmp128 = (size8s_t) (tmp128 >> #u);`
	`  Rdd = sxt128->64(tmp128);`
	`} else {`
	`  size16s_t rndbit_128 = sxt64->128((1LL << (#u - 1)));`
	`  size16s_t src_128 = sxt64->128(Rss);`
	`  size16s_t tmp128 = src_128+rndbit_128;`
	`  tmp128 = (size8s_t) (tmp128 >> #u);`
	`  Rdd = sxt128->64(tmp128);`
	`};`

Syntax	Behavior

Rdd=cround(Rss,Rt)	if ($zxt_{6->32}$(Rt) == 0) {
	  Rdd = Rss;
	} else if ((Rss & (size8s_t)((1LL << ($zxt_{6->32}$(Rt) - 1)) - 1LL)) == 0) {
	  src_128 = $sxt_{64->128}$(Rss);
	  rndbit_128 = $sxt_{64->128}$(1LL);
	  rndbit_128 = (rndbit_128 << $zxt_{6->32}$(Rt));
	  rndbit_128 = (rndbit_128 & src_128);
	  rndbit_128 = (size8s_t) (rndbit_128 >> 1);
	  tmp128 = src_128+rndbit_128;
	  tmp128 = (size8s_t) (tmp128 >> $zxt_{6->32}$(Rt));
	  Rdd = $sxt_{128->64}$(tmp128);
	} else {
	  size16s_t rndbit_128 = $sxt_{64->128}$((1LL << ($zxt_{6->32}$(Rt) - 1)));
	  size16s_t src_128 = $sxt_{64->128}$(Rss);
	  size16s_t tmp128 = src_128+rndbit_128;
	  tmp128 = (size8s_t) (tmp128 >> $zxt_{6->32}$(Rt));
	  Rdd = $sxt_{128->64}$(tmp128);
	};

Class: XTYPE (slots 2,3)

Notes

- This instruction can only execute on a core with the Hexagon audio extensions
- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

Rd=cround(Rs,#u5)	Word32 Q6_R_cround_RI(Word32 Rs, Word32 Iu5)
Rd=cround(Rs,Rt)	Word32 Q6_R_cround_RR(Word32 Rs, Word32 Rt)
Rd=round(Rs,#u5)	Word32 Q6_R_round_RI(Word32 Rs, Word32 Iu5)
Rd=round(Rs,#u5):sat	Word32 Q6_R_round_RI_sat(Word32 Rs, Word32 Iu5)
Rd=round(Rs,Rt)	Word32 Q6_R_round_RR(Word32 Rs, Word32 Rt)
Rd=round(Rs,Rt):sat	Word32 Q6_R_round_RR_sat(Word32 Rs, Word32 Rt)
Rd=round(Rss):sat	Word32 Q6_R_round_P_sat(Word64 Rss)
Rdd=cround(Rss,#u6)	Word64 Q6_P_cround_PI(Word64 Rss, Word32 Iu6)
Rdd=cround(Rss,Rt)	Word64 Q6_P_cround_PR(Word64 Rss, Word32 Rt)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**-**	**-**	**-**	**-**	**-**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=round(Rss):sat
1	0	0	0	1	1	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	0	**-**	**d**	**d**	**d**	**d**	**d**	Rd=cround(Rs,#u5)
1	0	0	0	1	1	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	1	0	**-**	**d**	**d**	**d**	**d**	**d**	Rd=round(Rs,#u5)
1	0	0	0	1	1	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	1	1	**-**	**d**	**d**	**d**	**d**	**d**	Rd=round(Rs,#u5):sat
1	0	0	0	1	1	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	0	1	**-**	**d**	**d**	**d**	**d**	**d**	Rdd=cround(Rss,#u6)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	0	0	0	1	1	0	1	1	**-**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	0	0	**-**	**d**	**d**	**d**	**d**	**d**	Rd=cround(Rs,Rt)
1	1	0	0	0	1	1	0	1	1	**-**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	0	1	**-**	**d**	**d**	**d**	**d**	**d**	Rdd=cround(Rss,Rt)
1	1	0	0	0	1	1	0	1	1	**-**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	1	0	**-**	**d**	**d**	**d**	**d**	**d**	Rd=round(Rs,Rt)
1	1	0	0	0	1	1	0	1	1	**-**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	1	1	**-**	**d**	**d**	**d**	**d**	**d**	Rd=round(Rs,Rt):sat

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

Subtract doublewords

Subtract the 64-bit register Rss from register Rtt.

Syntax	Behavior
`Rd=sub(Rt,Rs):sat:deprecated`	`Rd=sat32(Rt - Rs);`
`Rdd=sub(Rtt,Rss)`	`Rdd=Rtt-Rss;`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=sub(Rtt,Rss)`	`Word64 Q6_P_sub_PP(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=sub(Rtt,Rss)
1	1	0	1	0	1	0	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	-	-	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt,Rs):sat:deprecated

Field name	Description
`RegType`	Register type
`MinOp`	Minor opcode
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Subtract and accumulate words

Subtract Rs from Rt, then add the resulting value with Rx. The result is saved in Rx.

Syntax	Behavior
`Rx+=sub(Rt,Rs)`	`Rx=Rx + Rt - Rs;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rx+=sub(Rt,Rs)`	`Word32 Q6_R_subacc_RR(Word32 Rx, Word32 Rt, Word32 Rs)`

Encoding

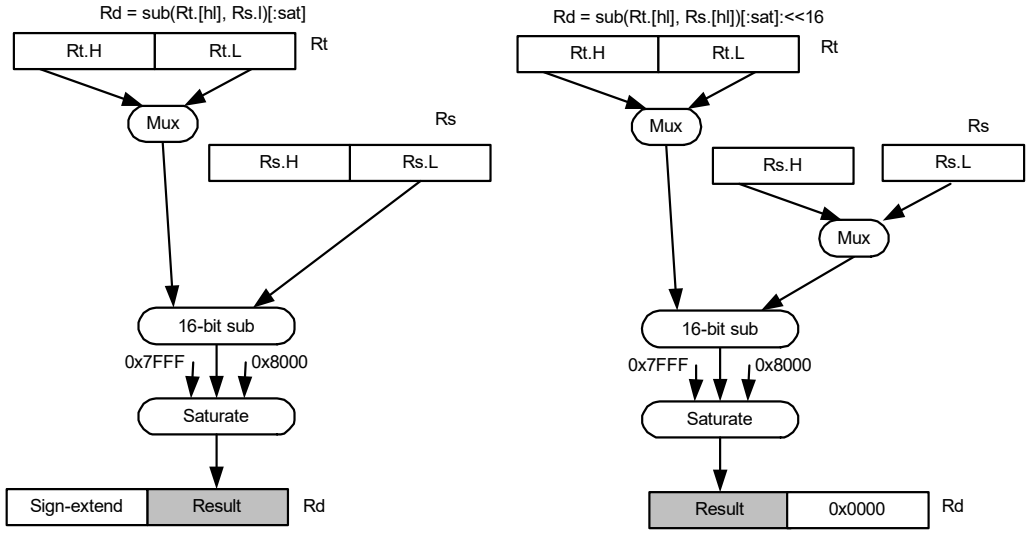
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rx+=sub(Rt,Rs)

Field name	Description
`ICLASS`	Instruction class
`MajOp`	Major opcode
`MinOp`	Minor opcode
`RegType`	Register type
`Parse`	Packet/loop parse bits
`s5`	Field to encode register s
`t5`	Field to encode register t
`x5`	Field to encode register x

Subtract halfword

Perform a 16-bit subtract with optional saturation and place the result in either the upper or lower half of a register. If the result goes in the upper half, the sources can be any high or low halfword of Rs and Rt. The lower 16 bits of the result are zeroed.

If the result is to be placed in the lower 16 bits of Rd, the Rs source can be either high or low, but the other source must be the low halfword of Rt. In this case, the upper halfword of Rd is the sign-extension of the low halfword.



Syntax	Behavior
`Rd=sub(Rt.L,Rs.[HL])[:sat]`	`Rd=[sat16](Rt.h[0]-Rs.h[01]);`
`Rd=sub(Rt.[HL],Rs.[HL])[:sat]:<<16`	`Rd=([sat16](Rt.h[01]-Rs.h[01]))<<16;`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=sub(Rt.H,Rs.H):<<16`	`Word32 Q6_R_sub_RhRh_s16(Word32 Rt, Word32 Rs)`
`Rd=sub(Rt.H,Rs.H):sat:<<16`	`Word32 Q6_R_sub_RhRh_sat_s16(Word32 Rt, Word32 Rs)`
`Rd=sub(Rt.H,Rs.L):<<16`	`Word32 Q6_R_sub_RhRl_s16(Word32 Rt, Word32 Rs)`
`Rd=sub(Rt.H,Rs.L):sat:<<16`	`Word32 Q6_R_sub_RhRl_sat_s16(Word32 Rt, Word32 Rs)`
`Rd=sub(Rt.L,Rs.H)`	`Word32 Q6_R_sub_RlRh(Word32 Rt, Word32 Rs)`

`Rd=sub(Rt.L,Rs.H):<<16`	`Word32 Q6_R_sub_RlRh_s16(Word32 Rt, Word32 Rs)`
`Rd=sub(Rt.L,Rs.H):sat`	`Word32 Q6_R_sub_RlRh_sat(Word32 Rt, Word32 Rs)`
`Rd=sub(Rt.L,Rs.H):sat:<<16`	`Word32 Q6_R_sub_RlRh_sat_s16(Word32 Rt, Word32 Rs)`
`Rd=sub(Rt.L,Rs.L)`	`Word32 Q6_R_sub_RlRl(Word32 Rt, Word32 Rs)`
`Rd=sub(Rt.L,Rs.L):<<16`	`Word32 Q6_R_sub_RlRl_s16(Word32 Rt, Word32 Rs)`
`Rd=sub(Rt.L,Rs.L):sat`	`Word32 Q6_R_sub_RlRl_sat(Word32 Rt, Word32 Rs)`
`Rd=sub(Rt.L,Rs.L):sat:<<16`	`Word32 Q6_R_sub_RlRl_sat_s16(Word32 Rt, Word32 Rs)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	1	0	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt.L,Rs.L)
1	1	0	1	0	1	0	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt.L,Rs.H)
1	1	0	1	0	1	0	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt.L,Rs.L):sat
1	1	0	1	0	1	0	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt.L,Rs.H):sat
1	1	0	1	0	1	0	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt.L,Rs.L):<<16
1	1	0	1	0	1	0	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt.L,Rs.H):<<16
1	1	0	1	0	1	0	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt.H,Rs.L):<<16
1	1	0	1	0	1	0	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt.H,Rs.H):<<16
1	1	0	1	0	1	0	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt.L,Rs.L):sat:<<16
1	1	0	1	0	1	0	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt.L,Rs.H):sat:<<16
1	1	0	1	0	1	0	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt.H,Rs.L):sat:<<16
1	1	0	1	0	1	0	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt.H,Rs.H):sat:<<16

Field name	Description
`RegType`	Register type
`MinOp`	Minor opcode
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Sign extend word to doubleword

Sign-extend a 32-bit word to a 64-bit doubleword.

Syntax	Behavior
`Rdd=sxtw(Rs)`	`Rdd = sxt32->64(Rs);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=sxtw(Rs)`	`Word64 Q6_P_sxtw_R(Word32 Rs)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	1	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=sxtw(Rs)

Field name	Description
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major opcode
`MinOp`	Minor opcode
`RegType`	Register type

Vector absolute value halfwords

Take the absolute value of each of the four halfwords in the 64-bit source vector Rss. Place the result in Rdd.

Saturation is optionally available.

Syntax	Behavior

`Rdd=vabsh(Rss)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=ABS(Rss.h[i]);`
	`}`

`Rdd=vabsh(Rss):sat`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=sat`$_{16}$`(ABS(Rss.h[i]));`
	`}`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vabsh(Rss)`	`Word64 Q6_P_vabsh_P(Word64 Rss)`
`Rdd=vabsh(Rss):sat`	`Word64 Q6_P_vabsh_P_sat(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vabsh(Rss)
1	0	0	0	0	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vabsh(Rss):sat

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

Vector absolute value words

Take the absolute value of each of the two words in the 64-bit source vector Rss. Place the result in Rdd.

Saturation is optionally available.

Syntax	Behavior

`Rdd=vabsw(Rss)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=ABS(Rss.w[i]);`
	`}`

`Rdd=vabsw(Rss):sat`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=sat`$_{32}$`(ABS(Rss.w[i]));`
	`}`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vabsw(Rss)`	`Word64 Q6_P_vabsw_P(Word64 Rss)`
`Rdd=vabsw(Rss):sat`	`Word64 Q6_P_vabsw_P_sat(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vabsw(Rss)
1	0	0	0	0	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vabsw(Rss):sat

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

Vector absolute difference bytes

For each element in the source vector Rss, subtract the corresponding element in source vector Rtt. Take the absolute value of the results, and store into Rdd.

Syntax	Behavior

`Rdd=vabsdiffb(Rtt,Rss)`	`for (i=0;i<8;i++) {`
	`  Rdd.b[i]=ABS(Rtt.b[i] - Rss.b[i]);`
	`}`

`Rdd=vabsdiffub(Rtt,Rss)`	`for (i=0;i<8;i++) {`
	`  Rdd.b[i]=ABS(Rtt.ub[i] - Rss.ub[i]);`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vabsdiffb(Rtt,Rss)`	`Word64 Q6_P_vabsdiffb_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vabsdiffub(Rtt,Rss)`	`Word64 Q6_P_vabsdiffub_PP(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5						MinOp			d5					
1	1	1	0	1	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vabsdiffub(Rtt,Rss)
1	1	1	0	1	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vabsdiffb(Rtt,Rss)

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector absolute difference halfwords

For each element in the source vector Rss, subtract the corresponding element in source vector Rtt. Take the absolute value of the results, and store into Rdd.

Syntax	Behavior

`Rdd=vabsdiffh(Rtt,Rss)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=ABS(Rtt.h[i] - Rss.h[i]);`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vabsdiffh(Rtt,Rss)`	`Word64 Q6_P_vabsdiffh_PP(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vabsdiffh(Rtt,Rss)

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector absolute difference words

For each element in the source vector Rss, subtract the corresponding element in source vector Rtt. Take the absolute value of the results, and store into Rdd.

Syntax	Behavior

`Rdd=vabsdiffw(Rtt,Rss)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=ABS(Rtt.w[i] - Rss.w[i]);`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vabsdiffw(Rtt,Rss)`	`Word64 Q6_P_vabsdiffw_PP(Word64 Rtt, Word64 Rss)`

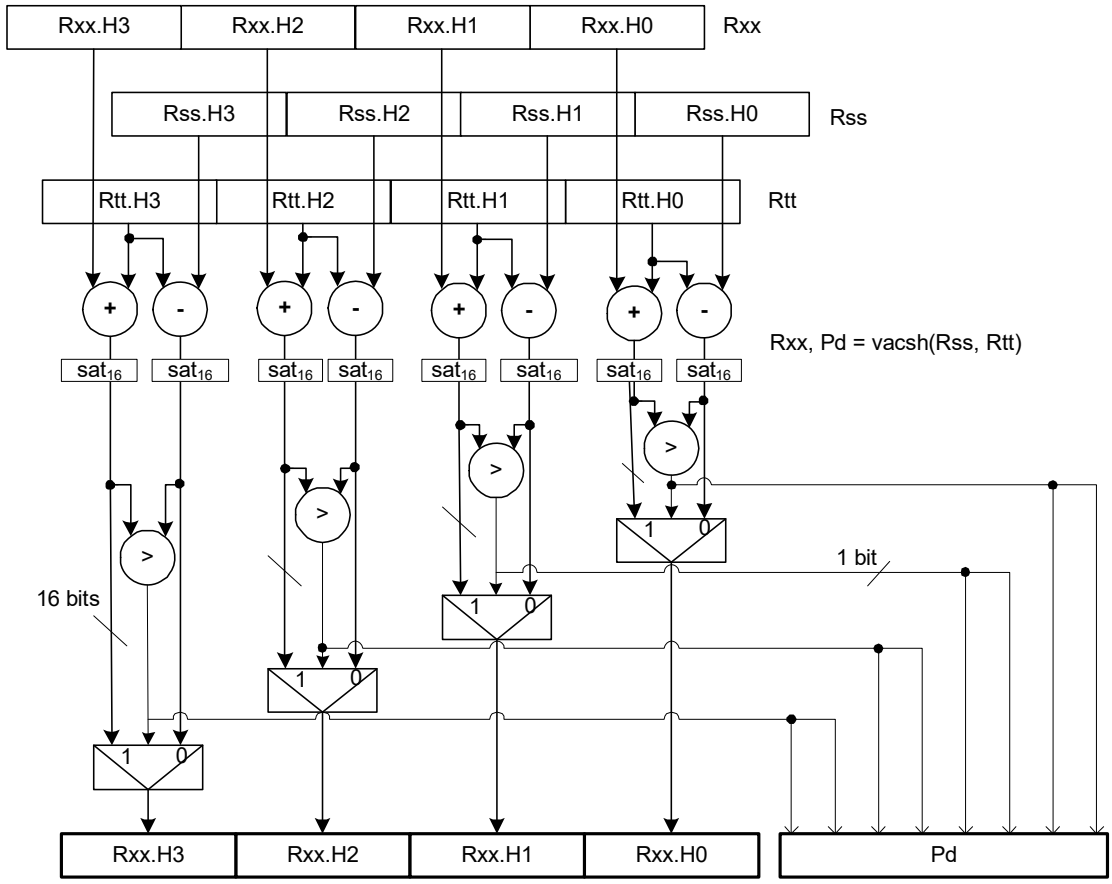
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vabsdiffw(Rtt,Rss)

Field name	Description
`ICLASS`	Instruction class
`MajOp`	Major opcode
`MinOp`	Minor opcode
`RegType`	Register type
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector add compare and select maximum bytes

Add each byte element in Rxx and Rtt, and compare the resulting sums with the corresponding differences between Rss and Rtt. Store the maximum value of each compare in Rxx, and set the corresponding bits in a predicate destination to '1' if the compare result is greater, '0' if not. Each sum and difference is saturated to 8 bits before the compare, and the compare operation is a signed byte compare.

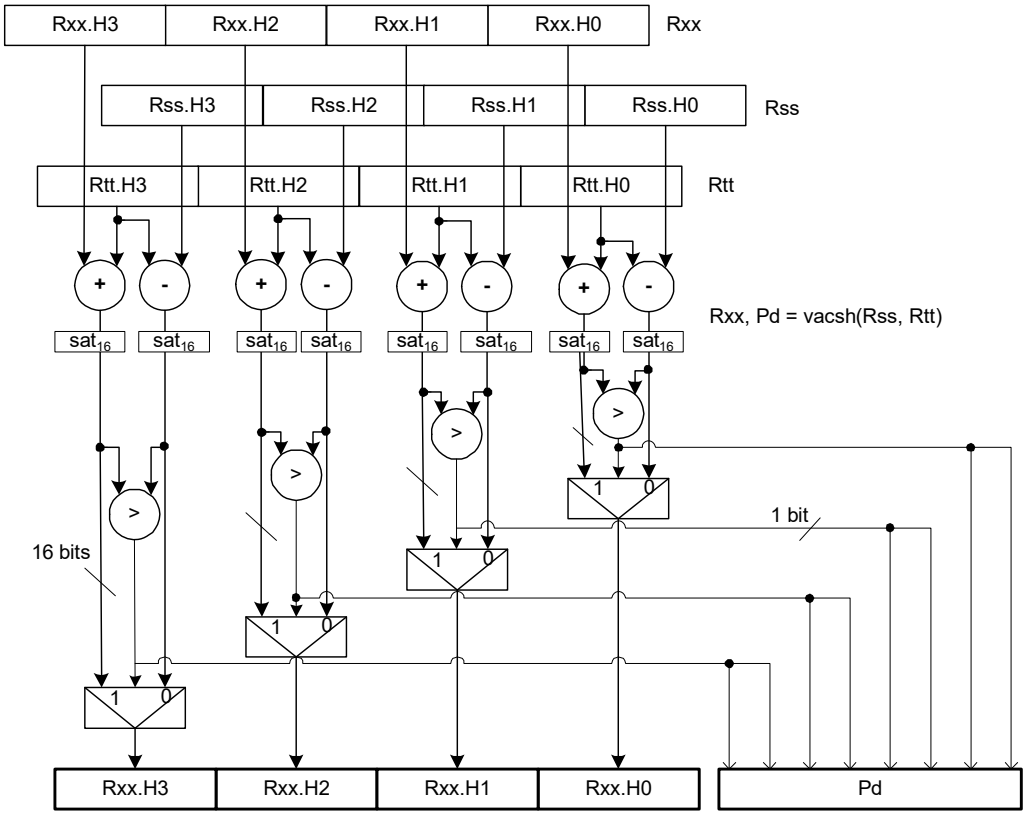


Syntax	Behavior

Class: N/A

Vector add compare and select maximum halfwords

Add each halfword element in Rxx and Rtt, and compare the resulting sums with the corresponding differences between Rss and Rtt. Store the maximum value of each compare in Rxx, and set the corresponding bits in a predicate destination to '11' if the compare result is greater, '00' if not. Each sum and difference is saturated to 16 bits before the compare, and the compare operation is a signed halfword compare.



Syntax	Behavior

`Rxx,Pe=vacsh(Rss,Rtt)`	`for (i = 0; i < 4; i++) {`
	`  xv = (int) Rxx.h[i];`
	`  sv = (int) Rss.h[i];`
	`  tv = (int) Rtt.h[i];`
	`  xv = xv + tv;`
	`  sv = sv - tv;`
	`  Pe.i*2 = (xv > sv);`
	`  Pe.i*2+1 = (xv > sv);`
	`  Rxx.h[i]=sat16(max(xv,sv));`
	`}`

Class: XTYPE (slots 2,3)

Notes

- The predicate generated by this instruction cannot be used as a .new predicate, nor can it be automatically AND'd with another predicate.

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5						e2		x5						
1	1	1	0	1	0	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**e**	**e**	**x**	**x**	**x**	**x**	**x**	Rxx,Pe=vacsh(Rss,Rtt)

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
e2	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector add halfwords

Add each of the four halfwords in 64-bit vector Rss to the corresponding halfword in vector Rtt.

Optionally saturate each 16-bit addition to either a signed or unsigned 16-bit value. Applying saturation to the `vaddh` instruction clamps the result to the signed range 0x8000 to 0x7fff, whereas applying saturation to the `vadduh` instruction ensures that the unsigned result falls within the range 0 to 0xffff. When saturation is not needed, the `vaddh` form should be used.

For the 32-bit version of this vector operation, see the ALU32 instructions.

Syntax	Behavior

`Rdd=vaddh(Rss,Rtt)[:sat]`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=[sat16](Rss.h[i]+Rtt.h[i]);`
	`}`

`Rdd=vadduh(Rss,Rtt):sat`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=usat16(Rss.uh[i]+Rtt.uh[i]);`
	`}`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vaddh(Rss,Rtt)`	`Word64 Q6_P_vaddh_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vaddh(Rss,Rtt):sat`	`Word64 Q6_P_vaddh_PP_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vadduh(Rss,Rtt):sat`	`Word64 Q6_P_vadduh_PP_sat(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5						MinOp			d5					
1	1	0	1	0	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vaddh(Rss,Rtt)
1	1	0	1	0	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vaddh(Rss,Rtt):sat
1	1	0	1	0	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vadduh(Rss,Rtt):sat

Field name	Description
`RegType`	Register type
`MinOp`	Minor opcode
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector add halfwords with saturate and pack to unsigned bytes

Add the four 16-bit halfwords of Rss to the four 16-bit halfwords of Rtt. The results are saturated to unsigned 8-bits and packed in destination register Rd.

Syntax	Behavior

`Rd=vaddhub(Rss,Rtt):sat`	`for (i=0;i<4;i++) {`
	`  Rd.b[i]=usat8(Rss.h[i]+Rtt.h[i]);`
	`}`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=vaddhub(Rss,Rtt):sat`	`Word32 Q6_R_vaddhub_PP_sat(Word64 Rss, Word64 Rtt)`

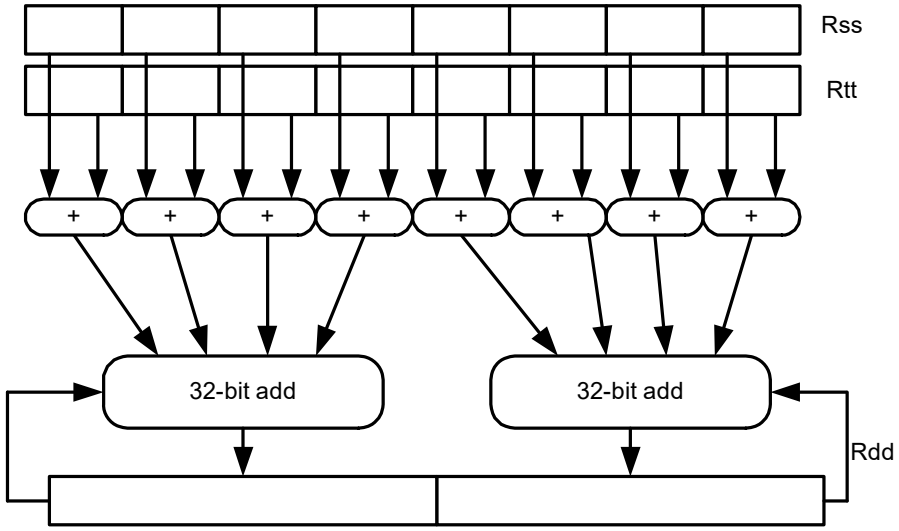
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	0	1	0	1	-	s	s	s	s	s	P	P	-	t	t	t	t	t	0	0	1	d	d	d	d	d	Rd=vaddhub(Rss,Rtt):sat

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
Maj	Major opcode
Min	Minor opcode
RegType	Register type

Vector reduce add unsigned bytes

For each byte in the source vector Rss, add the corresponding byte in the source vector Rtt. Add the four upper intermediate results and optionally the upper word of the destination. Add the four lower results and optionally the lower word of the destination.



Syntax	Behavior

`Rdd=vraddub(Rss,Rtt)`	`Rdd = 0;`
	`for (i=0;i<4;i++) {`
	`  Rdd.w[0]=(Rdd.w[0] + (Rss.ub[i]+Rtt.ub[i]));`
	`}`
	`for (i=4;i<8;i++) {`
	`  Rdd.w[1]=(Rdd.w[1] + (Rss.ub[i]+Rtt.ub[i]));`
	`}`

`Rxx+=vraddub(Rss,Rtt)`	`for (i = 0; i < 4; i++) {`
	`  Rxx.w[0]=(Rxx.w[0] + (Rss.ub[i]+Rtt.ub[i]));`
	`}`
	`for (i = 4; i < 8; i++) {`
	`  Rxx.w[1]=(Rxx.w[1] + (Rss.ub[i]+Rtt.ub[i]));`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vraddub(Rss,Rtt)`	`Word64 Q6_P_vraddub_PP(Word64 Rss, Word64 Rtt)`
`Rxx+=vraddub(Rss,Rtt)`	`Word64 Q6_P_vraddubacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`

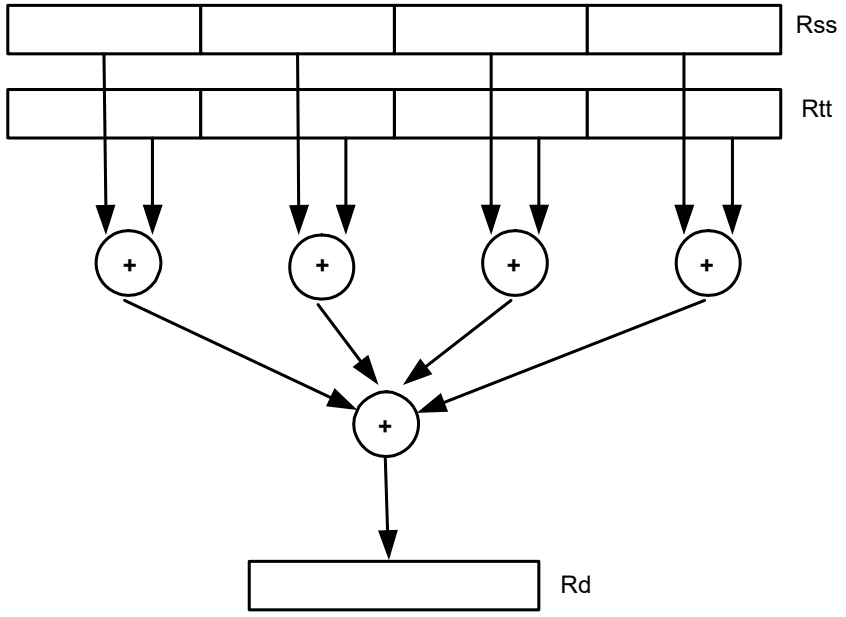
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vraddub(Rss,Rtt)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vraddub(Rss,Rtt)

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector reduce add halfwords

For each halfword in the source vector Rss, add the corresponding halfword in the source vector Rtt. Add these intermediate results together, and place the result in Rd.



Syntax	Behavior

`Rd=vraddh(Rss,Rtt)`	`Rd = 0;`
	`for (i=0;i<4;i++) {`
	`  Rd += (Rss.h[i]+Rtt.h[i]);`
	`}`

`Rd=vradduh(Rss,Rtt)`	`Rd = 0;`
	`for (i=0;i<4;i++) {`
	`  Rd += (Rss.uh[i]+Rtt.uh[i]);`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=vraddh(Rss,Rtt)`	`Word32 Q6_R_vraddh_PP(Word64 Rss, Word64 Rtt)`
`Rd=vradduh(Rss,Rtt)`	`Word32 Q6_R_vradduh_PP(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	1	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	-	0	1	**d**	**d**	**d**	**d**	**d**	Rd=vradduh(Rss,Rtt)
1	1	1	0	1	0	0	1	0	-	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=vraddh(Rss,Rtt)

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector add bytes

Add each of the eight bytes in 64-bit vector Rss to the corresponding byte in vector Rtt. Optionally, saturate each 8-bit addition to an unsigned value between 0 and 255. The eight results are stored in destination register Rdd.

Syntax	Behavior
`Rdd=vaddb(Rss,Rtt)`	`Assembler mapped to: "Rdd=vaddub(Rss,Rtt)"`

`Rdd=vaddub(Rss,Rtt)[:sat]`	`for (i = 0; i < 8; i++) {`
	`  Rdd.b[i]=[usat8](Rss.ub[i]+Rtt.ub[i]);`
	`}`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vaddb(Rss,Rtt)`	`Word64 Q6_P_vaddb_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vaddub(Rss,Rtt)`	`Word64 Q6_P_vaddub_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vaddub(Rss,Rtt):sat`	`Word64 Q6_P_vaddub_PP_sat(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vaddub(Rss,Rtt)
1	1	0	1	0	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vaddub(Rss,Rtt):sat

Field name	Description
RegType	Register type
MinOp	Minor opcode
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector add words

Add each of the two words in 64-bit vector Rss to the corresponding word in vector Rtt. Optionally, saturate each 32-bit addition to a signed value between 0x80000000 and 0x7fffffff. The two word results are stored in destination register Rdd.

Syntax	Behavior

`Rdd=vaddw(Rss,Rtt)[:sat]`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=[sat32](Rss.w[i]+Rtt.w[i]);`
	`}`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vaddw(Rss,Rtt)`	`Word64 Q6_P_vaddw_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vaddw(Rss,Rtt):sat`	`Word64 Q6_P_vaddw_PP_sat(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vaddw(Rss,Rtt)
1	1	0	1	0	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vaddw(Rss,Rtt):sat

Field name	Description
RegType	Register type
MinOp	Minor opcode
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector average halfwords

Average each of the four halfwords in the 64-bit source vector Rss with the corresponding halfword in Rtt. The average operation performed on each halfword adds the two halfwords and shifts the result right by 1 bit. Unsigned average uses a logical right shift (shift in 0), whereas signed average uses an arithmetic right shift (shift in the sign bit). If the round option is used, a 0x0001 is also added to each result before shifting. This operation does not overflow. If a summation (before right shift by 1) causes an overflow of 32 bits, the value shifted in is the most-significant carry out.

The signed average and negative average halfwords is available with optional convergent rounding. In convergent rounding, if the two LSBs after the addition or subtraction are 11, a rounding constant of 1 is added, otherwise a 0 is added. This result is shifted right by one bit. Convergent rounding accumulates less error than arithmetic rounding.

Syntax	Behavior

`Rdd=vavgh(Rss,Rtt)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=(Rss.h[i]+Rtt.h[i])>>1;`
	`}`

`Rdd=vavgh(Rss,Rtt):crnd`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=convround(Rss.h[i]+Rtt.h[i])>>1;`
	`}`

`Rdd=vavgh(Rss,Rtt):rnd`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=(Rss.h[i]+Rtt.h[i]+1)>>1;`
	`}`

`Rdd=vavguh(Rss,Rtt)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=(Rss.uh[i]+Rtt.uh[i])>>1;`
	`}`

`Rdd=vavguh(Rss,Rtt):rnd`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=(Rss.uh[i]+Rtt.uh[i]+1)>>1;`
	`}`

`Rdd=vnavgh(Rtt,Rss)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=(Rtt.h[i]-Rss.h[i])>>1;`
	`}`

`Rdd=vnavgh(Rtt,Rss):crnd:sat`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=`sat_{16}`(convround(Rtt.h[i]-Rss.h[i])>>1);`
	`}`

`Rdd=vnavgh(Rtt,Rss):rnd:sat`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=`sat_{16}`((Rtt.h[i]-Rss.h[i] + 1)>>1);`
	`}`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vavgh(Rss,Rtt)`	`Word64 Q6_P_vavgh_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vavgh(Rss,Rtt):crnd`	`Word64 Q6_P_vavgh_PP_crnd(Word64 Rss, Word64 Rtt)`
`Rdd=vavgh(Rss,Rtt):rnd`	`Word64 Q6_P_vavgh_PP_rnd(Word64 Rss, Word64 Rtt)`
`Rdd=vavguh(Rss,Rtt)`	`Word64 Q6_P_vavguh_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vavguh(Rss,Rtt):rnd`	`Word64 Q6_P_vavguh_PP_rnd(Word64 Rss, Word64 Rtt)`
`Rdd=vnavgh(Rtt,Rss)`	`Word64 Q6_P_vnavgh_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vnavgh(Rtt,Rss):crnd:sat`	`Word64 Q6_P_vnavgh_PP_crnd_sat(Word64 Rtt, Word64 Rss)`
`Rdd=vnavgh(Rtt,Rss):rnd:sat`	`Word64 Q6_P_vnavgh_PP_rnd_sat(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vavgh(Rss,Rtt)
1	1	0	1	0	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vavgh(Rss,Rtt):rnd
1	1	0	1	0	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vavgh(Rss,Rtt):crnd
1	1	0	1	0	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vavguh(Rss,Rtt)
1	1	0	1	0	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vavguh(Rss,Rtt):rnd
1	1	0	1	0	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vnavgh(Rtt,Rss)
1	1	0	1	0	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vnavgh(Rtt,Rss):rnd:sat
1	1	0	1	0	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vnavgh(Rtt,Rss):crnd:sat

Field name	Description
RegType	Register type
MinOp	Minor opcode
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector average unsigned bytes

Average each of the eight unsigned bytes in the 64-bit source vector Rss with the corresponding byte in Rtt. The average operation performed on each byte is the sum of the two bytes shifted right by 1 bit. If the round option is used, a 0x01 is also added to each result before shifting. This operation does not overflow. In the case that a summation (before right shift by 1) causes an overflow of 8 bits, the value shifted in is the most-significant carry out.

Syntax	Behavior

`Rdd=vavgub(Rss,Rtt)`	`for (i = 0; i < 8; i++) {`
	`  Rdd.b[i]=((Rss.ub[i] + Rtt.ub[i])>>1);`
	`}`

`Rdd=vavgub(Rss,Rtt):rnd`	`for (i = 0; i < 8; i++) {`
	`  Rdd.b[i]=((Rss.ub[i]+Rtt.ub[i]+1)>>1);`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vavgub(Rss,Rtt)`	`Word64 Q6_P_vavgub_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vavgub(Rss,Rtt):rnd`	`Word64 Q6_P_vavgub_PP_rnd(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vavgub(Rss,Rtt)
1	1	0	1	0	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vavgub(Rss,Rtt):rnd

Field name	Description
`RegType`	Register type
`MinOp`	Minor opcode
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector average words

Average each of the two words in the 64-bit source vector Rss with the corresponding word in Rtt. The average operation performed on each halfword adds the two words and shifts the result right by 1 bit. Unsigned average uses a logical right shift (shift in 0), whereas signed average uses an arithmetic right shift (shift in the sign bit). If the round option is used, a 0x1 is also added to each result before shifting. This operation does not overflow. In the case that a summation (before right shift by 1) causes an overflow of 32 bits, the value shifted in is the most-significant carry out.

The signed average and negative average words is available with optional convergent rounding. In convergent rounding, if the two LSBs after the addition/subtraction are 11, a rounding constant of 1 is added, otherwise a 0 is added. This result is then shifted right by one bit. Convergent rounding accumulates less error than arithmetic rounding.

Syntax	Behavior

`Rdd=vavguw(Rss,Rtt)[:rnd]`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=(zxt32->33(Rss.uw[i])+zxt32->33(Rtt.uw[i])+1)>>1;`
	`}`

`Rdd=vavgw(Rss,Rtt):crnd`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=(convround(sxt32->33(Rss.w[i])+sxt32->33(Rtt.w[i]))>>1);`
	`}`

`Rdd=vavgw(Rss,Rtt)[:rnd]`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=(sxt32->33(Rss.w[i])+sxt32->33(Rtt.w[i])+1)>>1;`
	`}`

`Rdd=vnavgw(Rtt,Rss)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=(sxt32->33(Rtt.w[i])-sxt32->33(Rss.w[i]))>>1;`
	`}`

`Rdd=vnavgw(Rtt,Rss):crnd:sat`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=sat32(convround(sxt32->33(Rtt.w[i])-sxt32->33(Rss.w[i]))>>1);`
	`}`

`Rdd=vnavgw(Rtt,Rss):rnd:sat`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=sat32((sxt32->33(Rtt.w[i])-sxt32->33(Rss.w[i])+1)>>1);`
	`}`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vavguw(Rss,Rtt)`	`Word64 Q6_P_vavguw_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vavguw(Rss,Rtt):rnd`	`Word64 Q6_P_vavguw_PP_rnd(Word64 Rss, Word64 Rtt)`
`Rdd=vavgw(Rss,Rtt)`	`Word64 Q6_P_vavgw_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vavgw(Rss,Rtt):crnd`	`Word64 Q6_P_vavgw_PP_crnd(Word64 Rss, Word64 Rtt)`
`Rdd=vavgw(Rss,Rtt):rnd`	`Word64 Q6_P_vavgw_PP_rnd(Word64 Rss, Word64 Rtt)`
`Rdd=vnavgw(Rtt,Rss)`	`Word64 Q6_P_vnavgw_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vnavgw(Rtt,Rss):crnd:sat`	`Word64 Q6_P_vnavgw_PP_crnd_sat(Word64 Rtt, Word64 Rss)`
`Rdd=vnavgw(Rtt,Rss):rnd:sat`	`Word64 Q6_P_vnavgw_PP_rnd_sat(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vavgw(Rss,Rtt)
1	1	0	1	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vavgw(Rss,Rtt):rnd
1	1	0	1	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vavgw(Rss,Rtt):crnd
1	1	0	1	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vavguw(Rss,Rtt)
1	1	0	1	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vavguw(Rss,Rtt):rnd
1	1	0	1	0	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vnavgw(Rtt,Rss)
1	1	0	1	0	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=vnavgw(Rtt,Rss):rnd:sat
1	1	0	1	0	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vnavgw(Rtt,Rss):crnd:sat

Field name	Description
RegType	Register type
MinOp	Minor opcode
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector clip to unsigned

Clip input to unsigned int

Syntax	Behavior

`Rdd=vclip(Rss,#u5)`	`tmp=MIN((1<<#u)-1,MAX(Rss.w[0],-(1<<#u)));`
	`Rdd.w[0]=tmp;`
	`tmp=MIN((1<<#u)-1,MAX(Rss.w[1],-(1<<#u)));`
	`Rdd.w[1]=tmp;`
	`;`

Class: XTYPE (slots 2,3)

Notes

- This instruction can only execute on a core with the Hexagon audio extensions

Intrinsics

`Rdd=vclip(Rss,#u5)`	`Word64 Q6_P_vclip_PI(Word64 Rss, Word32 Iu5)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	1	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vclip(Rss,#u5)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

Vector conditional negate

Based on bits in Rt, conditionally negate halves in Rss.

Syntax	Behavior

`Rdd=vcnegh(Rss,Rt)`	`for (i = 0; i < 4; i++) {`
	`  if (Rt.i) {`
	`  Rdd.h[i]=sat16(-Rss.h[i]);`
	`  } else {`
	`  Rdd.h[i]=Rss.h[i];`
	`  }}`

`Rxx+=vrcnegh(Rss,Rt)`	`for (i = 0; i < 4; i++) {`
	`  if (Rt.i) {`
	`  Rxx += -Rss.h[i];`
	`  } else {`
	`  Rxx += Rss.h[i];`
	`  }};`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vcnegh(Rss,Rt)`	`Word64 Q6_P_vcnegh_PR(Word64 Rss, Word32 Rt)`
`Rxx+=vrcnegh(Rss,Rt)`	`Word64 Q6_P_vrcneghacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	1	1	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vcnegh(Rss,Rt)
ICLASS				RegType				Maj			s5					Parse			t5					Min			x5					
1	1	0	0	1	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vrcnegh(Rss,Rt)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x
Maj	Major opcode
Min	Minor opcode
RegType	Register type

Vector maximum bytes

Compare each of the eight unsigned bytes in the 64-bit source vector Rss to the corresponding byte in Rtt. For each comparison, select the maximum of the two bytes and place that byte in the corresponding location in Rdd.

Syntax	Behavior

`Rdd=vmaxb(Rtt,Rss)`	`for (i = 0; i < 8; i++) {`
	`  Rdd.b[i]=max(Rtt.b[i],Rss.b[i]);`
	`}`

`Rdd=vmaxub(Rtt,Rss)`	`for (i = 0; i < 8; i++) {`
	`  Rdd.b[i]=max(Rtt.ub[i],Rss.ub[i]);`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vmaxb(Rtt,Rss)`	`Word64 Q6_P_vmaxb_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vmaxub(Rtt,Rss)`	`Word64 Q6_P_vmaxub_PP(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vmaxub(Rtt,Rss)
1	1	0	1	0	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vmaxb(Rtt,Rss)

Field name	Description
RegType	Register type
MinOp	Minor opcode
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector maximum halfwords

Compare each of the four halfwords in the 64-bit source vector Rss to the corresponding halfword in Rtt. For each comparison, select the maximum of the two halfwords and place that halfword in the corresponding location in Rdd. Comparisons are available in both signed and unsigned form.

Syntax	Behavior
`Rdd=vmaxh(Rtt,Rss)`	`for (i = 0; i < 4; i++) { Rdd.h[i]=max(Rtt.h[i],Rss.h[i]); }`
`Rdd=vmaxuh(Rtt,Rss)`	`for (i = 0; i < 4; i++) { Rdd.h[i]=max(Rtt.uh[i],Rss.uh[i]); }`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vmaxh(Rtt,Rss)`	`Word64 Q6_P_vmaxh_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vmaxuh(Rtt,Rss)`	`Word64 Q6_P_vmaxuh_PP(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5						MinOp			d5					
1	1	0	1	0	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmaxh(Rtt,Rss)
1	1	0	1	0	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vmaxuh(Rtt,Rss)

Field name	Description
`RegType`	Register type
`MinOp`	Minor opcode
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector reduce maximum halfwords

Register Rxx contains a maximum value in the low word and the address of that maximum value in the high word. Register Rss contains a vector of four halfword values, and register Ru contains the address of this data. The instruction finds the maximum halfword between the previous maximum in Rxx[0] and the four values in Rss. The address of the new maximum is stored in Rxx[1].

Syntax	Behavior

`Rxx=vrmaxh(Rss,Ru)`	`max = Rxx.h[0];`
	`addr = Rxx.w[1];`
	`for (i = 0; i < 4; i++) {`
	`  if (max < Rss.h[i]) {`
	`  max = Rss.h[i];`
	`  addr = Ru | i<<1;`
	`  }`
	`}`
	`Rxx.w[0]=max;`
	`Rxx.w[1]=addr;`

`Rxx=vrmaxuh(Rss,Ru)`	`max = Rxx.uh[0];`
	`addr = Rxx.w[1];`
	`for (i = 0; i < 4; i++) {`
	`  if (max < Rss.uh[i]) {`
	`  max = Rss.uh[i];`
	`  addr = Ru | i<<1;`
	`  }`
	`}`
	`Rxx.w[0]=max;`
	`Rxx.w[1]=addr;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rxx=vrmaxh(Rss,Ru)`	`Word64 Q6_P_vrmaxh_PR(Word64 Rxx, Word64 Rss, Word32 Ru)`
`Rxx=vrmaxuh(Rss,Ru)`	`Word64 Q6_P_vrmaxuh_PR(Word64 Rxx, Word64 Rss, Word32 Ru)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			x5					Min			u5					
1	1	0	0	1	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**x**	**x**	**x**	**x**	**x**	0	0	1	**u**	**u**	**u**	**u**	**u**	Rxx=vrmaxh(Rss,Ru)
1	1	0	0	1	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**x**	**x**	**x**	**x**	**x**	0	0	1	**u**	**u**	**u**	**u**	**u**	Rxx=vrmaxuh(Rss,Ru)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
s5	Field to encode register s
u5	Field to encode register u
x5	Field to encode register x
Maj	Major opcode
Min	Minor opcode
RegType	Register type

Vector reduce maximum words

Find the maximum word between the previous maximum in Rxx[0] and the two values in Rss. The address of the new maximum is stored in Rxx[1].

Register Rxx contains a maximum value in the low word and the address of that maximum value in the high word. Register Rss contains a vector of two word values, and register Ru contains the address of this data.

Syntax	Behavior

`Rxx=vrmaxuw(Rss,Ru)`	`max = Rxx.uw[0];`
	`addr = Rxx.w[1];`
	`for (i = 0; i < 2; i++) {`
	`  if (max < Rss.uw[i]) {`
	`  max = Rss.uw[i];`
	`  addr = Ru | i<<2;`
	`  }`
	`}`
	`Rxx.w[0]=max;`
	`Rxx.w[1]=addr;`

`Rxx=vrmaxw(Rss,Ru)`	`max = Rxx.w[0];`
	`addr = Rxx.w[1];`
	`for (i = 0; i < 2; i++) {`
	`  if (max < Rss.w[i]) {`
	`  max = Rss.w[i];`
	`  addr = Ru | i<<2;`
	`  }`
	`}`
	`Rxx.w[0]=max;`
	`Rxx.w[1]=addr;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rxx=vrmaxuw(Rss,Ru)`	`Word64 Q6_P_vrmaxuw_PR(Word64 Rxx, Word64 Rss, Word32 Ru)`
`Rxx=vrmaxw(Rss,Ru)`	`Word64 Q6_P_vrmaxw_PR(Word64 Rxx, Word64 Rss, Word32 Ru)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			x5					Min			u5					
1	1	0	0	1	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**x**	**x**	**x**	**x**	**x**	0	1	0	**u**	**u**	**u**	**u**	**u**	Rxx=vrmaxw(Rss,Ru)
1	1	0	0	1	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**x**	**x**	**x**	**x**	**x**	0	1	0	**u**	**u**	**u**	**u**	**u**	Rxx=vrmaxuw(Rss,Ru)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
s5	Field to encode register s
u5	Field to encode register u
x5	Field to encode register x
Maj	Major opcode
Min	Minor opcode
RegType	Register type

Vector maximum words

Compare each of the two words in the 64-bit source vector Rss to the corresponding word in Rtt. For each comparison, select the maximum of the two words and place that word in the corresponding location in Rdd.

Comparisons are available in both signed and unsigned form.

Syntax	Behavior

`Rdd=vmaxuw(Rtt,Rss)`	`for (i = 0; i < 2; i++) {`
	`  Rdd.w[i]=max(Rtt.uw[i],Rss.uw[i]);`
	`}`

`Rdd=vmaxw(Rtt,Rss)`	`for (i = 0; i < 2; i++) {`
	`  Rdd.w[i]=max(Rtt.w[i],Rss.w[i]);`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vmaxuw(Rtt,Rss)`	`Word64 Q6_P_vmaxuw_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vmaxw(Rtt,Rss)`	`Word64 Q6_P_vmaxw_PP(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmaxuw(Rtt,Rss)
1	1	0	1	0	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmaxw(Rtt,Rss)

Field name	Description
`RegType`	Register type
`MinOp`	Minor opcode
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector minimum bytes

Compare each of the eight unsigned bytes in the 64-bit source vector Rss to the corresponding byte in Rtt. For each comparison, select the minimum of the two bytes and place that byte in the corresponding location in Rdd.

Syntax	Behavior

`Rdd,Pe=vminub(Rtt,Rss)`	`for (i = 0; i < 8; i++) {`
	`  Pe.i = (Rtt.ub[i] > Rss.ub[i]);`
	`  Rdd.b[i]=min(Rtt.ub[i],Rss.ub[i]);`
	`}`

`Rdd=vminb(Rtt,Rss)`	`for (i = 0; i < 8; i++) {`
	`  Rdd.b[i]=min(Rtt.b[i],Rss.b[i]);`
	`}`

`Rdd=vminub(Rtt,Rss)`	`for (i = 0; i < 8; i++) {`
	`  Rdd.b[i]=min(Rtt.ub[i],Rss.ub[i]);`
	`}`

Class: XTYPE (slots 2,3)

Notes

- The predicate generated by this instruction cannot be used as a .new predicate, nor can it be automatically AND'd with another predicate.

Intrinsics

`Rdd=vminb(Rtt,Rss)`	`Word64 Q6_P_vminb_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vminub(Rtt,Rss)`	`Word64 Q6_P_vminub_PP(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5						MinOp			d5					
1	1	0	1	0	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vminub(Rtt,Rss)
1	1	0	1	0	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vminb(Rtt,Rss)
ICLASS				RegType				MajOp			s5					Parse			t5						e2			d5					
1	1	1	0	1	0	1	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**e**	**e**	**d**	**d**	**d**	**d**	**d**	Rdd,Pe=vminub(Rtt,Rss)

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
e2	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t

Vector minimum halfwords

Compare each of the four halfwords in the 64-bit source vector Rss to the corresponding halfword in Rtt. For each comparison, select the minimum of the two halfwords and place that halfword in the corresponding location in Rdd.

Comparisons are available in both signed and unsigned form.

Syntax	Behavior

`Rdd=vminh(Rtt,Rss)`	`for (i = 0; i < 4; i++) {`
	`  Rdd.h[i]=min(Rtt.h[i],Rss.h[i]);`
	`}`

`Rdd=vminuh(Rtt,Rss)`	`for (i = 0; i < 4; i++) {`
	`  Rdd.h[i]=min(Rtt.uh[i],Rss.uh[i]);`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vminh(Rtt,Rss)`	`Word64 Q6_P_vminh_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vminuh(Rtt,Rss)`	`Word64 Q6_P_vminuh_PP(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vminh(Rtt,Rss)
1	1	0	1	0	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vminuh(Rtt,Rss)

Field name	Description
`RegType`	Register type
`MinOp`	Minor opcode
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector reduce minimum halfwords

Find the minimum halfword between the previous minimum in Rxx[0] and the four values in Rss. The address of the new minimum is stored in Rxx[1].

Register Rxx contains a minimum value in the low word and the address of that minimum value in the high word. Register Rss contains a vector of four halfword values, and register Ru contains the address of this data.

Syntax	Behavior

`Rxx=vrminh(Rss,Ru)`	`min = Rxx.h[0];`
	`addr = Rxx.w[1];`
	`for (i = 0; i < 4; i++) {`
	`  if (min > Rss.h[i]) {`
	`  min = Rss.h[i];`
	`  addr = Ru | i<<1;`
	`  }`
	`}`
	`Rxx.w[0]=min;`
	`Rxx.w[1]=addr;`

`Rxx=vrminuh(Rss,Ru)`	`min = Rxx.uh[0];`
	`addr = Rxx.w[1];`
	`for (i = 0; i < 4; i++) {`
	`  if (min > Rss.uh[i]) {`
	`  min = Rss.uh[i];`
	`  addr = Ru | i<<1;`
	`  }`
	`}`
	`Rxx.w[0]=min;`
	`Rxx.w[1]=addr;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rxx=vrminh(Rss,Ru)`	`Word64 Q6_P_vrminh_PR(Word64 Rxx, Word64 Rss, Word32 Ru)`
`Rxx=vrminuh(Rss,Ru)`	`Word64 Q6_P_vrminuh_PR(Word64 Rxx, Word64 Rss, Word32 Ru)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			x5					Min			u5					
1	1	0	0	1	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**x**	**x**	**x**	**x**	**x**	1	0	1	**u**	**u**	**u**	**u**	**u**	Rxx=vrminh(Rss,Ru)
1	1	0	0	1	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**x**	**x**	**x**	**x**	**x**	1	0	1	**u**	**u**	**u**	**u**	**u**	Rxx=vrminuh(Rss,Ru)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
s5	Field to encode register s
u5	Field to encode register u
x5	Field to encode register x
Maj	Major opcode
Min	Minor opcode
RegType	Register type

Vector reduce minimum words

Find the minimum word between the previous minimum in Rxx[0] and the two values in Rss. The address of the new minimum is stored in Rxx[1].

Register Rxx contains a minimum value in the low word and the address of that minimum value in the high word. Register Rss contains a vector of two word values, and register Ru contains the address of this data.

Syntax	Behavior

`Rxx=vrminuw(Rss,Ru)`	`min = Rxx.uw[0];`
	`addr = Rxx.w[1];`
	`for (i = 0; i < 2; i++) {`
	`  if (min > Rss.uw[i]) {`
	`  min = Rss.uw[i];`
	`  addr = Ru | i<<2;`
	`  }`
	`}`
	`Rxx.w[0]=min;`
	`Rxx.w[1]=addr;`

`Rxx=vrminw(Rss,Ru)`	`min = Rxx.w[0];`
	`addr = Rxx.w[1];`
	`for (i = 0; i < 2; i++) {`
	`  if (min > Rss.w[i]) {`
	`  min = Rss.w[i];`
	`  addr = Ru | i<<2;`
	`  }`
	`}`
	`Rxx.w[0]=min;`
	`Rxx.w[1]=addr;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rxx=vrminuw(Rss,Ru)`	`Word64 Q6_P_vrminuw_PR(Word64 Rxx, Word64 Rss, Word32 Ru)`
`Rxx=vrminw(Rss,Ru)`	`Word64 Q6_P_vrminw_PR(Word64 Rxx, Word64 Rss, Word32 Ru)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			x5					Min			u5					
1	1	0	0	1	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**x**	**x**	**x**	**x**	**x**	1	1	0	**u**	**u**	**u**	**u**	**u**	Rxx=vrminw(Rss,Ru)
1	1	0	0	1	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**x**	**x**	**x**	**x**	**x**	1	1	0	**u**	**u**	**u**	**u**	**u**	Rxx=vrminuw(Rss,Ru)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
s5	Field to encode register s
u5	Field to encode register u
x5	Field to encode register x
Maj	Major opcode
Min	Minor opcode
RegType	Register type

Vector minimum words

Compare each of the two words in the 64-bit source vector Rss to the corresponding word in Rtt. For each comparison, select the minimum of the two words and place that word in the corresponding location in Rdd.

Comparisons are available in both signed and unsigned form.

Syntax	Behavior

`Rdd=vminuw(Rtt,Rss)`	`for (i = 0; i < 2; i++) {`
	`  Rdd.w[i]=min(Rtt.uw[i],Rss.uw[i]);`
	`}`

`Rdd=vminw(Rtt,Rss)`	`for (i = 0; i < 2; i++) {`
	`  Rdd.w[i]=min(Rtt.w[i],Rss.w[i]);`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vminuw(Rtt,Rss)`	`Word64 Q6_P_vminuw_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vminw(Rtt,Rss)`	`Word64 Q6_P_vminw_PP(Word64 Rtt, Word64 Rss)`

Encoding

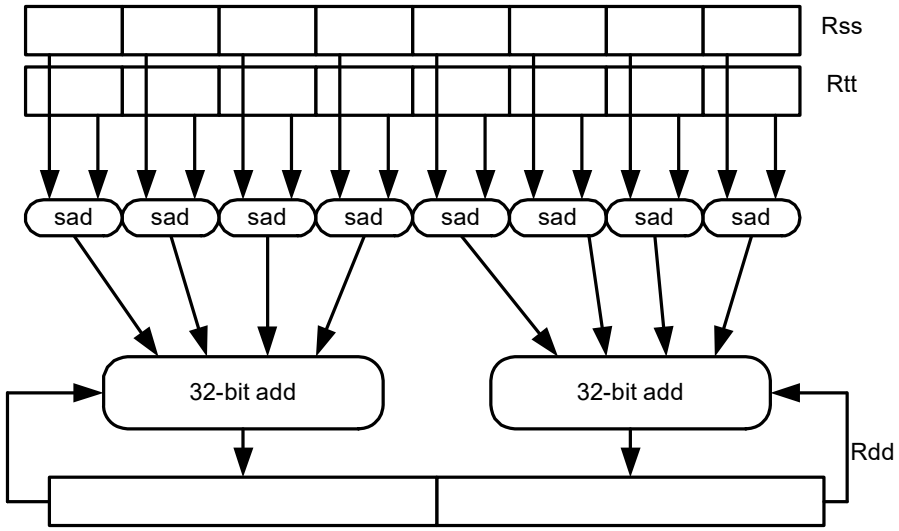
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vminw(Rtt,Rss)
1	1	0	1	0	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vminuw(Rtt,Rss)

Field name	Description
`RegType`	Register type
`MinOp`	Minor opcode
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector sum of absolute differences unsigned bytes

For each byte in the source vector Rss, subtract the corresponding byte in source vector Rtt. Take the absolute value of the intermediate results, and the upper four together and add the lower four together. Optionally, add the destination upper and lower words to these results.

This instruction is useful in determining distance between two vectors, in applications such as motion estimation.



Syntax	Behavior

`Rdd=vrsadub(Rss,Rtt)`	`Rdd = 0;`
	`for (i = 0; i < 4; i++) {`
	`  Rdd.w[0]=(Rdd.w[0] + ABS((Rss.ub[i] - Rtt.ub[i])));`
	`}`
	`for (i = 4; i < 8; i++) {`
	`  Rdd.w[1]=(Rdd.w[1] + ABS((Rss.ub[i] - Rtt.ub[i])));`
	`}`

`Rxx+=vrsadub(Rss,Rtt)`	`for (i = 0; i < 4; i++) {`
	`  Rxx.w[0]=(Rxx.w[0] + ABS((Rss.ub[i] - Rtt.ub[i])));`
	`}`
	`for (i = 4; i < 8; i++) {`
	`  Rxx.w[1]=(Rxx.w[1] + ABS((Rss.ub[i] - Rtt.ub[i])));`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vrsadub(Rss,Rtt)`	`Word64 Q6_P_vrsadub_PP(Word64 Rss, Word64 Rtt)`
`Rxx+=vrsadub(Rss,Rtt)`	`Word64 Q6_P_vrsadubacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vrsadub(Rss,Rtt)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vrsadub(Rss,Rtt)

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector subtract halfwords

Subtract each of the four halfwords in 64-bit vector Rss from the corresponding halfword in vector Rtt.

Optionally, saturate each 16-bit addition to either a signed or unsigned 16-bit value. Applying saturation to the `vsubh` instruction clamps the result to the signed range 0x8000 to 0x7fff, whereas applying saturation to the `vsubuh` instruction ensures that the unsigned result falls within the range 0 to 0xffff.

When saturation is not needed, `vsubh` should be used.

Syntax	Behavior

`Rdd=vsubh(Rtt,Rss)[:sat]`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=[sat16](Rtt.h[i]-Rss.h[i]);`
	`}`

`Rdd=vsubuh(Rtt,Rss):sat`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=usat16(Rtt.uh[i]-Rss.uh[i]);`
	`}`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vsubh(Rtt,Rss)`	`Word64 Q6_P_vsubh_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vsubh(Rtt,Rss):sat`	`Word64 Q6_P_vsubh_PP_sat(Word64 Rtt, Word64 Rss)`
`Rdd=vsubuh(Rtt,Rss):sat`	`Word64 Q6_P_vsubuh_PP_sat(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vsubh(Rtt,Rss)
1	1	0	1	0	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vsubh(Rtt,Rss):sat
1	1	0	1	0	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vsubuh(Rtt,Rss):sat

Field name	Description
RegType	Register type
MinOp	Minor opcode
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector subtract bytes

Subtract each of the eight bytes in 64-bit vector Rss from the corresponding byte in vector Rtt.

Optionally, saturate each 8-bit subtraction to an unsigned value between 0 and 255. The eight results are stored in destination register Rdd.

Syntax	Behavior
`Rdd=vsubb(Rss,Rtt)`	`Assembler mapped to: "Rdd=vsubub(Rss,Rtt)"`
`Rdd=vsubub(Rtt,Rss)[:sat]`	`for (i = 0; i < 8; i++) { Rdd.b[i]=[usat8](Rtt.ub[i]-Rss.ub[i]); }`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vsubb(Rss,Rtt)`	`Word64 Q6_P_vsubb_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vsubub(Rtt,Rss)`	`Word64 Q6_P_vsubub_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vsubub(Rtt,Rss):sat`	`Word64 Q6_P_vsubub_PP_sat(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vsubub(Rtt,Rss)
1	1	0	1	0	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vsubub(Rtt,Rss):sat

Field name	Description
RegType	Register type
MinOp	Minor opcode
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector subtract words

Subtract each of the two words in 64-bit vector Rss from the corresponding word in vector Rtt.

Optionally, saturate each 32-bit subtraction to a signed value between 0x8000_0000 and 0x7fff_ffff. The two word results are stored in destination register Rdd.

Syntax	Behavior

`Rdd=vsubw(Rtt,Rss)[:sat]`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=[sat32](Rtt.w[i]-Rss.w[i]);`
	`}`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vsubw(Rtt,Rss)`	`Word64 Q6_P_vsubw_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vsubw(Rtt,Rss):sat`	`Word64 Q6_P_vsubw_PP_sat(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vsubw(Rtt,Rss)
1	1	0	1	0	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vsubw(Rtt,Rss):sat

Field name	Description
RegType	Register type
MinOp	Minor opcode
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

11.10.2 XTYPE bit

The XTYPE bit instruction subclass includes instructions for bit manipulation.

Count leading

Count leading zeros (`cl0`) counts the number of consecutive zeros starting with the most significant bit.

Count leading ones (`cl1`) counts the number of consecutive ones starting with the most significant bit.

Count leading bits (`clb`) counts both leading ones and leading zeros and then selects the maximum.

The `normamt` instruction returns the number of leading bits minus one.

For a two's-complement number, the number of leading zeros is zero for negative numbers. The number of leading ones is zero for positive numbers.

The number of leading bits can be used to judge the magnitude of the value.

Syntax	Behavior
`Rd=add(clb(Rs),#s6)`	`Rd = (max(count_leading_ones(Rs), count_leading_ones(~Rs)))+#s;`
`Rd=add(clb(Rss),#s6)`	`Rd = (max(count_leading_ones(Rss), count_leading_ones(~Rss)))+#s;`
`Rd=cl0(Rs)`	`Rd = count_leading_ones(~Rs);`
`Rd=cl0(Rss)`	`Rd = count_leading_ones(~Rss);`
`Rd=cl1(Rs)`	`Rd = count_leading_ones(Rs);`
`Rd=cl1(Rss)`	`Rd = count_leading_ones(Rss);`
`Rd=clb(Rs)`	`Rd = max(count_leading_ones(Rs), count_leading_ones(~Rs));`
`Rd=clb(Rss)`	`Rd = max(count_leading_ones(Rss), count_leading_ones(~Rss));`

`Rd=normamt(Rs)`	`if (Rs == 0) {`
	`  Rd = 0;`
	`} else {`
	`  Rd = (max(count_leading_ones(Rs), count_leading_ones(~Rs)))-1;`
	`}`

`Rd=normamt(Rss)`	`if (Rss == 0) {`
	`  Rd = 0;`
	`} else {`
	`  Rd = (max(count_leading_ones(Rss),count_leading_ones(~Rss)))-1;`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

Rd=add(clb(Rs),#s6)	Word32 Q6_R_add_clb_RI(Word32 Rs, Word32 Is6)
Rd=add(clb(Rss),#s6)	Word32 Q6_R_add_clb_PI(Word64 Rss, Word32 Is6)
Rd=cl0(Rs)	Word32 Q6_R_cl0_R(Word32 Rs)
Rd=cl0(Rss)	Word32 Q6_R_cl0_P(Word64 Rss)
Rd=cl1(Rs)	Word32 Q6_R_cl1_R(Word32 Rs)
Rd=cl1(Rss)	Word32 Q6_R_cl1_P(Word64 Rss)
Rd=clb(Rs)	Word32 Q6_R_clb_R(Word32 Rs)
Rd=clb(Rss)	Word32 Q6_R_clb_P(Word64 Rss)
Rd=normamt(Rs)	Word32 Q6_R_normamt_R(Word32 Rs)
Rd=normamt(Rss)	Word32 Q6_R_normamt_P(Word64 Rss)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=clb(Rss)
1	0	0	0	1	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=cl0(Rss)
1	0	0	0	1	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=cl1(Rss)
1	0	0	0	1	0	0	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=normamt(Rss)
1	0	0	0	1	0	0	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=add(clb(Rss),#s6)
1	0	0	0	1	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=clb(Rs)
1	0	0	0	1	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	1	**d**	**d**	**d**	**d**	**d**	Rd=cl0(Rs)
1	0	0	0	1	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=cl1(Rs)
1	0	0	0	1	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=normamt(Rs)
1	0	0	0	1	1	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=add(clb(Rs),#s6)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

Count population

Population Count (`popcount`) counts the number of bits in Rss that are set.

Syntax	Behavior
`Rd = popcount(Rss)`	`Rd = count_ones(Rss);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=popcount(Rss)`	`Word32 Q6_R_popcount_P(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	1	0	0	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	1	1	**d**	**d**	**d**	**d**	**d**	Rd=popcount(Rss)

Field name	Description
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major opcode
`MinOp`	Minor opcode
`RegType`	Register type

Count trailing

Count trailing zeros (`ct0`) counts the number of consecutive zeros starting with the least significant bit.

Count trailing ones (`ct1`) counts the number of consecutive ones starting with the least significant bit.

Syntax	Behavior
`Rd=ct0(Rs)`	`Rd = count_leading_ones(~reverse_bits(Rs));`
`Rd=ct0(Rss)`	`Rd = count_leading_ones(~reverse_bits(Rss));`
`Rd=ct1(Rs)`	`Rd = count_leading_ones(reverse_bits(Rs));`
`Rd=ct1(Rss)`	`Rd = count_leading_ones(reverse_bits(Rss));`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=ct0(Rs)`	`Word32 Q6_R_ct0_R(Word32 Rs)`
`Rd=ct0(Rss)`	`Word32 Q6_R_ct0_P(Word64 Rss)`
`Rd=ct1(Rs)`	`Word32 Q6_R_ct1_R(Word32 Rs)`
`Rd=ct1(Rss)`	`Word32 Q6_R_ct1_P(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=ct0(Rss)
1	0	0	0	1	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=ct1(Rss)
1	0	0	0	1	1	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=ct0(Rs)
1	0	0	0	1	1	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	1	**d**	**d**	**d**	**d**	**d**	Rd=ct1(Rs)

Field name	Description
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major opcode
`MinOp`	Minor opcode
`RegType`	Register type

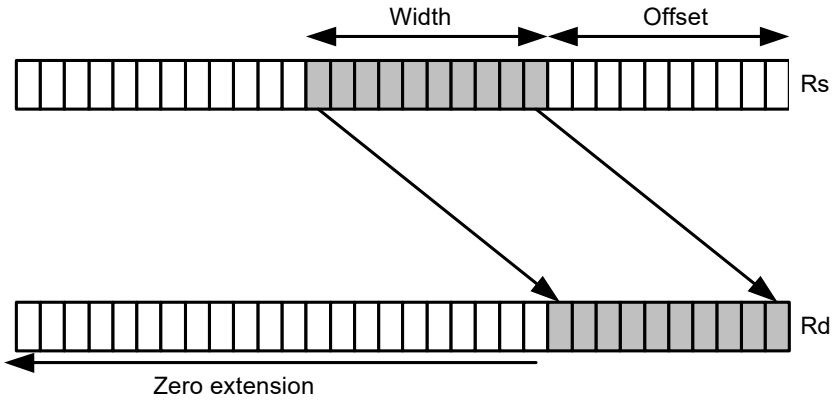
Extract bit field

Extract a bit field from the source register (or register pair) and deposit into the least significant bits of the destination register (or register pair). The other, more significant bits in the destination are either cleared or sign-extended, depending on the instruction.

The width of the extracted field is obtained from the first immediate or from the most-significant word of Rtt. The field offset is obtained from either the second immediate or from the least-significant word of Rtt.

For register-based extract, where Rtt supplies the offset and width, the offset value is treated as a signed 7-bit number. If this value is negative, the source register Rss is shifted left (the reverse direction). Width number of bits are then taken from the least-significant portion of this result.

If the shift amount and/or offset captures data beyond the most significant end of the input, these bits are taken as zero.



Syntax	Behavior

`Rd=extract(Rs,#u5,#U5)`	`width=#u;`
	`offset=#U;`
	`Rd = sxt`$_{width->32}$`((Rs >> offset));`

`Rd=extract(Rs,Rtt)`	`width=zxt`$_{6->32}$`((Rtt.w[1]));`
	`offset=sxt`$_{7->32}$`((Rtt.w[0]));`
	`Rd = sxt`$_{width->64}$`((offset>0)?(zxt`$_{32->64}$`(zxt`$_{32->64}$`(Rs))`
	`>>>offset):(zxt`$_{32->64}$`(zxt`$_{32->64}$`(Rs)) <<offset));`

`Rd=extractu(Rs,#u5,#U5)`	`width=#u;`
	`offset=#U;`
	`Rd = zxt`$_{width->32}$`((Rs >> offset));`

`Rd=extractu(Rs,Rtt)`	`width=zxt`$_{6->32}$`((Rtt.w[1]));`
	`offset=sxt`$_{7->32}$`((Rtt.w[0]));`
	`Rd = zxt`$_{width->64}$`((offset>0)?(zxt`$_{32->64}$`(zxt`$_{32->64}$`(Rs))`
	`>>> offset):(zxt`$_{32->64}$`(zxt`$_{32->64}$`(Rs)) <<offset));`

`Rdd=extract(Rss,#u6,#U6)`	`width=#u;`
	`offset=#U;`
	`Rdd = sxt`$_{width->64}$`((Rss >> offset));`

Syntax	Behavior

Rdd=extract(Rss,Rtt)	width=zxt$_{6->32}$((Rtt.w[1]));
	offset=sxt$_{7->32}$((Rtt.w[0]));
	Rdd = sxt$_{width->64}$((offset>0)?(Rss >>> offset):(Rss << offset));

Rdd=extractu(Rss,#u6,#U6)	width=#u;
	offset=#U;
	Rdd = zxt$_{width->64}$((Rss >> offset));

Rdd=extractu(Rss,Rtt)	width=zxt$_{6->32}$((Rtt.w[1]));
	offset=sxt$_{7->32}$((Rtt.w[0]));
	Rdd = zxt$_{width->64}$((offset > 0)?(Rss >>> offset):(Rss << offset));

Class: XTYPE (slots 2,3)

Intrinsics

Rd=extract(Rs,#u5,#U5)	Word32 Q6_R_extract_RII(Word32 Rs, Word32 Iu5, Word32 IU5)
Rd=extract(Rs,Rtt)	Word32 Q6_R_extract_RP(Word32 Rs, Word64 Rtt)
Rd=extractu(Rs,#u5,#U5)	Word32 Q6_R_extractu_RII(Word32 Rs, Word32 Iu5, Word32 IU5)
Rd=extractu(Rs,Rtt)	Word32 Q6_R_extractu_RP(Word32 Rs, Word64 Rtt)
Rdd=extract(Rss,#u6,#U6)	Word64 Q6_P_extract_PII(Word64 Rss, Word32 Iu6, Word32 IU6)
Rdd=extract(Rss,Rtt)	Word64 Q6_P_extract_PP(Word64 Rss, Word64 Rtt)
Rdd=extractu(Rss,#u6,#U6)	Word64 Q6_P_extractu_PII(Word64 Rss, Word32 Iu6, Word32 IU6)
Rdd=extractu(Rss,Rtt)	Word64 Q6_P_extractu_PP(Word64 Rss, Word64 Rtt)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	1	l	l	l	s	s	s	s	s	P	P	i	i	i	i	i	i	l	l	l	d	d	d	d	d	Rdd=extractu(Rss,#u6,#U6)
1	0	0	0	1	0	1	0	l	l	l	s	s	s	s	s	P	P	i	i	i	i	i	i	l	l	l	d	d	d	d	d	Rdd=extract(Rss,#u6,#U6)
1	0	0	0	1	1	0	1	0	l	l	s	s	s	s	s	P	P	0	i	i	i	i	i	l	l	l	d	d	d	d	d	Rd=extractu(Rs,#u5,#U5)
1	0	0	0	1	1	0	1	1	l	l	s	s	s	s	s	P	P	0	i	i	i	i	i	l	l	l	d	d	d	d	d	Rd=extract(Rs,#u5,#U5)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	0	0	0	0	0	1	0	0	-	s	s	s	s	s	P	P	-	t	t	t	t	t	0	0	-	d	d	d	d	d	Rdd=extractu(Rss,Rtt)
1	1	0	0	0	0	0	1	1	1	-	s	s	s	s	s	P	P	-	t	t	t	t	t	1	0	-	d	d	d	d	d	Rdd=extract(Rss,Rtt)
1	1	0	0	1	0	0	1	0	0	-	s	s	s	s	s	P	P	-	t	t	t	t	t	0	0	-	d	d	d	d	d	Rd=extractu(Rs,Rtt)
1	1	0	0	1	0	0	1	0	0	-	s	s	s	s	s	P	P	-	t	t	t	t	t	0	1	-	d	d	d	d	d	Rd=extract(Rs,Rtt)

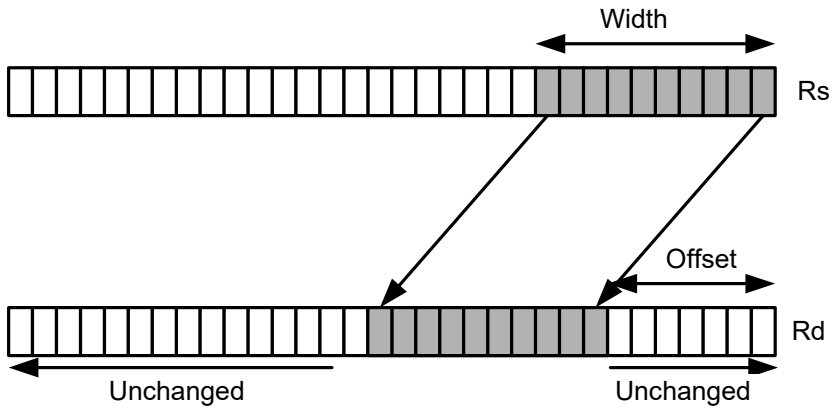
Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

Insert bit field

Replace a bit field in the destination register (or register pair) with bits from the least significant portion of Rs/Rss. The number of bits is obtained from the first immediate or the most-significant word of Rtt. The bits are shifted by the second immediate or the least significant word of Rtt.

If register Rtt specifies the offset, the low 7-bits of Rtt are treated as a signed 7-bit value. If this value is negative, the result is zero.

Shift amounts and offsets that are too large may push bits beyond the end of the destination register, in this case the bits do not appear in the destination register.



Syntax	Behavior

`Rx=insert(Rs,#u5,#U5)`	`width=#u;`
	`offset=#U;`
	`Rx &= ~(((1<<width)-1)<<offset);`
	`Rx |= ((Rs & ((1<<width)-1)) << offset);`

`Rx=insert(Rs,Rtt)`	`width=zxt`$_{6->32}$`((Rtt.w[1]));`
	`offset=sxt`$_{7->32}$`((Rtt.w[0]));`
	`mask = ((1<<width)-1);`
	`if (offset < 0) {`
	`  Rx = 0;`
	`} else {`
	`  Rx &= ~(mask<<offset);`
	`  Rx |= ((Rs & mask) << offset);`
	`}`

`Rxx=insert(Rss,#u6,#U6)`	`width=#u;`
	`offset=#U;`
	`Rxx &= ~(((1<<width)-1)<<offset);`
	`Rxx |= ((Rss & ((1<<width)-1)) << offset);`

Syntax	Behavior

`Rxx=insert(Rss,Rtt)`	`width=zxt6->32((Rtt.w[1]));`
	`offset=sxt7->32((Rtt.w[0]));`
	`mask = ((1<<width)-1);`
	`if (offset < 0) {`
	`  Rxx = 0;`
	`} else {`
	`  Rxx &= ~(mask<<offset);`
	`  Rxx |= ((Rss & mask) << offset);`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

`Rx=insert(Rs,#u5,#U5)`	`Word32 Q6_R_insert_RII(Word32 Rx, Word32 Rs, Word32 Iu5, Word32 IU5)`
`Rx=insert(Rs,Rtt)`	`Word32 Q6_R_insert_RP(Word32 Rx, Word32 Rs, Word64 Rtt)`
`Rxx=insert(Rss,#u6,#U6)`	`Word64 Q6_P_insert_PII(Word64 Rxx, Word64 Rss, Word32 Iu6, Word32 IU6)`
`Rxx=insert(Rss,Rtt)`	`Word64 Q6_P_insert_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			x5					
1	0	0	0	0	0	1	1	**I**	**I**	**I**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**I**	**I**	**I**	**x**	**x**	**x**	**x**	**x**	Rxx=insert(Rss,#u6,#U6)
1	0	0	0	1	1	1	1	0	**I**	**I**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**I**	**I**	**I**	**x**	**x**	**x**	**x**	**x**	Rx=insert(Rs,#u5,#U5)
ICLASS				RegType							s5					Parse			t5								x5					
1	1	0	0	1	0	0	0	-	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**x**	**x**	**x**	**x**	**x**	Rx=insert(Rs,Rtt)
ICLASS				RegType				MajOp			s5					Parse			t5								x5					
1	1	0	0	1	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	-	-	-	**x**	**x**	**x**	**x**	**x**	Rxx=insert(Rss,Rtt)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

Interleave/deinterleave

For `interleave`, bits I+32 of Rss (which are the bits from the upper source word) get placed in the odd bits (I*2)+1 of Rdd, while bits I of Rss (which are the bits from the lower source word) get placed in the even bits (I*2) of Rdd.

For `deinterleave`, the even bits of the source register are placed in the even register of the result pair, and the odd bits of the source register are placed in the odd register of the result pair.

r1:0 = `deinterleave`(r1:0) is the inverse of r1:0 = `interleave`(r1:0).

Syntax	Behavior
`Rdd=deinterleave(Rss)`	`Rdd = deinterleave(ODD,EVEN);`
`Rdd=interleave(Rss)`	`Rdd = interleave(Rss.w[1],Rss.w[0]);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=deinterleave(Rss)`	`Word64 Q6_P_deinterleave_P(Word64 Rss)`
`Rdd=interleave(Rss)`	`Word64 Q6_P_interleave_P(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=deinterleave(Rss)
1	0	0	0	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=interleave(Rss)

Field name	Description
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major opcode
`MinOp`	Minor opcode
`RegType`	Register type

Linear feedback-shift iteration

Count the number of ones of the logical AND of the two source input values, and take the least significant value of that sum. The first source value is shifted right by one bit, and the parity is placed in the MSB.

Syntax	Behavior
`Rdd=lfs(Rss,Rtt)`	`Rdd = (Rss.u64 >> 1) | ((1&count_ones(Rss & Rtt)).u64<<63) ;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=lfs(Rss,Rtt)`	`Word64 Q6_P_lfs_PP(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	0	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=lfs(Rss,Rtt)

Field name	Description
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`Maj`	Major opcode
`Min`	Minor opcode
`RegType`	Register type

Masked parity

Count the number of ones of the logical AND of the two source input values, and take the least significant bit of that sum.

