Cereon

Architecture Reference Manual
Version 1

Authors:
Andrei Kapustin

Revision history

Date	Comment
15 June 2006	Initial draft.
22 June 2006	Documented bit manipulation instructions.
2 July 2006	Added the B (Base) bit to CPUID long word.
1 September 2006	Documented arithmetical & shift instructions for 8-, 16- and 32-bit values (BASE3/BASE5 opcode spaces). Documented arithmetical & conversion instructions for single precision floating point values.
2 September 2006	Changed instruction mnemonics to use two-level instruction naming scheme.
2 September 2006	Defined less-than-long-word shift/FP conversion instructions.
10 October 2006	Defined 8-, 16- and 32-bit immediate arithmetic instructions (BASE4 opcode space).
23 November 2006	Modified definitions of `ldm`/`stm` instructions to adjust base register.
13 December 2006	Defined `not.b`, `not.h` and `not.w` instructions.
14 December 2006	Redefined shift instructions to always treat shift counter as an unsigned value.
20 December 2006	Modified bootstrapping sequence to allow `HARDWARE` interrupts upon reset. Updated definition of `ldm` and `stm` instruction to disallow using one of the loaded/stored registers as a base.
4 January 2007	Defined less-than-long-word logical & FP-conversion instructions (BASE5 opcode space). Did some serious reworking of the instruction encoding scheme to keep encodings of similar instructions close.
8 January 2007	Defined “compare-with-immediate-and-branch” instructions.

9 January 2007	Defined “2’s complement of an unsigned value” instructions.
11 January 2007	Added the `A` (accessed) and `D` (dirty) bits to virtual page descriptors. Defined the “for current process only” version of cache/TLB control instructions.
1 March 2007	Updated the definition of memory barrier instructions to include TLB invalidation.
2 March 2007	`ZDIV` exception is now maskable. The `$flags` register has been introduced for user-mode program event tracking.
5 March 2007	Made virtual memory an optional feature and documented protected memory feature.
23 March 2007	Split `PAGEFAULT` exception into `IPAGEFAULT` and `DPAGEFAULT` and made sure the virtual page address that caused an exception is available to the handler as part of the exception code.
25 March 2007	Allowed `mov.cr` instruction in user mode (this will allow user code to read current state and cycle counter). Allowed `cpuid` instruction in user mode (this will allow user code to determine the properties of a CPU it’s running on).
30 June 2007	Defined the performance monitoring feature.
5 July 2007	Redesigned I/O subsystem for each port to occupy a single address regardless of its size.
4 November 2007	Fixed a few typos; made minor cosmetic changes.
30 November 2007	Added the on-chip memory specification.
20 February 2008	Disallowed the use if `$ip` as a base register in `ldm/stm`.
17 March 2008	Arithmetic shifts left will now preserve the sign bit.
18 March 2008	Fixed erroneous bit patterns for several FP instructions.
29 July 2008	Made DMA channel instructions 32-bit long and added immediate DMA shifts.
26 July 2010	Updated the Pending Trap behaviour – PT is now checked before each instruction, not after.
29 July 2010	The `brk` instruction now embeds the break code as an immediate operand.
15 October 2014	Allowed signed `div` instructions to cause integer overflow when maximum representable negative number is divided by -1, thus producing a negative result.

1 Contents

1 CONTENTS 4

2 OVERVIEW 16

2.1 MAIN FEATURES 16

3 INTRODUCTION TO CEREON 17

3.1 ARCHITECTURE 17
3.2 FEATURES 18
3.3 REGISTERS 19
3.4 INTERRUPTS 19
3.5 PROCESSOR STATE 20
3.6 PROCESSOR IDENTIFICATION 20

4 PROGRAMMER'S MODEL 21

4.1 DATA TYPES 21
4.1.1 Integer types 21
4.1.2 Real types 21
4.2 PROCESSOR MODES 22
4.2.1 Kernel vs. User 22
4.2.2 Real vs. Virtual 23
4.2.3 Big-endian vs. Little-endian 23
4.2.4 Working vs. Idle 23
4.3 REGISTERS 23
4.3.1 General purpose registers 24
4.3.1.1 $rv/$frv 25
4.3.1.2 $a0..$a3/$fa0..$fa3 25
4.3.1.3 $t0..$t7/$ft0..$ft7 25
4.3.1.4 $s0..$s12/$fs0..$fs12 25
4.3.1.5 $gp 25
4.3.1.6 $sp 25
4.3.1.7 $fp 26
4.3.1.8 $dp 26
4.3.1.9 $ra 26
4.3.1.10 $ip 26
4.3.2 Control registers 26
4.3.2.1 $state 27
4.3.2.1.1 K (Kernel mode) 28
4.3.2.1.2 V (Virtual mode) 28
4.3.2.1.3 T (Trap mode) and P (Pending trap) 28
4.3.2.1.4 B (Big-endian) 28
4.3.2.1.5 W (Working) 28
4.3.2.1.6 D (Debug events) 29
4.3.2.1.7 R (real opeRand) 29
4.3.2.1.8 Z (real division by Zero) 29
4.3.2.1.9 E (real ovErflow) 29
4.3.2.1.10 U (real Underflow) 29
4.3.2.1.11 I (real Inexact) 29
4.3.2.1.12 N (iNteger division by zero) 29
4.3.2.1.13 O (integer Overflow) 29
4.3.2.1.14 RM (Rounding Mode) 29
4.3.2.1.15 IM (Interrupt Mask) 30
4.3.2.1.16 CID (Context ID) 30
4.3.2.2 $pth 30
4.3.2.3 $itc 30
4.3.2.4 $cc 30
4.3.3 Debug registers 31
4.3.4 Performance monitoring registers 31
4.3.5 $flags register 31

4.3.5.1.1 R (real opeRand) 31
4.3.5.1.2 Z (real division by Zero) 31
4.3.5.1.3 E (real ovErflow) 31
4.3.5.1.4 U (real Underflow) 31
4.3.5.1.5 I (real Inexact) 32
4.3.5.1.6 N (iNteger division by zero) 32
4.3.5.1.7 O (integer Overflow) 32
4.4 MEMORY 32
4.4.1 Alignment 32
4.4.2 Caches and write buffers 33
4.4.3 Memory barriers 33
5 CEREON INSTRUCTION SET 35
5.1 INSTRUCTION FORMAT 35
5.1.1 I-type (Immediate) 35
5.1.2 L-type (Long immediate) 35
5.1.3 J-type (jump) 35
5.1.4 R-type (register) 36
5.2 INSTRUCTION MNEMONICS 36
5.3 INSTRUCTION SET MATRIX 37
5.3.1 Encoding of the opcode field 37
5.3.2 COP1 opcode encoding of the function field 38
5.3.3 SHIFT1 encoding of the r3 field 38
5.3.4 SHIFT2 encoding of the r3 field 38
5.3.5 BASE1 function encoding of the shift amount field 38
5.3.6 BASE2 function encoding of the shift amount field 39
5.3.7 BASE3 function encoding of the shift amount field 39
5.3.8 BASE4 encoding of the r3 field 39
5.3.9 BASE5 function encoding of the shift amount field 40
5.3.10 FP1 function encoding of the shift amount field 40
5.3.11 DBG1 function encoding of the shift amount field 40
5.3.12 PM1 function encoding of the shift amount field 41
5.4 INSTRUCTION SET REFERENCE 41
6 THE CEREON BASE 43
6.1 UNSUPPORTED INSTRUCTIONS 43
6.1.1 invi (invalid Instruction) 43
6.2 DATA MOVEMENT 43
6.2.1 mov.l (move long word) 43
6.2.2 mov.cr (move control register to general purpose register) 43
6.2.3 mov.rc (move general purpose register to control register) [privileged] 43
6.2.4 li.l (load lmmediate long word) 44
6.2.5 swap.h (swap bytes in half-word) 44
6.2.6 swap.uh (swap bytes in unsigned half-word) 44
6.2.7 swap.w (swap bytes in word) 44
6.2.8 swap.uw (swap bytes in unsigned word) 45
6.2.9 swap.l (swap bytes in long word) 45
6.3 ARITHMETIC INSTRUCTIONS 45
6.3.1 add.b (add byte) 45
6.3.2 add.ub (add unsigned byte) 45
6.3.3 add.h (add half-word) 46
6.3.4 add.uh (add unsigned half-word) 46
6.3.5 add.w (add word) 46
6.3.6 add.uw (add unsigned word) 47
6.3.7 add.l (add long word) 47
6.3.8 add.ul (add unsigned long word) 47
6.3.9 sub.b (subtract byte) 48
6.3.10 sub.ub (subtract unsigned byte) 48
6.3.11 sub.h (subtract half-word) 48

6.3.12 sub.uh (subtract unsigned half-word) ... 49
6.3.13 sub.w (subtract word) ... 49
6.3.14 sub.uw (subtract unsigned word) ... 49
6.3.15 sub.l (subtract long word) ... 50
6.3.16 sub.ul (subtract unsigned long word) ... 50
6.3.17 mul.b (multiply byte) ... 50
6.3.18 mul.ub (multiply unsigned byte) ... 50
6.3.19 mul.h (multiply half-word) ... 51
6.3.20 mul.uh (multiply unsigned half-word) ... 51
6.3.21 mul.w (multiply word) ... 51
6.3.22 mul.uw (multiply unsigned word) ... 52
6.3.23 mul.l (multiply long word) ... 52
6.3.24 mul.ul (multiply unsigned long word) ... 52
6.3.25 div.b (divide byte) ... 53
6.3.26 div.ub (divide unsigned byte) ... 53
6.3.27 div.h (divide half-word) ... 53
6.3.28 div.uh (divide unsigned half-word) ... 54
6.3.29 div.w (divide word) ... 54
6.3.30 div.uw (divide unsigned word) ... 54
6.3.31 div.l (divide long word) ... 55
6.3.32 div.ul (divide unsigned long word) ... 55
6.3.33 mod.b (modulo byte) ... 55
6.3.34 mod.ub (modulo unsigned byte) ... 56
6.3.35 mod.h (modulo half-word) ... 56
6.3.36 mod.uh (modulo unsigned half-word) ... 56
6.3.37 mod.w (modulo word) ... 57
6.3.38 mod.uw (modulo unsigned word) ... 57
6.3.39 mod.l (modulo long word) ... 57
6.3.40 mod.ul (modulo unsigned long word) ... 57
6.3.41 abs.b (absolute value byte) ... 58
6.3.42 abs.h (absolute value half-word) ... 58
6.3.43 abs.w (absolute value word) ... 58
6.3.44 abs.l (absolute value long word) ... 59
6.3.45 neg.b (negate byte) ... 59
6.3.46 cpl2.ub (2’s complement of an unsigned byte) ... 59
6.3.47 neg.h (negate half-word) ... 60
6.3.48 cpl2.uh (2’s complement of an unsigned half-word) ... 60
6.3.49 neg.w (negate word) ... 60
6.3.50 cpl2.uw (2’s complement of an unsigned word) ... 61
6.3.51 neg.l (negate long word) ... 61
6.3.52 cpl2.ul (2’s complement of an unsigned long word) ... 61
6.3.53 addi.b (add immediate byte) ... 61
6.3.54 addi.ub (add immediate unsigned byte) ... 62
6.3.55 addi.h (add immediate half-word) ... 62
6.3.56 addi.uh (add immediate unsigned half-word) ... 62
6.3.57 addi.w (add immediate word) ... 63
6.3.58 addi.uw (add immediate unsigned word) ... 63
6.3.59 addi.l (add immediate long word) ... 63
6.3.60 addi.ul (add immediate unsigned long word) ... 64
6.3.61 subi.b (subtract immediate byte) ... 64
6.3.62 subi.ub (subtract immediate unsigned byte) ... 64
6.3.63 subi.h (subtract immediate half-word) ... 65
6.3.64 subi.uh (subtract immediate unsigned half-word) ... 65
6.3.65 subi.w (subtract immediate word) ... 65
6.3.66 subi.uw (subtract immediate unsigned word) ... 66
6.3.67 subi.l (subtract immediate long word) ... 66
6.3.68 subi.ul (subtract immediate unsigned long word) ... 66
6.3.69 muli.b (multiply immediate byte) ... 67
6.3.70 muli.ub (multiply immediate unsigned byte) ... 67
6.3.71 muli.h (multiply immediate half-word) ... 67

6.3.72 muli.uh (multiply immediate unsigned half-word) ... 68
6.3.73 muli.w (multiply immediate word) ... 68
6.3.74 muli.uw (multiply immediate unsigned word) ... 68
6.3.75 muli.l (multiply immediate long word) ... 69
6.3.76 muli.ul (multiply immediate unsigned long word) ... 69
6.3.77 divi.b (divide immediate byte) ... 69
6.3.78 divi.ub (divide immediate unsigned byte) ... 70
6.3.79 divi.h (divide immediate half-word) ... 70
6.3.80 divi.uh (divide immediate unsigned half-word) ... 70
6.3.81 divi.w (divide immediate word) ... 71
6.3.82 divi.uw (divide immediate unsigned word) ... 71
6.3.83 divi.l (divide immediate long word) ... 71
6.3.84 divi.ul (divide immediate unsigned long word) ... 72
6.3.85 modi.b (modulo immediate byte) ... 72
6.3.86 modi.ub (modulo immediate unsigned byte) ... 72
6.3.87 modi.h (modulo immediate half-word) ... 73
6.3.88 modi.uh (modulo immediate unsigned half-word) ... 73
6.3.89 modi.w (modulo immediate word) ... 73
6.3.90 modi.uw (modulo immediate unsigned word) ... 74
6.3.91 modi.l (modulo immediate long word) ... 74
6.3.92 modi.ul (modulo immediate unsigned long word) ... 74
6.4 BOOLEAN INSTRUCTIONS ... 75
6.4.1 and.b (bitwise and byte) ... 75
6.4.2 and.ub (bitwise and unsigned byte) ... 75
6.4.3 and.h (bitwise and half-word) ... 75
6.4.4 and.uh (bitwise and unsigned half-word) ... 75
6.4.5 and.w (bitwise and word) ... 75
6.4.6 and.uw (bitwise and unsigned word) ... 76
6.4.7 and.l (bitwise and long word) ... 76
6.4.8 or.b (bitwise or byte) ... 76
6.4.9 or.ub (bitwise or unsigned byte) ... 76
6.4.10 or.h (bitwise or half-word) ... 76
6.4.11 or.uh (bitwise or unsigned half-word) ... 77
6.4.12 or.w (bitwise or word) ... 77
6.4.13 or.uw (bitwise or unsigned word) ... 77
6.4.14 or.l (bitwise or long word) ... 77
6.4.15 xor.b (bitwise exclusive or byte) ... 77
6.4.16 xor.ub (bitwise exclusive or unsigned byte) ... 78
6.4.17 xor.h (bitwise exclusive or half-word) ... 78
6.4.18 xor.uh (bitwise exclusive or unsigned half-word) ... 78
6.4.19 xor.w (bitwise exclusive or word) ... 78
6.4.20 xor.uw (bitwise exclusive or unsigned word) ... 78
6.4.21 xor.l (bitwise exclusive or long word) ... 79
6.4.22 impl.b (bitwise implication byte) ... 79
6.4.23 impl.ub (bitwise implication unsigned byte) ... 79
6.4.24 impl.h (bitwise implication half-word) ... 79
6.4.25 impl.uh (bitwise implication unsigned half-word) ... 79
6.4.26 impl.w (bitwise implication word) ... 80
6.4.27 impl.uw (bitwise implication unsigned word) ... 80
6.4.28 impl.l (bitwise implication long word) ... 80
6.4.29 andi.l (bitwise and immediate long word) ... 80
6.4.30 ori.l (bitwise or immediate long word) ... 80
6.4.31 xori.l (bitwise exclusive or immediate long word) ... 81
6.4.32 impli.l (bitwise implication immediate long word) ... 81
6.4.33 not.b (bitwise not byte) ... 81
6.4.34 not.ub (bitwise not unsigned byte) ... 81
6.4.35 not.h (bitwise not half-word) ... 81
6.4.36 not.uh (bitwise not unsigned half-word) ... 82
6.4.37 not.w (bitwise not word) ... 82
6.4.38 not.uw (bitwise not unsigned word) ... 82