Syntax	Behavior
`Rd=parity(Rs,Rt)`	`Rd = 1&count_ones(Rs & Rt);`
`Rd=parity(Rss,Rtt)`	`Rd = 1&count_ones(Rss & Rtt);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=parity(Rs,Rt)`	`Word32 Q6_R_parity_RR(Word32 Rs, Word32 Rt)`
`Rd=parity(Rss,Rtt)`	`Word32 Q6_R_parity_PP(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5								d5					
1	1	0	1	0	0	0	0	-	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=parity(Rss,Rtt)
1	1	0	1	0	1	0	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=parity(Rs,Rt)

Field name	Description
`RegType`	Register type
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Bit reverse

Reverse the order of bits. The most significant gets swapped with the least significant, bit 30 gets swapped with bit 1, and so on.

Syntax	Behavior
`Rd=brev(Rs)`	`Rd = reverse_bits(Rs);`
`Rdd=brev(Rss)`	`Rdd = reverse_bits(Rss);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=brev(Rs)`	`Word32 Q6_R_brev_R(Word32 Rs)`
`Rdd=brev(Rss)`	`Word64 Q6_P_brev_P(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=brev(Rss)
1	0	0	0	1	1	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=brev(Rs)

Field name	Description
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major opcode
`MinOp`	Minor opcode
`RegType`	Register type

Set/clear/toggle bit

Set (to 1), clear (to 0), or toggle a single bit in the source, and place the resulting value in the destination. The bit to manipulate can be indicated using an immediate or register value.

If a register is used to indicate the bit position, and the value of the least-significant seven bits of Rt is out of range, the destination register is unchanged.

Syntax	Behavior
`Rd=clrbit(Rs,#u5)`	`Rd = (Rs & (~(1<<#u)));`
`Rd=clrbit(Rs,Rt)`	`Rd = (Rs & (~((sxt7->32(Rt)>0)?(zxt32->64(1)<<sxt7->32(Rt)):(zxt32->64(1)>>>sxt7->32(Rt)))));`
`Rd=setbit(Rs,#u5)`	`Rd = (Rs | (1<<#u));`
`Rd=setbit(Rs,Rt)`	`Rd = (Rs | (sxt7->32(Rt)>0)?(zxt32->64(1)<<sxt7->32(Rt)):(zxt32->64(1)>>>sxt7->32(Rt)));`
`Rd=togglebit(Rs,#u5)`	`Rd = (Rs ^ (1<<#u));`
`Rd=togglebit(Rs,Rt)`	`Rd = (Rs ^ (sxt7->32(Rt)>0)?(zxt32->64(1)<<sxt7->32(Rt)):(zxt32->64(1)>>>sxt7->32(Rt)));`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=clrbit(Rs,#u5)`	`Word32 Q6_R_clrbit_RI(Word32 Rs, Word32 Iu5)`
`Rd=clrbit(Rs,Rt)`	`Word32 Q6_R_clrbit_RR(Word32 Rs, Word32 Rt)`
`Rd=setbit(Rs,#u5)`	`Word32 Q6_R_setbit_RI(Word32 Rs, Word32 Iu5)`
`Rd=setbit(Rs,Rt)`	`Word32 Q6_R_setbit_RR(Word32 Rs, Word32 Rt)`
`Rd=togglebit(Rs,#u5)`	`Word32 Q6_R_togglebit_RI(Word32 Rs, Word32 Iu5)`
`Rd=togglebit(Rs,Rt)`	`Word32 Q6_R_togglebit_RR(Word32 Rs, Word32 Rt)`

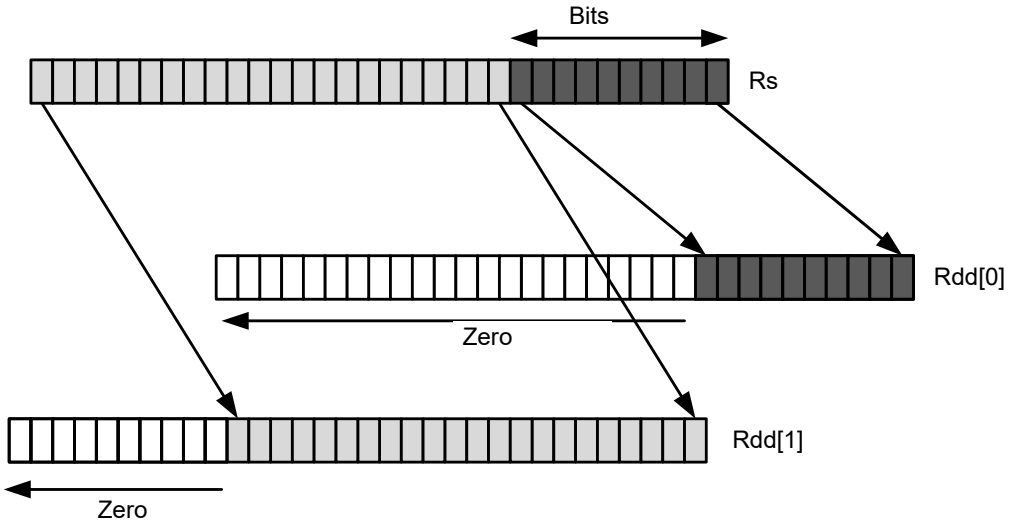
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	1	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=setbit(Rs,#u5)
1	0	0	0	1	1	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=clrbit(Rs,#u5)
1	0	0	0	1	1	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=togglebit(Rs,#u5)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	0	0	0	1	1	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rd=setbit(Rs,Rt)
1	1	0	0	0	1	1	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rd=clrbit(Rs,Rt)
1	1	0	0	0	1	1	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rd=togglebit(Rs,Rt)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

Split bit field

Split the bit field in a register into upper and lower parts of variable size. The lower part is placed in the lower word of a destination register pair, and the upper part is placed in the upper word of the destination. An immediate value or register Rt is used to determine the bit position of the split.



Syntax	Behavior
`Rdd=bitsplit(Rs,#u5)`	`Rdd.w[1]=(Rs>>#u);` `Rdd.w[0]=zxt`$_{\#u->32}$`(Rs);`

`Rdd=bitsplit(Rs,Rt)`	`shamt = zxt`$_{5->32}$`(Rt);`
	`Rdd.w[1]=(Rs>>shamt);`
	`Rdd.w[0]=zxt`$_{shamt->32}$`(Rs);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=bitsplit(Rs,#u5)`	`Word64 Q6_P_bitsplit_RI(Word32 Rs, Word32 Iu5)`
`Rdd=bitsplit(Rs,Rt)`	`Word64 Q6_P_bitsplit_RR(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	1	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=bitsplit(Rs,#u5)
ICLASS				RegType							s5					Parse			t5								d5					
1	1	0	1	0	1	0	0	-	-	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rdd=bitsplit(Rs,Rt)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

Table index

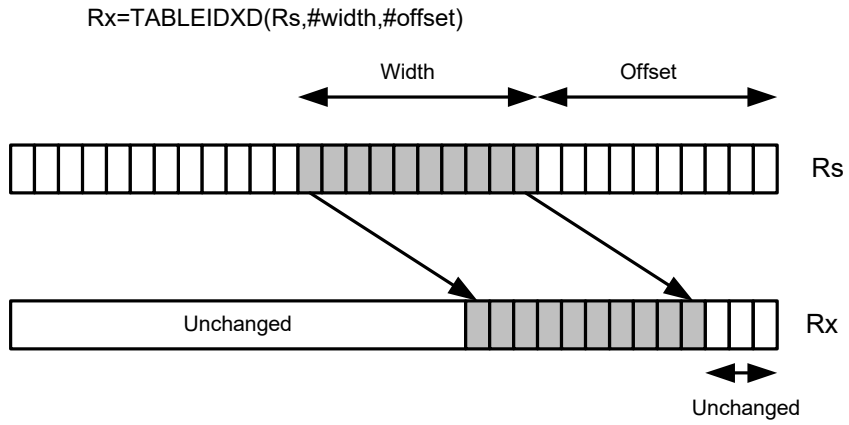
The table index instruction supports fast lookup tables where the index into the table is stored in a bit-field. The instruction forms the address of a table element by extracting the bit-field and inserting it into the appropriate bits of a pointer to the table element.

Tables are defined to contain entries of bytes, halfwords, words, or doublewords. The table must be aligned to a power-of-2 size greater than or equal to the table size. For example, a 4 KB table should be aligned to a 4 KB boundary. This instruction supports tables with a maximum of 32 K table entries.

Register Rx contains a pointer to within the table. Register Rs contains a field to be extracted and used as a table index. This instruction first extracts the field from register Rs and then inserts it into register Rx. The insertion point is bit 0 for tables of bytes, bit 1 for tables of halfwords, bit 2 for tables of words, and bit 3 for tables of doublewords.

In the assembly syntax, the width and offset values represent the field in Rs to be extracted. Unsigned constants should be used to specify the width and offsets in assembly. In the encoded instruction, however, these values are adjusted by the assembler as follows.

- For `tableidxb`, no adjustment is necessary.
- For `tableidxh`, the assembler encodes offset-1 in the signed immediate field.
- For `tableidxw`, the assembler encodes offset-2 in the signed immediate field.
- For `tableidxd`, the assembler encodes offset-3 in the signed immediate field.



Syntax	Behavior

`Rx=tableidxb(Rs,#u4,#S6):raw`	`width=#u;`
	`offset=#S;`
	`field = Rs[(width+offset-1):offset];`
	`Rx[(width-1+0):0]=field;`

`Rx=tableidxb(Rs,#u4,#U5)`	`Assembler mapped to:` `"Rx=tableidxb(Rs,#u4,#U5):raw"`

`Rx=tableidxd(Rs,#u4,#S6):raw`	`width=#u;`
	`offset=#S+3;`
	`field = Rs[(width+offset-1):offset];`
	`Rx[(width-1+3):3]=field;`

Syntax	Behavior
`Rx=tableidxd(Rs,#u4,#U5)`	`Assembler mapped to: "Rx=tableidxd(Rs,#u4,#U5-3):raw"`
`Rx=tableidxh(Rs,#u4,#S6):raw`	`width=#u; offset=#S+1; field = Rs[(width+offset-1):offset]; Rx[(width-1+1):1]=field;`
`Rx=tableidxh(Rs,#u4,#U5)`	`Assembler mapped to: "Rx=tableidxh(Rs,#u4,#U5-1):raw"`
`Rx=tableidxw(Rs,#u4,#S6):raw`	`width=#u; offset=#S+2; field = Rs[(width+offset-1):offset]; Rx[(width-1+2):2]=field;`
`Rx=tableidxw(Rs,#u4,#U5)`	`Assembler mapped to: "Rx=tableidxw(Rs,#u4,#U5-2):raw"`

Class: XTYPE (slots 2,3)

Intrinsics

`Rx=tableidxb(Rs,#u4,#U5)`	`Word32 Q6_R_tableidxb_RII(Word32 Rx, Word32 Rs, Word32 Iu4, Word32 IU5)`
`Rx=tableidxd(Rs,#u4,#U5)`	`Word32 Q6_R_tableidxd_RII(Word32 Rx, Word32 Rs, Word32 Iu4, Word32 IU5)`
`Rx=tableidxh(Rs,#u4,#U5)`	`Word32 Q6_R_tableidxh_RII(Word32 Rx, Word32 Rs, Word32 Iu4, Word32 IU5)`
`Rx=tableidxw(Rs,#u4,#U5)`	`Word32 Q6_R_tableidxw_RII(Word32 Rx, Word32 Rs, Word32 Iu4, Word32 IU5)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			x5					
1	0	0	0	0	1	1	1	0	0	i	s	s	s	s	s	P	P	I	I	I	I	I	I	i	i	i	x	x	x	x	x	Rx=tableidxb(Rs,#u4,#S6):raw
1	0	0	0	0	1	1	1	0	1	i	s	s	s	s	s	P	P	I	I	I	I	I	I	i	i	i	x	x	x	x	x	Rx=tableidxh(Rs,#u4,#S6):raw
1	0	0	0	0	1	1	1	1	0	i	s	s	s	s	s	P	P	I	I	I	I	I	I	i	i	i	x	x	x	x	x	Rx=tableidxw(Rs,#u4,#S6):raw
1	0	0	0	0	1	1	1	1	1	i	s	s	s	s	s	P	P	I	I	I	I	I	I	i	i	i	x	x	x	x	x	Rx=tableidxd(Rs,#u4,#S6):raw

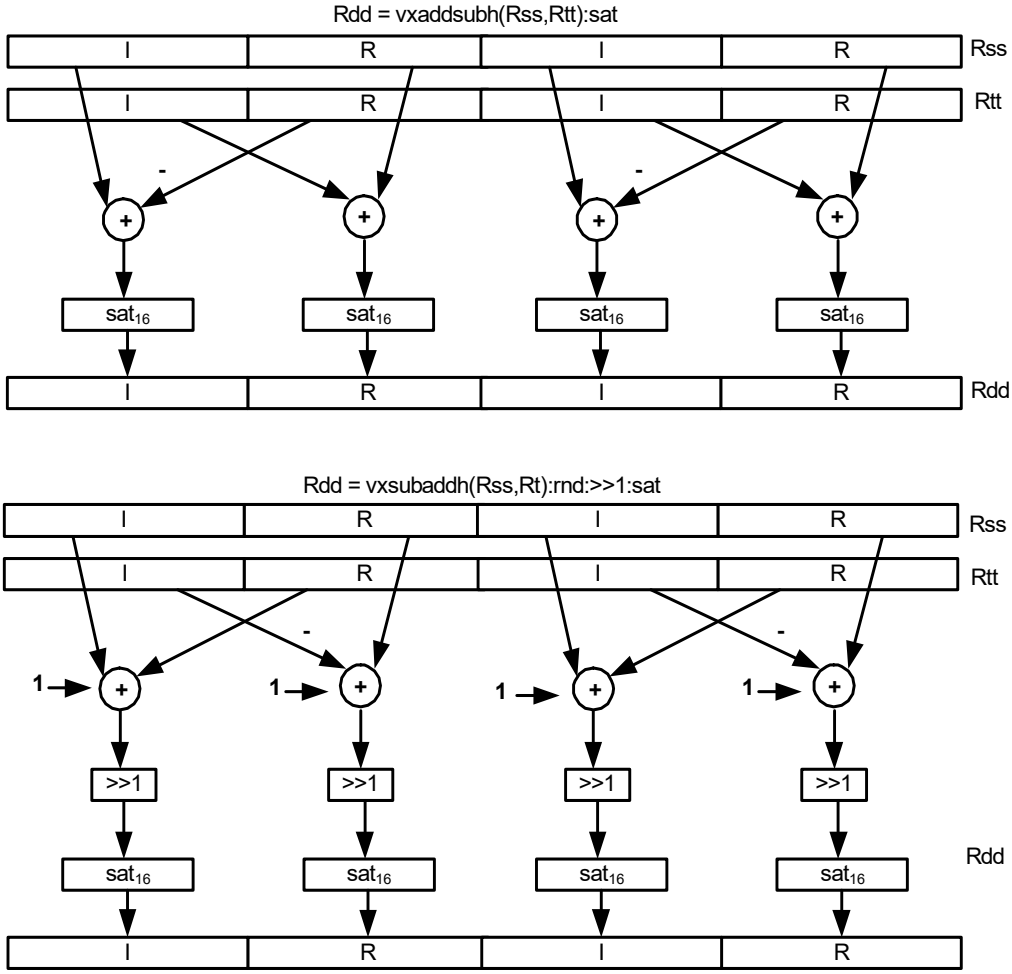
Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
s5	Field to encode register s
x5	Field to encode register x
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

11.10.3 XTYPE complex

The XTYPE complex instruction subclass includes instructions that are for complex math, using imaginary values.

Complex add/sub halfwords

Cross vector add-sub or sub-add used to perform X + jY and X - jY complex operations. Each 16-bit result is saturated to 16-bits.



Syntax	Behavior

`Rdd = vxaddsubh(Rss, Rtt):rnd:>>1:sat`	`Rdd.h[0]=sat16((Rss.h[0]+Rtt.h[1]+1)>>1);`
	`Rdd.h[1]=sat16((Rss.h[1]-Rtt.h[0]+1)>>1);`
	`Rdd.h[2]=sat16((Rss.h[2]+Rtt.h[3]+1)>>1);`
	`Rdd.h[3]=sat16((Rss.h[3]-Rtt.h[2]+1)>>1);`

`Rdd = vxaddsubh(Rss,Rtt):sat`	`Rdd.h[0]=sat16(Rss.h[0]+Rtt.h[1]);`
	`Rdd.h[1]=sat16(Rss.h[1]-Rtt.h[0]);`
	`Rdd.h[2]=ssat16(Rss.h[2]+Rtt.h[3]);`
	`Rdd.h[3]=sat16(Rss.h[3]-Rtt.h[2]);`

Syntax	Behavior

`Rdd = vxsubaddh(Rss, Rtt):rnd: >>1 :sat`	`Rdd.h[0]=sat16((Rss.h[0]-Rtt.h[1]+1)>>1);`
	`Rdd.h[1]=sat16((Rss.h[1]+Rtt.h[0]+1)>>1);`
	`Rdd.h[2]=sat16(Rss.h[2]-Rtt.h[3]+1)>>1);`
	`Rdd.h[3]=sat16((Rss.h[3]+Rtt.h[2]+1)>>1);`

`Rdd = vxsubaddh(Rss,Rtt):sat`	`Rdd.h[0]=ssat16(Rss.h[0]-Rtt.h[1]);`
	`Rdd.h[1]=sat16(Rss.h[1]+Rtt.h[0]);`
	`Rdd.h[2]=sat16(Rss.h[2]-Rtt.h[3]);`
	`Rdd.h[3]=sat16(Rss.h[3]+Rtt.h[2]);`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd= vxaddsubh(Rss, Rtt):rnd:>>1:sat`	`Word64 Q6_P_vxaddsubh_PP_rnd_rs1_sat(Word64 Rss, Word64 Rtt)`
`Rdd = vxaddsubh(Rss, Rtt):sat`	`Word64 Q6_P_vxaddsubh_PP_sat(Word64 Rss, Word64 Rtt)`
`Rdd= vxsubaddh(Rss, Rtt):rnd:>>1:sat`	`Word64 Q6_P_vxsubaddh_PP_rnd_rs1_sat(Word64 Rss, Word64 Rtt)`
`Rdd = vxsubaddh(Rss, Rtt):sat`	`Word64 Q6_P_vxsubaddh_PP_sat(Word64 Rss, Word64 Rtt)`

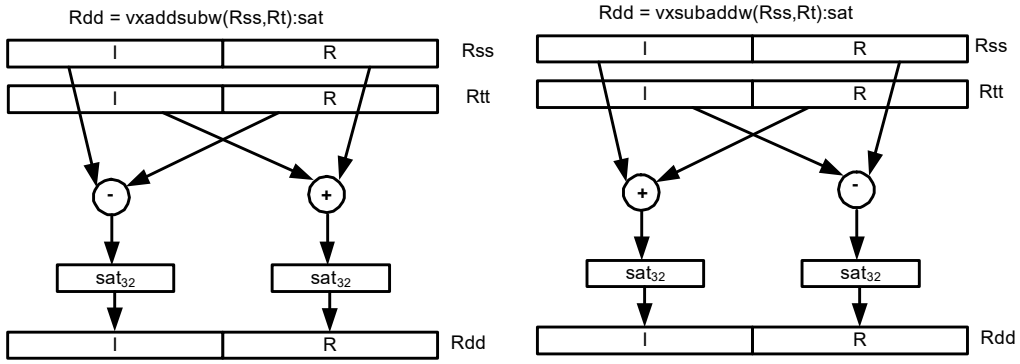
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	0	1	0	1	-	s	s	s	s	s	P	P	-	t	t	t	t	t	1	0	0	d	d	d	d	d	Rdd=vxaddsubh(Rss,Rtt):sat
1	1	0	0	0	0	0	1	0	1	-	s	s	s	s	s	P	P	-	t	t	t	t	t	1	1	0	d	d	d	d	d	Rdd=vxsubaddh(Rss,Rtt):sat
1	1	0	0	0	0	0	1	1	1	-	s	s	s	s	s	P	P	-	t	t	t	t	t	0	0	-	d	d	d	d	d	Rdd=vxaddsubh(Rss,Rtt):rnd:>>1:sat
1	1	0	0	0	0	0	1	1	1	-	s	s	s	s	s	P	P	-	t	t	t	t	t	0	1	-	d	d	d	d	d	Rdd=vxsubaddh(Rss,Rtt):rnd:>>1:sat

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
Maj	Major opcode
Min	Minor opcode
RegType	Register type

Complex add/sub words

Cross vector add-sub or sub-add used to perform X + jY and X - jY complex operations. Each 32-bit result is saturated to 32-bits.



Syntax	Behavior
`Rdd=vxaddsubw(Rss,Rtt):sat`	`Rdd.w[0]=sat32(Rss.w[0]+Rtt.w[1]);` `Rdd.w[1]=sat32(Rss.w[1]-Rtt.w[0]);`
`Rdd=vxsubaddw(Rss,Rtt):sat`	`Rdd.w[0]=sat32(Rss.w[0]-Rtt.w[1]);` `Rdd.w[1]=sat32(Rss.w[1]+Rtt.w[0]);`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vxaddsubw(Rss,Rtt):sat`	`Word64 Q6_P_vxaddsubw_PP_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vxsubaddw(Rss,Rtt):sat`	`Word64 Q6_P_vxsubaddw_PP_sat(Word64 Rss, Word64 Rtt)`

Encoding

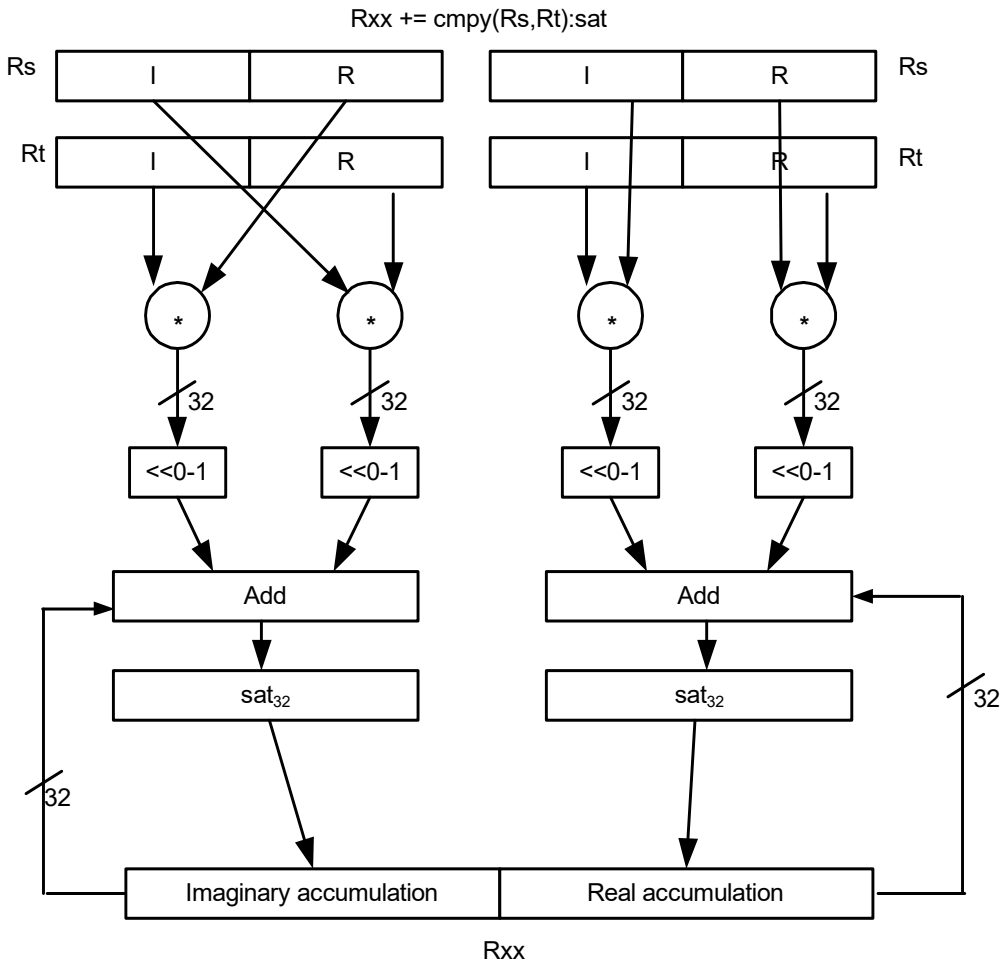
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	0	1	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vxaddsubw(Rss,Rtt):sat
1	1	0	0	0	0	0	1	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vxsubaddw(Rss,Rtt):sat

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
Maj	Major opcode
Min	Minor opcode
RegType	Register type

Complex multiply

Multiply complex values Rs and Rt. The inputs have a real 16-bit value in the low halfword and an imaginary 16-bit value in the high halfword. Optionally, scale the result by 0 to 1 bits. Optionally, add a complex accumulator. Saturate the real and imaginary portions to 32-bits. The output has a real 32-bit value in the low word and an imaginary 32-bit value in the high word. The Rt input can be optionally conjugated.

Another option is that the result can be subtracted from the destination rather than accumulated.



Syntax	Behavior
`Rdd=cmpy(Rs,Rt)[:<<1]:sat`	`Rdd.w[1]=sat32((Rs.h[1] * Rt.h[0])[<<1] + (Rs.h[0] * Rt.h[1])[<<1]);` `Rdd.w[0]=sat32((Rs.h[0] * Rt.h[0])[<<1] - (Rs.h[1] * Rt.h[1])[<<1]);`
`Rdd=cmpy(Rs,Rt*)[:<<1]:sat`	`Rdd.w[1]=sat32((Rs.h[1] * Rt.h[0])[<<1] - (Rs.h[0] * Rt.h[1])[<<1]);` `Rdd.w[0]=sat32((Rs.h[0] * Rt.h[0])[<<1] + (Rs.h[1] * Rt.h[1])[<<1]);`

Syntax	Behavior
`Rxx+=cmpy(Rs,Rt)[:<<1]:sat`	`Rxx.w[1]=sat32(Rxx.w[1] + (Rs.h[1] * Rt.h[0])[<<1] + (Rs.h[0] * Rt.h[1])[<<1]); Rxx.w[0]=sat32(Rxx.w[0] + (Rs.h[0] * Rt.h[0])[<<1] - (Rs.h[1] * Rt.h[1])[<<1]);`
`Rxx+=cmpy(Rs,Rt*)[:<<1]:sat`	`Rxx.w[1]=sat32(Rxx.w[1] + (Rs.h[1] * Rt.h[0])[<<1] - (Rs.h[0] * Rt.h[1])[<<1]); Rxx.w[0]=sat32(Rxx.w[0] + (Rs.h[0] * Rt.h[0])[<<1] + (Rs.h[1] * Rt.h[1])[<<1]);`
`Rxx-=cmpy(Rs,Rt)[:<<1]:sat`	`Rxx.w[1]=sat32(Rxx.w[1] - ((Rs.h[1] * Rt.h[0])[<<1] + (Rs.h[0] * Rt.h[1])[<<1])); Rxx.w[0]=sat32(Rxx.w[0] - ((Rs.h[0] * Rt.h[0])[<<1] - (Rs.h[1] * Rt.h[1])[<<1]));`
`Rxx-=cmpy(Rs,Rt*)[:<<1]:sat`	`Rxx.w[1]=sat32(Rxx.w[1] - ((Rs.h[1] * Rt.h[0])[<<1] - (Rs.h[0] * Rt.h[1])[<<1])); Rxx.w[0]=sat32(Rxx.w[0] - ((Rs.h[0] * Rt.h[0])[<<1] + (Rs.h[1] * Rt.h[1])[<<1]));`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=cmpy(Rs,Rt):<<1:sat`	`Word64 Q6_P_cmpy_RR_s1_sat(Word32 Rs, Word32 Rt)`
`Rdd=cmpy(Rs,Rt):sat`	`Word64 Q6_P_cmpy_RR_sat(Word32 Rs, Word32 Rt)`
`Rdd=cmpy(Rs,Rt*):<<1:sat`	`Word64 Q6_P_cmpy_RR_conj_s1_sat(Word32 Rs, Word32 Rt)`
`Rdd=cmpy(Rs,Rt*):sat`	`Word64 Q6_P_cmpy_RR_conj_sat(Word32 Rs, Word32 Rt)`
`Rxx+=cmpy(Rs,Rt):<<1:sat`	`Word64 Q6_P_cmpyacc_RR_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=cmpy(Rs,Rt):sat`	`Word64 Q6_P_cmpyacc_RR_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=cmpy(Rs,Rt*):<<1:sat`	`Word64 Q6_P_cmpyacc_RR_conj_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=cmpy(Rs,Rt*):sat`	`Word64 Q6_P_cmpyacc_RR_conj_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=cmpy(Rs,Rt):<<1:sat`	`Word64 Q6_P_cmpynac_RR_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=cmpy(Rs,Rt):sat`	`Word64 Q6_P_cmpynac_RR_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`

`Rxx-=cmpy(Rs,Rt*):<<1:sat`	`Word64 Q6_P_cmpynac_RR_conj_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=cmpy(Rs,Rt*):sat`	`Word64 Q6_P_cmpynac_RR_conj_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`

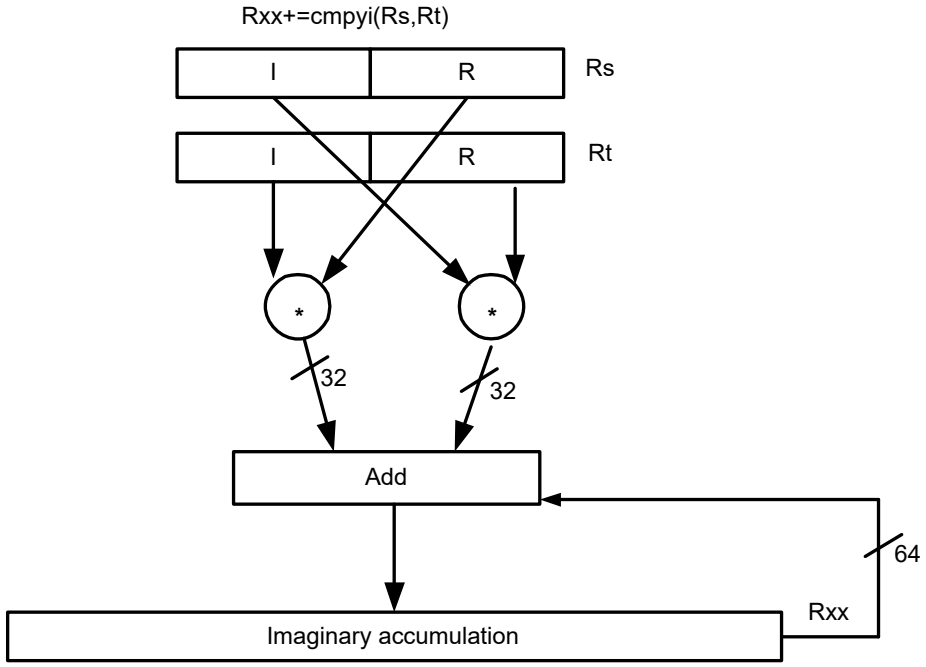
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	0	1	0	1	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=cmpy(Rs,Rt)[:<<N]:sat
1	1	1	0	0	1	0	1	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=cmpy(Rs,Rt*)[:<<N]:sat
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	0	1	1	1	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=cmpy(Rs,Rt)[:<<N]:sat
1	1	1	0	0	1	1	1	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rxx-=cmpy(Rs,Rt)[:<<N]:sat
1	1	1	0	0	1	1	1	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=cmpy(Rs,Rt*)[:<<N]:sat
1	1	1	0	0	1	1	1	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rxx-=cmpy(Rs,Rt*)[:<<N]:sat

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Complex multiply real or imaginary

Multiply complex values Rs and Rt. The inputs have a real 16-bit value in the low halfword and an imaginary 16-bit value in the high halfword. Take either the real or imaginary result and optionally accumulate with a 64-bit destination.



Syntax	Behavior
`Rdd=cmpyi(Rs,Rt)`	`Rdd = (Rs.h[1] * Rt.h[0]) + (Rs.h[0] * Rt.h[1]);`
`Rdd=cmpyr(Rs,Rt)`	`Rdd = (Rs.h[0] * Rt.h[0]) - (Rs.h[1] * Rt.h[1]);`
`Rxx+=cmpyi(Rs,Rt)`	`Rxx = Rxx + (Rs.h[1] * Rt.h[0]) + (Rs.h[0] * Rt.h[1]);`
`Rxx+=cmpyr(Rs,Rt)`	`Rxx = Rxx + (Rs.h[0] * Rt.h[0]) - (Rs.h[1] * Rt.h[1]);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=cmpyi(Rs,Rt)`	`Word64 Q6_P_cmpyi_RR(Word32 Rs, Word32 Rt)`
`Rdd=cmpyr(Rs,Rt)`	`Word64 Q6_P_cmpyr_RR(Word32 Rs, Word32 Rt)`
`Rxx+=cmpyi(Rs,Rt)`	`Word64 Q6_P_cmpyiacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=cmpyr(Rs,Rt)`	`Word64 Q6_P_cmpyracc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`

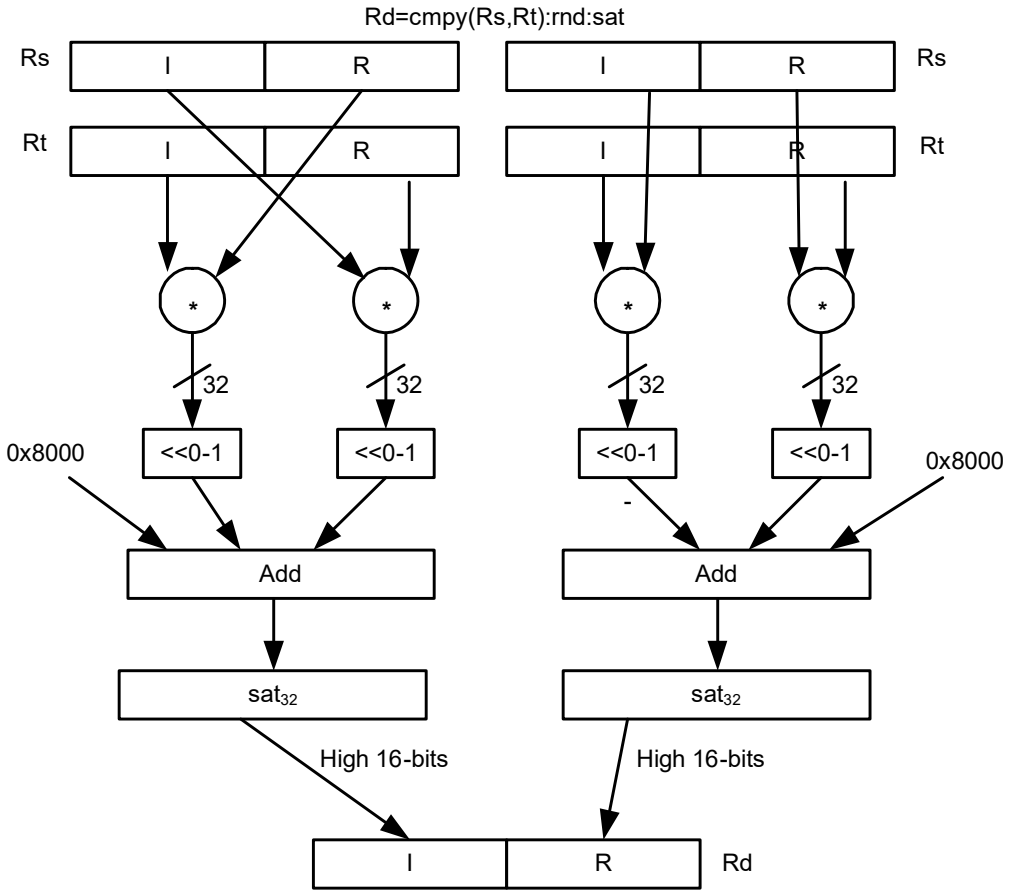
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	0	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=cmpyi(Rs,Rt)
1	1	1	0	0	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=cmpyr(Rs,Rt)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	0	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=cmpyi(Rs,Rt)
1	1	1	0	0	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=cmpyr(Rs,Rt)

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Complex multiply with round and pack

Multiply complex values Rs and Rt. The inputs have a real 16-bit value in the low halfword and an imaginary 16-bit value in the high halfword. The Rt input is optionally conjugated. The multiplier results are optionally scaled by 0 to 1 bits. A rounding constant is added to each real and imaginary sum. The real and imaginary parts are individually saturated to 32 bits. The upper 16 bits of each 32-bit result are packed in a 32-bit destination register.



Syntax	Behavior
`Rd = cmpy(Rs,Rt)[: << 1]:rnd:sat`	`Rd.h[1]=(sat32((Rs.h[1] * Rt.h[0])[<<1] + (Rs.h[0] * Rt.h[1])[<<1] + 0x8000)).h[1];` `Rd.h[0]=(sat32((Rs.h[0] * Rt.h[0])[<<1] - (Rs.h[1] * Rt.h[1])[<<1] + 0x8000)).h[1];`
`Rd = cmpy(Rs,Rt*)[:<< 1]:rnd:sat`	`Rd.h[1]=(sat32((Rs.h[1] * Rt.h[0])[<<1] - (Rs.h[0] * Rt.h[1])[<<1] + 0x8000)).h[1];` `Rd.h[0]=(sat32(Rs.h[0] * Rt.h[0])[<<1] + (Rs.h[1] * Rt.h[1])[<<1] + 0x8000)).h[1];`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=cmpy(Rs,Rt):<<1:rnd:sat`	`Word32 Q6_R_cmpy_RR_s1_rnd_sat(Word32 Rs, Word32 Rt)`
`Rd=cmpy(Rs,Rt):rnd:sat`	`Word32 Q6_R_cmpy_RR_rnd_sat(Word32 Rs, Word32 Rt)`
`Rd=cmpy(Rs,Rt*):<<1:rnd:sat`	`Word32 Q6_R_cmpy_RR_conj_s1_rnd_sat(Word32 Rs, Word32 Rt)`
`Rd=cmpy(Rs,Rt*):rnd:sat`	`Word32 Q6_R_cmpy_RR_conj_rnd_sat(Word32 Rs, Word32 Rt)`

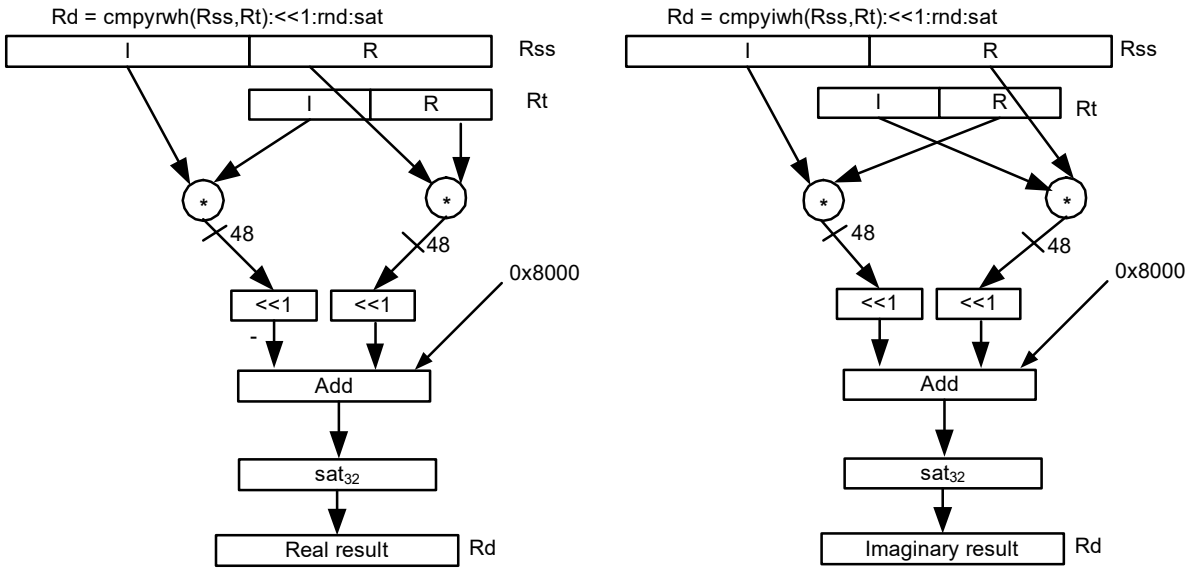
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	1	0	1	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=cmpy(Rs,Rt)[:<<N]:rnd:sat
1	1	1	0	1	1	0	1	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=cmpy(Rs,Rt*)[:<<N]:rnd:sat

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Complex multiply 32x16

Multiply 32 by 16 bit complex values Rss and Rt. The inputs have a real value in the low part of a register and the imaginary value in the upper part. The multiplier results are scaled by 1 bit and accumulated with a rounding constant. The result is saturated to 32 bits.



Syntax	Behavior
`Rd = cmpyiwh(Rss, Rt):<<1:rnd:sat`	`Rd = sat32(( (Rss.w[0] * Rt.h[1]) + (Rss.w[1] * Rt.h[0]) + 0x4000)>>15);`
`Rd = cmpyiwh(Rss, Rt*):<<1:rnd:sat`	`Rd = sat32(( (Rss.w[1] * Rt.h[0]) - (Rss.w[0] * Rt.h[1]) + 0x4000)>>15);`
`Rd = cmpyrwh(Rss, Rt):<<1:rnd:sat`	`Rd = sat32(( (Rss.w[0] * Rt.h[0]) - (Rss.w[1] * Rt.h[1]) + 0x4000)>>15);`
`Rd = cmpyrwh(Rss, Rt*):<<1:rnd:sat`	`Rd = sat32(( (Rss.w[0] * Rt.h[0]) + (Rss.w[1] * Rt.h[1]) + 0x4000)>>15);`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd = cmpyiwh(Rss, Rt):<< 1:rnd:sat`	`Word32 Q6_R_cmpyiwh_PR_s1_rnd_sat(Word64 Rss, Word32 Rt)`
`Rd = cmpyiwh(Rss, Rt*):<< 1:rnd:sat`	`Word32 Q6_R_cmpyiwh_PR_conj_s1_rnd_sat(Word64 Rss, Word32 Rt)`
`Rd = cmpyrwh(Rss, Rt):<< 1:rnd:sat`	`Word32 Q6_R_cmpyrwh_PR_s1_rnd_sat(Word64 Rss, Word32 Rt)`
`Rd = cmpyrwh(Rss, Rt*):<< 1:rnd:sat`	`Word32 Q6_R_cmpyrwh_PR_conj_s1_rnd_sat(Word64 Rss, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					Min			d5					
1	1	0	0	0	1	0	1	-	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	1	0	0	d	d	d	d	d	Rd=cmpyiwh(Rss,Rt):<<1:rnd:sat
1	1	0	0	0	1	0	1	-	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	1	0	1	d	d	d	d	d	Rd=cmpyiwh(Rss,Rt*):<<1:rnd:sat
1	1	0	0	0	1	0	1	-	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	1	1	0	d	d	d	d	d	Rd=cmpyrwh(Rss,Rt):<<1:rnd:sat
1	1	0	0	0	1	0	1	-	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	1	1	1	d	d	d	d	d	Rd=cmpyrwh(Rss,Rt*):<<1:rnd:sat

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
Min	Minor opcode
RegType	Register type

Complex multiply real or imaginary 32-bit

Multiply complex values Rss and Rtt. The inputs have a real 32-bit value in the low word and an imaginary 32-bit value in the high word. Take either the real or imaginary result and optionally accumulate with a 64-bit destination.

Syntax	Behavior

`Rd = cmpyiw(Rss, Rtt):<< 1:rnd:sat`	`tmp128 = sxt64->128((Rss.w[0] * Rtt.w[1]));`
	`acc128 = sxt64->128((Rss.w[1] * Rtt.w[0]));`
	`const128 = sxt64->128(0x40000000);`
	`acc128 = tmp128+acc128;`
	`acc128 = acc128+const128;`
	`acc128 = (size8s_t) (acc128 >> 31);`
	`acc64 = sxt128->64(acc128);`
	`Rd = sat32(acc64);`

`Rd = cmpyiw(Rss, Rtt):<< 1:sat`	`tmp128 = sxt64->128((Rss.w[0] * Rtt.w[1]));`
	`acc128 = sxt64->128((Rss.w[1] * Rtt.w[0]));`
	`acc128 = tmp128+acc128;`
	`acc128 = (size8s_t) (acc128 >> 31);`
	`acc64 = sxt128->64(acc128);`
	`Rd = sat32(acc64);`

`Rd = cmpyiw(Rss, Rtt*):<< 1:rnd:sat`	`tmp128 = sxt64->128((Rss.w[1] * Rtt.w[0]));`
	`acc128 = sxt64->128((Rss.w[0] * Rtt.w[1]));`
	`const128 = sxt64->128(0x40000000);`
	`acc128 = tmp128-acc128;`
	`acc128 = acc128+const128;`
	`acc128 = (size8s_t) (acc128 >> 31);`
	`acc64 = sxt128->64(acc128);`
	`Rd = sat32acc64);`

`Rd = cmpyiw(Rss, Rtt*):<< 1:sat`	`tmp128 = sxt64->128((Rss.w[1] * Rtt.w[0]));`
	`acc128 = sxt64->128((Rss.w[0] * Rtt.w[1]));`
	`acc128 = tmp128-acc128;`
	`acc128 = (size8s_t) (acc128 >> 31);`
	`acc64 = sxt128->64(acc128);`
	`Rd = sat32(acc64);`

`Rd = cmpyrw(Rss, Rtt):<< 1:rnd:sat`	`tmp128 = sxt64->128((Rss.w[0] * Rtt.w[0]));`
	`acc128 = sxt64->128((Rss.w[1] * Rtt.w[1]));`
	`const128 = sxt64->128(0x40000000);`
	`acc128 = tmp128-acc128;`
	`acc128 = acc128+const128;`
	`acc128 = (size8s_t) (acc128 >> 31);`
	`acc64 = sxt128->64(acc128);`
	`Rd = sat32(acc64);`

`Rd = cmpyrw(Rss, Rtt):<< 1:sat`	`tmp128 = sxt64->128((Rss.w[0] * Rtt.w[0]));`
	`acc128 = sxt64->128((Rss.w[1] * Rtt.w[1]));`
	`acc128 = tmp128-acc128;`
	`acc128 = (size8s_t) (acc128 >> 31);`
	`acc64 = sxt128->64(acc128);`
	`Rd = sat32(acc64);`

Syntax	Behavior

`Rd=cmpyrw(Rss, Rtt*):<< 1:rnd:sat`	`tmp128 = sxt64->128((Rss.w[0] * Rtt.w[0]));`
	`acc128 = sxt64->128((Rss.w[1] * Rtt.w[1]));`
	`const128 = sxt64->128(0x40000000);`
	`acc128 = tmp128+acc128;`
	`acc128 = acc128+const128;`
	`acc128 = (size8s_t) (acc128 >> 31);`
	`acc64 = sxt128->64(acc128);`
	`Rd = sat32(acc64);`

`Rd=cmpyrw(Rss, Rtt*):<< 1:sat`	`tmp128 = sxt64->128((Rss.w[0] * Rtt.w[0]));`
	`acc128 = sxt64->128((Rss.w[1] * Rtt.w[1]));`
	`acc128 = tmp128+acc128;`
	`acc128 = (size8s_t) (acc128 >> 31);`
	`acc64 = sxt128->64(acc128);`
	`Rd = sat32(acc64);`

`Rdd=cmpyiw(Rss,Rtt)`	`Rdd = ((Rss.w[0] * Rtt.w[1]) + (Rss.w[1] * Rtt.w[0]));`
`Rdd=cmpyiw(Rss,Rtt*)`	`Rdd = ((Rss.w[1] * Rtt.w[0]) - (Rss.w[0] * Rtt.w[1]));`
`Rdd=cmpyrw(Rss,Rtt)`	`Rdd = ((Rss.w[0] * Rtt.w[0]) - (Rss.w[1] * Rtt.w[1]));`
`Rdd=cmpyrw(Rss,Rtt*)`	`Rdd = ((Rss.w[0] * Rtt.w[0]) + (Rss.w[1] * Rtt.w[1]));`
`Rxx+=cmpyiw(Rss,Rtt)`	`Rxx += ((Rss.w[0] * Rtt.w[1]) + (Rss.w[1] * Rtt.w[0]));`
`Rxx+=cmpyiw(Rss,Rtt*)`	`Rxx += ((Rss.w[1] * Rtt.w[0]) - (Rss.w[0] * Rtt.w[1]));`
`Rxx+=cmpyrw(Rss,Rtt)`	`Rxx += ((Rss.w[0] * Rtt.w[0]) - (Rss.w[1] * Rtt.w[1]));`
`Rxx+=cmpyrw(Rss,Rtt*)`	`Rxx += ((Rss.w[0] * Rtt.w[0]) + (Rss.w[1] * Rtt.w[1]));`

Class: XTYPE (slots 3)

Notes

- This instruction can only execute on a core with the Hexagon audio extensions
- A packet with this instruction cannot have a slot 2 multiply instruction.
- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

Rd= cmpyiw(Rss, Rtt):<<1:rnd:sat	Word32 Q6_R_cmpyiw_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt)
Rd=cmpyiw(Rss, Rtt):<<1:sat	Word32 Q6_R_cmpyiw_PP_s1_sat(Word64 Rss, Word64 Rtt)
Rd=cmpyiw(Rss, Rtt*):<<1:rnd:sat	Word32 Q6_R_cmpyiw_PP_conj_s1_rnd_sat(Word64 Rss, Word64 Rtt)
Rd=cmpyiw(Rss, Rtt*):<<1:sat	Word32 Q6_R_cmpyiw_PP_conj_s1_sat(Word64 Rss, Word64 Rtt)
Rd=cmpyrw(Rss, Rtt):<<1:rnd:sat	Word32 Q6_R_cmpyrw_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt)
Rd=cmpyrw(Rss, Rtt):<< 1:sat	Word32 Q6_R_cmpyrw_PP_s1_sat(Word64 Rss, Word64 Rtt)
Rd=cmpyrw(Rss, Rtt*):<<1:rnd:sat	Word32 Q6_R_cmpyrw_PP_conj_s1_rnd_sat(Word64 Rss, Word64 Rtt)
Rd=cmpyrw(Rss,Rtt*):<<1:sat	Word32 Q6_R_cmpyrw_PP_conj_s1_sat(Word64 Rss, Word64 Rtt)
Rdd=cmpyiw(Rss,Rtt)	Word64 Q6_P_cmpyiw_PP(Word64 Rss, Word64 Rtt)
Rdd=cmpyiw(Rss,Rtt*)	Word64 Q6_P_cmpyiw_PP_conj(Word64 Rss, Word64 Rtt)
Rdd=cmpyrw(Rss,Rtt)	Word64 Q6_P_cmpyrw_PP(Word64 Rss, Word64 Rtt)
Rdd=cmpyrw(Rss,Rtt*)	Word64 Q6_P_cmpyrw_PP_conj(Word64 Rss, Word64 Rtt)
Rxx+=cmpyiw(Rss,Rtt)	Word64 Q6_P_cmpyiwacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)
Rxx+=cmpyiw(Rss,Rtt*)	Word64 Q6_P_cmpyiwacc_PP_conj(Word64 Rxx, Word64 Rss, Word64 Rtt)
Rxx+=cmpyrw(Rss,Rtt)	Word64 Q6_P_cmpyrwacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)
Rxx+=cmpyrw(Rss,Rtt*)	Word64 Q6_P_cmpyrwacc_PP_conj(Word64 Rxx, Word64 Rss, Word64 Rtt)

Encoding

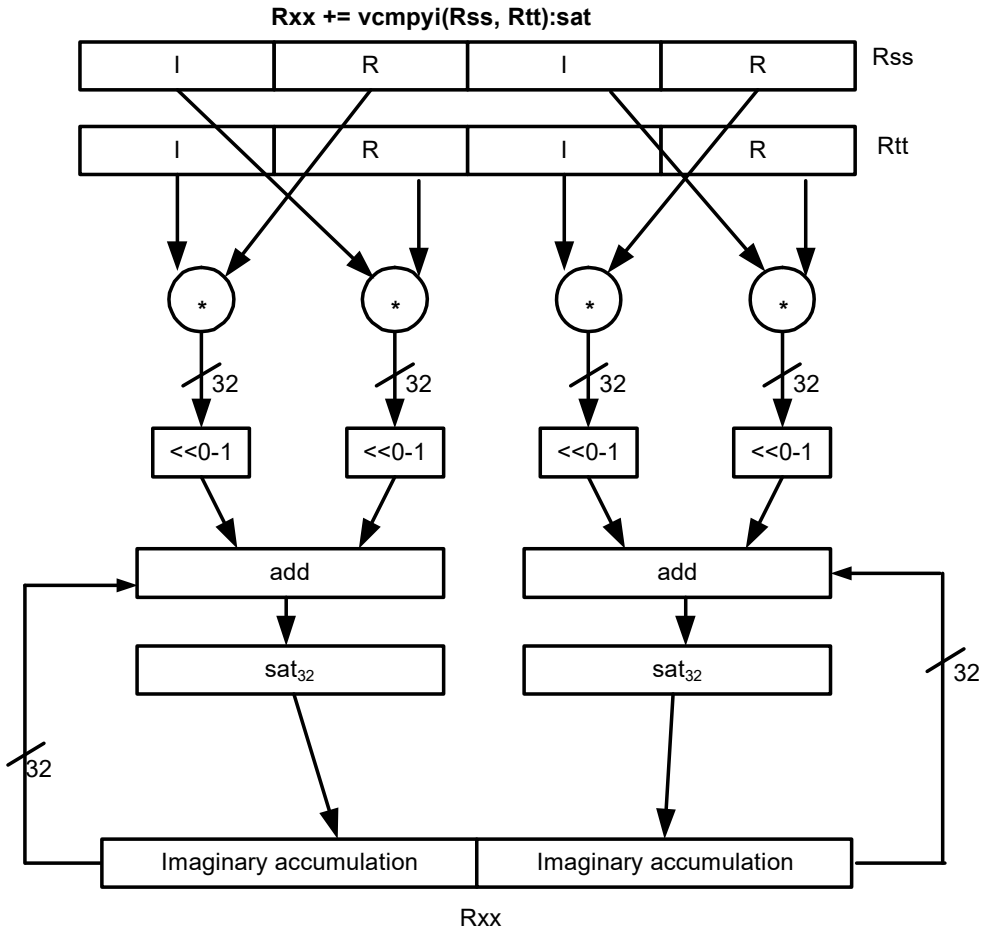
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=cmpyiw(Rss,Rtt)
1	1	1	0	1	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=cmpyrw(Rss,Rtt)
1	1	1	0	1	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=cmpyrw(Rss,Rtt*)
1	1	1	0	1	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=cmpyiw(Rss,Rtt*)
1	1	1	0	1	0	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=cmpyiw(Rss,Rtt*):<<1:sat
1	1	1	0	1	0	0	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=cmpyiw(Rss,Rtt):<<1:sat
1	1	1	0	1	0	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=cmpyrw(Rss,Rtt):<<1:sat
1	1	1	0	1	0	0	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=cmpyrw(Rss,Rtt*):<<1:sat

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	0	1	0	0	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=cmpyiw(Rss,Rtt*):<<1:rnd:sat
1	1	1	0	1	0	0	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=cmpyiw(Rss,Rtt):<<1:rnd:sat
1	1	1	0	1	0	0	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=cmpyrw(Rss,Rtt):<<1:rnd:sat
1	1	1	0	1	0	0	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=cmpyrw(Rss,Rtt*):<<1:rnd:sat
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=cmpyiw(Rss,Rtt*)
1	1	1	0	1	0	1	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=cmpyiw(Rss,Rtt)
1	1	1	0	1	0	1	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=cmpyrw(Rss,Rtt)
1	1	1	0	1	0	1	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=cmpyrw(Rss,Rtt*)

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector complex multiply real or imaginary

The inputs Rss and Rtt are a vector of two complex values. Each complex value is composed of a 16-bit imaginary portion in the upper halfword and a 16-bit real portion in the lower halfword. Generate two complex results, either the real result or the imaginary result. These results are optionally shifted left by 0-1 bits, and optionally accumulated with the destination register.



Syntax	Behavior
`Rdd= vcmpyi(Rss, Rtt)[:<< 1]:sat`	`Rdd.w[0]=sat32((Rss.h[1] * Rtt.h[0]) + (Rss.h[0] * Rtt.h[1])[<<1]);` `Rdd.w[1]=sat32((Rss.h[3] * Rtt.h[2]) + (Rss.h[2] * Rtt.h[3])[<<1]);`
`Rdd= vcmpyr(Rss, Rtt)[:<< 1]:sat`	`Rdd.w[0]=sat32((Rss.h[0] * Rtt.h[0]) - (Rss.h[1] * Rtt.h[1])[<<1]);` `Rdd.w[1]=sat32((Rss.h[2] * Rtt.h[2]) - (Rss.h[3] * Rtt.h[3])[<<1]);`
`Rxx+= vcmpyi(Rss, Rtt):sat`	`Rxx.w[0]=sat32(Rxx.w[0] + (Rss.h[1] * Rtt.h[0]) + (Rss.h[0] * Rtt.h[1])<<0);` `Rxx.w[1]=sat32(Rxx.w[1] + (Rss.h[3] * Rtt.h[2]) + (Rss.h[2] * Rtt.h[3])<<0);`

Syntax	Behavior
`Rxx+= vcmpyr(Rss, Rtt):sat`	`Rxx.w[0]=sat32(Rxx.w[0] + (Rss.h[0] * Rtt.h[0]) - (Rss.h[1] * Rtt.h[1])<<0);` `Rxx.w[1]=sat32(Rxx.w[1] + (Rss.h[2] * Rtt.h[2]) - (Rss.h[3] * Rtt.h[3])<<0);`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vcmpyi(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vcmpyi_PP_s1_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vcmpyi(Rss,Rtt):sat`	`Word64 Q6_P_vcmpyi_PP_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vcmpyr(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vcmpyr_PP_s1_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vcmpyr(Rss,Rtt):sat`	`Word64 Q6_P_vcmpyr_PP_sat(Word64 Rss, Word64 Rtt)`
`Rxx+=vcmpyi(Rss,Rtt):sat`	`Word64 Q6_P_vcmpyiacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vcmpyr(Rss,Rtt):sat`	`Word64 Q6_P_vcmpyracc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vcmpyr(Rss,Rtt)[:<<N]:sat
1	1	1	0	1	0	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vcmpyi(Rss,Rtt)[:<<N]:sat
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vcmpyr(Rss,Rtt):sat
1	1	1	0	1	0	1	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vcmpyi(Rss,Rtt):sat

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector complex conjugate

Perform a vector complex conjugate of both complex values in vector Rss. This is done by negating the imaginary halfwords, and placing the result in destination Rdd.

Syntax	Behavior

`Rdd=vconj(Rss):sat`	`Rdd.h[1]=sat16(-Rss.h[1]);`
	`Rdd.h[0]=Rss.h[0];`
	`Rdd.h[3]=sat16(-Rss.h[3]);`
	`Rdd.h[2]=Rss.h[2];`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vconj(Rss):sat`	`Word64 Q6_P_vconj_P_sat(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vconj(Rss):sat

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

Vector complex rotate

Take the least significant bits of Rt, and use these bits to rotate each of the two complex values in the source vector a multiple of 90 degrees. Bits 0 and 1 control the rotation factor for word 0, and bits 2 and 3 control the rotation factor for word 1.

If the rotation control bits are 0, the rotation is 0: the real and imaginary halves of the source appear unchanged and unmoved in the destination.

If the rotation control bits are 1, the rotation is -pi/2: the real half of the destination gets the imaginary half of the source, and the imaginary half of the destination gets the negative real half of the source.

If the rotation control bits are 2, the rotation is pi/2: the real half of the destination gets the negative imaginary half of the source, and the imaginary half of the destination gets the real half of the source.

If the rotation control bits are 3, the rotation is pi: the real half of the destination gets the negative real half of the source, and the imaginary half of the destination gets the negative imaginary half of the source.

Syntax	Behavior

`Rdd=vcrotate(Rss,Rt)`	`tmp = Rt[1:0];`
	`if (tmp == 0) {`
	`  Rdd.h[0]=Rss.h[0];`
	`  Rdd.h[1]=Rss.h[1];`
	`} else if (tmp == 1) {`
	`  Rdd.h[0]=Rss.h[1];`
	`  Rdd.h[1]=sat16(-Rss.h[0]);`
	`} else if (tmp == 2) {`
	`  Rdd.h[0]=sat16(-Rss.h[1]);`
	`  Rdd.h[1]=Rss.h[0];`
	`} else {`
	`  Rdd.h[0]=sat16(-Rss.h[0]);`
	`  Rdd.h[1]=sat16(-Rss.h[1]);`
	`}`
	`tmp = Rt[3:2];`
	`if (tmp == 0) {`
	`  Rdd.h[2]=Rss.h[2];`
	`  Rdd.h[3]=Rss.h[3];`
	`} else if (tmp == 1) {`
	`  Rdd.h[2]=Rss.h[3];`
	`  Rdd.h[3]=sat16(-Rss.h[2]);`
	`} else if (tmp == 2) {`
	`  Rdd.h[2]=sat16(-Rss.h[3]);`
	`  Rdd.h[3]=Rss.h[2];`
	`} else {`
	`  Rdd.h[2]=sat16(-Rss.h[2]);`
	`  Rdd.h[3]=sat16(-Rss.h[3]);`
	`}`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vcrotate(Rss,Rt)`	`Word64 Q6_P_vcrotate_PR(Word64 Rss, Word32 Rt)`

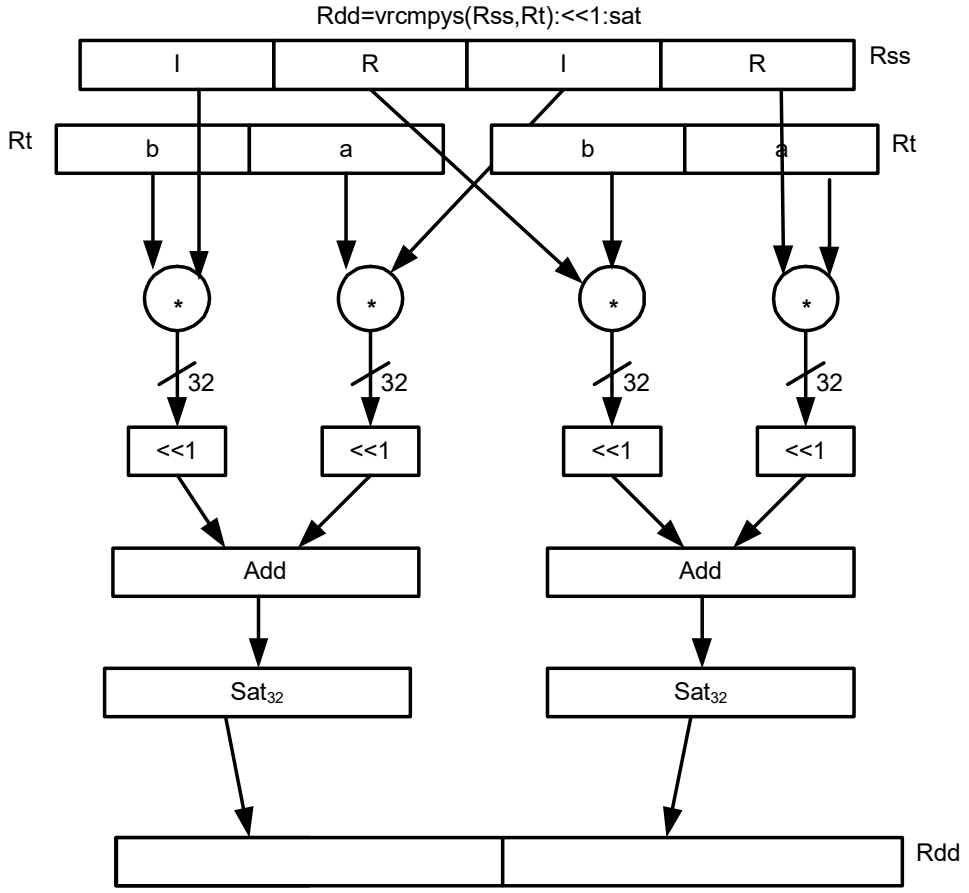
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	1	1	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=vcrotate(Rss,Rt)

Field name	Description
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`Maj`	Major opcode
`Min`	Minor opcode
`RegType`	Register type

Vector reduce complex multiply by scalar

Multiply a complex number by a scalar. Rss contains two complex numbers. The real portions are each multiplied by two scalars contained in register Rt, scaled, summed, optionally accumulated, saturated, and stored in the lower word of Rdd. A similar operation is done on the two imaginary portions of Rss.



Syntax	Behavior

`Rdd=vrcmpys(Rss,Rt):<<1:sat`	`if ("Rt & 1") {`
	`  Assembler mapped to:`
	`"Rdd=vrcmpys(Rss,Rtt):<<1:sat:raw:hi";`
	`} else {`
	`  Assembler mapped to:`
	`"Rdd=vrcmpys(Rss,Rtt):<<1:sat:raw:lo";`
	`}`

`Rdd=vrcmpys(Rss,Rtt):<<1:sat:raw:hi`	`Rdd.w[1]=sat32((Rss.h[1] * Rtt.w[1].h[0])<<1 +`
	`(Rss.h[3] * Rtt.w[1].h[1])<<1);`
	`Rdd.w[0]=sat32((Rss.h[0] * Rtt.w[1].h[0])<<1 +`
	`(Rss.h[2] * Rtt.w[1].h[1])<<1);`

`Rdd=vrcmpys(Rss,Rtt):<<1:sat:raw:lo`	`Rdd.w[1]=sat32((Rss.h[1] * Rtt.w[0].h[0])<<1 +`
	`(Rss.h[3] * Rtt.w[0].h[1])<<1);`
	`Rdd.w[0]=sat32((Rss.h[0] * Rtt.w[0].h[0])<<1 +`
	`(Rss.h[2] * Rtt.w[0].h[1])<<1);`

Syntax	Behavior

`Rxx+=vrcmpys(Rss,Rt):<< 1:sat`	`if ("Rt & 1") {`
	`  Assembler mapped to: "Rxx += vrcmpys(Rss, Rtt):<< 1:sat:raw:hi";`
	`} else {`
	`  Assembler mapped to: "Rxx+=vrcmpys(Rss,Rtt): << 1:sat:raw:lo";`
	`}`

`Rxx+=vrcmpys(Rss, Rtt):<< 1:sat:raw:hi`	`Rxx.w[1]=sat32(Rxx.w[1] + (Rss.h[1] * Rtt.w[1].h[0])<<1 + (Rss.h[3] * Rtt.w[1].h[1]) << 1);` `Rxx.w[0]=sat32(Rxx.w[0] + (Rss.h[0] * Rtt.w[1].h[0]) << 1 + (Rss.h[2] * Rtt.w[1].h[1]) << 1);`
`Rxx+=vrcmpys(Rss, Rtt):<< 1:sat:raw:lo`	`Rxx.w[1]=sat32(Rxx.w[1] + (Rss.h[1] * Rtt.w[0].h[0]) << 1 + (Rss.h[3] * Rtt.w[0].h[1]) << 1);` `Rxx.w[0]=sat32(Rxx.w[0] + (Rss.h[0] * Rtt.w[0].h[0]) << 1 + (Rss.h[2] * Rtt.w[0].h[1]) << 1);`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vrcmpys(Rss,Rt):<<1:sat`	`Word64 Q6_P_vrcmpys_PR_s1_sat(Word64 Rss, Word32 Rt)`
`Rxx+=vrcmpys(Rss,Rt):<<1:sat`	`Word64 Q6_P_vrcmpysacc_PR_s1_sat(Word64 Rxx, Word64 Rss, Word32 Rt)`

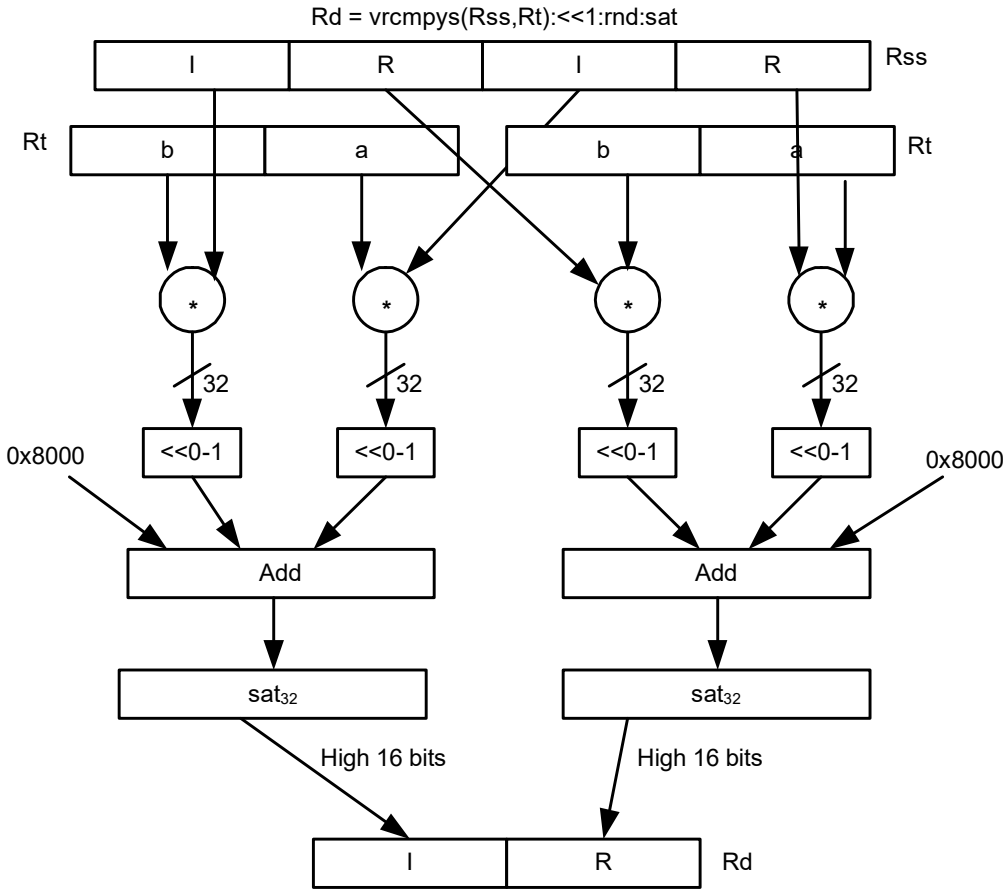
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5						MinOp			d5					
1	1	1	0	1	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vrcmpys(Rss,Rtt):<<1 :sat:raw:hi
1	1	1	0	1	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vrcmpys(Rss,Rtt):<<1 :sat:raw:lo
ICLASS				RegType				MajOp			s5					Parse			t5						MinOp			x5					
1	1	1	0	1	0	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vrcmpys(Rss,Rtt):<< 1:sat:raw:hi
1	1	1	0	1	0	1	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vrcmpys(Rss,Rtt):<< 1:sat:raw:lo

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector reduce complex multiply by scalar with round and pack

Multiply a complex number by scalar. Rss contains two complex numbers. The real portions are each multiplied by two scalars contained in register Rt, scaled, summed, rounded, and saturated. The upper 16 bits of this result are packed in the lower halfword of Rd. A similar operation is done on the two imaginary portions of Rss.



Syntax	Behavior

`Rd=vrcmpys(Rss,Rt):<< 1:rnd:sat`	`if ("Rt & 1") {`
	`  Assembler mapped to: "Rd = vrcmpys(Rss, Rtt):<< 1:rnd:sat:raw:hi";`
	`} else {`
	`  Assembler mapped to: "Rd = vrcmpys(Rss, Rtt):<< 1:rnd:sat:raw:lo";`
	`}`

`Rd=vrcmpys(Rss,Rtt):<< 1:rnd:sat:raw:hi`	`Rd.h[1]=sat32((Rss.h[1] * Rtt.w[1].h[0])<<1 + (Rss.h[3] * Rtt.w[1].h[1])<<1 + 0x8000).h[1];` `Rd.h[0]=sat32((Rss.h[0] * Rtt.w[1].h[0])<<1 + (Rss.h[2] * Rtt.w[1].h[1])<<1 + 0x8000).h[1];`
`Rd=vrcmpys(Rss,Rtt):<< 1:rnd:sat:raw:lo`	`Rd.h[1]=sat32((Rss.h[1] * Rtt.w[0].h[0])<<1 + (Rss.h[3] * Rtt.w[0].h[1])<<1 + 0x8000).h[1];` `Rd.h[0]=sat32((Rss.h[0] * Rtt.w[0].h[0])<<1 + (Rss.h[2] * Rtt.w[0].h[1])<<1 + 0x8000).h[1];`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=vrcmpys(Rss,Rt):<< 1:rnd:sat`	`Word32 Q6_R_vrcmpys_PR_s1_rnd_sat(Word64 Rss, Word32 Rt)`

Encoding

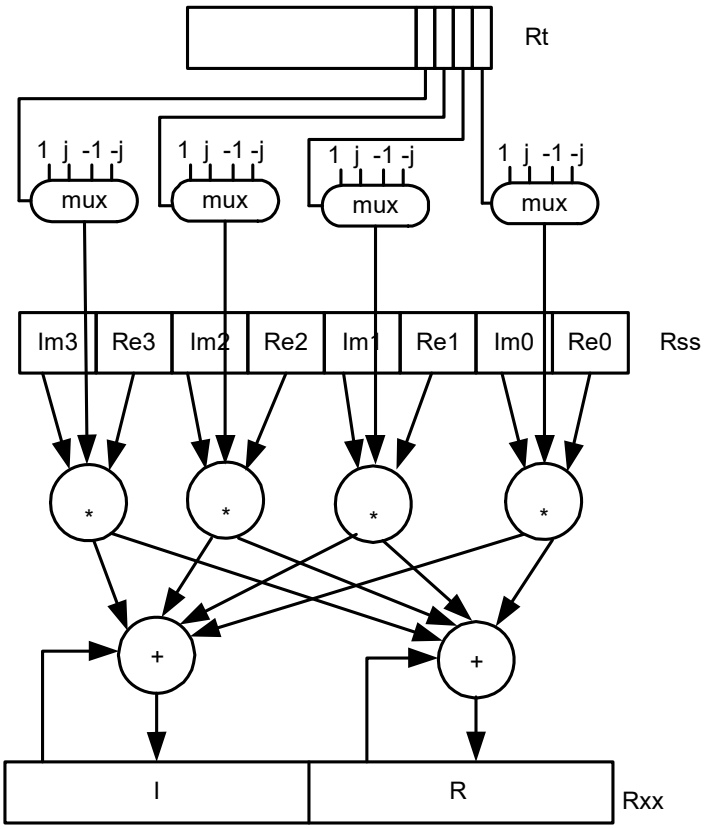
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	1	1	-	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=vrcmpys(Rss,Rtt):<<1:rnd:sat:raw:hi
1	1	1	0	1	0	0	1	1	-	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=vrcmpys(Rss,Rtt):<<1:rnd:sat:raw:lo

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector reduce complex rotate

This instruction is useful for CDMA despreading. An unsigned 2-bit immediate specifies a byte to use in Rt. Each of four 2-bit fields in the specified byte selects a rotation amount for one of the four complex numbers in Rss. The real and imaginary products are accumulated and stored as a 32-bit complex number in Rd. Optionally, the destination register can also be accumulated.

Rxx += vrcrotate(Rss,Rt,#0)



Syntax	Behavior

`Rdd=vrcrotate(Rss,Rt,#u2)`	`sumr = 0;`
	`sumi = 0;`
	`control = Rt.ub[#u];`
	`for (i = 0; i < 8; i += 2) {`
	`  tmpr = Rss.b[i];`
	`  tmpi = Rss.b[i+1];`
	`  switch (control & 3) {`
	`  case 0: sumr += tmpr;`
	`  sumi += tmpi;`
	`  break;`
	`  case 1: sumr += tmpi;`
	`  sumi -= tmpr;`
	`  break;`
	`  case 2: sumr -= tmpi;`
	`  sumi += tmpr;`
	`  break;`
	`  case 3: sumr -= tmpr;`
	`  sumi -= tmpi;`
	`  break;`
	`  }`
	`  control = control >> 2;`
	`}`
	`Rdd.w[0]=sumr;`
	`Rdd.w[1]=sumi;`

`Rxx+=vrcrotate(Rss,Rt,#u2)`	`sumr = 0;`
	`sumi = 0;`
	`control = Rt.ub[#u];`
	`for (i = 0; i < 8; i += 2) {`
	`  tmpr = Rss.b[i];`
	`  tmpi = Rss.b[i+1];`
	`  switch (control & 3) {`
	`  case 0: sumr += tmpr;`
	`  sumi += tmpi;`
	`  break;`
	`  case 1: sumr += tmpi;`
	`  sumi -= tmpr;`
	`  break;`
	`  case 2: sumr -= tmpi;`
	`  sumi += tmpr;`
	`  break;`
	`  case 3: sumr -= tmpr;`
	`  sumi -= tmpi;`
	`  break;`
	`  }`
	`  control = control >> 2;`
	`}`
	`Rxx.w[0]=Rxx.w[0] + sumr;`
	`Rxx.w[1]=Rxx.w[1] + sumi;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vrcrotate(Rss,Rt,#u2)`	`Word64 Q6_P_vrcrotate_PRI(Word64 Rss, Word32 Rt, Word32 Iu2)`
`Rxx+=vrcrotate(Rss,Rt,#u2)`	`Word64 Q6_P_vrcrotateacc_PRI(Word64 Rxx, Word64 Rss, Word32 Rt, Word32 Iu2)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	1	1	1	1	-	s	s	s	s	s	P	P	i	t	t	t	t	t	1	1	i	d	d	d	d	d	Rdd=vrcrotate(Rss,Rt,#u2)
ICLASS				RegType				Maj			s5					Parse			t5								x5					
1	1	0	0	1	0	1	1	1	0	1	s	s	s	s	s	P	P	i	t	t	t	t	t	-	-	i	x	x	x	x	x	Rxx+=vrcrotate(Rss,Rt,#u2)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x
Maj	Major opcode
Min	Minor opcode
RegType	Register type

11.10.4 XTYPE FP

The XTYPE FP instruction subclass includes instructions for floating point math.

Floating point addition

Add two floating point values.

Syntax	Behavior
`Rd=sfadd(Rs,Rt)`	`Rd=Rs+Rt;`
`Rdd=dfadd(Rss,Rtt)`	`Rdd=Rss+Rtt;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=sfadd(Rs,Rt)`	`Word32 Q6_R_sfadd_RR(Word32 Rs, Word32 Rt)`
`Rdd=dfadd(Rss,Rtt)`	`Word64 Q6_P_dfadd_PP(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=dfadd(Rss,Rtt)
1	1	1	0	1	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=sfadd(Rs,Rt)

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Classify floating-point value

Classify floating point values. Classes are normal, subnormal, zero, NaN, or infinity. If the number is one of the specified classes, return true.

Syntax	Behavior

`Pd=dfclass(Rss,#u5)`	`Pd = 0;`
	`class = fpclassify(Rss);`
	`if (#u.0 && (class == FP_ZERO)) Pd = 0xff;`
	`if (#u.1 && (class == FP_NORMAL)) Pd = 0xff;`
	`if (#u.2 && (class == FP_SUBNORMAL)) Pd = 0xff;`
	`if (#u.3 && (class == FP_INFINITE)) Pd = 0xff;`
	`if (#u.4 && (class == FP_NAN)) Pd = 0xff;`
	`cancel_flags();`

`Pd=sfclass(Rs,#u5)`	`Pd = 0;`
	`class = fpclassify(Rs);`
	`if (#u.0 && (class == FP_ZERO)) Pd = 0xff;`
	`if (#u.1 && (class == FP_NORMAL)) Pd = 0xff;`
	`if (#u.2 && (class == FP_SUBNORMAL)) Pd = 0xff;`
	`if (#u.3 && (class == FP_INFINITE)) Pd = 0xff;`
	`if (#u.4 && (class == FP_NAN)) Pd = 0xff;`
	`cancel_flags();`

Class: XTYPE (slots 2,3)

Intrinsics

`Pd=dfclass(Rss,#u5)`	`Byte Q6_p_dfclass_PI(Word64 Rss, Word32 Iu5)`
`Pd=sfclass(Rs,#u5)`	`Byte Q6_p_sfclass_RI(Word32 Rs, Word32 Iu5)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse														d2		
1	0	0	0	0	1	0	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	-	-	-	-	-	-	**d**	**d**	Pd=sfclass(Rs,#u5)
ICLASS				RegType							s5					Parse														d2		
1	1	0	1	1	1	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	0	0	0	**i**	**i**	**i**	**i**	**i**	1	0	-	**d**	**d**	Pd=dfclass(Rss,#u5)

Field name	Description
`RegType`	Register type
`MajOp`	Major opcode
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d2`	Field to encode register d
`s5`	Field to encode register s

Compare floating-point value

Compare floating point values. p0 returns true if at least one value is a NaN, zero otherwise.