6.4.39 not.l (bitwise not long word) 82
6.5 SHIFT INSTRUCTIONS 83
6.5.1 shl.b (shift left byte) 83
6.5.2 shl.ub (shift left unsigned byte) 83
6.5.3 shl.h (shift left half-word) 83
6.5.4 shl.uh (shift left unsigned half-word) 83
6.5.5 shl.w (shift left word) 84
6.5.6 shl.uw (shift left unsigned word) 84
6.5.7 shl.l (shift left long word) 84
6.5.8 shr.b (shift right byte) 84
6.5.9 shr.ub (shift right unsigned byte) 84
6.5.10 shr.h (shift right half-word) 85
6.5.11 shr.uh (shift right unsigned half-word) 85
6.5.12 shr.w (shift right word) 85
6.5.13 shr.uw (shift right unsigned word) 85
6.5.14 shr.l (shift right long word) 86
6.5.15 asl.b (arithmetic shift left byte) 86
6.5.16 asl.ub (arithmetic shift left unsigned byte) 86
6.5.17 asl.h (arithmetic shift left half-word) 86
6.5.18 asl.uh (arithmetic shift left unsigned half-word) 86
6.5.19 asl.w (arithmetic shift left word) 87
6.5.20 asl.uw (arithmetic shift left unsigned word) 87
6.5.21 asl.l (arithmetic shift left long word) 87
6.5.22 asr.b (arithmetic shift right byte) 87
6.5.23 asr.ub (arithmetic shift right unsigned byte) 88
6.5.24 asr.h (arithmetic shift right half-word) 88
6.5.25 asr.uh (arithmetic shift right unsigned half-word) 88
6.5.26 asr.w (arithmetic shift right word) 88
6.5.27 asr.uw (arithmetic shift right unsigned word) 89
6.5.28 asr.l (arithmetic shift right long word) 89
6.5.29 rol.b (rotate left byte) 89
6.5.30 rol.ub (rotate left unsigned byte) 89
6.5.31 rol.h (rotate left half-word) 89
6.5.32 rol.uh (rotate left unsigned half-word) 90
6.5.33 rol.w (rotate left word) 90
6.5.34 rol.uw (rotate left unsigned word) 90
6.5.35 rol.l (rotate left) 90
6.5.36 ror.b (rotate right byte) 91
6.5.37 ror.ub (rotate right unsigned byte) 91
6.5.38 ror.h (rotate right half-word) 91
6.5.39 ror.uh (rotate right unsigned half-word) 91
6.5.40 ror.w (rotate right word) 92
6.5.41 ror.uw (rotate right unsigned word) 92
6.5.42 ror.l (rotate right long word) 92
6.5.43 shli.b (shift left immediate byte) 92
6.5.44 shli.ub (shift left immediate unsigned byte) 93
6.5.45 shli.h (shift left immediate half-word) 93
6.5.46 shli.uh (shift left immediate unsigned half-word) 93
6.5.47 shli.w (shift left immediate word) 93
6.5.48 shli.uw (shift left immediate unsigned word) 94
6.5.49 shli.l (shift left immediate long word) 94
6.5.50 shri.b (shift right immediate byte) 94
6.5.51 shri.ub (shift right immediate unsigned byte) 94
6.5.52 shri.h (shift right immediate half-word) 94
6.5.53 shri.uh (shift right immediate unsigned half-word) 95
6.5.54 shri.w (shift right immediate word) 95
6.5.55 shri.uw (shift right immediate unsigned word) 95
6.5.56 shri.l (shift right immediate long word) 95
6.5.57 asli.b (arithmetic shift left immediate byte) 96
6.5.58 asli.ub (arithmetic shift left immediate unsigned byte) 96

6.5.59 asli.h (arithmetic shift left immediate half-word) 96
6.5.60 asli.uh (arithmetic shift left immediate unsigned half-word) 96
6.5.61 asli.w (arithmetic shift left immediate word) 97
6.5.62 asli.uw (arithmetic shift left immediate unsigned word) 97
6.5.63 asli.l (arithmetic shift left immediate long word) 97
6.5.64 asri.b (arithmetic shift right immediate byte) 97
6.5.65 asri.ub (arithmetic shift right immediate unsigned byte) 98
6.5.66 asri.h (arithmetic shift right immediate half-word) 98
6.5.67 asri.uh (arithmetic shift right immediate unsigned half-word) 98
6.5.68 asri.w (arithmetic shift right immediate word) 98
6.5.69 asri.uw (arithmetic shift right immediate unsigned word) 99
6.5.70 asri.l (arithmetic shift right immediate long word) 99
6.5.71 roli.b (rotate left immediate byte) 99
6.5.72 roli.ub (rotate left immediate unsigned byte) 99
6.5.73 roli.h (rotate left immediate half-word) 100
6.5.74 roli.uh (rotate left immediate unsigned half-word) 100
6.5.75 roli.w (rotate left immediate word) 100
6.5.76 roli.uw (rotate left immediate unsigned word) 100
6.5.77 roli.l (rotate left immediate long word) 101
6.5.78 rori.b (rotate right immediate byte) 101
6.5.79 rori.ub (rotate right immediate unsigned byte) 101
6.5.80 rori.h (rotate right immediate half-word) 101
6.5.81 rori.uh (rotate right immediate unsigned half-word) 101
6.5.82 rori.w (rotate right immediate word) 102
6.5.83 rori.uw (rotate right immediate unsigned word) 102
6.5.84 rori.l (rotate right immediate long word) 102
6.6 BIT MANIPULATION INSTRUCTIONS 102
6.6.1 clz (Count Leading Zeroes) 102
6.6.2 ctz (Count Tailing Zeroes) 103
6.6.3 clo (Count Leading Ones) 103
6.6.4 cto (Count Tailing Ones) 103
6.6.5 bfe.l (bit field extract from long word) 103
6.6.6 bfe.ul (bit field extract from unsigned long word) 104
6.6.7 bfi.l (bit field inject into long word) 104
6.6.8 brev.b (bit reversal in byte) 104
6.6.9 brev.ub (bit reversal in unsigned byte) 105
6.6.10 brev.h (bit reversal in half-word) 105
6.6.11 brev.uh (bit reversal in unsigned half-word) 105
6.6.12 brev.w (bit reversal in word) 105
6.6.13 brev.uw (bit reversal in unsigned word) 106
6.6.14 brev.l (bit reversal in long word) 106
6.7 DATA TYPE CONVERSION INSTRUCTIONS 106
6.7.1 cvt.bl (convert byte to long word) 106
6.7.2 cvt.ubl (convert unsigned byte to long word) 106
6.7.3 cvt.hl (convert half-word to long word) 107
6.7.4 cvt.uhl (convert unsigned half-word to long word) 107
6.7.5 cvt.wl (convert word to long word) 107
6.7.6 cvt.uwl (convert unsigned word to long word) 108
6.8 COMPARISON INSTRUCTIONS 108
6.8.1 seq.l (set equal long word) 108
6.8.2 sne.l (set not equal long word) 108
6.8.3 slt.l (set less than long word) 108
6.8.4 sle.l (set less than or equal long word) 109
6.8.5 sgt.l (set greater than long word) 109
6.8.6 sge.l (set greater than or equal) 109
6.8.7 slt.ul (set less than unsigned long word) 109
6.8.8 sle.ul (set less than or equal unsigned long word) 109
6.8.9 sgt.ul (set greater than unsigned long word) 109
6.8.10 sge.ul (set greater than or equal unsigned long word) 110
6.8.11 seqi.l (set equal immediate long word) 110

6.8.12 snei.l (set not equal immediate long word) 110
6.8.13 slti.l (set less than immediate long word) 110
6.8.14 slei.l (set less than or equal immediate long word) 110
6.8.15 sgti.l (set greater than immediate long word) 111
6.8.16 sgei.l (set greater than or equal immediate long word) 111
6.8.17 slti.ul (set less than immediate unsigned long word) 111
6.8.18 slei.ul (set less than or equal immediate unsigned long word) 111
6.8.19 sgti.ul (set greater than immediate unsigned long word) 112
6.8.20 sgei.ul (set greater than or equal immediate unsigned long word) 112
6.9 LOAD/STORE INSTRUCTIONS 112
6.9.1 l.b (load byte) 112
6.9.2 l.ub (load unsigned byte) 112
6.9.3 l.h (load half-word) 112
6.9.4 l.uh (load unsigned half-word) 113
6.9.5 l.w (load word) 113
6.9.6 l.uw (load unsigned word) 113
6.9.7 l.l (load long word) 113
6.9.8 xchg (exchange) 114
6.9.9 s.b (store byte) 114
6.9.10 s.h (store half-word) 114
6.9.11 s.w (store word) 114
6.9.12 s.l (store long word) 114
6.9.13 lir (Load IP-relative) 115
6.9.14 ldm (Load Multiple) 115
6.9.15 stm (Store Multiple) 116
6.10 FLOW CONTROL INSTRUCTIONS 116
6.10.1 j (Jump) 116
6.10.2 jal (Jump And Link) 116
6.10.3 jr (Jump Register) 117
6.10.4 jalr (Jump And Link Register) 117
6.10.5 beq.l (branch on equal long word) 117
6.10.6 bne.l (branch on not equal long word) 117
6.10.7 blt.l (branch on less than long word) 117
6.10.8 ble.l (branch on less than or equal long word) 117
6.10.9 bgt.l (branch on greater than long word) 118
6.10.10 bge.l (branch on greater than or equal long word) 118
6.10.11 blt.ul (branch on less than unsigned long word) 118
6.10.12 ble.ul (branch on less than or equal unsigned long word) 118
6.10.13 bgt.ul (branch on greater than unsigned long word) 118
6.10.14 bge.ul (branch on greater than or equal unsigned long word) 119
6.10.15 beqi.l (branch on equal immediate long word) 119
6.10.16 bnei.l (branch on not equal immediate long word) 119
6.10.17 blti.l (branch on less than immediate long word) 119
6.10.18 blei.l (branch on less than or equal immediate long word) 119
6.10.19 bgti.l (branch on greater than immediate long word) 120
6.10.20 bgei.l (branch on greater than or equal immediate long word) 120
6.10.21 blti.ul (branch on less than immediate unsigned long word) 120
6.10.22 blei.ul (branch on less than or equal immediate unsigned long word) 120
6.10.23 bgti.ul (branch on greater than immediate unsigned long word) 121
6.10.24 bgei.ul (branch on greater than or equal immediate unsigned long word) 121
6.11 TLB CONTROL INSTRUCTIONS 121
6.11.1 iitlb (invalidate Instruction TLB) [privileged] 121
6.11.2 idtlb (invalidate Data TLB) [privileged] 121
6.11.3 iitlbe (invalidate Instruction TLB entry) [privileged] 122
6.11.4 idtlbe (invalidate data TLB entry) [privileged] 122
6.11.5 iitlbc (invalidate Instruction TLB for current process) [privileged] 122
6.11.6 idtlbc (invalidate Data TLB for current process) [privileged] 123
6.11.7 iitlbec (invalidate Instruction TLB entry for current process) [privileged] 123
6.11.8 idtlbec (invalidate data TLB entry for current process) [privileged] 123

6.12 CACHE CONTROL INSTRUCTIONS 124
6.12.1 *imb (Instruction Memory Barrier) [privileged]* *124*
6.12.2 *dmb (Data Memory Barrier) [privileged]* *124*
6.12.3 *imbc (Instruction Memory Barrier for current process) [privileged]* *124*
6.12.4 *dmbc (Data Memory Barrier for current process) [privileged]* *125*
6.13 I/O INSTRUCTIONS 125
6.13.1 *in.b (input byte) [privileged]* *125*
6.13.2 *in.ub (input unsigned byte) [privileged]* *126*
6.13.3 *in.h (input half-word) [privileged]* *126*
6.13.4 *in.uh (input unsigned half-word) [privileged]* *126*
6.13.5 *in.w (input word) [privileged]* *126*
6.13.6 *in.uw (input unsigned word) [privileged]* *127*
6.13.7 *in.l (input long word) [privileged]* *127*
6.13.8 *out.b (output byte) [privileged]* *127*
6.13.9 *out.h (output half-word) [privileged]* *127*
6.13.10 *out.w (output word) [privileged]* *128*
6.13.11 *out.l (output long word) [privileged]* *128*
6.13.12 *tstp (Test Port status) [privileged]* *128*
6.13.13 *setp (Set Port status) [privileged]* *129*
6.14 MISCELLANEOUS INSTRUCTIONS 129
6.14.1 *halt (Halt processor) [privileged]* *129*
6.14.2 *iret (Return from Interrupt Handler) [privileged]* *129*
6.14.3 *getfl (Get Flags)* *130*
6.14.4 *setfl (Set Flags)* *130*
6.14.5 *rstfl (Reset Flags)* *130*
6.14.6 *svc (Supervisor Call)* *130*
6.14.7 *brk (break)* *130*
6.14.8 *cpuid (CPU Identification)* *131*
6.14.9 *sigp (Signal Processor) [privileged]* *132*
6.14.10 *nop (No Operation)* *132*

7 THE FLOATING POINT FEATURE 133

7.1 DATA MOVEMENT INSTRUCTIONS 133
7.1.1 *mov.d (move double precision floating point)* *133*
7.1.2 *li.d (load immediate double precision floating point)* *133*
7.2 ARITHMETIC INSTRUCTIONS 133
7.2.1 *add.f (add single precision floating point)* *133*
7.2.2 *add.d (add double precision floating point)* *134*
7.2.3 *sub.f (subtract single precision floating point)* *134*
7.2.4 *sub.d (subtract double precision floating point)* *135*
7.2.5 *mul.f (multiply single precision floating point)* *135*
7.2.6 *mul.d (multiply double precision floating point)* *136*
7.2.7 *div.f (divide single precision floating point)* *136*
7.2.8 *div.d (divide double precision floating point)* *137*
7.2.9 *abs.f (absolute value single precision floating point)* *137*
7.2.10 *abs.d (absolute value double precision floating point)* *138*
7.2.11 *neg.f (negate single precision floating point)* *138*
7.2.12 *neg.d (negate double precision floating point)* *138*
7.2.13 *sqrt.f (square root single precision floating point)* *139*
7.2.14 *sqrt.d (square root double precision floating point)* *139*
7.3 DATA TYPE CONVERSION INSTRUCTIONS 140
7.3.1 *cvt.fb (Convert single precision floating point to byte)* *140*
7.3.2 *cvt.fub (Convert single precision floating point to unsigned byte)* *140*
7.3.3 *cvt.fh (Convert single precision floating point to half-word)* *141*
7.3.4 *cvt.fuh (Convert single precision floating point to unsigned half-word)* *141*
7.3.5 *cvt.fw (Convert single precision floating point to word)* *141*
7.3.6 *cvt.fuw (Convert single precision floating point to unsigned word)* *142*
7.3.7 *cvt.fl (Convert single precision floating point to long word)* *142*
7.3.8 *cvt.ful (convert single precision floating point to unsigned long word)* *143*

7.3.9 cvt.db (Convert single precision floating point to byte) 143
7.3.10 cvt.dub (Convert double precision floating point to unsigned byte) 144
7.3.11 cvt.dh (Convert double precision floating point to half-word) 144
7.3.12 cvt.duh (Convert double precision floating point to unsigned half-word) 144
7.3.13 cvt.dw (Convert double precision floating point to word) 145
7.3.14 cvt.duw (Convert double precision floating point to unsigned word) 145
7.3.15 cvt.dl (Convert double precision floating point to long word) 146
7.3.16 cvt.dul (convert double precision floating point to unsigned long word) 146
7.3.17 cvt.bf (convert byte to single precision floating point) 147
7.3.18 cvt.ubf (convert unsigned byte to single precision floating point) 147
7.3.19 cvt.hf (convert half-word to single precision floating point) 147
7.3.20 cvt.uhf (convert unsigned half-word to single precision floating point) 147
7.3.21 cvt.wf (convert word to single precision floating point) 148
7.3.22 cvt.uwf (convert unsigned word to single precision floating point) 148
7.3.23 cvt.lf (convert long word to single precision floating point) 149
7.3.24 cvt.ulf (convert unsigned long word to single precision floating point) 149
7.3.25 cvt.bd (convert byte to double precision floating point) 149
7.3.26 cvt.ubd (convert unsigned byte to double precision floating point) 150
7.3.27 cvt.hd (convert half-word to double precision floating point) 150
7.3.28 cvt.uhd (convert unsigned half-word to double precision floating point) 150
7.3.29 cvt.wd (convert word to double precision floating point) 150
7.3.30 cvt.uwd (convert unsigned word to double precision floating point) 151
7.3.31 cvt.ld (convert long word to double precision floating point) 151
7.3.32 cvt.uld (convert unsigned long word to double precision floating point) 151
7.3.33 cvt.df (Convert double precision floating point to single precision floating point) 152
7.4 COMPARISON INSTRUCTIONS 152
7.4.1 seq.d (set equal double precision floating point) 152
7.4.2 sne.d (set not equal double precision floating point) 153
7.4.3 slt.d (set less than double precision floating point) 153
7.4.4 sle.d (set less than or equal double precision floating point) 153
7.4.5 sgt.d (set greater than double precision floating point) 154
7.4.6 sge.d (set greater than or equal double precision floating point) 154
7.5 LOAD/STORE INSTRUCTIONS 154
7.5.1 l.f (load single precision floating point) 154
7.5.2 l.d (load double precision floating point) 155
7.5.3 s.f (store single precision floating point) 155
7.5.4 s.d (store double precision floating point) 155
7.6 FLOW CONTROL INSTRUCTIONS 155
7.6.1 beq.d (branch on equal double precision floating point) 155
7.6.2 bne.d (branch on not equal double precision floating point) 156
7.6.3 blt.d (branch on less than double precision floating point) 156
7.6.4 ble.d (branch on less than or equal double precision floating point) 156
7.6.5 bgt.d (branch on greater than double precision floating point) 157
7.6.6 bge.d (branch on greater than or equal double precision floating point) 157

8 THE DEBUG FEATURE 158

8.1 DEBUG EVENTS 158
8.2 DEBUG REGISTERS 158
8.2.1 Debug event specification 158
8.2.1.1 K (Kernel mode) 159
8.2.1.2 U (User mode) 159
8.2.1.3 Type 159
8.2.2 Memory Access debug event 159
8.2.2.1 Address 159
8.2.2.2 Scale and Size 159
8.2.2.3 X (eXecute) 159
8.2.2.4 W (Write) 159
8.2.2.5 R (Read) 160
8.2.3 Disabling debug events 160
8.3 DATA MOVEMENT INSTRUCTIONS 160

8.3.1 mov.gr (move debug register to general purpose register) [privileged] 160
8.3.2 mov.rg (move general purpose register to debug register) [privileged] 160

9 THE PERFORMANCE MONITORING FEATURE 161

9.1 PROGRAM EVENTS 161
9.2 PERFORMANCE MONITORING REGISTERS 162
9.2.1 The $mflags register 163
9.3 DATA MOVEMENT INSTRUCTIONS 163
9.3.1 mov.mr (move performance monitoring register to general purpose register) [privileged] 163
9.3.2 mov.rm (move general purpose register to performance monitoring register) [privileged] 164

10 THE PROTECTED MEMORY FEATURE 165

10.1 PROTECTED MEMORY CONCEPTS 165
10.1.1 Region table 165
10.1.2 Region table pointer 165
10.1.3 Preparing region table for use 165
10.2 VIRTUAL ADDRESS TRANSLATION 166
10.3 ALIASING 167

11 THE VIRTUAL MEMORY FEATURE 168

11.1 VIRTUAL MEMORY CONCEPTS 168
11.1.1 Page table format descriptor 168
11.1.2 Page table header pointer 169
11.1.3 Page table structure 169
11.1.3.1 base 169
11.1.3.2 A (accessed) 170
11.1.3.3 D (dirty) 170
11.1.3.4 U (used) 170
11.1.3.5 L (loaded) 170
11.1.3.6 K (kernel) 170
11.1.3.7 R (read) 170
11.1.3.8 W (write) 170
11.1.3.9 X (execute) 170
11.2 VIRTUAL ADDRESS TRANSLATION 170
11.3 ACCESSING VIRTUAL MEMORY ACROSS PAGE BOUNDARIES 171

12 INTERRUPTS AND EXCEPTIONS 173

12.1 TIMER 173
12.2 IO 173
12.2.1 Port 173
12.2.2 I/O status code 173
12.3 SVC 174
12.4 PROGRAM 174
12.5 EXTERNAL 174
12.6 HARDWARE 174
12.7 INTERRUPT HANDLERS 174
12.8 INTERRUPT MASKING 175
12.9 SYNCHRONOUS AND ASYNCHRONOUS INTERRUPTS 175

13 I/O SUBSYSTEM 177

13.1 OVERVIEW 177
13.1.1 I/O ports 177
13.1.2 I/O controllers 177
13.1.3 DMA channels 178

13.2 I/O PORTS 179
13.2.1 *I/O port state* 179
13.2.2 *I/O port assignment* 179
13.3 DMA CHANNELS 180
13.3.1 *Programmer's model* 180
13.3.1.1 *Registers* 180
13.3.1.2 *Accessing registers* 180
13.3.1.3 *DMA channel state* 181
13.3.1.3.1 *P (Present)* 181
13.3.1.3.2 *B (Big-endian)* 181
13.3.1.3.3 *W (Working)* 181
13.3.2 *Channel programs* 181
13.3.2.1 *dma.stop (DMA stop)* 181
13.3.2.2 *dma.t.b (DMA transfer bytes)* 182
13.3.2.2.1 *00_2 – I/O port* 182
13.3.2.2.2 *01_2 – Memory* 182
13.3.2.2.3 *10_2 – Memory post-increment* 182
13.3.2.2.4 *11_2 – Memory pre-decrement* 182
13.3.2.2.5 *Operand type combinations* 182
13.3.2.3 *dma.t.h (DMA transfer half-words)* 183
13.3.2.4 *dma.t.w (DMA transfer words)* 183
13.3.2.5 *dma.t.l (DMA transfer long words)* 183
13.3.2.6 *dma.vt.b (DMA variable-length transfer bytes)* 184
13.3.2.7 *dma.vt.h (DMA variable-length transfer half-words)* 184
13.3.2.8 *dma.vt.w (DMA variable-length transfer words)* 184
13.3.2.9 *dma.vt.l (DMA variable-length transfer long words)* 184
13.3.2.10 *dma.l.b (DMA load byte)* 185
13.3.2.11 *dma.l.ub (DMA load unsigned byte)* 185
13.3.2.12 *dma.l.h (DMA load half-word)* 185
13.3.2.13 *dma.l.uh (DMA load unsigned half-word)* 185
13.3.2.14 *dma.l.w (DMA load word)* 186
13.3.2.15 *dma.l.uw (DMA load unsigned word)* 186
13.3.2.16 *dma.l.l (DMA load long word)* 186
13.3.2.17 *dma.s.b (DMA store byte)* 186
13.3.2.18 *dma.s.h (DMA store half-word)* 186
13.3.2.19 *dma.s.w (DMA store word)* 187
13.3.2.20 *dma.s.l (DMA store long word)* 187
13.3.2.21 *dma.li.l (DMA load immediate long word)* 187
13.3.2.22 *dma.add.l (DMA add long word)* 187
13.3.2.23 *dma.sub.l (DMA subtract long word)* 188
13.3.2.24 *dma.and.l (DMA bitwise and long word)* 188
13.3.2.25 *dma.or.l (DMA bitwise or long word)* 188
13.3.2.26 *dma.xor.l (DMA bitwise exclusive or long word)* 188
13.3.2.27 *dma.addi.l (DMA add immediate long word)* 188
13.3.2.28 *dma.subi.l (DMA subtract immediate long word)* 189
13.3.2.29 *dma.andi.l (DMA and immediate long word)* 189
13.3.2.30 *dma.ori.l (DMA or immediate long word)* 189
13.3.2.31 *dma.xori.l (DMA exclusive or immediate long word)* 189
13.3.2.32 *dma.neg.l (DMA negate long word)* 189
13.3.2.33 *dma.not.l (DMA not long word)* 190
13.3.2.34 *dma.tstp (DMA Test Port status)* 190
13.3.2.35 *dma.setp (DMA Set Port status)* 190
13.3.2.36 *dma.j (DMA Jump)* 191
13.3.2.37 *dma.beq.l (DMA branch on equal long word)* 191
13.3.2.38 *dma.bne.l (DMA branch on not equal long word)* 191
13.3.2.39 *dma.blt.l (DMA branch on less than)* 191
13.3.2.40 *dma.ble.l (DMA branch on less than or equal long word)* 191
13.3.2.41 *dma.bgt.l (DMA branch on greater than long word)* 192
13.3.2.42 *dma.bge.l (DMA branch on greater than or equal long word)* 192
13.3.2.43 *dma.blt.ul (DMA branch on less than unsigned long word)* 192
13.3.2.44 *dma.ble.ul (DMA branch on less than or equal unsigned long word)* 192
13.3.2.45 *dma.bgt.ul (DMA branch on greater than unsigned long word)* 192
13.3.2.46 *dma.bge.ul (DMA branch on greater than or equal unsigned long word)* 193
13.3.2.47 *dma.shl.l (DMA shift left long word)* 193
13.3.2.48 *dma.shr.l (DMA shift right long word)* 193
13.3.2.49 *dma.asl.l (DMA arithmetic shift left long word)* 193
13.3.2.50 *dma.asr.l (DMA arithmetic shift right long word)* 194

13.3.2.51 *dma.rol.l (DMA rotate left long word)* 194
13.3.2.52 *dma.ror.l (DMA rotate right long word)* 194
13.3.2.53 *dma.shli.l (DMA shift left long word immediate)* 194
13.3.2.54 *dma.shri.l (DMA shift right long word immediate)* 195
13.3.2.55 *dma.asli.l (DMA arithmetic shift left long word immediate)* 195
13.3.2.56 *dma.asri.l (DMA arithmetic shift right long word immediate)* 195
13.3.2.57 *dma.roli.l (DMA rotate left long word immediate)* 195
13.3.2.58 *dma.rori.l (DMA rotate right long word immediate)* 195
13.3.3 *DMA interrupts* 196

14 THE ON-CHIP MEMORY 197

14.1 OCM AND VIRTUAL MODE 197
14.2 OCM CONTROLLER 197
14.3 OCM CONCEPTS 199
14.3.1 *OCM bank* 199
14.3.2 *OCM regions* 199
14.3.3 *OCM and memory protection* 199
14.3.4 *OCM and data alignment* 200
14.4 OCM OPERATION 200
14.4.1 *Region control registers* 200
14.4.2 *Address overflows and wraparounds* 201

15 BOOTSTRAPPING 202

16 APPENDIX A: GNU FREE DOCUMENTATION LICENSE 203

17 APPENDIX B: EXCEPTION CODES 210

18 APPENDIX C: EXTERNAL INTERRUPT TYPES 213

19 APPENDIX D: HARDWARE ERROR CODES 214

20 APPENDIX E: DMA INTERRUPT STATUS CODES 215

2 Overview

This document is a definitive guide into the version 1 of the Cereon computer architecture.

2.1 Main features

The Cereon architecture addresses the needs of the medium-to-large scale workstations and servers. Hence, the main features of the Cereon architecture include:

- A fully 64-bit architecture, which provides applications with an access to virtually unlimited memory. In the past, there have been some negative experience in getting 64-bit architectures to handle 8-, 16- and 32-bit data efficiently; however, Cereon has been specifically designed to overcome these problems.
- The Cereon instruction set is simple and orthogonal, which allows Cereon processors to be implemented with a very small number of electronic components as compared to other 64-bit architectures currently on the market.
- A simple orthogonal RISC instruction set is an ideal target for code generation by compilers.
- As Cereon processors are cheap, it is expected that multi-core and multiprocessor configurations will be commonplace. The Cereon architecture further assists in creating multiprocessor configurations by providing semi-independent I/O and memory access facilities, which reduces the number of conflicts between tasks running on different processors.

The unique feature of the Cereon architecture is in that, unlike other architectures, where data processing instructions each perform a single operation, Cereon data processing instructions allow specifying optional pre- and post-processing data conversions. While not having any effect on execution speed (as convert-operation-convert datapath is still significantly shorter than critical data paths involving memory access), this allows programs operating on 8-, 16- and 32-bit data items to be as efficient as those operating on 64-bit items – something which is very rare amongst 64-bit architectures.