Syntax	Behavior
`Pd=dfcmp.eq(Rss,Rtt)`	`Pd=Rss==Rtt ? 0xff : 0x00;`
`Pd=dfcmp.ge(Rss,Rtt)`	`Pd=Rss>=Rtt ? 0xff : 0x00;`
`Pd=dfcmp.gt(Rss,Rtt)`	`Pd=Rss>Rtt ? 0xff : 0x00;`
`Pd=dfcmp.uo(Rss,Rtt)`	`Pd=isunordered(Rss,Rtt) ? 0xff : 0x00;`
`Pd=sfcmp.eq(Rs,Rt)`	`Pd=Rs==Rt ? 0xff : 0x00;`
`Pd=sfcmp.ge(Rs,Rt)`	`Pd=Rs>=Rt ? 0xff : 0x00;`
`Pd=sfcmp.gt(Rs,Rt)`	`Pd=Rs>Rt ? 0xff : 0x00;`
`Pd=sfcmp.uo(Rs,Rt)`	`Pd=isunordered(Rs,Rt) ? 0xff : 0x00;`

Class: XTYPE (slots 2,3)

Intrinsics

`Pd=dfcmp.eq(Rss,Rtt)`	`Byte Q6_p_dfcmp_eq_PP(Word64 Rss, Word64 Rtt)`
`Pd=dfcmp.ge(Rss,Rtt)`	`Byte Q6_p_dfcmp_ge_PP(Word64 Rss, Word64 Rtt)`
`Pd=dfcmp.gt(Rss,Rtt)`	`Byte Q6_p_dfcmp_gt_PP(Word64 Rss, Word64 Rtt)`
`Pd=dfcmp.uo(Rss,Rtt)`	`Byte Q6_p_dfcmp_uo_PP(Word64 Rss, Word64 Rtt)`
`Pd=sfcmp.eq(Rs,Rt)`	`Byte Q6_p_sfcmp_eq_RR(Word32 Rs, Word32 Rt)`
`Pd=sfcmp.ge(Rs,Rt)`	`Byte Q6_p_sfcmp_ge_RR(Word32 Rs, Word32 Rt)`
`Pd=sfcmp.gt(Rs,Rt)`	`Byte Q6_p_sfcmp_gt_RR(Word32 Rs, Word32 Rt)`
`Pd=sfcmp.uo(Rs,Rt)`	`Byte Q6_p_sfcmp_uo_RR(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min						d2		
1	1	0	0	0	1	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	-	-	-	**d**	**d**	Pd=sfcmp.ge(Rs,Rt)
1	1	0	0	0	1	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	-	-	-	**d**	**d**	Pd=sfcmp.uo(Rs,Rt)
1	1	0	0	0	1	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	-	-	-	**d**	**d**	Pd=sfcmp.eq(Rs,Rt)
1	1	0	0	0	1	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	-	-	-	**d**	**d**	Pd=sfcmp.gt(Rs,Rt)
ICLASS				RegType							s5					Parse			t5					MinOp						d2		
1	1	0	1	0	0	1	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	-	-	-	**d**	**d**	Pd=dfcmp.eq(Rss,Rtt)
1	1	0	1	0	0	1	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	-	-	-	**d**	**d**	Pd=dfcmp.gt(Rss,Rtt)
1	1	0	1	0	0	1	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	-	-	-	**d**	**d**	Pd=dfcmp.ge(Rss,Rtt)
1	1	0	1	0	0	1	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	-	-	-	**d**	**d**	Pd=dfcmp.uo(Rss,Rtt)

Field name	Description
RegType	Register type
MinOp	Minor opcode
ICLASS	Instruction class
Parse	Packet/loop parse bits
d2	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Convert floating-point value to other format

Convert floating point values. If rounding is required, it happens according to the rounding mode.

Syntax	Behavior
`Rd=convert_df2sf(Rss)`	`Rd = conv_df_to_sf(Rss);`
`Rdd=convert_sf2df(Rs)`	`Rdd = conv_sf_to_df(Rs);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=convert_df2sf(Rss)`	`Word32 Q6_R_convert_df2sf_P(Word64 Rss)`
`Rdd=convert_sf2df(Rs)`	`Word64 Q6_P_convert_sf2df_R(Word32 Rs)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	1	0	0	1	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=convert_sf2df(Rs)
1	0	0	0	1	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=convert_df2sf(Rss)

Field name	Description
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major opcode
`MinOp`	Minor opcode
`RegType`	Register type

Convert integer to floating-point value

Convert floating point values. If rounding is required, it happens according to the rounding mode unless the :chop option is specified.

Syntax	Behavior
`Rd=convert_d2sf(Rss)`	`Rd = conv_8s_to_sf(Rss.s64);`
`Rd=convert_ud2sf(Rss)`	`Rd = conv_8u_to_sf(Rss.u64);`
`Rd=convert_uw2sf(Rs)`	`Rd = conv_4u_to_sf(Rs.uw[0]);`
`Rd=convert_w2sf(Rs)`	`Rd = conv_4s_to_sf(Rs.s32);`
`Rdd=convert_d2df(Rss)`	`Rdd = conv_8s_to_df(Rss.s64);`
`Rdd=convert_ud2df(Rss)`	`Rdd = conv_8u_to_df(Rss.u64);`
`Rdd=convert_uw2df(Rs)`	`Rdd = conv_4u_to_df(Rs.uw[0]);`
`Rdd=convert_w2df(Rs)`	`Rdd = conv_4s_to_df(Rs.s32);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=convert_d2sf(Rss)`	`Word32 Q6_R_convert_d2sf_P(Word64 Rss)`
`Rd=convert_ud2sf(Rss)`	`Word32 Q6_R_convert_ud2sf_P(Word64 Rss)`
`Rd=convert_uw2sf(Rs)`	`Word32 Q6_R_convert_uw2sf_R(Word32 Rs)`
`Rd=convert_w2sf(Rs)`	`Word32 Q6_R_convert_w2sf_R(Word32 Rs)`
`Rdd=convert_d2df(Rss)`	`Word64 Q6_P_convert_d2df_P(Word64 Rss)`
`Rdd=convert_ud2df(Rss)`	`Word64 Q6_P_convert_ud2df_P(Word64 Rss)`
`Rdd=convert_uw2df(Rs)`	`Word64 Q6_P_convert_uw2df_R(Word32 Rs)`
`Rdd=convert_w2df(Rs)`	`Word64 Q6_P_convert_w2df_R(Word32 Rs)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=convert_ud2df(Rss)
1	0	0	0	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=convert_d2df(Rss)
1	0	0	0	0	1	0	0	1	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=convert_uw2df(Rs)
1	0	0	0	0	1	0	0	1	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=convert_w2df(Rs)
1	0	0	0	1	0	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=convert_ud2sf(Rss)
1	0	0	0	1	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=convert_d2sf(Rss)
1	0	0	0	1	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=convert_uw2sf(Rs)
1	0	0	0	1	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=convert_w2sf(Rs)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

Convert floating-point value to integer

Convert floating point values. If rounding is required, it occurs according to the rounding mode unless the :chop option is specified. If the value is out of range of the destination integer type, the INVALID flag is raised and closest integer is chosen, including for infinite inputs. For NaN inputs, the INVALID flag is also raised, and the output value is IMPLEMENTATION DEFINED.

Syntax	Behavior
`Rd=convert_df2uw(Rss)`	`Rd = conv_df_to_4u(Rss).uw[0];`
`Rd=convert_df2uw(Rss):chop`	`round_to_zero();` `Rd = conv_df_to_4u(Rss).uw[0];`
`Rd=convert_df2w(Rss)`	`Rd = conv_df_to_4s(Rss).s32;`
`Rd=convert_df2w(Rss):chop`	`round_to_zero();` `Rd = conv_df_to_4s(Rss).s32;`
`Rd=convert_sf2uw(Rs)`	`Rd = conv_sf_to_4u(Rs).uw[0];`
`Rd=convert_sf2uw(Rs):chop`	`round_to_zero();` `Rd = conv_sf_to_4u(Rs).uw[0];`
`Rd=convert_sf2w(Rs)`	`Rd = conv_sf_to_4s(Rs).s32;`
`Rd=convert_sf2w(Rs):chop`	`round_to_zero();` `Rd = conv_sf_to_4s(Rs).s32;`
`Rdd=convert_df2d(Rss)`	`Rdd = conv_df_to_8s(Rss).s64;`
`Rdd=convert_df2d(Rss):chop`	`round_to_zero();` `Rdd = conv_df_to_8s(Rss).s64;`
`Rdd=convert_df2ud(Rss)`	`Rdd = conv_df_to_8u(Rss).u64;`
`Rdd=convert_df2ud(Rss):chop`	`round_to_zero();` `Rdd = conv_df_to_8u(Rss).u64;`
`Rdd=convert_sf2d(Rs)`	`Rdd = conv_sf_to_8s(Rs).s64;`
`Rdd=convert_sf2d(Rs):chop`	`round_to_zero();` `Rdd = conv_sf_to_8s(Rs).s64;`
`Rdd=convert_sf2ud(Rs)`	`Rdd = conv_sf_to_8u(Rs).u64;`
`Rdd=convert_sf2ud(Rs):chop`	`round_to_zero();` `Rdd = conv_sf_to_8u(Rs).u64;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=convert_df2uw(Rss)`	`Word32 Q6_R_convert_df2uw_P(Word64 Rss)`
`Rd=convert_df2uw(Rss):chop`	`Word32 Q6_R_convert_df2uw_P_chop(Word64 Rss)`
`Rd=convert_df2w(Rss)`	`Word32 Q6_R_convert_df2w_P(Word64 Rss)`
`Rd=convert_df2w(Rss):chop`	`Word32 Q6_R_convert_df2w_P_chop(Word64 Rss)`
`Rd=convert_sf2uw(Rs)`	`Word32 Q6_R_convert_sf2uw_R(Word32 Rs)`
`Rd=convert_sf2uw(Rs):chop`	`Word32 Q6_R_convert_sf2uw_R_chop(Word32 Rs)`

`Rd=convert_sf2w(Rs)`	`Word32 Q6_R_convert_sf2w_R(Word32 Rs)`
`Rd=convert_sf2w(Rs):chop`	`Word32 Q6_R_convert_sf2w_R_chop(Word32 Rs)`
`Rdd=convert_df2d(Rss)`	`Word64 Q6_P_convert_df2d_P(Word64 Rss)`
`Rdd=convert_df2d(Rss):chop`	`Word64 Q6_P_convert_df2d_P_chop(Word64 Rss)`
`Rdd=convert_df2ud(Rss)`	`Word64 Q6_P_convert_df2ud_P(Word64 Rss)`
`Rdd=convert_df2ud(Rss):chop`	`Word64 Q6_P_convert_df2ud_P_chop(Word64 Rss)`
`Rdd=convert_sf2d(Rs)`	`Word64 Q6_P_convert_sf2d_R(Word32 Rs)`
`Rdd=convert_sf2d(Rs):chop`	`Word64 Q6_P_convert_sf2d_R_chop(Word32 Rs)`
`Rdd=convert_sf2ud(Rs)`	`Word64 Q6_P_convert_sf2ud_R(Word32 Rs)`
`Rdd=convert_sf2ud(Rs):chop`	`Word64 Q6_P_convert_sf2ud_R_chop(Word32 Rs)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=convert_df2d(Rss)
1	0	0	0	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=convert_df2ud(Rss)
1	0	0	0	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=convert_df2d(Rss):chop
1	0	0	0	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=convert_df2ud(Rss):chop
1	0	0	0	0	1	0	0	1	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=convert_sf2ud(Rs)
1	0	0	0	0	1	0	0	1	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=convert_sf2d(Rs)
1	0	0	0	0	1	0	0	1	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=convert_sf2ud(Rs):chop
1	0	0	0	0	1	0	0	1	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=convert_sf2d(Rs):chop
1	0	0	0	1	0	0	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=convert_df2uw(Rss)
1	0	0	0	1	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=convert_df2w(Rss)
1	0	0	0	1	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=convert_df2uw(Rss):chop
1	0	0	0	1	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=convert_df2w(Rss):chop
1	0	0	0	1	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=convert_sf2uw(Rs)
1	0	0	0	1	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=convert_sf2uw(Rs):chop
1	0	0	0	1	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=convert_sf2w(Rs)
1	0	0	0	1	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=convert_sf2w(Rs):chop

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

Floating point extreme value assistance

For divide and square root routines, certain values are problematic for the default routine. These instructions appropriately fix up the numerator (`fixupn`), denominator (`fixupd`), or radicand (`fixupr`) for proper calculations when combined with the divide or square root approximation instructions.

Syntax	Behavior
`Rd=sffixupd(Rs,Rt)`	`(Rs,Rt,Rd,adjust)=recip_common(Rs,Rt);` `Rd = Rt;`
`Rd=sffixupn(Rs,Rt)`	`(Rs,Rt,Rd,adjust)=recip_common(Rs,Rt);` `Rd = Rs;`
`Rd=sffixupr(Rs)`	`(Rs,Rd,adjust)=invsqrt_common(Rs);` `Rd = Rs;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=sffixupd(Rs,Rt)`	`Word32 Q6_R_sffixupd_RR(Word32 Rs, Word32 Rt)`
`Rd=sffixupn(Rs,Rt)`	`Word32 Q6_R_sffixupn_RR(Word32 Rs, Word32 Rt)`
`Rd=sffixupr(Rs)`	`Word32 Q6_R_sffixupr_R(Word32 Rs)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	1	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=sffixupr(Rs)
ICLASS				RegType				MajOp			s5					Parse			t5						MinOp			d5					
1	1	1	0	1	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=sffixupn(Rs,Rt)
1	1	1	0	1	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=sffixupd(Rs,Rt)

Field name	Description
`ICLASS`	Instruction class
`MajOp`	Major opcode
`MinOp`	Minor opcode
`RegType`	Register type
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Floating point fused multiply-add

Multiply two values, and add to (or subtract from) the accumulator. Full intermediate precision is kept.

Syntax	Behavior
`Rx+=sfmpy(Rs,Rt)`	`Rx=fmaf(Rs,Rt,Rx);`
`Rx-=sfmpy(Rs,Rt)`	`Rx=fmaf(-Rs,Rt,Rx);`
`Rxx+=dfmpyhh(Rss,Rtt)`	`Rxx = Rss*Rtt with partial product Rxx;`
`Rxx+=dfmpylh(Rss,Rtt)`	`Rxx += (Rss.uw[0] * (0x00100000 | zxt`$_{20->64}$`(Rtt.uw[1]))) << 1;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rx+=sfmpy(Rs,Rt)`	`Word32 Q6_R_sfmpyacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx-=sfmpy(Rs,Rt)`	`Word32 Q6_R_sfmpynac_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rxx+=dfmpyhh(Rss,Rtt)`	`Word64 Q6_P_dfmpyhhacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=dfmpylh(Rss,Rtt)`	`Word64 Q6_P_dfmpylhacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rxx+=dfmpylh(Rss,Rtt)
1	1	1	0	1	0	1	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rxx+=dfmpyhh(Rss,Rtt)
1	1	1	0	1	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	**x**	**x**	**x**	**x**	**x**	Rx+=sfmpy(Rs,Rt)
1	1	1	0	1	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	1	**x**	**x**	**x**	**x**	**x**	Rx-=sfmpy(Rs,Rt)

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Floating point fused multiply-add with scaling

Multiply two values, and add to (or subtract from) the accumulator. Full intermediate precision is kept. Additionally, scale the output. This instruction has special handling of corner cases. If a multiplicand source is zero and a NaN is not produced, the accumulator is left unchanged; this means the sign of a zero accumulator does not change if the product is a true zero. The scaling factor is the predicate taken as a two's compliment number for single precision. The scaling factor is twice the predicate taken as a two's compliment number for double precision. The implementation can change denormal accumulator values to zero for positive scale factors.

Syntax	Behavior

`Rx+=sfmpy(Rs,Rt,Pu):scale`	`PREDUSE_TIMING;`
	`if (isnan(Rx) || isnan(Rs) || isnan(Rt)) Rx = NaN;`
	`tmp=fmaf(Rs,Rt,Rx) * 2**(Pu);`
	`if (!((Rx == 0.0) && is_true_zero(Rs*Rt))) Rx = tmp;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rx+=sfmpy(Rs,Rt,Pu):scale`	`Word32 Q6_R_sfmpyacc_RRp_scale(Word32 Rx, Word32 Rs, Word32 Rt, Byte Pu)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5						u2		x5						
1	1	1	0	1	1	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**x**	**x**	**x**	**x**	**x**	Rx+=sfmpy(Rs,Rt,Pu):scale

Field name	Description
`ICLASS`	Instruction class
`MajOp`	Major opcode
`MinOp`	Minor opcode
`RegType`	Register type
`Parse`	Packet/loop parse bits
`s5`	Field to encode register s
`t5`	Field to encode register t
`u2`	Field to encode register u
`x5`	Field to encode register x

Floating point reciprocal square root approximation

Provides an approximation of the reciprocal square root of the radicand (Rs), if combined with the appropriate fixup instruction. Certain values (such as infinities or zeros) in the numerator or denominator may yield values that are not reciprocal approximations, but yield the correct answer when combined with fixup instructions and the appropriate routines.

For compatibility, exact results of these instructions cannot be relied on. The precision of the approximation for this architecture and later is at least 6.6 bits.

Syntax	Behavior

`Rd,Pe=sfinvsqrta(Rs)`	`if ((Rs,Rd,adjust)=invsqrt_common(Rs)) {`
	`  Pe = adjust;`
	`  idx = (Rs >> 17) & 0x7f;`
	`  mant = (invsqrt_lut[idx] << 15);`
	`  exp = 127 - ((exponent(Rs) - 127) >> 1) - 1;`
	`  Rd = -1**Rs.31 * 1.MANT * 2**(exp-BIAS);`
	`}`

Class: XTYPE (slots 2,3)

Notes

- This instruction provides a certain amount of accuracy. In future versions the accuracy may increase. For future compatibility, dependence on exact values must be avoided.
- The predicate generated by this instruction cannot be used as a .new predicate, nor can it be automatically AND'd with another predicate.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									e2		d5						
1	0	0	0	1	0	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	**e**	**e**	**d**	**d**	**d**	**d**	**d**	Rd,Pe=sfinvsqrta(Rs)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
e2	Field to encode register e
s5	Field to encode register s
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

Floating point fused multiply-add for library routines

Multiply two values, and add to (or subtract from) the accumulator. Full intermediate precision is kept. This instruction has special handling of corner cases. Addition of infinities with opposite signs, or subtraction of infinities with like signs, is defined as (positive) zero. Rounding is always Nearest-Even, except that overflows to infinity round to maximal finite values. If a multiplicand source is zero and a NaN is not produced, the accumulator is left unchanged; this means the sign of a zero accumulator does not change if the product is a true zero. Flags and exceptions are not generated.

Syntax	Behavior

`Rx+=sfmpy(Rs,Rt):lib`	`round_to_nearest();`
	`infminusinf = ((isinf(Rx)) && (isinf(Rs*Rt)) && (Rs^Rx^Rt.31 != 0));`
	`infinp = (isinf(Rx)) || (isinf(Rt)) || (isinf(Rs));`
	`if (isnan(Rx) || isnan(Rs) || isnan(Rt)) Rx = NaN;`
	`tmp = fmaf(Rs,Rt,Rx);`
	`if (!((Rx == 0.0) && is_true_zero(Rs*Rt))) Rx = tmp;`
	`cancel_flags();`
	`if (isinf(Rx) && !infinp) Rx = Rx - 1;`
	`if (infminusinf) Rx = 0;`

`Rx-=sfmpy(Rs,Rt):lib`	`round_to_nearest();`
	`infminusinf = ((isinf(Rx)) && (isinf(Rs*Rt)) && (Rs^Rx^Rt.31 == 0));`
	`infinp = (isinf(Rx)) || (isinf(Rt)) || (isinf(Rs));`
	`if (isnan(Rx) || isnan(Rs) || isnan(Rt)) Rx = NaN;`
	`tmp = fmaf(-Rs,Rt,Rx);`
	`if (!((Rx == 0.0) && is_true_zero(Rs*Rt))) Rx = tmp;`
	`cancel_flags();`
	`if (isinf(Rx) && !infinp) Rx = Rx - 1;`
	`if (infminusinf) Rx = 0;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rx+=sfmpy(Rs,Rt):lib`	`Word32 Q6_R_sfmpyacc_RR_lib(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx-=sfmpy(Rs,Rt):lib`	`Word32 Q6_R_sfmpynac_RR_lib(Word32 Rx, Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**x**	**x**	**x**	**x**	**x**	Rx+=sfmpy(Rs,Rt):lib
1	1	1	0	1	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rx-=sfmpy(Rs,Rt):lib

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Create floating-point constant

Using ten bits of immediate, form a floating-point constant.

Syntax	Behavior

`Rd=sfmake(#u10):neg`	`Rd = (127 - 6) << 23;`
	`Rd += (#u << 17);`
	`Rd |= (1 << 31);`

`Rd=sfmake(#u10):pos`	`Rd = (127 - 6) << 23;` `Rd += #u << 17;`

`Rdd=dfmake(#u10):neg`	`Rdd = (1023ULL - 6) << 52;`
	`Rdd += (#u) << 46;`
	`Rdd |= ((1ULL) << 63);`

`Rdd=dfmake(#u10):pos`	`Rdd = (1023ULL - 6) << 52;` `Rdd += (#u) << 46;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=sfmake(#u10):neg`	`Word32 Q6_R_sfmake_I_neg(Word32 Iu10)`
`Rd=sfmake(#u10):pos`	`Word32 Q6_R_sfmake_I_pos(Word32 Iu10)`
`Rdd=dfmake(#u10):neg`	`Word64 Q6_P_dfmake_I_neg(Word32 Iu10)`
`Rdd=dfmake(#u10):pos`	`Word64 Q6_P_dfmake_I_pos(Word32 Iu10)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType												Parse								MinOp			d5					
1	1	0	1	0	1	1	0	0	0	**i**	-	-	-	-	-	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=sfmake(#u10):pos
1	1	0	1	0	1	1	0	0	1	**i**	-	-	-	-	-	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=sfmake(#u10):neg
ICLASS				RegType												Parse											d5					
1	1	0	1	1	0	0	1	0	0	**i**	-	-	-	-	-	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rdd=dfmake(#u10):pos
1	1	0	1	1	0	0	1	0	1	**i**	-	-	-	-	-	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rdd=dfmake(#u10):neg

Field name	Description
`RegType`	Register type
`MinOp`	Minor opcode
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d

Floating point maximum

Maximum of two floating point values. If one value is a NaN, the other is chosen.

Syntax	Behavior
`Rd=sfmax(Rs,Rt)`	`Rd = fmaxf(Rs,Rt);`
`Rdd=dfmax(Rss,Rtt)`	`Rdd = fmax(Rss,Rtt);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=sfmax(Rs,Rt)`	`Word32 Q6_R_sfmax_RR(Word32 Rs, Word32 Rt)`
`Rdd=dfmax(Rss,Rtt)`	`Word64 Q6_P_dfmax_PP(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=dfmax(Rss,Rtt)
1	1	1	0	1	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=sfmax(Rs,Rt)

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Floating point minimum

Minimum of two floating point values. If one value is a NaN, the other is chosen.

Syntax	Behavior
`Rd=sfmin(Rs,Rt)`	`Rd = fmin(Rs,Rt);`
`Rdd=dfmin(Rss,Rtt)`	`Rdd = fmin(Rss,Rtt);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=sfmin(Rs,Rt)`	`Word32 Q6_R_sfmin_RR(Word32 Rs, Word32 Rt)`
`Rdd=dfmin(Rss,Rtt)`	`Word64 Q6_P_dfmin_PP(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=dfmin(Rss,Rtt)
1	1	1	0	1	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=sfmin(Rs,Rt)

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Floating point multiply

Multiply two floating point values.

Syntax	Behavior
`Rd=sfmpy(Rs,Rt)`	`Rd=Rs*Rt;`
`Rdd=dfmpyfix(Rss,Rtt)`	`if (is_denormal(Rss) && (df_exponent(Rtt) >= 512) && is_normal(Rtt)) Rdd = Rss * 0x1.0p52; else if (is_denormal(Rtt) && (df_exponent(Rss) >= 512) && is_normal(Rss)) Rdd = Rss * 0x1.0p-52; else Rdd = Rss;`
`Rdd=dfmpyll(Rss,Rtt)`	`prod = (Rss.uw[0] * Rtt.uw[0]); Rdd = (prod >> 32) << 1; if (prod.uw[0] != 0) Rdd.0 = 1;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=sfmpy(Rs,Rt)`	`Word32 Q6_R_sfmpy_RR(Word32 Rs, Word32 Rt)`
`Rdd=dfmpyfix(Rss,Rtt)`	`Word64 Q6_P_dfmpyfix_PP(Word64 Rss, Word64 Rtt)`
`Rdd=dfmpyll(Rss,Rtt)`	`Word64 Q6_P_dfmpyll_PP(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=dfmpyfix(Rss,Rtt)
1	1	1	0	1	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=dfmpyll(Rss,Rtt)
1	1	1	0	1	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=sfmpy(Rs,Rt)

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Floating point reciprocal approximation

Provides an approximation of the reciprocal of the denominator (Rt), if combined with the appropriate fixup instructions. Certain values (such as infinities or zeros) in the numerator or denominator may yield values that are not reciprocal approximations, but yield the correct answer when combined with fixup instructions and the appropriate routines.

For compatibility, exact results of these instructions cannot be relied on. The precision of the approximation for this architecture and later is at least 6.6 bits.

Syntax	Behavior

`Rd,Pe=sfrecipa(Rs,Rt)`	`if ((Rs,Rt,Rd,adjust)=recip_common(Rs,Rt))`
	`{`
	`  Pe = adjust;`
	`  idx = (Rt >> 16) & 0x7f;`
	`  mant = (recip_lut[idx] << 15) | 1;`
	`  exp = 127 - (exponent(Rt) - 127) - 1;`
	`  Rd = -1**Rt.31 * 1.MANT * 2**(exp-BIAS);`
	`}`

Class: XTYPE (slots 2,3)

Notes

- This instruction provides a certain amount of accuracy. In future versions the accuracy may increase. For future compatibility, dependence on exact values must be avoided.
- The predicate generated by this instruction cannot be used as a .new predicate, nor can it be automatically AND'd with another predicate.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5						e2		d5						
1	1	1	0	1	0	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**e**	**e**	**d**	**d**	**d**	**d**	**d**	Rd,Pe=sfrecipa(Rs,Rt)

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
e2	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t

Floating point subtraction

Subtract two floating point values.

Syntax	Behavior
`Rd=sfsub(Rs,Rt)`	`Rd=Rs-Rt;`
`Rdd=dfsub(Rss,Rtt)`	`Rdd=Rss-Rtt;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=sfsub(Rs,Rt)`	`Word32 Q6_R_sfsub_RR(Word32 Rs, Word32 Rt)`
`Rdd=dfsub(Rss,Rtt)`	`Word64 Q6_P_dfsub_PP(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=dfsub(Rss,Rtt)
1	1	1	0	1	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=sfsub(Rs,Rt)

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

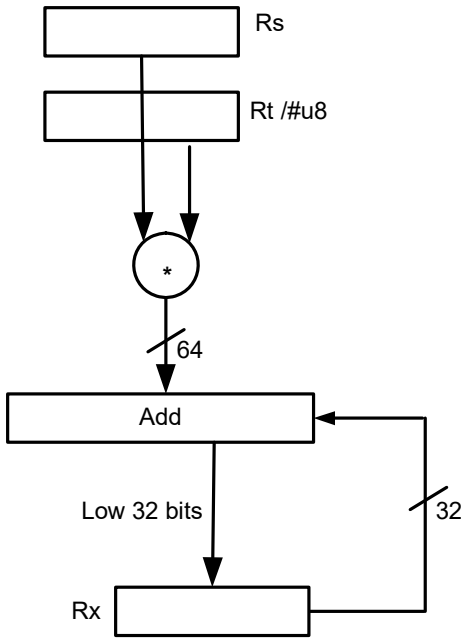
11.10.5 XTYPE MPY

The XTYPE MPY instruction subclass includes instructions that perform multiplication.

Multiply and use lower result

Multiply the signed 32-bit integer in Rs by either the signed 32-bit integer in Rt or an unsigned immediate value. The 64-bit result is optionally accumulated with the 32-bit destination, or added to an immediate. The least-significant 32-bits of the result are written to the single destination register.

This multiply produces the correct results for the ANSI C multiplication of two signed or unsigned integers with an integer result.



Syntax	Behavior
`Rd=+mpyi(Rs,#u8)`	`apply_extension(#u);` `Rd=Rs*#u;`
`Rd=-mpyi(Rs,#u8)`	`Rd=Rs*-#u;`
`Rd=add(#u6,mpyi(Rs,#U6))`	`apply_extension(#u);` `Rd = #u + Rs*#U;`
`Rd=add(#u6,mpyi(Rs,Rt))`	`apply_extension(#u);` `Rd = #u + Rs*Rt;`
`Rd=add(Ru,mpyi(#u6:2,Rs))`	`Rd = Ru + Rs*#u;`
`Rd=add(Ru,mpyi(Rs,#u6))`	`apply_extension(#u);` `Rd = Ru + Rs*#u;`

Syntax	Behavior

Rd=mpyi(Rs,#m9)	if ("((#m9<0) && (#m9>-256))") {
	Assembler mapped to: "Rd=-mpyi(Rs, #m9*(-1))";
	} else {
	Assembler mapped to: "Rd =+ mpyi(Rs, #m9)";
	}

Rd=mpyi(Rs,Rt)	Rd=Rs*Rt;
Rd=mpyui(Rs,Rt)	Assembler mapped to: "Rd=mpyi(Rs,Rt)"
Rx+=mpyi(Rs,#u8)	apply_extension(#u); Rx=Rx + (Rs*#u);
Rx+=mpyi(Rs,Rt)	Rx=Rx + Rs*Rt;
Rx-=mpyi(Rs,#u8)	apply_extension(#u); Rx=Rx - (Rs*#u);
Rx-=mpyi(Rs,Rt)	Rx=Rx - Rs*Rt;
Ry=add(Ru,mpyi(Ry,Rs))	Ry = Ru + Rs*Ry;

Class: XTYPE (slots 2,3)

Intrinsics

Rd=add(#u6,mpyi(Rs,#U6))	Word32 Q6_R_add_mpyi_IRI(Word32 Iu6, Word32 Rs, Word32 IU6)
Rd=add(#u6,mpyi(Rs,Rt))	Word32 Q6_R_add_mpyi_IRR(Word32 Iu6, Word32 Rs, Word32 Rt)
Rd=add(Ru,mpyi(#u6:2,Rs))	Word32 Q6_R_add_mpyi_RIR(Word32 Ru, Word32 Iu6_2, Word32 Rs)
Rd=add(Ru,mpyi(Rs,#u6))	Word32 Q6_R_add_mpyi_RRI(Word32 Ru, Word32 Rs, Word32 Iu6)
Rd=mpyi(Rs,#m9)	Word32 Q6_R_mpyi_RI(Word32 Rs, Word32 Im9)
Rd=mpyi(Rs,Rt)	Word32 Q6_R_mpyi_RR(Word32 Rs, Word32 Rt)
Rd=mpyui(Rs,Rt)	Word32 Q6_R_mpyui_RR(Word32 Rs, Word32 Rt)
Rx+=mpyi(Rs,#u8)	Word32 Q6_R_mpyiacc_RI(Word32 Rx, Word32 Rs, Word32 Iu8)
Rx+=mpyi(Rs,Rt)	Word32 Q6_R_mpyiacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpyi(Rs,#u8)	Word32 Q6_R_mpyinac_RI(Word32 Rx, Word32 Rs, Word32 Iu8)
Rx-=mpyi(Rs,Rt)	Word32 Q6_R_mpyinac_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Ry=add(Ru,mpyi(Ry,Rs))	Word32 Q6_R_add_mpyi_RRR(Word32 Ru, Word32 Ry, Word32 Rs)

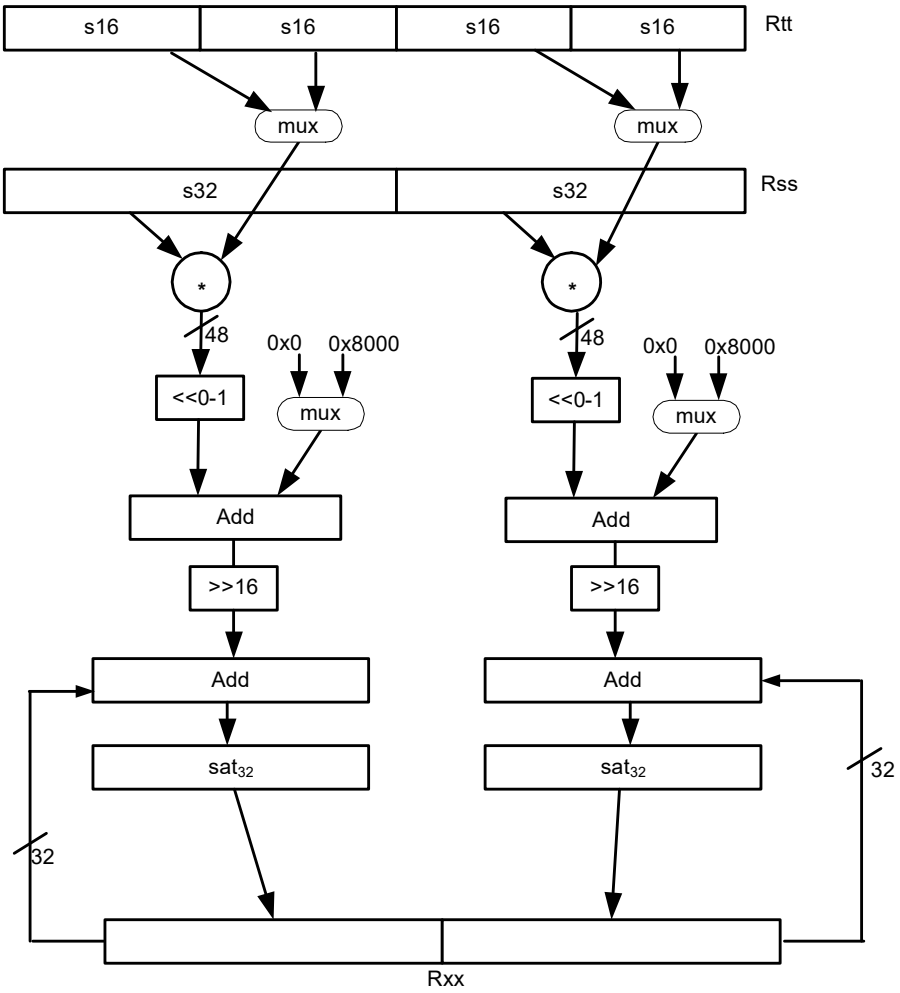
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	1	1	1	0	**i**	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=add(#u6,mpyi(Rs,Rt))
ICLASS				RegType							s5					Parse			d5													
1	1	0	1	1	0	0	0	**I**	**i**	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**d**	**d**	**d**	**d**	**d**	**i**	**i**	**i**	**I**	**I**	**I**	**I**	**I**	Rd=add(#u6,mpyi(Rs,#U6))
ICLASS				RegType							s5					Parse			d5								u5					
1	1	0	1	1	1	1	1	0	**i**	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**d**	**d**	**d**	**d**	**d**	**i**	**i**	**i**	**u**	**u**	**u**	**u**	**u**	Rd=add(Ru,mpyi(#u6:2,Rs))
1	1	0	1	1	1	1	1	1	**i**	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**d**	**d**	**d**	**d**	**d**	**i**	**i**	**i**	**u**	**u**	**u**	**u**	**u**	Rd=add(Ru,mpyi(Rs,#u6))
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	1	1	0	0	0	0	0	0	**-**	**-**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=+mpyi(Rs,#u8)
1	1	1	0	0	0	0	0	1	**-**	**-**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=-mpyi(Rs,#u8)
ICLASS				RegType				MajOp			s5					Parse									MinOp			x5					
1	1	1	0	0	0	0	1	0	**-**	**-**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**x**	**x**	**x**	**x**	**x**	Rx+=mpyi(Rs,#u8)
1	1	1	0	0	0	0	1	1	**-**	**-**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**x**	**x**	**x**	**x**	**x**	Rx-=mpyi(Rs,#u8)
ICLASS				RegType				MajOp			s5					Parse			y5								u5					
1	1	1	0	0	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**y**	**y**	**y**	**y**	**y**	**-**	**-**	**-**	**u**	**u**	**u**	**u**	**u**	Ry=add(Ru,mpyi(Ry,Rs))
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=mpyi(Rs,Rt)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rx+=mpyi(Rs,Rt)
1	1	1	0	1	1	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rx-=mpyi(Rs,Rt)

Field name	Description
RegType	Register type
MajOp	Major opcode
MinOp	Minor opcode
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
u5	Field to encode register u
x5	Field to encode register x
y5	Field to encode register y

Vector multiply word by signed half (32×16)

Perform mixed precision vector multiply operations. A 32-bit word from vector Rss is multiplied by a 16-bit halfword (either even or odd) from vector Rtt. The multiplication is performed as a signed 32×16, which produces a 48-bit result. This result is optionally scaled left by one bit. This result is then shifted right by 16 bits, optionally accumulated and then saturated to 32-bits. This operation is available in vector form (`vmpyweh`/`vmpywoh`) and non-vector form (multiply and use upper result).



Syntax	Behavior
`Rdd=vmpyweh(Rss, Rtt)[:<<1]:rnd:sat`	`Rdd.w[1]=sat32(((Rss.w[1] * Rtt.h[2])[<<1] +0x8000)>>16);` `Rdd.w[0]=sat32(((Rss.w[0] * Rtt.h[0])[<<1] +0x8000)>>16);`
`Rdd=vmpyweh(Rss, Rtt)[:<<1]:sat`	`Rdd.w[1]=sat32(((Rss.w[1] * Rtt.h[2])[<<1])>>16);` `Rdd.w[0]=sat32(((Rss.w[0] * Rtt.h[0])[<<1])>>16);`

Syntax	Behavior
`Rdd=vmpywoh(Rss, Rtt)[:<<1]:rnd:sat`	`Rdd.w[1]=sat32(((Rss.w[1] * Rtt.h[3])[<<1] +0x8000)>>16);` `Rdd.w[0]=sat32(((Rss.w[0] * Rtt.h[1])[<<1] +0x8000)>>16);`
`Rdd=vmpywoh(Rss, Rtt)[:<<1]:sat`	`Rdd.w[1]=sat32(((Rss.w[1] * Rtt.h[3])[<<1])>>16);` `Rdd.w[0]=sat32(((Rss.w[0] * Rtt.h[1])[<<1])>>16);`
`Rxx+=vmpyweh(Rss, Rtt)[:<<1]:rnd:sat`	`Rxx.w[1]=sat32(Rxx.w[1] + (((Rss.w[1] * Rtt.h[2])[<<1]+0x8000)>>16));` `Rxx.w[0]=sat32(Rxx.w[0] + (((Rss.w[0] * Rtt.h[0])[<<1]+0x8000)>>16));`
`Rxx+=vmpyweh(Rss, Rtt)[:<<1]:sat`	`Rxx.w[1]=sat32(Rxx.w[1] + (((Rss.w[1] * Rtt.h[2])[<<1])>>16));` `Rxx.w[0]=sat32(Rxx.w[0] + (((Rss.w[0] * Rtt.h[0])[<<1])>>16));`
`Rxx+=vmpywoh(Rss, Rtt)[:<<1]:rnd:sat`	`Rxx.w[1]=sat32(Rxx.w[1] + (((Rss.w[1] * Rtt.h[3])[<<1]+0x8000)>>16));` `Rxx.w[0]=sat32(Rxx.w[0] + (((Rss.w[0] * Rtt.h[1])[<<1]+0x8000)>>16 ));`
`Rxx+=vmpywoh(Rss, Rtt)[:<<1]:sat`	`Rxx.w[1]=sat32(Rxx.w[1] + (((Rss.w[1] * Rtt.h[3])[<<1])>>16));` `Rxx.w[0]=sat32(Rxx.w[0] + (((Rss.w[0] * Rtt.h[1])[<<1])>>16 ));`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd= vmpyweh(Rss,Rtt):<<1:rnd:sat`	`Word64 Q6_P_vmpyweh_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vmpyweh(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vmpyweh_PP_s1_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vmpyweh(Rss,Rtt):rnd:sat`	`Word64 Q6_P_vmpyweh_PP_rnd_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vmpyweh(Rss,Rtt):sat`	`Word64 Q6_P_vmpyweh_PP_sat(Word64 Rss, Word64 Rtt)`
`Rdd= vmpywoh(Rss,Rtt):<<1:rnd:sat`	`Word64 Q6_P_vmpywoh_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vmpywoh(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vmpywoh_PP_s1_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vmpywoh(Rss,Rtt):rnd:sat`	`Word64 Q6_P_vmpywoh_PP_rnd_sat(Word64 Rss, Word64 Rtt)`

`Rdd=vmpywoh(Rss,Rtt):sat`	`Word64 Q6_P_vmpywoh_PP_sat(Word64 Rss, Word64 Rtt)`
`Rxx+=vmpyweh(Rss,Rtt): << 1:rnd:sat`	`Word64 Q6_P_vmpywehacc_PP_s1_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx += vmpyweh(Rss,Rtt): << 1:sat`	`Word64 Q6_P_vmpywehacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx += vmpyweh(Rss, Rtt):rnd:sat`	`Word64 Q6_P_vmpywehacc_PP_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vmpyweh(Rss,Rtt):sat`	`Word64 Q6_P_vmpywehacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx += vmpywoh(Rss, Rtt):<<1:rnd:sat`	`Word64 Q6_P_vmpywohacc_PP_s1_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx += vmpywoh(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vmpywohacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+= vmpywoh(Rss,Rtt):rnd:sat`	`Word64 Q6_P_vmpywohacc_PP_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vmpywoh(Rss,Rtt):sat`	`Word64 Q6_P_vmpywohacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`

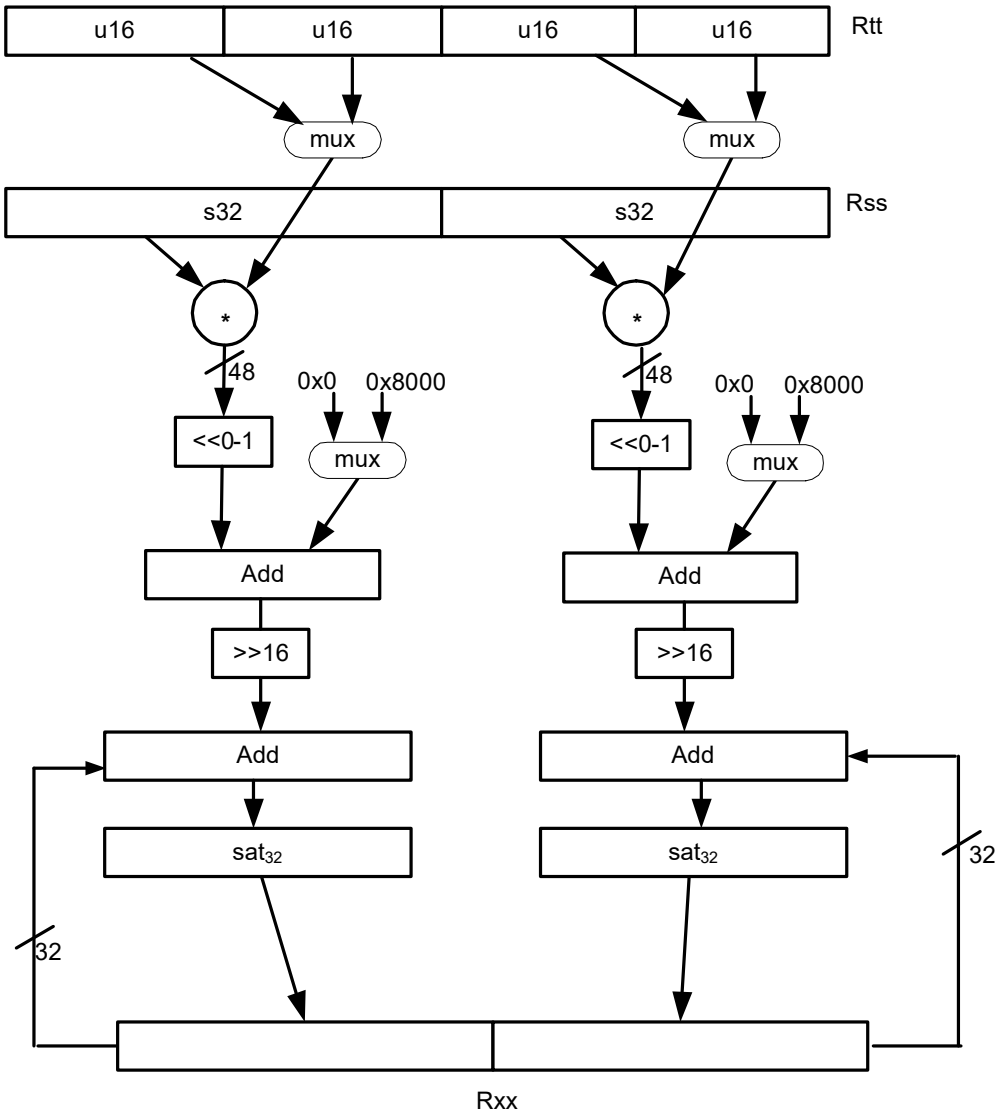
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpyweh(Rss,Rtt)[:<<N]:sat
1	1	1	0	1	0	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpywoh(Rss,Rtt)[:<<N]:sat
1	1	1	0	1	0	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpyweh(Rss,Rtt)[:<<N]:rnd:sat
1	1	1	0	1	0	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpywoh(Rss,Rtt)[:<<N]:rnd:sat
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpyweh(Rss,Rtt)[:<<N]:sat
1	1	1	0	1	0	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpywoh(Rss,Rtt)[:<<N]:sat
1	1	1	0	1	0	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpyweh(Rss,Rtt)[:<<N]:rnd:sat
1	1	1	0	1	0	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpywoh(Rss,Rtt)[:<<N]:rnd:sat

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector multiply word by unsigned half (32×16)

Perform mixed precision vector multiply operations. A 32-bit signed word from vector Rss is multiplied by a 16-bit unsigned halfword (either odd or even) from vector Rtt. This multiplication produces a 48-bit result. This result is optionally scaled left by one bit, and then a rounding constant is optionally added to the lower 16-bits. This result is then shifted right by 16 bits, optionally accumulated and then saturated to 32-bits. This is a dual vector operation and is performed for both high and low word of Rss.



Syntax	Behavior
`Rdd= vmpyweuh(Rss,Rtt)[:<<1]:rnd:sat`	`Rdd.w[1]=sat32(((Rss.w[1] * Rtt.uh[2])[<<1]+0x8000) >> 16);` `Rdd.w[0]=sat32(((Rss.w[0] * Rtt.uh[0])[<<1]+0x8000) >> 16);`

Syntax	Behavior
Rdd= vmpyweuh(Rss,Rtt)[:<<1]:sat	Rdd.w[1]=sat$_{32}$(((Rss.w[1] * Rtt.uh[2])[<<1])>>16); Rdd.w[0]=sat$_{32}$(((Rss.w[0] * Rtt.uh[0])[<<1])>>16);
Rdd= vmpywouh(Rss,Rtt)[:<<1]:rnd:sat	Rdd.w[1]=sat$_{32}$(((Rss.w[1] * Rtt.uh[3])[<<1]+0x8000)>>16); Rdd.w[0]=sat$_{32}$(((Rss.w[0] * Rtt.uh[1])[<<1]+0x8000)>>16);
Rdd= vmpywouh(Rss,Rtt)[:<<1]:sat	Rdd.w[1]=sat$_{32}$(((Rss.w[1] * Rtt.uh[3])[<<1])>>16); Rdd.w[0]=sat$_{32}$(((Rss.w[0] * Rtt.uh[1])[<<1])>>16);
Rxx+= vmpyweuh(Rss,Rtt)[:<<1]:rnd:sat	Rxx.w[1]=sat$_{32}$(Rxx.w[1] + (((Rss.w[1] * Rtt.uh[2])[<<1]+0x8000)>>16)); Rxx.w[0]=sat$_{32}$(Rxx.w[0] + (((Rss.w[0] * Rtt.uh[0])[<<1]+0x8000)>>16));
Rxx+= vmpyweuh(Rss,Rtt)[:<<1]:sat	Rxx.w[1]=sat$_{32}$(Rxx.w[1] + (((Rss.w[1] * Rtt.uh[2])[<<1])>>16)); Rxx.w[0]=sat$_{32}$(Rxx.w[0] + (((Rss.w[0] * Rtt.uh[0])[<<1])>>16));
Rxx+= vmpywouh(Rss,Rtt)[:<<1]:rnd:sat	Rxx.w[1]=sat$_{32}$(Rxx.w[1] + (((Rss.w[1] * Rtt.uh[3])[<<1]+0x8000)>>16)); Rxx.w[0]=sat$_{32}$(Rxx.w[0] + (((Rss.w[0] * Rtt.uh[1])[<<1]+0x8000)>>16));
Rxx+= vmpywouh(Rss,Rtt)[:<<1]:sat	Rxx.w[1]=sat$_{32}$(Rxx.w[1] + (((Rss.w[1] * Rtt.uh[3])[<<1])>>16)); Rxx.w[0]=sat$_{32}$(Rxx.w[0] + (((Rss.w[0] * Rtt.uh[1])[<<1])>>16));

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

Rdd= vmpyweuh(Rss, Rtt):<<1:rnd:sat	Word64 Q6_P_vmpyweuh_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt)
Rdd= vmpyweuh(Rss,Rtt):<<1:sat	Word64 Q6_P_vmpyweuh_PP_s1_sat(Word64 Rss, Word64 Rtt)
Rdd= vmpyweuh(Rss,Rtt):rnd:sat	Word64 Q6_P_vmpyweuh_PP_rnd_sat(Word64 Rss, Word64 Rtt)

`Rdd=vmpyweuh(Rss,Rtt):sat`	`Word64 Q6_P_vmpyweuh_PP_sat(Word64 Rss, Word64 Rtt)`
`Rdd= vmpywouh(Rss, Rtt):<<1:rnd:sat`	`Word64 Q6_P_vmpywouh_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt)`
`Rdd= vmpywouh(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vmpywouh_PP_s1_sat(Word64 Rss, Word64 Rtt)`
`Rdd= vmpywouh(Rss,Rtt):rnd:sat`	`Word64 Q6_P_vmpywouh_PP_rnd_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vmpywouh(Rss,Rtt):sat`	`Word64 Q6_P_vmpywouh_PP_sat(Word64 Rss, Word64 Rtt)`
`Rxx+= vmpyweuh(Rss, Rtt):<<1:rnd:sat`	`Word64 Q6_P_vmpyweuhacc_PP_s1_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+= vmpyweuh(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vmpyweuhacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+= vmpyweuh(Rss,Rtt):rnd:sat`	`Word64 Q6_P_vmpyweuhacc_PP_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vmpyweuh(Rss,Rtt):sat`	`Word64 Q6_P_vmpyweuhacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+= vmpywouh(Rss, Rtt):<<1:rnd:sat`	`Word64 Q6_P_vmpywouhacc_PP_s1_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+= vmpywouh(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vmpywouhacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+= vmpywouh(Rss,Rtt):rnd:sat`	`Word64 Q6_P_vmpywouhacc_PP_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vmpywouh(Rss,Rtt):sat`	`Word64 Q6_P_vmpywouhacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`

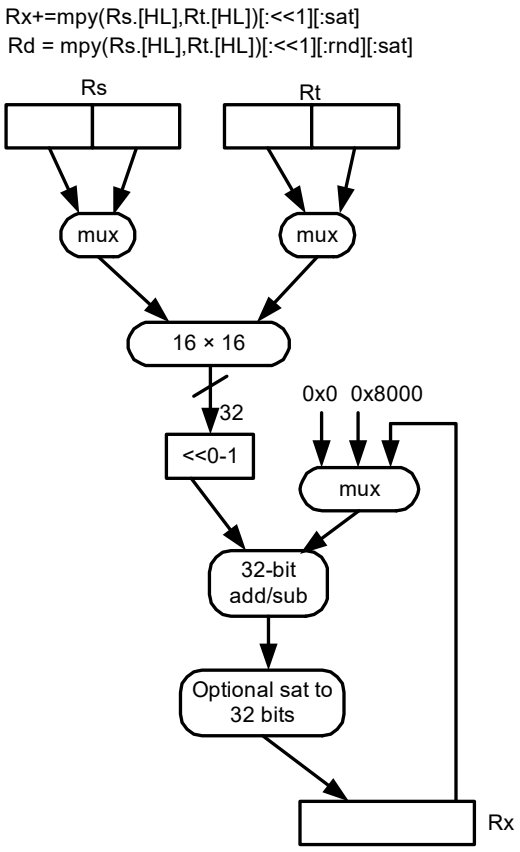
Encoding

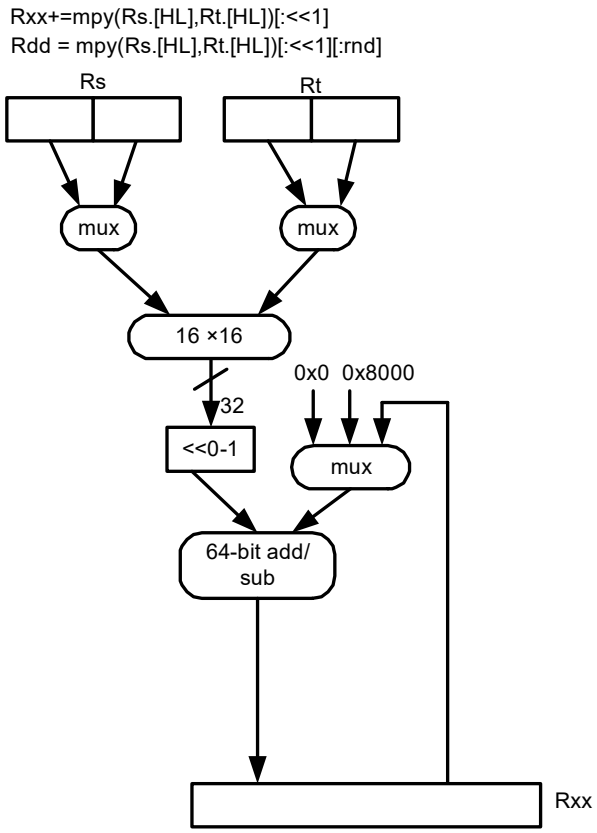
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpyweuh(Rss,Rtt)[:<<N]:sat
1	1	1	0	1	0	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpywouh(Rss,Rtt)[:<<N]:sat
1	1	1	0	1	0	0	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpyweuh(Rss,Rtt)[:<<N]:rnd:sat
1	1	1	0	1	0	0	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpywouh(Rss,Rtt)[:<<N]:rnd:sat
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpyweuh(Rss,Rtt)[:<<N]:sat
1	1	1	0	1	0	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpywouh(Rss,Rtt)[:<<N]:sat
1	1	1	0	1	0	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpyweuh(Rss,Rtt)[:<<N]:rnd:sat
1	1	1	0	1	0	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpywouh(Rss,Rtt)[:<<N]:rnd:sat

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Multiply signed halfwords

Multiply two signed halfwords. Optionally shift the multiplier result by 1 bit. This result can be accumulated or rounded. The destination/accumulator can be either 32 or 64-bits. For 32-bit results, saturation is optional.





Syntax	Behavior
`Rd=mpy(Rs.[HL],Rt.[HL])[:<< 1][:rnd][:sat]`	`Rd =[sat32]([round]((Rs.h[01] * Rt.h[01])[<< 1]));`
`Rdd=mpy(Rs.[HL],Rt.[HL])[:<< 1][:rnd]`	`Rdd =[round]((Rs.h[01] * Rt.h[01])[<<1]);`
`Rx+=mpy(Rs.[HL],Rt.[HL])[:<< 1][:sat]`	`Rx =[sat32](Rx+ (Rs.h[01] * Rt.h[01])[<<1]);`
`Rx-=mpy(Rs.[HL],Rt.[HL])[:<< 1][:sat]`	`Rx =[sat32](Rx- (Rs.h[01] * Rt.h[01])[<< 1]);`
`Rxx+=mpy(Rs.[HL],Rt.[HL])[:< < 1]`	`Rxx =Rxx+ (Rs.h[01] * Rt.h[01])[<<1];`
`Rxx-= mpy(Rs.[HL], Rt.[HL])[:<< 1]`	`Rxx =Rxx- (Rs.h[01] * Rt.h[01])[<<1];`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

Rd=mpy(Rs.H,Rt.H)	Word32 Q6_R_mpy_RhRh(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.H):<< 1	Word32 Q6_R_mpy_RhRh_s1(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.H):<< 1:rnd	Word32 Q6_R_mpy_RhRh_s1_rnd(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.H):<< 1:rnd:sat	Word32 Q6_R_mpy_RhRh_s1_rnd_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.H):<< 1:sat	Word32 Q6_R_mpy_RhRh_s1_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.H):rnd	Word32 Q6_R_mpy_RhRh_rnd(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.H):rnd:sat	Word32 Q6_R_mpy_RhRh_rnd_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.H):sat	Word32 Q6_R_mpy_RhRh_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.L)	Word32 Q6_R_mpy_RhRl(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.L):<<1	Word32 Q6_R_mpy_RhRl_s1(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.L):<<1:rnd	Word32 Q6_R_mpy_RhRl_s1_rnd(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.L):<< 1:rnd:sat	Word32 Q6_R_mpy_RhRl_s1_rnd_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.L):<< 1:sat	Word32 Q6_R_mpy_RhRl_s1_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.L):rnd	Word32 Q6_R_mpy_RhRl_rnd(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.L):rnd:sat	Word32 Q6_R_mpy_RhRl_rnd_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.L):sat	Word32 Q6_R_mpy_RhRl_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.H)	Word32 Q6_R_mpy_RlRh(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.H):<<1	Word32 Q6_R_mpy_RlRh_s1(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.H):<<1:rnd	Word32 Q6_R_mpy_RlRh_s1_rnd(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.H):<< 1:rnd:sat	Word32 Q6_R_mpy_RlRh_s1_rnd_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.H):<< 1:sat	Word32 Q6_R_mpy_RlRh_s1_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.H):rnd	Word32 Q6_R_mpy_RlRh_rnd(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.H):rnd:sat	Word32 Q6_R_mpy_RlRh_rnd_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.H):sat	Word32 Q6_R_mpy_RlRh_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.L)	Word32 Q6_R_mpy_RlRl(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.L):<<1	Word32 Q6_R_mpy_RlRl_s1(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.L):<<1:rnd	Word32 Q6_R_mpy_RlRl_s1_rnd(Word32 Rs, Word32 Rt)

Rd=mpy(Rs.L,Rt.L):<<1:rnd:sat	Word32 Q6_R_mpy_RlRl_s1_rnd_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.L):<<1:sat	Word32 Q6_R_mpy_RlRl_s1_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.L):rnd	Word32 Q6_R_mpy_RlRl_rnd(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.L):rnd:sat	Word32 Q6_R_mpy_RlRl_rnd_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.L):sat	Word32 Q6_R_mpy_RlRl_sat(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.H,Rt.H)	Word64 Q6_P_mpy_RhRh(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.H,Rt.H):<<1	Word64 Q6_P_mpy_RhRh_s1(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.H,Rt.H):<<1:rnd	Word64 Q6_P_mpy_RhRh_s1_rnd(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.H,Rt.H):rnd	Word64 Q6_P_mpy_RhRh_rnd(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.H,Rt.L)	Word64 Q6_P_mpy_RhRl(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.H,Rt.L):<<1	Word64 Q6_P_mpy_RhRl_s1(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.H,Rt.L):<<1:rnd	Word64 Q6_P_mpy_RhRl_s1_rnd(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.H,Rt.L):rnd	Word64 Q6_P_mpy_RhRl_rnd(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.L,Rt.H)	Word64 Q6_P_mpy_RlRh(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.L,Rt.H):<<1	Word64 Q6_P_mpy_RlRh_s1(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.L,Rt.H):<<1:rnd	Word64 Q6_P_mpy_RlRh_s1_rnd(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.L,Rt.H):rnd	Word64 Q6_P_mpy_RlRh_rnd(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.L,Rt.L)	Word64 Q6_P_mpy_RlRl(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.L,Rt.L):<<1	Word64 Q6_P_mpy_RlRl_s1(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.L,Rt.L):<<1:rnd	Word64 Q6_P_mpy_RlRl_s1_rnd(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.L,Rt.L):rnd	Word64 Q6_P_mpy_RlRl_rnd(Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.H,Rt.H)	Word32 Q6_R_mpyacc_RhRh(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.H,Rt.H):<<1	Word32 Q6_R_mpyacc_RhRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.H,Rt.H):<<1:sat	Word32 Q6_R_mpyacc_RhRh_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.H,Rt.H):sat	Word32 Q6_R_mpyacc_RhRh_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.H,Rt.L)	Word32 Q6_R_mpyacc_RhRl(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.H,Rt.L):<<1	Word32 Q6_R_mpyacc_RhRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.H,Rt.L):<<1:sat	Word32 Q6_R_mpyacc_RhRl_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.H,Rt.L):sat	Word32 Q6_R_mpyacc_RhRl_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.L,Rt.H)	Word32 Q6_R_mpyacc_RlRh(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.L,Rt.H):<<1	Word32 Q6_R_mpyacc_RlRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)

Rx+=mpy(Rs.L,Rt.H):<<1:sat	Word32 Q6_R_mpyacc_RlRh_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.L,Rt.H):sat	Word32 Q6_R_mpyacc_RlRh_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.L,Rt.L)	Word32 Q6_R_mpyacc_RlRl(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.L,Rt.L):<<1	Word32 Q6_R_mpyacc_RlRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.L,Rt.L):<<1:sat	Word32 Q6_R_mpyacc_RlRl_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.L,Rt.L):sat	Word32 Q6_R_mpyacc_RlRl_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.H,Rt.H)	Word32 Q6_R_mpynac_RhRh(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.H,Rt.H):<<1	Word32 Q6_R_mpynac_RhRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.H,Rt.H):<<1:sat	Word32 Q6_R_mpynac_RhRh_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.H,Rt.H):sat	Word32 Q6_R_mpynac_RhRh_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.H,Rt.L)	Word32 Q6_R_mpynac_RhRl(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.H,Rt.L):<<1	Word32 Q6_R_mpynac_RhRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.H,Rt.L):<<1:sat	Word32 Q6_R_mpynac_RhRl_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.H,Rt.L):sat	Word32 Q6_R_mpynac_RhRl_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.L,Rt.H)	Word32 Q6_R_mpynac_RlRh(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.L,Rt.H):<<1	Word32 Q6_R_mpynac_RlRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.L,Rt.H):<<1:sat	Word32 Q6_R_mpynac_RlRh_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.L,Rt.H):sat	Word32 Q6_R_mpynac_RlRh_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.L,Rt.L)	Word32 Q6_R_mpynac_RlRl(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.L,Rt.L):<<1	Word32 Q6_R_mpynac_RlRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.L,Rt.L):<<1:sat	Word32 Q6_R_mpynac_RlRl_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.L,Rt.L):sat	Word32 Q6_R_mpynac_RlRl_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rxx+=mpy(Rs.H,Rt.H)	Word64 Q6_P_mpyacc_RhRh(Word64 Rxx, Word32 Rs, Word32 Rt)

`Rxx+=mpy(Rs.H,Rt.H):<<1`	`Word64 Q6_P_mpyacc_RhRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=mpy(Rs.H,Rt.L)`	`Word64 Q6_P_mpyacc_RhRl(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=mpy(Rs.H,Rt.L):<<1`	`Word64 Q6_P_mpyacc_RhRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=mpy(Rs.L,Rt.H)`	`Word64 Q6_P_mpyacc_RlRh(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=mpy(Rs.L,Rt.H):<<1`	`Word64 Q6_P_mpyacc_RlRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=mpy(Rs.L,Rt.L)`	`Word64 Q6_P_mpyacc_RlRl(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=mpy(Rs.L,Rt.L):<<1`	`Word64 Q6_P_mpyacc_RlRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpy(Rs.H,Rt.H)`	`Word64 Q6_P_mpynac_RhRh(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpy(Rs.H,Rt.H):<<1`	`Word64 Q6_P_mpynac_RhRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpy(Rs.H,Rt.L)`	`Word64 Q6_P_mpynac_RhRl(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpy(Rs.H,Rt.L):<<1`	`Word64 Q6_P_mpynac_RhRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpy(Rs.L,Rt.H)`	`Word64 Q6_P_mpynac_RlRh(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpy(Rs.L,Rt.H):<<1`	`Word64 Q6_P_mpynac_RlRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpy(Rs.L,Rt.L)`	`Word64 Q6_P_mpynac_RlRl(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpy(Rs.L,Rt.L):<<1`	`Word64 Q6_P_mpynac_RlRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	0	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=mpy(Rs.L,Rt.L)[:<<N]
1	1	1	0	0	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=mpy(Rs.L,Rt.H)[:<<N]
1	1	1	0	0	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=mpy(Rs.H,Rt.L)[:<<N]
1	1	1	0	0	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=mpy(Rs.H,Rt.H)[:<<N]
1	1	1	0	0	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=mpy(Rs.L,Rt.L)[:<<N]:rnd
1	1	1	0	0	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=mpy(Rs.L,Rt.H)[:<<N]:rnd
1	1	1	0	0	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=mpy(Rs.H,Rt.L)[:<<N]:rnd
1	1	1	0	0	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=mpy(Rs.H,Rt.H)[:<<N]:rnd
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					

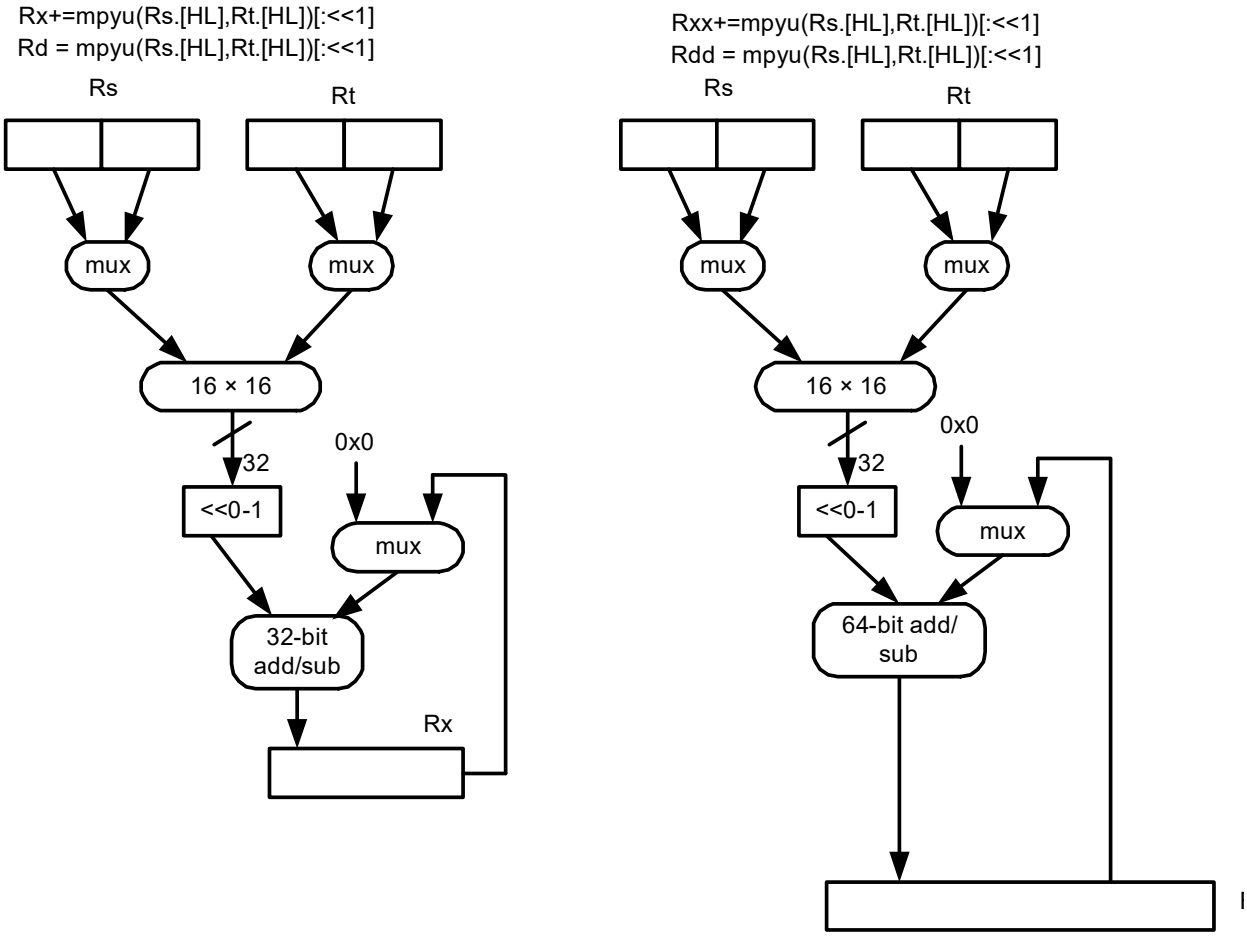
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	0	0	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=mpy(Rs.L,Rt.L)[:<<N]
1	1	1	0	0	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=mpy(Rs.L,Rt.H)[:<<N]
1	1	1	0	0	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=mpy(Rs.H,Rt.L)[:<<N]
1	1	1	0	0	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rxx+=mpy(Rs.H,Rt.H)[:<<N]
1	1	1	0	0	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rxx-=mpy(Rs.L,Rt.L)[:<<N]
1	1	1	0	0	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx-=mpy(Rs.L,Rt.H)[:<<N]
1	1	1	0	0	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rxx-=mpy(Rs.H,Rt.L)[:<<N]
1	1	1	0	0	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rxx-=mpy(Rs.H,Rt.H)[:<<N]
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.L,Rt.L)[:<<N]
1	1	1	0	1	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.L,Rt.H)[:<<N]
1	1	1	0	1	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.H,Rt.L)[:<<N]
1	1	1	0	1	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.H,Rt.H)[:<<N]
1	1	1	0	1	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.L,Rt.L)[:<<N]:sat
1	1	1	0	1	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.L,Rt.H)[:<<N]:sat
1	1	1	0	1	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.H,Rt.L)[:<<N]:sat
1	1	1	0	1	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.H,Rt.H)[:<<N]:sat
1	1	1	0	1	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.L,Rt.L)[:<<N]:rnd
1	1	1	0	1	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.L,Rt.H)[:<<N]:rnd
1	1	1	0	1	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.H,Rt.L)[:<<N]:rnd
1	1	1	0	1	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.H,Rt.H)[:<<N]:rnd
1	1	1	0	1	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.L,Rt.L)[:<<N]:rnd:sat
1	1	1	0	1	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.L,Rt.H)[:<<N]:rnd:sat
1	1	1	0	1	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.H,Rt.L)[:<<N]:rnd:sat
1	1	1	0	1	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.H,Rt.H)[:<<N]:rnd:sat
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rx+=mpy(Rs.L,Rt.L)[:<<N]
1	1	1	0	1	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rx+=mpy(Rs.L,Rt.H)[:<<N]
1	1	1	0	1	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rx+=mpy(Rs.H,Rt.L)[:<<N]
1	1	1	0	1	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rx+=mpy(Rs.H,Rt.H)[:<<N]
1	1	1	0	1	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**x**	**x**	**x**	**x**	**x**	Rx+=mpy(Rs.L,Rt.L)[:<<N]:sat
1	1	1	0	1	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**x**	**x**	**x**	**x**	**x**	Rx+=mpy(Rs.L,Rt.H)[:<<N]:sat
1	1	1	0	1	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**x**	**x**	**x**	**x**	**x**	Rx+=mpy(Rs.H,Rt.L)[:<<N]:sat
1	1	1	0	1	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rx+=mpy(Rs.H,Rt.H)[:<<N]:sat
1	1	1	0	1	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rx-=mpy(Rs.L,Rt.L)[:<<N]
1	1	1	0	1	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rx-=mpy(Rs.L,Rt.H)[:<<N]
1	1	1	0	1	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rx-=mpy(Rs.H,Rt.L)[:<<N]

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	0	1	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rx-=mpy(Rs.H,Rt.H)[:<<N]
1	1	1	0	1	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	1	0	0	**x**	**x**	**x**	**x**	**x**	Rx-=mpy(Rs.L,Rt.L)[:<<N]:sat
1	1	1	0	1	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	1	0	1	**x**	**x**	**x**	**x**	**x**	Rx-=mpy(Rs.L,Rt.H)[:<<N]:sat
1	1	1	0	1	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	1	1	0	**x**	**x**	**x**	**x**	**x**	Rx-=mpy(Rs.H,Rt.L)[:<<N]:sat
1	1	1	0	1	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rx-=mpy(Rs.H,Rt.H)[:<<N]:sat

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
sH	Rs is high
tH	Rt is high
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Multiply unsigned halfwords

Multiply two unsigned halfwords. Scale the result by 0 to 3 bits. Optionally, add or subtract the result from the accumulator.



Syntax	Behavior
`Rd = mpyu(Rs.[HL], Rt.[HL])[:<<1]`	`Rd=(Rs.uh[01] * Rt.uh[01])[<<1];`
`Rdd = mpyu(Rs.[HL], Rt.[HL])[:<<1]`	`Rdd=(Rs.uh[01] * Rt.uh[01])[<<1];`
`Rx += mpyu(Rs.[HL], Rt.[HL])[:<<1]`	`Rx=Rx+ (Rs.uh[01] * Rt.uh[01])[<<1];`
`Rx-= mpyu(Rs.[HL], Rt.[HL])[:<<1]`	`Rx=Rx- (Rs.uh[01] * Rt.uh[01])[<<1];`
`Rxx += mpyu(Rs.[HL], Rt.[HL])[:<< 1]`	`Rxx=Rxx+ (Rs.uh[01] * Rt.uh[01])[<<1];`
`Rxx -= mpyu(Rs.[HL], Rt.[HL])[:<< 1]`	`Rxx=Rxx- (Rs.uh[01] * Rt.uh[01])[<<1];`

Class: XTYPE (slots 2,3)

Intrinsics

Rd=mpyu(Rs.H,Rt.H)	UWord32 Q6_R_mpyu_RhRh(Word32 Rs, Word32 Rt)
Rd=mpyu(Rs.H,Rt.H):<<1	UWord32 Q6_R_mpyu_RhRh_s1(Word32 Rs, Word32 Rt)
Rd=mpyu(Rs.H,Rt.L)	UWord32 Q6_R_mpyu_RhRl(Word32 Rs, Word32 Rt)
Rd=mpyu(Rs.H,Rt.L):<<1	UWord32 Q6_R_mpyu_RhRl_s1(Word32 Rs, Word32 Rt)
Rd=mpyu(Rs.L,Rt.H)	UWord32 Q6_R_mpyu_RlRh(Word32 Rs, Word32 Rt)
Rd=mpyu(Rs.L,Rt.H):<<1	UWord32 Q6_R_mpyu_RlRh_s1(Word32 Rs, Word32 Rt)
Rd=mpyu(Rs.L,Rt.L)	UWord32 Q6_R_mpyu_RlRl(Word32 Rs, Word32 Rt)
Rd=mpyu(Rs.L,Rt.L):<<1	UWord32 Q6_R_mpyu_RlRl_s1(Word32 Rs, Word32 Rt)
Rdd=mpyu(Rs.H,Rt.H)	UWord64 Q6_P_mpyu_RhRh(Word32 Rs, Word32 Rt)
Rdd=mpyu(Rs.H,Rt.H):<<1	UWord64 Q6_P_mpyu_RhRh_s1(Word32 Rs, Word32 Rt)
Rdd=mpyu(Rs.H,Rt.L)	UWord64 Q6_P_mpyu_RhRl(Word32 Rs, Word32 Rt)
Rdd=mpyu(Rs.H,Rt.L):<<1	UWord64 Q6_P_mpyu_RhRl_s1(Word32 Rs, Word32 Rt)
Rdd=mpyu(Rs.L,Rt.H)	UWord64 Q6_P_mpyu_RlRh(Word32 Rs, Word32 Rt)
Rdd=mpyu(Rs.L,Rt.H):<<1	UWord64 Q6_P_mpyu_RlRh_s1(Word32 Rs, Word32 Rt)
Rdd=mpyu(Rs.L,Rt.L)	UWord64 Q6_P_mpyu_RlRl(Word32 Rs, Word32 Rt)
Rdd=mpyu(Rs.L,Rt.L):<<1	UWord64 Q6_P_mpyu_RlRl_s1(Word32 Rs, Word32 Rt)
Rx+=mpyu(Rs.H,Rt.H)	Word32 Q6_R_mpyuacc_RhRh(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpyu(Rs.H,Rt.H):<<1	Word32 Q6_R_mpyuacc_RhRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpyu(Rs.H,Rt.L)	Word32 Q6_R_mpyuacc_RhRl(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpyu(Rs.H,Rt.L):<<1	Word32 Q6_R_mpyuacc_RhRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpyu(Rs.L,Rt.H)	Word32 Q6_R_mpyuacc_RlRh(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpyu(Rs.L,Rt.H):<<1	Word32 Q6_R_mpyuacc_RlRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpyu(Rs.L,Rt.L)	Word32 Q6_R_mpyuacc_RlRl(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpyu(Rs.L,Rt.L):<<1	Word32 Q6_R_mpyuacc_RlRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpyu(Rs.H,Rt.H)	Word32 Q6_R_mpyunac_RhRh(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpyu(Rs.H,Rt.H):<<1	Word32 Q6_R_mpyunac_RhRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpyu(Rs.H,Rt.L)	Word32 Q6_R_mpyunac_RhRl(Word32 Rx, Word32 Rs, Word32 Rt)

Rx-=mpyu(Rs.H,Rt.L):<<1	Word32 Q6_R_mpyunac_RhRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpyu(Rs.L,Rt.H)	Word32 Q6_R_mpyunac_RlRh(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpyu(Rs.L,Rt.H):<<1	Word32 Q6_R_mpyunac_RlRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpyu(Rs.L,Rt.L)	Word32 Q6_R_mpyunac_RlRl(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpyu(Rs.L,Rt.L):<<1	Word32 Q6_R_mpyunac_RlRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rxx+=mpyu(Rs.H,Rt.H)	Word64 Q6_P_mpyuacc_RhRh(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx+=mpyu(Rs.H,Rt.H):<<1	Word64 Q6_P_mpyuacc_RhRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx+=mpyu(Rs.H,Rt.L)	Word64 Q6_P_mpyuacc_RhRl(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx+=mpyu(Rs.H,Rt.L):<<1	Word64 Q6_P_mpyuacc_RhRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx+=mpyu(Rs.L,Rt.H)	Word64 Q6_P_mpyuacc_RlRh(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx+=mpyu(Rs.L,Rt.H):<<1	Word64 Q6_P_mpyuacc_RlRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx+=mpyu(Rs.L,Rt.L)	Word64 Q6_P_mpyuacc_RlRl(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx+=mpyu(Rs.L,Rt.L):<<1	Word64 Q6_P_mpyuacc_RlRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx-=mpyu(Rs.H,Rt.H)	Word64 Q6_P_mpyunac_RhRh(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx-=mpyu(Rs.H,Rt.H):<<1	Word64 Q6_P_mpyunac_RhRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx-=mpyu(Rs.H,Rt.L)	Word64 Q6_P_mpyunac_RhRl(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx-=mpyu(Rs.H,Rt.L):<<1	Word64 Q6_P_mpyunac_RhRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx-=mpyu(Rs.L,Rt.H)	Word64 Q6_P_mpyunac_RlRh(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx-=mpyu(Rs.L,Rt.H):<<1	Word64 Q6_P_mpyunac_RlRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx-=mpyu(Rs.L,Rt.L)	Word64 Q6_P_mpyunac_RlRl(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx-=mpyu(Rs.L,Rt.L):<<1	Word64 Q6_P_mpyunac_RlRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)

Encoding

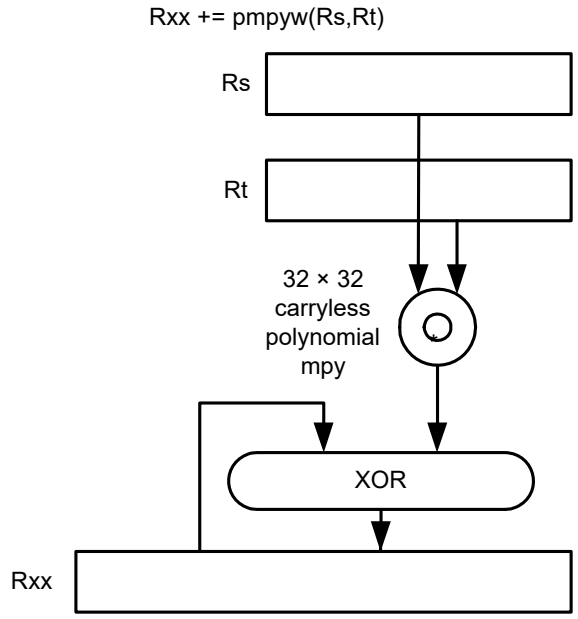
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	0	1	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=mpyu(Rs.L,Rt.L)[:<<N]
1	1	1	0	0	1	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=mpyu(Rs.L,Rt.H)[:<<N]
1	1	1	0	0	1	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=mpyu(Rs.H,Rt.L)[:<<N]
1	1	1	0	0	1	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=mpyu(Rs.H,Rt.H)[:<<N]
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	0	1	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=mpyu(Rs.L,Rt.L)[:<<N]
1	1	1	0	0	1	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=mpyu(Rs.L,Rt.H)[:<<N]
1	1	1	0	0	1	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=mpyu(Rs.H,Rt.L)[:<<N]
1	1	1	0	0	1	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rxx+=mpyu(Rs.H,Rt.H)[:<<N]
1	1	1	0	0	1	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rxx-=mpyu(Rs.L,Rt.L)[:<<N]
1	1	1	0	0	1	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx-=mpyu(Rs.L,Rt.H)[:<<N]
1	1	1	0	0	1	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rxx-=mpyu(Rs.H,Rt.L)[:<<N]
1	1	1	0	0	1	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rxx-=mpyu(Rs.H,Rt.H)[:<<N]
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	1	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=mpyu(Rs.L,Rt.L)[:<<N]
1	1	1	0	1	1	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=mpyu(Rs.L,Rt.H)[:<<N]
1	1	1	0	1	1	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=mpyu(Rs.H,Rt.L)[:<<N]
1	1	1	0	1	1	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rd=mpyu(Rs.H,Rt.H)[:<<N]
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	1	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rx+=mpyu(Rs.L,Rt.L)[:<<N]
1	1	1	0	1	1	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rx+=mpyu(Rs.L,Rt.H)[:<<N]
1	1	1	0	1	1	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rx+=mpyu(Rs.H,Rt.L)[:<<N]
1	1	1	0	1	1	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rx+=mpyu(Rs.H,Rt.H)[:<<N]
1	1	1	0	1	1	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rx-=mpyu(Rs.L,Rt.L)[:<<N]
1	1	1	0	1	1	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rx-=mpyu(Rs.L,Rt.H)[:<<N]
1	1	1	0	1	1	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rx-=mpyu(Rs.H,Rt.L)[:<<N]
1	1	1	0	1	1	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rx-=mpyu(Rs.H,Rt.H)[:<<N]

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
sH	Rs is high
tH	Rt is high
Parse	Packet/loop parse bits

Field name	Description
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Polynomial multiply words

Perform a 32×32 carryless polynomial multiply using 32-bit source registers Rs and Rt. The 64-bit result is optionally accumulated (XOR'd) with the destination register. Finite field multiply instructions are useful for many algorithms including scramble code generation, cryptographic algorithms, convolutional, and Reed Solomon codes.