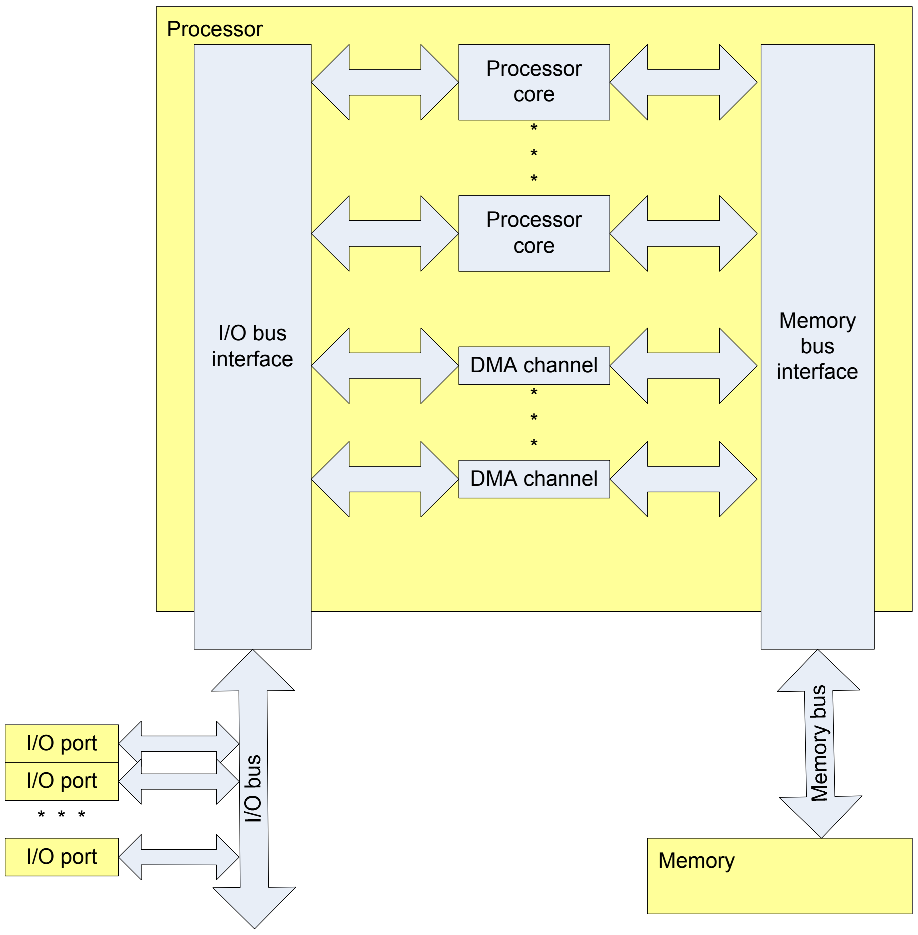
3 Introduction to Cereon

Cereon architecture follows a Reduced Instruction Set Computer (RISC) paradigm, incorporating the following common RISC features:

- A large number of registers that can be uniformly accessed.
- A load/store architecture.
- Simple and uniform memory addressing modes.
- Uniform instruction lengths and formats.

3.1 Architecture

An overall architecture of a Cereon processor and its immediate environment is illustrated on the following diagram:



Individual elements of the Cereon architecture are:

- Processor cores – units that actually fetches, decodes and executes instructions. Historically a processor with only one core is by far the most common; however, recent advances in processor design led to the proliferation of two- (or even more) core processors.
- Memory bus interface – is used by processor cores (and other processor components, such as DMA channels) to transfer data from and to the main memory. The memory bus interface connects to a high-speed memory bus. In

a system with more than one processor, each processor will have an independent connection to the same memory bus.
- I/O bus interface – is used by the processor to talk to the outside world in a uniform and efficient manner. The I/O bus interface connects, through a dedicated I/O bus, to a number of I/O ports.
- I/O ports – are connection points where external device controllers are attached.
- DMA channel – provides means of transferring data between external devices and main memory without using the resources of a processor core. In effect, a DMA channel is a miniature processor core which, unlike general purpose Cereon processor core, is geared towards executing data transfer instructions.

Other Cereon architectural elements not shown on the diagram above are:

- Caches and write buffers – are used to retain frequently used data within the processor instead of accessing it from main memory. Instructions and data can be cached separately or in the same cache.
- Translation-lookaside buffers – are used to speed up virtual address translation.
- Branch prediction tables – are used to make instruction prefetching efficient in the presence of conditional branch instructions.

Depending on the processor model, each processor core can have its own private copy of the above data structures, or a single copy can be shared between several processor cores. For example, a configuration is possible where each processor core in a 4-core processor has its own TLB, but all 4 cores share the same branch prediction table and cache.

3.2 Features

The Cereon architecture is modular. It consists of:

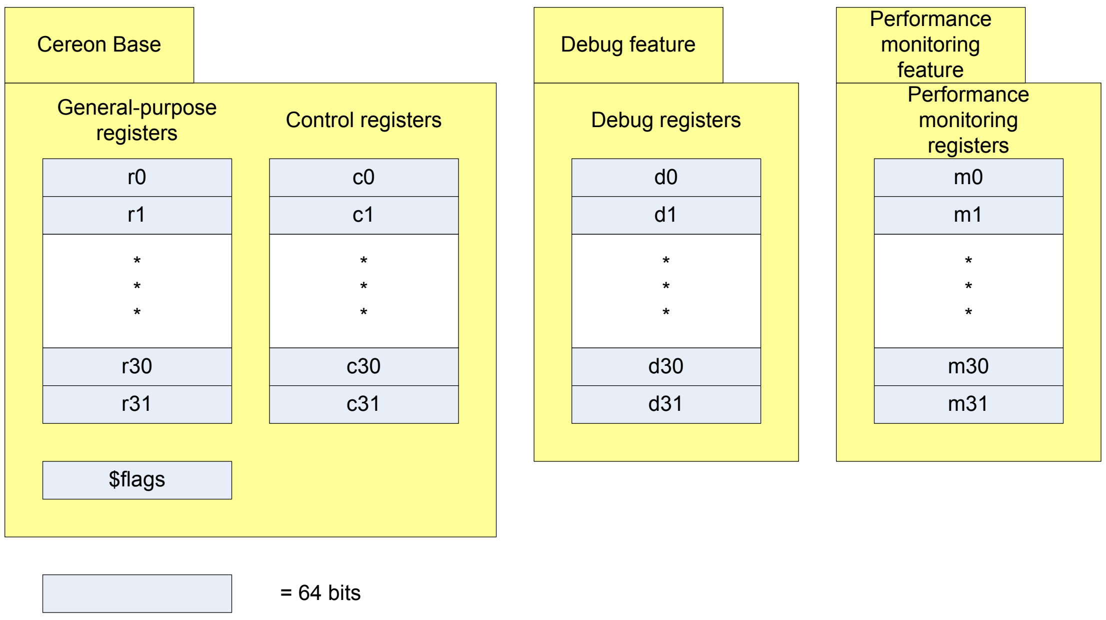
- The Cereon Base functionality, which is present in all Cereon processors. This functionality includes support for general purpose instructions and registers, control registers, etc.
- A number of optional features, which can be present or absent in a specific processor in a more-or-less independent manner. Currently these features are:
  - Floating point feature – permits hardware support for floating point calculations.
  - Debug feature – provides hardware support for advanced debugging.
  - Unaligned operand feature – allows operands of load and store instructions not to be naturally aligned.
  - Virtual memory feature – provides hardware support for page-based virtual memory with adjustable page table structure and page size.
  - Protected memory feature – provides a lightweight alternative to a virtual memory.
  - Performance monitoring feature – provides hardware support for performance monitoring and program event counting.

3.3 Registers

The Cereon processor has the following internal registers:

- 32 64-bit general purpose registers `r0..r31`. Of these 32 registers two registers (instruction pointer and return address register) have a special hardware-enforced purpose, while the remaining 30 registers are free for programmer to access as he sees fit, to store either integer or floating point data.
- 32 64-bit control registers, `c0..c31`. These registers are used to maintain the current state of a processor and manage interrupts.
- A 64-bit hidden `$flags` register, used to track unusual situations that can occur during program execution.
- (If debug feature is present) 32 64-bit debug registers, `d0..d31`. These registers are used to track debug events that may occur during program execution.
- (If performance monitoring feature is present) 32 64-bit performance monitoring registers, `m0..m31`. These registers are used to count various program events.

The following diagram illustrates the Cereon registers available to a programmer:



3.4 Interrupts

Cereon supports 6 types of interrupts:

- Timer interrupts, that are generated internally by processor's high-performance timer.
- I/O interrupts, that are generated by external I/O hardware.
- Supervisor Call (SVC) interrupts, that provide the standard means for the user-mode code to call a system service.

- Program interrupts, which occur when a program attempts to perform an illegal action.
- External interrupts, which originate from sources external to a particular processor. Examples of external interrupts include system resets and interprocessor signals.
- Hardware interrupts, which occur when a hardware error is detected.

When an interrupt occurs, the processor that services that interrupt stops whatever its doing, records its current state in dedicated control registers and switches to the interrupt handler defined with more dedicated control registers. The process of handling an interrupt does not incur any memory accesses and, therefore, allows for very fast interrupt response times.

For more details on Cereon interrupt handling, refer to the “Interrupts and exceptions” chapter of this document.

3.5 Processor state

A dedicated set of control registers is used to record the current processor state, which, among other things, includes information about:

- Whether the processor is currently operating or is idle and just sits there waiting for an interrupt.
- Whether the processor operates in Kernel or User mode.
- Whether the processor accesses memory in big-endian or little-endian mode.
- Which IEEE-754 rounding and error handling modes are currently in effect.
- Whether trap handling is currently in effect or not. Traps allow direct support for debugging the machine code at a single instruction level.
- Whether paged or segmented virtual memory is in effect and, if it is, how it is implemented.
- Which of the 6 interrupt types are currently allowed to occur.
- The ID of the currently executing process.

3.6 Processor identification

Each processor core in a Cereon machine is assigned a unique 16-bit number. The upper 8 bits of this number refer to a particular processor, the lower 8 bits identify a specific processor core within that processor. As a consequence, all cores in a multi-core processor have the same 8 upper bits in their IDs. Among other things, this allows OS kernels to determine if they run on a “true” multiprocessor machine or on a multi-core/multi-threaded machine and adjust their behaviour accordingly (for example, on a multi-threaded machine the OS task scheduler may assign several threads of the same application to execute on different cores within the same processor, on the assumption that since these threads will often access the same data, using the same cache for these threads may increase performance).

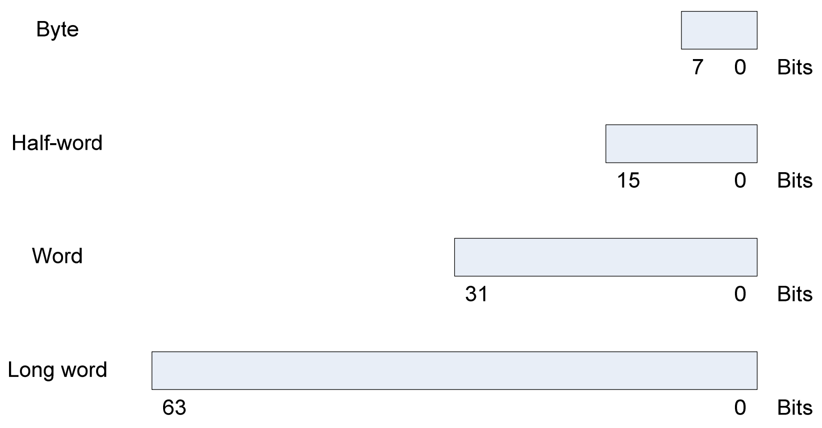
4 Programmer's model

4.1 Data types

Cereon processor supports the following data types:

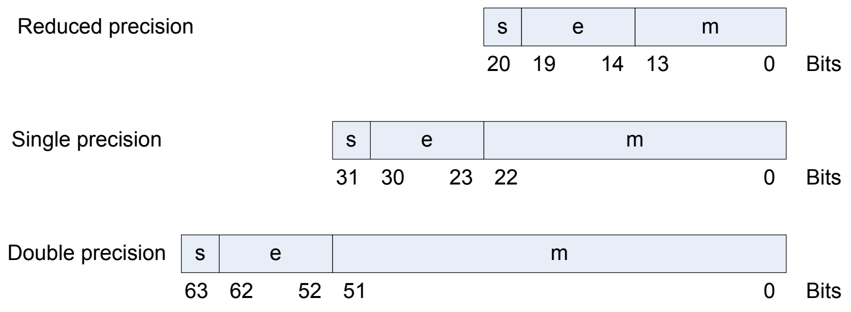
- Byte – 8-bit integer.
- Half-word – 16-bit integer.
- Word – 32-bit integer.
- Long word – 64-bit integer.
- Reduced-precision real – 21-bit floating point value.
- Single precision real – 32-bit IEEE-754 floating point value.
- Double precision real – 64-bit IEEE-754 floating point value.

4.1.1 Integer types



All integer values can be interpreted as either unsigned integer value or signed integer values that use 2's complement code for negative values. The exact interpretation used in each specific case depends upon the instruction that works with these values.

4.1.2 Real types



In all three formats, the following rules are used to interpret the value of a real number (in the formulas below `d` is 31 for a reduced-precision value, 127 for a single precision value and 1023 for a double precision value):

- If `e` = 0 and `m` = 0, then the value is `+0.0` regardless of the `s` bit.
- If `e` = 0 and `m` ≠ 0, then the value is $s \times 0.m \times 2^{-d}$. This is a positive (`s` = 0) or negative (`s` = 1) denormalized value
- If `e` ≠ 0 and `e` ≠ 2×`d`+1, then the value is $s \times 1.m \times 2^{e-d}$. This is a positive (`s` = 0) or negative (`s` = 1) normalized value.
- If `e` = 2×`d`+1 and `m` = 0, then the value is a positive (`s` = 0) or negative (`s` = 1) infinity.
- If `e` = 2×`d`+1, `m` ≠ 0 and the leftmost bit of `m` is 0, then the value is a signalling NaN (Not-a-Number).
- If `e` = 2×`d`+1, `m` ≠ 0 and the leftmost bit of `m` is 1, then the value is a quiet NaN (Not-a-Number).

Note that single- and double precision real numbers are represented in a IEEE-754 format. The reduced-precision format allows placing small real constants (such as 0, 1, 0.5 and -1.25) as immediate operands into instructions instead of having to load them from memory.

4.2 Processor modes

The control registers recording the current processor state store several more or less independent characteristics, the combination of which defines the current processor mode.

4.2.1 Kernel vs. User

In a Kernel mode, the entire set of Cereon facilities is available to the processor. In User mode, some of these facilities are disabled and a `PROGRAM` interrupt occurs if they are used. Examples of such facilities include accessing processor's control registers, performing I/O operations, etc. – in other words, anything that shall remain under control of an operating system.

The following table summarizes differences between Kernel and User mode.

Feature	Processor in Kernel mode	Processor in User mode
Direct access to processor control registers	Enabled	Disabled (causes an exception)
Access to Kernel virtual memory pages (in virtual mode only)	Subject to access control	Disabled (causes an exception)
Access to User virtual memory pages (in virtual mode only)	Enabled	Subject to access control
Use of privileged processor instructions	Enabled	Disabled (causes an exception)

4.2.2 Real vs. Virtual

In Real mode, all memory addresses generated by processor for both instruction and data access are physical memory addresses. Memory protection is not performed; i.e. instructions can be executed from any memory location and have free read/write access to entire available physical memory (note, however, that specific Cereon models may impose hardcoded access restrictions on memory areas in Real mode as well, such as requiring the upper half of the real address space to be read-only, etc.) An attempt to access the memory address at which no memory exists (or an attempt to write to read-only memory) will still cause an exception.

In Virtual mode, all memory addresses are virtual memory addresses; these are translated into physical memory addresses using the virtual address translation mechanism described in the corresponding section of this manual (using protected memory or virtual memory mechanisms). In addition, all memory accesses are subject to access control, which depends on current processor state and whether a memory area being accessed is designated as Kernel or User memory area:

Processor mode	Virtual memory area marked as	Memory access validity
Kernel	Kernel	Access is allowed based on access flags associated with virtual memory page.
Kernel	User	Any type of memory access is allowed.
User	Kernel	Memory access is not allowed.
User	User	Access is allowed based on access flags associated with virtual memory page.

4.2.3 Big-endian vs. Little-endian

Depending on whether the processor is in big- or little-endian mode, values that occupy more than one byte in memory are expected to be stored with the most significant byte first (big-endian mode) or the least significant byte first (little-endian mode). Note that, although a processor can always determine whether it is currently in big- or little-endian mode, it may or may not be able to switch to the opposite mode depending on the processor model (the switch is only possible while in Kernel mode).

4.2.4 Working vs. Idle

When in Working mode the processor is executing instructions normally. When in Idle mode the processor is not executing any instructions. In the latter case, the processor can be re-activated and brought back to a Working mode by an incoming interrupt.

4.3 Registers

The Cereon processor has the following internal registers:

- 32 64-bit general purpose registers `r0..r31`.
- 32 64-bit control registers, `c0..c31`.
- A 64-bit hidden `$flags` register.

- 32 64-bit debug registers, `d0..d31` (if Debug feature is available).
- 32 64-bit performance monitoring registers, `m0..m31` (if Performance monitoring feature is available).

4.3.1 General purpose registers

General purpose registers are numbered from `r0` to `r31`. Each general purpose register can contain a 64-bit value, which is interpreted as signed integer, unsigned integer or double precision real value depending on instructions that use the register.

In assembly-language programs, general purpose registers can be specified by their numbers `r0..r31`. In addition, these registers are assigned symbolic names, which reflect conventions for their use. These symbolic names, apart from designating a specific register, reflect on the interpretation of its contents, so `$rv` refers to `r0` containing an integer value, while `$frv` refers to `r0` containing a real value. This provides an additional measure of safety for an assembler-language programmer.

The table below summarizes assignment of symbolic names to general purpose registers and their usage.

Register	Symbolic name	Usage convention
`r0`	`$rv, $frv`	Procedure return value
`r1`	`$a0, $fa0`	Parameter register
`r2`	`$a1, $fa1`	Parameter register
`r3`	`$a2, $fa2`	Parameter register
`r4`	`$a3, $fa3`	Parameter register
`r5`	`$t0, $ft0`	Temporary register
`r6`	`$t1, $ft1`	Temporary register
`r7`	`$t2, $ft2`	Temporary register
`r8`	`$t3, $ft3`	Temporary register
`r9`	`$t4, $ft4`	Temporary register
`r10`	`$t5, $ft5`	Temporary register
`r11`	`$t6, $ft6`	Temporary register
`r12`	`$t7, $ft7`	Temporary register
`r13`	`$s0, $fs0`	Saved temporary register
`r14`	`$s1, $fs1`	Saved temporary register
`r15`	`$s2, $fs2`	Saved temporary register
`r16`	`$s3, $fs3`	Saved temporary register
`r17`	`$s4, $fs4`	Saved temporary register
`r18`	`$s5, $fs5`	Saved temporary register
`r19`	`$s6, $fs6`	Saved temporary register
`r20`	`$s7, $fs7`	Saved temporary register
`r21`	`$s8, $fs8`	Saved temporary register
`r22`	`$s9, $fs9`	Saved temporary register
`r23`	`$s10, $fs10`	Saved temporary register
`r24`	`$s11, $fs11`	Saved temporary register
`r25`	`$s12, $fs12`	Saved temporary register
`r26`	`$gp`	Global data pointer

`r27`	`$sp`	Stack pointer
`r28`	`$fp`	Frame pointer
`r29`	`$dp`	Display pointer
`r30`	`$ra`	Procedure return address
`r31`	`$ip`	Instruction pointer

All general purpose registers are free for the programmer to use as he sees fit, except `$ra` and `$ip`, which have a special purpose. However, there exists a convention as to what different general purpose registers are used for; this convention is reflected in the symbolic name of these general purpose registers. In particular, this convention is the reason why `$gp`, `$sp`, `$fp`, `$dp`, `$ra` and `$ip` register names do not name a floating point counterpart – as values in these registers shall, by convention, always be pointers instead of numeric.

Note, that use of general purpose registers for floating point calculations is only allowed if the processor has a floating point feature.

4.3.1.1 $rv/$frv

By convention, when a procedure must return an integer, real or pointer result, it is returned in `$rv` (integer/pointer) or `$frv` (real), extended to 64 bits.

4.3.1.2 $a0..$a3/$fa0..$fa3

By convention, the first four integer, real or pointer arguments for a procedure call are placed into these registers by the caller, extended to 64 bits.

4.3.1.3 $t0..$t7/$ft0..$ft7

By convention, these registers are used as temporary registers for storing intermediate results. When a procedure call is performed, these registers are not expected to retain their values upon return, so the caller must take care of saving these registers before the call and restoring them after the call if necessary.

4.3.1.4 $s0..$s12/$fs0..$fs12

By convention, these registers are used as permanent registers for keeping values with a long lifetime. When a procedure call is performed, these registers are expected to retain their values upon return, so the callee must take care of saving these registers when called and restoring them before return if it wishes to use them.

4.3.1.5 $gp

By convention, this register contains a pointer to a global static data area of the currently executing program. If the specific toolchain does not use this register for that purpose, `$gp` can be used as another temporary register; however, such use is not generally recommended as it would make the code incompatible with other toolchains that do use `$gp` as a global static data pointer.

4.3.1.6 $sp

By convention, this register contains a pointer to the lowest memory address occupied by the stack. By the same convention, stacks shall start at higher addresses and grow down.

4.3.1.7 $fp
By convention, this register contains a pointer to a fixed location within the topmost procedure activation frame in the stack.

4.3.1.8 $dp
By convention, this register contains a display pointer to a display data containing the static chain for the currently executing procedure. If the language in question does not support nonlocal nonstatic data (C and C++ are such languages), or static chains are implemented using a technique other that displays, `$dp` can be used as another temporary register; however, such use is not generally recommended as it would make the code incompatible with other code that does use `$dp` as a display pointer.

4.3.1.9 $ra
When a jump-and-link (`jal` or `jalr`) instruction is executed to call a subroutine, the address of the instruction immediately following the `jal`/`jalr` instruction is stored into the `$ra` register. When the subroutine has finished, it must use the contents of `$ra` to return to the caller by performing an indirect register jump there.

4.3.1.10 $ip
The `$ip` register is an instruction pointer, used by the processor to fetch instructions for execution. At any time, it contains the address of the next instruction to be executed. Any explicit change to the contents of `$ip` (such as loading it from memory, or adding a constant to it) causes a jump.

An instruction reading from `$ip` always reads the address of the immediately following instruction, i.e. the address of the current instruction + 4. An instruction writing to `$ip` performs an unconditional jump to the specified address.

4.3.2 Control registers
Control registers are numbered from `c0` to `31`. Each control register can contain a 64-bit value, whose interpretation is register-specific.

In assembly-language programs, control registers can be specified by their numbers. In addition, these registers are assigned symbolic names, which reflect their usage. Note, that control registers have specific uses enforced by hardware. The table below summarizes assignment of symbolic names to control registers and their usage.

Register	Symbolic name	Usage convention
`c0`	`$state`	Current processor state.
`c1`	`$pth`	Page table header pointer
`c2`	`$itc`	Interval timer counter.
`c3`	`$cc`	Cycle counter
`c4`	`$isaveip.tm`	The saved `$ip` for `TIMER` interrupt.
`c5`	`$isavestate.tm`	The saved state for `TIMER` interrupt.
`c6`	`$ihstate.tm`	The new state for `TIMER` interrupt handler.
`c7`	`$iha.tm`	The address of the `TIMER` interrupt handler.
`c8`	`$isaveip.io`	The saved `$ip` for `IO` interrupt.

`c9`	`$isavestate.io`	The saved state for `IO` interrupt.
`c10`	`$ihstate.io`	The new state for `IO` interrupt handler.
`c11`	`$iha.io`	The address of the `IO` interrupt handler.
`c12`	`$isc.io`	The Interrupt Status Code associated with the `IO` interrupt.
`c13`	`$isaveip.svc`	The saved `$ip` for `SVC` interrupt.
`c14`	`$isavestate.svc`	The saved state for `SVC` interrupt.
`c15`	`$ihstate.svc`	The new state for `SVC` interrupt handler.
`c16`	`$iha.svc`	The address of the `SVC` interrupt handler.
`c17`	`$isaveip.prg`	The saved `$ip` for `PROGRAM` interrupt.
`c18`	`$isavestate.prg`	The saved state for `PROGRAM` interrupt.
`c19`	`$ihstate.prg`	The new state for `PROGRAM` interrupt handler.
`c20`	`$iha.prg`	The address of the `PROGRAM` interrupt handler.
`c21`	`$isc.prg`	The Interrupt Status Code associated with the `PROGRAM` interrupt.
`c22`	`$isaveip.ext`	The saved `$ip` for `EXTERNAL` interrupt.
`c23`	`$isavestate.ext`	The saved state for `EXTERNAL` interrupt.
`c24`	`$ihstate.ext`	The new state for `EXTERNAL` interrupt handler.
`c25`	`$iha.ext`	The address of the `EXTERNAL` interrupt handler.
`c26`	`$isc.ext`	The Interrupt Status Code associated with the `EXTERNAL` interrupt.
`c27`	`$isaveip.hw`	The saved `$ip` for `HARDWARE` interrupt.
`c28`	`$isavestate.hw`	The saved state for `HARDWARE` interrupt.
`c29`	`$ihstate.hw`	The new state for `HARDWARE` interrupt handler.
`c30`	`$iha.hw`	The address of the `HARDWARE` interrupt handler.
`c31`	`$isc.hw`	The Interrupt Status Code associated with the `HARDWARE` interrupt.

Individual control registers are described below.

4.3.2.1 $state

This register describes the current state of the processor. It has the following format:

Bits	63 32
	CID

Bits	31 26	25 16
	IM	-

Bits	14 15	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	RM	O	N	I	U	E	Z	R	D	W	B	P	T	V	K

The meaning of individual fields within the `$state` register is explained below.

4.3.2.1.1 K (Kernel mode)

When this bit is 1, the processor is running in Kernel mode; otherwise, the processor is running in the User mode.

4.3.2.1.2 V (Virtual mode)

When this bit is 1, the processor is running in Virtual mode; otherwise, the processor is running in the Real mode. If the processor supports either protected or virtal memory feature (the two are mutually exclusive), the corresponding address translation mechanism is used in Virtual mode. If the processor does not have either of these features, it cannot operate in a Virtual mode; an attempt to switch such a processor into a Virtual mode causes an `OPERAND` exception.

4.3.2.1.3 T (Trap mode) and P (Pending trap)

When the T bit is 1 the processor traps are enabled, otherwise traps are disabled. When traps are enabled, the `TRAP` interrupt occurs before the execution of each instruction unless `PROGRAM` interrupts are disabled.

The P flag is checked before execution of each instruction. If it is 0, no extra actions are taken. If it is 1, it is set to 0 and T is set to 1.

If both T and PT flags are 1 before the instruction, the PT flag is set to 0 and the trap occurs normally.

When returning from the `TRAP` interrupt handler, the interrupt handler sets the T flag to 0, and the P flag to 1. This behaviour ensures that, once a `TRAP` interrupt is handled, the processor will be able to execute one instruction before the next `TRAP` interrupt occurs.

4.3.2.1.4 B (Big-endian)

When this bit is 1, the processor uses the big-endian byte ordering when transferring data to and from memory; otherwise the little-endian byte ordering is used.

4.3.2.1.5 W (Working)

When this bit is 1, the processor is in Working mode; otherwise it is in Idle mode.

4.3.2.1.6 D (Debug events)

When this bit is 1, debug events are enabled and can cause `PROGRAM` exceptions. When it is 0, debug exceptions do not occur. If the processor has no debug feature, this bit is ignored.

Note that if a debug exception is triggered and a `PROGRAM` interrupt is disabled, the processor reacts in the same way as if any other (i.e. non-debug) exception has occurred while the `PROGRAM` interrupt is masked, namely halt the processor. Therefore, debug events shall only be enabled whenever `PROGRAM` interrupt is enabled.

4.3.2.1.7 R (real opeRand)

When this bit is 1, `FOPERAND` exceptions are enabled. When it is 0, `FOPERAND` exceptions do not occur; signalling NaN results are produced instead. If the processor has no floating point feature, this bit is ignored.

4.3.2.1.8 Z (real division by Zero)

When this bit is 1, `FZDIV` exceptions are enabled. When it is 0, `FZDIV` exceptions do not occur; infinite results are produced instead. If the processor has no floating point feature, this bit is ignored.

4.3.2.1.9 E (real ovErflow)

When this bit is 1, `FOVERFLOW` exceptions are enabled. When it is 0, `FOVERFLOW` exceptions do not occur; infinite results are produced instead. If the processor has no floating point feature, this bit is ignored.

4.3.2.1.10 U (real Underflow)

When this bit is 1, `FUNDERFLOW` exceptions are enabled. When it is 0, `FUNDERFLOW` exceptions do not occur; zero result is produced instead. If the processor has no floating point feature, this bit is ignored.

4.3.2.1.11 I (real Inexact)

When this bit is 1, `FINEXACT` exceptions are enabled. When it is 0, `FINEXACT` exceptions do not occur; result is rounded instead using the current rounding mode. If the processor has no floating point feature, this bit is ignored.

4.3.2.1.12 N (iNteger division by zero)

When this bit is 1, `ZDIV` exceptions are enabled. When it is 0, `ZDIV` exceptions do not occur; an attempt to perform an integer division by zero results in both quotient and remainder of 0.

4.3.2.1.13 O (integer Overflow)

When this bit is 1, `IOVERFLOW` exceptions are enabled. When it is 0, `IOVERFLOW` exceptions do not occur; wraparound happens instead.

4.3.2.1.14 RM (Rounding Mode)

The contents of these 2 bits specified the floating point rounding mode currently in effect:

- `00` – rounding is performed towards the nearest representable value, with "even" values preferred whenever there are two nearest representable values.
- `01` – rounding is performed towards the negative infinity (down).
- `10` – rounding is performed towards the positive infinity (up).
- `11` – rounding is performed towards zero (chop).

If the processor has no floating point feature, these bits are ignored.

4.3.2.1.15 IM (Interrupt Mask)

This field contains 6 1-bit flags specifying which interrupts are allowed to occur. Bit `(N-26)` corresponds to interrupt N. For more details on Cereon interrupt handling, refer to the "Interrupts and exceptions" chapter of this document.