Syntax	Behavior

`Rdd=pmpyw(Rs,Rt)`	`x = Rs.uw[0];`
	`y = Rt.uw[0];`
	`prod = 0;`
	`for(i=0; i < 32; i++) {`
	`  if((y >> i) & 1) prod ^= (x << i);`
	`}`
	`Rdd = prod;`

`Rxx^=pmpyw(Rs,Rt)`	`x = Rs.uw[0];`
	`y = Rt.uw[0];`
	`prod = 0;`
	`for(i=0; i < 32; i++) {`
	`  if((y >> i) & 1) prod ^= (x << i);`
	`}`
	`Rxx ^= prod;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=pmpyw(Rs,Rt)`	`Word64 Q6_P_pmpyw_RR(Word32 Rs, Word32 Rt)`
`Rxx^=pmpyw(Rs,Rt)`	`Word64 Q6_P_pmpywxacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`

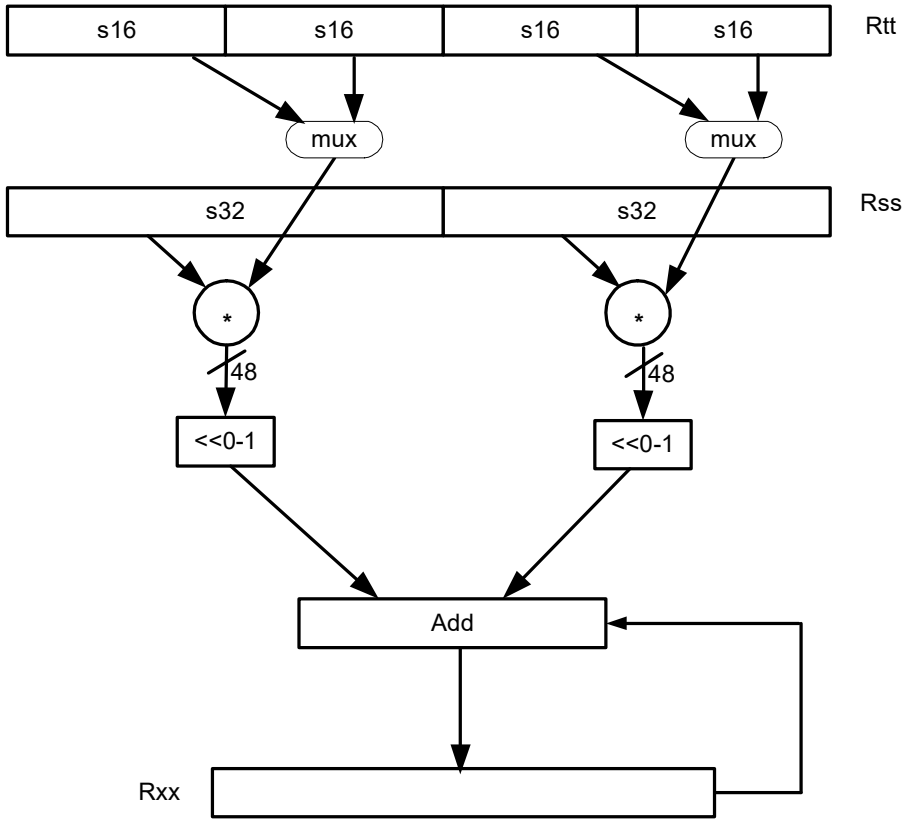
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	0	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=pmpyw(Rs,Rt)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	0	1	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rxx^=pmpyw(Rs,Rt)

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector reduce multiply word by signed half (32×16)

Perform mixed precision vector multiply operations and accumulate the results. A 32-bit word from vector Rss is multiplied by a 16-bit halfword (either even or odd) from vector Rtt. The multiplication is performed as a signed 32×16, which produces a 48-bit result. This result is optionally scaled left by one bit. A similar operation is performed for both words in Rss, and the two results are accumulated. The final result is optionally accumulated with Rxx.



Syntax	Behavior
`Rdd=vrmpyweh(Rss,Rtt)[:<<1]`	`Rdd = (Rss.w[1] * Rtt.h[2])[<<1] + (Rss.w[0] * Rtt.h[0])[<<1];`
`Rdd=vrmpywoh(Rss,Rtt)[:<<1]`	`Rdd = (Rss.w[1] * Rtt.h[3])[<<1] + (Rss.w[0] * Rtt.h[1])[<<1];`
`Rxx+=vrmpyweh(Rss,Rtt)[:<<1]`	`Rxx += (Rss.w[1] * Rtt.h[2])[<<1] + (Rss.w[0] * Rtt.h[0])[<<1];`
`Rxx+=vrmpywoh(Rss,Rtt)[:<<1]`	`Rxx += (Rss.w[1] * Rtt.h[3])[<<1] + (Rss.w[0] * Rtt.h[1])[<<1];`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vrmpyweh(Rss,Rtt)`	`Word64 Q6_P_vrmpyweh_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vrmpyweh(Rss,Rtt):<<1`	`Word64 Q6_P_vrmpyweh_PP_s1(Word64 Rss, Word64 Rtt)`
`Rdd=vrmpywoh(Rss,Rtt)`	`Word64 Q6_P_vrmpywoh_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vrmpywoh(Rss,Rtt):<<1`	`Word64 Q6_P_vrmpywoh_PP_s1(Word64 Rss, Word64 Rtt)`
`Rxx+=vrmpyweh(Rss,Rtt)`	`Word64 Q6_P_vrmpywehacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vrmpyweh(Rss,Rtt):<<1`	`Word64 Q6_P_vrmpywehacc_PP_s1(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vrmpywoh(Rss,Rtt)`	`Word64 Q6_P_vrmpywohacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vrmpywoh(Rss,Rtt):<<1`	`Word64 Q6_P_vrmpywohacc_PP_s1(Word64 Rxx, Word64 Rss, Word64 Rtt)`

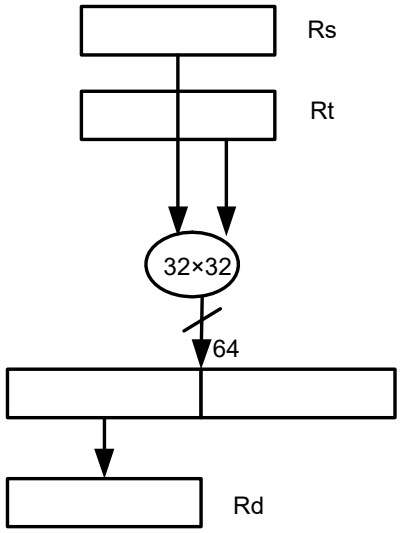
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vrmpywoh(Rss,Rtt)[:<<N]
1	1	1	0	1	0	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vrmpyweh(Rss,Rtt)[:<<N]
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vrmpyweh(Rss,Rtt)[:<<N]
1	1	1	0	1	0	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vrmpywoh(Rss,Rtt)[:<<N]

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Multiply and use upper result

Multiply two signed or unsigned 32-bit words. Take the upper 32-bits of this results store to a single destination register. Optional rounding is available.



Syntax	Behavior
`Rd=mpy(Rs,Rt.H):<<1:rnd:sat`	`Rd = sat32(((Rs * Rt.h[1])<<1+0x8000)>>16);`
`Rd=mpy(Rs,Rt.H):<<1:sat`	`Rd = sat32(((Rs * Rt.h[1])<<1)>>16);`
`Rd=mpy(Rs,Rt.L):<<1:rnd:sat`	`Rd = sat32(((Rs * Rt.h[0])<<1+0x8000)>>16);`
`Rd=mpy(Rs,Rt.L):<<1:sat`	`Rd = sat32(((Rs * Rt.h[0])<<1)>>16);`
`Rd=mpy(Rs,Rt)`	`Rd=(Rs * Rt)>>32;`
`Rd=mpy(Rs,Rt):<<1`	`Rd=(Rs * Rt)>>31;`
`Rd=mpy(Rs,Rt):<<1:sat`	`Rd=sat32((Rs * Rt)>>31);`
`Rd=mpy(Rs,Rt):rnd`	`Rd=((Rs * Rt)+0x80000000)>>32;`
`Rd=mpysu(Rs,Rt)`	`Rd=(Rs * Rt.uw[0])>>32;`
`Rd=mpyu(Rs,Rt)`	`Rd=(Rs.uw[0] * Rt.uw[0])>>32;`
`Rx+=mpy(Rs,Rt):<<1:sat`	`Rx=sat32((Rx) + ((Rs * Rt)>>31));`
`Rx-=mpy(Rs,Rt):<<1:sat`	`Rx=sat32((Rx) - ((Rs * Rt)>>31));`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=mpy(Rs,Rt.H):<<1:rnd:sat`	`Word32 Q6_R_mpy_RRh_s1_rnd_sat(Word32 Rs, Word32 Rt)`
`Rd=mpy(Rs,Rt.H):<<1:sat`	`Word32 Q6_R_mpy_RRh_s1_sat(Word32 Rs, Word32 Rt)`
`Rd=mpy(Rs,Rt.L):<<1:rnd:sat`	`Word32 Q6_R_mpy_RRl_s1_rnd_sat(Word32 Rs, Word32 Rt)`
`Rd=mpy(Rs,Rt.L):<<1:sat`	`Word32 Q6_R_mpy_RRl_s1_sat(Word32 Rs, Word32 Rt)`
`Rd=mpy(Rs,Rt)`	`Word32 Q6_R_mpy_RR(Word32 Rs, Word32 Rt)`
`Rd=mpy(Rs,Rt):<<1`	`Word32 Q6_R_mpy_RR_s1(Word32 Rs, Word32 Rt)`
`Rd=mpy(Rs,Rt):<<1:sat`	`Word32 Q6_R_mpy_RR_s1_sat(Word32 Rs, Word32 Rt)`
`Rd=mpy(Rs,Rt):rnd`	`Word32 Q6_R_mpy_RR_rnd(Word32 Rs, Word32 Rt)`
`Rd=mpysu(Rs,Rt)`	`Word32 Q6_R_mpysu_RR(Word32 Rs, Word32 Rt)`
`Rd=mpyu(Rs,Rt)`	`UWord32 Q6_R_mpyu_RR(Word32 Rs, Word32 Rt)`
`Rx+=mpy(Rs,Rt):<<1:sat`	`Word32 Q6_R_mpyacc_RR_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx-=mpy(Rs,Rt):<<1:sat`	`Word32 Q6_R_mpynac_RR_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)`

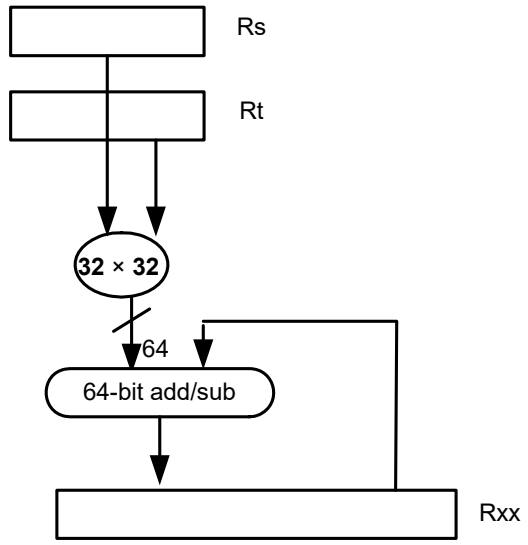
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	1	0	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs,Rt):rnd
1	1	1	0	1	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=mpyu(Rs,Rt)
1	1	1	0	1	1	0	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=mpysu(Rs,Rt)
1	1	1	0	1	1	0	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs,Rt.H):<<1:sat
1	1	1	0	1	1	0	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs,Rt.L):<<1:sat
1	1	1	0	1	1	0	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs,Rt.H):<<1:rnd:sat
1	1	1	0	1	1	0	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs,Rt):<<1:sat
1	1	1	0	1	1	0	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs,Rt.L):<<1:rnd:sat
1	1	1	0	1	1	0	1	**N**	0	**N**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**N**	**N**	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs,Rt)[:<<N]
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	1	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rx+=mpy(Rs,Rt):<<1:sat
1	1	1	0	1	1	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rx-=mpy(Rs,Rt):<<1:sat

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Multiply and use full result

Multiply two signed or unsigned 32-bit words. Optionally, add or subtract this value from the 64-bit accumulator. The result is a full-precision 64-bit value.



Syntax	Behavior
`Rdd=mpy(Rs,Rt)`	`Rdd=(Rs * Rt);`
`Rdd=mpyu(Rs,Rt)`	`Rdd=(Rs.uw[0] * Rt.uw[0]);`
`Rxx[+-]=mpy(Rs,Rt)`	`Rxx= Rxx [+-] (Rs * Rt);`
`Rxx[+-]=mpyu(Rs,Rt)`	`Rxx= Rxx [+-] (Rs.uw[0] * Rt.uw[0]);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=mpy(Rs,Rt)`	`Word64 Q6_P_mpy_RR(Word32 Rs, Word32 Rt)`
`Rdd=mpyu(Rs,Rt)`	`UWord64 Q6_P_mpyu_RR(Word32 Rs, Word32 Rt)`
`Rxx+=mpy(Rs,Rt)`	`Word64 Q6_P_mpyacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=mpyu(Rs,Rt)`	`Word64 Q6_P_mpyuacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpy(Rs,Rt)`	`Word64 Q6_P_mpynac_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpyu(Rs,Rt)`	`Word64 Q6_P_mpyunac_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`

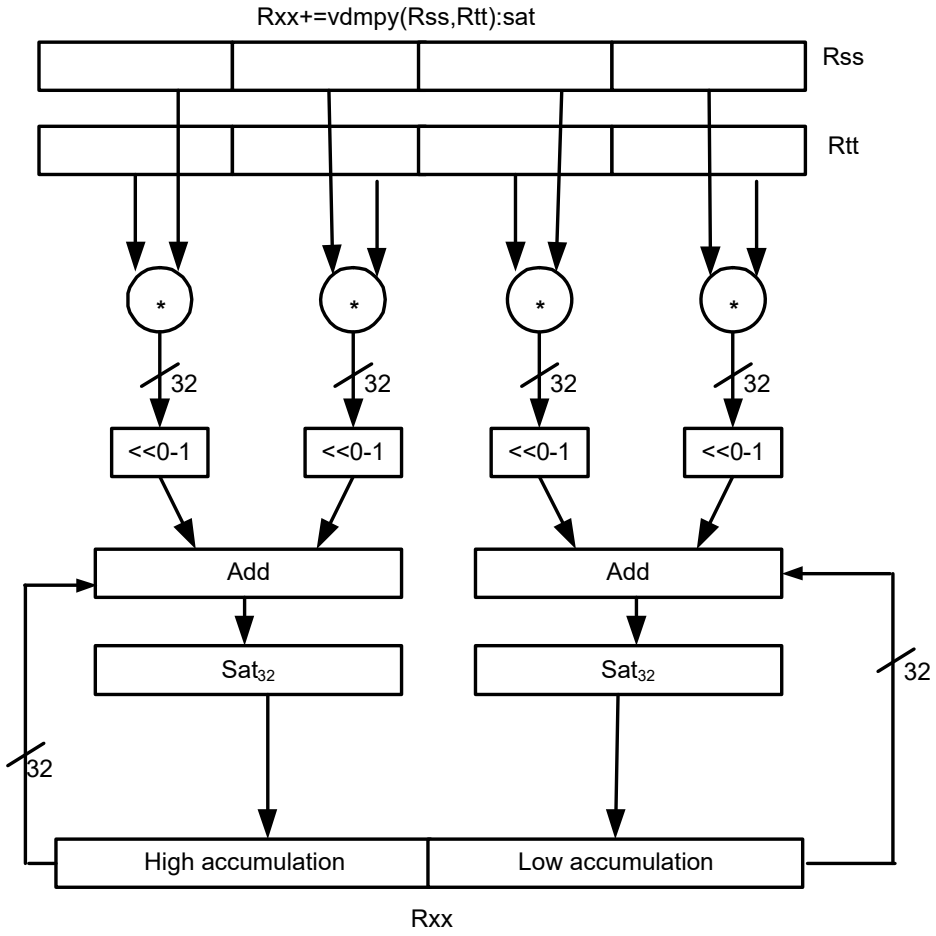
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	0	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=mpy(Rs,Rt)
1	1	1	0	0	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=mpyu(Rs,Rt)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	0	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=mpy(Rs,Rt)
1	1	1	0	0	1	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rxx-=mpy(Rs,Rt)
1	1	1	0	0	1	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=mpyu(Rs,Rt)
1	1	1	0	0	1	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rxx-=mpyu(Rs,Rt)

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector dual multiply

Multiply four 16-bit halfwords in Rss by the corresponding 16-bit halfwords in Rtt. The two lower results are scaled and added. The lower word of the accumulator is optionally added. This result is saturated to 32-bits and stored in the lower word of the accumulator. The same operation is performed on the upper two products using the upper word of the accumulator.



Syntax	Behavior
`Rdd=vdmpy(Rss,Rtt):<<1:sat`	`Rdd.w[0]=sat32((Rss.h[0] * Rtt.h[0])<<1 + (Rss.h[1] * Rtt.h[1])<<1);` `Rdd.w[1]=sat32((Rss.h[2] * Rtt.h[2])<<1 + (Rss.h[3] * Rtt.h[3])<<1);`
`Rdd=vdmpy(Rss,Rtt):sat`	`Rdd.w[0]=sat32((Rss.h[0] * Rtt.h[0])<<0 + (Rss.h[1] * Rtt.h[1])<<0);` `Rdd.w[1]=sat32(Rss.h[2] * Rtt.h[2])<<0 + (Rss.h[3] * Rtt.h[3])<<0);`
`Rxx+=vdmpy(Rss,Rtt):<<1:sat`	`Rxx.w[0]=sat32(Rxx.w[0] + (Rss.h[0] * Rtt.h[0])<<1 + (Rss.h[1] * Rtt.h[1])<<1);` `Rxx.w[1]=sat32(Rxx.w[1] + (Rss.h[2] * Rtt.h[2])<<1 + (Rss.h[3] * Rtt.h[3])<<1);`

Syntax	Behavior
`Rxx+=vdmpy(Rss,Rtt):sat`	`Rxx.w[0]=sat32(Rxx.w[0] + (Rss.h[0] * Rtt.h[0])<<0 + (Rss.h[1] * Rtt.h[1])<<0); Rxx.w[1]=sat32(Rxx.w[1] + (Rss.h[2] * Rtt.h[2])<<0 + (Rss.h[3] * Rtt.h[3])<<0);`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vdmpy(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vdmpy_PP_s1_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vdmpy(Rss,Rtt):sat`	`Word64 Q6_P_vdmpy_PP_sat(Word64 Rss, Word64 Rtt)`
`Rxx+=vdmpy(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vdmpyacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vdmpy(Rss,Rtt):sat`	`Word64 Q6_P_vdmpyacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`

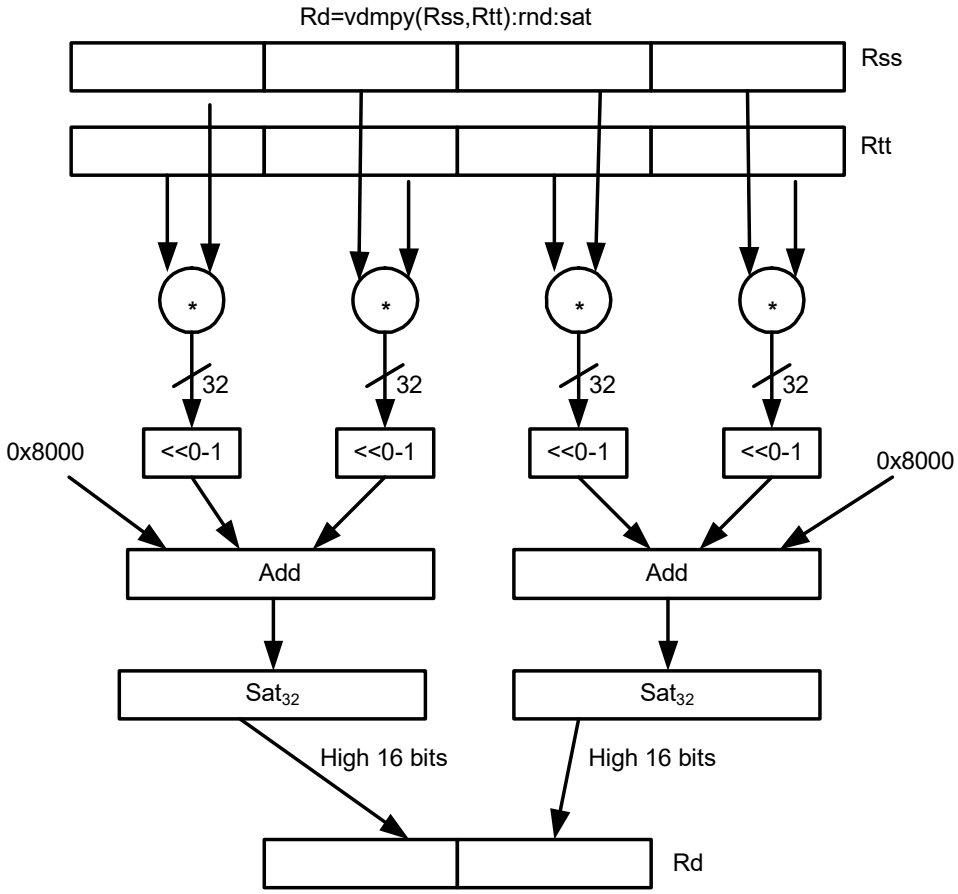
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vdmpy(Rss,Rtt)[:<<N]:sat
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vdmpy(Rss,Rtt)[:<<N]:sat

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector dual multiply with round and pack

Multiply four 16-bit halfwords in Rss by the corresponding 16-bit halfwords in Rtt. The two lower results are scaled and added together with a rounding constant. This result is saturated to 32-bits, and the upper 16-bits of this result are stored in the lower 16-bits of the destination register. The same operation is performed on the upper two products and the result is stored in the upper 16-bit halfword of the destination.



Syntax	Behavior
`Rd=vdmpy(Rss,Rtt)[:<<1]:rnd:sat`	`Rd.h[0]=(sat32((Rss.h[0] * Rtt.h[0])[<<1] + (Rss.h[1] * Rtt.h[1])[<<1] + 0x8000)).h[1];` `Rd.h[1]=(sat32((Rss.h[2] * Rtt.h[2])[<<1] + (Rss.h[3] * Rtt.h[3])[<<1] + 0x8000)).h[1];`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=vdmpy(Rss,Rtt):<< 1:rnd:sat`	`Word32 Q6_R_vdmpy_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt)`
`Rd=vdmpy(Rss,Rtt):rnd:sat`	`Word32 Q6_R_vdmpy_PP_rnd_sat(Word64 Rss, Word64 Rtt)`

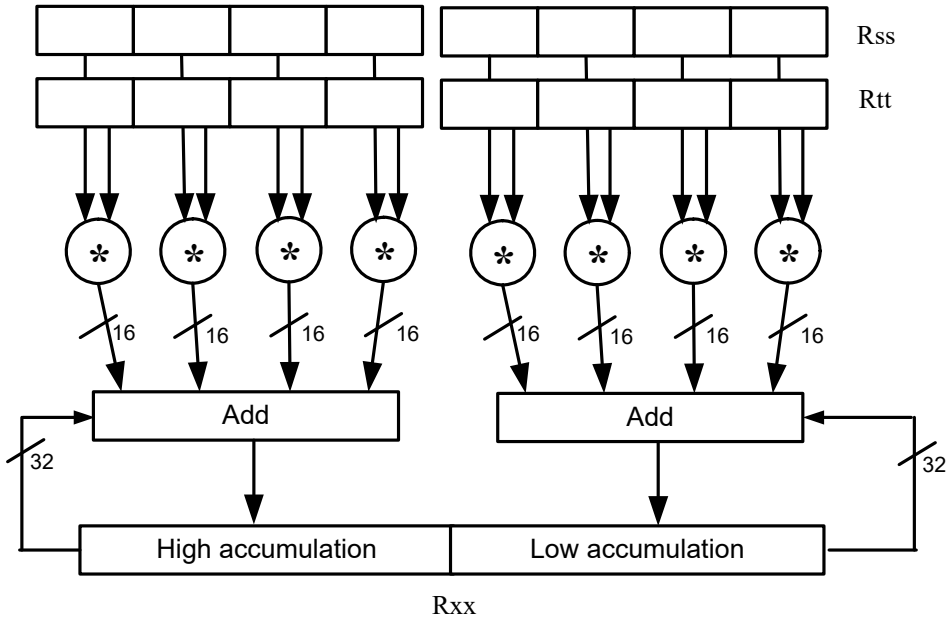
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	1	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=vdmpy(Rss,Rtt)[:<<N]:rnd:sat

Field name	Description
`ICLASS`	Instruction class
`MajOp`	Major opcode
`MinOp`	Minor opcode
`RegType`	Register type
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector reduce multiply bytes

Multiply eight 8-bit bytes in Rss by the corresponding 8-bit bytes in Rtt. The four lower results are accumulated. The lower word of the accumulator is optionally added. This result is stored in the lower 32-bits of the accumulator. The same operation is performed on the upper four products using the upper word of the accumulator. The eight bytes of Rss can be treated as either signed or unsigned.



Syntax	Behavior
`Rdd = vrmpybsu(Rss,Rtt)`	`Rdd.w[0]=((Rss.b[0] * Rtt.ub[0]) + (Rss.b[1] * Rtt.ub[1]) + (Rss.b[2] * Rtt.ub[2]) + (Rss.b[3] * Rtt.ub[3]));` `Rdd.w[1]=((Rss.b[4] * Rtt.ub[4]) + (Rss.b[5] * Rtt.ub[5]) + (Rss.b[6] * Rtt.ub[6]) + (Rss.b[7] * Rtt.ub[7]));`
`Rdd = vrmpybu(Rss,Rtt)`	`Rdd.w[0]=((Rss.ub[0] * Rtt.ub[0]) + (Rss.ub[1] * Rtt.ub[1]) + (Rss.ub[2] * Rtt.ub[2]) + (Rss.ub[3] * Rtt.ub[3]));` `Rdd.w[1]=((Rss.ub[4] * Rtt.ub[4]) + (Rss.ub[5] * Rtt.ub[5]) + (Rss.ub[6] * Rtt.ub[6]) + (Rss.ub[7] * Rtt.ub[7]));`
`Rxx += vrmpybsu(Rss,Rtt)`	`Rxx.w[0]=(Rxx.w[0] + (Rss.b[0] * Rtt.ub[0]) + (Rss.b[1] * Rtt.ub[1]) + (Rss.b[2] * Rtt.ub[2]) + (Rss.b[3] * Rtt.ub[3]));` `Rxx.w[1]=(Rxx.w[1] + (Rss.b[4] * Rtt.ub[4]) + (Rss.b[5] * Rtt.ub[5]) + (Rss.b[6] * Rtt.ub[6]) + (Rss.b[7] * Rtt.ub[7]));`

Syntax	Behavior
`Rxx+=vrmpybu(Rss,Rtt)`	`Rxx.w[0]=(Rxx.w[0] + (Rss.ub[0] * Rtt.ub[0]) + (Rss.ub[1] * Rtt.ub[1]) + (Rss.ub[2] * Rtt.ub[2]) + (Rss.ub[3] * Rtt.ub[3]));` `Rxx.w[1]=(Rxx.w[1] + (Rss.ub[4] * Rtt.ub[4]) + (Rss.ub[5] * Rtt.ub[5]) + (Rss.ub[6] * Rtt.ub[6]) + (Rss.ub[7] * Rtt.ub[7]));`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vrmpybsu(Rss,Rtt)`	`Word64 Q6_P_vrmpybsu_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vrmpybu(Rss,Rtt)`	`Word64 Q6_P_vrmpybu_PP(Word64 Rss, Word64 Rtt)`
`Rxx+=vrmpybsu(Rss,Rtt)`	`Word64 Q6_P_vrmpybsuacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vrmpybu(Rss,Rtt)`	`Word64 Q6_P_vrmpybuacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`

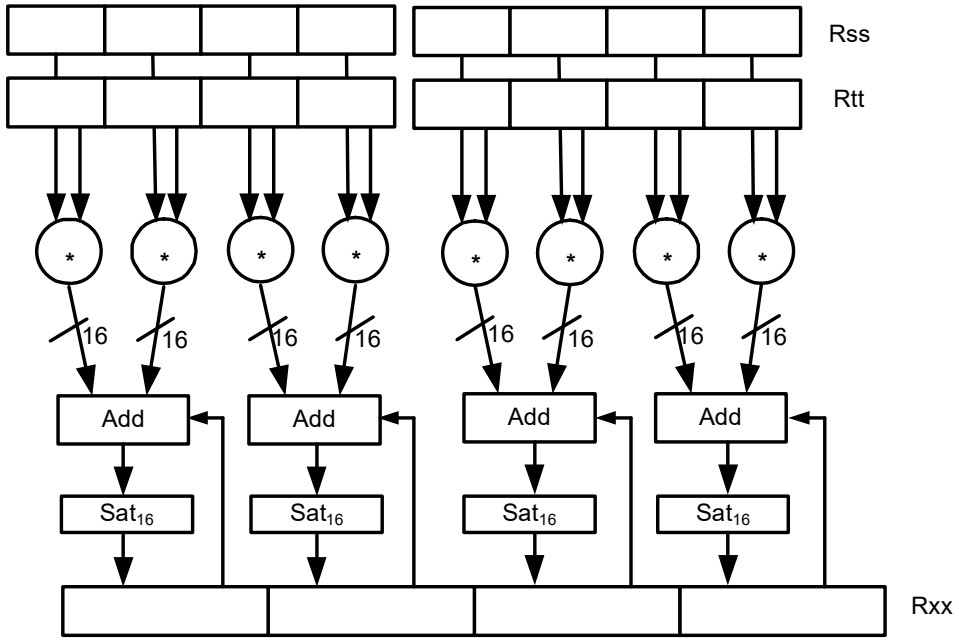
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vrmpybu(Rss,Rtt)
1	1	1	0	1	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vrmpybsu(Rss,Rtt)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vrmpybu(Rss,Rtt)
1	1	1	0	1	0	1	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vrmpybsu(Rss,Rtt)

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector dual multiply signed by unsigned bytes

Multiply eight 8-bit signed bytes in Rss by the corresponding 8-bit unsigned bytes in Rtt. Add the results in pairs, and optionally add the accumulator. The results are saturated to signed 16-bits and stored in the four halfwords of the destination register.



Syntax	Behavior

`Rdd=vdmpybsu(Rss,Rtt):sat`	`Rdd.h[0]=sat16(((Rss.b[0] * Rtt.ub[0]) + (Rss.b[1] * Rtt.ub[1])));`
	`Rdd.h[1]=sat16(((Rss.b[2] * Rtt.ub[2]) + (Rss.b[3] * Rtt.ub[3])));`
	`Rdd.h[2]=sat16(((Rss.b[4] * Rtt.ub[4]) + (Rss.b[5] * Rtt.ub[5])));`
	`Rdd.h[3]=sat16(((Rss.b[6] * Rtt.ub[6]) + (Rss.b[7] * Rtt.ub[7])));`

`Rxx+=vdmpybsu(Rss,Rtt):sat`	`Rxx.h[0]=sat16((Rxx.h[0] + (Rss.b[0] * Rtt.ub[0]) + (Rss.b[1] * Rtt.ub[1])));`
	`Rxx.h[1]=sat16((Rxx.h[1] + (Rss.b[2] * Rtt.ub[2]) + (Rss.b[3] * Rtt.ub[3])));`
	`Rxx.h[2]=sat16((Rxx.h[2] + (Rss.b[4] * Rtt.ub[4]) + (Rss.b[5] * Rtt.ub[5])));`
	`Rxx.h[3]=sat16((Rxx.h[3] + (Rss.b[6] * Rtt.ub[6]) + (Rss.b[7] * Rtt.ub[7])));`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vdmpybsu(Rss,Rtt):sat`	`Word64 Q6_P_vdmpybsu_PP_sat(Word64 Rss, Word64 Rtt)`
`Rxx+=vdmpybsu(Rss,Rtt):sat`	`Word64 Q6_P_vdmpybsuacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`

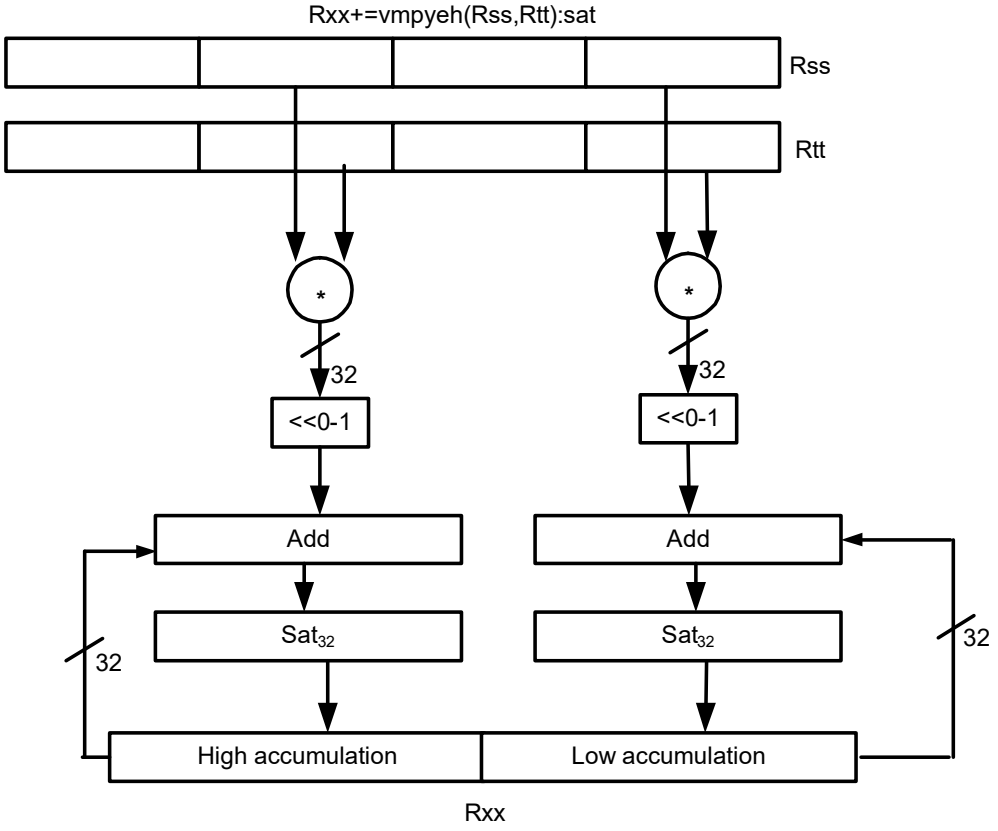
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vdmpybsu(Rss,Rtt):sat
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vdmpybsu(Rss,Rtt):sat

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector multiply even halfwords

Multiply the even 16-bit halfwords from Rss and Rtt separately. Optionally accumulate with the low and high words of the destination register pair and optionally saturate.



Syntax	Behavior
`Rdd=vmpyeh(Rss,Rtt):<<1:sat`	`Rdd.w[0]=sat32((Rss.h[0] * Rtt.h[0])<<1);` `Rdd.w[1]=sat32((Rss.h[2] * Rtt.h[2])<<1);`
`Rdd=vmpyeh(Rss,Rtt):sat`	`Rdd.w[0]=sat32((Rss.h[0] * Rtt.h[0])<<0);` `Rdd.w[1]=sat32((Rss.h[2] * Rtt.h[2])<<0);`
`Rxx+=vmpyeh(Rss,Rtt)`	`Rxx.w[0]=Rxx.w[0] + (Rss.h[0] * Rtt.h[0]);` `Rxx.w[1]=Rxx.w[1] + (Rss.h[2] * Rtt.h[2]);`
`Rxx+=vmpyeh(Rss,Rtt):<<1:sat`	`Rxx.w[0]=sat32(Rxx.w[0] + (Rss.h[0] * Rtt.h[0])<<1);` `Rxx.w[1]=sat32(Rxx.w[1] + (Rss.h[2] * Rtt.h[2])<<1);`
`Rxx+=vmpyeh(Rss,Rtt):sat`	`Rxx.w[0]=sat32(Rxx.w[0] + (Rss.h[0] * Rtt.h[0])<<0);` `Rxx.w[1]=sat32(Rxx.w[1] + (Rss.h[2] * Rtt.h[2])<<0);`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vmpyeh(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vmpyeh_PP_s1_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vmpyeh(Rss,Rtt):sat`	`Word64 Q6_P_vmpyeh_PP_sat(Word64 Rss, Word64 Rtt)`
`Rxx+=vmpyeh(Rss,Rtt)`	`Word64 Q6_P_vmpyehacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vmpyeh(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vmpyehacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vmpyeh(Rss,Rtt):sat`	`Word64 Q6_P_vmpyehacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`

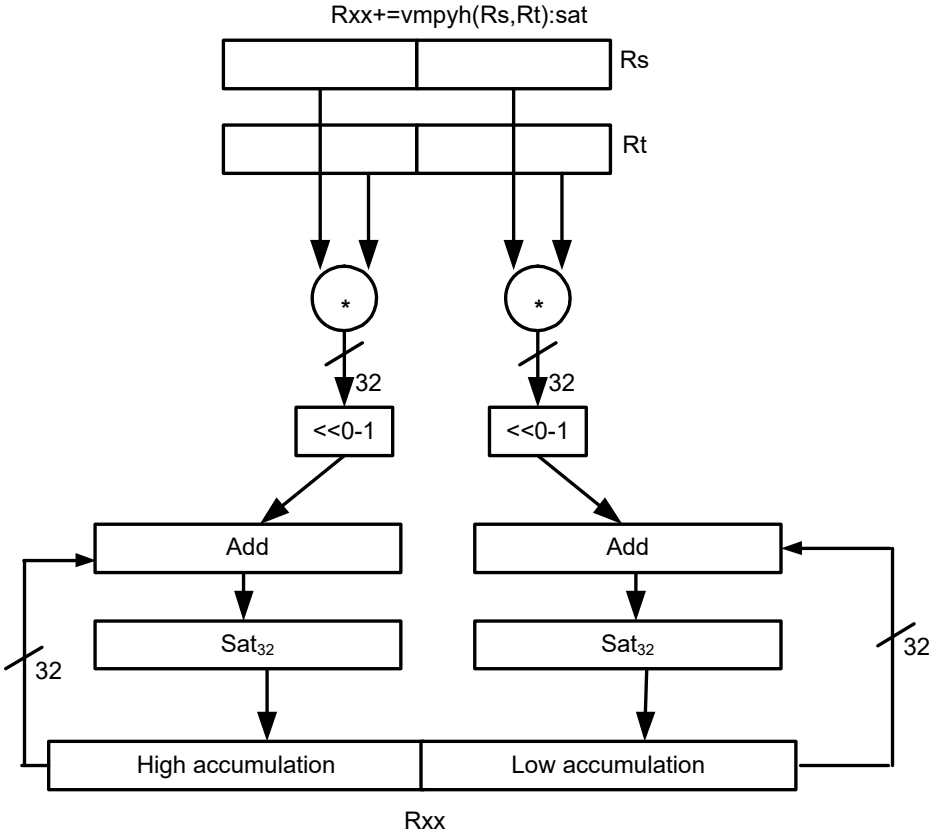
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vmpyeh(Rss,Rtt)[:<<N]:sat
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpyeh(Rss,Rtt)
1	1	1	0	1	0	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpyeh(Rss,Rtt)[:<<N]:sat

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector multiply halfwords

Multiply two 16-bit halfwords separately, and optionally accumulate with the low and high words of the destination. Optionally saturate, and store the results back to the destination register pair.



Syntax	Behavior
`Rdd=vmpyh(Rs,Rt)[:<<1]:sat`	`Rdd.w[0]=sat32((Rs.h[0] * Rt.h[0])[<<1]);` `Rdd.w[1]=sat32((Rs.h[1] * Rt.h[1])[<<1]);`
`Rxx+=vmpyh(Rs,Rt)`	`Rxx.w[0]=Rxx.w[0] + (Rs.h[0] * Rt.h[0]);` `Rxx.w[1]=Rxx.w[1] + (Rs.h[1] * Rt.h[1]);`
`Rxx+=vmpyh(Rs,Rt)[:<<1]:sat`	`Rxx.w[0]=sat32(Rxx.w[0] + (Rs.h[0] * Rt.h[0])[<<1]);` `Rxx.w[1]=sat32(Rxx.w[1] + (Rs.h[1] * Rt.h[1])[<<1]);`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vmpyh(Rs,Rt):<<1:sat`	`Word64 Q6_P_vmpyh_RR_s1_sat(Word32 Rs, Word32 Rt)`
`Rdd=vmpyh(Rs,Rt):sat`	`Word64 Q6_P_vmpyh_RR_sat(Word32 Rs, Word32 Rt)`
`Rxx+=vmpyh(Rs,Rt)`	`Word64 Q6_P_vmpyhacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=vmpyh(Rs,Rt):<<1:sat`	`Word64 Q6_P_vmpyhacc_RR_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=vmpyh(Rs,Rt):sat`	`Word64 Q6_P_vmpyhacc_RR_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`

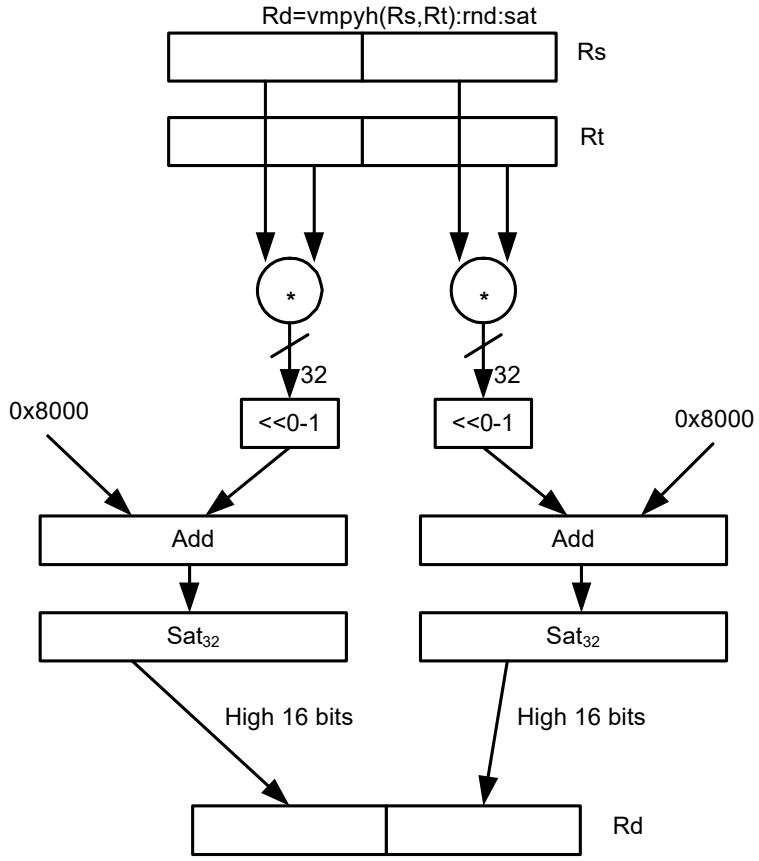
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	0	1	0	1	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpyh(Rs,Rt)[:<<N]:sat
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	0	1	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpyh(Rs,Rt)
1	1	1	0	0	1	1	1	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpyh(Rs,Rt)[:<<N]:sat

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector multiply halfwords with round and pack

Multiply two 16-bit halfwords separately. Round the results, and store the high halfwords packed in a single register destination.



Syntax	Behavior
`Rd=vmpyh(Rs,Rt)[:<<1]:rnd:sat`	`Rd.h[1]=(sat32((Rs.h[1] * Rt.h[1])[<<1] + 0x8000)).h[1];` `Rd.h[0]=(sat32((Rs.h[0] * Rt.h[0])[<<1] + 0x8000)).h[1];`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=vmpyh(Rs,Rt):<<1:rnd:sat`	`Word32 Q6_R_vmpyh_RR_s1_rnd_sat(Word32 Rs, Word32 Rt)`
`Rd=vmpyh(Rs,Rt):rnd:sat`	`Word32 Q6_R_vmpyh_RR_rnd_sat(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	1	0	1	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=vmpyh(Rs,Rt)[:<<N]:rnd:sat

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector multiply halfwords, signed by unsigned

Multiply two 16-bit halfwords. Rs is considered signed, Ru unsigned.

Syntax	Behavior
`Rdd = vmpyhsu(Rs, Rt)[:<<1]:sat`	`Rdd.w[0]=sat32((Rs.h[0] * Rt.uh[0])[<<1]);` `Rdd.w[1]=sat32((Rs.h[1] * Rt.uh[1])[<<1]);`
`Rxx += vmpyhsu(Rs, Rt)[:<<1]:sat`	`Rxx.w[0]=sat32(Rxx.w[0]+ (Rs.h[0] * Rt.uh[0])[<< 1]);` `Rxx.w[1]=sat32(Rxx.w[1]+ (Rs.h[1] * Rt.uh[1])[<< 1]);`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vmpyhsu(Rs,Rt):<<1:sat`	`Word64 Q6_P_vmpyhsu_RR_s1_sat(Word32 Rs, Word32 Rt)`
`Rdd=vmpyhsu(Rs,Rt):sat`	`Word64 Q6_P_vmpyhsu_RR_sat(Word32 Rs, Word32 Rt)`
`Rxx+=vmpyhsu(Rs,Rt):<<1:sat`	`Word64 Q6_P_vmpyhsuacc_RR_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=vmpyhsu(Rs,Rt):sat`	`Word64 Q6_P_vmpyhsuacc_RR_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`

Encoding

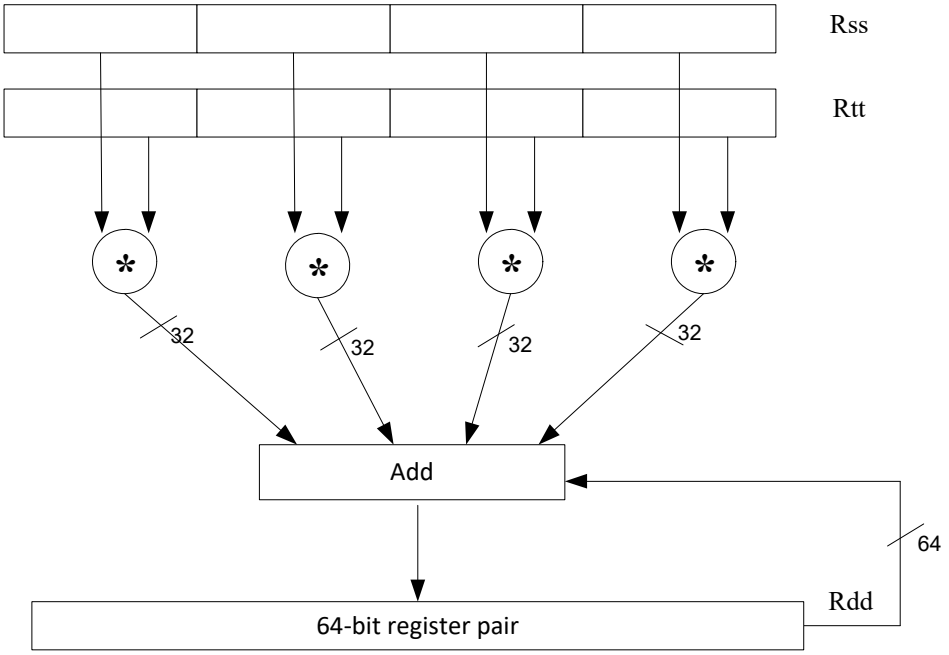
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5						MinOp			d5					
1	1	1	0	0	1	0	1	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpyhsu(Rs,Rt)[:<<N]:sat
ICLASS				RegType				MajOp			s5					Parse			t5						MinOp			x5					
1	1	1	0	0	1	1	1	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpyhsu(Rs,Rt)[:<<N]:sat

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector reduce multiply halfwords

Multiply each halfword of Rss by the corresponding halfword in Rtt. Add the intermediate products together and optionally add the accumulator. Store the full 64-bit result in the destination register pair.

This instruction is known as "big mac".



Syntax	Behavior
`Rdd=vrmpyh(Rss,Rtt)`	`Rdd = (Rss.h[0] * Rtt.h[0]) + (Rss.h[1] * Rtt.h[1]) + (Rss.h[2] * Rtt.h[2]) + (Rss.h[3] * Rtt.h[3]);`
`Rxx+=vrmpyh(Rss,Rtt)`	`Rxx = Rxx + (Rss.h[0] * Rtt.h[0]) + (Rss.h[1] * Rtt.h[1]) + (Rss.h[2] * Rtt.h[2]) + (Rss.h[3] * Rtt.h[3]);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vrmpyh(Rss,Rtt)`	`Word64 Q6_P_vrmpyh_PP(Word64 Rss, Word64 Rtt)`
`Rxx+=vrmpyh(Rss,Rtt)`	`Word64 Q6_P_vrmpyhacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`

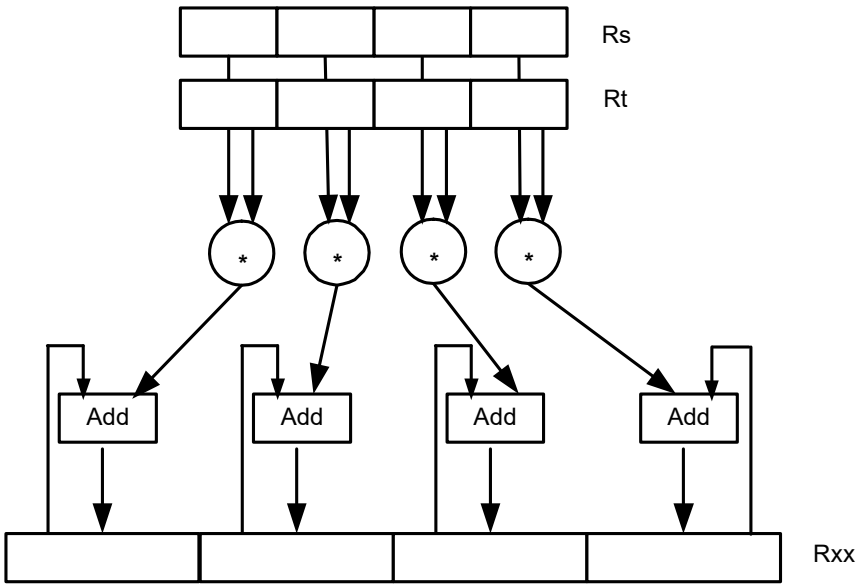
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vrmpyh(Rss,Rtt)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vrmpyh(Rss,Rtt)

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector multiply bytes

Four 8-bit bytes from register Rs are multiplied by four 8-bit bytes from Rt. The product is optionally accumulated with the 16-bit value from the destination register. The 16-bit results are packed in the destination register pair. The bytes of Rs can be treated as either signed or unsigned.



Syntax	Behavior

`Rdd=vmpybsu(Rs,Rt)`	`Rdd.h[0]=((Rs.b[0] * Rt.ub[0]));`
	`Rdd.h[1]=((Rs.b[1] * Rt.ub[1]));`
	`Rdd.h[2]=((Rs.b[2] * Rt.ub[2]));`
	`Rdd.h[3]=((Rs.b[3] * Rt.ub[3]));`

`Rdd=vmpybu(Rs,Rt)`	`Rdd.h[0]=((Rs.ub[0] * Rt.ub[0]));`
	`Rdd.h[1]=((Rs.ub[1] * Rt.ub[1]));`
	`Rdd.h[2]=((Rs.ub[2] * Rt.ub[2]));`
	`Rdd.h[3]=((Rs.ub[3] * Rt.ub[3]));`

`Rxx+=vmpybsu(Rs,Rt)`	`Rxx.h[0]=(Rxx.h[0]+(Rs.b[0] * Rt.ub[0]));`
	`Rxx.h[1]=(Rxx.h[1]+(Rs.b[1] * Rt.ub[1]));`
	`Rxx.h[2]=(Rxx.h[2]+(Rs.b[2] * Rt.ub[2]));`
	`Rxx.h[3]=(Rxx.h[3]+(Rs.b[3] * Rt.ub[3]));`

`Rxx+=vmpybu(Rs,Rt)`	`Rxx.h[0]=(Rxx.h[0]+(Rs.ub[0] * Rt.ub[0]));`
	`Rxx.h[1]=(Rxx.h[1]+(Rs.ub[1] * Rt.ub[1]));`
	`Rxx.h[2]=(Rxx.h[2]+(Rs.ub[2] * Rt.ub[2]));`
	`Rxx.h[3]=(Rxx.h[3]+(Rs.ub[3] * Rt.ub[3]));`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vmpybsu(Rs,Rt)`	`Word64 Q6_P_vmpybsu_RR(Word32 Rs, Word32 Rt)`
`Rdd=vmpybu(Rs,Rt)`	`Word64 Q6_P_vmpybu_RR(Word32 Rs, Word32 Rt)`

`Rxx+=vmpybsu(Rs,Rt)`	`Word64 Q6_P_vmpybsuacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=vmpybu(Rs,Rt)`	`Word64 Q6_P_vmpybuacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`

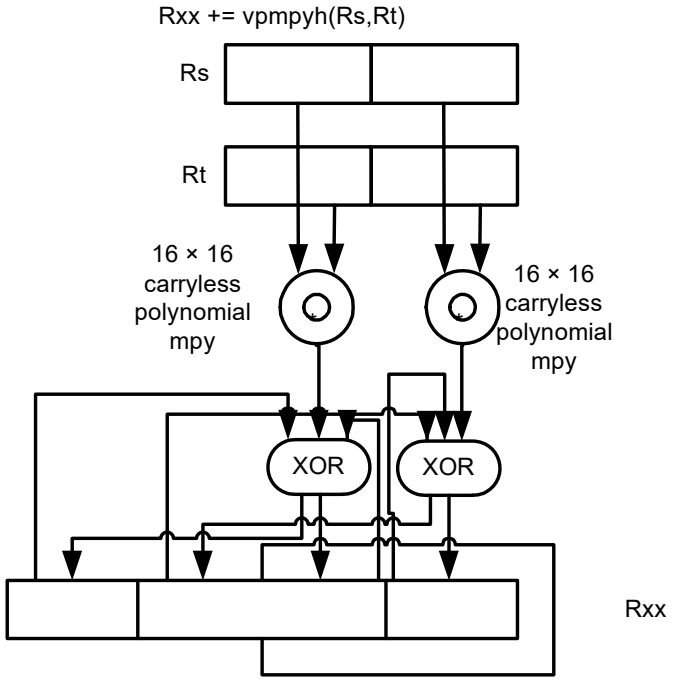
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	0	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpybsu(Rs,Rt)
1	1	1	0	0	1	0	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpybu(Rs,Rt)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	0	1	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpybu(Rs,Rt)
1	1	1	0	0	1	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpybsu(Rs,Rt)

Field name	Description
`ICLASS`	Instruction class
`MajOp`	Major opcode
`MinOp`	Minor opcode
`RegType`	Register type
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`x5`	Field to encode register x

Vector polynomial multiply halfwords

Perform a vector 16×16 carryless polynomial multiply using 32-bit source registers Rs and Rt. The 64-bit result is stored in packed H,H,L,L format in the destination register. The destination register can also be optionally accumulated (XOR'd). Finite field multiply instructions are useful for many algorithms including scramble code generation, cryptographic algorithms, convolutional, and Reed Solomon codes.



Syntax	Behavior

`Rdd=vpmpyh(Rs,Rt)`	`x0 = Rs.uh[0];`
	`x1 = Rs.uh[1];`
	`y0 = Rt.uh[0];`
	`y1 = Rt.uh[1];`
	`prod0 = prod1 = 0;`
	`for(i=0; i < 16; i++) {`
	`  if((y0 >> i) & 1) prod0 ^= (x0 << i);`
	`  if((y1 >> i) & 1) prod1 ^= (x1 << i);`
	`}`
	`Rdd.h[0]=prod0.uh[0];`
	`Rdd.h[1]=prod1.uh[0];`
	`Rdd.h[2]=prod0.uh[1];`
	`Rdd.h[3]=prod1.uh[1];`

Syntax	Behavior

`Rxx^=vpmpyh(Rs,Rt)`	`x0 = Rs.uh[0];`
	`x1 = Rs.uh[1];`
	`y0 = Rt.uh[0];`
	`y1 = Rt.uh[1];`
	`prod0 = prod1 = 0;`
	`for(i=0; i < 16; i++) {`
	`  if((y0 >> i) & 1) prod0 ^= (x0 << i);`
	`  if((y1 >> i) & 1) prod1 ^= (x1 << i);`
	`}`
	`Rxx.h[0]=Rxx.uh[0] ^ prod0.uh[0];`
	`Rxx.h[1]=Rxx.uh[1] ^ prod1.uh[0];`
	`Rxx.h[2]=Rxx.uh[2] ^ prod0.uh[1];`
	`Rxx.h[3]=Rxx.uh[3] ^ prod1.uh[1];`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vpmpyh(Rs,Rt)`	`Word64 Q6_P_vpmpyh_RR(Word32 Rs, Word32 Rt)`
`Rxx^=vpmpyh(Rs,Rt)`	`Word64 Q6_P_vpmpyhxacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5						MinOp			d5					
1	1	1	0	0	1	0	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vpmpyh(Rs,Rt)
ICLASS				RegType				MajOp			s5					Parse			t5						MinOp			x5					
1	1	1	0	0	1	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rxx^=vpmpyh(Rs,Rt)

Field name	Description
ICLASS	Instruction class
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

11.10.6 XTYPE perm

The XTYPE perm instruction subclass includes instructions that perform permutations.

CABAC decode bin

This is a special-purpose instruction to support H.264 Context Adaptive Binary Arithmetic Coding (CABAC). See Section 4.8.1 for a complete description.

Syntax	Behavior

`Rdd=decbin(Rss,Rtt)`	`state = Rtt.w[1][5:0];`
	`valMPS = Rtt.w[1][8:8];`
	`bitpos = Rtt.w[0][4:0];`
	`range = Rss.w[0];`
	`offset = Rss.w[1];`
	`range <<= bitpos;`
	`offset <<= bitpos;`
	`rLPS = rLPS_table_64x4[state][ (range >>29)&3];`
	`rLPS = rLPS << 23;`
	`rMPS= (range&0xff800000) - rLPS;`
	`if (offset < rMPS) {`
	`  Rdd = AC_next_state_MPS_64[state];`
	`  Rdd[8:8]=valMPS;`
	`  Rdd[31:23]=(rMPS>>23);`
	`  Rdd.w[1]=offset;`
	`  P0=valMPS;`
	`} else {`
	`  Rdd = AC_next_state_LPS_64[state];`
	`  Rdd[8:8]=((!state)?(1-valMPS):(valMPS));`
	`  Rdd[31:23]=(rLPS>>23);`
	`  Rdd.w[1]=(offset-rMPS);`
	`  P0=(valMPS^1);`
	`}`

Class: XTYPE (slots 2,3)

Notes

- The predicate generated by this instruction cannot be used as a .new predicate, nor can it be automatically AND'd with another predicate.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	0	1	1	1	-	s	s	s	s	s	P	P	-	t	t	t	t	t	1	1	-	d	d	d	d	d	Rdd=decbin(Rss,Rtt)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
Maj	Major opcode
Min	Minor opcode
RegType	Register type

Saturate

Saturate a single scalar value.

The `sath` instruction saturates a signed 32-bit number to a signed 16-bit number, which is sign-extended back to 32 bits and placed in the destination register. The minimum negative value of the result is 0xffff8000 and the maximum positive value is 0x00007fff.

The `satuh` instruction saturates a signed 32-bit number to an unsigned 16-bit number, which is zero-extended back to 32 bits and placed in the destination register. The minimum value of the result is 0 and the maximum value is 0x0000ffff.

The `satb` instruction saturates a signed 32-bit number to an signed 8-bit number, which is sign-extended back to 32 bits and placed in the destination register. The minimum value of the result is 0xffffff80 and the maximum value is 0x0000007f.

The `satub` instruction saturates a signed 32-bit number to an unsigned 8-bit number, which is zero-extended back to 32 bits and placed in the destination register. The minimum value of the result is 0 and the maximum value is 0x000000ff.

Syntax	Behavior
`Rd=sat(Rss)`	`Rd = sat32(Rss);`
`Rd=satb(Rs)`	`Rd = sat8(Rs);`
`Rd=sath(Rs)`	`Rd = sat16(Rs);`
`Rd=satub(Rs)`	`Rd = usat8(Rs);`
`Rd=satuh(Rs)`	`Rd = usat16(Rs);`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=sat(Rss)`	`Word32 Q6_R_sat_P(Word64 Rss)`
`Rd=satb(Rs)`	`Word32 Q6_R_satb_R(Word32 Rs)`
`Rd=sath(Rs)`	`Word32 Q6_R_sath_R(Word32 Rs)`
`Rd=satub(Rs)`	`Word32 Q6_R_satub_R(Word32 Rs)`
`Rd=satuh(Rs)`	`Word32 Q6_R_satuh_R(Word32 Rs)`

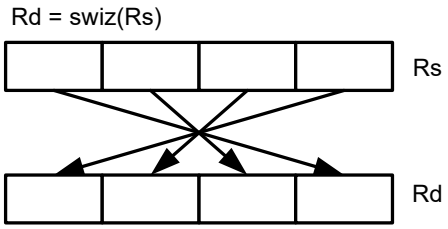
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**-**	**-**	**-**	**-**	**-**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=sat(Rss)
1	0	0	0	1	1	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**-**	**-**	**-**	**-**	**-**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=sath(Rs)
1	0	0	0	1	1	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**-**	**-**	**-**	**-**	**-**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rd=satuh(Rs)
1	0	0	0	1	1	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**-**	**-**	**-**	**-**	**-**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=satub(Rs)
1	0	0	0	1	1	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**-**	**-**	**-**	**-**	**-**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=satb(Rs)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

Swizzle bytes

Swizzle the bytes of a word. This instruction is useful in converting between little and big endian formats.



Syntax	Behavior

`Rd=swiz(Rs)`	`Rd.b[0]=Rs.b[3];`
	`Rd.b[1]=Rs.b[2];`
	`Rd.b[2]=Rs.b[1];`
	`Rd.b[3]=Rs.b[0];`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=swiz(Rs)`	`Word32 Q6_R_swiz_R(Word32 Rs)`

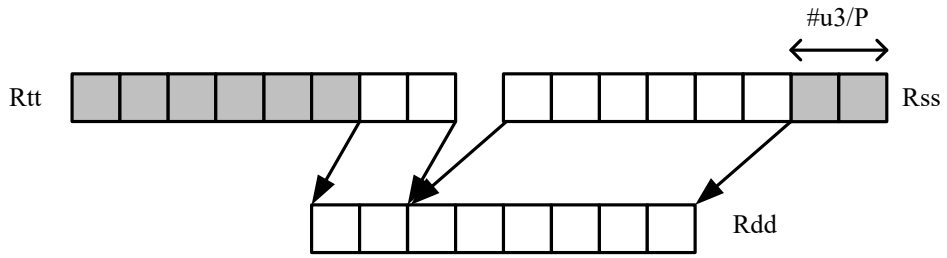
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5						
1	0	0	0	1	1	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=swiz(Rs)

Field name	Description
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major opcode
`MinOp`	Minor opcode
`RegType`	Register type

Vector align

Align a vector. Use the immediate amount, or the least significant 3 bits of a predicate register, as the number of bytes to align. Shift the Rss register pair right by this number of bytes. Fill the vacated positions with the least significant elements from Rtt.



Syntax	Behavior
`Rdd=valignb(Rtt,Rss,#u3)`	`Rdd = (Rss >>> #u*8)|(Rtt << ((8-#u)*8));`
`Rdd=valignb(Rtt,Rss,Pu)`	`PREDUSE_TIMING;` `Rdd = Rss >>> (Pu&0x7)*8|(Rtt << (8-(Pu&0x7))*8);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=valignb(Rtt,Rss,#u3)`	`Word64 Q6_P_valignb_PPI(Word64 Rtt, Word64 Rss, Word32 Iu3)`
`Rdd=valignb(Rtt,Rss,Pu)`	`Word64 Q6_P_valignb_PPp(Word64 Rtt, Word64 Rss, Byte Pu)`

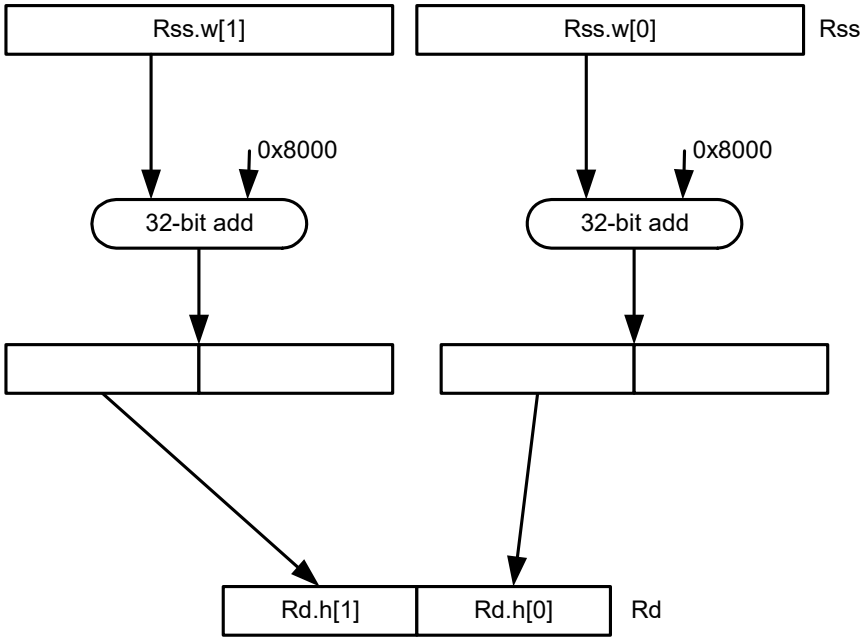
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	0	0	0	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	i	i	i	d	d	d	d	d	Rdd=valignb(Rtt,Rss,#u3)
ICLASS				RegType				Maj			s5					Parse			t5						u2		d5					
1	1	0	0	0	0	1	0	0	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	-	u	u	d	d	d	d	d	Rdd=valignb(Rtt,Rss,Pu)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
u2	Field to encode register u
Maj	Major opcode
Min	Minor opcode
RegType	Register type

Vector round and pack

Add the constant 0x00008000 to each word in the 64-bit source vector Rss. Optionally saturate this addition to 32 bits. Pack the high halfwords of the result into the corresponding halfword of the 32-bit destination register.



Syntax	Behavior

`Rd=vrndwh(Rss)`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=(Rss.w[i]+0x08000).h[1];`
	`}`

`Rd=vrndwh(Rss):sat`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=sat`$_{32}$`(Rss.w[i]+0x08000).h[1];`
	`}`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=vrndwh(Rss)`	`Word32 Q6_R_vrndwh_P(Word64 Rss)`
`Rd=vrndwh(Rss):sat`	`Word32 Q6_R_vrndwh_P_sat(Word64 Rss)`

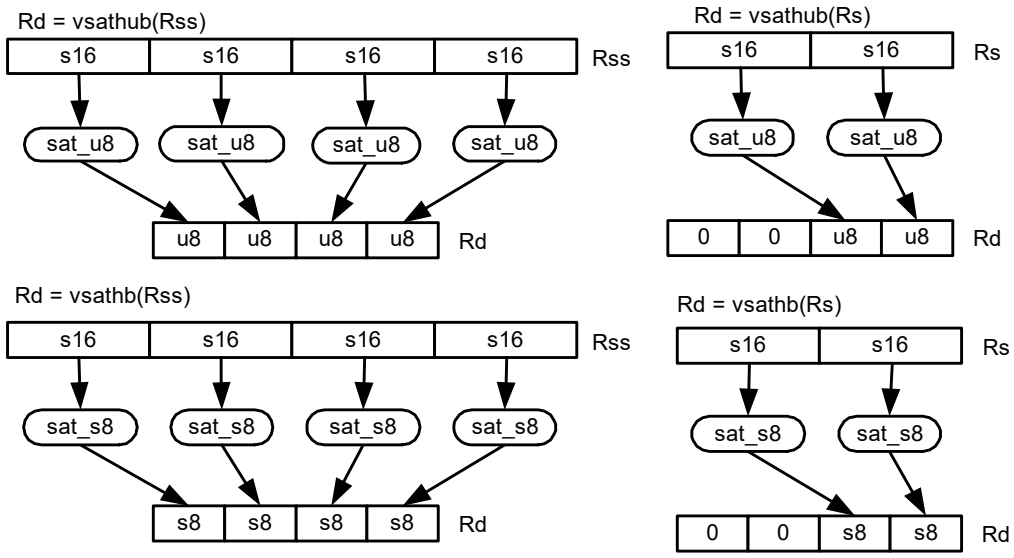
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**-**	**-**	**-**	**-**	**-**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=vrndwh(Rss)
1	0	0	0	1	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**-**	**-**	**-**	**-**	**-**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=vrndwh(Rss):sat

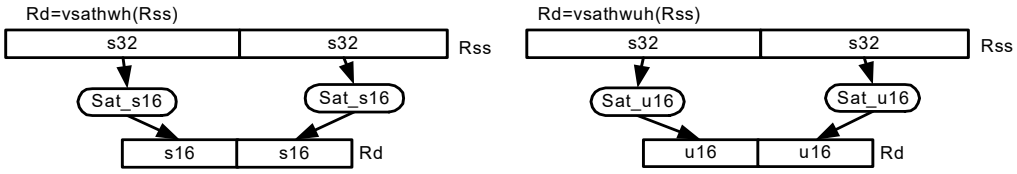
Field name	Description
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major opcode
`MinOp`	Minor opcode
`RegType`	Register type

Vector saturate and pack

For each element in the vector, saturate the value to the next smaller size. `vsathub` saturates signed halfwords to unsigned bytes, while `vsathb` saturates signed halfwords to signed bytes.



The `vsatwh` operation saturates signed words to signed halfwords, while `vsatwuh` saturates signed words to unsigned halfwords. The resulting values are packed together into destination register Rd.



Syntax	Behavior

`Rd=vsathb(Rs)`	`Rd.b[0]=sat8(Rs.h[0]);`
	`Rd.b[1]=sat8(Rs.h[1]);`
	`Rd.b[2]=0;`
	`Rd.b[3]=0;`

`Rd=vsathb(Rss)`	`for (i=0;i<4;i++) {`
	`  Rd.b[i]=sat8(Rss.h[i]);`
	`}`

`Rd=vsathub(Rs)`	`Rd.b[0]=usat8(Rs.h[0]);`
	`Rd.b[1]=usat8(Rs.h[1]);`
	`Rd.b[2]=0;`
	`Rd.b[3]=0;`

`Rd=vsathub(Rss)`	`for (i=0;i<4;i++) {`
	`  Rd.b[i]=usat8(Rss.h[i]);`
	`}`

`Rd=vsatwh(Rss)`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=sat16(Rss.w[i]);`
	`}`

Syntax	Behavior

`Rd=vsatwuh(Rss)`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=usat16(Rss.w[i]);`
	`}`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=vsathb(Rs)`	`Word32 Q6_R_vsathb_R(Word32 Rs)`
`Rd=vsathb(Rss)`	`Word32 Q6_R_vsathb_P(Word64 Rss)`
`Rd=vsathub(Rs)`	`Word32 Q6_R_vsathub_R(Word32 Rs)`
`Rd=vsathub(Rss)`	`Word32 Q6_R_vsathub_P(Word64 Rss)`
`Rd=vsatwh(Rss)`	`Word32 Q6_R_vsatwh_P(Word64 Rss)`
`Rd=vsatwuh(Rss)`	`Word32 Q6_R_vsatwuh_P(Word64 Rss)`

Encoding

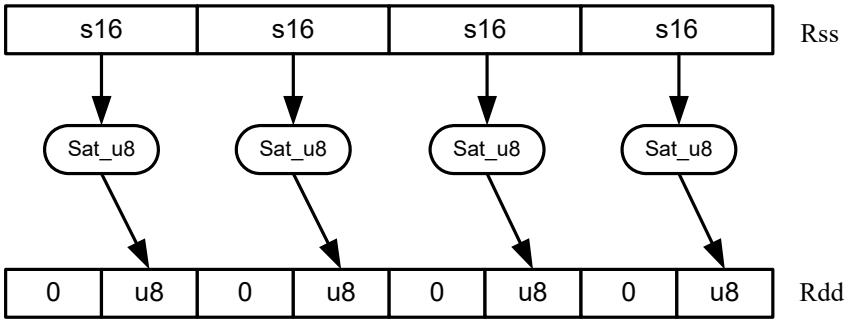
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=vsathub(Rss)
1	0	0	0	1	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=vsatwh(Rss)
1	0	0	0	1	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=vsatwuh(Rss)
1	0	0	0	1	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=vsathb(Rss)
1	0	0	0	1	1	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	-	**d**	**d**	**d**	**d**	**d**	Rd=vsathb(Rs)
1	0	0	0	1	1	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	1	-	**d**	**d**	**d**	**d**	**d**	Rd=vsathub(Rs)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

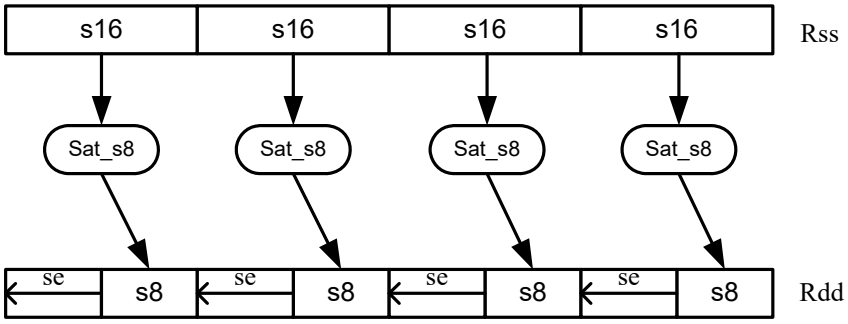
Vector saturate without pack

Saturate each element of source vector Rss to the next smaller size. `vsathub` saturates signed halfwords to unsigned bytes. `vsatwh` saturates signed words to signed halfwords, and `vsatwuh` saturates signed words to unsigned halfwords. The resulting values are placed in destination register Rdd in unpacked form.

Rdd=vsathub(Rss)



Rdd=vsathb(Rss)



Syntax	Behavior

`Rdd=vsathb(Rss)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=sat8(Rss.h[i]);`
	`}`

`Rdd=vsathub(Rss)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=usat8(Rss.h[i]);`
	`}`

`Rdd=vsatwh(Rss)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=sat16(Rss.w[i]);`
	`}`

`Rdd=vsatwuh(Rss)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=usat16(Rss.w[i]);`
	`}`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vsathb(Rss)`	`Word64 Q6_P_vsathb_P(Word64 Rss)`
`Rdd=vsathub(Rss)`	`Word64 Q6_P_vsathub_P(Word64 Rss)`
`Rdd=vsatwh(Rss)`	`Word64 Q6_P_vsatwh_P(Word64 Rss)`
`Rdd=vsatwuh(Rss)`	`Word64 Q6_P_vsatwuh_P(Word64 Rss)`

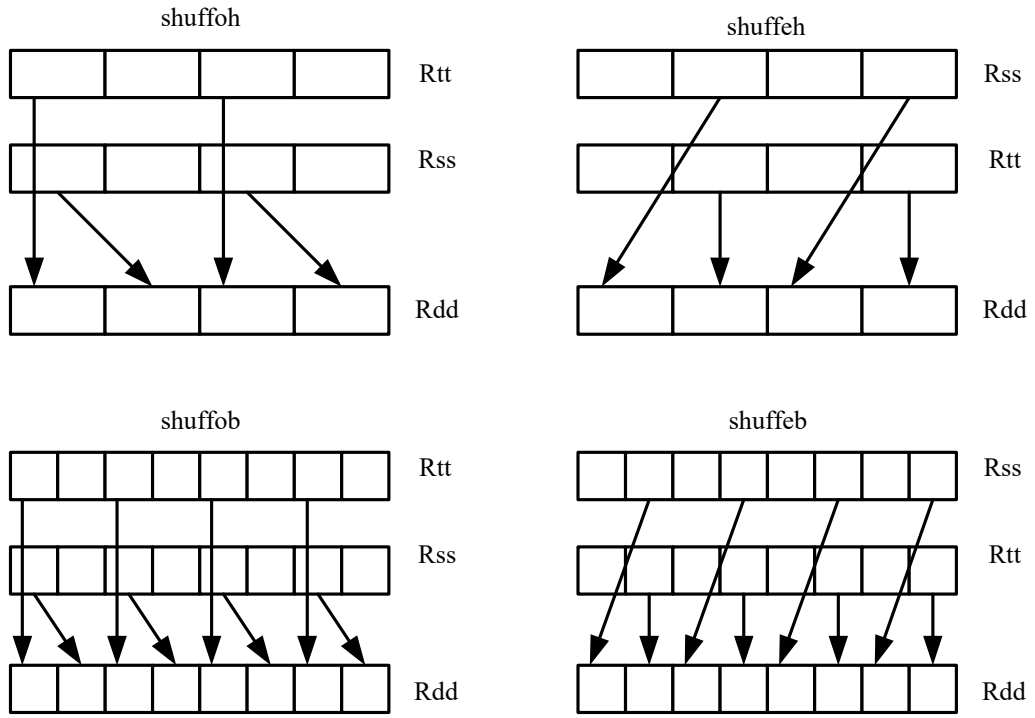
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vsathub(Rss)
1	0	0	0	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vsatwuh(Rss)
1	0	0	0	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vsatwh(Rss)
1	0	0	0	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vsathb(Rss)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

Vector shuffle

Shuffle odd halfwords (`shuffoh`) takes the odd halfwords from Rtt and the odd halfwords from Rss and merges them together into vector Rdd. Shuffle even halfwords (`shuffeh`) performs the same operation on every even halfword in Rss and Rtt. The same operation is available for odd and even bytes.