4.3.2.1.16 CID (Context ID)

This field contains the current 32-bit context ID. This value is used for virtual address translation.

4.3.2.2 $pth

This register contains the pointer to a page table header data structure. It is only used when processor is in virtual mode. For more details on Cereon virtual memory facilities, refer to the "Virtual memory feature" and "Protected memory feature" chapter of this document.

4.3.2.3 $itc

This register contains the 64-bit interval timer counter. Its value is examined during each CPU cycle, using the following algorithm:

- If `$itc = 0`, no further actions are taken.
- If `$itc > 1`, the value of `$itc` is decremented by 1.
- If `$itc = 1` and the `TIMER` interrupt is enabled and can occur, the value of `$itc` is set to 0 and a `TIMER` interrupt occurs.
- If `$itc = 1` and the `TIMER` interrupt is disabled, or cannot occur immediately (because current instruction is in the middle of execution) the `$itc` remains unchanged (i.e. 1), this ensures that the `TIMER` interrupt will occur when the processor is ready for it.

4.3.2.4 $cc

This register contains the 64-bit cycle counter. Its value is incremented by 1 each CPU cycle. Note, that an instruction taking more than one cycle to execute (because, for example, it is stalled to wait for a data to arrive from memory) will observe the `$cc` incremented by more than 1 after its execution is complete. This allows, among other things:

- Precise measurement of execution times for individual instructions.
- Detection of how many cycles a pending `TIMER` interrupt was delayed for.

Note that when processor is in Idle mode the cycle counter is not incremented with each cycle.

4.3.3 Debug registers

Debug registers are numbered from `d0` to `d31`. Each debug register can contain a 64-bit value. Debug registers are described in detail in the “Debug feature” chapter of this document.

4.3.4 Performance monitoring registers

Performance monitoring registers are numbered from `m0` to `m31`. Each performance monitoring register can contain a 64-bit value. Performance monitoring registers are described in detail in the “Performance monitoring feature” chapter of this document.

4.3.5 $flags register

This register acts as a sticky accumulator of flags describing various unusual situations that can occur during instruction execution. Its main purpose is to provide user code with an indication of whether a certain condition (such as overflow) occurred during the execution of a sequence of instructions. Typically, an application will reset `$flags`, execute a sequence of operations (usually arithmetics) and then check the corresponding bits of the `$flags` register to, for example, find out if an integer overflow (or a floating-point underflow) had occurred during the sequence. This provides a lightweight alternative to unusual arithmetic situations handled via exceptions.

The `$flags` register has the following format:

Bits	63 7	6	5	4	3	2	1	0
	-	O	N	I	U	E	Z	R

The meaning of individual fields within the `$flags` register is explained below.

4.3.5.1.1 R (real opeRand)

Whenever an attempt is made to perform a floating-point operation with one or both operands being incorrect (such as NaN, or negative operand for `sqrt`), this bit is set to 1, regardless of whether a `FZDIV` exception occurs or not. If the processor has no floating point feature, this bit is never set up to 1 by hardware.

4.3.5.1.2 Z (real division by Zero)

Whenever an attempt is made to perform a floating-point division by 0, this bit is set to 1, regardless of whether a `FZDIV` exception occurs or not. If the processor has no floating point feature, this bit is never set up to 1 by hardware.

4.3.5.1.3 E (real ovErflow)

Whenever an overflow occurs during a floating-point operation, this bit is set to 1, regardless of whether a `FOVERFLOW` exception occurs or not. If the processor has no floating point feature, this bit is never set up to 1 by hardware.

4.3.5.1.4 U (real Underflow)

Whenever an underflow occurs during a floating-point operation, this bit is set to 1, regardless of whether a `FUNDERFLOW` exception occurs or not. If the processor has no floating point feature, this bit is never set up to 1 by hardware.

4.3.5.1.5 I (real Inexact)
Whenever a floating-point operation produces an inexact result, this bit is set to 1, regardless of whether a `FINEXACT` exception occurs or not. If the processor has no floating point feature, this bit is never set up to 1 by hardware.

4.3.5.1.6 N (iNteger division by zero)
Whenever an attempt is made to perform an integer division by 0, this bit is set to 1, regardless of whether a `ZDIV` exception occurs or not.

4.3.5.1.7 O (integer Overflow)
Whenever an overflow occurs during an integer operation, this bit is set to 1, regardless of whether a `IOVERFLOW` exception occurs or not.

4.4 Memory
The Cereon architecture uses a single flat address space of 2^{64} 8-bit bytes. Memory addresses are treated as unsigned integer numbers running from 0 to 2^{64}-1.

All address calculations are performed using unsigned 64-bit integer arithmetic. That, in particular, means that:

- An overflow encountered during address calculation can cause an `IOVERFLOW` exception (if one is enabled) and always sets the `N` flag of `$state` to 1.
- Addresses wrap around if an integer overflow occurs and an `IOVERFLOW` exception is disabled.

4.4.1 Alignment
Unless an unaligned operand feature is available, all memory accesses must be naturally aligned. Specifically:

- A byte can be loaded from or stored to any memory address.
- A short word can be loaded from or stored to any memory address that is a multiple of 2.
- A word can be loaded from or stored to any memory address that is a multiple of 4. The same applies to single precision real values and instructions, since these are also 32 bits long.
- A long word can be loaded from or stored to any memory address that is a multiple of 8. The same applies to double precision real values, since these are also 64 bits long.

Any attempt to load or store a value that is not naturally aligned causes a `DALIGN` exception, unless the unaligned operand feature is available.

If the unaligned operand feature is available, an item of data can be loaded from or stored to at any address (instructions must be aligned at a 4-byte boundary regardless of whether the unaligned operand feature is available or not). In this case the processor checks if the operand is properly aligned before performing a load or store. If it is, then the load/store operation proceeds normally. If, on the other hand, the address is not naturally aligned, the processor performs a series of byte, half-word and/or word loads/stores, required to simulate the unaligned load/store. The exact rules of how an unaligned load/store is broken into a sequence of aligned loads/stores is processor-specific (for example, when asked to store a word W at address 3, one big-endian processor may choose to store the high byte of W at address 3, then the middle half-word of W at address 4, then the low byte of W at address 6, while another processor may choose to emit 4 byte stores, one for each byte of W, in a reverse sequence).

If an unaligned load is performed, it is guaranteed that the value being loaded, or pieces thereof, are not placed into the recipient register until the entire sequence of necessary aligned loads is completed. Specifically, if an exception occurs during an unaligned load, no register is modified.

If an unaligned store is performed, it is undefined whether, should an exception occur in the middle of the store sequence, none, some, or all bytes of memory affected by the store are actually modified.

Although an exception can occur during unaligned load or store, these instructions are not otherwise interruptible. Specifically, if a processor has initiated a sequence of aligned stores to simulate an unaligned store, and a `TIMER` interrupt is due while the sequence has not yet been completed, the `TIMER` interrupt will remain pending until the sequence is complete (or an exception occurs, making further stores impossible).

Only non-atomic single data item load/store instructions (both integer and floating-point) are affected by the presence of an unaligned operand feature. The operands of `ldm`, `stm` and `xchg` instructions must always be aligned at a 8-byte boundary, regardless of whether an unaligned operand feature is available or not.

4.4.2 Caches and write buffers

Cereon implementations can choose to reduce memory traffic by:

- Pre-fetching instructions that follow the currently executed instruction.
- Cache instructions and data loaded from memory in either two separate caches (one for instruction, another for data) or in a single shared cache for both.
- Defer writing data back to memory until the memory bus is free.

A specific implementation of a Cereon processor may choose to include an arbitrary combination of an instruction/data caches and/or write buffers. The exact policies governing the behaviour of these caches and write buffers is, also, specific to a given processor model.

4.4.3 Memory barriers

In order to ensure memory consistency when it is necessary, Cereon provides dedicated cache-control and memory barrier instructions. Specifically:

- An instruction memory barrier ensures that the instruction to be executed after the current one comes from memory and not from instruction cache or prefetch buffer.

- A data memory barrier ensures that all data written by processor to memory are actually committed to memory, and that all data subsequently read by processor will come from memory and not a cache.

Memory barrier instructions may not have the full set of effects described above if the processor does not support the required features (for example, if a processor model has a write-through cache, then executing the data memory barrier will not cause any data to be written to the memory).

5 Cereon instruction set

This chapter describes all instructions of the Cereon instruction set.

5.1 Instruction format

All Cereon instructions are 4 bytes long and are always aligned on 4-byte boundaries. There are 4 instruction formats supported:

5.1.1 I-type (Immediate)

Immediate instructions are used for operations where one of the operands is a constant encoded within the instruction itself. The same instruction format is also used for load, store and conditional branch instructions.

Bits	31 26	25 21	20 16	15 0
	op	r1	r2	immediate

Individual fields within the instruction have the following meanings:

- `op` – the operation code, specifies the action performed by this instruction.
- `r1` and `r1` – each of these fields specifies a general purpose register.
- `immediate` – this 16-bit operand can specify a signed or unsigned integer value (in arithmetic or compare instructions), a branch distance in branch instructions or address displacement in load/store instructions.

5.1.2 L-type (Long immediate)

Long immediate instructions are used to load constants into registers. As only one register is always involved, the constant field is extended, which allows for constants of a wider range. The same format is also used for load/store multiple instruction.

Bits	31 26	25 21	20 0
	op	r1	immediate

Individual fields within the instruction have the following meanings:

- `op` – the operation code, specifies the action performed by this instruction.
- `r1` – this field specifies a general purpose register (whether a general purpose register is treated as storing an integer or a real value depends on the operation code).
- `immediate` – this 20-bit operand can specify a signed or unsigned integer value or a real value (in arithmetic or compare instructions), or a register mask for load/store multiple instruction.

5.1.3 J-type (jump)

Instructions of this form are used for long jumps.

Bits	31 26	25 0
	op	target

Individual fields within the instruction have the following meanings:

- `op` – the operation code, specifies the action performed by this instruction.
- `target` – this 26-bit signed integer value specifies the jump target displacement.

5.1.4 R-type (register)

Instructions of this format constitute the majority of Cereon instructions. They are used to perform arithmetical and logical operations on data, to manage control, debug and performance monitoring registers, etc.

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	op	r1	r2	r3	function	sa

Individual fields within the instruction have the following meanings:

- `op` – the operation code, specifies the action performed by this instruction.
- `r1`, `r2` and `r3` – each of these fields specifies a general purpose, control, debug or performance monitoring register (whether general purpose, control, debug or performance monitoring register is used depends on the operation code, as does whether a general purpose register is treated as storing an integer or a real value). If the operation does not use some of these registers, the corresponding field must be 0.
- `function` – this 5-bit field represents a sub-function for come operation codes.
- `sa` – this 6-bit field specifies the shift amount for immediate shifts.

5.2 Instruction mnemonics

A mnemonic instruction name is a name that uniquely identifies the instruction. It is these mnemonic instruction names that are used in an assembler-language program.

All mnemonic names of Cereon instructions have the following format:

```
<operation>[.<data type>]
```

where:

- `<operation>` is a mnemonic name of the operation to be carried out (e.g. "`mov`" represents data movement, "`add`" represents addition, and so on).
- `<data type>` portion specifies the type of the instruction operands and/or results. Note that for some instructions (i.e. "`jal`" [jump and link], or "`svc`" [supervisor call]) there is no need to specify operand and/or result data types, as they either don't have explicit operands and/or results, or always work on operands and/or results of the same size. In this case, the data type suffix is omitted entirely.

If the `<data type>` suffix is present, it encodes one or more data types as a sequence of one of more data type specifiers. The following data type specifiers are used:

Specifier	Data type
b	Signed 8-bit byte.
ub	Unsigned 8-bit byte.
h	Signed 16-bit half-word.
uh	Unsigned 16-bit half-word.
w	Signed 32-bit word.
uw	Unsigned 32-bit word.
l	Signed 64-bit long word.
ul	Unsigned 64-bit long word.
f	Single precision (32-bit) floating point value.
d	Double precision (64-bit) floating point value.
r	General purpose register contents (64-bit).
c	Control register contents (64-bit).
g	Debug register contents (64-bit).
m	Performance monitoring register contents (64-bit).

5.3 Instruction set matrix

The following tables contain all Cereon instructions together with their operation codes.

The following notation is used in instruction encoding tables:

Symbol	Meaning
-	The instruction is not supported and is guaranteed never to be supported in the future.
*	Reserved for future use.
italic field name	Operations or codes denoted by this symbol denote a field class. The instruction must be further decoded by examining additional fields.

5.3.1 Encoding of the opcode field

Opcode		Bits 28..26							
		0	1	2	3	4	5	6	7
bits 31..29		000	001	010	011	100	101	110	111
0	000	li.l	*COP1*	addi.l	subi.l	muli.l	divi.l	modi.l	j
1	001	li.d	lir	addi.ul	subi.ul	muli.ul	divi.ul	modi.ul	jal

2	010	seqi.l	snei.l	slti.l	slei.l	sgti.l	sgei.l	slti.ul	slei.ul
3	011	andi.l	ori.l	xori.l	impli.l	ldm	stm	sgti.ul	sgei.ul
4	100	l.b	l.ub	l.h	l.uh	l.w	l.uw	l.l	xchg
5	101	s.b	s.h	s.w	s.l	l.f	l.d	s.f	s.d
6	110	beq.l	bne.l	blt.l	ble.l	bgt.l	bge.l	blt.ul	ble.ul
7	111	beq.d	bne.d	blt.d	ble.d	bgt.d	bge.d	bgt.ul	bge.ul

5.3.2 *COP1* opcode encoding of the function field

function		Bits 8..6							
		0	1	2	3	4	5	6	7
bits 10..9		000	001	010	011	100	101	110	111
0	00	*SHIFT1*	*SHIFT2*	bfi.l	bfi.l	bfe.l	bfe.l	bfe.ul	bfe.ul
1	01	*BASE1*	*BASE2*	*BASE3*	*BASE4*	*BASE5*	*	*	*
2	10	*FP1*	*DBG1*	*PM1*	*	*	*	*	*
3	11	*	*	*	*	*	*	*	*

5.3.3 *SHIFT1* encoding of the r3 field

function		Bits 13..11							
		0	1	2	3	4	5	6	7
bits 15..14		000	001	010	011	100	101	110	111
0	00	shli.b	shli.ub	shli.h	shli.uh	shli.w	shli.uw	shli.l	*
1	01	shri.b	shri.ub	shri.h	shri.uh	shri.w	shri.uw	shri.l	*
2	10	asli.b	asli.ub	asli.h	asli.uh	asli.w	asli.uw	asli.l	*
3	11	asri.b	asri.ub	asri.h	asri.uh	asri.w	asri.uw	asri.l	*

5.3.4 *SHIFT2* encoding of the r3 field

function		Bits 13..11							
		0	1	2	3	4	5	6	7
bits 15..14		000	001	010	011	100	101	110	111
0	00	roli.b	roli.ub	roli.h	roli.uh	roli.w	roli.uw	roli.l	*
1	01	rori.b	rori.ub	rori.h	rori.uh	rori.w	rori.uw	rori.l	*
2	10	beqi.l	bnei.l	blti.l	blei.l	bgti.l	bgei.l	*	*
3	11	*	*	blti.ul	blei.ul	bgti.ul	bgei.ul	*	*

5.3.5 *BASE1* function encoding of the shift amount field

Opcode		Bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	mov.cr	mov.rc	*	*	*	*	*	*

1	001	iitlb	idtlb	iitlbe	idtlbe	iitlbc	idtlbc	iitlbec	idtlbec
2	010	imb	dmb	*	*	imbc	dmbc	*	*
3	011	iret	halt	cpuid	sigp	*	*	*	*
4	100	tstp	setp	*	*	*	*	*	*
5	101	in.b	in.h	in.w	in.l	out.b	out.h	out.w	out.l
6	110	in.ub	in.uh	in.uw	*	*	*	*	*
7	111	*	*	*	*	*	*	*	–

5.3.6 *BASE2* function encoding of the shift amount field

Opcode		Bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	mov.l	cvt.bl	cvt.ubl	cvt.hl	cvt.uhl	cvt.wl	cvt.uwl	nop
1	001	and.b	and.ub	and.h	and.uh	and.w	and.uw	and.l	swap.h
2	010	or.b	or.ub	or.h	or.uh	or.w	or.uw	or.l	swap.uh
3	011	xor.b	xor.ub	xor.h	xor.uh	xor.w	xor.uw	xor.l	swap.w
4	100	not.b	not.ub	not.h	not.uh	not.w	not.uw	not.l	swap.uw
5	101	brev.b	brev.ub	brev.h	brev.uh	brev.w	brev.uw	brev.l	swap.l
6	110	seq.l	sne.l	slt.l	sle.l	sgt.l	sge.l	clz	ctz
7	111	jr	jalr	slt.ul	sle.ul	sgt.ul	sge.ul	clo	cto

5.3.7 *BASE3* function encoding of the shift amount field

Opcode		Bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	add.b	sub.b	mul.b	div.b	mod.b	abs.b	neg.b	impl.b
1	001	add.ub	sub.ub	mul.ub	div.ub	mod.ub	*	cpl2.ub	impl.ub
2	010	add.h	sub.h	mul.h	div.h	mod.h	abs.h	neg.h	impl.h
3	011	add.uh	sub.uh	mul.uh	div.uh	mod.uh	*	cpl2.uh	impl.uh
4	100	add.w	sub.w	mul.w	div.w	mod.w	abs.w	neg.w	impl.w
5	101	add.uw	sub.uw	mul.uw	div.uw	mod.uw	*	cpl2.uw	impl.uw
6	110	add.l	sub.l	mul.l	div.l	mod.l	abs.l	neg.l	impl.l
7	111	add.ul	sub.ul	mul.ul	div.ul	mod.ul	*	cpl2.ul	*

5.3.8 *BASE4* encoding of the r3 field

function		Bits 13..11							
		0	1	2	3	4	5	6	7
bits 15..14		000	001	010	011	100	101	110	111
0	00	addi.b	addi.ub	addi.h	addi.uh	addi.w	addi.uw	modi.b	modi.ub

1	01	subi.b	subi.ub	subi.h	subi.uh	subi.w	subi.uw	modi.h	modi.uh
2	10	muli.b	muli.ub	muli.h	muli.uh	muli.w	muli.uw	modi.w	modi.uw
3	11	divi.b	divi.ub	divi.h	divi.uh	divi.w	divi.uw	*	*

5.3.9 *BASE5* function encoding of the shift amount field

Opcode		Bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	shl.b	shr.b	asl.b	asr.b	rol.b	ror.b	*	*
1	001	shl.h	shr.h	asl.h	asr.h	rol.h	ror.h	*	*
2	010	shl.w	shr.w	asl.w	asr.w	rol.w	ror.w	*	*
3	011	shl.ub	shr.ub	asl.ub	asr.ub	rol.ub	ror.ub	*	*
4	100	shl.uh	shr.uh	asl.uh	asr.uh	rol.uh	ror.uh	*	*
5	101	shl.uw	shr.uw	asl.uw	asr.uw	rol.uw	ror.uw	*	*
6	110	shl.l	shr.l	asl.l	asr.l	rol.l	ror.l	*	*
7	111	getfl	setfl	rstfl	*	*	svc	brk	invi

5.3.10 *FP1* function encoding of the shift amount field

Opcode		Bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	mov.d	cvt.df	*	*	*	*	*	*
1	001	add.d	sub.d	mul.d	div.d	abs.d	neg.d	sqrt.d	*
2	010	add.f	sub.f	mul.f	div.f	abs.f	neg.f	sqrt.f	*
3	011	seq.d	sne.d	slt.d	sle.d	sgt.d	sge.d	*	*
4	100	cvt.fb	cvt.fh	cvt.fw	cvt.fl	cvt.fub	cvt.fuh	cvt.fuw	cvt.ful
5	101	cvt.bf	cvt.hf	cvt.wf	cvt.lf	cvt.ubf	cvt.uhf	cvt.uwf	cvt.ulf
6	110	cvt.db	cvt.dh	cvt.dw	cvt.dl	cvt.dub	cvt.duh	cvt.duw	cvt.dul
7	111	cvt.bd	cvt.hd	cvt.wd	cvt.ld	cvt.ubd	cvt.uhd	cvt.uwd	cvt.uld

5.3.11 *DBG1* function encoding of the shift amount field

Opcode		Bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	mov.gr	mov.rg	*	*	*	*	*	*
1	001	*	*	*	*	*	*	*	*
2	010	*	*	*	*	*	*	*	*
3	011	*	*	*	*	*	*	*	*
4	100	*	*	*	*	*	*	*	*

5	101	*	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	*	*	*	*	*

5.3.12 *PM1* function encoding of the shift amount field

Opcode		Bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	mov.mr	mov.rm	*	*	*	*	*	*
1	001	*	*	*	*	*	*	*	*
2	010	*	*	*	*	*	*	*	*
3	011	*	*	*	*	*	*	*	*
4	100	*	*	*	*	*	*	*	*
5	101	*	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	*	*	*	*	*

5.4 Instruction set reference

The following sections give detailed description of each Cereon instruction. For each instruction the following information is provided:

- Symbolic instruction name.
- Instruction bit pattern.
- The description of instruction's effect.
- The list of conditions which can cause exceptions during instruction execution. Note that these do not include exceptions that can occur during instruction fetch (`IADDRESS`, `IACCESS`, `IALIGN`, `PAGETABLE`, `IPAGEFAULT`, `IOVERFLOW`) or decoding (`OPCODE`).

The following abbreviations are used when describing the instruction set:

`?`	The instruction bit is ignored and can have any value.
`R<N>`	The contents of a general purpose register N (0<=N<=31) interpreted as an integer value
`F<N>`	The contents of a general purpose register N (0<=N<=31) interpreted as a real value
`C<N>`	The contents of a control register N (0<=N<=31)
`D<N>`	The contents of a debug register N (0<=N<=31)
`M<N>`	The contents of a performance monitoring register N (0<=N<=31)
`imm`	The 16-bit `immediate` field of the instruction

`target`	The 26-bit `target` field of the instruction
`addr`	The 64-bit address of a memory operand in an I-type instruction, calculated as `R<r2>+imm`. The `imm` is sign-extended to 64 bits before an address is calculated.
`jaddr`	The 64-bit jump target address in a J-type instruction, calculated as `$ip+(target<<2)`. The `target` is sign-extended to 64 bits before an address is calculated.
`baddr`	The 64-bit jump target address in a I-type branch instruction, calculated as `$ip+(imm<<2)`. The `imm` is sign-extended to 64 bits before an address is calculated. Note that when a branch instruction is executed, `$ip` already points to the instruction immediately following the branch, so `imm = 0` causes no actual branch.

6 The Cereon Base

This section describes instructions provided as a part of the Cereon Base functionality. These instructions are always available in every Cereon processor.

6.1 Unsupported instructions

6.1.1 invi (invalid Instruction)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	?????	?????	?????	01100	111111

All instructions that match the above bit pattern cause an `OPCODE` exception when executed. All these bit patterns are guaranteed never to be used in future Cereon ISA versions for valid instructions.

6.2 Data movement

6.2.1 mov.l (move long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	000000

Sets `R<r1> = R<r2>`.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.2.2 mov.cr (move control register to general purpose register)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01000	000000

Sets `R<r1> = C<r2>`.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.2.3 mov.rc (move general purpose register to control register) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01000	000001

Sets `C<r1> = R<r2>`.

Causes a `PRIVILEGED` exception if the `K` bit of `$state` is 0

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.2.4 li.l (load Immediate long word)

Bits	31 26	25 21	20 0
	000000	r1	immediate

Loads the integer `immediate` value, sign-extended to 64 bits, into `R<r1>`.

6.2.5 swap.h (swap bytes in half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	001111

The lower 2 bytes of `R<r2>` are swapped and placed into the lower 2 bytes of `R<r1>`. These lower 2 bytes of `R<r1>` are then sign-extended to 64 bits.

This instruction is used to convert 16-bit signed integer values between big-endian and little-endian representation.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.2.6 swap.uh (swap bytes in unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	010111

The lower 2 bytes of `R<r2>` are swapped and placed into the lower 2 bytes of `R<r1>`. These lower 2 bytes of `R<r1>` are then zero-extended to 64 bits.

This instruction is used to convert 16-bit unsigned integer values between big-endian and little-endian representation.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.2.7 swap.w (swap bytes in word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	011111

The lower 4 bytes of `R<r2>` are swapped and placed into the lower 4 bytes of `R<r1>`. These lower 4 bytes of `R<r1>` are then sign-extended to 64 bits.

This instruction is used to convert 32-bit integer values between big-endian and little-endian representation.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.2.8 swap.uw (swap bytes in unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	100111

The lower 4 bytes of `R<r2>` are swapped and placed into the lower 4 bytes of `R<r1>`. These lower 4 bytes of `R<r1>` are then zero-extended to 64 bits.

This instruction is used to convert 16-bit integer values between big-endian and little-endian representation.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.2.9 swap.l (swap bytes in long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	101111

All 8 bytes of `R<r2>` are swapped and placed into `R<r1>`.

This instruction is used to convert 64-bit integer values between big-endian and little-endian representation.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.3 Arithmetic instructions

6.3.1 add.b (add byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	000000

Sets `R<r1> = R<r2> + R<r3>`. Lower bytes of both operands are treated as signed 8-bit integer quantities that are sign-extended to 64 bits before the operation, the result is truncated to 8 lower bits and then sign-extended to 64 bits. Highest 7 bytes of both operands are ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.2 add.ub (add unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	001000

Sets `R<r1> = R<r2> + R<r3>`. Lower bytes of both operands are treated as unsigned 8-bit integer quantities that are zero-extended to 64 bits before the operation, the result is truncated to 8 lower bits and then zero-extended to 64 bits. Highest 7 bytes of both operands are ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.3 add.h (add half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	010000

Sets `R<r1> = R<r2> + R<r3>`. Lower half-words of both operands are treated as signed 16-bit integer quantities that are sign-extended to 64 bits before the operation, the result is truncated to 16 lower bits and then sign-extended to 64 bits. Highest 6 bytes of both operands are ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.4 add.uh (add unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	011000

Sets `R<r1> = R<r2> + R<r3>`. Lower half-words of both operands are treated as unsigned 16-bit integer quantities that are zero-extended to 64 bits before the operation, the result is truncated to 16 lower bits and then zero-extended to 64 bits. Highest 6 bytes of both operands are ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.5 add.w (add word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	100000

Sets `R<r1> = R<r2> + R<r3>`. Lower words of both operands are treated as signed 32-bit integer quantities that are sign-extended to 64 bits before the operation, the result is truncated to 32 lower bits and then sign-extended to 64 bits. Highest 4 bytes of both operands are ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.6 add.uw (add unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	101000

Sets `R<r1> = R<r2> + R<r3>`. Lower words of both operands are treated as unsigned 32-bit integer quantities that are zero-extended to 64 bits before the operation, the result is truncated to 32 lower bits and then zero-extended to 64 bits. Highest 4 bytes of both operands are ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.7 add.l (add long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	110000

Sets `R<r1> = R<r2> + R<r3>`. Operands are treated as signed 64-bit integer quantities.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.8 add.ul (add unsigned long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	111000

Sets `R<r1> = R<r2> + R<r3>`. Operands are treated as unsigned 64-bit integer quantities.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.9 sub.b (subtract byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	000001

Sets `R<r1> = R<r2> - R<r3>`. Lower bytes of both operands are treated as signed 8-bit integer quantities that are sign-extended to 64 bits before the operation, the result is truncated to 8 lower bits and then sign-extended to 64 bits. Highest 7 bytes of both operands are ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.10 sub.ub (subtract unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	001001

Sets `R<r1> = R<r2> - R<r3>`. Lower bytes of both operands are treated as unsigned 8-bit integer quantities that are zero-extended to 64 bits before the operation, the result is truncated to 8 lower bits and then zero-extended to 64 bits. Highest 7 bytes of both operands are ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.11 sub.h (subtract half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	010001

Sets `R<r1> = R<r2> - R<r3>`. Lower half-words of both operands are treated as signed 16-bit integer quantities that are sign-extended to 64 bits before the operation, the result is truncated to 16 lower bits and then sign-extended to 64 bits. Highest 6 bytes of both operands are ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.12 sub.uh (subtract unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	011001

Sets `R<r1> = R<r2> - R<r3>`. Lower half-words of both operands are treated as unsigned 16-bit integer quantities that are zero-extended to 64 bits before the operation, the result is truncated to 16 lower bits and then zero-extended to 64 bits. Highest 6 bytes of both operands are ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.13 sub.w (subtract word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	100001

Sets `R<r1> = R<r2> - R<r3>`. Lower words of both operands are treated as signed 32-bit integer quantities that are sign-extended to 64 bits before the operation, the result is truncated to 32 lower bits and then sign-extended to 64 bits. Highest 4 bytes of both operands are ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.14 sub.uw (subtract unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	101001

Sets `R<r1> = R<r2> - R<r3>`. Lower words of both operands are treated as unsigned 32-bit integer quantities that are zero-extended to 64 bits before the operation, the result is truncated to 32 lower bits and then zero-extended to 64 bits. Highest 4 bytes of both operands are ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.15 sub.l (subtract long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	110001

Sets `R<r1> = R<r2> - R<r3>`. Operands are treated as signed 64-bit integer quantities.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.16 sub.ul (subtract unsigned long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	111001

Sets `R<r1> = R<r2> - R<r3>`. Operands are treated as unsigned 64-bit integer quantities.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.17 mul.b (multiply byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	000010

Sets `R<r1> = R<r2> * R<r3>`. Lower bytes of both operands are treated as signed 8-bit integer quantities that are sign-extended to 64 bits before the operation, the result is truncated to 8 lower bits and then sign-extended to 64 bits. Highest 7 bytes of both operands are ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.18 mul.ub (multiply unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	001010

Sets `R<r1> = R<r2> * R<r3>`. Lower bytes of both operands are treated as unsigned 8-bit integer quantities that are zero-extended to 64 bits before the operation, the result is truncated to 8 lower bits and then zero-extended to 64 bits. Highest 7 bytes of both operands are ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.19 mul.h (multiply half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	010010

Sets `R<r1> = R<r2> * R<r3>`. Lower half-words of both operands are treated as signed 16-bit integer quantities that are sign-extended to 64 bits before the operation, the result is truncated to 16 lower bits and then sign-extended to 64 bits. Highest 6 bytes of both operands are ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.20 mul.uh (multiply unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	011010

Sets `R<r1> = R<r2> * R<r3>`. Lower half-words of both operands are treated as unsigned 16-bit integer quantities that are zero-extended to 64 bits before the operation, the result is truncated to 16 lower bits and then zero-extended to 64 bits. Highest 6 bytes of both operands are ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.21 mul.w (multiply word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	100010

Sets `R<r1> = R<r2> * R<r3>`. Lower words of both operands are treated as signed 32-bit integer quantities that are sign-extended to 64 bits before the operation, the result is truncated to 32 lower bits and then sign-extended to 64 bits. Highest 4 bytes of both operands are ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.22 mul.uw (multiply unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	101010

Sets `R<r1> = R<r2> * R<r3>`. Lower words of both operands are treated as unsigned 32-bit integer quantities that are zero-extended to 64 bits before the operation, the result is truncated to 32 lower bits and then zero-extended to 64 bits. Highest 4 bytes of both operands are ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.23 mul.l (multiply long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	110010

Sets `R<r1> = R<r2> * R<r3>`. Operands are treated as signed 64-bit integer quantities.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.24 mul.ul (multiply unsigned long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	111010

Sets `R<r1> = R<r2> * R<r3>`. Operands are treated as unsigned 64-bit integer quantities.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.25 div.b (divide byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	000011

Sets `R<r1> = R<r2> / R<r3>`. Lower bytes of both operands are treated as signed 8-bit integer quantities that are sign-extended to 64 bits before the operation, the result is truncated to 8 lower bits and then sign-extended to 64 bits. Highest 7 bytes of both operands are ignored.