Syntax	Behavior

`Rdd=shuffeb(Rss,Rtt)`	`for (i=0;i<4;i++) {`
	`  Rdd.b[i*2]=Rtt.b[i*2];`
	`  Rdd.b[i*2+1]=Rss.b[i*2];`
	`}`

`Rdd=shuffeh(Rss,Rtt)`	`for (i=0;i<2;i++) {`
	`  Rdd.h[i*2]=Rtt.h[i*2];`
	`  Rdd.h[i*2+1]=Rss.h[i*2];`
	`}`

`Rdd=shuffob(Rtt,Rss)`	`for (i=0;i<4;i++) {`
	`  Rdd.b[i*2]=Rss.b[i*2+1];`
	`  Rdd.b[i*2+1]=Rtt.b[i*2+1];`
	`}`

`Rdd=shuffoh(Rtt,Rss)`	`for (i=0;i<2;i++) {`
	`  Rdd.h[i*2]=Rss.h[i*2+1];`
	`  Rdd.h[i*2+1]=Rtt.h[i*2+1];`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=shuffeb(Rss,Rtt)`	`Word64 Q6_P_shuffeb_PP(Word64 Rss, Word64 Rtt)`
`Rdd=shuffeh(Rss,Rtt)`	`Word64 Q6_P_shuffeh_PP(Word64 Rss, Word64 Rtt)`
`Rdd=shuffob(Rtt,Rss)`	`Word64 Q6_P_shuffob_PP(Word64 Rtt, Word64 Rss)`
`Rdd=shuffoh(Rtt,Rss)`	`Word64 Q6_P_shuffoh_PP(Word64 Rtt, Word64 Rss)`

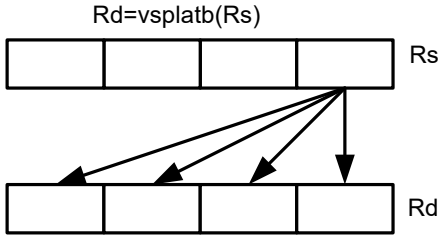
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	0	1	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=shuffeb(Rss,Rtt)
1	1	0	0	0	0	0	1	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=shuffob(Rtt,Rss)
1	1	0	0	0	0	0	1	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=shuffeh(Rss,Rtt)
1	1	0	0	0	0	0	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=shuffoh(Rtt,Rss)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
Maj	Major opcode
Min	Minor opcode
RegType	Register type

Vector splat bytes

Replicate the low 8-bits from register Rs into each of the four bytes of destination register Rd.



Syntax	Behavior

`Rd=vsplatb(Rs)`	`for (i=0;i<4;i++) {`
	`  Rd.b[i]=Rs.b[0];`
	`}`

`Rdd=vsplatb(Rs)`	`for (i=0;i<8;i++) {`
	`  Rdd.b[i]=Rs.b[0];`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=vsplatb(Rs)`	`Word32 Q6_R_vsplatb_R(Word32 Rs)`
`Rdd=vsplatb(Rs)`	`Word64 Q6_P_vsplatb_R(Word32 Rs)`

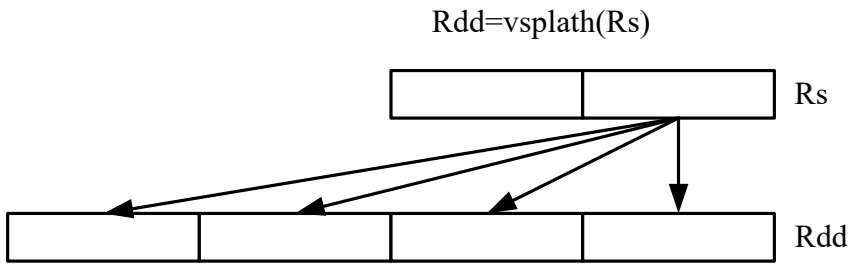
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	1	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=vsplatb(Rs)
1	0	0	0	1	1	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=vsplatb(Rs)

Field name	Description
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major opcode
`MinOp`	Minor opcode
`RegType`	Register type

Vector splat halfwords

Replicate the low 16-bits from register Rs into each of the four halfwords of destination Rdd.



Syntax	Behavior

`Rdd=vsplath(Rs)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=Rs.h[0];`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vsplath(Rs)`	`Word64 Q6_P_vsplath_R(Word32 Rs)`

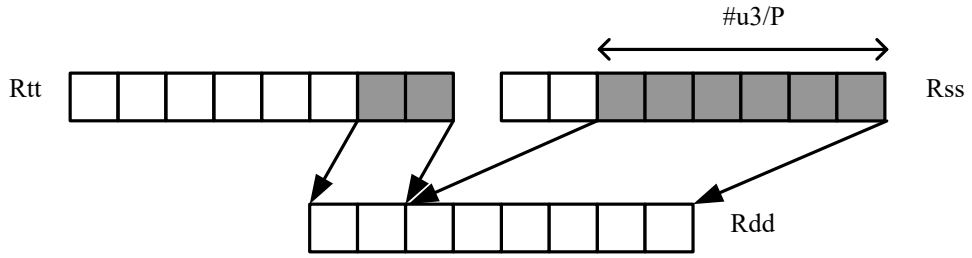
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	0	1	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vsplath(Rs)

Field name	Description
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major opcode
`MinOp`	Minor opcode
`RegType`	Register type

Vector splice

Concatenate the low (8-N) bytes of vector Rtt with the low N bytes of vector Rss. This instruction is helpful to vectorize unaligned stores.



Syntax	Behavior
`Rdd=vspliceb(Rss,Rtt,#u3)`	`Rdd = Rtt << #u*8 | zxt`$_{\#u*8->64}$`(Rss);`
`Rdd=vspliceb(Rss,Rtt,Pu)`	`PREDUSE_TIMING;` `Rdd = Rtt << (Pu&7)*8 | zxt`$_{(Pu\&7)*8->64}$`(Rss);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vspliceb(Rss,Rtt,#u3)`	`Word64 Q6_P_vspliceb_PPI(Word64 Rss, Word64 Rtt, Word32 Iu3)`
`Rdd=vspliceb(Rss,Rtt,Pu)`	`Word64 Q6_P_vspliceb_PPp(Word64 Rss, Word64 Rtt, Byte Pu)`

Encoding

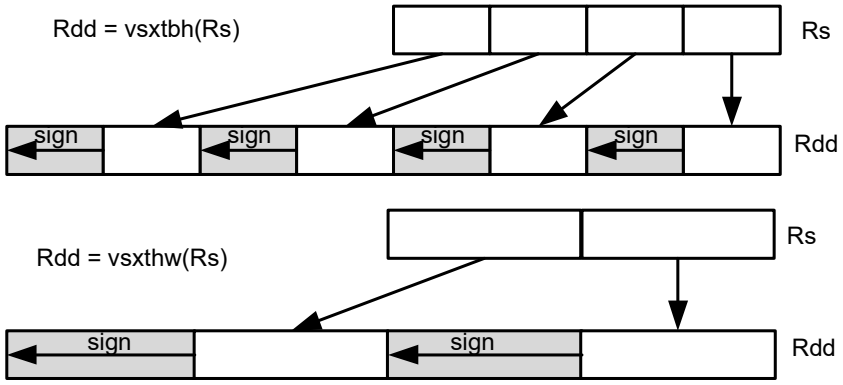
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	0	0	1	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	i	i	i	d	d	d	d	d	Rdd=vspliceb(Rss,Rtt,#u3)
ICLASS				RegType				Maj			s5					Parse			t5						u2		d5					
1	1	0	0	0	0	1	0	1	0	0	s	s	s	s	s	P	P	-	t	t	t	t	t	-	u	u	d	d	d	d	d	Rdd=vspliceb(Rss,Rtt,Pu)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
u2	Field to encode register u
Maj	Major opcode
Min	Minor opcode
RegType	Register type

Vector sign extend

`vsxtbh` sign-extends each byte of a single register source to halfwords, and places the result in the destination register pair.

`vsxthw` sign-extends each halfword of a single register source to words, and places the result in the destination register pair.



Syntax	Behavior

`Rdd=vsxtbh(Rs)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=Rs.b[i];`
	`}`

`Rdd=vsxthw(Rs)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=Rs.h[i];`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vsxtbh(Rs)`	`Word64 Q6_P_vsxtbh_R(Word32 Rs)`
`Rdd=vsxthw(Rs)`	`Word64 Q6_P_vsxthw_R(Word32 Rs)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	1	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=vsxtbh(Rs)
1	0	0	0	0	1	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=vsxthw(Rs)

Field name	Description
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major opcode
`MinOp`	Minor opcode
`RegType`	Register type

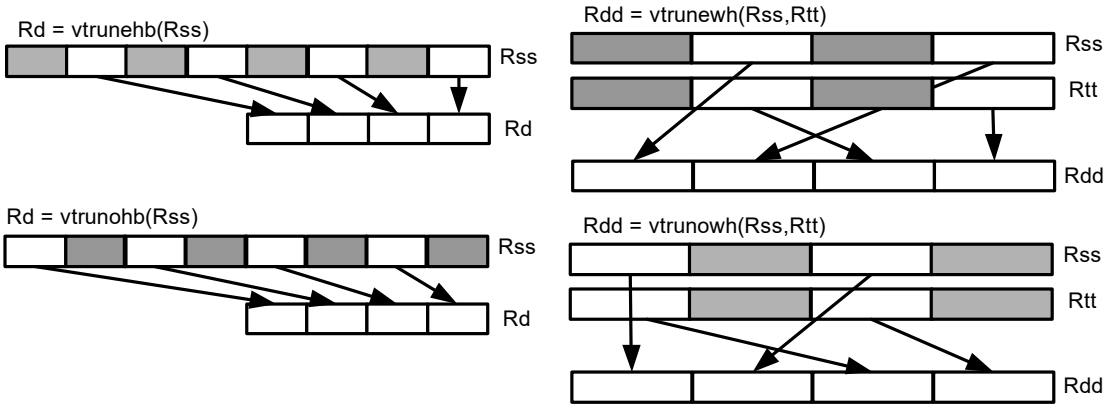
Vector truncate

In `vtrunehb`, for each halfword in a vector, take the even (lower) byte and ignore the other byte. The resulting values are packed into destination register Rd.

`vtrunohb` takes each odd byte of the source vector.

`vtrunewh` uses two source register pairs, Rss and Rtt. The even (lower) halfwords of Rss are packed in the upper word of Rdd, while the lower halfwords of Rtt are packed in the lower word of Rdd.

`vtrunowh` performs the same operation as `vtrunewh`, but uses the odd (upper) halfwords of the source vectors instead.



Syntax	Behavior

`Rd=vtrunehb(Rss)`	`for (i=0;i<4;i++) {`
	`  Rd.b[i]=Rss.b[i*2];`
	`}`

`Rd=vtrunohb(Rss)`	`for (i=0;i<4;i++) {`
	`  Rd.b[i]=Rss.b[i*2+1];`
	`}`

`Rdd=vtrunehb(Rss,Rtt)`	`for (i=0;i<4;i++) {`
	`  Rdd.b[i]=Rtt.b[i*2];`
	`  Rdd.b[i+4]=Rss.b[i*2];`
	`}`

`Rdd=vtrunewh(Rss,Rtt)`	`Rdd.h[0]=Rtt.h[0];`
	`Rdd.h[1]=Rtt.h[2];`
	`Rdd.h[2]=Rss.h[0];`
	`Rdd.h[3]=Rss.h[2];`

`Rdd=vtrunohb(Rss,Rtt)`	`for (i=0;i<4;i++) {`
	`  Rdd.b[i]=Rtt.b[i*2+1];`
	`  Rdd.b[i+4]=Rss.b[i*2+1];`
	`}`

`Rdd=vtrunowh(Rss,Rtt)`	`Rdd.h[0]=Rtt.h[1];`
	`Rdd.h[1]=Rtt.h[3];`
	`Rdd.h[2]=Rss.h[1];`
	`Rdd.h[3]=Rss.h[3];`

Class: XTYPE (slots 2,3)

Intrinsics

Rd=vtrunehb(Rss)	Word32 Q6_R_vtrunehb_P(Word64 Rss)
Rd=vtrunohb(Rss)	Word32 Q6_R_vtrunohb_P(Word64 Rss)
Rdd=vtrunehb(Rss,Rtt)	Word64 Q6_P_vtrunehb_PP(Word64 Rss, Word64 Rtt)
Rdd=vtrunewh(Rss,Rtt)	Word64 Q6_P_vtrunewh_PP(Word64 Rss, Word64 Rtt)
Rdd=vtrunohb(Rss,Rtt)	Word64 Q6_P_vtrunohb_PP(Word64 Rss, Word64 Rtt)
Rdd=vtrunowh(Rss,Rtt)	Word64 Q6_P_vtrunowh_PP(Word64 Rss, Word64 Rtt)

Encoding

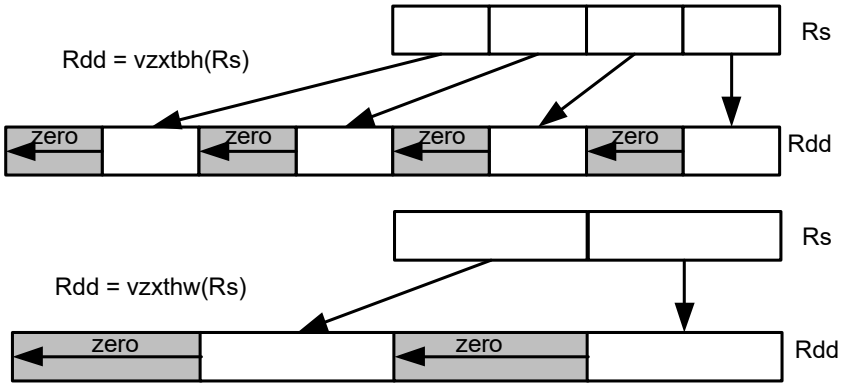
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=vtrunohb(Rss)
1	0	0	0	1	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=vtrunehb(Rss)
ICLASS				RegType				MajoP			s5					Parse			t5						MinoP			d5					
1	1	0	0	0	0	0	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vtrunewh(Rss,Rtt)
1	1	0	0	0	0	0	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vtrunehb(Rss,Rtt)
1	1	0	0	0	0	0	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vtrunowh(Rss,Rtt)
1	1	0	0	0	0	0	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vtrunohb(Rss,Rtt)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

Vector zero extend

The `vzxtbh` instruction zero-extends each byte of a single register source to halfwords, and places the result in the destination register pair.

The `vzxthw` instruction zero-extends each halfword of a single register source to words, and places the result in the destination register pair.



Syntax	Behavior

`Rdd=vzxtbh(Rs)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=Rs.ub[i];`
	`}`

`Rdd=vzxthw(Rs)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=Rs.uh[i];`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vzxtbh(Rs)`	`Word64 Q6_P_vzxtbh_R(Word32 Rs)`
`Rdd=vzxthw(Rs)`	`Word64 Q6_P_vzxthw_R(Word32 Rs)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	0	1	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vzxtbh(Rs)
1	0	0	0	0	1	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vzxthw(Rs)

Field name	Description
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major opcode
`MinOp`	Minor opcode
`RegType`	Register type

11.10.7 XTYPE PRED

The XTYPE PRED instruction subclass includes instructions that perform miscellaneous operations on predicates, including mask generation, predicate transfers, and the Viterbi pack operation.

Bounds check

Determines if Rs falls in the range defined by Rtt.

Rtt.w0 is set by the user to the lower bound, and Rtt.w1 is set by the user to the upper bound.

All bits of the destination predicate are set if the value falls within the range, or all cleared otherwise.

Syntax	Behavior
`Pd=boundscheck(Rs,Rtt)`	`if ("Rs & 1") { Assembler mapped to: "Pd=boundscheck(Rss,Rtt):raw:hi"; } else { Assembler mapped to: "Pd = boundscheck(Rss, Rtt):raw:lo"; }`
`Pd=boundscheck(Rss, Rtt):raw:hi`	`src = Rss.uw[1]; Pd = (src.uw[0] >= Rtt.uw[0]) && (src.uw[0] < Rtt.uw[1]) ? 0xff : 0x00;`
`Pd=boundscheck(Rss, Rtt):raw:lo`	`src = Rss.uw[0]; Pd = (src.uw[0] >= Rtt.uw[0]) && (src.uw[0] < Rtt.uw[1]) ? 0xff : 0x00;`

Class: XTYPE (slots 2,3)

Intrinsics

`Pd=boundscheck(Rs,Rtt)`	`Byte Q6_p_boundscheck_RP(Word32 Rs, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp						d2		
1	1	0	1	0	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	0	0	-	-	-	**d**	**d**	Pd=boundscheck(Rss,Rtt):raw:lo
1	1	0	1	0	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	0	1	-	-	-	**d**	**d**	Pd=boundscheck(Rss,Rtt):raw:hi

Field name	Description
RegType	Register type
MinOp	Minor opcode
ICLASS	Instruction class
Parse	Packet/loop parse bits
d2	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Compare byte

These instructions sign- or zero-extend the low 8-bits of the source registers and perform 32-bit comparisons on the result. For an extended 32-bit immediate operand, the full 32 immediate bits are used for the comparison.

Syntax	Behavior
`Pd=cmpb.eq(Rs,#u8)`	`Pd=Rs.ub[0] == #u ? 0xff : 0x00;`
`Pd=cmpb.eq(Rs,Rt)`	`Pd=Rs.b[0] == Rt.b[0] ? 0xff : 0x00;`
`Pd=cmpb.gt(Rs,#s8)`	`Pd=Rs.b[0] > #s ? 0xff : 0x00;`
`Pd=cmpb.gt(Rs,Rt)`	`Pd=Rs.b[0] > Rt.b[0] ? 0xff : 0x00;`
`Pd=cmpb.gtu(Rs,#u7)`	`apply_extension(#u);` `Pd=Rs.ub[0] > #u.uw[0] ? 0xff : 0x00;`
`Pd=cmpb.gtu(Rs,Rt)`	`Pd=Rs.ub[0] > Rt.ub[0] ? 0xff : 0x00;`

Class: XTYPE (slots 2,3)

Intrinsics

`Pd=cmpb.eq(Rs,#u8)`	`Byte Q6_p_cmpb_eq_RI(Word32 Rs, Word32 Iu8)`
`Pd=cmpb.eq(Rs,Rt)`	`Byte Q6_p_cmpb_eq_RR(Word32 Rs, Word32 Rt)`
`Pd=cmpb.gt(Rs,#s8)`	`Byte Q6_p_cmpb_gt_RI(Word32 Rs, Word32 Is8)`
`Pd=cmpb.gt(Rs,Rt)`	`Byte Q6_p_cmpb_gt_RR(Word32 Rs, Word32 Rt)`
`Pd=cmpb.gtu(Rs,#u7)`	`Byte Q6_p_cmpb_gtu_RI(Word32 Rs, Word32 Iu7)`
`Pd=cmpb.gtu(Rs,Rt)`	`Byte Q6_p_cmpb_gtu_RR(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min						d2		
1	1	0	0	0	1	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	-	-	-	**d**	**d**	Pd=cmpb.gt(Rs,Rt)
1	1	0	0	0	1	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	-	-	-	**d**	**d**	Pd=cmpb.eq(Rs,Rt)
1	1	0	0	0	1	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	-	-	-	**d**	**d**	Pd=cmpb.gtu(Rs,Rt)
ICLASS				RegType							s5					Parse														d2		
1	1	0	1	1	1	0	1	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	0	-	**d**	**d**	Pd=cmpb.eq(Rs,#u8)
1	1	0	1	1	1	0	1	-	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	0	-	**d**	**d**	Pd=cmpb.gt(Rs,#s8)
1	1	0	1	1	1	0	1	-	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	0	-	**d**	**d**	Pd=cmpb.gtu(Rs,#u7)

Field name	Description
`RegType`	Register type
`Maj`	Major opcode
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d2`	Field to encode register d
`t5`	Field to encode register t
`Min`	Minor opcode

Compare half

These instructions sign- or zero-extend the low 16-bits of the source registers and perform 32-bit comparisons on the result. In the case of an extended 32-bit immediate operand, the full 32 immediate bits are used for the comparison.

Syntax	Behavior
`Pd=cmph.eq(Rs,#s8)`	`apply_extension(#s);` `Pd=Rs.h[0] == #s ? 0xff : 0x00;`
`Pd=cmph.eq(Rs,Rt)`	`Pd=Rs.h[0] == Rt.h[0] ? 0xff : 0x00;`
`Pd=cmph.gt(Rs,#s8)`	`apply_extension(#s);` `Pd=Rs.h[0] > #s ? 0xff : 0x00;`
`Pd=cmph.gt(Rs,Rt)`	`Pd=Rs.h[0] > Rt.h[0] ? 0xff : 0x00;`
`Pd=cmph.gtu(Rs,#u7)`	`apply_extension(#u);` `Pd=Rs.uh[0] > #u.uw[0] ? 0xff : 0x00;`
`Pd=cmph.gtu(Rs,Rt)`	`Pd=Rs.uh[0] > Rt.uh[0] ? 0xff : 0x00;`

Class: XTYPE (slots 2,3)

Intrinsics

`Pd=cmph.eq(Rs,#s8)`	`Byte Q6_p_cmph_eq_RI(Word32 Rs, Word32 Is8)`
`Pd=cmph.eq(Rs,Rt)`	`Byte Q6_p_cmph_eq_RR(Word32 Rs, Word32 Rt)`
`Pd=cmph.gt(Rs,#s8)`	`Byte Q6_p_cmph_gt_RI(Word32 Rs, Word32 Is8)`
`Pd=cmph.gt(Rs,Rt)`	`Byte Q6_p_cmph_gt_RR(Word32 Rs, Word32 Rt)`
`Pd=cmph.gtu(Rs,#u7)`	`Byte Q6_p_cmph_gtu_RI(Word32 Rs, Word32 Iu7)`
`Pd=cmph.gtu(Rs,Rt)`	`Byte Q6_p_cmph_gtu_RR(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min						d2		
1	1	0	0	0	1	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	-	-	-	**d**	**d**	Pd=cmph.eq(Rs,Rt)
1	1	0	0	0	1	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	-	-	-	**d**	**d**	Pd=cmph.gt(Rs,Rt)
1	1	0	0	0	1	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	-	-	-	**d**	**d**	Pd=cmph.gtu(Rs,Rt)
ICLASS				RegType							s5					Parse														d2		
1	1	0	1	1	1	0	1	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	1	-	**d**	**d**	Pd=cmph.eq(Rs,#s8)
1	1	0	1	1	1	0	1	-	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	1	-	**d**	**d**	Pd=cmph.gt(Rs,#s8)
1	1	0	1	1	1	0	1	-	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	1	-	**d**	**d**	Pd=cmph.gtu(Rs,#u7)

Field name	Description
RegType	Register type
ICLASS	Instruction class
Parse	Packet/loop parse bits
d2	Field to encode register d
s5	Field to encode register s

Field name	Description
t5	Field to encode register t
Maj	Major opcode
Min	Minor opcode

Compare doublewords

Compare two 64-bit register pairs for unsigned greater than, greater than, or equal. The 8-bit predicate register Pd is set to all 1s or all 0s, depending on the result.

Syntax	Behavior
`Pd=cmp.eq(Rss, Rtt)`	`Pd = Rss==Rtt ? 0xff : 0x00;`
`Pd=cmp.gt(Rss, Rtt)`	`Pd = Rss > Rtt ? 0xff : 0x00;`
`Pd=cmp.gtu(Rss, Rtt)`	`Pd = Rss.u64 > Rtt.u64 ? 0xff : 0x00;`

Class: XTYPE (slots 2,3)

Intrinsics

`Pd=cmp.eq(Rss,Rtt)`	`Byte Q6_p_cmp_eq_PP(Word64 Rss, Word64 Rtt)`
`Pd=cmp.gt(Rss,Rtt)`	`Byte Q6_p_cmp_gt_PP(Word64 Rss, Word64 Rtt)`
`Pd=cmp.gtu(Rss,Rtt)`	`Byte Q6_p_cmp_gtu_PP(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5						MinOp						d2		
1	1	0	1	0	0	1	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	-	-	-	**d**	**d**	Pd=cmp.eq(Rss,Rtt)
1	1	0	1	0	0	1	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	-	-	-	**d**	**d**	Pd=cmp.gt(Rss,Rtt)
1	1	0	1	0	0	1	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	-	-	-	**d**	**d**	Pd=cmp.gtu(Rss,Rtt)

Field name	Description
`RegType`	Register type
`MinOp`	Minor opcode
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d2`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Compare bit mask

If all the bits in the mask in Rt or a short immediate are set (`bitsset`) or clear (`bitsclr` in Rs, set the Pd to true. Otherwise, set the bits in Pd to false.

Syntax	Behavior
`Pd=[!]bitsclr(Rs, #u6)`	`Pd = (Rs&#u)[!] = 0 ? 0xff : 0x00;`
`Pd=[!]bitsclr(Rs, Rt)`	`Pd = (Rs&Rt)[!] = 0 ? 0xff : 0x00;`
`Pd=[!]bitsset(Rs, Rt)`	`Pd = (Rs&Rt)[!] = Rt ? 0xff : 0x00;`

Class: XTYPE (slots 2,3)

Intrinsics

`Pd=!bitsclr(Rs,#u6)`	`Byte Q6_p_not_bitsclr_RI(Word32 Rs, Word32 Iu6)`
`Pd=!bitsclr(Rs,Rt)`	`Byte Q6_p_not_bitsclr_RR(Word32 Rs, Word32 Rt)`
`Pd=!bitsset(Rs,Rt)`	`Byte Q6_p_not_bitsset_RR(Word32 Rs, Word32 Rt)`
`Pd=bitsclr(Rs,#u6)`	`Byte Q6_p_bitsclr_RI(Word32 Rs, Word32 Iu6)`
`Pd=bitsclr(Rs,Rt)`	`Byte Q6_p_bitsclr_RR(Word32 Rs, Word32 Rt)`
`Pd=bitsset(Rs,Rt)`	`Byte Q6_p_bitsset_RR(Word32 Rs, Word32 Rt)`

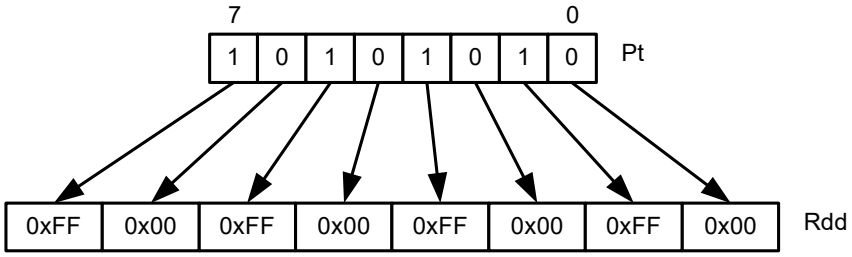
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse															d2		
1	0	0	0	0	1	0	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	-	-	-	-	-	-	**d**	**d**	Pd=bitsclr(Rs,#u6)
1	0	0	0	0	1	0	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	-	-	-	-	-	-	**d**	**d**	Pd=!bitsclr(Rs,#u6)
ICLASS				RegType				Maj			s5					Parse			t5											d2		
1	1	0	0	0	1	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=bitsset(Rs,Rt)
1	1	0	0	0	1	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=!bitsset(Rs,Rt)
1	1	0	0	0	1	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=bitsclr(Rs,Rt)
1	1	0	0	0	1	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=!bitsclr(Rs,Rt)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d2	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
MajOp	Major opcode
Maj	Major opcode
RegType	Register type

Mask generate from predicate

For each of the low 8 bits in predicate register Pt, if the bit is set, set the corresponding byte in 64-bit register pair Rdd to 0xff, otherwise, set the corresponding byte to 0x00.



Syntax	Behavior

`Rdd = mask(Pt)`	`PREDUSE_TIMING;`
	`for (i = 0; i < 8; i++) {`
	`  Rdd.b[i]=(Pt.i?(0xff):(0x00));`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=mask(Pt)`	`Word64 Q6_P_mask_p(Byte Pt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType												Parse						t2					d5						
1	0	0	0	0	1	1	0	-	-	-	-	-	-	-	-	**P**	**P**	-	-	-	-	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rdd=mask(Pt)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
t2	Field to encode register t
RegType	Register type

Check for TLB match

Determine if the TLB entry in Rss matches the ASID:PPN in Rt.

Syntax	Behavior

`Pd =tlbmatch(Rss,Rt)`	`MASK = 0x07ffffff;`
	`TLBLO = Rss.uw[0];`
	`TLBHI = Rss.uw[1];`
	`SIZE = min(6,`
	`count_leading_ones(~reverse_bits(TLBLO)));`
	`MASK &= (0xffffffff << 2*SIZE);`
	`Pd = TLBHI.31 && ((TLBHI & MASK) == (Rt & MASK)) ?`
	`0xff : 0x00;`

Class: XTYPE (slots 2,3)

Notes

- The predicate generated by this instruction cannot be used as a .new predicate, nor can it be automatically AND'd with another predicate.

Intrinsics

`Pd = tlbmatch(Rss, Rt)`	`Byte Q6_p_tlbmatch_PR(Word64 Rss, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp						d2		
1	1	0	1	0	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	1	1	-	-	-	**d**	**d**	Pd=tlbmatch(Rss,Rt)

Field name	Description
`RegType`	Register type
`MinOp`	Minor opcode
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d2`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Predicate transfer

`Pd = Rs` transfers a predicate to the eight least-significant bits of a general register and zeros the other bits.

`Rd = Ps` transfers the eight least-significant bits of a general register to a predicate.

Syntax	Behavior
`Pd = Rs`	`Pd = Rs.ub[0];`
`Rd = Ps`	`PREDUSE_TIMING;` `Rd = zxt`$_{8->32}$`(Ps);`

Class: XTYPE (slots 2,3)

Intrinsics

`Pd = Rs`	`Byte Q6_p_equals_R(Word32 Rs)`
`Rd = Ps`	`Word32 Q6_R_equals_p(Byte Ps)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse														d2		
1	0	0	0	0	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	**d**	**d**	Pd=Rs
ICLASS				RegType				MajOp						s2		Parse												d5				
1	0	0	0	1	0	0	1	-	1	-	-	-	-	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=Ps

Field name	Description
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d2`	Field to encode register d
`d5`	Field to encode register d
`s2`	Field to encode register s
`s5`	Field to encode register s
`MajOp`	Major opcode
`RegType`	Register type

Test bit

Extract a bit from a register. If the bit is true (1), set all the bits of the predicate register destination to 1. If the bit is false (0), set all the bits of the predicate register destination to 0. The bit to test can be indicated using an immediate or register value.

If a register is used to indicate the bit to test, and the value specified is out of range, the predicate result is zero.

Syntax	Behavior
`Pd=[!]tstbit(Rs,#u5)`	`Pd = (Rs & (1<<#u)) == 0 ? 0xff : 0x00;`
`Pd=[!]tstbit(Rs,Rt)`	`Pd = (zxt32->64(Rs) & (sxt7->32(Rt)>0)?(zxt32->64(1)<< sxt7->32(Rt)):(zxt32->64(1)>>>sxt7->32(Rt))) == 0 ? 0xff : 0x00;`

Class: XTYPE (slots 2,3)

Intrinsics

`Pd =!tstbit(Rs, #u5)`	`Byte Q6_p_not_tstbit_RI(Word32 Rs, Word32 Iu5)`
`Pd =!tstbit(Rs, Rt)`	`Byte Q6_p_not_tstbit_RR(Word32 Rs, Word32 Rt)`
`Pd =tstbit(Rs, #u5)`	`Byte Q6_p_tstbit_RI(Word32 Rs, Word32 Iu5)`
`Pd =tstbit(Rs, Rt)`	`Byte Q6_p_tstbit_RR(Word32 Rs, Word32 Rt)`

Encoding

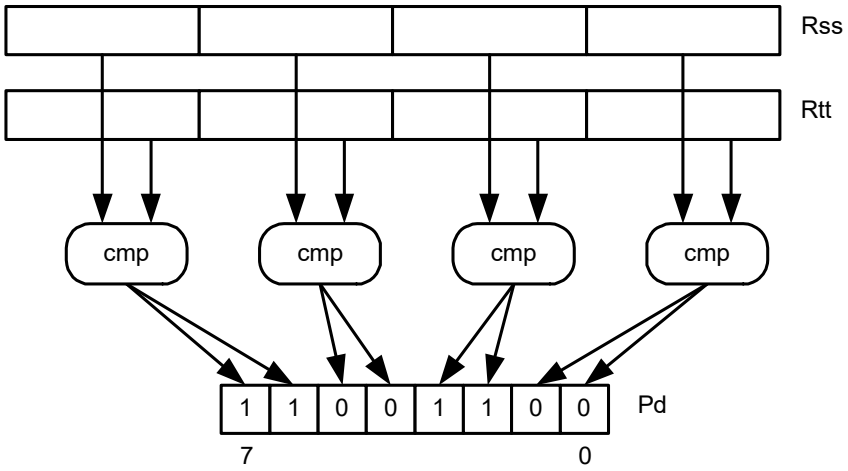
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse															d2		
1	0	0	0	0	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	-	-	-	-	-	-	**d**	**d**	Pd=tstbit(Rs,#u5)
1	0	0	0	0	1	0	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	-	-	-	-	-	-	**d**	**d**	Pd=!tstbit(Rs,#u5)
ICLASS				RegType				Maj			s5					Parse			t5											d2		
1	1	0	0	0	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=tstbit(Rs,Rt)
1	1	0	0	0	1	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=!tstbit(Rs,Rt)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d2	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
MajOp	Major opcode
Maj	Major opcode
RegType	Register type

Vector compare halfwords

Compare each of four 16-bit halfwords in two 64-bit vectors and set the corresponding bits in a predicate destination to '11' if true, '00' if false.

Halfword comparisons can be for equal, signed greater than, or unsigned greater than.



Syntax	Behavior

`Pd=vcmph.eq(Rss,#s8)`	`for (i = 0; i < 4; i++) {`
	`  Pd.i*2 = (Rss.h[i] == #s);`
	`  Pd.i*2+1 = (Rss.h[i] == #s);`
	`}`

`Pd=vcmph.eq(Rss,Rtt)`	`for (i = 0; i < 4; i++) {`
	`  Pd.i*2 = (Rss.h[i] == Rtt.h[i]);`
	`  Pd.i*2+1 = (Rss.h[i] == Rtt.h[i]);`
	`}`

`Pd=vcmph.gt(Rss,#s8)`	`for (i = 0; i < 4; i++) {`
	`  Pd.i*2 = (Rss.h[i] > #s);`
	`  Pd.i*2+1 = (Rss.h[i] > #s);`
	`}`

`Pd=vcmph.gt(Rss,Rtt)`	`for (i = 0; i < 4; i++) {`
	`  Pd.i*2 = (Rss.h[i] > Rtt.h[i]);`
	`  Pd.i*2+1 = (Rss.h[i] > Rtt.h[i]);`
	`}`

`Pd=vcmph.gtu(Rss,#u7)`	`for (i = 0; i < 4; i++) {`
	`  Pd.i*2 = (Rss.uh[i] > #u);`
	`  Pd.i*2+1 = (Rss.uh[i] > #u);`
	`}`

`Pd=vcmph.gtu(Rss,Rtt)`	`for (i = 0; i < 4; i++) {`
	`  Pd.i*2 = (Rss.uh[i] > Rtt.uh[i]);`
	`  Pd.i*2+1 = (Rss.uh[i] > Rtt.uh[i]);`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

```
Pd=vcmph.eq(Rss,#s8)    Byte Q6_p_vcmph_eq_PI(Word64 Rss, Word32 Is8)
Pd=vcmph.eq(Rss,Rtt)    Byte Q6_p_vcmph_eq_PP(Word64 Rss, Word64 Rtt)
Pd=vcmph.gt(Rss,#s8)    Byte Q6_p_vcmph_gt_PI(Word64 Rss, Word32 Is8)
Pd=vcmph.gt(Rss,Rtt)    Byte Q6_p_vcmph_gt_PP(Word64 Rss, Word64 Rtt)
Pd=vcmph.gtu(Rss,#u7)   Byte Q6_p_vcmph_gtu_PI(Word64 Rss, Word32 Iu7)
Pd=vcmph.gtu(Rss,Rtt)   Byte Q6_p_vcmph_gtu_PP(Word64 Rss, Word64 Rtt)
```

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp					d2		
1	1	0	1	0	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	1	-	-	-	**d**	**d**	Pd=vcmph.eq(Rss,Rtt)
1	1	0	1	0	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	-	-	-	**d**	**d**	Pd=vcmph.gt(Rss,Rtt)
1	1	0	1	0	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	1	-	-	-	**d**	**d**	Pd=vcmph.gtu(Rss,Rtt)
ICLASS				RegType							s5					Parse														d2		
1	1	0	1	1	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	1	-	**d**	**d**	Pd=vcmph.eq(Rss,#s8)
1	1	0	1	1	1	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	1	-	**d**	**d**	Pd=vcmph.gt(Rss,#s8)
1	1	0	1	1	1	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	1	-	**d**	**d**	Pd=vcmph.gtu(Rss,#u7)

Field name	Description
RegType	Register type
MajOp	Major opcode
MinOp	Minor opcode
ICLASS	Instruction class
Parse	Packet/loop parse bits
d2	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector compare bytes for any match

Compare each byte in two 64-bit source vectors and set a predicate if any of the 8 bytes are equal.

This instruction can be used to quickly find the null terminator in a string.

Syntax	Behavior

`Pd=!any8(vcmpb.eq(Rss,Rtt))`	`Pd = 0;`
	`for (i = 0; i < 8; i++) {`
	`  if (Rss.b[i] == Rtt.b[i]) Pd = 0xff;`
	`}`
	`Pd = ~Pd;`

`Pd=any8(vcmpb.eq(Rss,Rtt))`	`Pd = 0;`
	`for (i = 0; i < 8; i++) {`
	`  if (Rss.b[i] == Rtt.b[i]) Pd = 0xff;`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

`Pd=!any8(vcmpb.eq(Rss,Rtt))`	`Byte Q6_p_not_any8_vcmpb_eq_PP(Word64 Rss, Word64 Rtt)`
`Pd=any8(vcmpb.eq(Rss,Rtt))`	`Byte Q6_p_any8_vcmpb_eq_PP(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5						MinOp						d2		
1	1	0	1	0	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	0	0	-	-	-	**d**	**d**	Pd=any8(vcmpb.eq(Rss,Rtt))
1	1	0	1	0	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	0	1	-	-	-	**d**	**d**	Pd=!any8(vcmpb.eq(Rss,Rtt))

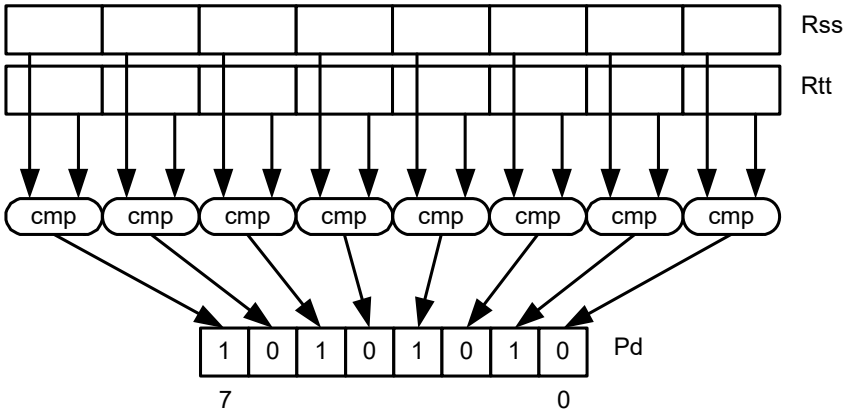
Field name	Description
`RegType`	Register type
`MinOp`	Minor opcode
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d2`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector compare bytes

Compare each of eight bytes in two 64-bit vectors and set the corresponding bit in a predicate destination to 1 if true, 0 if false.

Byte comparisons can be for equal or for unsigned greater than.

In the following example, every other comparison is true.



Syntax	Behavior

`Pd=vcmpb.eq(Rss,#u8)`	`for (i = 0; i < 8; i++) {`
	`  Pd.i = (Rss.ub[i] == #u);`
	`}`

`Pd=vcmpb.eq(Rss,Rtt)`	`for (i = 0; i < 8; i++) {`
	`  Pd.i = (Rss.b[i] == Rtt.b[i]);`
	`}`

`Pd=vcmpb.gt(Rss,#s8)`	`for (i = 0; i < 8; i++) {`
	`  Pd.i = (Rss.b[i] > #s);`
	`}`

`Pd=vcmpb.gt(Rss,Rtt)`	`for (i = 0; i < 8; i++) {`
	`  Pd.i = (Rss.b[i] > Rtt.b[i]);`
	`}`

`Pd=vcmpb.gtu(Rss,#u7)`	`for (i = 0; i < 8; i++) {`
	`  Pd.i = (Rss.ub[i] > #u);`
	`}`

`Pd=vcmpb.gtu(Rss,Rtt)`	`for (i = 0; i < 8; i++) {`
	`  Pd.i = (Rss.ub[i] > Rtt.ub[i]);`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

`Pd=vcmpb.eq(Rss,#u8)`	`Byte Q6_p_vcmpb_eq_PI(Word64 Rss, Word32 Iu8)`
`Pd=vcmpb.eq(Rss,Rtt)`	`Byte Q6_p_vcmpb_eq_PP(Word64 Rss, Word64 Rtt)`

```
Pd=vcmpb.gt(Rss,#s8)    Byte Q6_p_vcmpb_gt_PI(Word64 Rss, Word32 Is8)
Pd=vcmpb.gt(Rss,Rtt)    Byte Q6_p_vcmpb_gt_PP(Word64 Rss, Word64 Rtt)
Pd=vcmpb.gtu(Rss,#u7)   Byte Q6_p_vcmpb_gtu_PI(Word64 Rss, Word32 Iu7)
Pd=vcmpb.gtu(Rss,Rtt)   Byte Q6_p_vcmpb_gtu_PP(Word64 Rss, Word64 Rtt)
```

Encoding

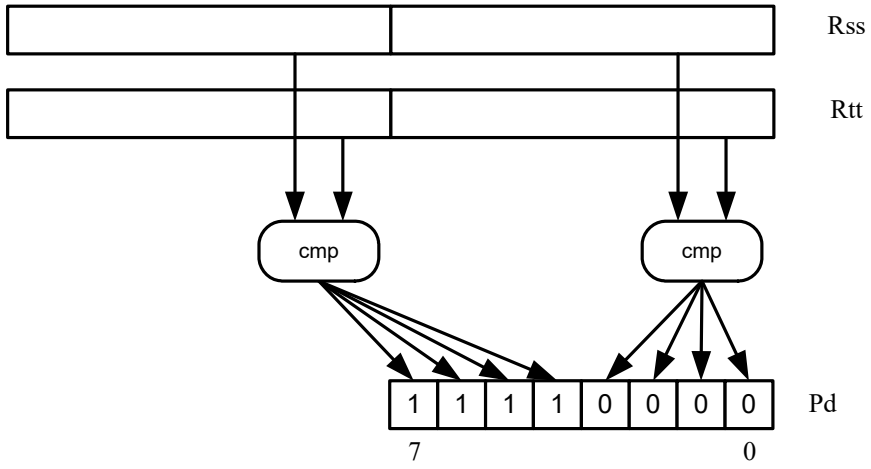
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp						d2		
1	1	0	1	0	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	-	-	-	**d**	**d**	Pd=vcmpb.eq(Rss,Rtt)
1	1	0	1	0	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	-	-	-	**d**	**d**	Pd=vcmpb.gtu(Rss,Rtt)
1	1	0	1	0	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	1	0	-	-	-	**d**	**d**	Pd=vcmpb.gt(Rss,Rtt)
ICLASS				RegType							s5					Parse														d2		
1	1	0	1	1	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	0	-	**d**	**d**	Pd=vcmpb.eq(Rss,#u8)
1	1	0	1	1	1	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	0	-	**d**	**d**	Pd=vcmpb.gt(Rss,#s8)
1	1	0	1	1	1	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	0	-	**d**	**d**	Pd=vcmpb.gtu(Rss,#u7)

Field name	Description
RegType	Register type
MinOp	Minor opcode
ICLASS	Instruction class
Parse	Packet/loop parse bits
d2	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector compare words

Compare each of two 32-bit words in two 64-bit vectors and set the corresponding bits in a predicate destination to '1111' if true, '0000' if false.

Word comparisons can be for equal, signed greater than, or unsigned greater than.



Syntax	Behavior
`Pd=vcmpw.eq(Rss,#s8)`	`Pd[3:0] = (Rss.w[0]==#s);` `Pd[7:4] = (Rss.w[1]==#s);`
`Pd=vcmpw.eq(Rss,Rtt)`	`Pd[3:0] = (Rss.w[0]==Rtt.w[0]);` `Pd[7:4] = (Rss.w[1]==Rtt.w[1]);`
`Pd=vcmpw.gt(Rss,#s8)`	`Pd[3:0] = (Rss.w[0]>#s);` `Pd[7:4] = (Rss.w[1]>#s);`
`Pd=vcmpw.gt(Rss,Rtt)`	`Pd[3:0] = (Rss.w[0]>Rtt.w[0]);` `Pd[7:4] = (Rss.w[1]>Rtt.w[1]);`
`Pd=vcmpw.gtu(Rss,#u7)`	`Pd[3:0] = (Rss.uw[0]>#u.uw[0]);` `Pd[7:4] = (Rss.uw[1]>#u.uw[0]);`
`Pd=vcmpw.gtu(Rss,Rtt)`	`Pd[3:0] = (Rss.uw[0]>Rtt.uw[0]);` `Pd[7:4] = (Rss.uw[1]>Rtt.uw[1]);`

Class: XTYPE (slots 2,3)

Intrinsics

`Pd=vcmpw.eq(Rss,#s8)`	`Byte Q6_p_vcmpw_eq_PI(Word64 Rss, Word32 Is8)`
`Pd=vcmpw.eq(Rss,Rtt)`	`Byte Q6_p_vcmpw_eq_PP(Word64 Rss, Word64 Rtt)`
`Pd=vcmpw.gt(Rss,#s8)`	`Byte Q6_p_vcmpw_gt_PI(Word64 Rss, Word32 Is8)`
`Pd=vcmpw.gt(Rss,Rtt)`	`Byte Q6_p_vcmpw_gt_PP(Word64 Rss, Word64 Rtt)`
`Pd=vcmpw.gtu(Rss,#u7)`	`Byte Q6_p_vcmpw_gtu_PI(Word64 Rss, Word32 Iu7)`
`Pd=vcmpw.gtu(Rss,Rtt)`	`Byte Q6_p_vcmpw_gtu_PP(Word64 Rss, Word64 Rtt)`

Encoding

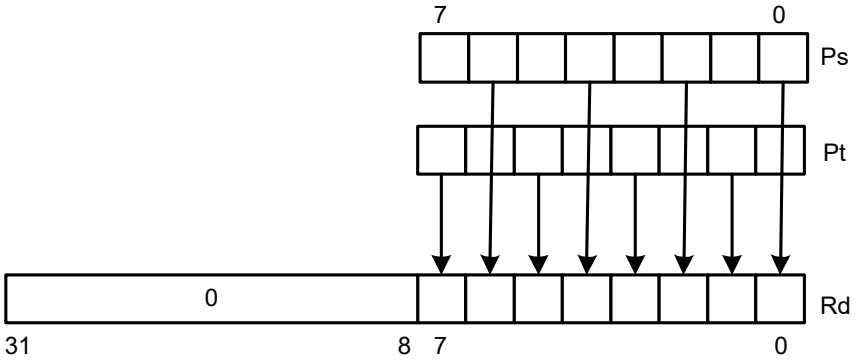
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5						MinOp					d2		
1	1	0	1	0	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	-	-	-	**d**	**d**	Pd=vcmpw.eq(Rss,Rtt)
1	1	0	1	0	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	-	-	-	**d**	**d**	Pd=vcmpw.gt(Rss,Rtt)
1	1	0	1	0	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	-	-	-	**d**	**d**	Pd=vcmpw.gtu(Rss,Rtt)
ICLASS				RegType							s5					Parse														d2		
1	1	0	1	1	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	1	0	-	**d**	**d**	Pd=vcmpw.eq(Rss,#s8)
1	1	0	1	1	1	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	1	0	-	**d**	**d**	Pd=vcmpw.gt(Rss,#s8)
1	1	0	1	1	1	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	1	0	-	**d**	**d**	Pd=vcmpw.gtu(Rss,#u7)

Field name	Description
RegType	Register type
MinOp	Minor opcode
ICLASS	Instruction class
Parse	Packet/loop parse bits
d2	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Viterbi pack even and odd predicate bits

Pack the even and odd bits of two predicate registers into a single destination register. A variant of this instruction is the R3:2 |= vitpack(P1,P0), which places the packed predicate bits into the lower 8 bits of the register pair that has been preshifted by 8 bits.

This instruction is useful in Viterbi decoding. Repeated use of the push version enables a history to be stored for traceback purposes.



Syntax	Behavior
`Rd = vitpack(Ps,Pt)`	`PREDUSE_TIMING;` `Rd = (Ps&0x55) | (Pt&0xAA);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd = vitpack(Ps, Pt)`	`Word32 Q6_R_vitpack_pp(Byte Ps, Byte Pt)`

Encoding

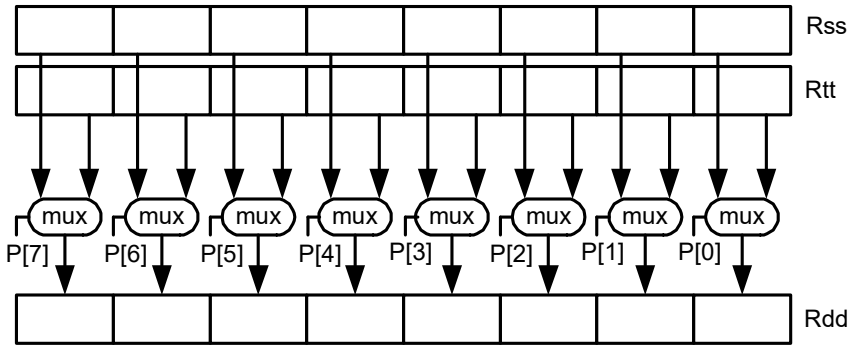
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp						s2		Parse						t2					d5					
1	0	0	0	1	0	0	1	-	0	0	-	-	-	**s**	**s**	**P**	**P**	-	-	-	-	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=vitpack(Ps,Pt)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s2	Field to encode register s
t2	Field to encode register t
MajOp	Major opcode
RegType	Register type

Vector mux

Perform an element-wise byte selection between two vectors.

For each of the low 8 bits of predicate register Pu, if the bit is set, the corresponding byte in Rdd is set to the corresponding byte from Rss. Otherwise, set the byte in Rdd to the byte from Rtt.



Syntax	Behavior

`Rdd=vmux(Pu,Rss,Rtt)`	`PREDUSE_TIMING;`
	`for (i = 0; i < 8; i++) {`
	`  Rdd.b[i]=(Pu.i?(Rss.b[i]):(Rtt.b[i]));`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vmux(Pu,Rss,Rtt)`	`Word64 Q6_P_vmux_pPP(Byte Pu, Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5						u2		d5						
1	1	0	1	0	0	0	1	-	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	-	u	u	d	d	d	d	d	Rdd=vmux(Pu,Rss,Rtt)

Field name	Description
RegType	Register type
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
u2	Field to encode register u

11.10.8 XTYPE SHIFT

The XTYPE SHIFT instruction subclass includes instructions that perform shifts.

Mask generate from immediate

Generates a mask from two immediate values.

Syntax	Behavior
`Rd = mask(#u5,#U5)`	`Rd = ((1<<#u)-1) << #U;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd = mask(#u5,#U5)`	`Word32 Q6_R_mask_II(Word32 Iu5, Word32 IU5)`

Encoding

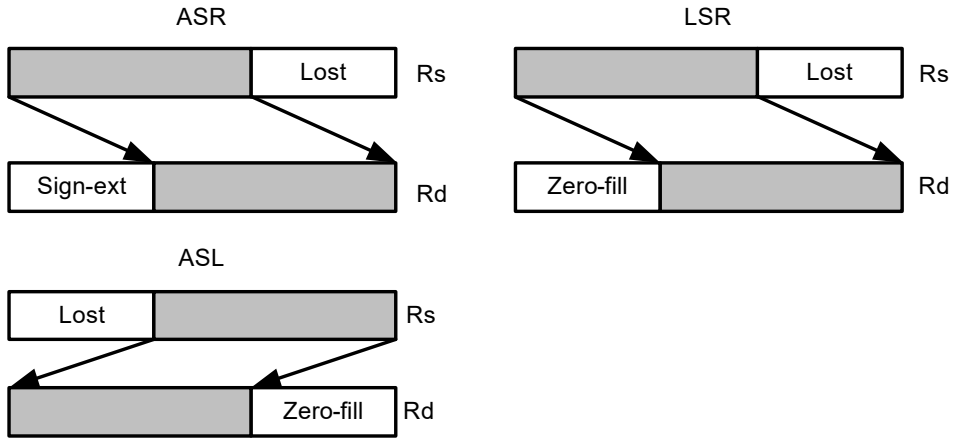
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp								Parse								MinOp			d5					
1	0	0	0	1	1	0	1	0	**I**	**I**	-	-	-	-	-	**P**	**P**	1	**i**	**i**	**i**	**i**	**i**	**I**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rd=mask(#u5,#U5)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

Shift by immediate

Shift the source register value right or left based on the type of instruction. In these instructions, the shift amount is contained in an unsigned immediate (5 bits for 32-bit shifts, 6 bits for 64-bit shifts) and the shift instruction gives the shift direction.

Arithmetic right shifts place the sign bit of the source value in the vacated positions, while logical right shifts place zeros in the vacated positions. Left shifts always zero-fill the vacated bits.



Syntax	Behavior
`Rd = asl(Rs,#u5)`	`Rd = Rs << #u;`
`Rd = asr(Rs,#u5)`	`Rd = Rs >> #u;`
`Rd = lsr(Rs,#u5)`	`Rd = Rs >>> #u;`
`Rd = rol(Rs,#u5)`	`Rd = Rs <<R #u;`
`Rdd = asl(Rss,#u6)`	`Rdd = Rss << #u;`
`Rdd = asr(Rss,#u6)`	`Rdd = Rss >> #u;`
`Rdd = lsr(Rss,#u6)`	`Rdd = Rss >>> #u;`
`Rdd = rol(Rss,#u6)`	`Rdd = Rss <<R #u;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd = asl(Rs,#u5)`	`Word32 Q6_R_asl_RI(Word32 Rs, Word32 Iu5)`
`Rd = asr(Rs,#u5)`	`Word32 Q6_R_asr_RI(Word32 Rs, Word32 Iu5)`
`Rd = lsr(Rs,#u5)`	`Word32 Q6_R_lsr_RI(Word32 Rs, Word32 Iu5)`
`Rd = rol(Rs,#u5)`	`Word32 Q6_R_rol_RI(Word32 Rs, Word32 Iu5)`
`Rdd = asl(Rss,#u6)`	`Word64 Q6_P_asl_PI(Word64 Rss, Word32 Iu6)`
`Rdd = asr(Rss,#u6)`	`Word64 Q6_P_asr_PI(Word64 Rss, Word32 Iu6)`
`Rdd = lsr(Rss,#u6)`	`Word64 Q6_P_lsr_PI(Word64 Rss, Word32 Iu6)`

```
Rdd = rol(Rss,#u6)          Word64 Q6_P_rol_PI(Word64 Rss, Word32 Iu6)
```

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=asr(Rss,#u6)
1	0	0	0	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=lsr(Rss,#u6)
1	0	0	0	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=asl(Rss,#u6)
1	0	0	0	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=rol(Rss,#u6)
1	0	0	0	1	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=asr(Rs,#u5)
1	0	0	0	1	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=lsr(Rs,#u5)
1	0	0	0	1	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=asl(Rs,#u5)
1	0	0	0	1	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rd=rol(Rs,#u5)

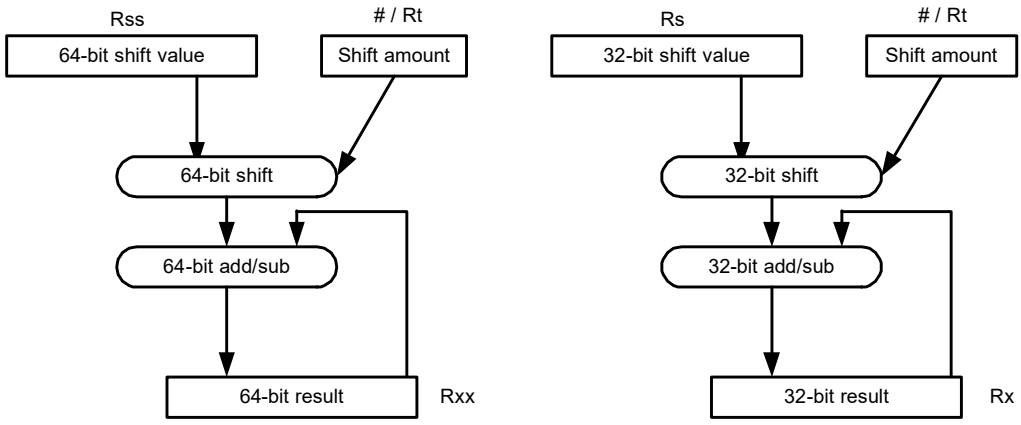
Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

Shift by immediate and accumulate

Shift the source register value right or left based on the type of instruction. In these instructions, the shift amount is contained in an unsigned immediate (5 bits for 32-bit shifts, 6 bits for 64-bit shifts) and the shift instruction gives the shift direction.

Arithmetic right shifts place the sign bit of the source value in the vacated positions, while logical right shifts place zeros in the vacated positions. Left shifts always zero-fill the vacated bits.

After shifting, add or subtract the shifted value from the destination register or register pair.



Syntax	Behavior
`Rx =add(#u8,asl(Rx,#U5))`	`Rx=apply_extension(#u)+(Rx<<#U);`
`Rx =add(#u8,lsr(Rx,#U5))`	`Rx=apply_extension(#u)+(((unsigned int)Rx)>>#U);`
`Rx =sub(#u8,asl(Rx,#U5))`	`Rx=apply_extension(#u)-(Rx<<#U);`
`Rx =sub(#u8,lsr(Rx,#U5))`	`Rx=apply_extension(#u)-(((unsigned int)Rx)>>#U);`
`Rx[+-] = asl(Rs,#u5)`	`Rx = Rx [+-] Rs << #u;`
`Rx[+-] = asr(Rs,#u5)`	`Rx = Rx [+-] Rs >> #u;`
`Rx[+-] = lsr(Rs,#u5)`	`Rx = Rx [+-] Rs >>> #u;`
`Rx[+-] = rol(Rs,#u5)`	`Rx = Rx [+-] Rs <<`$_R$ `#u;`
`Rxx[+-] = asl(Rss,#u6)`	`Rxx = Rxx [+-] Rss << #u;`
`Rxx[+-] = asr(Rss,#u6)`	`Rxx = Rxx [+-] Rss >> #u;`
`Rxx[+-] = lsr(Rss,#u6)`	`Rxx = Rxx [+-] Rss >>> #u;`
`Rxx[+-] = rol(Rss,#u6)`	`Rxx = Rxx [+-] Rss <<`$_R$ `#u;`

Class: XTYPE (slots 2, 3)

Intrinsics

Rx+=asl(Rs,#u5)	Word32 Q6_R_aslacc_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx+=asr(Rs,#u5)	Word32 Q6_R_asracc_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx+=lsr(Rs,#u5)	Word32 Q6_R_lsracc_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx+=rol(Rs,#u5)	Word32 Q6_R_rolacc_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx-=asl(Rs,#u5)	Word32 Q6_R_aslnac_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx-=asr(Rs,#u5)	Word32 Q6_R_asrnac_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx-=lsr(Rs,#u5)	Word32 Q6_R_lsrnac_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx-=rol(Rs,#u5)	Word32 Q6_R_rolnac_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx=add(#u8,asl(Rx,#U5))	Word32 Q6_R_add_asl_IRI(Word32 Iu8, Word32 Rx, Word32 IU5)
Rx=add(#u8,lsr(Rx,#U5))	Word32 Q6_R_add_lsr_IRI(Word32 Iu8, Word32 Rx, Word32 IU5)
Rx=sub(#u8,asl(Rx,#U5))	Word32 Q6_R_sub_asl_IRI(Word32 Iu8, Word32 Rx, Word32 IU5)
Rx=sub(#u8,lsr(Rx,#U5))	Word32 Q6_R_sub_lsr_IRI(Word32 Iu8, Word32 Rx, Word32 IU5)
Rxx+=asl(Rss,#u6)	Word64 Q6_P_aslacc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)
Rxx+=asr(Rss,#u6)	Word64 Q6_P_asracc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)
Rxx+=lsr(Rss,#u6)	Word64 Q6_P_lsracc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)
Rxx+=rol(Rss,#u6)	Word64 Q6_P_rolacc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)
Rxx-=asl(Rss,#u6)	Word64 Q6_P_aslnac_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)
Rxx-=asr(Rss,#u6)	Word64 Q6_P_asrnac_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)
Rxx-=lsr(Rss,#u6)	Word64 Q6_P_lsrnac_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)
Rxx-=rol(Rss,#u6)	Word64 Q6_P_rolnac_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			x5					
1	0	0	0	0	0	1	0	0	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	0	0	x	x	x	x	x	Rxx-=asr(Rss,#u6)
1	0	0	0	0	0	1	0	0	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	0	1	x	x	x	x	x	Rxx-=lsr(Rss,#u6)
1	0	0	0	0	0	1	0	0	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	1	0	x	x	x	x	x	Rxx-=asl(Rss,#u6)
1	0	0	0	0	0	1	0	0	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	1	1	x	x	x	x	x	Rxx-=rol(Rss,#u6)
1	0	0	0	0	0	1	0	0	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	1	0	0	x	x	x	x	x	Rxx+=asr(Rss,#u6)
1	0	0	0	0	0	1	0	0	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	1	0	1	x	x	x	x	x	Rxx+=lsr(Rss,#u6)
1	0	0	0	0	0	1	0	0	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	1	1	0	x	x	x	x	x	Rxx+=asl(Rss,#u6)
1	0	0	0	0	0	1	0	0	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	1	1	1	x	x	x	x	x	Rxx+=rol(Rss,#u6)
1	0	0	0	1	1	1	0	0	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	0	0	x	x	x	x	x	Rx-=asr(Rs,#u5)
1	0	0	0	1	1	1	0	0	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	0	1	x	x	x	x	x	Rx-=lsr(Rs,#u5)
1	0	0	0	1	1	1	0	0	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	1	0	x	x	x	x	x	Rx-=asl(Rs,#u5)
1	0	0	0	1	1	1	0	0	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	1	1	x	x	x	x	x	Rx-=rol(Rs,#u5)
1	0	0	0	1	1	1	0	0	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	1	0	0	x	x	x	x	x	Rx+=asr(Rs,#u5)
1	0	0	0	1	1	1	0	0	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	1	0	1	x	x	x	x	x	Rx+=lsr(Rs,#u5)
1	0	0	0	1	1	1	0	0	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	1	1	0	x	x	x	x	x	Rx+=asl(Rs,#u5)
1	0	0	0	1	1	1	0	0	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	1	1	1	x	x	x	x	x	Rx+=rol(Rs,#u5)
ICLASS				RegType							x5					Parse											MajOp					
1	1	0	1	1	1	1	0	i	i	i	x	x	x	x	x	P	P	i	I	I	I	I	I	i	i	i	0	i	1	0	-	Rx=add(#u8,asl(Rx,#U5))
1	1	0	1	1	1	1	0	i	i	i	x	x	x	x	x	P	P	i	I	I	I	I	I	i	i	i	0	i	1	1	-	Rx=sub(#u8,asl(Rx,#U5))
1	1	0	1	1	1	1	0	i	i	i	x	x	x	x	x	P	P	i	I	I	I	I	I	i	i	i	1	i	1	0	-	Rx=add(#u8,lsr(Rx,#U5))
1	1	0	1	1	1	1	0	i	i	i	x	x	x	x	x	P	P	i	I	I	I	I	I	i	i	i	1	i	1	1	-	Rx=sub(#u8,lsr(Rx,#U5))

Field name	Description
RegType	Register type
ICLASS	Instruction class
Parse	Packet/loop parse bits
s5	Field to encode register s
x5	Field to encode register x
MajOp	Major opcode
MinOp	Minor opcode

Shift by immediate and add

Shift Rs left by 0 to 7 bits, add to Rt, and place the result in Rd.

This instruction is useful for calculating array pointers, where destruction of the base pointer is undesirable.

Syntax	Behavior
`Rd = addasl(Rt,Rs,#u3)`	`Rd = Rt + Rs << #u;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=addasl(Rt,Rs,#u3)`	`Word32 Q6_R_addasl_RRI(Word32 Rt, Word32 Rs, Word32 Iu3)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=addasl(Rt,Rs,#u3)

Field name	Description
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`Maj`	Major opcode
`Min`	Minor opcode
`RegType`	Register type

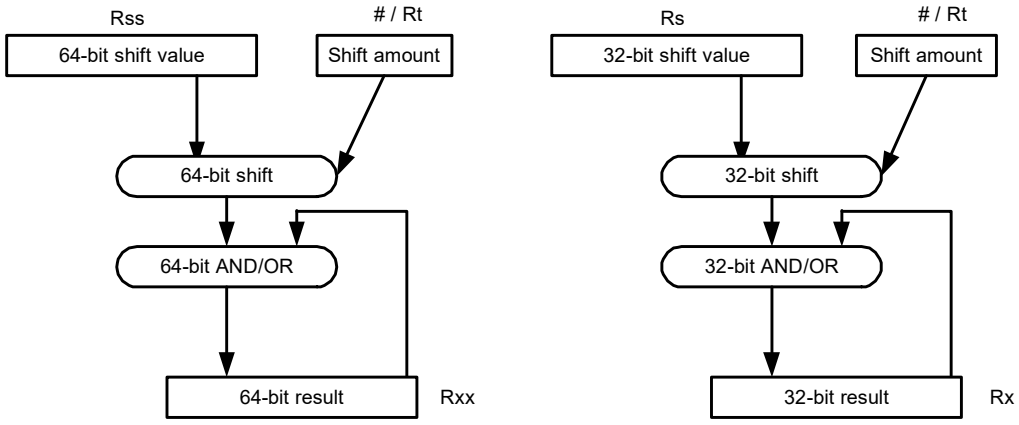
Shift by immediate and logical

Shift the source register value right or left based on the type of instruction. In these instructions, the shift amount is contained in an unsigned immediate (5 bits for 32-bit shifts, 6 bits for 64-bit shifts) and the shift instruction gives the shift direction.

Arithmetic right shifts place the sign bit of the source value in the vacated positions, while logical right shifts place zeros in the vacated positions. Left shifts always zero-fill the vacated bits.

After shifting, take the logical AND, OR, or XOR of the shifted amount and the destination register or register pair, and place the result back in the destination register or register pair.

Saturation is not available for these instructions.



Syntax	Behavior
`Rx=and(#u8,asl(Rx,#U5))`	`Rx =apply_extension(#u)&(Rx<<#U);`
`Rx=and(#u8,lsr(Rx,#U5))`	`Rx =apply_extension(#u)&(((unsigned int) Rx)>> #U);`
`Rx=or(#u8,asl(Rx,#U5))`	`Rx =apply_extension(#u)|(Rx<<#U);`
`Rx=or(#u8,lsr(Rx,#U5))`	`Rx=apply_extension(#u)|(((unsigned int) Rx)>> #U);`
`Rx[&|]=asl(Rs,#u5)`	`Rx = Rx [|&] Rs << #u;`
`Rx[&|]=asr(Rs,#u5)`	`Rx = Rx [|&] Rs >> #u;`
`Rx[&|]=lsr(Rs,#u5)`	`Rx = Rx [|&] Rs >>> #u;`
`Rx[&|]=rol(Rs,#u5)`	`Rx = Rx [|&] Rs <<R #u;`
`Rx^=asl(Rs,#u5)`	`Rx = Rx ^ Rs << #u;`
`Rx^=lsr(Rs,#u5)`	`Rx = Rx ^ Rs >>> #u;`
`Rx^=rol(Rs,#u5)`	`Rx = Rx ^ Rs <<R #u;`
`Rxx[&|]=asl(Rss,#u6)`	`Rxx = Rxx [|&] Rss << #u;`
`Rxx[&|]=asr(Rss,#u6)`	`Rxx = Rxx [|&] Rss >> #u;`

Syntax	Behavior
Rxx[&|]=lsr(Rss,#u6)	Rxx = Rxx [|&] Rss >>> #u;
Rxx[&|]=rol(Rss,#u6)	Rxx = Rxx [|&] Rss <<$_R$ #u;
Rxx^=asl(Rss,#u6)	Rxx = Rxx ^ Rss << #u;
Rxx^=lsr(Rss,#u6)	Rxx = Rxx ^ Rss >>> #u;
Rxx^=rol(Rss,#u6)	Rxx = Rxx ^ Rss <<$_R$ #u;

Class: XTYPE (slots 2,3)

Intrinsics

Rx&=asl(Rs,#u5)	Word32 Q6_R_asland_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx&=asr(Rs,#u5)	Word32 Q6_R_asrand_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx&=lsr(Rs,#u5)	Word32 Q6_R_lsrand_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx&=rol(Rs,#u5)	Word32 Q6_R_roland_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx=and(#u8,asl(Rx,#U5))	Word32 Q6_R_and_asl_IRI(Word32 Iu8, Word32 Rx, Word32 IU5)
Rx=and(#u8,lsr(Rx,#U5))	Word32 Q6_R_and_lsr_IRI(Word32 Iu8, Word32 Rx, Word32 IU5)
Rx=or(#u8,asl(Rx,#U5))	Word32 Q6_R_or_asl_IRI(Word32 Iu8, Word32 Rx, Word32 IU5)
Rx=or(#u8,lsr(Rx,#U5))	Word32 Q6_R_or_lsr_IRI(Word32 Iu8, Word32 Rx, Word32 IU5)
Rx^=asl(Rs,#u5)	Word32 Q6_R_aslxacc_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx^=lsr(Rs,#u5)	Word32 Q6_R_lsrxacc_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx^=rol(Rs,#u5)	Word32 Q6_R_rolxacc_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx|=asl(Rs,#u5)	Word32 Q6_R_aslor_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx|=asr(Rs,#u5)	Word32 Q6_R_asror_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx|=lsr(Rs,#u5)	Word32 Q6_R_lsror_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx|=rol(Rs,#u5)	Word32 Q6_R_rolor_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rxx&=asl(Rss,#u6)	Word64 Q6_P_asland_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)

`Rxx&=asr(Rss,#u6)`	`Word64 Q6_P_asrand_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)`
`Rxx&=lsr(Rss,#u6)`	`Word64 Q6_P_lsrand_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)`
`Rxx&=rol(Rss,#u6)`	`Word64 Q6_P_roland_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)`
`Rxx^=asl(Rss,#u6)`	`Word64 Q6_P_aslxacc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)`
`Rxx^=lsr(Rss,#u6)`	`Word64 Q6_P_lsrxacc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)`
`Rxx^=rol(Rss,#u6)`	`Word64 Q6_P_rolxacc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)`
`Rxx|=asl(Rss,#u6)`	`Word64 Q6_P_aslor_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)`
`Rxx|=asr(Rss,#u6)`	`Word64 Q6_P_asror_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)`
`Rxx|=lsr(Rss,#u6)`	`Word64 Q6_P_lsror_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)`
`Rxx|=rol(Rss,#u6)`	`Word64 Q6_P_rolor_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			x5					
1	0	0	0	0	0	1	0	0	1	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	0	0	x	x	x	x	x	Rxx&=asr(Rss,#u6)
1	0	0	0	0	0	1	0	0	1	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	0	1	x	x	x	x	x	Rxx&=lsr(Rss,#u6)
1	0	0	0	0	0	1	0	0	1	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	1	0	x	x	x	x	x	Rxx&=asl(Rss,#u6)
1	0	0	0	0	0	1	0	0	1	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	1	1	x	x	x	x	x	Rxx&=rol(Rss,#u6)
1	0	0	0	0	0	1	0	0	1	-	s	s	s	s	s	P	P	i	i	i	i	i	i	1	0	0	x	x	x	x	x	Rxx|=asr(Rss,#u6)
1	0	0	0	0	0	1	0	0	1	-	s	s	s	s	s	P	P	i	i	i	i	i	i	1	0	1	x	x	x	x	x	Rxx|=lsr(Rss,#u6)
1	0	0	0	0	0	1	0	0	1	-	s	s	s	s	s	P	P	i	i	i	i	i	i	1	1	0	x	x	x	x	x	Rxx|=asl(Rss,#u6)
1	0	0	0	0	0	1	0	0	1	-	s	s	s	s	s	P	P	i	i	i	i	i	i	1	1	1	x	x	x	x	x	Rxx|=rol(Rss,#u6)
1	0	0	0	0	0	1	0	1	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	0	1	x	x	x	x	x	Rxx^=lsr(Rss,#u6)
1	0	0	0	0	0	1	0	1	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	1	0	x	x	x	x	x	Rxx^=asl(Rss,#u6)
1	0	0	0	0	0	1	0	1	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	1	1	x	x	x	x	x	Rxx^=rol(Rss,#u6)
1	0	0	0	1	1	1	0	0	1	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	0	0	x	x	x	x	x	Rx&=asr(Rs,#u5)
1	0	0	0	1	1	1	0	0	1	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	0	1	x	x	x	x	x	Rx&=lsr(Rs,#u5)
1	0	0	0	1	1	1	0	0	1	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	1	0	x	x	x	x	x	Rx&=asl(Rs,#u5)
1	0	0	0	1	1	1	0	0	1	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	1	1	x	x	x	x	x	Rx&=rol(Rs,#u5)
1	0	0	0	1	1	1	0	0	1	-	s	s	s	s	s	P	P	0	i	i	i	i	i	1	0	0	x	x	x	x	x	Rx|=asr(Rs,#u5)
1	0	0	0	1	1	1	0	0	1	-	s	s	s	s	s	P	P	0	i	i	i	i	i	1	0	1	x	x	x	x	x	Rx|=lsr(Rs,#u5)
1	0	0	0	1	1	1	0	0	1	-	s	s	s	s	s	P	P	0	i	i	i	i	i	1	1	0	x	x	x	x	x	Rx|=asl(Rs,#u5)
1	0	0	0	1	1	1	0	0	1	-	s	s	s	s	s	P	P	0	i	i	i	i	i	1	1	1	x	x	x	x	x	Rx|=rol(Rs,#u5)
1	0	0	0	1	1	1	0	1	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	0	1	x	x	x	x	x	Rx^=lsr(Rs,#u5)
1	0	0	0	1	1	1	0	1	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	1	0	x	x	x	x	x	Rx^=asl(Rs,#u5)
1	0	0	0	1	1	1	0	1	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	1	1	x	x	x	x	x	Rx^=rol(Rs,#u5)

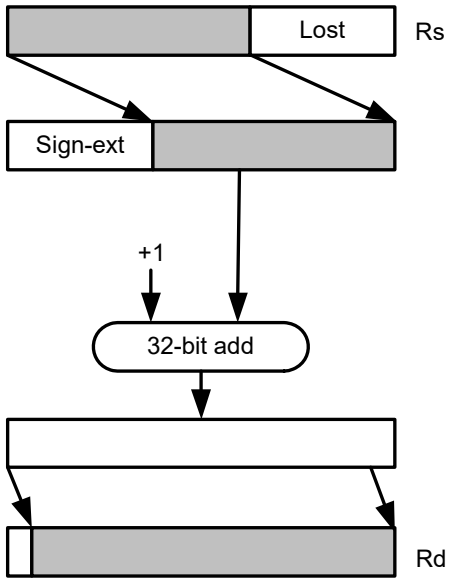
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							x5					Parse											MajOp					
1	1	0	1	1	1	1	0	**i**	**i**	**i**	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**i**	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	0	**i**	0	0	**-**	Rx=and(#u8,asl(Rx,#U5))
1	1	0	1	1	1	1	0	**i**	**i**	**i**	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**i**	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	0	**i**	0	1	**-**	Rx=or(#u8,asl(Rx,#U5))
1	1	0	1	1	1	1	0	**i**	**i**	**i**	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**i**	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	1	**i**	0	0	**-**	Rx=and(#u8,lsr(Rx,#U5))
1	1	0	1	1	1	1	0	**i**	**i**	**i**	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**i**	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	1	**i**	0	1	**-**	Rx=or(#u8,lsr(Rx,#U5))

Field name	Description
`RegType`	Register type
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`x5`	Field to encode register x
`MajOp`	Major opcode
`MinOp`	Minor opcode

Shift right by immediate with rounding

Perform an arithmetic right shift by an immediate amount, and then round the result. This instruction works by first shifting right, then adding the value +1 to the result, and finally shifting right again by one bit. The right shifts always inserts the sign-bit in the vacated position.

When using `asrrnd`, the assembler adjusts the immediate appropriately.



Syntax	Behavior
`Rd = asr(Rs,#u5):rnd`	`Rd = ((Rs >> #u)+1) >> 1;`

`Rd = asrrnd(Rs,#u5)`	`if ("#u5==0") {`
	`  Assembler mapped to: "Rd=Rs";`
	`} else {`
	`  Assembler mapped to: "Rd=asr(Rs,#u5-1):rnd";`
	`}`

`Rdd = asr(Rss,#u6):rnd`	`tmp = Rss >> #u;`
	`rnd = tmp & 1;`
	`Rdd = tmp >> 1 + rnd;`

`Rdd = asrrnd(Rss,#u6)`	`if ("#u6==0") {`
	`  Assembler mapped to: "Rdd=Rss";`
	`} else {`
	`  Assembler mapped to: "Rdd=asr(Rss, #u6 - 1):rnd";`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=asr(Rs,#u5):rnd`	`Word32 Q6_R_asr_RI_rnd(Word32 Rs, Word32 Iu5)`
`Rd=asrrnd(Rs,#u5)`	`Word32 Q6_R_asrrnd_RI(Word32 Rs, Word32 Iu5)`
`Rdd=asr(Rss,#u6):rnd`	`Word64 Q6_P_asr_PI_rnd(Word64 Rss, Word32 Iu6)`
`Rdd=asrrnd(Rss,#u6)`	`Word64 Q6_P_asrrnd_PI(Word64 Rss, Word32 Iu6)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=asr(Rss,#u6):rnd
1	0	0	0	1	1	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=asr(Rs,#u5):rnd

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

Shift left by immediate with saturation

Perform a left shift of the 32-bit source register value by an immediate amount and saturate.

Saturation works by first sign-extending the 32-bit Rs register to 64 bits. It is then left shifted by the immediate amount. If this 64-bit value cannot fit in a signed 32-bit number (the upper word is not the sign-extension of bit 31), saturation is performed based on the sign of the original value. Saturation clamps the 32-bit result to the range 0x8000_0000 to 0x7fff_ffff.

Syntax	Behavior
`Rd=asl(Rs,#u5):sat`	$Rd = sat_{32}(sxt_{32->64}(Rs) << \#u);$

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=asl(Rs,#u5):sat`	`Word32 Q6_R_asl_RI_sat(Word32 Rs, Word32 Iu5)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	1	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=asl(Rs,#u5):sat

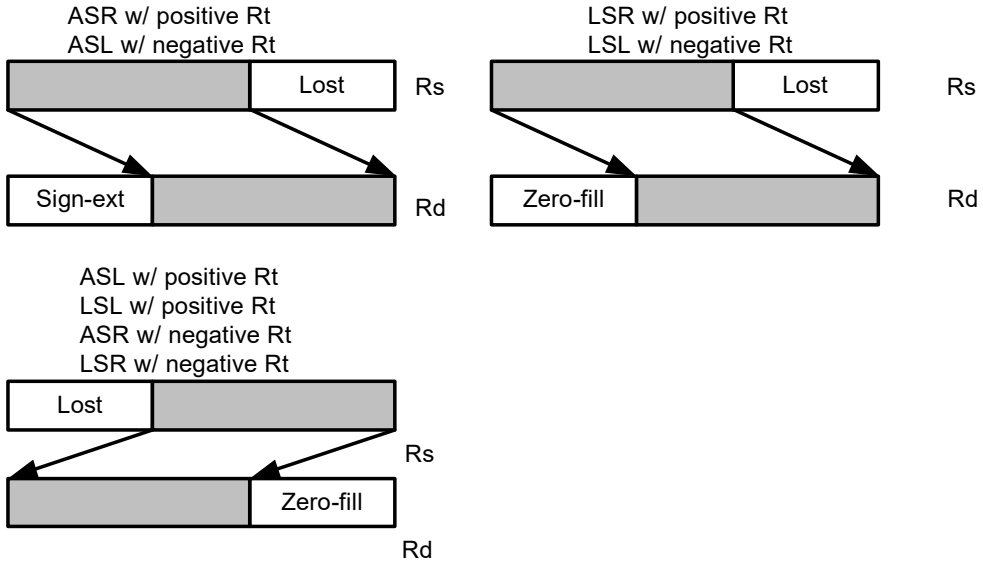
Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

Shift by register

The shift amount is the least significant seven bits of Rt, treated as a two's complement value. If the shift amount is negative (bit 6 of Rt is set), the direction of the shift indicated in the opcode is reversed (see Figure).

The source data to be shifted is always performed as a 64-bit shift. When the Rs source register is a 32-bit register, this register is first sign or zero-extended to 64-bits. Arithmetic shifts sign-extend the 32-bit source to 64-bits, while logical shifts zero extend.

The 64-bit source value is then right or left shifted based on the shift amount and the type of instruction. Arithmetic right shifts place the sign bit of the source value in the vacated positions. Logical right shifts place zeros in the vacated positions.



Syntax	Behavior
`Rd=asl(Rs,Rt)`	`shamt=sxt7->32(Rt); Rd = (shamt>0)?(sxt32->64(Rs)<<shamt):(sxt32->64(Rs)>>shamt);`
`Rd=asr(Rs,Rt)`	`shamt=sxt7->32(Rt); Rd = (shamt>0)?(sxt32->64(Rs)>>shamt):(sxt32->64(Rs)<<shamt);`
`Rd=lsl(#s6,Rt)`	`shamt = sxt7->32(Rt); Rd = (shamt>0)?(zxt32->64(#s)<<shamt):(zxt32->64(#s)>>>shamt);`
`Rd=lsl(Rs,Rt)`	`shamt=sxt7->32(Rt); Rd = (shamt>0)?(zxt32->64(Rs)<<shamt):(zxt32->64(Rs)>>>shamt);`
`Rd=lsr(Rs,Rt)`	`shamt=sxt7->32(Rt); Rd = (shamt>0)?(zxt32->64(Rs)>>>shamt):(zxt32->64(Rs)<<shamt);`

Syntax	Behavior
Rdd=asl(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rdd = (shamt>0)?(Rss<<shamt):(Rss>>shamt);
Rdd=asr(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rdd = (shamt>0)?(Rss>>shamt):(Rss<<shamt);
Rdd=lsl(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rdd = (shamt>0)?(Rss<<shamt):(Rss>>>shamt);
Rdd=lsr(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rdd = (shamt>0)?(Rss>>>shamt):(Rss<<shamt);

Class: XTYPE (slots 2,3)

Intrinsics

Rd=asl(Rs,Rt)	Word32 Q6_R_asl_RR(Word32 Rs, Word32 Rt)
Rd=asr(Rs,Rt)	Word32 Q6_R_asr_RR(Word32 Rs, Word32 Rt)
Rd=lsl(#s6,Rt)	Word32 Q6_R_lsl_IR(Word32 Is6, Word32 Rt)
Rd=lsl(Rs,Rt)	Word32 Q6_R_lsl_RR(Word32 Rs, Word32 Rt)
Rd=lsr(Rs,Rt)	Word32 Q6_R_lsr_RR(Word32 Rs, Word32 Rt)
Rdd=asl(Rss,Rt)	Word64 Q6_P_asl_PR(Word64 Rss, Word32 Rt)
Rdd=asr(Rss,Rt)	Word64 Q6_P_asr_PR(Word64 Rss, Word32 Rt)
Rdd=lsl(Rss,Rt)	Word64 Q6_P_lsl_PR(Word64 Rss, Word32 Rt)
Rdd=lsr(Rss,Rt)	Word64 Q6_P_lsr_PR(Word64 Rss, Word32 Rt)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	1	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=asr(Rss,Rt)
1	1	0	0	0	0	1	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=lsr(Rss,Rt)
1	1	0	0	0	0	1	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=asl(Rss,Rt)
1	1	0	0	0	0	1	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=lsl(Rss,Rt)
1	1	0	0	0	1	1	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rd=asr(Rs,Rt)
1	1	0	0	0	1	1	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rd=lsr(Rs,Rt)
1	1	0	0	0	1	1	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rd=asl(Rs,Rt)
1	1	0	0	0	1	1	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**d**	**d**	**d**	**d**	**d**	Rd=lsl(Rs,Rt)
ICLASS				RegType				Maj								Parse			t5					Min			d5					
1	1	0	0	0	1	1	0	1	0	-	**i**	**i**	**i**	**i**	**i**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	**i**	**d**	**d**	**d**	**d**	**d**	Rd=lsl(#s6,Rt)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d

Field name	Description
s5	Field to encode register s
t5	Field to encode register t
Maj	Major opcode
Min	Minor opcode
RegType	Register type

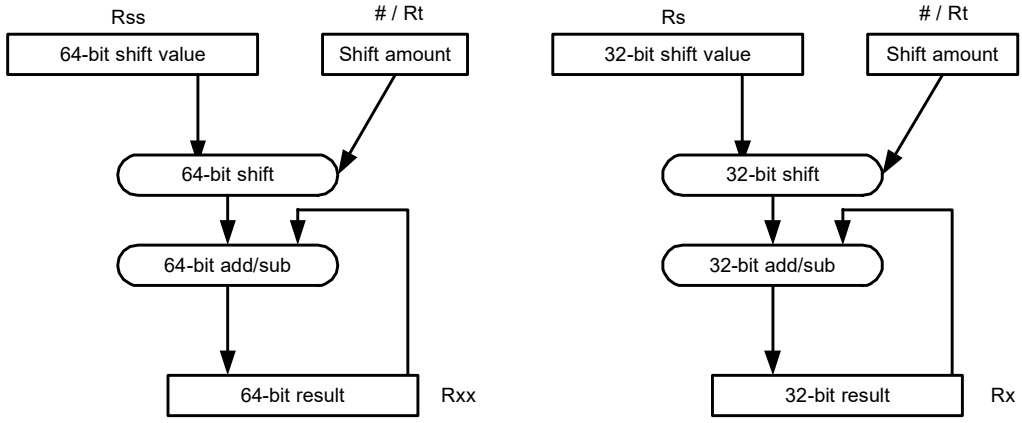
Shift by register and accumulate

The shift amount is the least significant 7 bits of Rt, treated as a two's complement value. If the shift amount is negative (bit 6 of Rt is set), the direction of the shift indicated in the opcode is reversed.

Shift the source register value right or left based on the shift amount and the type of instruction. Arithmetic right shifts place the sign bit of the source value in the vacated positions. Logical right shifts place zeros in the vacated positions.

The shift operation is always performed as a 64-bit shift. When Rs is a 32-bit register, this register is first sign- or zero-extended to 64-bits. Arithmetic shifts sign-extend the 32-bit source to 64-bits, while logical shifts zero extend.

After shifting, add or subtract the 64-bit shifted amount from the destination register or register pair.



Syntax	Behavior
`Rx[+-]=asl(Rs,Rt)`	`shamt=sxt7->32(Rt);` `Rx = Rx [+-] (shamt>0)?(sxt32->64(Rs)<<shamt):(sxt32->64(Rs)>>shamt);`
`Rx[+-]=asr(Rs,Rt)`	`shamt=sxt7->32(Rt);` `Rx = Rx [+-] (shamt>0)?(sxt32->64(Rs)>>shamt):(sxt32->64(Rs)<<shamt);`
`Rx[+-]=lsl(Rs,Rt)`	`shamt=sxt7->32(Rt);` `Rx = Rx [+-] (shamt>0)?(zxt32->64(Rs)<<shamt):(zxt32->64(Rs)>>>shamt);`
`Rx[+-]=lsr(Rs,Rt)`	`shamt=sxt7->32(Rt);` `Rx = Rx [+-] (shamt>0)?(zxt32->64(Rs) >>> shamt):(zxt32->64(Rs) << shamt);`
`Rxx[+-]=asl(Rss,Rt)`	`shamt=sxt7->32(Rt);` `Rxx = Rxx [+-] (shamt>0)?(Rss<<shamt):(Rss>>shamt);`
`Rxx[+-]=asr(Rss,Rt)`	`shamt=sxt7->32(Rt);` `Rxx = Rxx [+-] (shamt>0)?(Rss>>shamt):(Rss<<shamt);`

Syntax	Behavior
`Rxx[+-]=lsl(Rss,Rt)`	`shamt=sxt7->32(Rt); Rxx = Rxx [+-] (shamt > 0)?(Rss << shamt):(Rss >>> shamt);`
`Rxx[+-]=lsr(Rss,Rt)`	`shamt=sxt7->32(Rt); Rxx = Rxx [+-] (shamt>0)?(Rss >>> shamt):(Rss << shamt);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rx+=asl(Rs,Rt)`	`Word32 Q6_R_aslacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx+=asr(Rs,Rt)`	`Word32 Q6_R_asracc_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx+=lsl(Rs,Rt)`	`Word32 Q6_R_lslacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx+=lsr(Rs,Rt)`	`Word32 Q6_R_lsracc_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx-=asl(Rs,Rt)`	`Word32 Q6_R_aslnac_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx-=asr(Rs,Rt)`	`Word32 Q6_R_asrnac_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx-=lsl(Rs,Rt)`	`Word32 Q6_R_lslnac_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx-=lsr(Rs,Rt)`	`Word32 Q6_R_lsrnac_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rxx+=asl(Rss,Rt)`	`Word64 Q6_P_aslacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx+=asr(Rss,Rt)`	`Word64 Q6_P_asracc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx+=lsl(Rss,Rt)`	`Word64 Q6_P_lslacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx+=lsr(Rss,Rt)`	`Word64 Q6_P_lsracc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx-=asl(Rss,Rt)`	`Word64 Q6_P_aslnac_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx-=asr(Rss,Rt)`	`Word64 Q6_P_asrnac_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx-=lsl(Rss,Rt)`	`Word64 Q6_P_lslnac_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx-=lsr(Rss,Rt)`	`Word64 Q6_P_lsrnac_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			x5					
1	1	0	0	1	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**x**	**x**	**x**	**x**	**x**	Rxx-=asr(Rss,Rt)
1	1	0	0	1	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**x**	**x**	**x**	**x**	**x**	Rxx-=lsr(Rss,Rt)
1	1	0	0	1	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**x**	**x**	**x**	**x**	**x**	Rxx-=asl(Rss,Rt)
1	1	0	0	1	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**x**	**x**	**x**	**x**	**x**	Rxx-=lsl(Rss,Rt)
1	1	0	0	1	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**x**	**x**	**x**	**x**	**x**	Rxx+=asr(Rss,Rt)
1	1	0	0	1	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**x**	**x**	**x**	**x**	**x**	Rxx+=lsr(Rss,Rt)
1	1	0	0	1	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**x**	**x**	**x**	**x**	**x**	Rxx+=asl(Rss,Rt)
1	1	0	0	1	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**x**	**x**	**x**	**x**	**x**	Rxx+=lsl(Rss,Rt)
1	1	0	0	1	1	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**x**	**x**	**x**	**x**	**x**	Rx-=asr(Rs,Rt)
1	1	0	0	1	1	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**x**	**x**	**x**	**x**	**x**	Rx-=lsr(Rs,Rt)
1	1	0	0	1	1	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**x**	**x**	**x**	**x**	**x**	Rx-=asl(Rs,Rt)
1	1	0	0	1	1	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**x**	**x**	**x**	**x**	**x**	Rx-=lsl(Rs,Rt)
1	1	0	0	1	1	0	0	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**x**	**x**	**x**	**x**	**x**	Rx+=asr(Rs,Rt)
1	1	0	0	1	1	0	0	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**x**	**x**	**x**	**x**	**x**	Rx+=lsr(Rs,Rt)
1	1	0	0	1	1	0	0	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**x**	**x**	**x**	**x**	**x**	Rx+=asl(Rs,Rt)
1	1	0	0	1	1	0	0	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**x**	**x**	**x**	**x**	**x**	Rx+=lsl(Rs,Rt)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x
Maj	Major opcode
Min	Minor opcode
RegType	Register type

Shift by register and logical

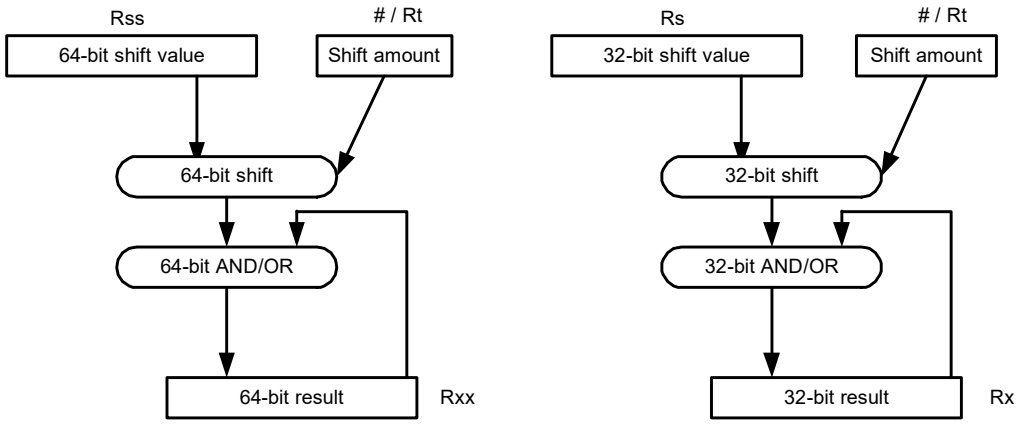
The shift amount is the least significant 7 bits of Rt, treated as a two's complement value. If the shift amount is negative (bit 6 of Rt is set), the direction of the shift indicated in the opcode is reversed.

Shift the source register value right or left based on the shift amount and the type of instruction. Arithmetic right shifts place the sign bit of the source value in the vacated positions. Logical right shifts place zeros in the vacated positions.

The shift operation is always performed as a 64-bit shift. When the Rs source register is a 32-bit register, this register is first sign or zero-extended to 64-bits. Arithmetic shifts sign-extend the 32-bit source to 64-bits, while logical shifts zero extend.

After shifting, take the logical AND or OR of the shifted amount and the destination register or register pair, and place the result back in the destination register or register pair.

Saturation is not available for these instructions.



Syntax	Behavior
`Rx[&|]=asl(Rs,Rt)`	`shamt=sxt7->32(Rt); Rx = Rx [|&] (shamt>0)?(sxt32->64(Rs) << shamt):(sxt32->64(Rs)>>shamt);`
`Rx[&|]=asr(Rs,Rt)`	`shamt=sxt7->32(Rt); Rx = Rx [|&] (shamt>0)?(sxt32->64(Rs) >> shamt):(sxt32->64(Rs)<<shamt);`
`Rx[&|]=lsl(Rs,Rt)`	`shamt=sxt7->32(Rt); Rx = Rx [|&] (shamt>0)?(zxt32->64(Rs) << shamt):(zxt32->64(Rs)>>>shamt);`
`Rx[&|]=lsr(Rs,Rt)`	`shamt=sxt7->32(Rt); Rx = Rx [|&] (shamt>0)?(zxt32->64(Rs) >>> shamt):(zxt32->64(Rs)<<shamt);`
`Rxx[&|]=asl(Rss,Rt)`	`shamt=sxt7->32(Rt); Rxx = Rxx [|&] (shamt>0)?(Rss << shamt):(Rss >> shamt);`

Syntax	Behavior
Rxx[&|]=asr(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rxx = Rxx [|&] (shamt > 0)?(Rss >> shamt):(Rss << shamt);
Rxx[&|]=lsl(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rxx = Rxx [|&] (shamt > 0)?(Rss << shamt):(Rss >>> shamt);
Rxx[&|]=lsr(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rxx = Rxx [|&] (shamt > 0)?(Rss >>> shamt):(Rss << shamt);
Rxx^=asl(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rxx = Rxx ^ (shamt>0)?(Rss<<shamt):(Rss>>shamt);
Rxx^=asr(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rxx = Rxx ^ (shamt>0)?(Rss>>shamt):(Rss<<shamt);
Rxx^=lsl(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rxx = Rxx ^ (shamt > 0)?(Rss << shamt):(Rss >>> shamt);
Rxx^=lsr(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rxx = Rxx ^ (shamt > 0)?(Rss >>> shamt):(Rss << shamt);

Class: XTYPE (slots 2,3)

Intrinsics

Rx&=asl(Rs,Rt)	Word32 Q6_R_asland_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx&=asr(Rs,Rt)	Word32 Q6_R_asrand_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx&=lsl(Rs,Rt)	Word32 Q6_R_lsland_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx&=lsr(Rs,Rt)	Word32 Q6_R_lsrand_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx|=asl(Rs,Rt)	Word32 Q6_R_aslor_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx|=asr(Rs,Rt)	Word32 Q6_R_asror_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx|=lsl(Rs,Rt)	Word32 Q6_R_lslor_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx|=lsr(Rs,Rt)	Word32 Q6_R_lsror_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rxx&=asl(Rss,Rt)	Word64 Q6_P_asland_PR(Word64 Rxx, Word64 Rss, Word32 Rt)
Rxx&=asr(Rss,Rt)	Word64 Q6_P_asrand_PR(Word64 Rxx, Word64 Rss, Word32 Rt)
Rxx&=lsl(Rss,Rt)	Word64 Q6_P_lsland_PR(Word64 Rxx, Word64 Rss, Word32 Rt)
Rxx&=lsr(Rss,Rt)	Word64 Q6_P_lsrand_PR(Word64 Rxx, Word64 Rss, Word32 Rt)

`Rxx^=asl(Rss,Rt)`	`Word64 Q6_P_aslxacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx^=asr(Rss,Rt)`	`Word64 Q6_P_asrxacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx^=lsl(Rss,Rt)`	`Word64 Q6_P_lslxacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx^=lsr(Rss,Rt)`	`Word64 Q6_P_lsrxacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx|=asl(Rss,Rt)`	`Word64 Q6_P_aslor_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx|=asr(Rss,Rt)`	`Word64 Q6_P_asror_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx|=lsl(Rss,Rt)`	`Word64 Q6_P_lslor_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx|=lsr(Rss,Rt)`	`Word64 Q6_P_lsror_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			x5					
1	1	0	0	1	0	1	1	0	0	0	s	s	s	s	s	P	P	-	t	t	t	t	t	0	0	-	x	x	x	x	x	Rxx|=asr(Rss,Rt)
1	1	0	0	1	0	1	1	0	0	0	s	s	s	s	s	P	P	-	t	t	t	t	t	0	1	-	x	x	x	x	x	Rxx|=lsr(Rss,Rt)
1	1	0	0	1	0	1	1	0	0	0	s	s	s	s	s	P	P	-	t	t	t	t	t	1	0	-	x	x	x	x	x	Rxx|=asl(Rss,Rt)
1	1	0	0	1	0	1	1	0	0	0	s	s	s	s	s	P	P	-	t	t	t	t	t	1	1	-	x	x	x	x	x	Rxx|=lsl(Rss,Rt)
1	1	0	0	1	0	1	1	0	1	0	s	s	s	s	s	P	P	-	t	t	t	t	t	0	0	-	x	x	x	x	x	Rxx&=asr(Rss,Rt)
1	1	0	0	1	0	1	1	0	1	0	s	s	s	s	s	P	P	-	t	t	t	t	t	0	1	-	x	x	x	x	x	Rxx&=lsr(Rss,Rt)
1	1	0	0	1	0	1	1	0	1	0	s	s	s	s	s	P	P	-	t	t	t	t	t	1	0	-	x	x	x	x	x	Rxx&=asl(Rss,Rt)
1	1	0	0	1	0	1	1	0	1	0	s	s	s	s	s	P	P	-	t	t	t	t	t	1	1	-	x	x	x	x	x	Rxx&=lsl(Rss,Rt)
1	1	0	0	1	0	1	1	0	1	1	s	s	s	s	s	P	P	-	t	t	t	t	t	0	0	-	x	x	x	x	x	Rxx^=asr(Rss,Rt)
1	1	0	0	1	0	1	1	0	1	1	s	s	s	s	s	P	P	-	t	t	t	t	t	0	1	-	x	x	x	x	x	Rxx^=lsr(Rss,Rt)
1	1	0	0	1	0	1	1	0	1	1	s	s	s	s	s	P	P	-	t	t	t	t	t	1	0	-	x	x	x	x	x	Rxx^=asl(Rss,Rt)
1	1	0	0	1	0	1	1	0	1	1	s	s	s	s	s	P	P	-	t	t	t	t	t	1	1	-	x	x	x	x	x	Rxx^=lsl(Rss,Rt)
1	1	0	0	1	1	0	0	0	0	-	s	s	s	s	s	P	P	-	t	t	t	t	t	0	0	-	x	x	x	x	x	Rx|=asr(Rs,Rt)
1	1	0	0	1	1	0	0	0	0	-	s	s	s	s	s	P	P	-	t	t	t	t	t	0	1	-	x	x	x	x	x	Rx|=lsr(Rs,Rt)
1	1	0	0	1	1	0	0	0	0	-	s	s	s	s	s	P	P	-	t	t	t	t	t	1	0	-	x	x	x	x	x	Rx|=asl(Rs,Rt)
1	1	0	0	1	1	0	0	0	0	-	s	s	s	s	s	P	P	-	t	t	t	t	t	1	1	-	x	x	x	x	x	Rx|=lsl(Rs,Rt)
1	1	0	0	1	1	0	0	0	1	-	s	s	s	s	s	P	P	-	t	t	t	t	t	0	0	-	x	x	x	x	x	Rx&=asr(Rs,Rt)
1	1	0	0	1	1	0	0	0	1	-	s	s	s	s	s	P	P	-	t	t	t	t	t	0	1	-	x	x	x	x	x	Rx&=lsr(Rs,Rt)
1	1	0	0	1	1	0	0	0	1	-	s	s	s	s	s	P	P	-	t	t	t	t	t	1	0	-	x	x	x	x	x	Rx&=asl(Rs,Rt)
1	1	0	0	1	1	0	0	0	1	-	s	s	s	s	s	P	P	-	t	t	t	t	t	1	1	-	x	x	x	x	x	Rx&=lsl(Rs,Rt)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
s5	Field to encode register s
t5	Field to encode register t

Field name	Description
x5	Field to encode register x
Maj	Major opcode
Min	Minor opcode
RegType	Register type

Shift by register with saturation

The shift amount is the least significant seven bits of Rt, treated as a two's complement value. If the shift amount is negative (bit 6 of Rt is set), the direction of the shift indicated in the opcode is reversed.

Saturation is available for 32-bit arithmetic left shifts. This can be either an ASL instruction with positive Rt, or an ASR instruction with negative Rt. Saturation works by first sign-extending the 32-bit Rs register to 64 bits. It is then shifted by the shift amount. If this 64-bit value cannot fit in a signed 32-bit number (the upper word is not the sign-extension of bit 31), saturation is performed based on the sign of the original value. Saturation clamps the 32-bit result to the range 0x80000000 to 0x7fffffff.

Syntax	Behavior
`Rd=asl(Rs,Rt):sat`	`shamt=sxt`$_{7->32}$`(Rt);` `Rd = bidir_shiftl(Rs,shamt);`
`Rd=asr(Rs,Rt):sat`	`shamt=sxt`$_{7->32}$`(Rt);` `Rd = bidir_shiftr(Rs,shamt);`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=asl(Rs,Rt):sat`	`Word32 Q6_R_asl_RR_sat(Word32 Rs, Word32 Rt)`
`Rd=asr(Rs,Rt):sat`	`Word32 Q6_R_asr_RR_sat(Word32 Rs, Word32 Rt)`

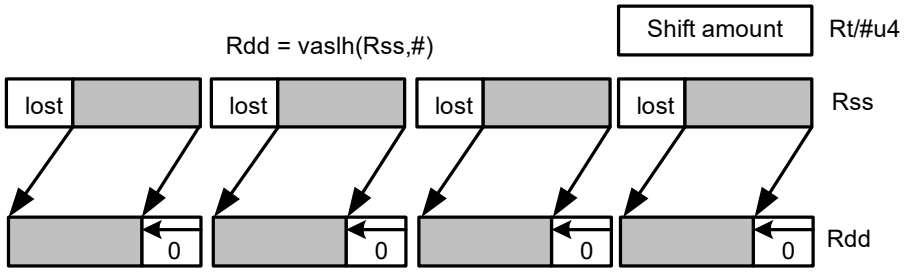
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	1	1	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rd=asr(Rs,Rt):sat
1	1	0	0	0	1	1	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rd=asl(Rs,Rt):sat

Field name	Description
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`Maj`	Major opcode
`Min`	Minor opcode
`RegType`	Register type

Vector shift halfwords by immediate

Shift individual halfwords of the source vector. Arithmetic right shifts place the sign bit of the source values in the vacated positions. Logical right shifts place zeros in the vacated positions.



Syntax	Behavior

`Rdd=vaslh(Rss,#u4)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=(Rss.h[i]<<#u);`
	`}`

`Rdd=vasrh(Rss,#u4)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=(Rss.h[i]>>#u);`
	`}`

`Rdd=vlsrh(Rss,#u4)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=(Rss.uh[i]>>#u);`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vaslh(Rss,#u4)`	`Word64 Q6_P_vaslh_PI(Word64 Rss, Word32 Iu4)`
`Rdd=vasrh(Rss,#u4)`	`Word64 Q6_P_vasrh_PI(Word64 Rss, Word32 Iu4)`
`Rdd=vlsrh(Rss,#u4)`	`Word64 Q6_P_vlsrh_PI(Word64 Rss, Word32 Iu4)`

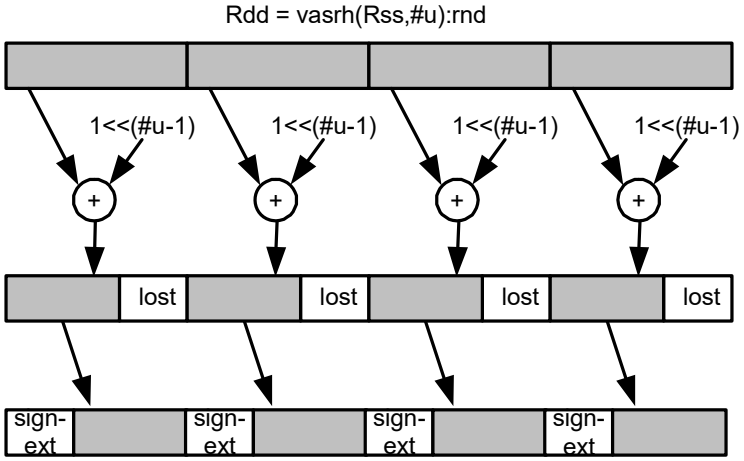
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	**i**	**i**	**i**	**i**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vasrh(Rss,#u4)
1	0	0	0	0	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	**i**	**i**	**i**	**i**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vlsrh(Rss,#u4)
1	0	0	0	0	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	**i**	**i**	**i**	**i**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vaslh(Rss,#u4)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

Vector arithmetic shift halfwords with round

For each halfword in the vector, round then arithmetic shift right by an immediate amount. The results are stored in the destination register.



Syntax	Behavior

`Rdd=vasrh(Rss, #u4):raw`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=( ((Rss.h[i] >> #u)+1)>>1 );`
	`}`

`Rdd=vasrh(Rss, #u4):rnd`	`if ("#u4==0") {`
	`  Assembler mapped to: "Rdd=Rss";`
	`} else {`
	`  Assembler mapped to: "Rdd=vasrh(Rss,#u4 -1):raw";`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vasrh(Rss,#u4):rnd`	`Word64 Q6_P_vasrh_PI_rnd(Word64 Rss, Word32 Iu4)`

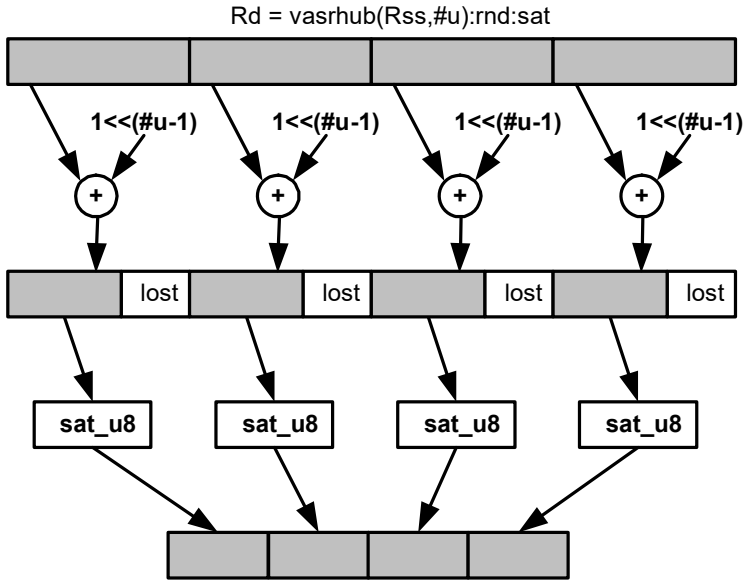
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse							MinOp			d5					
1	0	0	0	0	0	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	**i**	**i**	**i**	**i**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vasrh(Rss,#u4):raw

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

Vector arithmetic shift halfwords with saturate and pack

For each halfword in the vector, optionally round, then arithmetic shift right by an immediate amount. The results are saturated to unsigned [0-255] and then packed in the destination register.



Syntax	Behavior

`Rd=vasrhub(Rss,#u4):raw`	`for (i=0;i<4;i++) {`
	`  Rd.b[i]=usat8(((Rss.h[i] >> #u`
	`)+1)>>1);`
	`}`

`Rd=vasrhub(Rss,#u4):rnd:sat`	`if ("#u4==0") {`
	`  Assembler mapped to: "Rd=vsathub(Rss)";`
	`} else {`
	`  Assembler mapped to: "Rd = vasrhub(Rss,`
	`#u4-1):raw";`
	`}`

`Rd=vasrhub(Rss,#u4):sat`	`for (i = 0; i < 4; i++) {`
	`  Rd.b[i] = usat8(Rss.h[i] >> #u);`
	`}`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), the OVF bit in the status register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=vasrhub(Rss,#u4):rnd:sat`	`Word32 Q6_R_vasrhub_PI_rnd_sat(Word64 Rss, Word32 Iu4)`
`Rd=vasrhub(Rss,#u4):sat`	`Word32 Q6_R_vasrhub_PI_sat(Word64 Rss, Word32 Iu4)`

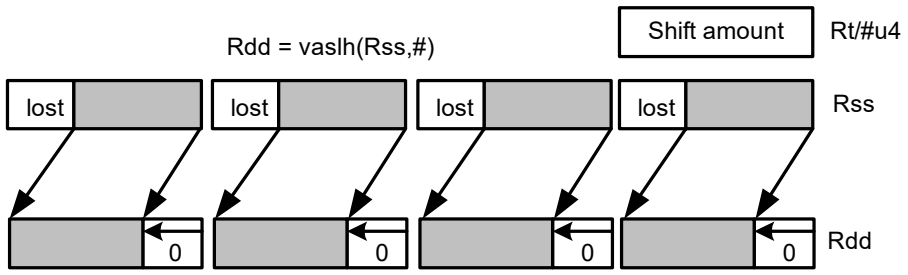
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	0	0	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	**i**	**i**	**i**	**i**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=vasrhub(Rss,#u4):raw
1	0	0	0	1	0	0	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	**i**	**i**	**i**	**i**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rd=vasrhub(Rss,#u4):sat

Field name	Description
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major opcode
`MinOp`	Minor opcode
`RegType`	Register type

Vector shift halfwords by register

The shift amount is the least significant 7 bits of Rt, treated as a two's complement value. If the shift amount is negative, the direction of the shift is reversed. Shift the source values right or left based on the shift amount and the type of instruction. Arithmetic right shifts place the sign bit of the source value in the vacated positions. Logical right shifts place zeros in the vacated positions.



Syntax	Behavior
`Rdd=vaslh(Rss,Rt)`	`for (i=0;i<4;i++) { Rdd.h[i]=(sxt7->32(Rt)>0)?(sxt16->64(Rss.h[i]) <<sxt7->32(Rt)):(sxt16->64(Rss.h[i])>>sxt7->32(Rt)); }`
`Rdd=vasrh(Rss,Rt)`	`for (i=0;i<4;i++) { Rdd.h[i]=(sxt7->32(Rt)>0)?(sxt16->64(Rss.h[i]) >>sxt7->32(Rt)):(sxt16->64(Rss.h[i])<<sxt7->32(Rt)); }`
`Rdd=vlslh(Rss,Rt)`	`for (i=0;i<4;i++) { Rdd.h[i]=(sxt7->32(Rt)>0)?(zxt16->64(Rss.uh[i]) <<sxt7->32(Rt)):(zxt16->64(Rss.uh[i])>>>sxt7->32(Rt)); }`
`Rdd=vlsrh(Rss,Rt)`	`for (i=0;i<4;i++) { Rdd.h[i]=(sxt7->32(Rt)>0)?(zxt16->64(Rss.uh[i]) >>>sxt7->32(Rt)):(zxt16->64(Rss.uh[i])<<sxt7->32(Rt)); }`

Class: XTYPE (slots 2,3)

Notes

- If the number of bits to be shifted is greater than the width of the vector element, the result is either all sign-bits (for arithmetic right shifts) or all zeros for logical and left shifts.

Intrinsics

`Rdd=vaslh(Rss,Rt)`	`Word64 Q6_P_vaslh_PR(Word64 Rss, Word32 Rt)`
`Rdd=vasrh(Rss,Rt)`	`Word64 Q6_P_vasrh_PR(Word64 Rss, Word32 Rt)`
`Rdd=vlslh(Rss,Rt)`	`Word64 Q6_P_vlslh_PR(Word64 Rss, Word32 Rt)`
`Rdd=vlsrh(Rss,Rt)`	`Word64 Q6_P_vlsrh_PR(Word64 Rss, Word32 Rt)`

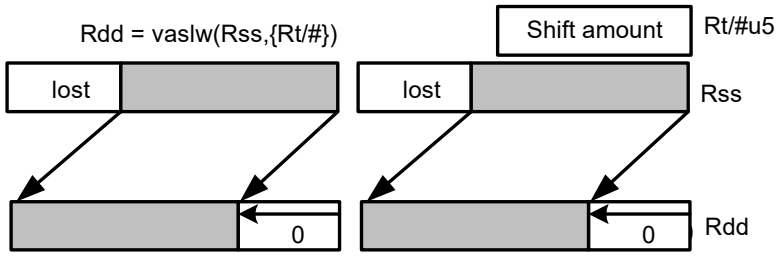
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	1	1	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=vasrh(Rss,Rt)
1	1	0	0	0	0	1	1	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vlsrh(Rss,Rt)
1	1	0	0	0	0	1	1	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=vaslh(Rss,Rt)
1	1	0	0	0	0	1	1	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vlslh(Rss,Rt)

Field name	Description
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`Maj`	Major opcode
`Min`	Minor opcode
`RegType`	Register type

Vector shift words by immediate

Shift individual words of the source vector. Arithmetic right shifts place the sign bit of the source values in the vacated positions. Logical right shifts place zeros in the vacated positions.



Syntax	Behavior

`Rdd=vaslw(Rss,#u5)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=(Rss.w[i]<<#u);`
	`}`

`Rdd=vasrw(Rss,#u5)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=(Rss.w[i]>>#u);`
	`}`

`Rdd=vlsrw(Rss,#u5)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=(Rss.uw[i]>>#u);`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vaslw(Rss,#u5)`	`Word64 Q6_P_vaslw_PI(Word64 Rss, Word32 Iu5)`
`Rdd=vasrw(Rss,#u5)`	`Word64 Q6_P_vasrw_PI(Word64 Rss, Word32 Iu5)`
`Rdd=vlsrw(Rss,#u5)`	`Word64 Q6_P_vlsrw_PI(Word64 Rss, Word32 Iu5)`

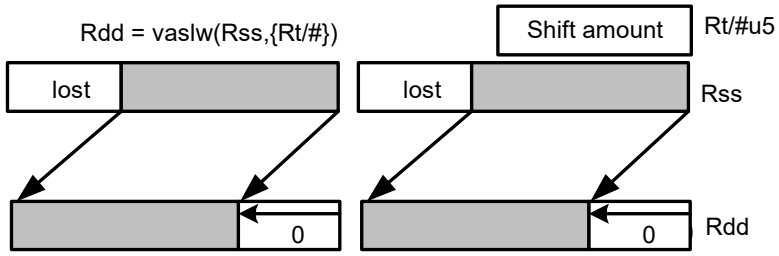
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vasrw(Rss,#u5)
1	0	0	0	0	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vlsrw(Rss,#u5)
1	0	0	0	0	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vaslw(Rss,#u5)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

Vector shift words by register

The shift amount is the least significant seven bits of Rt, treated as a two's complement value. If the shift amount is negative, the direction of the shift is reversed. Shift the source values right or left based on the shift amount and the type of instruction. Arithmetic right shifts place the sign bit of the source value in the vacated positions. Logical right shifts place zeros in the vacated positions.



Syntax	Behavior
`Rdd=vaslw(Rss,Rt)`	`for (i=0;i<2;i++) { Rdd.w[i]=(sxt7->32(Rt)>0)?(sxt32->64(Rss.w[i]) <<sxt7->32(Rt)):(sxt32->64(Rss.w[i])>>sxt7->32(Rt)); }`
`Rdd=vasrw(Rss,Rt)`	`for (i=0;i<2;i++) { Rdd.w[i]=(sxt7->32(Rt)>0)?(sxt32->64(Rss.w[i]) >>sxt7->32(Rt)):(sxt32->64(Rss.w[i])<<sxt7->32(Rt)); }`
`Rdd=vlslw(Rss,Rt)`	`for (i=0;i<2;i++) { Rdd.w[i]=(sxt7->32(Rt)>0)?(zxt32->64(Rss.uw[i]) <<sxt7->32(Rt)):(zxt32->64(Rss.uw[i])>>>sxt7->32(Rt)); }`
`Rdd=vlsrw(Rss,Rt)`	`for (i=0;i<2;i++) { Rdd.w[i]=(sxt7->32(Rt)>0)?(zxt32->64(Rss.uw[i]) >>>sxt7->32(Rt)):(zxt32->64(Rss.uw[i])<< sxt7->32(Rt)); }`

Class: XTYPE (slots 2,3)

Notes

- If the number of bits to be shifted is greater than the width of the vector element, the result is either all sign-bits (for arithmetic right shifts) or all zeros for logical and left shifts.

Intrinsics

`Rdd=vaslw(Rss,Rt)`	`Word64 Q6_P_vaslw_PR(Word64 Rss, Word32 Rt)`
`Rdd=vasrw(Rss,Rt)`	`Word64 Q6_P_vasrw_PR(Word64 Rss, Word32 Rt)`
`Rdd=vlslw(Rss,Rt)`	`Word64 Q6_P_vlslw_PR(Word64 Rss, Word32 Rt)`
`Rdd=vlsrw(Rss,Rt)`	`Word64 Q6_P_vlsrw_PR(Word64 Rss, Word32 Rt)`

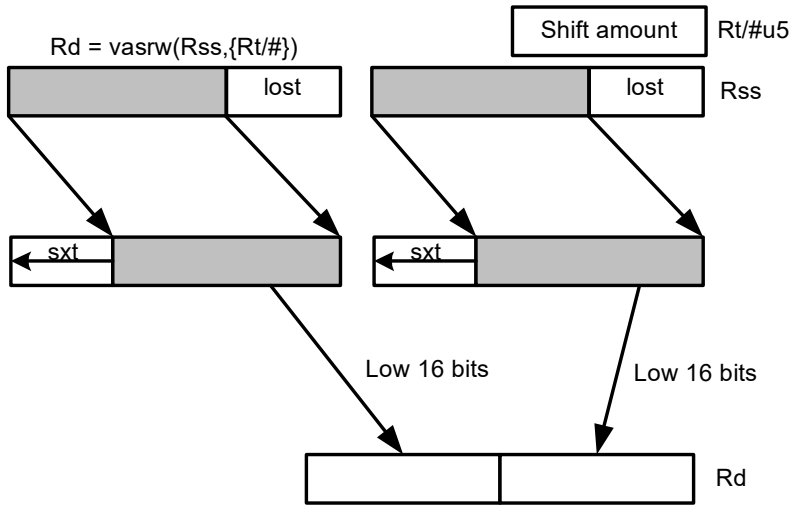
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	1	1	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=vasrw(Rss,Rt)
1	1	0	0	0	0	1	1	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vlsrw(Rss,Rt)
1	1	0	0	0	0	1	1	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=vaslw(Rss,Rt)
1	1	0	0	0	0	1	1	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vlslw(Rss,Rt)

Field name	Description
`ICLASS`	Instruction class
`Parse`	Packet/loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`Maj`	Major opcode
`Min`	Minor opcode
`RegType`	Register type

Vector shift words with truncate and pack

Shift individual words of the source vector Rss right by a register or immediate amount. The low 16-bits of each word are packed into destination register Rd.



Syntax	Behavior

`Rd=vasrw(Rss,#u5)`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=(Rss.w[i]>>#u).h[0];`
	`}`

`Rd=vasrw(Rss,Rt)`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=(sxt`$_{7->32}$`(Rt)>0)?(sxt`$_{32->64}$`(Rss.w[i]) >>sxt`$_{7->32}$`(Rt)):(sxt`$_{32->64}$`(Rss.w[i])<< sxt`$_{7->32}$`(Rt)).h[0];`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=vasrw(Rss,#u5)`	`Word32 Q6_R_vasrw_PI(Word64 Rss, Word32 Iu5)`
`Rd=vasrw(Rss,Rt)`	`Word32 Q6_R_vasrw_PR(Word64 Rss, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=vasrw(Rss,#u5)
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	0	0	1	0	1	-	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=vasrw(Rss,Rt)

Field name	Description
ICLASS	Instruction class
Parse	Packet/loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
MajOp	Major opcode
MinOp	Minor opcode
RegType	Register type

Instruction Index

A

abs
    Rd=abs(Rs)[:sat] 326
    Rdd=abs(Rss) 325

add
    if ([!]Pu[.new]) Rd=add(Rs,#s8) 178
    if ([!]Pu[.new]) Rd=add(Rs,Rt) 178
    Rd=add(#u6,mpyi(Rs,#U6)) 465
    Rd=add(#u6,mpyi(Rs,Rt)) 465
    Rd=add(Rs,#s16) 157
    Rd=add(Rs,add(Ru,#s6)) 327
    Rd=add(Rs,Rt) 157
    Rd=add(Rs,Rt):sat 157
    Rd=add(Rs,Rt):sat:deprecated 329
    Rd=add(Rt.[HL],Rs.[HL])[:sat]:<<16 331
    Rd=add(Rt.L,Rs.[HL])[:sat] 331
    Rd=add(Ru,mpyi(#u6:2,Rs)) 465
    Rd=add(Ru,mpyi(Rs,#u6)) 465
    Rdd=add(Rs,Rtt) 329
    Rdd=add(Rss,Rtt,Px):carry 333
    Rdd=add(Rss,Rtt) 329
    Rdd=add(Rss,Rtt):raw:hi 329
    Rdd=add(Rss,Rtt):raw:lo 329
    Rdd=add(Rss,Rtt):sat 329
    Rx+=add(Rs,#s8) 327
    Rx+=add(Rs,Rt) 327
    Rx-=add(Rs,#s8) 327
    Rx-=add(Rs,Rt) 327
    Ry=add(Ru,mpyi(Ry,Rs)) 466

addasl
    Rd=addasl(Rt,Rs,#u3) 562

all8
    Pd=all8(Ps) 195

allocframe
    allocframe(#u11:3) 303
    allocframe(Rx,#u11:3):raw 303

and
    if ([!]Pu[.new]) Rd=and(Rs,Rt) 183
    Pd=and(Ps,and(Pt,[!]Pu)) 201
    Pd=and(Pt,[!]Ps) 201
    Rd=and(Rs,#s10) 159
    Rd=and(Rs,Rt) 159
    Rd=and(Rt,~Rs) 159
    Rdd=and(Rss,Rtt) 335
    Rdd=and(Rtt,~Rss) 335
    Rx[&|^]=and(Rs,~Rt) 337
    Rx[&|^]=and(Rs,Rt) 337
    Rx|=and(Rs,#s10) 337

any8
    Pd=any8(Ps) 195

asl
Rd=asl(Rs,#u5) 557
Rd=asl(Rs,#u5):sat 569
Rd=asl(Rs,Rt) 570
Rd=asl(Rs,Rt):sat 580
Rdd=asl(Rss,#u6) 557
Rdd=asl(Rss,Rt) 571
Rx[&|]=asl(Rs,#u5) 563
Rx[&|]=asl(Rs,Rt) 576
Rx[+-]=asl(Rs,#u5) 559
Rx[+-]=asl(Rs,Rt) 573
Rx^=asl(Rs,#u5) 563
Rx=add(#u8,asl(Rx,#U5)) 559
Rx=and(#u8,asl(Rx,#U5)) 563
Rx=or(#u8,asl(Rx,#U5)) 563
Rx=sub(#u8,asl(Rx,#U5)) 559
Rxx[&|]=asl(Rss,#u6) 563
Rxx[&|]=asl(Rss,Rt) 576
Rxx[+-]=asl(Rss,#u6) 559
Rxx[+-]=asl(Rss,Rt) 573
Rxx^=asl(Rss,#u6) 564
Rxx^=asl(Rss,Rt) 577

aslh
if ([!]Pu[.new]) Rd=aslh(Rs) 180
Rd=aslh(Rs) 176

asr
Rd=asr(Rs,#u5) 557
Rd=asr(Rs,#u5):rnd 567
Rd=asr(Rs,Rt) 570
Rd=asr(Rs,Rt):sat 580
Rdd=asr(Rss,#u6) 557
Rdd=asr(Rss,#u6):rnd 567
Rdd=asr(Rss,Rt) 571
Rx[&|]=asr(Rs,#u5) 563
Rx[&|]=asr(Rs,Rt) 576
Rx[+-]=asr(Rs,#u5) 559
Rx[+-]=asr(Rs,Rt) 573
Rxx[&|]=asr(Rss,#u6) 563
Rxx[&|]=asr(Rss,Rt) 577
Rxx[+-]=asr(Rss,#u6) 559
Rxx[+-]=asr(Rss,Rt) 573
Rxx^=asr(Rss,Rt) 577

asrh
if ([!]Pu[.new]) Rd=asrh(Rs) 180
Rd=asrh(Rs) 176

asrrnd
Rd=asrrnd(Rs,#u5) 567
Rdd=asrrnd(Rss,#u6) 567

B

barrier
barrier 308, 308

bitsclr
Pd=[!]bitsclr(Rs,#u6) 542
Pd=[!]bitsclr(Rs,Rt) 542

bitsplit
Rdd=bitsplit(Rs,#u5) 410
Rdd=bitsplit(Rs,Rt) 410

bitsset
Pd=[!]bitsset(Rs,Rt) 542

boundscheck
Pd=boundscheck(Rs,Rtt) 537
Pd=boundscheck(Rss,Rtt):raw:hi 537
Pd=boundscheck(Rss,Rtt):raw:lo 537

brev
Rd=brev(Rs) 407
Rdd=brev(Rss) 407

brkpt
brkpt 309, 309

C

call
call #r22:2 209
if ([!]Pu) call #r15:2 209

callr
callr Rs 204
if ([!]Pu) callr Rs 204

callrh
callrh Rs 204, 205

cl0
Rd=cl0(Rs) 395
Rd=cl0(Rss) 395

cl1
Rd=cl1(Rs) 395
Rd=cl1(Rss) 395

clb
Rd=add(clb(Rs),#s6) 395
Rd=add(clb(Rss),#s6) 395
Rd=clb(Rs) 395
Rd=clb(Rss) 395

clip
Rd=clip(Rs,#u5) 334

clrbit
memb(Rs+#u6:0)=clrbit(#U5) 263
memh(Rs+#u6:1)=clrbit(#U5) 264
memw(Rs+#u6:2)=clrbit(#U5) 265
Rd=clrbit(Rs,#u5) 408
Rd=clrbit(Rs,Rt) 408

cmp.eq
if ([!]cmp.eq(Ns.new,#-1)) jump:<hint> #r9:2 266
if ([!]cmp.eq(Ns.new,#U5)) jump:<hint> #r9:2 266
if ([!]cmp.eq(Ns.new,Rt)) jump:<hint> #r9:2 266
p[01]=cmp.eq(Rs,#-1) 211
p[01]=cmp.eq(Rs,#U5) 211
p[01]=cmp.eq(Rs,Rt) 211
Pd=[!]cmp.eq(Rs,#s10) 190
Pd=[!]cmp.eq(Rs,Rt) 190
Pd=cmp.eq(Rss,Rtt) 541
Rd=[!]cmp.eq(Rs,#s8) 192
Rd=[!]cmp.eq(Rs,Rt) 192

cmp.ge
Pd=cmp.ge(Rs,#s8) 190

cmp.geu
Pd=cmp.geu(Rs,#u8) 190

cmp.gt
if ([!]cmp.gt(Ns.new,#-1)) jump:<hint> #r9:2 266
if ([!]cmp.gt(Ns.new,#U5)) jump:<hint> #r9:2 266
if ([!]cmp.gt(Ns.new,Rt)) jump:<hint> #r9:2 267
if ([!]cmp.gt(Rt,Ns.new)) jump:<hint> #r9:2 267
p[01]=cmp.gt(Rs,#-1) 211
p[01]=cmp.gt(Rs,#U5) 211
p[01]=cmp.gt(Rs,Rt) 211
Pd=[!]cmp.gt(Rs,#s10) 190
Pd=[!]cmp.gt(Rs,Rt) 190
Pd=cmp.gt(Rss,Rtt) 541

cmp.gtu
if ([!]cmp.gtu(Ns.new,#U5)) jump:<hint> #r9:2 267
if ([!]cmp.gtu(Ns.new,Rt)) jump:<hint> #r9:2 267
if ([!]cmp.gtu(Rt,Ns.new)) jump:<hint> #r9:2 267
p[01]=cmp.gtu(Rs,#U5) 212
p[01]=cmp.gtu(Rs,Rt) 212
Pd=[!]cmp.gtu(Rs,#u9) 190
Pd=[!]cmp.gtu(Rs,Rt) 190
Pd=cmp.gtu(Rss,Rtt) 541

cmp.lt
Pd=cmp.lt(Rs,Rt) 190

cmp.ltu
Pd=cmp.ltu(Rs,Rt) 190

cmpb.eq
Pd=cmpb.eq(Rs,#u8) 538
Pd=cmpb.eq(Rs,Rt) 538

cmpb.gt
Pd=cmpb.gt(Rs,#s8) 538
Pd=cmpb.gt(Rs,Rt) 538

cmpb.gtu
Pd=cmpb.gtu(Rs,#u7) 538
Pd=cmpb.gtu(Rs,Rt) 538

cmph.eq
Pd=cmph.eq(Rs,#s8) 539
Pd=cmph.eq(Rs,Rt) 539

cmph.gt
Pd=cmph.gt(Rs,#s8) 539
Pd=cmph.gt(Rs,Rt) 539

cmph.gtu
Pd=cmph.gtu(Rs,#u7) 539
Pd=cmph.gtu(Rs,Rt) 539

cmpy
Rd=cmpy(Rs,Rt)[:<<1]:rnd:sat 423
Rd=cmpy(Rs,Rt*)[:<<1]:rnd:sat 423
Rdd=cmpy(Rs,Rt)[:<<1]:sat 418
Rdd=cmpy(Rs,Rt*)[:<<1]:sat 418
Rxx+=cmpy(Rs,Rt)[:<<1]:sat 419
Rxx+=cmpy(Rs,Rt*)[:<<1]:sat 419
Rxx-=cmpy(Rs,Rt)[:<<1]:sat 419
Rxx-=cmpy(Rs,Rt*)[:<<1]:sat 419

cmpyi
Rdd=cmpyi(Rs,Rt) 421
Rxx+=cmpyi(Rs,Rt) 421

cmpyiw
Rd=cmpyiw(Rss,Rtt):<<1:rnd:sat 427
Rd=cmpyiw(Rss,Rtt):<<1:sat 427
Rd=cmpyiw(Rss,Rtt*):<<1:rnd:sat 427
Rd=cmpyiw(Rss,Rtt*):<<1:sat 427
Rdd=cmpyiw(Rss,Rtt) 428
Rdd=cmpyiw(Rss,Rtt*) 428
Rxx+=cmpyiw(Rss,Rtt) 428
Rxx+=cmpyiw(Rss,Rtt*) 428

cmpyiwh
Rd=cmpyiwh(Rss,Rt):<<1:rnd:sat 425
Rd=cmpyiwh(Rss,Rt*):<<1:rnd:sat 425

cmpyr
Rdd=cmpyr(Rs,Rt) 421
Rxx+=cmpyr(Rs,Rt) 421

cmpyrw
Rd=cmpyrw(Rss,Rtt):<<1:rnd:sat 427
Rd=cmpyrw(Rss,Rtt):<<1:sat 427
Rd=cmpyrw(Rss,Rtt*):<<1:rnd:sat 428
Rd=cmpyrw(Rss,Rtt*):<<1:sat 428
Rdd=cmpyrw(Rss,Rtt) 428
Rdd=cmpyrw(Rss,Rtt*) 428
Rxx+=cmpyrw(Rss,Rtt) 428
Rxx+=cmpyrw(Rss,Rtt*) 428

cmpyrwh
Rd=cmpyrwh(Rss,Rt):<<1:rnd:sat 425
Rd=cmpyrwh(Rss,Rt*):<<1:rnd:sat 425

combine
if ([!]Pu[.new]) Rdd=combine(Rs,Rt) 182
Rd=combine(Rt.[HL],Rs.[HL]) 172
Rdd=combine(#s8,#S8) 172
Rdd=combine(#s8,#U6) 172
Rdd=combine(#s8,Rs) 172
Rdd=combine(Rs,#s8) 172
Rdd=combine(Rs,Rt) 172

convert_d2df
Rdd=convert_d2df(Rss) 449

convert_d2sf
Rd=convert_d2sf(Rss) 449

convert_df2d
Rdd=convert_df2d(Rss) 451
Rdd=convert_df2d(Rss):chop 451

convert_df2sf
Rd=convert_df2sf(Rss) 448

convert_df2ud
Rdd=convert_df2ud(Rss) 451
Rdd=convert_df2ud(Rss):chop 451

convert_df2uw
Rd=convert_df2uw(Rss) 451
Rd=convert_df2uw(Rss):chop 451

convert_df2w
Rd=convert_df2w(Rss) 451
Rd=convert_df2w(Rss):chop 451

convert_sf2d
Rdd=convert_sf2d(Rs) 451
Rdd=convert_sf2d(Rs):chop 451

convert_sf2df
Rdd=convert_sf2df(Rs) 448

convert_sf2ud
Rdd=convert_sf2ud(Rs) 451
Rdd=convert_sf2ud(Rs):chop 451

convert_sf2uw
Rd=convert_sf2uw(Rs) 451
Rd=convert_sf2uw(Rs):chop 451

convert_sf2w
Rd=convert_sf2w(Rs) 451
Rd=convert_sf2w(Rs):chop 451

convert_ud2df
Rdd=convert_ud2df(Rss) 449

convert_ud2sf
Rd=convert_ud2sf(Rss) 449

convert_uw2df
Rdd=convert_uw2df(Rs) 449

convert_uw2sf
Rd=convert_uw2sf(Rs) 449

convert_w2df
Rdd=convert_w2df(Rs) 449

convert_w2sf
Rd=convert_w2sf(Rs) 449

cround
Rd=cround(Rs,#u5) 345
Rd=cround(Rs,Rt) 345
Rdd=cround(Rss,#u6) 345
Rdd=cround(Rss,Rt) 346

ct0
Rd=ct0(Rs) 398
Rd=ct0(Rss) 398

ct1
Rd=ct1(Rs) 398
Rd=ct1(Rss) 398

D

dccleana
dccleana(Rs) 312

dccleaninva
dccleaninva(Rs) 312

dcfetch
dcfetch(Rs) 310
dcfetch(Rs+#u11:3) 310

dcinva
dcinva(Rs) 312

dczeroa
dczeroa(Rs) 307

dealloc_return
dealloc_return 253
if ([!]Pv.new) Rdd=dealloc_return(Rs):nt:raw 253
if ([!]Pv.new) Rdd=dealloc_return(Rs):t:raw 253
if ([!]Pv) dealloc_return 253
if ([!]Pv) Rdd=dealloc_return(Rs):raw 253
nt
if ([!]Pv.new) dealloc_return:nt 253
Rdd=dealloc_return(Rs):raw 253
t
if ([!]Pv.new) dealloc_return:t 253

deallocframe
deallocframe 251
Rdd=deallocframe(Rs):raw 251

decbin
Rdd=decbin(Rss,Rtt) 516

deinterleave
Rdd=deinterleave(Rss) 404

dfadd
Rdd=dfadd(Rss,Rtt) 444

dfclass
Pd=dfclass(Rss,#u5) 445

dfcmp.eq
Pd=dfcmp.eq(Rss,Rtt) 446

dfcmp.ge
Pd=dfcmp.ge(Rss,Rtt) 446

dfcmp.gt
Pd=dfcmp.gt(Rss,Rtt) 446

dfcmp.uo
Pd=dfcmp.uo(Rss,Rtt) 446

dfmake
Rdd=dfmake(#u10):neg 459
Rdd=dfmake(#u10):pos 459

dfmax
Rdd=dfmax(Rss,Rtt) 460

dfmin
Rdd=dfmin(Rss,Rtt) 461

dfmpyfix
Rdd=dfmpyfix(Rss,Rtt) 462

dfmpyhh
Rxx+=dfmpyhh(Rss,Rtt) 454

dfmpylh
Rxx+=dfmpylh(Rss,Rtt) 454

dfmpyll
Rdd=dfmpyll(Rss,Rtt) 462

dfsub
Rdd=dfsub(Rss,Rtt) 464

diag
diag(Rs) 314

diag0
diag0(Rss,Rtt) 314

diag1
diag1(Rss,Rtt) 314

dmsyncht
Rd=dmsyncht 321

E

endloop0
endloop0 193

endloop01
endloop01 193

endloop1
endloop1 193

extract
Rd=extract(Rs,#u5,#U5) 399
Rd=extract(Rs,Rtt) 399
Rdd=extract(Rss,#u6,#U6) 399
Rdd=extract(Rss,Rtt) 400

extractu
Rd=extractu(Rs,#u5,#U5) 399
Rd=extractu(Rs,Rtt) 399
Rdd=extractu(Rss,#u6,#U6) 400
Rdd=extractu(Rss,Rtt) 400

F

fastcorner9
Pd=[!]fastcorner9(Ps,Pt) 194

H

hintjr
hintjr(Rs) 206

I

icinva
icinva(Rs) 315

if ([!]p[01].new) jump:<hint> #r9:2 211, 211, 211, 211, 211, 211, 212, 212, 212

insert
Rx=insert(Rs,#u5,#U5) 402
Rx=insert(Rs,Rtt) 402
Rxx=insert(Rss,#u6,#U6) 402
Rxx=insert(Rss,Rtt) 403

interleave
Rdd=interleave(Rss) 404

isync
isync 316, 316

J

jump
if ([!]Pu.new) jump:<hint> #r15:2 216
if ([!]Pu) jump #r15:2 215
if ([!]Pu) jump:<hint> #r15:2 215
jump #r22:2 215
nt
if (Rs!=#0) jump:nt #r13:2 217
if (Rs<=#0) jump:nt #r13:2 217
if (Rs==#0) jump:nt #r13:2 217
if (Rs>=#0) jump:nt #r13:2 217
Rd=#U6 219
Rd=Rs 219
t
if (Rs!=#0) jump:t #r13:2 217
if (Rs<=#0) jump:t #r13:2 217
if (Rs==#0) jump:t #r13:2 217
if (Rs>=#0) jump:t #r13:2 217

jump #r9:2 219, 219

jumpr
if ([!]Pu) jumpr Rs 207
if ([!]Pu[.new]) jumpr:<hint> Rs 207
jumpr Rs 207

jumprh
jumprh Rs 207, 208

L

l2fetch
l2fetch(Rs,Rt) 318
l2fetch(Rs,Rtt) 318

lfs
Rdd=lfs(Rss,Rtt) 405

loop0
loop0(#r7:2,#U10) 196
loop0(#r7:2,Rs) 196

loop1
loop1(#r7:2,#U10) 196
loop1(#r7:2,Rs) 196

lsl
Rd=lsl(#s6,Rt) 570
Rd=lsl(Rs,Rt) 570
Rdd=lsl(Rss,Rt) 571
Rx[&|]=lsl(Rs,Rt) 576
Rx[+-]=lsl(Rs,Rt) 573
Rxx[&|]=lsl(Rss,Rt) 577
Rxx[+-]=lsl(Rss,Rt) 574
Rxx^=lsl(Rss,Rt) 577

lsr
Rd=lsr(Rs,#u5) 557
Rd=lsr(Rs,Rt) 570
Rdd=lsr(Rss,#u6) 557
Rdd=lsr(Rss,Rt) 571
Rx[&|]=lsr(Rs,#u5) 563
Rx[&|]=lsr(Rs,Rt) 576
Rx[+-]=lsr(Rs,#u5) 559
Rx[+-]=lsr(Rs,Rt) 573
Rx^=lsr(Rs,#u5) 563
Rx=add(#u8,lsr(Rx,#U5)) 559
Rx=and(#u8,lsr(Rx,#U5)) 563
Rx=or(#u8,lsr(Rx,#U5)) 563
Rx=sub(#u8,lsr(Rx,#U5)) 559
Rxx[&|]=lsr(Rss,#u6) 564
Rxx[&|]=lsr(Rss,Rt) 577
Rxx[+-]=lsr(Rss,#u6) 559
Rxx[+-]=lsr(Rss,Rt) 574
Rxx^=lsr(Rss,#u6) 564
Rxx^=lsr(Rss,Rt) 577

M

mask
Rd=mask(#u5,#U5) 556
Rdd=mask(Pt) 543

max
Rd=max(Rs,Rt) 339
Rdd=max(Rss,Rtt) 340

maxu
Rd=maxu(Rs,Rt) 339
Rdd=maxu(Rss,Rtt) 340

memb
- if ([!]Pt[.new]) Rd=memb(#u6) 227
- if ([!]Pt[.new]) Rd=memb(Rs+#u6:0) 227
- if ([!]Pt[.new]) Rd=memb(Rx++#s4:0) 227
- if ([!]Pv[.new]) memb(#u6)=Nt.new 272
- if ([!]Pv[.new]) memb(#u6)=Rt 289
- if ([!]Pv[.new]) memb(Rs+#u6:0)=#S6 289
- if ([!]Pv[.new]) memb(Rs+#u6:0)=Nt.new 272
- if ([!]Pv[.new]) memb(Rs+#u6:0)=Rt 289
- if ([!]Pv[.new]) memb(Rs+Ru<<#u2)=Nt.new 272
- if ([!]Pv[.new]) memb(Rs+Ru<<#u2)=Rt 289
- if ([!]Pv[.new]) memb(Rx++#s4:0)=Nt.new 272
- if ([!]Pv[.new]) memb(Rx++#s4:0)=Rt 289
- if ([!]Pv[.new]) Rd=memb(Rs+Rt<<#u2) 227
- memb(gp+#u16:0)=Nt.new 270
- memb(gp+#u16:0)=Rt 287
- memb(Re=#U6)=Nt.new 270
- memb(Re=#U6)=Rt 287
- memb(Rs+#s11:0)=Nt.new 270
- memb(Rs+#s11:0)=Rt 287
- memb(Rs+#u6:0)[+-]=#U5 263
- memb(Rs+#u6:0)[+-|&]=Rt 263
- memb(Rs+#u6:0)=#S8 287
- memb(Rs+Ru<<#u2)=Nt.new 270
- memb(Rs+Ru<<#u2)=Rt 287
- memb(Ru<<#u2+#U6)=Nt.new 270
- memb(Ru<<#u2+#U6)=Rt 287
- memb(Rx++#s4:0:circ(Mu))=Nt.new 270
- memb(Rx++#s4:0:circ(Mu))=Rt 287
- memb(Rx++#s4:0)=Nt.new 270
- memb(Rx++#s4:0)=Rt 287
- memb(Rx++I:circ(Mu))=Nt.new 270
- memb(Rx++I:circ(Mu))=Rt 287
- memb(Rx++Mu:brev)=Nt.new 270
- memb(Rx++Mu:brev)=Rt 287
- memb(Rx++Mu)=Nt.new 270
- memb(Rx++Mu)=Rt 287
- Rd=memb(gp+#u16:0) 225
- Rd=memb(Re=#U6) 225
- Rd=memb(Rs+#s11:0) 225
- Rd=memb(Rs+Rt<<#u2) 225
- Rd=memb(Rt<<#u2+#U6) 225
- Rd=memb(Rx++#s4:0:circ(Mu)) 225
- Rd=memb(Rx++#s4:0) 225
- Rd=memb(Rx++I:circ(Mu)) 225
- Rd=memb(Rx++Mu:brev) 225
- Rd=memb(Rx++Mu) 225

memb_fifo
- Ryy=memb_fifo(Re=#U6) 229
- Ryy=memb_fifo(Rs) 229
- Ryy=memb_fifo(Rs+#s11:0) 229
- Ryy=memb_fifo(Rt<<#u2+#U6) 229
- Ryy=memb_fifo(Rx++#s4:0:circ(Mu)) 229
- Ryy=memb_fifo(Rx++#s4:0) 229
- Ryy=memb_fifo(Rx++I:circ(Mu)) 229
- Ryy=memb_fifo(Rx++Mu:brev) 230
- Ryy=memb_fifo(Rx++Mu) 230

membh
Rd=membh(Re=#U6) 255
Rd=membh(Rs) 255
Rd=membh(Rs+#s11:1) 256
Rd=membh(Rt<<#u2+#U6) 256
Rd=membh(Rx++#s4:1:circ(Mu)) 256
Rd=membh(Rx++#s4:1) 256
Rd=membh(Rx++I:circ(Mu)) 256
Rd=membh(Rx++Mu:brev) 257
Rd=membh(Rx++Mu) 256
Rdd=membh(Re=#U6) 258
Rdd=membh(Rs) 258
Rdd=membh(Rs+#s11:2) 258
Rdd=membh(Rt<<#u2+#U6) 258
Rdd=membh(Rx++#s4:2:circ(Mu)) 259
Rdd=membh(Rx++#s4:2) 259
Rdd=membh(Rx++I:circ(Mu)) 259
Rdd=membh(Rx++Mu:brev) 259
Rdd=membh(Rx++Mu) 259

memd
if ([!]Pt[.new]) Rdd=memd(#u6) 223
if ([!]Pt[.new]) Rdd=memd(Rs+#u6:3) 223
if ([!]Pt[.new]) Rdd=memd(Rx++#s4:3) 223
if ([!]Pv[.new]) memd(#u6)=Rtt 285
if ([!]Pv[.new]) memd(Rs+#u6:3)=Rtt 285
if ([!]Pv[.new]) memd(Rs+Ru<<#u2)=Rtt 285
if ([!]Pv[.new]) memd(Rx++#s4:3)=Rtt 285
if ([!]Pv[.new]) Rdd=memd(Rs+Rt<<#u2) 223
memd(gp+#u16:3)=Rtt 282
memd(Re=#U6)=Rtt 282
memd(Rs+#s11:3)=Rtt 282
memd(Rs+Ru<<#u2)=Rtt 282
memd(Ru<<#u2+#U6)=Rtt 282
memd(Rx++#s4:3:circ(Mu))=Rtt 282
memd(Rx++#s4:3)=Rtt 282
memd(Rx++I:circ(Mu))=Rtt 282
memd(Rx++Mu:brev)=Rtt 282
memd(Rx++Mu)=Rtt 282
Rdd=memd(gp+#u16:3) 220
Rdd=memd(Re=#U6) 220
Rdd=memd(Rs+#s11:3) 220
Rdd=memd(Rs+Rt<<#u2) 220
Rdd=memd(Rt<<#u2+#U6) 220
Rdd=memd(Rx++#s4:3:circ(Mu)) 220
Rdd=memd(Rx++#s4:3) 220
Rdd=memd(Rx++I:circ(Mu)) 220
Rdd=memd(Rx++Mu:brev) 220
Rdd=memd(Rx++Mu) 220

memd_aq
Rdd=memd_aq(Rs) 222

memd_locked
memd_locked(Rs,Pd)=Rtt 306
Rdd=memd_locked(Rs) 305

memd_rl
memd_rl(Rs):at=Rtt 284
memd_rl(Rs):st=Rtt 284

memh
if ([!]Pt[.new]) Rd=memh(#u6) 236
if ([!]Pt[.new]) Rd=memh(Rs+#u6:1) 236
if ([!]Pt[.new]) Rd=memh(Rx++#s4:1) 236
if ([!]Pv[.new]) memh(#u6)=Nt.new 276
if ([!]Pv[.new]) memh(#u6)=Rt 294
if ([!]Pv[.new]) memh(#u6)=Rt.H 294
if ([!]Pv[.new]) memh(Rs+#u6:1)=#S6 294
if ([!]Pv[.new]) memh(Rs+#u6:1)=Nt.new 276
if ([!]Pv[.new]) memh(Rs+#u6:1)=Rt 294
if ([!]Pv[.new]) memh(Rs+#u6:1)=Rt.H 294
if ([!]Pv[.new]) memh(Rs+Ru<<#u2)=Nt.new 276
if ([!]Pv[.new]) memh(Rs+Ru<<#u2)=Rt 295
if ([!]Pv[.new]) memh(Rs+Ru<<#u2)=Rt.H 294
if ([!]Pv[.new]) memh(Rx++#s4:1)=Nt.new 276
if ([!]Pv[.new]) memh(Rx++#s4:1)=Rt 295
if ([!]Pv[.new]) memh(Rx++#s4:1)=Rt.H 295
if ([!]Pv[.new]) Rd=memh(Rs+Rt<<#u2) 236
memh(gp+#u16:1)=Nt.new 274
memh(gp+#u16:1)=Rt 292
memh(gp+#u16:1)=Rt.H 292
memh(Re=#U6)=Nt.new 274
memh(Re=#U6)=Rt 291
memh(Re=#U6)=Rt.H 291
memh(Rs+#s11:1)=Nt.new 274
memh(Rs+#s11:1)=Rt 291
memh(Rs+#s11:1)=Rt.H 291
memh(Rs+#u6:1)[+-]=#U5 264
memh(Rs+#u6:1)[+-|&]=Rt 264
memh(Rs+#u6:1)=#S8 291
memh(Rs+Ru<<#u2)=Nt.new 274
memh(Rs+Ru<<#u2)=Rt 291
memh(Rs+Ru<<#u2)=Rt.H 291
memh(Ru<<#u2+#U6)=Nt.new 274
memh(Ru<<#u2+#U6)=Rt 291
memh(Ru<<#u2+#U6)=Rt.H 291
memh(Rx++#s4:1:circ(Mu))=Nt.new 274
memh(Rx++#s4:1:circ(Mu))=Rt 291
memh(Rx++#s4:1:circ(Mu))=Rt.H 291
memh(Rx++#s4:1)=Nt.new 274
memh(Rx++#s4:1)=Rt 291
memh(Rx++#s4:1)=Rt.H 291
memh(Rx++I:circ(Mu))=Nt.new 274
memh(Rx++I:circ(Mu))=Rt 292
memh(Rx++I:circ(Mu))=Rt.H 291
memh(Rx++Mu:brev)=Nt.new 274
memh(Rx++Mu:brev)=Rt 292
memh(Rx++Mu:brev)=Rt.H 292
memh(Rx++Mu)=Nt.new 274
memh(Rx++Mu)=Rt 292
memh(Rx++Mu)=Rt.H 292
Rd=memh(gp+#u16:1) 234
Rd=memh(Re=#U6) 234
Rd=memh(Rs+#s11:1) 234
Rd=memh(Rs+Rt<<#u2) 234
Rd=memh(Rt<<#u2+#U6) 234
Rd=memh(Rx++#s4:1:circ(Mu)) 234
Rd=memh(Rx++#s4:1) 234
Rd=memh(Rx++I:circ(Mu)) 234
Rd=memh(Rx++Mu:brev) 234
Rd=memh(Rx++Mu) 234

memh_fifo
Ryy=memh_fifo(Re=#U6) 231
Ryy=memh_fifo(Rs) 231
Ryy=memh_fifo(Rs+#s11:1) 231
Ryy=memh_fifo(Rt<<#u2+#U6) 231
Ryy=memh_fifo(Rx++#s4:1:circ(Mu)) 231
Ryy=memh_fifo(Rx++#s4:1) 231
Ryy=memh_fifo(Rx++I:circ(Mu)) 232
Ryy=memh_fifo(Rx++Mu:brev) 232
Ryy=memh_fifo(Rx++Mu) 232

memub
if ([!]Pt[.new]) Rd=memub(#u6) 240
if ([!]Pt[.new]) Rd=memub(Rs+#u6:0) 240
if ([!]Pt[.new]) Rd=memub(Rx++#s4:0) 240
if ([!]Pv[.new]) Rd=memub(Rs+Rt<<#u2) 240
Rd=memub(gp+#u16:0) 238
Rd=memub(Re=#U6) 238
Rd=memub(Rs+#s11:0) 238
Rd=memub(Rs+Rt<<#u2) 238
Rd=memub(Rt<<#u2+#U6) 238
Rd=memub(Rx++#s4:0:circ(Mu)) 238
Rd=memub(Rx++#s4:0) 238
Rd=memub(Rx++I:circ(Mu)) 238
Rd=memub(Rx++Mu:brev) 238
Rd=memub(Rx++Mu) 238

memubh
Rd=memubh(Re=#U6) 257
Rd=memubh(Rs+#s11:1) 257
Rd=memubh(Rt<<#u2+#U6) 257
Rd=memubh(Rx++#s4:1:circ(Mu)) 257
Rd=memubh(Rx++#s4:1) 257
Rd=memubh(Rx++I:circ(Mu)) 258
Rd=memubh(Rx++Mu:brev) 258
Rd=memubh(Rx++Mu) 258
Rdd=memubh(Re=#U6) 259
Rdd=memubh(Rs+#s11:2) 260
Rdd=memubh(Rt<<#u2+#U6) 260
Rdd=memubh(Rx++#s4:2:circ(Mu)) 260
Rdd=memubh(Rx++#s4:2) 260
Rdd=memubh(Rx++I:circ(Mu)) 260
Rdd=memubh(Rx++Mu:brev) 261
Rdd=memubh(Rx++Mu) 260

memuh
if ([!]Pt[.new]) Rd=memuh(#u6) 244
if ([!]Pt[.new]) Rd=memuh(Rs+#u6:1) 244
if ([!]Pt[.new]) Rd=memuh(Rx++#s4:1) 244
if ([!]Pv[.new]) Rd=memuh(Rs+Rt<<#u2) 244
Rd=memuh(gp+#u16:1) 242
Rd=memuh(Re=#U6) 242
Rd=memuh(Rs+#s11:1) 242
Rd=memuh(Rs+Rt<<#u2) 242
Rd=memuh(Rt<<#u2+#U6) 242
Rd=memuh(Rx++#s4:1:circ(Mu)) 242
Rd=memuh(Rx++#s4:1) 242
Rd=memuh(Rx++I:circ(Mu)) 242
Rd=memuh(Rx++Mu:brev) 242
Rd=memuh(Rx++Mu) 242

memw
if ([!]Pt[.new]) Rd=memw(#u6) 249
if ([!]Pt[.new]) Rd=memw(Rs+#u6:2) 249
if ([!]Pt[.new]) Rd=memw(Rx++#s4:2) 249
if ([!]Pv[.new]) memw(#u6)=Nt.new 280
if ([!]Pv[.new]) memw(#u6)=Rt 301
if ([!]Pv[.new]) memw(Rs+#u6:2)=#S6 301
if ([!]Pv[.new]) memw(Rs+#u6:2)=Nt.new 280
if ([!]Pv[.new]) memw(Rs+#u6:2)=Rt 301
if ([!]Pv[.new]) memw(Rs+Ru<<#u2)=Nt.new 280
if ([!]Pv[.new]) memw(Rs+Ru<<#u2)=Rt 301
if ([!]Pv[.new]) memw(Rx++#s4:2)=Nt.new 280
if ([!]Pv[.new]) memw(Rx++#s4:2)=Rt 301
if ([!]Pv[.new]) Rd=memw(Rs+Rt<<#u2) 249
memw(gp+#u16:2)=Nt.new 278
memw(gp+#u16:2)=Rt 298
memw(Re=#U6)=Nt.new 278
memw(Re=#U6)=Rt 298
memw(Rs+#s11:2)=Nt.new 278
memw(Rs+#s11:2)=Rt 298
memw(Rs+#u6:2)[+-]=#U5 265
memw(Rs+#u6:2)[+-|&]=Rt 265
memw(Rs+#u6:2)=#S8 298
memw(Rs+Ru<<#u2)=Nt.new 278
memw(Rs+Ru<<#u2)=Rt 298
memw(Ru<<#u2+#U6)=Nt.new 278
memw(Ru<<#u2+#U6)=Rt 298
memw(Rx++#s4:2:circ(Mu))=Nt.new 278
memw(Rx++#s4:2:circ(Mu))=Rt 298
memw(Rx++#s4:2)=Nt.new 278
memw(Rx++#s4:2)=Rt 298
memw(Rx++I:circ(Mu))=Nt.new 278
memw(Rx++I:circ(Mu))=Rt 298
memw(Rx++Mu:brev)=Nt.new 278
memw(Rx++Mu:brev)=Rt 298
memw(Rx++Mu)=Nt.new 278
memw(Rx++Mu)=Rt 298
Rd=memw(gp+#u16:2) 246
Rd=memw(Re=#U6) 246
Rd=memw(Rs+#s11:2) 246
Rd=memw(Rs+Rt<<#u2) 246
Rd=memw(Rt<<#u2+#U6) 246
Rd=memw(Rx++#s4:2:circ(Mu)) 246
Rd=memw(Rx++#s4:2) 246
Rd=memw(Rx++I:circ(Mu)) 246
Rd=memw(Rx++Mu:brev) 246
Rd=memw(Rx++Mu) 246

memw_aq
Rd=memw_aq(Rs) 248

memw_locked
memw_locked(Rs,Pd)=Rt 306
Rd=memw_locked(Rs) 305

memw_rl
memw_rl(Rs):at=Rt 300
memw_rl(Rs):st=Rt 300

min
Rd=min(Rt,Rs) 341
Rdd=min(Rtt,Rss) 342

minu
Rd=minu(Rt,Rs) 341
Rdd=minu(Rtt,Rss) 342

modwrap
Rd=modwrap(Rs,Rt) 343

mpy
Rd=mpy(Rs,Rt.H):<<1:rnd:sat 491
Rd=mpy(Rs,Rt.H):<<1:sat 491
Rd=mpy(Rs,Rt.L):<<1:rnd:sat 491
Rd=mpy(Rs,Rt.L):<<1:sat 491
Rd=mpy(Rs,Rt) 491
Rd=mpy(Rs,Rt):<<1 491
Rd=mpy(Rs,Rt):<<1:sat 491
Rd=mpy(Rs,Rt):rnd 491
Rd=mpy(Rs.[HL],Rt.[HL])[:<<1][:rnd][:sat] 475
Rdd=mpy(Rs,Rt) 493
Rdd=mpy(Rs.[HL],Rt.[HL])[:<<1][:rnd] 475
Rx+=mpy(Rs,Rt):<<1:sat 491
Rx+=mpy(Rs.[HL],Rt.[HL])[:<<1][:sat] 475
Rx-=mpy(Rs,Rt):<<1:sat 491
Rx-=mpy(Rs.[HL],Rt.[HL])[:<<1][:sat] 475
Rxx[+-]=mpy(Rs,Rt) 493
Rxx+=mpy(Rs.[HL],Rt.[HL])[:<<1] 475
Rxx-=mpy(Rs.[HL],Rt.[HL])[:<<1] 475

mpyi
Rd=+mpyi(Rs,#u8) 465
Rd=mpyi(Rs,#m9) 466
Rd=-mpyi(Rs,#u8) 465
Rd=mpyi(Rs,Rt) 466
Rx+=mpyi(Rs,#u8) 466
Rx+=mpyi(Rs,Rt) 466
Rx-=mpyi(Rs,#u8) 466
Rx-=mpyi(Rs,Rt) 466

mpysu
Rd=mpysu(Rs,Rt) 491

mpyu
Rd=mpyu(Rs,Rt) 491
Rd=mpyu(Rs.[HL],Rt.[HL])[:<<1] 482
Rdd=mpyu(Rs,Rt) 493
Rdd=mpyu(Rs.[HL],Rt.[HL])[:<<1] 482
Rx+=mpyu(Rs.[HL],Rt.[HL])[:<<1] 482
Rx-=mpyu(Rs.[HL],Rt.[HL])[:<<1] 482
Rxx[+-]=mpyu(Rs,Rt) 493
Rxx+=mpyu(Rs.[HL],Rt.[HL])[:<<1] 482
Rxx-=mpyu(Rs.[HL],Rt.[HL])[:<<1] 482

mpyui
Rd=mpyui(Rs,Rt) 466

mux
Rd=mux(Pu,#s8,#S8) 174
Rd=mux(Pu,#s8,Rs) 174
Rd=mux(Pu,Rs,#s8) 174
Rd=mux(Pu,Rs,Rt) 174

N

neg
Rd=neg(Rs) 161
Rd=neg(Rs):sat 344
Rdd=neg(Rss) 344

no mnemonic
Cd=Rs 203
Cdd=Rss 203
if ([!]Pu[.new]) Rd=#s12 188
if ([!]Pu[.new]) Rd=Rs 188
if ([!]Pu[.new]) Rdd=Rss 188
Pd=Ps 201
Pd=Rs 545, 545
Rd=#s16 165
Rd=Cs 203
Rd=Ps 545, 545
Rd=Rs 167
Rdd=#s8 165
Rdd=Css 203
Rdd=Rss 167
Rx.[HL]=#u16 165

nop
nop 162

normamt
Rd=normamt(Rs) 395
Rd=normamt(Rss) 395

not
Pd=not(Ps) 201
Rd=not(Rs) 159
Rdd=not(Rss) 335

O

or
if ([!]Pu[.new]) Rd=or(Rs,Rt) 183
Pd=and(Ps,or(Pt,[!]Pu)) 201
Pd=or(Ps,and(Pt,[!]Pu)) 201
Pd=or(Ps,or(Pt,[!]Pu)) 201
Pd=or(Pt,[!]Ps) 201
Rd=or(Rs,#s10) 159
Rd=or(Rs,Rt) 159
Rd=or(Rt,~Rs) 159
Rdd=or(Rss,Rtt) 335
Rdd=or(Rtt,~Rss) 335
Rx[&|^]=or(Rs,Rt) 337
Rx=or(Ru,and(Rx,#s10)) 337
Rx|=or(Rs,#s10) 337

P

packhl
Rdd=packhl(Rs,Rt) 177

parity
Rd=parity(Rs,Rt) 406
Rd=parity(Rss,Rtt) 406

pause
pause(#u10) 320

pc
Rd=add(pc,#u6) 198

pmpyw
Rdd=pmpyw(Rs,Rt) 487
Rxx^=pmpyw(Rs,Rt) 487

popcount
Rd=popcount(Rss) 397

R

release
release(Rs):at 297
release(Rs):st 297

rol
Rd=rol(Rs,#u5) 557
Rdd=rol(Rss,#u6) 557
Rx[&|]=rol(Rs,#u5) 563
Rx[+-]=rol(Rs,#u5) 559
Rx^=rol(Rs,#u5) 563
Rxx[&|]=rol(Rss,#u6) 564
Rxx[+-]=rol(Rss,#u6) 559
Rxx^=rol(Rss,#u6) 564

round
Rd=round(Rs,#u5)[:sat] 345
Rd=round(Rs,Rt)[:sat] 345
Rd=round(Rss):sat 345

S

sat
Rd=sat(Rss) 518

satb
Rd=satb(Rs) 518

sath
Rd=sath(Rs) 518

satub
Rd=satub(Rs) 518

satuh
Rd=satuh(Rs) 518

setbit
memb(Rs+#u6:0)=setbit(#U5) 263
memh(Rs+#u6:1)=setbit(#U5) 264
memw(Rs+#u6:2)=setbit(#U5) 265
Rd=setbit(Rs,#u5) 408
Rd=setbit(Rs,Rt) 408

sfadd
Rd=sfadd(Rs,Rt) 444

sfclass
Pd=sfclass(Rs,#u5) 445

sfcmp.eq
Pd=sfcmp.eq(Rs,Rt) 446

sfcmp.ge
Pd=sfcmp.ge(Rs,Rt) 446

sfcmp.gt
Pd=sfcmp.gt(Rs,Rt) 446

sfcmp.uo
Pd=sfcmp.uo(Rs,Rt) 446

sffixupd
Rd=sffixupd(Rs,Rt) 453

sffixupn
Rd=sffixupn(Rs,Rt) 453

sffixupr
Rd=sffixupr(Rs) 453

sfinvsqrta
Rd,Pe=sfinvsqrta(Rs) 456

sfmake
Rd=sfmake(#u10):neg 459
Rd=sfmake(#u10):pos 459

sfmax
Rd=sfmax(Rs,Rt) 460

sfmin
Rd=sfmin(Rs,Rt) 461

sfmpy
Rd=sfmpy(Rs,Rt) 462
Rx+=sfmpy(Rs,Rt,Pu):scale 455
Rx+=sfmpy(Rs,Rt) 454
Rx+=sfmpy(Rs,Rt):lib 457
Rx-=sfmpy(Rs,Rt) 454
Rx-=sfmpy(Rs,Rt):lib 457

sfrecipa
Rd,Pe=sfrecipa(Rs,Rt) 463

sfsub
Rd=sfsub(Rs,Rt) 464

shuffeb
Rdd=shuffeb(Rss,Rtt) 528

shuffeh
Rdd=shuffeh(Rss,Rtt) 528

shuffob
Rdd=shuffob(Rtt,Rss) 528

shuffoh
Rdd=shuffoh(Rtt,Rss) 528

sp1loop0
p3=sp1loop0(#r7:2,#U10) 199
p3=sp1loop0(#r7:2,Rs) 199

sp2loop0
p3=sp2loop0(#r7:2,#U10) 199
p3=sp2loop0(#r7:2,Rs) 199

sp3loop0
p3=sp3loop0(#r7:2,#U10) 199
p3=sp3loop0(#r7:2,Rs) 199

sub
if ([!]Pu[.new]) Rd=sub(Rt,Rs) 185
Rd=add(Rs,sub(#s6,Ru)) 327
Rd=sub(#s10,Rs) 163
Rd=sub(Rt,Rs) 163
Rd=sub(Rt,Rs):sat 163
Rd=sub(Rt,Rs):sat:deprecated 348
Rd=sub(Rt.[HL],Rs.[HL])[:sat]:<<16 350
Rd=sub(Rt.L,Rs.[HL])[:sat] 350
Rdd=sub(Rss,Rtt,Px):carry 333
Rdd=sub(Rtt,Rss) 348
Rx+=sub(Rt,Rs) 349

swiz
Rd=swiz(Rs) 520

sxtb
if ([!]Pu[.new]) Rd=sxtb(Rs) 186
Rd=sxtb(Rs) 164

sxth
if ([!]Pu[.new]) Rd=sxth(Rs) 186
Rd=sxth(Rs) 164

sxtw
Rdd=sxtw(Rs) 352

syncht
syncht 321, 321, 321, 321

T

tableidxb
Rx=tableidxb(Rs,#u4,#S6):raw 412
Rx=tableidxb(Rs,#u4,#U5) 412

tableidxd
Rx=tableidxd(Rs,#u4,#S6):raw 412
Rx=tableidxd(Rs,#u4,#U5) 413

tableidxh
Rx=tableidxh(Rs,#u4,#S6):raw 413
Rx=tableidxh(Rs,#u4,#U5) 413

tableidxw
Rx=tableidxw(Rs,#u4,#S6):raw 413
Rx=tableidxw(Rs,#u4,#U5) 413

tlbmatch
Pd=tlbmatch(Rss,Rt) 544

togglebit
Rd=togglebit(Rs,#u5) 408
Rd=togglebit(Rs,Rt) 408

trace
trace(Rs) 322

trap0
trap0(#u8) 323

trap1
trap1(#u8) 323
trap1(Rx,#u8) 323

tstbit
if ([!]tstbit(Ns.new,#0)) jump:<hint> #r9:2 267
p[01]=tstbit(Rs,#0) 212
Pd=[!]tstbit(Rs,#u5) 546
Pd=[!]tstbit(Rs,Rt) 546

U

unpause
unpause 324, 324

V

vabsdiffb
Rdd=vabsdiffb(Rtt,Rss) 355

vabsdiffh
Rdd=vabsdiffh(Rtt,Rss) 356

vabsdiffub
Rdd=vabsdiffub(Rtt,Rss) 355

vabsdiffw
Rdd=vabsdiffw(Rtt,Rss) 357

vabsh
Rdd=vabsh(Rss) 353
Rdd=vabsh(Rss):sat 353

vabsw
Rdd=vabsw(Rss) 354
Rdd=vabsw(Rss):sat 354

vacsh
Rxx,Pe=vacsh(Rss,Rtt) 359

vaddb
Rdd=vaddb(Rss,Rtt) 367

vaddh
Rd=vaddh(Rs,Rt)[:sat] 168
Rdd=vaddh(Rss,Rtt)[:sat] 361

vaddhub
Rd=vaddhub(Rss,Rtt):sat 362

vaddub
Rdd=vaddub(Rss,Rtt)[:sat] 367

vadduh
Rd=vadduh(Rs,Rt):sat 168
Rdd=vadduh(Rss,Rtt):sat 361

vaddw
Rdd=vaddw(Rss,Rtt)[:sat] 368

valignb
Rdd=valignb(Rtt,Rss,#u3) 521
Rdd=valignb(Rtt,Rss,Pu) 521

vaslh
Rdd=vaslh(Rss,#u4) 581
Rdd=vaslh(Rss,Rt) 585

vaslw
Rdd=vaslw(Rss,#u5) 587
Rdd=vaslw(Rss,Rt) 588

vasrh
Rdd=vasrh(Rss,#u4) 581
Rdd=vasrh(Rss,#u4):raw 582
Rdd=vasrh(Rss,#u4):rnd 582
Rdd=vasrh(Rss,Rt) 585

vasrhub
Rd=vasrhub(Rss,#u4):raw 583
Rd=vasrhub(Rss,#u4):rnd:sat 583
Rd=vasrhub(Rss,#u4):sat 583

vasrw
Rd=vasrw(Rss,#u5) 590
Rd=vasrw(Rss,Rt) 590
Rdd=vasrw(Rss,#u5) 587
Rdd=vasrw(Rss,Rt) 588

vavgh
Rd=vavgh(Rs,Rt) 169
Rd=vavgh(Rs,Rt):rnd 169
Rdd=vavgh(Rss,Rtt) 369
Rdd=vavgh(Rss,Rtt):crnd 369
Rdd=vavgh(Rss,Rtt):rnd 369

vavgub
Rdd=vavgub(Rss,Rtt) 371
Rdd=vavgub(Rss,Rtt):rnd 371

vavguh
Rdd=vavguh(Rss,Rtt) 369
Rdd=vavguh(Rss,Rtt):rnd 369

vavguw
Rdd=vavguw(Rss,Rtt)[:rnd] 372

vavgw
Rdd=vavgw(Rss,Rtt):crnd 372
Rdd=vavgw(Rss,Rtt)[:rnd] 372

vclip
Rdd=vclip(Rss,#u5) 374

vcmpb.eq
Pd=!any8(vcmpb.eq(Rss,Rtt)) 549
Pd=any8(vcmpb.eq(Rss,Rtt)) 549
Pd=vcmpb.eq(Rss,#u8) 550
Pd=vcmpb.eq(Rss,Rtt) 550

vcmpb.gt
Pd=vcmpb.gt(Rss,#s8) 550
Pd=vcmpb.gt(Rss,Rtt) 550

vcmpb.gtu
Pd=vcmpb.gtu(Rss,#u7) 550
Pd=vcmpb.gtu(Rss,Rtt) 550

vcmph.eq
Pd=vcmph.eq(Rss,#s8) 547
Pd=vcmph.eq(Rss,Rtt) 547

vcmph.gt
Pd=vcmph.gt(Rss,#s8) 547
Pd=vcmph.gt(Rss,Rtt) 547

vcmph.gtu
Pd=vcmph.gtu(Rss,#u7) 547
Pd=vcmph.gtu(Rss,Rtt) 547

vcmpw.eq
Pd=vcmpw.eq(Rss,#s8) 552
Pd=vcmpw.eq(Rss,Rtt) 552

vcmpw.gt
Pd=vcmpw.gt(Rss,#s8) 552
Pd=vcmpw.gt(Rss,Rtt) 552

vcmpw.gtu
Pd=vcmpw.gtu(Rss,#u7) 552
Pd=vcmpw.gtu(Rss,Rtt) 552

vcmpyi
Rdd=vcmpyi(Rss,Rtt)[:<<1]:sat 431
Rxx+=vcmpyi(Rss,Rtt):sat 431

vcmpyr
Rdd=vcmpyr(Rss,Rtt)[:<<1]:sat 431
Rxx+=vcmpyr(Rss,Rtt):sat 432

vcnegh
Rdd=vcnegh(Rss,Rt) 375

vconj
Rdd=vconj(Rss):sat 433

vcrotate
Rdd=vcrotate(Rss,Rt) 434

vdmpy
Rd=vdmpy(Rss,Rtt)[:<<1]:rnd:sat 497
Rdd=vdmpy(Rss,Rtt):<<1:sat 495
Rdd=vdmpy(Rss,Rtt):sat 495
Rxx+=vdmpy(Rss,Rtt):<<1:sat 495
Rxx+=vdmpy(Rss,Rtt):sat 496

vdmpybsu
Rdd=vdmpybsu(Rss,Rtt):sat 501
Rxx+=vdmpybsu(Rss,Rtt):sat 501

vitpack
Rd=vitpack(Ps,Pt) 554

vlslh
Rdd=vlslh(Rss,Rt) 585

vlslw
Rdd=vlslw(Rss,Rt) 588

vlsrh
Rdd=vlsrh(Rss,#u4) 581
Rdd=vlsrh(Rss,Rt) 585

vlsrw
Rdd=vlsrw(Rss,#u5) 587
Rdd=vlsrw(Rss,Rt) 588

vmaxb
Rdd=vmaxb(Rtt,Rss) 376

vmaxh
Rdd=vmaxh(Rtt,Rss) 377

vmaxub
Rdd=vmaxub(Rtt,Rss) 376

vmaxuh
Rdd=vmaxuh(Rtt,Rss) 377

vmaxuw
Rdd=vmaxuw(Rtt,Rss) 382

vmaxw
Rdd=vmaxw(Rtt,Rss) 382

vminb
Rdd=vminb(Rtt,Rss) 383

vminh
Rdd=vminh(Rtt,Rss) 384

vminub
Rdd,Pe=vminub(Rtt,Rss) 383
Rdd=vminub(Rtt,Rss) 383

vminuh
Rdd=vminuh(Rtt,Rss) 384

vminuw
Rdd=vminuw(Rtt,Rss) 388

vminw
Rdd=vminw(Rtt,Rss) 388

vmpybsu
Rdd=vmpybsu(Rs,Rt) 512
Rxx+=vmpybsu(Rs,Rt) 512

vmpybu
Rdd=vmpybu(Rs,Rt) 512
Rxx+=vmpybu(Rs,Rt) 512

vmpyeh
Rdd=vmpyeh(Rss,Rtt):<<1:sat 503
Rdd=vmpyeh(Rss,Rtt):sat 503
Rxx+=vmpyeh(Rss,Rtt) 503
Rxx+=vmpyeh(Rss,Rtt):<<1:sat 503
Rxx+=vmpyeh(Rss,Rtt):sat 503

vmpyh
Rd=vmpyh(Rs,Rt)[:<<1]:rnd:sat 507
Rdd=vmpyh(Rs,Rt)[:<<1]:sat 505
Rxx+=vmpyh(Rs,Rt) 505
Rxx+=vmpyh(Rs,Rt)[:<<1]:sat 505

vmpyhsu
Rdd=vmpyhsu(Rs,Rt)[:<<1]:sat 509
Rxx+=vmpyhsu(Rs,Rt)[:<<1]:sat 509

vmpyweh
Rdd=vmpyweh(Rss,Rtt)[:<<1]:rnd:sat 468
Rdd=vmpyweh(Rss,Rtt)[:<<1]:sat 468
Rxx+=vmpyweh(Rss,Rtt)[:<<1]:rnd:sat 469
Rxx+=vmpyweh(Rss,Rtt)[:<<1]:sat 469

vmpyweuh
Rdd=vmpyweuh(Rss,Rtt)[:<<1]:rnd:sat 471
Rdd=vmpyweuh(Rss,Rtt)[:<<1]:sat 472
Rxx+=vmpyweuh(Rss,Rtt)[:<<1]:rnd:sat 472
Rxx+=vmpyweuh(Rss,Rtt)[:<<1]:sat 472

vmpywoh
Rdd=vmpywoh(Rss,Rtt)[:<<1]:rnd:sat 469
Rdd=vmpywoh(Rss,Rtt)[:<<1]:sat 469
Rxx+=vmpywoh(Rss,Rtt)[:<<1]:rnd:sat 469
Rxx+=vmpywoh(Rss,Rtt)[:<<1]:sat 469

vmpywouh
Rdd=vmpywouh(Rss,Rtt)[:<<1]:rnd:sat 472
Rdd=vmpywouh(Rss,Rtt)[:<<1]:sat 472
Rxx+=vmpywouh(Rss,Rtt)[:<<1]:rnd:sat 472
Rxx+=vmpywouh(Rss,Rtt)[:<<1]:sat 472

vmux
Rdd=vmux(Pu,Rss,Rtt) 555

vnavgh
Rd=vnavgh(Rt,Rs) 169
Rdd=vnavgh(Rtt,Rss) 369
Rdd=vnavgh(Rtt,Rss):crnd:sat 369
Rdd=vnavgh(Rtt,Rss):rnd:sat 369

vnavgw
Rdd=vnavgw(Rtt,Rss) 372
Rdd=vnavgw(Rtt,Rss):crnd:sat 372
Rdd=vnavgw(Rtt,Rss):rnd:sat 372

vpmpyh
Rdd=vpmpyh(Rs,Rt) 514
Rxx^=vpmpyh(Rs,Rt) 515

vraddh
Rd=vraddh(Rss,Rtt) 365

vraddub
Rdd=vraddub(Rss,Rtt) 363
Rxx+=vraddub(Rss,Rtt) 363

vradduh
Rd=vradduh(Rss,Rtt) 365

vrcmpys
Rd=vrcmpys(Rss,Rt):<<1:rnd:sat 439
Rd=vrcmpys(Rss,Rtt):<<1:rnd:sat:raw:hi 439
Rd=vrcmpys(Rss,Rtt):<<1:rnd:sat:raw:lo 439
Rdd=vrcmpys(Rss,Rt):<<1:sat 436
Rdd=vrcmpys(Rss,Rtt):<<1:sat:raw:hi 436
Rdd=vrcmpys(Rss,Rtt):<<1:sat:raw:lo 436
Rxx+=vrcmpys(Rss,Rt):<<1:sat 437
Rxx+=vrcmpys(Rss,Rtt):<<1:sat:raw:hi 437
Rxx+=vrcmpys(Rss,Rtt):<<1:sat:raw:lo 437

vrcnegh
Rxx+=vrcnegh(Rss,Rt) 375

vrcrotate
Rdd=vrcrotate(Rss,Rt,#u2) 442
Rxx+=vrcrotate(Rss,Rt,#u2) 442

vrmaxh
Rxx=vrmaxh(Rss,Ru) 378

vrmaxuh
Rxx=vrmaxuh(Rss,Ru) 378

vrmaxuw
Rxx=vrmaxuw(Rss,Ru) 380

vrmaxw
Rxx=vrmaxw(Rss,Ru) 380

vrminh
Rxx=vrminh(Rss,Ru) 385

vrminuh
Rxx=vrminuh(Rss,Ru) 385

vrminuw
Rxx=vrminuw(Rss,Ru) 386

vrminw
Rxx=vrminw(Rss,Ru) 386

vrmpybsu
Rdd=vrmpybsu(Rss,Rtt) 499
Rxx+=vrmpybsu(Rss,Rtt) 499

vrmpybu
Rdd=vrmpybu(Rss,Rtt) 499
Rxx+=vrmpybu(Rss,Rtt) 500

vrmpyh
Rdd=vrmpyh(Rss,Rtt) 510
Rxx+=vrmpyh(Rss,Rtt) 510

vrmpyweh
Rdd=vrmpyweh(Rss,Rtt)[:<<1] 489
Rxx+=vrmpyweh(Rss,Rtt)[:<<1] 489

vrmpywoh
Rdd=vrmpywoh(Rss,Rtt)[:<<1] 489
Rxx+=vrmpywoh(Rss,Rtt)[:<<1] 489

vrndwh
Rd=vrndwh(Rss) 522
Rd=vrndwh(Rss):sat 522

vrsadub
Rdd=vrsadub(Rss,Rtt) 389
Rxx+=vrsadub(Rss,Rtt) 389

vsathb
Rd=vsathb(Rs) 524
Rd=vsathb(Rss) 524
Rdd=vsathb(Rss) 526

vsathub
Rd=vsathub(Rs) 524
Rd=vsathub(Rss) 524
Rdd=vsathub(Rss) 526

vsatwh
Rd=vsatwh(Rss) 524
Rdd=vsatwh(Rss) 526

vsatwuh
Rd=vsatwuh(Rss) 525
Rdd=vsatwuh(Rss) 526

vsplatb
Rd=vsplatb(Rs) 530
Rdd=vsplatb(Rs) 530

vsplath
Rdd=vsplath(Rs) 531

vspliceb
Rdd=vspliceb(Rss,Rtt,#u3) 532
Rdd=vspliceb(Rss,Rtt,Pu) 532

vsubb
Rdd=vsubb(Rss,Rtt) 393

vsubh
Rd=vsubh(Rt,Rs)[:sat] 170
Rdd=vsubh(Rtt,Rss)[:sat] 391

vsubub
Rdd=vsubub(Rtt,Rss)[:sat] 393

vsubuh
Rd=vsubuh(Rt,Rs):sat 170
Rdd=vsubuh(Rtt,Rss):sat 391

vsubw
Rdd=vsubw(Rtt,Rss)[:sat] 394

vsxtbh
Rdd=vsxtbh(Rs) 533

vsxthw
Rdd=vsxthw(Rs) 533

vtrunehb
Rd=vtrunehb(Rss) 534
Rdd=vtrunehb(Rss,Rtt) 534

vtrunewh
Rdd=vtrunewh(Rss,Rtt) 534

vtrunohb
Rd=vtrunohb(Rss) 534
Rdd=vtrunohb(Rss,Rtt) 534

vtrunowh
Rdd=vtrunowh(Rss,Rtt) 534

vxaddsubh
Rdd=vxaddsubh(Rss,Rtt):rnd:>>1:sat 414
Rdd=vxaddsubh(Rss,Rtt):sat 414

vxaddsubw
Rdd=vxaddsubw(Rss,Rtt):sat 416

vxsubaddh
Rdd=vxsubaddh(Rss,Rtt):rnd:>>1:sat 415
Rdd=vxsubaddh(Rss,Rtt):sat 415

vxsubaddw
Rdd=vxsubaddw(Rss,Rtt):sat 416

vzxtbh
Rdd=vzxtbh(Rs) 536

vzxthw
Rdd=vzxthw(Rs) 536

X

xor
if ([!]Pu[.new]) Rd=xor(Rs,Rt) 183
Pd=xor(Ps,Pt) 201
Rd=xor(Rs,Rt) 159
Rdd=xor(Rss,Rtt) 335
Rx[&|^]=xor(Rs,Rt) 337
Rxx^=xor(Rss,Rtt) 336

Z

zxtb
if ([!]Pu[.new]) Rd=zxtb(Rs) 189
Rd=zxtb(Rs) 171

zxth
if ([!]Pu[.new]) Rd=zxth(Rs) 189
Rd=zxth(Rs) 171

Intrinsics Index

A

abs
- Rd=abs(Rs)
  - Word32 Q6_R_abs_R(Word32 Rs) 326
- Rd=abs(Rs):sat
  - Word32 Q6_R_abs_R_sat(Word32 Rs) 326
- Rdd=abs(Rss)
  - Word64 Q6_P_abs_P(Word64 Rss) 325

add
- Rd=add(#u6,mpyi(Rs,#U6))
  - Word32 Q6_R_add_mpyi_IRI(Word32 Iu6, Word32 Rs, Word32 IU6) 466
- Rd=add(#u6,mpyi(Rs,Rt))
  - Word32 Q6_R_add_mpyi_IRR(Word32 Iu6, Word32 Rs, Word32 Rt) 466
- Rd=add(Rs,#s16)
  - Word32 Q6_R_add_RI(Word32 Rs, Word32 Is16) 157
- Rd=add(Rs,add(Ru,#s6))
  - Word32 Q6_R_add_add_RRI(Word32 Rs, Word32 Ru, Word32 Is6) 327
- Rd=add(Rs,Rt)
  - Word32 Q6_R_add_RR(Word32 Rs, Word32 Rt) 157
- Rd=add(Rs,Rt):sat
  - Word32 Q6_R_add_RR_sat(Word32 Rs, Word32 Rt) 157
- Rd=add(Rt.H,Rs.H):<<16
  - Word32 Q6_R_add_RhRh_s16(Word32 Rt, Word32 Rs) 331
- Rd=add(Rt.H,Rs.H):sat:<<16
  - Word32 Q6_R_add_RhRh_sat_s16(Word32 Rt, Word32 Rs) 332
- Rd=add(Rt.H,Rs.L):<<16
  - Word32 Q6_R_add_RhRl_s16(Word32 Rt, Word32 Rs) 332
- Rd=add(Rt.H,Rs.L):sat:<<16
  - Word32 Q6_R_add_RhRl_sat_s16(Word32 Rt, Word32 Rs) 332
- Rd=add(Rt.L,Rs.H)
  - Word32 Q6_R_add_RlRh(Word32 Rt, Word32 Rs) 332
- Rd=add(Rt.L,Rs.H):<<16
  - Word32 Q6_R_add_RlRh_s16(Word32 Rt, Word32 Rs) 332
- Rd=add(Rt.L,Rs.H):sat
  - Word32 Q6_R_add_RlRh_sat(Word32 Rt, Word32 Rs) 332
- Rd=add(Rt.L,Rs.H):sat:<<16
  - Word32 Q6_R_add_RlRh_sat_s16(Word32 Rt, Word32 Rs) 332
- Rd=add(Rt.L,Rs.L)
  - Word32 Q6_R_add_RlRl(Word32 Rt, Word32 Rs) 332
- Rd=add(Rt.L,Rs.L):<<16
  - Word32 Q6_R_add_RlRl_s16(Word32 Rt, Word32 Rs) 332
- Rd=add(Rt.L,Rs.L):sat
  - Word32 Q6_R_add_RlRl_sat(Word32 Rt, Word32 Rs) 332
- Rd=add(Rt.L,Rs.L):sat:<<16
  - Word32 Q6_R_add_RlRl_sat_s16(Word32 Rt, Word32 Rs) 332
- Rd=add(Ru,mpyi(#u6:2,Rs))
  - Word32 Q6_R_add_mpyi_RIR(Word32 Ru, Word32 Iu6_2, Word32 Rs) 466
- Rd=add(Ru,mpyi(Rs,#u6))
  - Word32 Q6_R_add_mpyi_RRI(Word32 Ru, Word32 Rs, Word32 Iu6) 466
- Rdd=add(Rs,Rtt)

```
        Word64 Q6_P_add_RP(Word32 Rs, Word64 Rtt) 329
    Rdd=add(Rss,Rtt)
        Word64 Q6_P_add_PP(Word64 Rss, Word64 Rtt) 329
    Rdd=add(Rss,Rtt):sat
        Word64 Q6_P_add_PP_sat(Word64 Rss, Word64 Rtt) 329
    Rx+=add(Rs,#s8)
        Word32 Q6_R_addacc_RI(Word32 Rx, Word32 Rs, Word32 Is8) 327
    Rx+=add(Rs,Rt)
        Word32 Q6_R_addacc_RR(Word32 Rx, Word32 Rs, Word32 Rt) 327
    Rx-=add(Rs,#s8)
        Word32 Q6_R_addnac_RI(Word32 Rx, Word32 Rs, Word32 Is8) 327
    Rx-=add(Rs,Rt)
        Word32 Q6_R_addnac_RR(Word32 Rx, Word32 Rs, Word32 Rt) 327
    Ry=add(Ru,mpyi(Ry,Rs))
        Word32 Q6_R_add_mpyi_RRR(Word32 Ru, Word32 Ry, Word32 Rs) 466

addasl
    Rd=addasl(Rt,Rs,#u3)
        Word32 Q6_R_addasl_RRI(Word32 Rt, Word32 Rs, Word32 Iu3) 562

all8
    Pd=all8(Ps)
        Byte Q6_p_all8_p(Byte Ps) 195

and
    Pd=and(Ps,and(Pt,!Pu))
        Byte Q6_p_and_and_ppnp(Byte Ps, Byte Pt, Byte Pu) 201
    Pd=and(Ps,and(Pt,Pu))
        Byte Q6_p_and_and_ppp(Byte Ps, Byte Pt, Byte Pu) 201
    Pd=and(Pt,!Ps)
        Byte Q6_p_and_pnp(Byte Pt, Byte Ps) 201
    Pd=and(Pt,Ps)
        Byte Q6_p_and_pp(Byte Pt, Byte Ps) 201
    Rd=and(Rs,#s10)
        Word32 Q6_R_and_RI(Word32 Rs, Word32 Is10) 159
    Rd=and(Rs,Rt)
        Word32 Q6_R_and_RR(Word32 Rs, Word32 Rt) 159
    Rd=and(Rt,~Rs)
        Word32 Q6_R_and_RnR(Word32 Rt, Word32 Rs) 159
    Rdd=and(Rss,Rtt)
        Word64 Q6_P_and_PP(Word64 Rss, Word64 Rtt) 335
    Rdd=and(Rtt,~Rss)
        Word64 Q6_P_and_PnP(Word64 Rtt, Word64 Rss) 335
    Rx&=and(Rs,~Rt)
        Word32 Q6_R_andand_RnR(Word32 Rx, Word32 Rs, Word32 Rt) 337
    Rx&=and(Rs,Rt)
        Word32 Q6_R_andand_RR(Word32 Rx, Word32 Rs, Word32 Rt) 337
    Rx^=and(Rs,~Rt)
        Word32 Q6_R_andxacc_RnR(Word32 Rx, Word32 Rs, Word32 Rt) 337
    Rx^=and(Rs,Rt)
        Word32 Q6_R_andxacc_RR(Word32 Rx, Word32 Rs, Word32 Rt) 337
    Rx|=and(Rs,#s10)
        Word32 Q6_R_andor_RI(Word32 Rx, Word32 Rs, Word32 Is10) 337
    Rx|=and(Rs,~Rt)
        Word32 Q6_R_andor_RnR(Word32 Rx, Word32 Rs, Word32 Rt) 337
    Rx|=and(Rs,Rt)
        Word32 Q6_R_andor_RR(Word32 Rx, Word32 Rs, Word32 Rt) 337
```

```
any8
    Pd=any8(Ps)
       Byte Q6_p_any8_p(Byte Ps) 195

asl
    Rd=asl(Rs,#u5)
       Word32 Q6_R_asl_RI(Word32 Rs, Word32 Iu5) 557
    Rd=asl(Rs,#u5):sat
       Word32 Q6_R_asl_RI_sat(Word32 Rs, Word32 Iu5) 569
    Rd=asl(Rs,Rt)
       Word32 Q6_R_asl_RR(Word32 Rs, Word32 Rt) 571
    Rd=asl(Rs,Rt):sat
       Word32 Q6_R_asl_RR_sat(Word32 Rs, Word32 Rt) 580
    Rdd=asl(Rss,#u6)
       Word64 Q6_P_asl_PI(Word64 Rss, Word32 Iu6) 557
    Rdd=asl(Rss,Rt)
       Word64 Q6_P_asl_PR(Word64 Rss, Word32 Rt) 571
    Rx&=asl(Rs,#u5)
       Word32 Q6_R_asland_RI(Word32 Rx, Word32 Rs, Word32 Iu5) 564
    Rx&=asl(Rs,Rt)
       Word32 Q6_R_asland_RR(Word32 Rx, Word32 Rs, Word32 Rt) 577
    Rx^=asl(Rs,#u5)
       Word32 Q6_R_aslxacc_RI(Word32 Rx, Word32 Rs, Word32 Iu5) 564
    Rx+=asl(Rs,#u5)
       Word32 Q6_R_aslacc_RI(Word32 Rx, Word32 Rs, Word32 Iu5) 560
    Rx+=asl(Rs,Rt)
       Word32 Q6_R_aslacc_RR(Word32 Rx, Word32 Rs, Word32 Rt) 574
    Rx=add(#u8,asl(Rx,#U5))
       Word32 Q6_R_add_asl_IRI(Word32 Iu8, Word32 Rx, Word32 IU5) 560
    Rx=and(#u8,asl(Rx,#U5))
       Word32 Q6_R_and_asl_IRI(Word32 Iu8, Word32 Rx, Word32 IU5) 564
    Rx-=asl(Rs,#u5)
       Word32 Q6_R_aslnac_RI(Word32 Rx, Word32 Rs, Word32 Iu5) 560
    Rx-=asl(Rs,Rt)
       Word32 Q6_R_aslnac_RR(Word32 Rx, Word32 Rs, Word32 Rt) 574
    Rx=or(#u8,asl(Rx,#U5))
       Word32 Q6_R_or_asl_IRI(Word32 Iu8, Word32 Rx, Word32 IU5) 564
    Rx=sub(#u8,asl(Rx,#U5))
       Word32 Q6_R_sub_asl_IRI(Word32 Iu8, Word32 Rx, Word32 IU5) 560
    Rx|=asl(Rs,#u5)
       Word32 Q6_R_aslor_RI(Word32 Rx, Word32 Rs, Word32 Iu5) 564
    Rx|=asl(Rs,Rt)
       Word32 Q6_R_aslor_RR(Word32 Rx, Word32 Rs, Word32 Rt) 577
    Rxx&=asl(Rss,#u6)
       Word64 Q6_P_asland_PI(Word64 Rxx, Word64 Rss, Word32 Iu6) 564
    Rxx&=asl(Rss,Rt)
       Word64 Q6_P_asland_PR(Word64 Rxx, Word64 Rss, Word32 Rt) 577
    Rxx^=asl(Rss,#u6)
       Word64 Q6_P_aslxacc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6) 565
    Rxx^=asl(Rss,Rt)
       Word64 Q6_P_aslxacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt) 578
    Rxx+=asl(Rss,#u6)
       Word64 Q6_P_aslacc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6) 560
    Rxx+=asl(Rss,Rt)
       Word64 Q6_P_aslacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt) 574
    Rxx-=asl(Rss,#u6)
       Word64 Q6_P_aslnac_PI(Word64 Rxx, Word64 Rss, Word32 Iu6) 560
    Rxx-=asl(Rss,Rt)
```

Word64 Q6_P_aslnac_PR(Word64 Rxx, Word64 Rss, Word32 Rt) 574
Rxx|=asl(Rss,#u6)
Word64 Q6_P_aslor_PI(Word64 Rxx, Word64 Rss, Word32 Iu6) 565
Rxx|=asl(Rss,Rt)
Word64 Q6_P_aslor_PR(Word64 Rxx, Word64 Rss, Word32 Rt) 578

aslh
Rd=aslh(Rs)
Word32 Q6_R_aslh_R(Word32 Rs) 176

asr
Rd=asr(Rs,#u5)
Word32 Q6_R_asr_RI(Word32 Rs, Word32 Iu5) 557
Rd=asr(Rs,#u5):rnd
Word32 Q6_R_asr_RI_rnd(Word32 Rs, Word32 Iu5) 568
Rd=asr(Rs,Rt)
Word32 Q6_R_asr_RR(Word32 Rs, Word32 Rt) 571
Rd=asr(Rs,Rt):sat
Word32 Q6_R_asr_RR_sat(Word32 Rs, Word32 Rt) 580
Rdd=asr(Rss,#u6)
Word64 Q6_P_asr_PI(Word64 Rss, Word32 Iu6) 557
Rdd=asr(Rss,#u6):rnd
Word64 Q6_P_asr_PI_rnd(Word64 Rss, Word32 Iu6) 568
Rdd=asr(Rss,Rt)
Word64 Q6_P_asr_PR(Word64 Rss, Word32 Rt) 571
Rx&=asr(Rs,#u5)
Word32 Q6_R_asrand_RI(Word32 Rx, Word32 Rs, Word32 Iu5) 564
Rx&=asr(Rs,Rt)
Word32 Q6_R_asrand_RR(Word32 Rx, Word32 Rs, Word32 Rt) 577
Rx+=asr(Rs,#u5)
Word32 Q6_R_asracc_RI(Word32 Rx, Word32 Rs, Word32 Iu5) 560
Rx+=asr(Rs,Rt)
Word32 Q6_R_asracc_RR(Word32 Rx, Word32 Rs, Word32 Rt) 574
Rx-=asr(Rs,#u5)
Word32 Q6_R_asrnac_RI(Word32 Rx, Word32 Rs, Word32 Iu5) 560
Rx-=asr(Rs,Rt)
Word32 Q6_R_asrnac_RR(Word32 Rx, Word32 Rs, Word32 Rt) 574
Rx|=asr(Rs,#u5)
Word32 Q6_R_asror_RI(Word32 Rx, Word32 Rs, Word32 Iu5) 564
Rx|=asr(Rs,Rt)
Word32 Q6_R_asror_RR(Word32 Rx, Word32 Rs, Word32 Rt) 577
Rxx&=asr(Rss,#u6)
Word64 Q6_P_asrand_PI(Word64 Rxx, Word64 Rss, Word32 Iu6) 565
Rxx&=asr(Rss,Rt)
Word64 Q6_P_asrand_PR(Word64 Rxx, Word64 Rss, Word32 Rt) 577
Rxx^=asr(Rss,Rt)
Word64 Q6_P_asrxacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt) 578
Rxx+=asr(Rss,#u6)
Word64 Q6_P_asracc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6) 560
Rxx+=asr(Rss,Rt)
Word64 Q6_P_asracc_PR(Word64 Rxx, Word64 Rss, Word32 Rt) 574
Rxx-=asr(Rss,#u6)
Word64 Q6_P_asrnac_PI(Word64 Rxx, Word64 Rss, Word32 Iu6) 560
Rxx-=asr(Rss,Rt)
Word64 Q6_P_asrnac_PR(Word64 Rxx, Word64 Rss, Word32 Rt) 574
Rxx|=asr(Rss,#u6)
Word64 Q6_P_asror_PI(Word64 Rxx, Word64 Rss, Word32 Iu6) 565
Rxx|=asr(Rss,Rt)
Word64 Q6_P_asror_PR(Word64 Rxx, Word64 Rss, Word32 Rt) 578

asrh
Rd=asrh(Rs)
Word32 Q6_R_asrh_R(Word32 Rs) 176

asrrnd
Rd=asrrnd(Rs,#u5)
Word32 Q6_R_asrrnd_RI(Word32 Rs, Word32 Iu5) 568
Rdd=asrrnd(Rss,#u6)
Word64 Q6_P_asrrnd_PI(Word64 Rss, Word32 Iu6) 568

B

bitsclr
Pd=!bitsclr(Rs,#u6)
Byte Q6_p_not_bitsclr_RI(Word32 Rs, Word32 Iu6) 542
Pd=!bitsclr(Rs,Rt)
Byte Q6_p_not_bitsclr_RR(Word32 Rs, Word32 Rt) 542
Pd=bitsclr(Rs,#u6)
Byte Q6_p_bitsclr_RI(Word32 Rs, Word32 Iu6) 542
Pd=bitsclr(Rs,Rt)
Byte Q6_p_bitsclr_RR(Word32 Rs, Word32 Rt) 542

bitsplit
Rdd=bitsplit(Rs,#u5)
Word64 Q6_P_bitsplit_RI(Word32 Rs, Word32 Iu5) 410
Rdd=bitsplit(Rs,Rt)
Word64 Q6_P_bitsplit_RR(Word32 Rs, Word32 Rt) 410

bitsset
Pd=!bitsset(Rs,Rt)
Byte Q6_p_not_bitsset_RR(Word32 Rs, Word32 Rt) 542
Pd=bitsset(Rs,Rt)
Byte Q6_p_bitsset_RR(Word32 Rs, Word32 Rt) 542

boundscheck
Pd=boundscheck(Rs,Rtt)
Byte Q6_p_boundscheck_RP(Word32 Rs, Word64 Rtt) 537

brev
Rd=brev(Rs)
Word32 Q6_R_brev_R(Word32 Rs) 407
Rdd=brev(Rss)
Word64 Q6_P_brev_P(Word64 Rss) 407

C

cl0
Rd=cl0(Rs)
Word32 Q6_R_cl0_R(Word32 Rs) 396
Rd=cl0(Rss)
Word32 Q6_R_cl0_P(Word64 Rss) 396

cl1
Rd=cl1(Rs)
Word32 Q6_R_cl1_R(Word32 Rs) 396
Rd=cl1(Rss)
Word32 Q6_R_cl1_P(Word64 Rss) 396

clb
Rd=add(clb(Rs),#s6)

```
        Word32 Q6_R_add_clb_RI(Word32 Rs, Word32 Is6) 396
    Rd=add(clb(Rss),#s6)
        Word32 Q6_R_add_clb_PI(Word64 Rss, Word32 Is6) 396
    Rd=clb(Rs)
        Word32 Q6_R_clb_R(Word32 Rs) 396
    Rd=clb(Rss)
        Word32 Q6_R_clb_P(Word64 Rss) 396

clip
    Rd=clip(Rs,#u5)
        Word32 Q6_R_clip_RI(Word32 Rs, Word32 Iu5) 334

clrbit
    Rd=clrbit(Rs,#u5)
        Word32 Q6_R_clrbit_RI(Word32 Rs, Word32 Iu5) 408
    Rd=clrbit(Rs,Rt)
        Word32 Q6_R_clrbit_RR(Word32 Rs, Word32 Rt) 408

cmp.eq
    Pd=!cmp.eq(Rs,#s10)
        Byte Q6_p_not_cmp_eq_RI(Word32 Rs, Word32 Is10) 190
    Pd=!cmp.eq(Rs,Rt)
        Byte Q6_p_not_cmp_eq_RR(Word32 Rs, Word32 Rt) 190
    Pd=cmp.eq(Rs,#s10)
        Byte Q6_p_cmp_eq_RI(Word32 Rs, Word32 Is10) 190
    Pd=cmp.eq(Rs,Rt)
        Byte Q6_p_cmp_eq_RR(Word32 Rs, Word32 Rt) 190
    Pd=cmp.eq(Rss,Rtt)
        Byte Q6_p_cmp_eq_PP(Word64 Rss, Word64 Rtt) 541
    Rd=!cmp.eq(Rs,#s8)
        Word32 Q6_R_not_cmp_eq_RI(Word32 Rs, Word32 Is8) 192
    Rd=!cmp.eq(Rs,Rt)
        Word32 Q6_R_not_cmp_eq_RR(Word32 Rs, Word32 Rt) 192
    Rd=cmp.eq(Rs,#s8)
        Word32 Q6_R_cmp_eq_RI(Word32 Rs, Word32 Is8) 192
    Rd=cmp.eq(Rs,Rt)
        Word32 Q6_R_cmp_eq_RR(Word32 Rs, Word32 Rt) 192

cmp.ge
    Pd=cmp.ge(Rs,#s8)
        Byte Q6_p_cmp_ge_RI(Word32 Rs, Word32 Is8) 190

cmp.geu
    Pd=cmp.geu(Rs,#u8)
        Byte Q6_p_cmp_geu_RI(Word32 Rs, Word32 Iu8) 190

cmp.gt
    Pd=!cmp.gt(Rs,#s10)
        Byte Q6_p_not_cmp_gt_RI(Word32 Rs, Word32 Is10) 190
    Pd=!cmp.gt(Rs,Rt)
        Byte Q6_p_not_cmp_gt_RR(Word32 Rs, Word32 Rt) 190
    Pd=cmp.gt(Rs,#s10)
        Byte Q6_p_cmp_gt_RI(Word32 Rs, Word32 Is10) 190
    Pd=cmp.gt(Rs,Rt)
        Byte Q6_p_cmp_gt_RR(Word32 Rs, Word32 Rt) 191
    Pd=cmp.gt(Rss,Rtt)
        Byte Q6_p_cmp_gt_PP(Word64 Rss, Word64 Rtt) 541
```

```
cmp.gtu
    Pd=!cmp.gtu(Rs,#u9)
        Byte Q6_p_not_cmp_gtu_RI(Word32 Rs, Word32 Iu9) 190
    Pd=!cmp.gtu(Rs,Rt)
        Byte Q6_p_not_cmp_gtu_RR(Word32 Rs, Word32 Rt) 190
    Pd=cmp.gtu(Rs,#u9)
        Byte Q6_p_cmp_gtu_RI(Word32 Rs, Word32 Iu9) 191
    Pd=cmp.gtu(Rs,Rt)
        Byte Q6_p_cmp_gtu_RR(Word32 Rs, Word32 Rt) 191
    Pd=cmp.gtu(Rss,Rtt)
        Byte Q6_p_cmp_gtu_PP(Word64 Rss, Word64 Rtt) 541

cmp.lt
    Pd=cmp.lt(Rs,Rt)
        Byte Q6_p_cmp_lt_RR(Word32 Rs, Word32 Rt) 191

cmp.ltu
    Pd=cmp.ltu(Rs,Rt)
        Byte Q6_p_cmp_ltu_RR(Word32 Rs, Word32 Rt) 191

cmpb.eq
    Pd=cmpb.eq(Rs,#u8)
        Byte Q6_p_cmpb_eq_RI(Word32 Rs, Word32 Iu8) 538
    Pd=cmpb.eq(Rs,Rt)
        Byte Q6_p_cmpb_eq_RR(Word32 Rs, Word32 Rt) 538

cmpb.gt
    Pd=cmpb.gt(Rs,#s8)
        Byte Q6_p_cmpb_gt_RI(Word32 Rs, Word32 Is8) 538
    Pd=cmpb.gt(Rs,Rt)
        Byte Q6_p_cmpb_gt_RR(Word32 Rs, Word32 Rt) 538

cmpb.gtu
    Pd=cmpb.gtu(Rs,#u7)
        Byte Q6_p_cmpb_gtu_RI(Word32 Rs, Word32 Iu7) 538
    Pd=cmpb.gtu(Rs,Rt)
        Byte Q6_p_cmpb_gtu_RR(Word32 Rs, Word32 Rt) 538

cmph.eq
    Pd=cmph.eq(Rs,#s8)
        Byte Q6_p_cmph_eq_RI(Word32 Rs, Word32 Is8) 539
    Pd=cmph.eq(Rs,Rt)
        Byte Q6_p_cmph_eq_RR(Word32 Rs, Word32 Rt) 539

cmph.gt
    Pd=cmph.gt(Rs,#s8)
        Byte Q6_p_cmph_gt_RI(Word32 Rs, Word32 Is8) 539
    Pd=cmph.gt(Rs,Rt)
        Byte Q6_p_cmph_gt_RR(Word32 Rs, Word32 Rt) 539

cmph.gtu
    Pd=cmph.gtu(Rs,#u7)
        Byte Q6_p_cmph_gtu_RI(Word32 Rs, Word32 Iu7) 539
    Pd=cmph.gtu(Rs,Rt)
        Byte Q6_p_cmph_gtu_RR(Word32 Rs, Word32 Rt) 539

cmpy
    Rd=cmpy(Rs,Rt):<<1:rnd:sat
```

Word32 Q6_R_cmpy_RR_s1_rnd_sat(Word32 Rs, Word32 Rt) 424
Rd=cmpy(Rs,Rt):rnd:sat
Word32 Q6_R_cmpy_RR_rnd_sat(Word32 Rs, Word32 Rt) 424
Rd=cmpy(Rs,Rt*):<<1:rnd:sat
Word32 Q6_R_cmpy_RR_conj_s1_rnd_sat(Word32 Rs, Word32 Rt) 424
Rd=cmpy(Rs,Rt*):rnd:sat
Word32 Q6_R_cmpy_RR_conj_rnd_sat(Word32 Rs, Word32 Rt) 424
Rdd=cmpy(Rs,Rt):<<1:sat
Word64 Q6_P_cmpy_RR_s1_sat(Word32 Rs, Word32 Rt) 419
Rdd=cmpy(Rs,Rt):sat
Word64 Q6_P_cmpy_RR_sat(Word32 Rs, Word32 Rt) 419
Rdd=cmpy(Rs,Rt*):<<1:sat
Word64 Q6_P_cmpy_RR_conj_s1_sat(Word32 Rs, Word32 Rt) 419
Rdd=cmpy(Rs,Rt*):sat
Word64 Q6_P_cmpy_RR_conj_sat(Word32 Rs, Word32 Rt) 419
Rxx+=cmpy(Rs,Rt):<<1:sat
Word64 Q6_P_cmpyacc_RR_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt) 419
Rxx+=cmpy(Rs,Rt):sat
Word64 Q6_P_cmpyacc_RR_sat(Word64 Rxx, Word32 Rs, Word32 Rt) 419
Rxx+=cmpy(Rs,Rt*):<<1:sat
Word64 Q6_P_cmpyacc_RR_conj_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt) 419
Rxx+=cmpy(Rs,Rt*):sat
Word64 Q6_P_cmpyacc_RR_conj_sat(Word64 Rxx, Word32 Rs, Word32 Rt) 419
Rxx-=cmpy(Rs,Rt):<<1:sat
Word64 Q6_P_cmpynac_RR_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt) 419
Rxx-=cmpy(Rs,Rt):sat
Word64 Q6_P_cmpynac_RR_sat(Word64 Rxx, Word32 Rs, Word32 Rt) 419
Rxx-=cmpy(Rs,Rt*):<<1:sat
Word64 Q6_P_cmpynac_RR_conj_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt) 420
Rxx-=cmpy(Rs,Rt*):sat
Word64 Q6_P_cmpynac_RR_conj_sat(Word64 Rxx, Word32 Rs, Word32 Rt) 420

cmpyi
Rdd=cmpyi(Rs,Rt)
Word64 Q6_P_cmpyi_RR(Word32 Rs, Word32 Rt) 421
Rxx+=cmpyi(Rs,Rt)
Word64 Q6_P_cmpyiacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt) 421

cmpyiw
Rd=cmpyiw(Rss,Rtt):<<1:rnd:sat
Word32 Q6_R_cmpyiw_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt) 429
Rd=cmpyiw(Rss,Rtt):<<1:sat
Word32 Q6_R_cmpyiw_PP_s1_sat(Word64 Rss, Word64 Rtt) 429
Rd=cmpyiw(Rss,Rtt*):<<1:rnd:sat
Word32 Q6_R_cmpyiw_PP_conj_s1_rnd_sat(Word64 Rss, Word64 Rtt) 429
Rd=cmpyiw(Rss,Rtt*):<<1:sat
Word32 Q6_R_cmpyiw_PP_conj_s1_sat(Word64 Rss, Word64 Rtt) 429
Rdd=cmpyiw(Rss,Rtt)
Word64 Q6_P_cmpyiw_PP(Word64 Rss, Word64 Rtt) 429
Rdd=cmpyiw(Rss,Rtt*)
Word64 Q6_P_cmpyiw_PP_conj(Word64 Rss, Word64 Rtt) 429
Rxx+=cmpyiw(Rss,Rtt)
Word64 Q6_P_cmpyiwacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt) 429
Rxx+=cmpyiw(Rss,Rtt*)
Word64 Q6_P_cmpyiwacc_PP_conj(Word64 Rxx, Word64 Rss, Word64 Rtt) 429

cmpyiwh
Rd=cmpyiwh(Rss,Rt):<<1:rnd:sat

```
        Word32 Q6_R_cmpyiwh_PR_s1_rnd_sat(Word64 Rss, Word32 Rt) 426
    Rd=cmpyiwh(Rss,Rt*):<<1:rnd:sat
        Word32 Q6_R_cmpyiwh_PR_conj_s1_rnd_sat(Word64 Rss, Word32 Rt) 426

cmpyr
    Rdd=cmpyr(Rs,Rt)
        Word64 Q6_P_cmpyr_RR(Word32 Rs, Word32 Rt) 421
    Rxx+=cmpyr(Rs,Rt)
        Word64 Q6_P_cmpyracc_RR(Word64 Rxx, Word32 Rs, Word32 Rt) 421

cmpyrw
    Rd=cmpyrw(Rss,Rtt):<<1:rnd:sat
        Word32 Q6_R_cmpyrw_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt) 429
    Rd=cmpyrw(Rss,Rtt):<<1:sat
        Word32 Q6_R_cmpyrw_PP_s1_sat(Word64 Rss, Word64 Rtt) 429
    Rd=cmpyrw(Rss,Rtt*):<<1:rnd:sat
        Word32 Q6_R_cmpyrw_PP_conj_s1_rnd_sat(Word64 Rss, Word64 Rtt) 429
    Rd=cmpyrw(Rss,Rtt*):<<1:sat
        Word32 Q6_R_cmpyrw_PP_conj_s1_sat(Word64 Rss, Word64 Rtt) 429
    Rdd=cmpyrw(Rss,Rtt)
        Word64 Q6_P_cmpyrw_PP(Word64 Rss, Word64 Rtt) 429
    Rdd=cmpyrw(Rss,Rtt*)
        Word64 Q6_P_cmpyrw_PP_conj(Word64 Rss, Word64 Rtt) 429
    Rxx+=cmpyrw(Rss,Rtt)
        Word64 Q6_P_cmpyrwacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt) 429
    Rxx+=cmpyrw(Rss,Rtt*)
        Word64 Q6_P_cmpyrwacc_PP_conj(Word64 Rxx, Word64 Rss, Word64 Rtt) 429

cmpyrwh
    Rd=cmpyrwh(Rss,Rt):<<1:rnd:sat
        Word32 Q6_R_cmpyrwh_PR_s1_rnd_sat(Word64 Rss, Word32 Rt) 426
    Rd=cmpyrwh(Rss,Rt*):<<1:rnd:sat
        Word32 Q6_R_cmpyrwh_PR_conj_s1_rnd_sat(Word64 Rss, Word32 Rt) 426

combine
    Rd=combine(Rt.H,Rs.H)
        Word32 Q6_R_combine_RhRh(Word32 Rt, Word32 Rs) 173
    Rd=combine(Rt.H,Rs.L)
        Word32 Q6_R_combine_RhRl(Word32 Rt, Word32 Rs) 173
    Rd=combine(Rt.L,Rs.H)
        Word32 Q6_R_combine_RlRh(Word32 Rt, Word32 Rs) 173
    Rd=combine(Rt.L,Rs.L)
        Word32 Q6_R_combine_RlRl(Word32 Rt, Word32 Rs) 173
    Rdd=combine(#s8,#S8)
        Word64 Q6_P_combine_II(Word32 Is8, Word32 IS8) 173
    Rdd=combine(#s8,Rs)
        Word64 Q6_P_combine_IR(Word32 Is8, Word32 Rs) 173
    Rdd=combine(Rs,#s8)
        Word64 Q6_P_combine_RI(Word32 Rs, Word32 Is8) 173
    Rdd=combine(Rs,Rt)
        Word64 Q6_P_combine_RR(Word32 Rs, Word32 Rt) 173

convert_d2df
    Rdd=convert_d2df(Rss)
        Word64 Q6_P_convert_d2df_P(Word64 Rss) 449

convert_d2sf
    Rd=convert_d2sf(Rss)
        Word32 Q6_R_convert_d2sf_P(Word64 Rss) 449
```

```
convert_df2d
    Rdd=convert_df2d(Rss)
        Word64 Q6_P_convert_df2d_P(Word64 Rss) 452
    Rdd=convert_df2d(Rss):chop
        Word64 Q6_P_convert_df2d_P_chop(Word64 Rss) 452

convert_df2sf
    Rd=convert_df2sf(Rss)
        Word32 Q6_R_convert_df2sf_P(Word64 Rss) 448

convert_df2ud
    Rdd=convert_df2ud(Rss)
        Word64 Q6_P_convert_df2ud_P(Word64 Rss) 452
    Rdd=convert_df2ud(Rss):chop
        Word64 Q6_P_convert_df2ud_P_chop(Word64 Rss) 452

convert_df2uw
    Rd=convert_df2uw(Rss)
        Word32 Q6_R_convert_df2uw_P(Word64 Rss) 451
    Rd=convert_df2uw(Rss):chop
        Word32 Q6_R_convert_df2uw_P_chop(Word64 Rss) 451

convert_df2w
    Rd=convert_df2w(Rss)
        Word32 Q6_R_convert_df2w_P(Word64 Rss) 451
    Rd=convert_df2w(Rss):chop
        Word32 Q6_R_convert_df2w_P_chop(Word64 Rss) 451

convert_sf2d
    Rdd=convert_sf2d(Rs)
        Word64 Q6_P_convert_sf2d_R(Word32 Rs) 452
    Rdd=convert_sf2d(Rs):chop
        Word64 Q6_P_convert_sf2d_R_chop(Word32 Rs) 452

convert_sf2df
    Rdd=convert_sf2df(Rs)
        Word64 Q6_P_convert_sf2df_R(Word32 Rs) 448

convert_sf2ud
    Rdd=convert_sf2ud(Rs)
        Word64 Q6_P_convert_sf2ud_R(Word32 Rs) 452
    Rdd=convert_sf2ud(Rs):chop
        Word64 Q6_P_convert_sf2ud_R_chop(Word32 Rs) 452

convert_sf2uw
    Rd=convert_sf2uw(Rs)
        Word32 Q6_R_convert_sf2uw_R(Word32 Rs) 451
    Rd=convert_sf2uw(Rs):chop
        Word32 Q6_R_convert_sf2uw_R_chop(Word32 Rs) 451

convert_sf2w
    Rd=convert_sf2w(Rs)
        Word32 Q6_R_convert_sf2w_R(Word32 Rs) 452
    Rd=convert_sf2w(Rs):chop
        Word32 Q6_R_convert_sf2w_R_chop(Word32 Rs) 452

convert_ud2df
    Rdd=convert_ud2df(Rss)
        Word64 Q6_P_convert_ud2df_P(Word64 Rss) 449
```

convert_ud2sf
Rd=convert_ud2sf(Rss)
Word32 Q6_R_convert_ud2sf_P(Word64 Rss) 449

convert_uw2df
Rdd=convert_uw2df(Rs)
Word64 Q6_P_convert_uw2df_R(Word32 Rs) 449

convert_uw2sf
Rd=convert_uw2sf(Rs)
Word32 Q6_R_convert_uw2sf_R(Word32 Rs) 449

convert_w2df
Rdd=convert_w2df(Rs)
Word64 Q6_P_convert_w2df_R(Word32 Rs) 449

convert_w2sf
Rd=convert_w2sf(Rs)
Word32 Q6_R_convert_w2sf_R(Word32 Rs) 449

cround
Rd=cround(Rs,#u5)
Word32 Q6_R_cround_RI(Word32 Rs, Word32 Iu5) 346
Rd=cround(Rs,Rt)
Word32 Q6_R_cround_RR(Word32 Rs, Word32 Rt) 346
Rdd=cround(Rss,#u6)
Word64 Q6_P_cround_PI(Word64 Rss, Word32 Iu6) 346
Rdd=cround(Rss,Rt)
Word64 Q6_P_cround_PR(Word64 Rss, Word32 Rt) 346

ct0
Rd=ct0(Rs)
Word32 Q6_R_ct0_R(Word32 Rs) 398
Rd=ct0(Rss)
Word32 Q6_R_ct0_P(Word64 Rss) 398

ct1
Rd=ct1(Rs)
Word32 Q6_R_ct1_R(Word32 Rs) 398
Rd=ct1(Rss)
Word32 Q6_R_ct1_P(Word64 Rss) 398

D

dccleana
dccleana(Rs)
void Q6_dccleana_A(Address a) 312

dccleaninva
dccleaninva(Rs)
void Q6_dccleaninva_A(Address a) 312

dcfetch
dcfetch(Rs)
void Q6_dcfetch_A(Address a) 310

dcinva
dcinva(Rs)
void Q6_dcinva_A(Address a) 312

dczeroa
dczeroa(Rs)
void Q6_dczeroa_A(Address a) 307

deinterleave
Rdd=deinterleave(Rss)
Word64 Q6_P_deinterleave_P(Word64 Rss) 404

dfadd
Rdd=dfadd(Rss,Rtt)
Word64 Q6_P_dfadd_PP(Word64 Rss, Word64 Rtt) 444

dfclass
Pd=dfclass(Rss,#u5)
Byte Q6_p_dfclass_PI(Word64 Rss, Word32 Iu5) 445

dfcmp.eq
Pd=dfcmp.eq(Rss,Rtt)
Byte Q6_p_dfcmp_eq_PP(Word64 Rss, Word64 Rtt) 446

dfcmp.ge
Pd=dfcmp.ge(Rss,Rtt)
Byte Q6_p_dfcmp_ge_PP(Word64 Rss, Word64 Rtt) 446

dfcmp.gt
Pd=dfcmp.gt(Rss,Rtt)
Byte Q6_p_dfcmp_gt_PP(Word64 Rss, Word64 Rtt) 446

dfcmp.uo
Pd=dfcmp.uo(Rss,Rtt)
Byte Q6_p_dfcmp_uo_PP(Word64 Rss, Word64 Rtt) 446

dfmake
Rdd=dfmake(#u10):neg
Word64 Q6_P_dfmake_I_neg(Word32 Iu10) 459
Rdd=dfmake(#u10):pos
Word64 Q6_P_dfmake_I_pos(Word32 Iu10) 459

dfmax
Rdd=dfmax(Rss,Rtt)
Word64 Q6_P_dfmax_PP(Word64 Rss, Word64 Rtt) 460

dfmin
Rdd=dfmin(Rss,Rtt)
Word64 Q6_P_dfmin_PP(Word64 Rss, Word64 Rtt) 461

dfmpyfix
Rdd=dfmpyfix(Rss,Rtt)
Word64 Q6_P_dfmpyfix_PP(Word64 Rss, Word64 Rtt) 462

dfmpyhh
Rxx+=dfmpyhh(Rss,Rtt)
Word64 Q6_P_dfmpyhhacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt) 454

dfmpylh
Rxx+=dfmpylh(Rss,Rtt)
Word64 Q6_P_dfmpylhacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt) 454

dfmpyll
Rdd=dfmpyll(Rss,Rtt)

Word64 Q6_P_dfmpyll_PP(Word64 Rss, Word64 Rtt) 462

dfsub
Rdd=dfsub(Rss,Rtt)
Word64 Q6_P_dfsub_PP(Word64 Rss, Word64 Rtt) 464

dmsyncht
Rd=dmsyncht
Word32 Q6_R_dmsyncht() 321

E

extract
Rd=extract(Rs,#u5,#U5)
Word32 Q6_R_extract_RII(Word32 Rs, Word32 Iu5, Word32 IU5) 400
Rd=extract(Rs,Rtt)
Word32 Q6_R_extract_RP(Word32 Rs, Word64 Rtt) 400
Rdd=extract(Rss,#u6,#U6)
Word64 Q6_P_extract_PII(Word64 Rss, Word32 Iu6, Word32 IU6) 400
Rdd=extract(Rss,Rtt)
Word64 Q6_P_extract_PP(Word64 Rss, Word64 Rtt) 400

extractu
Rd=extractu(Rs,#u5,#U5)
Word32 Q6_R_extractu_RII(Word32 Rs, Word32 Iu5, Word32 IU5) 400
Rd=extractu(Rs,Rtt)
Word32 Q6_R_extractu_RP(Word32 Rs, Word64 Rtt) 400
Rdd=extractu(Rss,#u6,#U6)
Word64 Q6_P_extractu_PII(Word64 Rss, Word32 Iu6, Word32 IU6) 400
Rdd=extractu(Rss,Rtt)
Word64 Q6_P_extractu_PP(Word64 Rss, Word64 Rtt) 400

F

fastcorner9
Pd=!fastcorner9(Ps,Pt)
Byte Q6_p_not_fastcorner9_pp(Byte Ps, Byte Pt) 194
Pd=fastcorner9(Ps,Pt)
Byte Q6_p_fastcorner9_pp(Byte Ps, Byte Pt) 194

I

insert
Rx=insert(Rs,#u5,#U5)
Word32 Q6_R_insert_RII(Word32 Rx, Word32 Rs, Word32 Iu5, Word32 IU5) 403
Rx=insert(Rs,Rtt)
Word32 Q6_R_insert_RP(Word32 Rx, Word32 Rs, Word64 Rtt) 403
Rxx=insert(Rss,#u6,#U6)
Word64 Q6_P_insert_PII(Word64 Rxx, Word64 Rss, Word32 Iu6, Word32 IU6) 403
Rxx=insert(Rss,Rtt)
Word64 Q6_P_insert_PP(Word64 Rxx, Word64 Rss, Word64 Rtt) 403

interleave
Rdd=interleave(Rss)
Word64 Q6_P_interleave_P(Word64 Rss) 404

L

l2fetch
l2fetch(Rs,Rt)

```
      void Q6_l2fetch_AR(Address a, Word32 Rt) 318
   l2fetch(Rs,Rtt)
      void Q6_l2fetch_AP(Address a, Word64 Rtt) 318

lfs
   Rdd=lfs(Rss,Rtt)
      Word64 Q6_P_lfs_PP(Word64 Rss, Word64 Rtt) 405

lsl
   Rd=lsl(#s6,Rt)
      Word32 Q6_R_lsl_IR(Word32 Is6, Word32 Rt) 571
   Rd=lsl(Rs,Rt)
      Word32 Q6_R_lsl_RR(Word32 Rs, Word32 Rt) 571
   Rdd=lsl(Rss,Rt)
      Word64 Q6_P_lsl_PR(Word64 Rss, Word32 Rt) 571
   Rx&=lsl(Rs,Rt)
      Word32 Q6_R_lsland_RR(Word32 Rx, Word32 Rs, Word32 Rt) 577
   Rx+=lsl(Rs,Rt)
      Word32 Q6_R_lslacc_RR(Word32 Rx, Word32 Rs, Word32 Rt) 574
   Rx-=lsl(Rs,Rt)
      Word32 Q6_R_lslnac_RR(Word32 Rx, Word32 Rs, Word32 Rt) 574
   Rx|=lsl(Rs,Rt)
      Word32 Q6_R_lslor_RR(Word32 Rx, Word32 Rs, Word32 Rt) 577
   Rxx&=lsl(Rss,Rt)
      Word64 Q6_P_lsland_PR(Word64 Rxx, Word64 Rss, Word32 Rt) 577
   Rxx^=lsl(Rss,Rt)
      Word64 Q6_P_lslxacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt) 578
   Rxx+=lsl(Rss,Rt)
      Word64 Q6_P_lslacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt) 574
   Rxx-=lsl(Rss,Rt)
      Word64 Q6_P_lslnac_PR(Word64 Rxx, Word64 Rss, Word32 Rt) 574
   Rxx|=lsl(Rss,Rt)
      Word64 Q6_P_lslor_PR(Word64 Rxx, Word64 Rss, Word32 Rt) 578

lsr
   Rd=lsr(Rs,#u5)
      Word32 Q6_R_lsr_RI(Word32 Rs, Word32 Iu5) 557
   Rd=lsr(Rs,Rt)
      Word32 Q6_R_lsr_RR(Word32 Rs, Word32 Rt) 571
   Rdd=lsr(Rss,#u6)
      Word64 Q6_P_lsr_PI(Word64 Rss, Word32 Iu6) 557
   Rdd=lsr(Rss,Rt)
      Word64 Q6_P_lsr_PR(Word64 Rss, Word32 Rt) 571
   Rx&=lsr(Rs,#u5)
      Word32 Q6_R_lsrand_RI(Word32 Rx, Word32 Rs, Word32 Iu5) 564
   Rx&=lsr(Rs,Rt)
      Word32 Q6_R_lsrand_RR(Word32 Rx, Word32 Rs, Word32 Rt) 577
   Rx^=lsr(Rs,#u5)
      Word32 Q6_R_lsrxacc_RI(Word32 Rx, Word32 Rs, Word32 Iu5) 564
   Rx+=lsr(Rs,#u5)
      Word32 Q6_R_lsracc_RI(Word32 Rx, Word32 Rs, Word32 Iu5) 560
   Rx+=lsr(Rs,Rt)
      Word32 Q6_R_lsracc_RR(Word32 Rx, Word32 Rs, Word32 Rt) 574
   Rx=add(#u8,lsr(Rx,#U5))
      Word32 Q6_R_add_lsr_IRI(Word32 Iu8, Word32 Rx, Word32 IU5) 560
   Rx=and(#u8,lsr(Rx,#U5))
      Word32 Q6_R_and_lsr_IRI(Word32 Iu8, Word32 Rx, Word32 IU5) 564
   Rx-=lsr(Rs,#u5)
```

Word32 Q6_R_lsrnac_RI(Word32 Rx, Word32 Rs, Word32 Iu5) 560
Rx-=lsr(Rs,Rt)
Word32 Q6_R_lsrnac_RR(Word32 Rx, Word32 Rs, Word32 Rt) 574
Rx=or(#u8,lsr(Rx,#U5))
Word32 Q6_R_or_lsr_IRI(Word32 Iu8, Word32 Rx, Word32 IU5) 564
Rx=sub(#u8,lsr(Rx,#U5))
Word32 Q6_R_sub_lsr_IRI(Word32 Iu8, Word32 Rx, Word32 IU5) 560
Rx|=lsr(Rs,#u5)
Word32 Q6_R_lsror_RI(Word32 Rx, Word32 Rs, Word32 Iu5) 564
Rx|=lsr(Rs,Rt)
Word32 Q6_R_lsror_RR(Word32 Rx, Word32 Rs, Word32 Rt) 577
Rxx&=lsr(Rss,#u6)
Word64 Q6_P_lsrand_PI(Word64 Rxx, Word64 Rss, Word32 Iu6) 565
Rxx&=lsr(Rss,Rt)
Word64 Q6_P_lsrand_PR(Word64 Rxx, Word64 Rss, Word32 Rt) 577
Rxx^=lsr(Rss,#u6)
Word64 Q6_P_lsrxacc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6) 565
Rxx^=lsr(Rss,Rt)
Word64 Q6_P_lsrxacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt) 578
Rxx+=lsr(Rss,#u6)
Word64 Q6_P_lsracc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6) 560
Rxx+=lsr(Rss,Rt)
Word64 Q6_P_lsracc_PR(Word64 Rxx, Word64 Rss, Word32 Rt) 574
Rxx-=lsr(Rss,#u6)
Word64 Q6_P_lsrnac_PI(Word64 Rxx, Word64 Rss, Word32 Iu6) 560
Rxx-=lsr(Rss,Rt)
Word64 Q6_P_lsrnac_PR(Word64 Rxx, Word64 Rss, Word32 Rt) 574
Rxx|=lsr(Rss,#u6)
Word64 Q6_P_lsror_PI(Word64 Rxx, Word64 Rss, Word32 Iu6) 565
Rxx|=lsr(Rss,Rt)
Word64 Q6_P_lsror_PR(Word64 Rxx, Word64 Rss, Word32 Rt) 578

M

mask
Rd=mask(#u5,#U5)
Word32 Q6_R_mask_II(Word32 Iu5, Word32 IU5) 556
Rdd=mask(Pt)
Word64 Q6_P_mask_p(Byte Pt) 543

max
Rd=max(Rs,Rt)
Word32 Q6_R_max_RR(Word32 Rs, Word32 Rt) 339
Rdd=max(Rss,Rtt)
Word64 Q6_P_max_PP(Word64 Rss, Word64 Rtt) 340

maxu
Rd=maxu(Rs,Rt)
UWord32 Q6_R_maxu_RR(Word32 Rs, Word32 Rt) 339
Rdd=maxu(Rss,Rtt)
UWord64 Q6_P_maxu_PP(Word64 Rss, Word64 Rtt) 340

memb
memb(Rx++#s4:0:circ(Mu))=Rt
void Q6_memb_IMR_circ(void** StartAddress, Word32 Is4_0, Word32 Mu, Word32 Rt, void* BaseAddress) 287
memb(Rx++I:circ(Mu))=Rt
void Q6_memb_MR_circ(void** StartAddress, Word32 Mu, Word32 Rt, void*

```
            BaseAddress) 287
    Rd=memb(Rx++#s4:0:circ(Mu))
        Word32 Q6_R_memb_IM_circ(void** StartAddress, Word32 Is4_0, Word32 Mu,
            void* BaseAddress) 225
    Rd=memb(Rx++I:circ(Mu))
        Word32 Q6_R_memb_M_circ(void** StartAddress, Word32 Mu, void* BaseAddress)
            225

memd
    memd(Rx++#s4:3:circ(Mu))=Rtt
        void Q6_memd_IMP_circ(void** StartAddress, Word32 Is4_3, Word32 Mu, Word64
            Rtt, void* BaseAddress) 283
    memd(Rx++I:circ(Mu))=Rtt
        void Q6_memd_MP_circ(void** StartAddress, Word32 Mu, Word64 Rtt, void*
            BaseAddress) 283
    Rdd=memd(Rx++#s4:3:circ(Mu))
        Word32 Q6_R_memd_IM_circ(void** StartAddress, Word32 Is4_3, Word32 Mu,
            void* BaseAddress) 221
    Rdd=memd(Rx++I:circ(Mu))
        Word32 Q6_R_memd_M_circ(void** StartAddress, Word32 Mu, void* BaseAddress)
            221

memh
    memh(Rx++#s4:1:circ(Mu))=Rt
        void Q6_memh_IMR_circ(void** StartAddress, Word32 Is4_1, Word32 Mu, Word32
            Rt, void* BaseAddress) 292
    memh(Rx++#s4:1:circ(Mu))=Rt.H
        void Q6_memh_IMRh_circ(void** StartAddress, Word32 Is4_1, Word32 Mu, Word32
            Rt, void* BaseAddress) 292
    memh(Rx++I:circ(Mu))=Rt
        void Q6_memh_MR_circ(void** StartAddress, Word32 Mu, Word32 Rt, void*
            BaseAddress) 292
    memh(Rx++I:circ(Mu))=Rt.H
        void Q6_memh_MRh_circ(void** StartAddress, Word32 Mu, Word32 Rt, void*
            BaseAddress) 292
    Rd=memh(Rx++#s4:1:circ(Mu))
        Word32 Q6_R_memh_IM_circ(void** StartAddress, Word32 Is4_1, Word32 Mu,
            void* BaseAddress) 234
    Rd=memh(Rx++I:circ(Mu))
        Word32 Q6_R_memh_M_circ(void** StartAddress, Word32 Mu, void* BaseAddress)
            234

memub
    Rd=memub(Rx++#s4:0:circ(Mu))
        Word32 Q6_R_memub_IM_circ(void** StartAddress, Word32 Is4_0, Word32 Mu,
            void* BaseAddress) 238
    Rd=memub(Rx++I:circ(Mu))
        Word32   Q6_R_memub_M_circ(void**   StartAddress,   Word32   Mu,   void*
            BaseAddress) 238

memuh
    Rd=memuh(Rx++#s4:1:circ(Mu))
        Word32 Q6_R_memuh_IM_circ(void** StartAddress, Word32 Is4_1, Word32 Mu,
            void* BaseAddress) 242
    Rd=memuh(Rx++I:circ(Mu))
        Word32   Q6_R_memuh_M_circ(void**   StartAddress,   Word32   Mu,   void*
            BaseAddress) 242
```

```
memw
    memw(Rx++#s4:2:circ(Mu))=Rt
        void Q6_memw_IMR_circ(void** StartAddress, Word32 Is4_2, Word32 Mu, Word32
             Rt, void* BaseAddress) 298
    memw(Rx++I:circ(Mu))=Rt
        void  Q6_memw_MR_circ(void**  StartAddress,  Word32  Mu,  Word32  Rt,  void*
             BaseAddress) 298
    Rd=memw(Rx++#s4:2:circ(Mu))
        Word32  Q6_R_memw_IM_circ(void**  StartAddress,  Word32  Is4_2,  Word32  Mu,
             void* BaseAddress) 246
    Rd=memw(Rx++I:circ(Mu))
        Word32 Q6_R_memw_M_circ(void** StartAddress, Word32 Mu, void* BaseAddress)
             246

min
    Rd=min(Rt,Rs)
        Word32 Q6_R_min_RR(Word32 Rt, Word32 Rs) 341
    Rdd=min(Rtt,Rss)
        Word64 Q6_P_min_PP(Word64 Rtt, Word64 Rss) 342

minu
    Rd=minu(Rt,Rs)
        UWord32 Q6_R_minu_RR(Word32 Rt, Word32 Rs) 341
    Rdd=minu(Rtt,Rss)
        UWord64 Q6_P_minu_PP(Word64 Rtt, Word64 Rss) 342

modwrap
    Rd=modwrap(Rs,Rt)
        Word32 Q6_R_modwrap_RR(Word32 Rs, Word32 Rt) 343

mpy
    Rd=mpy(Rs,Rt.H):<<1:rnd:sat
        Word32 Q6_R_mpy_RRh_s1_rnd_sat(Word32 Rs, Word32 Rt) 492
    Rd=mpy(Rs,Rt.H):<<1:sat
        Word32 Q6_R_mpy_RRh_s1_sat(Word32 Rs, Word32 Rt) 492
    Rd=mpy(Rs,Rt.L):<<1:rnd:sat
        Word32 Q6_R_mpy_RRl_s1_rnd_sat(Word32 Rs, Word32 Rt) 492
    Rd=mpy(Rs,Rt.L):<<1:sat
        Word32 Q6_R_mpy_RRl_s1_sat(Word32 Rs, Word32 Rt) 492
    Rd=mpy(Rs,Rt)
        Word32 Q6_R_mpy_RR(Word32 Rs, Word32 Rt) 492
    Rd=mpy(Rs,Rt):<<1
        Word32 Q6_R_mpy_RR_s1(Word32 Rs, Word32 Rt) 492
    Rd=mpy(Rs,Rt):<<1:sat
        Word32 Q6_R_mpy_RR_s1_sat(Word32 Rs, Word32 Rt) 492
    Rd=mpy(Rs,Rt):rnd
        Word32 Q6_R_mpy_RR_rnd(Word32 Rs, Word32 Rt) 492
    Rd=mpy(Rs.H,Rt.H)
        Word32 Q6_R_mpy_RhRh(Word32 Rs, Word32 Rt) 476
    Rd=mpy(Rs.H,Rt.H):<<1
        Word32 Q6_R_mpy_RhRh_s1(Word32 Rs, Word32 Rt) 476
    Rd=mpy(Rs.H,Rt.H):<<1:rnd
        Word32 Q6_R_mpy_RhRh_s1_rnd(Word32 Rs, Word32 Rt) 476
    Rd=mpy(Rs.H,Rt.H):<<1:rnd:sat
        Word32 Q6_R_mpy_RhRh_s1_rnd_sat(Word32 Rs, Word32 Rt) 476
    Rd=mpy(Rs.H,Rt.H):<<1:sat
        Word32 Q6_R_mpy_RhRh_s1_sat(Word32 Rs, Word32 Rt) 476
    Rd=mpy(Rs.H,Rt.H):rnd
```

Word32 Q6_R_mpy_RhRh_rnd(Word32 Rs, Word32 Rt) 476
Rd=mpy(Rs.H,Rt.H):rnd:sat
Word32 Q6_R_mpy_RhRh_rnd_sat(Word32 Rs, Word32 Rt) 476
Rd=mpy(Rs.H,Rt.H):sat
Word32 Q6_R_mpy_RhRh_sat(Word32 Rs, Word32 Rt) 476
Rd=mpy(Rs.H,Rt.L)
Word32 Q6_R_mpy_RhRl(Word32 Rs, Word32 Rt) 476
Rd=mpy(Rs.H,Rt.L):<<1
Word32 Q6_R_mpy_RhRl_s1(Word32 Rs, Word32 Rt) 476
Rd=mpy(Rs.H,Rt.L):<<1:rnd
Word32 Q6_R_mpy_RhRl_s1_rnd(Word32 Rs, Word32 Rt) 476
Rd=mpy(Rs.H,Rt.L):<<1:rnd:sat
Word32 Q6_R_mpy_RhRl_s1_rnd_sat(Word32 Rs, Word32 Rt) 476
Rd=mpy(Rs.H,Rt.L):<<1:sat
Word32 Q6_R_mpy_RhRl_s1_sat(Word32 Rs, Word32 Rt) 476
Rd=mpy(Rs.H,Rt.L):rnd
Word32 Q6_R_mpy_RhRl_rnd(Word32 Rs, Word32 Rt) 476
Rd=mpy(Rs.H,Rt.L):rnd:sat
Word32 Q6_R_mpy_RhRl_rnd_sat(Word32 Rs, Word32 Rt) 476
Rd=mpy(Rs.H,Rt.L):sat
Word32 Q6_R_mpy_RhRl_sat(Word32 Rs, Word32 Rt) 476
Rd=mpy(Rs.L,Rt.H)
Word32 Q6_R_mpy_RlRh(Word32 Rs, Word32 Rt) 476
Rd=mpy(Rs.L,Rt.H):<<1
Word32 Q6_R_mpy_RlRh_s1(Word32 Rs, Word32 Rt) 476
Rd=mpy(Rs.L,Rt.H):<<1:rnd
Word32 Q6_R_mpy_RlRh_s1_rnd(Word32 Rs, Word32 Rt) 476
Rd=mpy(Rs.L,Rt.H):<<1:rnd:sat
Word32 Q6_R_mpy_RlRh_s1_rnd_sat(Word32 Rs, Word32 Rt) 476
Rd=mpy(Rs.L,Rt.H):<<1:sat
Word32 Q6_R_mpy_RlRh_s1_sat(Word32 Rs, Word32 Rt) 476
Rd=mpy(Rs.L,Rt.H):rnd
Word32 Q6_R_mpy_RlRh_rnd(Word32 Rs, Word32 Rt) 476
Rd=mpy(Rs.L,Rt.H):rnd:sat
Word32 Q6_R_mpy_RlRh_rnd_sat(Word32 Rs, Word32 Rt) 476
Rd=mpy(Rs.L,Rt.H):sat
Word32 Q6_R_mpy_RlRh_sat(Word32 Rs, Word32 Rt) 476
Rd=mpy(Rs.L,Rt.L)
Word32 Q6_R_mpy_RlRl(Word32 Rs, Word32 Rt) 476
Rd=mpy(Rs.L,Rt.L):<<1
Word32 Q6_R_mpy_RlRl_s1(Word32 Rs, Word32 Rt) 476
Rd=mpy(Rs.L,Rt.L):<<1:rnd
Word32 Q6_R_mpy_RlRl_s1_rnd(Word32 Rs, Word32 Rt) 476
Rd=mpy(Rs.L,Rt.L):<<1:rnd:sat
Word32 Q6_R_mpy_RlRl_s1_rnd_sat(Word32 Rs, Word32 Rt) 477
Rd=mpy(Rs.L,Rt.L):<<1:sat
Word32 Q6_R_mpy_RlRl_s1_sat(Word32 Rs, Word32 Rt) 477
Rd=mpy(Rs.L,Rt.L):rnd
Word32 Q6_R_mpy_RlRl_rnd(Word32 Rs, Word32 Rt) 477
Rd=mpy(Rs.L,Rt.L):rnd:sat
Word32 Q6_R_mpy_RlRl_rnd_sat(Word32 Rs, Word32 Rt) 477
Rd=mpy(Rs.L,Rt.L):sat
Word32 Q6_R_mpy_RlRl_sat(Word32 Rs, Word32 Rt) 477
Rdd=mpy(Rs,Rt)
Word64 Q6_P_mpy_RR(Word32 Rs, Word32 Rt) 493
Rdd=mpy(Rs.H,Rt.H)
Word64 Q6_P_mpy_RhRh(Word32 Rs, Word32 Rt) 477
Rdd=mpy(Rs.H,Rt.H):<<1

Word64 Q6_P_mpy_RhRh_s1(Word32 Rs, Word32 Rt) 477
Rdd=mpy(Rs.H,Rt.H):<<1:rnd
Word64 Q6_P_mpy_RhRh_s1_rnd(Word32 Rs, Word32 Rt) 477
Rdd=mpy(Rs.H,Rt.H):rnd
Word64 Q6_P_mpy_RhRh_rnd(Word32 Rs, Word32 Rt) 477
Rdd=mpy(Rs.H,Rt.L)
Word64 Q6_P_mpy_RhRl(Word32 Rs, Word32 Rt) 477
Rdd=mpy(Rs.H,Rt.L):<<1
Word64 Q6_P_mpy_RhRl_s1(Word32 Rs, Word32 Rt) 477
Rdd=mpy(Rs.H,Rt.L):<<1:rnd
Word64 Q6_P_mpy_RhRl_s1_rnd(Word32 Rs, Word32 Rt) 477
Rdd=mpy(Rs.H,Rt.L):rnd
Word64 Q6_P_mpy_RhRl_rnd(Word32 Rs, Word32 Rt) 477
Rdd=mpy(Rs.L,Rt.H)
Word64 Q6_P_mpy_RlRh(Word32 Rs, Word32 Rt) 477
Rdd=mpy(Rs.L,Rt.H):<<1
Word64 Q6_P_mpy_RlRh_s1(Word32 Rs, Word32 Rt) 477
Rdd=mpy(Rs.L,Rt.H):<<1:rnd
Word64 Q6_P_mpy_RlRh_s1_rnd(Word32 Rs, Word32 Rt) 477
Rdd=mpy(Rs.L,Rt.H):rnd
Word64 Q6_P_mpy_RlRh_rnd(Word32 Rs, Word32 Rt) 477
Rdd=mpy(Rs.L,Rt.L)
Word64 Q6_P_mpy_RlRl(Word32 Rs, Word32 Rt) 477
Rdd=mpy(Rs.L,Rt.L):<<1
Word64 Q6_P_mpy_RlRl_s1(Word32 Rs, Word32 Rt) 477
Rdd=mpy(Rs.L,Rt.L):<<1:rnd
Word64 Q6_P_mpy_RlRl_s1_rnd(Word32 Rs, Word32 Rt) 477
Rdd=mpy(Rs.L,Rt.L):rnd
Word64 Q6_P_mpy_RlRl_rnd(Word32 Rs, Word32 Rt) 477
Rx+=mpy(Rs,Rt):<<1:sat
Word32 Q6_R_mpyacc_RR_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt) 492
Rx+=mpy(Rs.H,Rt.H)
Word32 Q6_R_mpyacc_RhRh(Word32 Rx, Word32 Rs, Word32 Rt) 477
Rx+=mpy(Rs.H,Rt.H):<<1
Word32 Q6_R_mpyacc_RhRh_s1(Word32 Rx, Word32 Rs, Word32 Rt) 477
Rx+=mpy(Rs.H,Rt.H):<<1:sat
Word32 Q6_R_mpyacc_RhRh_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt) 477
Rx+=mpy(Rs.H,Rt.H):sat
Word32 Q6_R_mpyacc_RhRh_sat(Word32 Rx, Word32 Rs, Word32 Rt) 477
Rx+=mpy(Rs.H,Rt.L)
Word32 Q6_R_mpyacc_RhRl(Word32 Rx, Word32 Rs, Word32 Rt) 477
Rx+=mpy(Rs.H,Rt.L):<<1
Word32 Q6_R_mpyacc_RhRl_s1(Word32 Rx, Word32 Rs, Word32 Rt) 477
Rx+=mpy(Rs.H,Rt.L):<<1:sat
Word32 Q6_R_mpyacc_RhRl_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt) 477
Rx+=mpy(Rs.H,Rt.L):sat
Word32 Q6_R_mpyacc_RhRl_sat(Word32 Rx, Word32 Rs, Word32 Rt) 477
Rx+=mpy(Rs.L,Rt.H)
Word32 Q6_R_mpyacc_RlRh(Word32 Rx, Word32 Rs, Word32 Rt) 477
Rx+=mpy(Rs.L,Rt.H):<<1
Word32 Q6_R_mpyacc_RlRh_s1(Word32 Rx, Word32 Rs, Word32 Rt) 477
Rx+=mpy(Rs.L,Rt.H):<<1:sat
Word32 Q6_R_mpyacc_RlRh_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt) 478
Rx+=mpy(Rs.L,Rt.H):sat
Word32 Q6_R_mpyacc_RlRh_sat(Word32 Rx, Word32 Rs, Word32 Rt) 478
Rx+=mpy(Rs.L,Rt.L)
Word32 Q6_R_mpyacc_RlRl(Word32 Rx, Word32 Rs, Word32 Rt) 478
Rx+=mpy(Rs.L,Rt.L):<<1

Word32 Q6_R_mpyacc_RlRl_s1(Word32 Rx, Word32 Rs, Word32 Rt) 478
Rx+=mpy(Rs.L,Rt.L):<<1:sat
Word32 Q6_R_mpyacc_RlRl_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt) 478
Rx+=mpy(Rs.L,Rt.L):sat
Word32 Q6_R_mpyacc_RlRl_sat(Word32 Rx, Word32 Rs, Word32 Rt) 478
Rx-=mpy(Rs,Rt):<<1:sat
Word32 Q6_R_mpynac_RR_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt) 492
Rx-=mpy(Rs.H,Rt.H)
Word32 Q6_R_mpynac_RhRh(Word32 Rx, Word32 Rs, Word32 Rt) 478
Rx-=mpy(Rs.H,Rt.H):<<1
Word32 Q6_R_mpynac_RhRh_s1(Word32 Rx, Word32 Rs, Word32 Rt) 478
Rx-=mpy(Rs.H,Rt.H):<<1:sat
Word32 Q6_R_mpynac_RhRh_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt) 478
Rx-=mpy(Rs.H,Rt.H):sat
Word32 Q6_R_mpynac_RhRh_sat(Word32 Rx, Word32 Rs, Word32 Rt) 478
Rx-=mpy(Rs.H,Rt.L)
Word32 Q6_R_mpynac_RhRl(Word32 Rx, Word32 Rs, Word32 Rt) 478
Rx-=mpy(Rs.H,Rt.L):<<1
Word32 Q6_R_mpynac_RhRl_s1(Word32 Rx, Word32 Rs, Word32 Rt) 478
Rx-=mpy(Rs.H,Rt.L):<<1:sat
Word32 Q6_R_mpynac_RhRl_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt) 478
Rx-=mpy(Rs.H,Rt.L):sat
Word32 Q6_R_mpynac_RhRl_sat(Word32 Rx, Word32 Rs, Word32 Rt) 478
Rx-=mpy(Rs.L,Rt.H)
Word32 Q6_R_mpynac_RlRh(Word32 Rx, Word32 Rs, Word32 Rt) 478
Rx-=mpy(Rs.L,Rt.H):<<1
Word32 Q6_R_mpynac_RlRh_s1(Word32 Rx, Word32 Rs, Word32 Rt) 478
Rx-=mpy(Rs.L,Rt.H):<<1:sat
Word32 Q6_R_mpynac_RlRh_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt) 478
Rx-=mpy(Rs.L,Rt.H):sat
Word32 Q6_R_mpynac_RlRh_sat(Word32 Rx, Word32 Rs, Word32 Rt) 478
Rx-=mpy(Rs.L,Rt.L)
Word32 Q6_R_mpynac_RlRl(Word32 Rx, Word32 Rs, Word32 Rt) 478
Rx-=mpy(Rs.L,Rt.L):<<1
Word32 Q6_R_mpynac_RlRl_s1(Word32 Rx, Word32 Rs, Word32 Rt) 478
Rx-=mpy(Rs.L,Rt.L):<<1:sat
Word32 Q6_R_mpynac_RlRl_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt) 478
Rx-=mpy(Rs.L,Rt.L):sat
Word32 Q6_R_mpynac_RlRl_sat(Word32 Rx, Word32 Rs, Word32 Rt) 478
Rxx+=mpy(Rs,Rt)
Word64 Q6_P_mpyacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt) 493
Rxx+=mpy(Rs.H,Rt.H)
Word64 Q6_P_mpyacc_RhRh(Word64 Rxx, Word32 Rs, Word32 Rt) 478
Rxx+=mpy(Rs.H,Rt.H):<<1
Word64 Q6_P_mpyacc_RhRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt) 479
Rxx+=mpy(Rs.H,Rt.L)
Word64 Q6_P_mpyacc_RhRl(Word64 Rxx, Word32 Rs, Word32 Rt) 479
Rxx+=mpy(Rs.H,Rt.L):<<1
Word64 Q6_P_mpyacc_RhRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt) 479
Rxx+=mpy(Rs.L,Rt.H)
Word64 Q6_P_mpyacc_RlRh(Word64 Rxx, Word32 Rs, Word32 Rt) 479
Rxx+=mpy(Rs.L,Rt.H):<<1
Word64 Q6_P_mpyacc_RlRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt) 479
Rxx+=mpy(Rs.L,Rt.L)
Word64 Q6_P_mpyacc_RlRl(Word64 Rxx, Word32 Rs, Word32 Rt) 479
Rxx+=mpy(Rs.L,Rt.L):<<1
Word64 Q6_P_mpyacc_RlRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt) 479
Rxx-=mpy(Rs,Rt)

```
        Word64 Q6_P_mpynac_RR(Word64 Rxx, Word32 Rs, Word32 Rt) 493
    Rxx-=mpy(Rs.H,Rt.H)
        Word64 Q6_P_mpynac_RhRh(Word64 Rxx, Word32 Rs, Word32 Rt) 479
    Rxx-=mpy(Rs.H,Rt.H):<<1
        Word64 Q6_P_mpynac_RhRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt) 479
    Rxx-=mpy(Rs.H,Rt.L)
        Word64 Q6_P_mpynac_RhRl(Word64 Rxx, Word32 Rs, Word32 Rt) 479
    Rxx-=mpy(Rs.H,Rt.L):<<1
        Word64 Q6_P_mpynac_RhRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt) 479
    Rxx-=mpy(Rs.L,Rt.H)
        Word64 Q6_P_mpynac_RlRh(Word64 Rxx, Word32 Rs, Word32 Rt) 479
    Rxx-=mpy(Rs.L,Rt.H):<<1
        Word64 Q6_P_mpynac_RlRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt) 479
    Rxx-=mpy(Rs.L,Rt.L)
        Word64 Q6_P_mpynac_RlRl(Word64 Rxx, Word32 Rs, Word32 Rt) 479
    Rxx-=mpy(Rs.L,Rt.L):<<1
        Word64 Q6_P_mpynac_RlRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt) 479

mpyi
    Rd=mpyi(Rs,#m9)
        Word32 Q6_R_mpyi_RI(Word32 Rs, Word32 Im9) 466
    Rd=mpyi(Rs,Rt)
        Word32 Q6_R_mpyi_RR(Word32 Rs, Word32 Rt) 466
    Rx+=mpyi(Rs,#u8)
        Word32 Q6_R_mpyiacc_RI(Word32 Rx, Word32 Rs, Word32 Iu8) 466
    Rx+=mpyi(Rs,Rt)
        Word32 Q6_R_mpyiacc_RR(Word32 Rx, Word32 Rs, Word32 Rt) 466
    Rx-=mpyi(Rs,#u8)
        Word32 Q6_R_mpyinac_RI(Word32 Rx, Word32 Rs, Word32 Iu8) 466
    Rx-=mpyi(Rs,Rt)
        Word32 Q6_R_mpyinac_RR(Word32 Rx, Word32 Rs, Word32 Rt) 466

mpysu
    Rd=mpysu(Rs,Rt)
        Word32 Q6_R_mpysu_RR(Word32 Rs, Word32 Rt) 492

mpyu
    Rd=mpyu(Rs,Rt)
        UWord32 Q6_R_mpyu_RR(Word32 Rs, Word32 Rt) 492
    Rd=mpyu(Rs.H,Rt.H)
        UWord32 Q6_R_mpyu_RhRh(Word32 Rs, Word32 Rt) 483
    Rd=mpyu(Rs.H,Rt.H):<<1
        UWord32 Q6_R_mpyu_RhRh_s1(Word32 Rs, Word32 Rt) 483
    Rd=mpyu(Rs.H,Rt.L)
        UWord32 Q6_R_mpyu_RhRl(Word32 Rs, Word32 Rt) 483
    Rd=mpyu(Rs.H,Rt.L):<<1
        UWord32 Q6_R_mpyu_RhRl_s1(Word32 Rs, Word32 Rt) 483
    Rd=mpyu(Rs.L,Rt.H)
        UWord32 Q6_R_mpyu_RlRh(Word32 Rs, Word32 Rt) 483
    Rd=mpyu(Rs.L,Rt.H):<<1
        UWord32 Q6_R_mpyu_RlRh_s1(Word32 Rs, Word32 Rt) 483
    Rd=mpyu(Rs.L,Rt.L)
        UWord32 Q6_R_mpyu_RlRl(Word32 Rs, Word32 Rt) 483
    Rd=mpyu(Rs.L,Rt.L):<<1
        UWord32 Q6_R_mpyu_RlRl_s1(Word32 Rs, Word32 Rt) 483
    Rdd=mpyu(Rs,Rt)
        UWord64 Q6_P_mpyu_RR(Word32 Rs, Word32 Rt) 493
    Rdd=mpyu(Rs.H,Rt.H)
```

UWord64 Q6_P_mpyu_RhRh(Word32 Rs, Word32 Rt) 483
Rdd=mpyu(Rs.H,Rt.H):<<1
UWord64 Q6_P_mpyu_RhRh_s1(Word32 Rs, Word32 Rt) 483
Rdd=mpyu(Rs.H,Rt.L)
UWord64 Q6_P_mpyu_RhRl(Word32 Rs, Word32 Rt) 483
Rdd=mpyu(Rs.H,Rt.L):<<1
UWord64 Q6_P_mpyu_RhRl_s1(Word32 Rs, Word32 Rt) 483
Rdd=mpyu(Rs.L,Rt.H)
UWord64 Q6_P_mpyu_RlRh(Word32 Rs, Word32 Rt) 483
Rdd=mpyu(Rs.L,Rt.H):<<1
UWord64 Q6_P_mpyu_RlRh_s1(Word32 Rs, Word32 Rt) 483
Rdd=mpyu(Rs.L,Rt.L)
UWord64 Q6_P_mpyu_RlRl(Word32 Rs, Word32 Rt) 483
Rdd=mpyu(Rs.L,Rt.L):<<1
UWord64 Q6_P_mpyu_RlRl_s1(Word32 Rs, Word32 Rt) 483
Rx+=mpyu(Rs.H,Rt.H)
Word32 Q6_R_mpyuacc_RhRh(Word32 Rx, Word32 Rs, Word32 Rt) 483
Rx+=mpyu(Rs.H,Rt.H):<<1
Word32 Q6_R_mpyuacc_RhRh_s1(Word32 Rx, Word32 Rs, Word32 Rt) 483
Rx+=mpyu(Rs.H,Rt.L)
Word32 Q6_R_mpyuacc_RhRl(Word32 Rx, Word32 Rs, Word32 Rt) 483
Rx+=mpyu(Rs.H,Rt.L):<<1
Word32 Q6_R_mpyuacc_RhRl_s1(Word32 Rx, Word32 Rs, Word32 Rt) 483
Rx+=mpyu(Rs.L,Rt.H)
Word32 Q6_R_mpyuacc_RlRh(Word32 Rx, Word32 Rs, Word32 Rt) 483
Rx+=mpyu(Rs.L,Rt.H):<<1
Word32 Q6_R_mpyuacc_RlRh_s1(Word32 Rx, Word32 Rs, Word32 Rt) 483
Rx+=mpyu(Rs.L,Rt.L)
Word32 Q6_R_mpyuacc_RlRl(Word32 Rx, Word32 Rs, Word32 Rt) 483
Rx+=mpyu(Rs.L,Rt.L):<<1
Word32 Q6_R_mpyuacc_RlRl_s1(Word32 Rx, Word32 Rs, Word32 Rt) 483
Rx-=mpyu(Rs.H,Rt.H)
Word32 Q6_R_mpyunac_RhRh(Word32 Rx, Word32 Rs, Word32 Rt) 483
Rx-=mpyu(Rs.H,Rt.H):<<1
Word32 Q6_R_mpyunac_RhRh_s1(Word32 Rx, Word32 Rs, Word32 Rt) 483
Rx-=mpyu(Rs.H,Rt.L)
Word32 Q6_R_mpyunac_RhRl(Word32 Rx, Word32 Rs, Word32 Rt) 483
Rx-=mpyu(Rs.H,Rt.L):<<1
Word32 Q6_R_mpyunac_RhRl_s1(Word32 Rx, Word32 Rs, Word32 Rt) 484
Rx-=mpyu(Rs.L,Rt.H)
Word32 Q6_R_mpyunac_RlRh(Word32 Rx, Word32 Rs, Word32 Rt) 484
Rx-=mpyu(Rs.L,Rt.H):<<1
Word32 Q6_R_mpyunac_RlRh_s1(Word32 Rx, Word32 Rs, Word32 Rt) 484
Rx-=mpyu(Rs.L,Rt.L)
Word32 Q6_R_mpyunac_RlRl(Word32 Rx, Word32 Rs, Word32 Rt) 484
Rx-=mpyu(Rs.L,Rt.L):<<1
Word32 Q6_R_mpyunac_RlRl_s1(Word32 Rx, Word32 Rs, Word32 Rt) 484
Rxx+=mpyu(Rs,Rt)
Word64 Q6_P_mpyuacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt) 493
Rxx+=mpyu(Rs.H,Rt.H)
Word64 Q6_P_mpyuacc_RhRh(Word64 Rxx, Word32 Rs, Word32 Rt) 484
Rxx+=mpyu(Rs.H,Rt.H):<<1
Word64 Q6_P_mpyuacc_RhRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt) 484
Rxx+=mpyu(Rs.H,Rt.L)
Word64 Q6_P_mpyuacc_RhRl(Word64 Rxx, Word32 Rs, Word32 Rt) 484
Rxx+=mpyu(Rs.H,Rt.L):<<1
Word64 Q6_P_mpyuacc_RhRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt) 484
Rxx+=mpyu(Rs.L,Rt.H)

```
        Word64 Q6_P_mpyuacc_RlRh(Word64 Rxx, Word32 Rs, Word32 Rt) 484
    Rxx+=mpyu(Rs.L,Rt.H):<<1
        Word64 Q6_P_mpyuacc_RlRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt) 484
    Rxx+=mpyu(Rs.L,Rt.L)
        Word64 Q6_P_mpyuacc_RlRl(Word64 Rxx, Word32 Rs, Word32 Rt) 484
    Rxx+=mpyu(Rs.L,Rt.L):<<1
        Word64 Q6_P_mpyuacc_RlRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt) 484
    Rxx-=mpyu(Rs,Rt)
        Word64 Q6_P_mpyunac_RR(Word64 Rxx, Word32 Rs, Word32 Rt) 493
    Rxx-=mpyu(Rs.H,Rt.H)
        Word64 Q6_P_mpyunac_RhRh(Word64 Rxx, Word32 Rs, Word32 Rt) 484
    Rxx-=mpyu(Rs.H,Rt.H):<<1
        Word64 Q6_P_mpyunac_RhRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt) 484
    Rxx-=mpyu(Rs.H,Rt.L)
        Word64 Q6_P_mpyunac_RhRl(Word64 Rxx, Word32 Rs, Word32 Rt) 484
    Rxx-=mpyu(Rs.H,Rt.L):<<1
        Word64 Q6_P_mpyunac_RhRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt) 484
    Rxx-=mpyu(Rs.L,Rt.H)
        Word64 Q6_P_mpyunac_RlRh(Word64 Rxx, Word32 Rs, Word32 Rt) 484
    Rxx-=mpyu(Rs.L,Rt.H):<<1
        Word64 Q6_P_mpyunac_RlRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt) 484
    Rxx-=mpyu(Rs.L,Rt.L)
        Word64 Q6_P_mpyunac_RlRl(Word64 Rxx, Word32 Rs, Word32 Rt) 484
    Rxx-=mpyu(Rs.L,Rt.L):<<1
        Word64 Q6_P_mpyunac_RlRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt) 484

mpyui
    Rd=mpyui(Rs,Rt)
        Word32 Q6_R_mpyui_RR(Word32 Rs, Word32 Rt) 466

mux
    Rd=mux(Pu,#s8,#S8)
        Word32 Q6_R_mux_pII(Byte Pu, Word32 Is8, Word32 IS8) 174
    Rd=mux(Pu,#s8,Rs)
        Word32 Q6_R_mux_pIR(Byte Pu, Word32 Is8, Word32 Rs) 174
    Rd=mux(Pu,Rs,#s8)
        Word32 Q6_R_mux_pRI(Byte Pu, Word32 Rs, Word32 Is8) 174
    Rd=mux(Pu,Rs,Rt)
        Word32 Q6_R_mux_pRR(Byte Pu, Word32 Rs, Word32 Rt) 174
```

N

```
neg
    Rd=neg(Rs)
        Word32 Q6_R_neg_R(Word32 Rs) 161
    Rd=neg(Rs):sat
        Word32 Q6_R_neg_R_sat(Word32 Rs) 344
    Rdd=neg(Rss)
        Word64 Q6_P_neg_P(Word64 Rss) 344

no mnemonic
    Pd=Ps
        Byte Q6_p_equals_p(Byte Ps) 201
    Pd=Rs
        Byte Q6_p_equals_R(Word32 Rs) 545
    Rd=#s16
        Word32 Q6_R_equals_I(Word32 Is16) 165
    Rd=Ps
```

Word32 Q6_R_equals_p(Byte Ps) 545
Rd=Rs
Word32 Q6_R_equals_R(Word32 Rs) 167
Rdd=#s8
Word64 Q6_P_equals_I(Word32 Is8) 165
Rdd=Rss
Word64 Q6_P_equals_P(Word64 Rss) 167
Rx.H=#u16
Word32 Q6_Rh_equals_I(Word32 Rx, Word32 Iu16) 165
Rx.L=#u16
Word32 Q6_Rl_equals_I(Word32 Rx, Word32 Iu16) 165

normamt
Rd=normamt(Rs)
Word32 Q6_R_normamt_R(Word32 Rs) 396
Rd=normamt(Rss)
Word32 Q6_R_normamt_P(Word64 Rss) 396

not
Pd=not(Ps)
Byte Q6_p_not_p(Byte Ps) 201
Rd=not(Rs)
Word32 Q6_R_not_R(Word32 Rs) 159
Rdd=not(Rss)
Word64 Q6_P_not_P(Word64 Rss) 335

O

or
Pd=and(Ps,or(Pt,!Pu))
Byte Q6_p_and_or_ppnp(Byte Ps, Byte Pt, Byte Pu) 201
Pd=and(Ps,or(Pt,Pu))
Byte Q6_p_and_or_ppp(Byte Ps, Byte Pt, Byte Pu) 201
Pd=or(Ps,and(Pt,!Pu))
Byte Q6_p_or_and_ppnp(Byte Ps, Byte Pt, Byte Pu) 201
Pd=or(Ps,and(Pt,Pu))
Byte Q6_p_or_and_ppp(Byte Ps, Byte Pt, Byte Pu) 201
Pd=or(Ps,or(Pt,!Pu))
Byte Q6_p_or_or_ppnp(Byte Ps, Byte Pt, Byte Pu) 202
Pd=or(Ps,or(Pt,Pu))
Byte Q6_p_or_or_ppp(Byte Ps, Byte Pt, Byte Pu) 202
Pd=or(Pt,!Ps)
Byte Q6_p_or_pnp(Byte Pt, Byte Ps) 202
Pd=or(Pt,Ps)
Byte Q6_p_or_pp(Byte Pt, Byte Ps) 202
Rd=or(Rs,#s10)
Word32 Q6_R_or_RI(Word32 Rs, Word32 Is10) 159
Rd=or(Rs,Rt)
Word32 Q6_R_or_RR(Word32 Rs, Word32 Rt) 159
Rd=or(Rt,~Rs)
Word32 Q6_R_or_RnR(Word32 Rt, Word32 Rs) 159
Rdd=or(Rss,Rtt)
Word64 Q6_P_or_PP(Word64 Rss, Word64 Rtt) 335
Rdd=or(Rtt,~Rss)
Word64 Q6_P_or_PnP(Word64 Rtt, Word64 Rss) 335
Rx&=or(Rs,Rt)
Word32 Q6_R_orand_RR(Word32 Rx, Word32 Rs, Word32 Rt) 337
Rx^=or(Rs,Rt)

Word32 Q6_R_orxacc_RR(Word32 Rx, Word32 Rs, Word32 Rt) 337
Rx=or(Ru,and(Rx,#s10))
Word32 Q6_R_or_and_RRI(Word32 Ru, Word32 Rx, Word32 Is10) 337
Rx|=or(Rs,#s10)
Word32 Q6_R_oror_RI(Word32 Rx, Word32 Rs, Word32 Is10) 338
Rx|=or(Rs,Rt)
Word32 Q6_R_oror_RR(Word32 Rx, Word32 Rs, Word32 Rt) 338

P

packhl
Rdd=packhl(Rs,Rt)
Word64 Q6_P_packhl_RR(Word32 Rs, Word32 Rt) 177

parity
Rd=parity(Rs,Rt)
Word32 Q6_R_parity_RR(Word32 Rs, Word32 Rt) 406
Rd=parity(Rss,Rtt)
Word32 Q6_R_parity_PP(Word64 Rss, Word64 Rtt) 406

pmpyw
Rdd=pmpyw(Rs,Rt)
Word64 Q6_P_pmpyw_RR(Word32 Rs, Word32 Rt) 487
Rxx^=pmpyw(Rs,Rt)
Word64 Q6_P_pmpywxacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt) 487

popcount
Rd=popcount(Rss)
Word32 Q6_R_popcount_P(Word64 Rss) 397

R

rol
Rd=rol(Rs,#u5)
Word32 Q6_R_rol_RI(Word32 Rs, Word32 Iu5) 557
Rdd=rol(Rss,#u6)
Word64 Q6_P_rol_PI(Word64 Rss, Word32 Iu6) 558
Rx&=rol(Rs,#u5)
Word32 Q6_R_roland_RI(Word32 Rx, Word32 Rs, Word32 Iu5) 564
Rx^=rol(Rs,#u5)
Word32 Q6_R_rolxacc_RI(Word32 Rx, Word32 Rs, Word32 Iu5) 564
Rx+=rol(Rs,#u5)
Word32 Q6_R_rolacc_RI(Word32 Rx, Word32 Rs, Word32 Iu5) 560
Rx-=rol(Rs,#u5)
Word32 Q6_R_rolnac_RI(Word32 Rx, Word32 Rs, Word32 Iu5) 560
Rx|=rol(Rs,#u5)
Word32 Q6_R_rolor_RI(Word32 Rx, Word32 Rs, Word32 Iu5) 564
Rxx&=rol(Rss,#u6)
Word64 Q6_P_roland_PI(Word64 Rxx, Word64 Rss, Word32 Iu6) 565
Rxx^=rol(Rss,#u6)
Word64 Q6_P_rolxacc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6) 565
Rxx+=rol(Rss,#u6)
Word64 Q6_P_rolacc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6) 560
Rxx-=rol(Rss,#u6)
Word64 Q6_P_rolnac_PI(Word64 Rxx, Word64 Rss, Word32 Iu6) 560
Rxx|=rol(Rss,#u6)
Word64 Q6_P_rolor_PI(Word64 Rxx, Word64 Rss, Word32 Iu6) 565

round
Rd=round(Rs,#u5)

Word32 Q6_R_round_RI(Word32 Rs, Word32 Iu5) 346
Rd=round(Rs,#u5):sat
Word32 Q6_R_round_RI_sat(Word32 Rs, Word32 Iu5) 346
Rd=round(Rs,Rt)
Word32 Q6_R_round_RR(Word32 Rs, Word32 Rt) 346
Rd=round(Rs,Rt):sat
Word32 Q6_R_round_RR_sat(Word32 Rs, Word32 Rt) 346
Rd=round(Rss):sat
Word32 Q6_R_round_P_sat(Word64 Rss) 346

S

sat
Rd=sat(Rss)
Word32 Q6_R_sat_P(Word64 Rss) 518

satb
Rd=satb(Rs)
Word32 Q6_R_satb_R(Word32 Rs) 518

sath
Rd=sath(Rs)
Word32 Q6_R_sath_R(Word32 Rs) 518

satub
Rd=satub(Rs)
Word32 Q6_R_satub_R(Word32 Rs) 518

satuh
Rd=satuh(Rs)
Word32 Q6_R_satuh_R(Word32 Rs) 518

setbit
Rd=setbit(Rs,#u5)
Word32 Q6_R_setbit_RI(Word32 Rs, Word32 Iu5) 408
Rd=setbit(Rs,Rt)
Word32 Q6_R_setbit_RR(Word32 Rs, Word32 Rt) 408

sfadd
Rd=sfadd(Rs,Rt)
Word32 Q6_R_sfadd_RR(Word32 Rs, Word32 Rt) 444

sfclass
Pd=sfclass(Rs,#u5)
Byte Q6_p_sfclass_RI(Word32 Rs, Word32 Iu5) 445

sfcmp.eq
Pd=sfcmp.eq(Rs,Rt)
Byte Q6_p_sfcmp_eq_RR(Word32 Rs, Word32 Rt) 446

sfcmp.ge
Pd=sfcmp.ge(Rs,Rt)
Byte Q6_p_sfcmp_ge_RR(Word32 Rs, Word32 Rt) 446

sfcmp.gt
Pd=sfcmp.gt(Rs,Rt)
Byte Q6_p_sfcmp_gt_RR(Word32 Rs, Word32 Rt) 446

sfcmp.uo
Pd=sfcmp.uo(Rs,Rt)

Byte Q6_p_sfcmp_uo_RR(Word32 Rs, Word32 Rt) 446

sffixupd
Rd=sffixupd(Rs,Rt)
Word32 Q6_R_sffixupd_RR(Word32 Rs, Word32 Rt) 453

sffixupn
Rd=sffixupn(Rs,Rt)
Word32 Q6_R_sffixupn_RR(Word32 Rs, Word32 Rt) 453

sffixupr
Rd=sffixupr(Rs)
Word32 Q6_R_sffixupr_R(Word32 Rs) 453

sfmake
Rd=sfmake(#u10):neg
Word32 Q6_R_sfmake_I_neg(Word32 Iu10) 459
Rd=sfmake(#u10):pos
Word32 Q6_R_sfmake_I_pos(Word32 Iu10) 459

sfmax
Rd=sfmax(Rs,Rt)
Word32 Q6_R_sfmax_RR(Word32 Rs, Word32 Rt) 460

sfmin
Rd=sfmin(Rs,Rt)
Word32 Q6_R_sfmin_RR(Word32 Rs, Word32 Rt) 461

sfmpy
Rd=sfmpy(Rs,Rt)
Word32 Q6_R_sfmpy_RR(Word32 Rs, Word32 Rt) 462
Rx+=sfmpy(Rs,Rt,Pu):scale
Word32 Q6_R_sfmpyacc_RRp_scale(Word32 Rx, Word32 Rs, Word32 Rt, Byte Pu) 455
Rx+=sfmpy(Rs,Rt)
Word32 Q6_R_sfmpyacc_RR(Word32 Rx, Word32 Rs, Word32 Rt) 454
Rx+=sfmpy(Rs,Rt):lib
Word32 Q6_R_sfmpyacc_RR_lib(Word32 Rx, Word32 Rs, Word32 Rt) 457
Rx-=sfmpy(Rs,Rt)
Word32 Q6_R_sfmpynac_RR(Word32 Rx, Word32 Rs, Word32 Rt) 454
Rx-=sfmpy(Rs,Rt):lib
Word32 Q6_R_sfmpynac_RR_lib(Word32 Rx, Word32 Rs, Word32 Rt) 457

sfsub
Rd=sfsub(Rs,Rt)
Word32 Q6_R_sfsub_RR(Word32 Rs, Word32 Rt) 464

shuffeb
Rdd=shuffeb(Rss,Rtt)
Word64 Q6_P_shuffeb_PP(Word64 Rss, Word64 Rtt) 529

shuffeh
Rdd=shuffeh(Rss,Rtt)
Word64 Q6_P_shuffeh_PP(Word64 Rss, Word64 Rtt) 529

shuffob
Rdd=shuffob(Rtt,Rss)
Word64 Q6_P_shuffob_PP(Word64 Rtt, Word64 Rss) 529

shuffoh
Rdd=shuffoh(Rtt,Rss)

```
        Word64 Q6_P_shuffoh_PP(Word64 Rtt, Word64 Rss) 529

sub
    Rd=add(Rs,sub(#s6,Ru))
        Word32 Q6_R_add_sub_RIR(Word32 Rs, Word32 Is6, Word32 Ru) 327
    Rd=sub(#s10,Rs)
        Word32 Q6_R_sub_IR(Word32 Is10, Word32 Rs) 163
    Rd=sub(Rt,Rs)
        Word32 Q6_R_sub_RR(Word32 Rt, Word32 Rs) 163
    Rd=sub(Rt,Rs):sat
        Word32 Q6_R_sub_RR_sat(Word32 Rt, Word32 Rs) 163
    Rd=sub(Rt.H,Rs.H):<<16
        Word32 Q6_R_sub_RhRh_s16(Word32 Rt, Word32 Rs) 350
    Rd=sub(Rt.H,Rs.H):sat:<<16
        Word32 Q6_R_sub_RhRh_sat_s16(Word32 Rt, Word32 Rs) 350
    Rd=sub(Rt.H,Rs.L):<<16
        Word32 Q6_R_sub_RhRl_s16(Word32 Rt, Word32 Rs) 350
    Rd=sub(Rt.H,Rs.L):sat:<<16
        Word32 Q6_R_sub_RhRl_sat_s16(Word32 Rt, Word32 Rs) 350
    Rd=sub(Rt.L,Rs.H)
        Word32 Q6_R_sub_RlRh(Word32 Rt, Word32 Rs) 350
    Rd=sub(Rt.L,Rs.H):<<16
        Word32 Q6_R_sub_RlRh_s16(Word32 Rt, Word32 Rs) 351
    Rd=sub(Rt.L,Rs.H):sat
        Word32 Q6_R_sub_RlRh_sat(Word32 Rt, Word32 Rs) 351
    Rd=sub(Rt.L,Rs.H):sat:<<16
        Word32 Q6_R_sub_RlRh_sat_s16(Word32 Rt, Word32 Rs) 351
    Rd=sub(Rt.L,Rs.L)
        Word32 Q6_R_sub_RlRl(Word32 Rt, Word32 Rs) 351
    Rd=sub(Rt.L,Rs.L):<<16
        Word32 Q6_R_sub_RlRl_s16(Word32 Rt, Word32 Rs) 351
    Rd=sub(Rt.L,Rs.L):sat
        Word32 Q6_R_sub_RlRl_sat(Word32 Rt, Word32 Rs) 351
    Rd=sub(Rt.L,Rs.L):sat:<<16
        Word32 Q6_R_sub_RlRl_sat_s16(Word32 Rt, Word32 Rs) 351
    Rdd=sub(Rtt,Rss)
        Word64 Q6_P_sub_PP(Word64 Rtt, Word64 Rss) 348
    Rx+=sub(Rt,Rs)
        Word32 Q6_R_subacc_RR(Word32 Rx, Word32 Rt, Word32 Rs) 349

swiz
    Rd=swiz(Rs)
        Word32 Q6_R_swiz_R(Word32 Rs) 520

sxtb
    Rd=sxtb(Rs)
        Word32 Q6_R_sxtb_R(Word32 Rs) 164

sxth
    Rd=sxth(Rs)
        Word32 Q6_R_sxth_R(Word32 Rs) 164

sxtw
    Rdd=sxtw(Rs)
        Word64 Q6_P_sxtw_R(Word32 Rs) 352
```

T

tableidxb
Rx=tableidxb(Rs,#u4,#U5)
Word32 Q6_R_tableidxb_RII(Word32 Rx, Word32 Rs, Word32 Iu4, Word32 IU5) 413

tableidxd
Rx=tableidxd(Rs,#u4,#U5)
Word32 Q6_R_tableidxd_RII(Word32 Rx, Word32 Rs, Word32 Iu4, Word32 IU5) 413

tableidxh
Rx=tableidxh(Rs,#u4,#U5)
Word32 Q6_R_tableidxh_RII(Word32 Rx, Word32 Rs, Word32 Iu4, Word32 IU5) 413

tableidxw
Rx=tableidxw(Rs,#u4,#U5)
Word32 Q6_R_tableidxw_RII(Word32 Rx, Word32 Rs, Word32 Iu4, Word32 IU5) 413

tlbmatch
Pd=tlbmatch(Rss,Rt)
Byte Q6_p_tlbmatch_PR(Word64 Rss, Word32 Rt) 544

togglebit
Rd=togglebit(Rs,#u5)
Word32 Q6_R_togglebit_RI(Word32 Rs, Word32 Iu5) 408
Rd=togglebit(Rs,Rt)
Word32 Q6_R_togglebit_RR(Word32 Rs, Word32 Rt) 408

tstbit
Pd=!tstbit(Rs,#u5)
Byte Q6_p_not_tstbit_RI(Word32 Rs, Word32 Iu5) 546
Pd=!tstbit(Rs,Rt)
Byte Q6_p_not_tstbit_RR(Word32 Rs, Word32 Rt) 546
Pd=tstbit(Rs,#u5)
Byte Q6_p_tstbit_RI(Word32 Rs, Word32 Iu5) 546
Pd=tstbit(Rs,Rt)
Byte Q6_p_tstbit_RR(Word32 Rs, Word32 Rt) 546

V

vabsdiffb
Rdd=vabsdiffb(Rtt,Rss)
Word64 Q6_P_vabsdiffb_PP(Word64 Rtt, Word64 Rss) 355

vabsdiffh
Rdd=vabsdiffh(Rtt,Rss)
Word64 Q6_P_vabsdiffh_PP(Word64 Rtt, Word64 Rss) 356

vabsdiffub
Rdd=vabsdiffub(Rtt,Rss)
Word64 Q6_P_vabsdiffub_PP(Word64 Rtt, Word64 Rss) 355

vabsdiffw
Rdd=vabsdiffw(Rtt,Rss)
Word64 Q6_P_vabsdiffw_PP(Word64 Rtt, Word64 Rss) 357

vabsh
Rdd=vabsh(Rss)
Word64 Q6_P_vabsh_P(Word64 Rss) 353
Rdd=vabsh(Rss):sat

```
        Word64 Q6_P_vabsh_P_sat(Word64 Rss) 353

vabsw
    Rdd=vabsw(Rss)
        Word64 Q6_P_vabsw_P(Word64 Rss) 354
    Rdd=vabsw(Rss):sat
        Word64 Q6_P_vabsw_P_sat(Word64 Rss) 354

vaddb
    Rdd=vaddb(Rss,Rtt)
        Word64 Q6_P_vaddb_PP(Word64 Rss, Word64 Rtt) 367

vaddh
    Rd=vaddh(Rs,Rt)
        Word32 Q6_R_vaddh_RR(Word32 Rs, Word32 Rt) 168
    Rd=vaddh(Rs,Rt):sat
        Word32 Q6_R_vaddh_RR_sat(Word32 Rs, Word32 Rt) 168
    Rdd=vaddh(Rss,Rtt)
        Word64 Q6_P_vaddh_PP(Word64 Rss, Word64 Rtt) 361
    Rdd=vaddh(Rss,Rtt):sat
        Word64 Q6_P_vaddh_PP_sat(Word64 Rss, Word64 Rtt) 361

vaddhub
    Rd=vaddhub(Rss,Rtt):sat
        Word32 Q6_R_vaddhub_PP_sat(Word64 Rss, Word64 Rtt) 362

vaddub
    Rdd=vaddub(Rss,Rtt)
        Word64 Q6_P_vaddub_PP(Word64 Rss, Word64 Rtt) 367
    Rdd=vaddub(Rss,Rtt):sat
        Word64 Q6_P_vaddub_PP_sat(Word64 Rss, Word64 Rtt) 367

vadduh
    Rd=vadduh(Rs,Rt):sat
        Word32 Q6_R_vadduh_RR_sat(Word32 Rs, Word32 Rt) 168
    Rdd=vadduh(Rss,Rtt):sat
        Word64 Q6_P_vadduh_PP_sat(Word64 Rss, Word64 Rtt) 361

vaddw
    Rdd=vaddw(Rss,Rtt)
        Word64 Q6_P_vaddw_PP(Word64 Rss, Word64 Rtt) 368
    Rdd=vaddw(Rss,Rtt):sat
        Word64 Q6_P_vaddw_PP_sat(Word64 Rss, Word64 Rtt) 368

valignb
    Rdd=valignb(Rtt,Rss,#u3)
        Word64 Q6_P_valignb_PPI(Word64 Rtt, Word64 Rss, Word32 Iu3) 521
    Rdd=valignb(Rtt,Rss,Pu)
        Word64 Q6_P_valignb_PPp(Word64 Rtt, Word64 Rss, Byte Pu) 521

vaslh
    Rdd=vaslh(Rss,#u4)
        Word64 Q6_P_vaslh_PI(Word64 Rss, Word32 Iu4) 581
    Rdd=vaslh(Rss,Rt)
        Word64 Q6_P_vaslh_PR(Word64 Rss, Word32 Rt) 586

vaslw
    Rdd=vaslw(Rss,#u5)
        Word64 Q6_P_vaslw_PI(Word64 Rss, Word32 Iu5) 587
    Rdd=vaslw(Rss,Rt)
```

```
        Word64 Q6_P_vaslw_PR(Word64 Rss, Word32 Rt) 589

vasrh
    Rdd=vasrh(Rss,#u4)
        Word64 Q6_P_vasrh_PI(Word64 Rss, Word32 Iu4) 581
    Rdd=vasrh(Rss,#u4):rnd
        Word64 Q6_P_vasrh_PI_rnd(Word64 Rss, Word32 Iu4) 582
    Rdd=vasrh(Rss,Rt)
        Word64 Q6_P_vasrh_PR(Word64 Rss, Word32 Rt) 586

vasrhub
    Rd=vasrhub(Rss,#u4):rnd:sat
        Word32 Q6_R_vasrhub_PI_rnd_sat(Word64 Rss, Word32 Iu4) 584
    Rd=vasrhub(Rss,#u4):sat
        Word32 Q6_R_vasrhub_PI_sat(Word64 Rss, Word32 Iu4) 584

vasrw
    Rd=vasrw(Rss,#u5)
        Word32 Q6_R_vasrw_PI(Word64 Rss, Word32 Iu5) 590
    Rd=vasrw(Rss,Rt)
        Word32 Q6_R_vasrw_PR(Word64 Rss, Word32 Rt) 590
    Rdd=vasrw(Rss,#u5)
        Word64 Q6_P_vasrw_PI(Word64 Rss, Word32 Iu5) 587
    Rdd=vasrw(Rss,Rt)
        Word64 Q6_P_vasrw_PR(Word64 Rss, Word32 Rt) 589

vavgh
    Rd=vavgh(Rs,Rt)
        Word32 Q6_R_vavgh_RR(Word32 Rs, Word32 Rt) 169
    Rd=vavgh(Rs,Rt):rnd
        Word32 Q6_R_vavgh_RR_rnd(Word32 Rs, Word32 Rt) 169
    Rdd=vavgh(Rss,Rtt)
        Word64 Q6_P_vavgh_PP(Word64 Rss, Word64 Rtt) 370
    Rdd=vavgh(Rss,Rtt):crnd
        Word64 Q6_P_vavgh_PP_crnd(Word64 Rss, Word64 Rtt) 370
    Rdd=vavgh(Rss,Rtt):rnd
        Word64 Q6_P_vavgh_PP_rnd(Word64 Rss, Word64 Rtt) 370

vavgub
    Rdd=vavgub(Rss,Rtt)
        Word64 Q6_P_vavgub_PP(Word64 Rss, Word64 Rtt) 371
    Rdd=vavgub(Rss,Rtt):rnd
        Word64 Q6_P_vavgub_PP_rnd(Word64 Rss, Word64 Rtt) 371

vavguh
    Rdd=vavguh(Rss,Rtt)
        Word64 Q6_P_vavguh_PP(Word64 Rss, Word64 Rtt) 370
    Rdd=vavguh(Rss,Rtt):rnd
        Word64 Q6_P_vavguh_PP_rnd(Word64 Rss, Word64 Rtt) 370

vavguw
    Rdd=vavguw(Rss,Rtt)
        Word64 Q6_P_vavguw_PP(Word64 Rss, Word64 Rtt) 373
    Rdd=vavguw(Rss,Rtt):rnd
        Word64 Q6_P_vavguw_PP_rnd(Word64 Rss, Word64 Rtt) 373

vavgw
    Rdd=vavgw(Rss,Rtt)
```

```
        Word64 Q6_P_vavgw_PP(Word64 Rss, Word64 Rtt) 373
    Rdd=vavgw(Rss,Rtt):crnd
        Word64 Q6_P_vavgw_PP_crnd(Word64 Rss, Word64 Rtt) 373
    Rdd=vavgw(Rss,Rtt):rnd
        Word64 Q6_P_vavgw_PP_rnd(Word64 Rss, Word64 Rtt) 373

vclip
    Rdd=vclip(Rss,#u5)
        Word64 Q6_P_vclip_PI(Word64 Rss, Word32 Iu5) 374

vcmpb.eq
    Pd=!any8(vcmpb.eq(Rss,Rtt))
        Byte Q6_p_not_any8_vcmpb_eq_PP(Word64 Rss, Word64 Rtt) 549
    Pd=any8(vcmpb.eq(Rss,Rtt))
        Byte Q6_p_any8_vcmpb_eq_PP(Word64 Rss, Word64 Rtt) 549
    Pd=vcmpb.eq(Rss,#u8)
        Byte Q6_p_vcmpb_eq_PI(Word64 Rss, Word32 Iu8) 550
    Pd=vcmpb.eq(Rss,Rtt)
        Byte Q6_p_vcmpb_eq_PP(Word64 Rss, Word64 Rtt) 550

vcmpb.gt
    Pd=vcmpb.gt(Rss,#s8)
        Byte Q6_p_vcmpb_gt_PI(Word64 Rss, Word32 Is8) 551
    Pd=vcmpb.gt(Rss,Rtt)
        Byte Q6_p_vcmpb_gt_PP(Word64 Rss, Word64 Rtt) 551

vcmpb.gtu
    Pd=vcmpb.gtu(Rss,#u7)
        Byte Q6_p_vcmpb_gtu_PI(Word64 Rss, Word32 Iu7) 551
    Pd=vcmpb.gtu(Rss,Rtt)
        Byte Q6_p_vcmpb_gtu_PP(Word64 Rss, Word64 Rtt) 551

vcmph.eq
    Pd=vcmph.eq(Rss,#s8)
        Byte Q6_p_vcmph_eq_PI(Word64 Rss, Word32 Is8) 548
    Pd=vcmph.eq(Rss,Rtt)
        Byte Q6_p_vcmph_eq_PP(Word64 Rss, Word64 Rtt) 548

vcmph.gt
    Pd=vcmph.gt(Rss,#s8)
        Byte Q6_p_vcmph_gt_PI(Word64 Rss, Word32 Is8) 548
    Pd=vcmph.gt(Rss,Rtt)
        Byte Q6_p_vcmph_gt_PP(Word64 Rss, Word64 Rtt) 548

vcmph.gtu
    Pd=vcmph.gtu(Rss,#u7)
        Byte Q6_p_vcmph_gtu_PI(Word64 Rss, Word32 Iu7) 548
    Pd=vcmph.gtu(Rss,Rtt)
        Byte Q6_p_vcmph_gtu_PP(Word64 Rss, Word64 Rtt) 548

vcmpw.eq
    Pd=vcmpw.eq(Rss,#s8)
        Byte Q6_p_vcmpw_eq_PI(Word64 Rss, Word32 Is8) 552
    Pd=vcmpw.eq(Rss,Rtt)
        Byte Q6_p_vcmpw_eq_PP(Word64 Rss, Word64 Rtt) 552

vcmpw.gt
    Pd=vcmpw.gt(Rss,#s8)
```

```
        Byte Q6_p_vcmpw_gt_PI(Word64 Rss, Word32 Is8) 552
    Pd=vcmpw.gt(Rss,Rtt)
        Byte Q6_p_vcmpw_gt_PP(Word64 Rss, Word64 Rtt) 552

vcmpw.gtu
    Pd=vcmpw.gtu(Rss,#u7)
        Byte Q6_p_vcmpw_gtu_PI(Word64 Rss, Word32 Iu7) 552
    Pd=vcmpw.gtu(Rss,Rtt)
        Byte Q6_p_vcmpw_gtu_PP(Word64 Rss, Word64 Rtt) 552

vcmpyi
    Rdd=vcmpyi(Rss,Rtt):<<1:sat
        Word64 Q6_P_vcmpyi_PP_s1_sat(Word64 Rss, Word64 Rtt) 432
    Rdd=vcmpyi(Rss,Rtt):sat
        Word64 Q6_P_vcmpyi_PP_sat(Word64 Rss, Word64 Rtt) 432
    Rxx+=vcmpyi(Rss,Rtt):sat
        Word64 Q6_P_vcmpyiacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt) 432

vcmpyr
    Rdd=vcmpyr(Rss,Rtt):<<1:sat
        Word64 Q6_P_vcmpyr_PP_s1_sat(Word64 Rss, Word64 Rtt) 432
    Rdd=vcmpyr(Rss,Rtt):sat
        Word64 Q6_P_vcmpyr_PP_sat(Word64 Rss, Word64 Rtt) 432
    Rxx+=vcmpyr(Rss,Rtt):sat
        Word64 Q6_P_vcmpyracc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt) 432

vcnegh
    Rdd=vcnegh(Rss,Rt)
        Word64 Q6_P_vcnegh_PR(Word64 Rss, Word32 Rt) 375

vconj
    Rdd=vconj(Rss):sat
        Word64 Q6_P_vconj_P_sat(Word64 Rss) 433

vcrotate
    Rdd=vcrotate(Rss,Rt)
        Word64 Q6_P_vcrotate_PR(Word64 Rss, Word32 Rt) 435

vdmpy
    Rd=vdmpy(Rss,Rtt):<<1:rnd:sat
        Word32 Q6_R_vdmpy_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt) 498
    Rd=vdmpy(Rss,Rtt):rnd:sat
        Word32 Q6_R_vdmpy_PP_rnd_sat(Word64 Rss, Word64 Rtt) 498
    Rdd=vdmpy(Rss,Rtt):<<1:sat
        Word64 Q6_P_vdmpy_PP_s1_sat(Word64 Rss, Word64 Rtt) 496
    Rdd=vdmpy(Rss,Rtt):sat
        Word64 Q6_P_vdmpy_PP_sat(Word64 Rss, Word64 Rtt) 496
    Rxx+=vdmpy(Rss,Rtt):<<1:sat
        Word64 Q6_P_vdmpyacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt) 496
    Rxx+=vdmpy(Rss,Rtt):sat
        Word64 Q6_P_vdmpyacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt) 496

vdmpybsu
    Rdd=vdmpybsu(Rss,Rtt):sat
        Word64 Q6_P_vdmpybsu_PP_sat(Word64 Rss, Word64 Rtt) 502
    Rxx+=vdmpybsu(Rss,Rtt):sat
        Word64 Q6_P_vdmpybsuacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt) 502
```

vitpack
Rd=vitpack(Ps,Pt)
Word32 Q6_R_vitpack_pp(Byte Ps, Byte Pt) 554

vlslh
Rdd=vlslh(Rss,Rt)
Word64 Q6_P_vlslh_PR(Word64 Rss, Word32 Rt) 586

vlslw
Rdd=vlslw(Rss,Rt)
Word64 Q6_P_vlslw_PR(Word64 Rss, Word32 Rt) 589

vlsrh
Rdd=vlsrh(Rss,#u4)
Word64 Q6_P_vlsrh_PI(Word64 Rss, Word32 Iu4) 581
Rdd=vlsrh(Rss,Rt)
Word64 Q6_P_vlsrh_PR(Word64 Rss, Word32 Rt) 586

vlsrw
Rdd=vlsrw(Rss,#u5)
Word64 Q6_P_vlsrw_PI(Word64 Rss, Word32 Iu5) 587
Rdd=vlsrw(Rss,Rt)
Word64 Q6_P_vlsrw_PR(Word64 Rss, Word32 Rt) 589

vmaxb
Rdd=vmaxb(Rtt,Rss)
Word64 Q6_P_vmaxb_PP(Word64 Rtt, Word64 Rss) 376

vmaxh
Rdd=vmaxh(Rtt,Rss)
Word64 Q6_P_vmaxh_PP(Word64 Rtt, Word64 Rss) 377

vmaxub
Rdd=vmaxub(Rtt,Rss)
Word64 Q6_P_vmaxub_PP(Word64 Rtt, Word64 Rss) 376

vmaxuh
Rdd=vmaxuh(Rtt,Rss)
Word64 Q6_P_vmaxuh_PP(Word64 Rtt, Word64 Rss) 377

vmaxuw
Rdd=vmaxuw(Rtt,Rss)
Word64 Q6_P_vmaxuw_PP(Word64 Rtt, Word64 Rss) 382

vmaxw
Rdd=vmaxw(Rtt,Rss)
Word64 Q6_P_vmaxw_PP(Word64 Rtt, Word64 Rss) 382

vminb
Rdd=vminb(Rtt,Rss)
Word64 Q6_P_vminb_PP(Word64 Rtt, Word64 Rss) 383

vminh
Rdd=vminh(Rtt,Rss)
Word64 Q6_P_vminh_PP(Word64 Rtt, Word64 Rss) 384

vminub
Rdd=vminub(Rtt,Rss)
Word64 Q6_P_vminub_PP(Word64 Rtt, Word64 Rss) 383

vminuh
Rdd=vminuh(Rtt,Rss)
Word64 Q6_P_vminuh_PP(Word64 Rtt, Word64 Rss) 384

vminuw
Rdd=vminuw(Rtt,Rss)
Word64 Q6_P_vminuw_PP(Word64 Rtt, Word64 Rss) 388

vminw
Rdd=vminw(Rtt,Rss)
Word64 Q6_P_vminw_PP(Word64 Rtt, Word64 Rss) 388

vmpybsu
Rdd=vmpybsu(Rs,Rt)
Word64 Q6_P_vmpybsu_RR(Word32 Rs, Word32 Rt) 512
Rxx+=vmpybsu(Rs,Rt)
Word64 Q6_P_vmpybsuacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt) 513

vmpybu
Rdd=vmpybu(Rs,Rt)
Word64 Q6_P_vmpybu_RR(Word32 Rs, Word32 Rt) 512
Rxx+=vmpybu(Rs,Rt)
Word64 Q6_P_vmpybuacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt) 513

vmpyeh
Rdd=vmpyeh(Rss,Rtt):<<1:sat
Word64 Q6_P_vmpyeh_PP_s1_sat(Word64 Rss, Word64 Rtt) 504
Rdd=vmpyeh(Rss,Rtt):sat
Word64 Q6_P_vmpyeh_PP_sat(Word64 Rss, Word64 Rtt) 504
Rxx+=vmpyeh(Rss,Rtt)
Word64 Q6_P_vmpyehacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt) 504
Rxx+=vmpyeh(Rss,Rtt):<<1:sat
Word64 Q6_P_vmpyehacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt) 504
Rxx+=vmpyeh(Rss,Rtt):sat
Word64 Q6_P_vmpyehacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt) 504

vmpyh
Rd=vmpyh(Rs,Rt):<<1:rnd:sat
Word32 Q6_R_vmpyh_RR_s1_rnd_sat(Word32 Rs, Word32 Rt) 508
Rd=vmpyh(Rs,Rt):rnd:sat
Word32 Q6_R_vmpyh_RR_rnd_sat(Word32 Rs, Word32 Rt) 508
Rdd=vmpyh(Rs,Rt):<<1:sat
Word64 Q6_P_vmpyh_RR_s1_sat(Word32 Rs, Word32 Rt) 506
Rdd=vmpyh(Rs,Rt):sat
Word64 Q6_P_vmpyh_RR_sat(Word32 Rs, Word32 Rt) 506
Rxx+=vmpyh(Rs,Rt)
Word64 Q6_P_vmpyhacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt) 506
Rxx+=vmpyh(Rs,Rt):<<1:sat
Word64 Q6_P_vmpyhacc_RR_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt) 506
Rxx+=vmpyh(Rs,Rt):sat
Word64 Q6_P_vmpyhacc_RR_sat(Word64 Rxx, Word32 Rs, Word32 Rt) 506

vmpyhsu
Rdd=vmpyhsu(Rs,Rt):<<1:sat
Word64 Q6_P_vmpyhsu_RR_s1_sat(Word32 Rs, Word32 Rt) 509
Rdd=vmpyhsu(Rs,Rt):sat
Word64 Q6_P_vmpyhsu_RR_sat(Word32 Rs, Word32 Rt) 509
Rxx+=vmpyhsu(Rs,Rt):<<1:sat

```
        Word64 Q6_P_vmpyhsuacc_RR_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt) 509
    Rxx+=vmpyhsu(Rs,Rt):sat
        Word64 Q6_P_vmpyhsuacc_RR_sat(Word64 Rxx, Word32 Rs, Word32 Rt) 509

vmpyweh
    Rdd=vmpyweh(Rss,Rtt):<<1:rnd:sat
        Word64 Q6_P_vmpyweh_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt) 469
    Rdd=vmpyweh(Rss,Rtt):<<1:sat
        Word64 Q6_P_vmpyweh_PP_s1_sat(Word64 Rss, Word64 Rtt) 469
    Rdd=vmpyweh(Rss,Rtt):rnd:sat
        Word64 Q6_P_vmpyweh_PP_rnd_sat(Word64 Rss, Word64 Rtt) 469
    Rdd=vmpyweh(Rss,Rtt):sat
        Word64 Q6_P_vmpyweh_PP_sat(Word64 Rss, Word64 Rtt) 469
    Rxx+=vmpyweh(Rss,Rtt):<<1:rnd:sat
        Word64 Q6_P_vmpywehacc_PP_s1_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)
               470
    Rxx+=vmpyweh(Rss,Rtt):<<1:sat
        Word64 Q6_P_vmpywehacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt) 470
    Rxx+=vmpyweh(Rss,Rtt):rnd:sat
        Word64 Q6_P_vmpywehacc_PP_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt) 470
    Rxx+=vmpyweh(Rss,Rtt):sat
        Word64 Q6_P_vmpywehacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt) 470

vmpyweuh
    Rdd=vmpyweuh(Rss,Rtt):<<1:rnd:sat
        Word64 Q6_P_vmpyweuh_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt) 472
    Rdd=vmpyweuh(Rss,Rtt):<<1:sat
        Word64 Q6_P_vmpyweuh_PP_s1_sat(Word64 Rss, Word64 Rtt) 472
    Rdd=vmpyweuh(Rss,Rtt):rnd:sat
        Word64 Q6_P_vmpyweuh_PP_rnd_sat(Word64 Rss, Word64 Rtt) 472
    Rdd=vmpyweuh(Rss,Rtt):sat
        Word64 Q6_P_vmpyweuh_PP_sat(Word64 Rss, Word64 Rtt) 473
    Rxx+=vmpyweuh(Rss,Rtt):<<1:rnd:sat
        Word64 Q6_P_vmpyweuhacc_PP_s1_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)
               473
    Rxx+=vmpyweuh(Rss,Rtt):<<1:sat
        Word64 Q6_P_vmpyweuhacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt) 473
    Rxx+=vmpyweuh(Rss,Rtt):rnd:sat
        Word64 Q6_P_vmpyweuhacc_PP_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt) 473
    Rxx+=vmpyweuh(Rss,Rtt):sat
        Word64 Q6_P_vmpyweuhacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt) 473

vmpywoh
    Rdd=vmpywoh(Rss,Rtt):<<1:rnd:sat
        Word64 Q6_P_vmpywoh_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt) 469
    Rdd=vmpywoh(Rss,Rtt):<<1:sat
        Word64 Q6_P_vmpywoh_PP_s1_sat(Word64 Rss, Word64 Rtt) 469
    Rdd=vmpywoh(Rss,Rtt):rnd:sat
        Word64 Q6_P_vmpywoh_PP_rnd_sat(Word64 Rss, Word64 Rtt) 469
    Rdd=vmpywoh(Rss,Rtt):sat
        Word64 Q6_P_vmpywoh_PP_sat(Word64 Rss, Word64 Rtt) 470
    Rxx+=vmpywoh(Rss,Rtt):<<1:rnd:sat
        Word64 Q6_P_vmpywohacc_PP_s1_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)
               470
    Rxx+=vmpywoh(Rss,Rtt):<<1:sat
        Word64 Q6_P_vmpywohacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt) 470
    Rxx+=vmpywoh(Rss,Rtt):rnd:sat
        Word64 Q6_P_vmpywohacc_PP_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt) 470
    Rxx+=vmpywoh(Rss,Rtt):sat
```

```
        Word64 Q6_P_vmpywohacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt) 470

vmpywouh
    Rdd=vmpywouh(Rss,Rtt):<<1:rnd:sat
        Word64 Q6_P_vmpywouh_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt) 473
    Rdd=vmpywouh(Rss,Rtt):<<1:sat
        Word64 Q6_P_vmpywouh_PP_s1_sat(Word64 Rss, Word64 Rtt) 473
    Rdd=vmpywouh(Rss,Rtt):rnd:sat
        Word64 Q6_P_vmpywouh_PP_rnd_sat(Word64 Rss, Word64 Rtt) 473
    Rdd=vmpywouh(Rss,Rtt):sat
        Word64 Q6_P_vmpywouh_PP_sat(Word64 Rss, Word64 Rtt) 473
    Rxx+=vmpywouh(Rss,Rtt):<<1:rnd:sat
        Word64 Q6_P_vmpywouhacc_PP_s1_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)
               473
    Rxx+=vmpywouh(Rss,Rtt):<<1:sat
        Word64 Q6_P_vmpywouhacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt) 473
    Rxx+=vmpywouh(Rss,Rtt):rnd:sat
        Word64 Q6_P_vmpywouhacc_PP_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt) 473
    Rxx+=vmpywouh(Rss,Rtt):sat
        Word64 Q6_P_vmpywouhacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt) 473

vmux
    Rdd=vmux(Pu,Rss,Rtt)
        Word64 Q6_P_vmux_pPP(Byte Pu, Word64 Rss, Word64 Rtt) 555

vnavgh
    Rd=vnavgh(Rt,Rs)
        Word32 Q6_R_vnavgh_RR(Word32 Rt, Word32 Rs) 169
    Rdd=vnavgh(Rtt,Rss)
        Word64 Q6_P_vnavgh_PP(Word64 Rtt, Word64 Rss) 370
    Rdd=vnavgh(Rtt,Rss):crnd:sat
        Word64 Q6_P_vnavgh_PP_crnd_sat(Word64 Rtt, Word64 Rss) 370
    Rdd=vnavgh(Rtt,Rss):rnd:sat
        Word64 Q6_P_vnavgh_PP_rnd_sat(Word64 Rtt, Word64 Rss) 370

vnavgw
    Rdd=vnavgw(Rtt,Rss)
        Word64 Q6_P_vnavgw_PP(Word64 Rtt, Word64 Rss) 373
    Rdd=vnavgw(Rtt,Rss):crnd:sat
        Word64 Q6_P_vnavgw_PP_crnd_sat(Word64 Rtt, Word64 Rss) 373
    Rdd=vnavgw(Rtt,Rss):rnd:sat
        Word64 Q6_P_vnavgw_PP_rnd_sat(Word64 Rtt, Word64 Rss) 373

vpmpyh
    Rdd=vpmpyh(Rs,Rt)
        Word64 Q6_P_vpmpyh_RR(Word32 Rs, Word32 Rt) 515
    Rxx^=vpmpyh(Rs,Rt)
        Word64 Q6_P_vpmpyhxacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt) 515

vraddh
    Rd=vraddh(Rss,Rtt)
        Word32 Q6_R_vraddh_PP(Word64 Rss, Word64 Rtt) 365

vraddub
    Rdd=vraddub(Rss,Rtt)
        Word64 Q6_P_vraddub_PP(Word64 Rss, Word64 Rtt) 363
    Rxx+=vraddub(Rss,Rtt)
        Word64 Q6_P_vraddubacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt) 363
```

vradduh
Rd=vradduh(Rss,Rtt)
Word32 Q6_R_vradduh_PP(Word64 Rss, Word64 Rtt) 365

vrcmpys
Rd=vrcmpys(Rss,Rt):<<1:rnd:sat
Word32 Q6_R_vrcmpys_PR_s1_rnd_sat(Word64 Rss, Word32 Rt) 440
Rdd=vrcmpys(Rss,Rt):<<1:sat
Word64 Q6_P_vrcmpys_PR_s1_sat(Word64 Rss, Word32 Rt) 437
Rxx+=vrcmpys(Rss,Rt):<<1:sat
Word64 Q6_P_vrcmpysacc_PR_s1_sat(Word64 Rxx, Word64 Rss, Word32 Rt) 437

vrcnegh
Rxx+=vrcnegh(Rss,Rt)
Word64 Q6_P_vrcneghacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt) 375

vrcrotate
Rdd=vrcrotate(Rss,Rt,#u2)
Word64 Q6_P_vrcrotate_PRI(Word64 Rss, Word32 Rt, Word32 Iu2) 443
Rxx+=vrcrotate(Rss,Rt,#u2)
Word64 Q6_P_vrcrotateacc_PRI(Word64 Rxx, Word64 Rss, Word32 Rt, Word32 Iu2)
443

vrmaxh
Rxx=vrmaxh(Rss,Ru)
Word64 Q6_P_vrmaxh_PR(Word64 Rxx, Word64 Rss, Word32 Ru) 378

vrmaxuh
Rxx=vrmaxuh(Rss,Ru)
Word64 Q6_P_vrmaxuh_PR(Word64 Rxx, Word64 Rss, Word32 Ru) 378

vrmaxuw
Rxx=vrmaxuw(Rss,Ru)
Word64 Q6_P_vrmaxuw_PR(Word64 Rxx, Word64 Rss, Word32 Ru) 380

vrmaxw
Rxx=vrmaxw(Rss,Ru)
Word64 Q6_P_vrmaxw_PR(Word64 Rxx, Word64 Rss, Word32 Ru) 380

vrminh
Rxx=vrminh(Rss,Ru)
Word64 Q6_P_vrminh_PR(Word64 Rxx, Word64 Rss, Word32 Ru) 385

vrminuh
Rxx=vrminuh(Rss,Ru)
Word64 Q6_P_vrminuh_PR(Word64 Rxx, Word64 Rss, Word32 Ru) 385

vrminuw
Rxx=vrminuw(Rss,Ru)
Word64 Q6_P_vrminuw_PR(Word64 Rxx, Word64 Rss, Word32 Ru) 386

vrminw
Rxx=vrminw(Rss,Ru)
Word64 Q6_P_vrminw_PR(Word64 Rxx, Word64 Rss, Word32 Ru) 386

vrmpybsu
Rdd=vrmpybsu(Rss,Rtt)
Word64 Q6_P_vrmpybsu_PP(Word64 Rss, Word64 Rtt) 500
Rxx+=vrmpybsu(Rss,Rtt)
Word64 Q6_P_vrmpybsuacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt) 500

```
vrmpybu
    Rdd=vrmpybu(Rss,Rtt)
        Word64 Q6_P_vrmpybu_PP(Word64 Rss, Word64 Rtt) 500
    Rxx+=vrmpybu(Rss,Rtt)
        Word64 Q6_P_vrmpybuacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt) 500

vrmpyh
    Rdd=vrmpyh(Rss,Rtt)
        Word64 Q6_P_vrmpyh_PP(Word64 Rss, Word64 Rtt) 510
    Rxx+=vrmpyh(Rss,Rtt)
        Word64 Q6_P_vrmpyhacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt) 510

vrmpyweh
    Rdd=vrmpyweh(Rss,Rtt)
        Word64 Q6_P_vrmpyweh_PP(Word64 Rss, Word64 Rtt) 490
    Rdd=vrmpyweh(Rss,Rtt):<<1
        Word64 Q6_P_vrmpyweh_PP_s1(Word64 Rss, Word64 Rtt) 490
    Rxx+=vrmpyweh(Rss,Rtt)
        Word64 Q6_P_vrmpywehacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt) 490
    Rxx+=vrmpyweh(Rss,Rtt):<<1
        Word64 Q6_P_vrmpywehacc_PP_s1(Word64 Rxx, Word64 Rss, Word64 Rtt) 490

vrmpywoh
    Rdd=vrmpywoh(Rss,Rtt)
        Word64 Q6_P_vrmpywoh_PP(Word64 Rss, Word64 Rtt) 490
    Rdd=vrmpywoh(Rss,Rtt):<<1
        Word64 Q6_P_vrmpywoh_PP_s1(Word64 Rss, Word64 Rtt) 490
    Rxx+=vrmpywoh(Rss,Rtt)
        Word64 Q6_P_vrmpywohacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt) 490
    Rxx+=vrmpywoh(Rss,Rtt):<<1
        Word64 Q6_P_vrmpywohacc_PP_s1(Word64 Rxx, Word64 Rss, Word64 Rtt) 490

vrndwh
    Rd=vrndwh(Rss)
        Word32 Q6_R_vrndwh_P(Word64 Rss) 522
    Rd=vrndwh(Rss):sat
        Word32 Q6_R_vrndwh_P_sat(Word64 Rss) 522

vrsadub
    Rdd=vrsadub(Rss,Rtt)
        Word64 Q6_P_vrsadub_PP(Word64 Rss, Word64 Rtt) 390
    Rxx+=vrsadub(Rss,Rtt)
        Word64 Q6_P_vrsadubacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt) 390

vsathb
    Rd=vsathb(Rs)
        Word32 Q6_R_vsathb_R(Word32 Rs) 525
    Rd=vsathb(Rss)
        Word32 Q6_R_vsathb_P(Word64 Rss) 525
    Rdd=vsathb(Rss)
        Word64 Q6_P_vsathb_P(Word64 Rss) 527

vsathub
    Rd=vsathub(Rs)
        Word32 Q6_R_vsathub_R(Word32 Rs) 525
    Rd=vsathub(Rss)
        Word32 Q6_R_vsathub_P(Word64 Rss) 525
    Rdd=vsathub(Rss)
        Word64 Q6_P_vsathub_P(Word64 Rss) 527
```

```
vsatwh
   Rd=vsatwh(Rss)
      Word32 Q6_R_vsatwh_P(Word64 Rss) 525
   Rdd=vsatwh(Rss)
      Word64 Q6_P_vsatwh_P(Word64 Rss) 527

vsatwuh
   Rd=vsatwuh(Rss)
      Word32 Q6_R_vsatwuh_P(Word64 Rss) 525
   Rdd=vsatwuh(Rss)
      Word64 Q6_P_vsatwuh_P(Word64 Rss) 527

vsplatb
   Rd=vsplatb(Rs)
      Word32 Q6_R_vsplatb_R(Word32 Rs) 530
   Rdd=vsplatb(Rs)
      Word64 Q6_P_vsplatb_R(Word32 Rs) 530

vsplath
   Rdd=vsplath(Rs)
      Word64 Q6_P_vsplath_R(Word32 Rs) 531

vspliceb
   Rdd=vspliceb(Rss,Rtt,#u3)
      Word64 Q6_P_vspliceb_PPI(Word64 Rss, Word64 Rtt, Word32 Iu3) 532
   Rdd=vspliceb(Rss,Rtt,Pu)
      Word64 Q6_P_vspliceb_PPp(Word64 Rss, Word64 Rtt, Byte Pu) 532

vsubb
   Rdd=vsubb(Rss,Rtt)
      Word64 Q6_P_vsubb_PP(Word64 Rss, Word64 Rtt) 393

vsubh
   Rd=vsubh(Rt,Rs)
      Word32 Q6_R_vsubh_RR(Word32 Rt, Word32 Rs) 170
   Rd=vsubh(Rt,Rs):sat
      Word32 Q6_R_vsubh_RR_sat(Word32 Rt, Word32 Rs) 170
   Rdd=vsubh(Rtt,Rss)
      Word64 Q6_P_vsubh_PP(Word64 Rtt, Word64 Rss) 391
   Rdd=vsubh(Rtt,Rss):sat
      Word64 Q6_P_vsubh_PP_sat(Word64 Rtt, Word64 Rss) 391

vsubub
   Rdd=vsubub(Rtt,Rss)
      Word64 Q6_P_vsubub_PP(Word64 Rtt, Word64 Rss) 393
   Rdd=vsubub(Rtt,Rss):sat
      Word64 Q6_P_vsubub_PP_sat(Word64 Rtt, Word64 Rss) 393

vsubuh
   Rd=vsubuh(Rt,Rs):sat
      Word32 Q6_R_vsubuh_RR_sat(Word32 Rt, Word32 Rs) 170
   Rdd=vsubuh(Rtt,Rss):sat
      Word64 Q6_P_vsubuh_PP_sat(Word64 Rtt, Word64 Rss) 391

vsubw
   Rdd=vsubw(Rtt,Rss)
      Word64 Q6_P_vsubw_PP(Word64 Rtt, Word64 Rss) 394
   Rdd=vsubw(Rtt,Rss):sat
      Word64 Q6_P_vsubw_PP_sat(Word64 Rtt, Word64 Rss) 394
```

vsxtbh
Rdd=vsxtbh(Rs)
Word64 Q6_P_vsxtbh_R(Word32 Rs) 533

vsxthw
Rdd=vsxthw(Rs)
Word64 Q6_P_vsxthw_R(Word32 Rs) 533

vtrunehb
Rd=vtrunehb(Rss)
Word32 Q6_R_vtrunehb_P(Word64 Rss) 535
Rdd=vtrunehb(Rss,Rtt)
Word64 Q6_P_vtrunehb_PP(Word64 Rss, Word64 Rtt) 535

vtrunewh
Rdd=vtrunewh(Rss,Rtt)
Word64 Q6_P_vtrunewh_PP(Word64 Rss, Word64 Rtt) 535

vtrunohb
Rd=vtrunohb(Rss)
Word32 Q6_R_vtrunohb_P(Word64 Rss) 535
Rdd=vtrunohb(Rss,Rtt)
Word64 Q6_P_vtrunohb_PP(Word64 Rss, Word64 Rtt) 535

vtrunowh
Rdd=vtrunowh(Rss,Rtt)
Word64 Q6_P_vtrunowh_PP(Word64 Rss, Word64 Rtt) 535

vxaddsubh
Rdd=vxaddsubh(Rss,Rtt):rnd:>>1:sat
Word64 Q6_P_vxaddsubh_PP_rnd_rs1_sat(Word64 Rss, Word64 Rtt) 415
Rdd=vxaddsubh(Rss,Rtt):sat
Word64 Q6_P_vxaddsubh_PP_sat(Word64 Rss, Word64 Rtt) 415

vxaddsubw
Rdd=vxaddsubw(Rss,Rtt):sat
Word64 Q6_P_vxaddsubw_PP_sat(Word64 Rss, Word64 Rtt) 416

vxsubaddh
Rdd=vxsubaddh(Rss,Rtt):rnd:>>1:sat
Word64 Q6_P_vxsubaddh_PP_rnd_rs1_sat(Word64 Rss, Word64 Rtt) 415
Rdd=vxsubaddh(Rss,Rtt):sat
Word64 Q6_P_vxsubaddh_PP_sat(Word64 Rss, Word64 Rtt) 415

vxsubaddw
Rdd=vxsubaddw(Rss,Rtt):sat
Word64 Q6_P_vxsubaddw_PP_sat(Word64 Rss, Word64 Rtt) 416

vzxtbh
Rdd=vzxtbh(Rs)
Word64 Q6_P_vzxtbh_R(Word32 Rs) 536

vzxthw
Rdd=vzxthw(Rs)
Word64 Q6_P_vzxthw_R(Word32 Rs) 536

X

xor
Pd=xor(Ps,Pt)

Byte Q6_p_xor_pp(Byte Ps, Byte Pt) 202
Rd=xor(Rs,Rt)
Word32 Q6_R_xor_RR(Word32 Rs, Word32 Rt) 159
Rdd=xor(Rss,Rtt)
Word64 Q6_P_xor_PP(Word64 Rss, Word64 Rtt) 335
Rx&=xor(Rs,Rt)
Word32 Q6_R_xorand_RR(Word32 Rx, Word32 Rs, Word32 Rt) 337
Rx^=xor(Rs,Rt)
Word32 Q6_R_xorxacc_RR(Word32 Rx, Word32 Rs, Word32 Rt) 337
Rx|=xor(Rs,Rt)
Word32 Q6_R_xoror_RR(Word32 Rx, Word32 Rs, Word32 Rt) 338
Rxx^=xor(Rss,Rtt)
Word64 Q6_P_xorxacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt) 336

Z

zxtb
Rd=zxtb(Rs)
Word32 Q6_R_zxtb_R(Word32 Rs) 171

zxth
Rd=zxth(Rs)
Word32 Q6_R_zxth_R(Word32 Rs) 171