Sets the `N` bit of `$flags` to 1 if the lower byte of `R<r3>=0`.
Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation (e.g. when a maximum representable 8-bit integer value is divided by -1).

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and the lower byte of `R<r3>=0`.
Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.26 div.ub (divide unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	001011

Sets `R<r1> = R<r2> / R<r3>`. Lower bytes of both operands are treated as unsigned 8-bit integer quantities that are zero-extended to 64 bits before the operation, the result is truncated to 8 lower bits and then zero-extended to 64 bits. Highest 7 bytes of both operands are ignored.

Sets the `N` bit of `$flags` to 1 if the lower byte of `R<r3>=0`.

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and the lower byte of `R<r3>=0`.

6.3.27 div.h (divide half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	010011

Sets `R<r1> = R<r2> / R<r3>`. Lower half-words of both operands are treated as signed 16-bit integer quantities that are sign-extended to 64 bits before the operation, the result is truncated to 16 lower bits and then sign-extended to 64 bits. Highest 6 bytes of both operands are ignored.

Sets the `N` bit of `$flags` to 1 if the lower half-word of `R<r3>=0`.
Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation (e.g. when a maximum representable 8-bit integer value is divided by -1).

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and the lower half-word of `R<r3>=0`.
Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.28 div.uh (divide unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	011011

Sets `R<r1> = R<r2> / R<r3>`. Lower half-words of both operands are treated as unsigned 16-bit integer quantities that are zero-extended to 64 bits before the operation, the result is truncated to 16 lower bits and then zero-extended to 64 bits. Highest 6 bytes of both operands are ignored.

Sets the `N` bit of `$flags` to 1 if the lower half-word of `R<r3>=0`.

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and the lower half-word of `R<r3>=0`.

6.3.29 div.w (divide word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	100011

Sets `R<r1> = R<r2> / R<r3>`. Lower words of both operands are treated as signed 32-bit integer quantities that are sign-extended to 64 bits before the operation, the result is truncated to 32 lower bits and then sign-extended to 64 bits. Highest 7 bytes of both operands are ignored.

Sets the `N` bit of `$flags` to 1 if the lower word of `R<r3>=0`.
Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation (e.g. when a maximum representable 8-bit integer value is divided by -1).

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and the lower word of `R<r3>=0`.
Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.30 div.uw (divide unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	101011

Sets `R<r1> = R<r2> / R<r3>`. Lower words of both operands are treated as unsigned 32-bit integer quantities that are zero-extended to 64 bits before the operation, the result is truncated to 32 lower bits and then zero-extended to 64 bits. Highest 4 bytes of both operands are ignored.

Sets the `N` bit of `$flags` to 1 if the lower word of `R<r3>=0`.

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and the lower word of `R<r3>=0`.

6.3.31 div.l (divide long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	110011

Sets `R<r1> = R<r2> / R<r3>`. Operands are treated as signed 64-bit integer quantities.

Sets the `N` bit of `$flags` to 1 if `R<r3>=0`.
Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation (e.g. when a maximum representable 8-bit integer value is divided by -1).

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and `R<r3>=0`.
Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.32 div.ul (divide unsigned long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	111011

Sets `R<r1> = R<r2> / R<r3>`. Operands are treated as unsigned 64-bit integer quantities.

Sets the `N` bit of `$flags` to 1 if `R<r3>=0`.

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and `R<r3>=0`.

6.3.33 mod.b (modulo byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	000100

Sets `R<r1> = R<r2> - (R<r2> / R<r3>) * R<r3>`. Lower bytes of both operands are treated as signed 8-bit integer quantities that are sign-extended to 64 bits before the operation, the result is truncated to 8 lower bits and then sign-extended to 64 bits. Highest 7 bytes of both operands are ignored.

Sets the `N` bit of `$flags` to 1 if the lower byte of `R<r3>=0`.

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and the lower byte of `R<r3>=0`.

6.3.34 mod.ub (modulo unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	001100

Sets `R<r1> = R<r2> - (R<r2> / R<r3>) * R<r3>`. Lower bytes of both operands are treated as unsigned 8-bit integer quantities that are zero-extended to 64 bits before the operation, the result is truncated to 8 lower bits and then zero-extended to 64 bits. Highest 7 bytes of both operands are ignored.

Sets the `N` bit of `$flags` to 1 if the lower byte of `R<r3>=0`.

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and the lower byte of `R<r3>=0`.

6.3.35 mod.h (modulo half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	010100

Sets `R<r1> = R<r2> - (R<r2> / R<r3>) * R<r3>`. Lower half-words of both operands are treated as signed 16-bit integer quantities that are sign-extended to 64 bits before the operation, the result is truncated to 16 lower bits and then sign-extended to 64 bits. Highest 6 bytes of both operands are ignored.

Sets the `N` bit of `$flags` to 1 if the lower half-word of `R<r3>=0`.

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and the lower half-word of `R<r3>=0`.

6.3.36 mod.uh (modulo unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	011100

Sets `R<r1> = R<r2> - (R<r2> / R<r3>) * R<r3>`. Lower half-words of both operands are treated as unsigned 16-bit integer quantities that are zero-extended to 64 bits before the operation, the result is truncated to 16 lower bits and then zero-extended to 64 bits. Highest 6 bytes of both operands are ignored.

Sets the `N` bit of `$flags` to 1 if the lower half-word of `R<r3>=0`.

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and the lower half-word of `R<r3>=0`.

6.3.37 mod.w (modulo word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	100100

Sets `R<r1> = R<r2> - (R<r2> / R<r3>) * R<r3>`. Lower words of both operands are treated as signed 32-bit integer quantities that are sign-extended to 64 bits before the operation, the result is truncated to 32 lower bits and then sign-extended to 64 bits. Highest 4 bytes of both operands are ignored.

Sets the `N` bit of `$flags` to 1 if the lower word of `R<r3>=0`.

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and the lower word of `R<r3>=0`.

6.3.38 mod.uw (modulo unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	101100

Sets `R<r1> = R<r2> - (R<r2> / R<r3>) * R<r3>`. Lower words of both operands are treated as unsigned 32-bit integer quantities that are zero-extended to 64 bits before the operation, the result is truncated to 32 lower bits and then zero-extended to 64 bits. Highest 4 bytes of both operands are ignored.

Sets the `N` bit of `$flags` to 1 if the lower word of `R<r3>=0`.

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and the lower word of `R<r3>=0`.

6.3.39 mod.l (modulo long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	110100

Sets `R<r1> = R<r2> - (R<r2> / R<r3>) * R<r3>`. Operands are treated as signed 64-bit integer quantities.

Sets the `N` bit of `$flags` to 1 if `R<r3>=0`.

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and the lower byte of `R<r3>=0`.

6.3.40 mod.ul (modulo unsigned long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	111100

Sets `R<r1> = R<r2> - (R<r2> / R<r3>) * R<r3>`. Operands are treated as unsigned 64-bit integer quantities.

Sets the `N` bit of `$flags` to 1 if `R<r3>=0`.

Causes a `ZDIV` exception an `N` bit of `$state` is 1 and if `R<r3>=0`.

6.3.41 abs.b (absolute value byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01010	000101

Set `R<r1>` = |`R<r2>|`. The lower byte of an operand is treated as a signed 8-bit integer quantity, which is sign-extended to 64 bits before the operation. The result is truncated to 8 lower bits and then sign-extended to 64 bits.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.
Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation (this can occur when the lower byte of an operand is 0x80).

6.3.42 abs.h (absolute value half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01010	010101

Set `R<r1>` = |`R<r2>|`. The lower half-word of an operand is treated as a signed 16-bit integer quantity, which is sign-extended to 64 bits before the operation. The result is truncated to 16 lower bits and then sign-extended to 64 bits.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.
Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation (this can occur when the lower half-word of an operand is 0x8000).

6.3.43 abs.w (absolute value word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01010	100101

Set `R<r1>` = |`R<r2>|`. The lower word of an operand is treated as a signed 32-bit integer quantity, which is sign-extended to 64 bits before the operation. The result is truncated to 32 lower bits and then sign-extended to 64 bits.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.
Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation (this can occur when the lower word of an operand is 0x80000000).

6.3.44 abs.l (absolute value long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01010	110101

Set `R<r1> =` |`R<r2>|`. Operand is treated as a signed 64-bit integer quantity.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.
Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation (this can occur when the operand is 0x8000000000000000).

6.3.45 neg.b (negate byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01010	000110

Set `R<r1> = -R<r2>`. The lower byte of an operand is treated as a signed 8-bit integer quantity, which is sign-extended to 64 bits before the operation. The result is truncated to 8 lower bits and then sign-extended to 64 bits.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.
Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation (this can occur when the lower byte of an operand is 0x80).

6.3.46 cpl2.ub (2's complement of an unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01010	001110

Set `R<r1> =` 2's complement of `R<r2>`. The lower byte of an operand is treated as an unsigned 8-bit integer quantity, which is zero-extended to 64 bits before the operation. The result is truncated to 8 lower bits and then zero-extended to 64 bits.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.3.47 neg.h (negate half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01010	010110

Set `R<r1> = -R<r2>`. The lower half-word of an operand is treated as a signed 16-bit integer quantity, which is sign-extended to 64 bits before the operation. The result is truncated to 16 lower bits and then sign-extended to 64 bits.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.
Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation (this can occur when the lower half-word of an operand is 0x8000).

6.3.48 cpl2.uh (2's complement of an unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01010	011110

Set `R<r1> =` 2's complement of `R<r2>`. The lower half-word of an operand is treated as an unsigned 16-bit integer quantity, which is zero-extended to 64 bits before the operation. The result is truncated to 16 lower bits and then zero-extended to 64 bits.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.3.49 neg.w (negate word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01010	100110

Set `R<r1> = -R<r2>`. The lower word of an operand is treated as a signed 32-bit integer quantity, which is sign-extended to 64 bits before the operation. The result is truncated to 32 lower bits and then sign-extended to 64 bits.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.
Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation (this can occur when the lower word of an operand is 0x80000000).

6.3.50 cpl2.uw (2’s complement of an unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01010	101110

Set `R<r1> =` 2’s complement of `R<r2>`. The lower word of an operand is treated as an unsigned 32-bit integer quantity, which is zero-extended to 64 bits before the operation. The result is truncated to 32 lower bits and then zero-extended to 64 bits.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.3.51 neg.l (negate long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01010	110110

Set `R<r1> =-R<r2>`. Operand is treated as a signed 64-bit integer quantity.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.
Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation (this can occur when the operand is 0x8000000000000000).

6.3.52 cpl2.ul (2’s complement of an unsigned long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01010	111110

Set `R<r1> =` 2’s complement of `R<r2>`. Operand is treated as an unsigned integer quantity.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.3.53 addi.b (add immediate byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01011	op3

Sets `R<r1> = R<r2> + op3`. Lower byte of the `R<r2>` operand is treated as signed 8-bit integer quantity that is sign-extended to 64 bits before the operation, `op3` is treated as signed 6-bit integer quantity that is sign-extended to 64 bits before the operation, the result is truncated to 8 lower bits and then sign-extended to 64 bits. Highest 7 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.54 addi.ub (add immediate unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00001	01011	op3

Sets `R<r1> = R<r2> + op3`. Lower byte of the `R<r2>` operand is treated as unsigned 8-bit integer quantity that is zero-extended to 64 bits before the operation, `op3` is treated as unsigned 6-bit integer quantity that is zero-extended to 64 bits before the operation, the result is truncated to 8 lower bits and then zero-extended to 64 bits. Highest 7 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.55 addi.h (add immediate half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00010	01011	op3

Sets `R<r1> = R<r2> + op3`. Lower half-word of the `R<r2>` operand is treated as signed 16-bit integer quantity that is sign-extended to 64 bits before the operation, `op3` is treated as signed 6-bit integer quantity that is sign-extended to 64 bits before the operation, the result is truncated to 16 lower bits and then sign-extended to 64 bits. Highest 6 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.56 addi.uh (add immediate unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00011	01011	op3

Sets `R<r1> = R<r2> + op3`. Lower half-word of the `R<r2>` operand is treated as unsigned 16-bit integer quantity that is zero-extended to 64 bits before the operation, `op3` is treated as unsigned 6-bit integer quantity that is zero-extended to 64 bits before the operation, the result is truncated to 16 lower bits and then zero-extended to 64 bits. Highest 6 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.57 addi.w (add immediate word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00100	01011	op3

Sets `R<r1> = R<r2> + op3`. Lower word of the `R<r2>` operand is treated as signed 32-bit integer quantity that is sign-extended to 64 bits before the operation, `op3` is treated as signed 6-bit integer quantity that is sign-extended to 64 bits before the operation, the result is truncated to 32 lower bits and then sign-extended to 64 bits. Highest 4 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.58 addi.uw (add immediate unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00101	01011	op3

Sets `R<r1> = R<r2> + op3`. Lower word of the `R<r2>` operand is treated as unsigned 32-bit integer quantity that is zero-extended to 64 bits before the operation, `op3` is treated as unsigned 6-bit integer quantity that is zero-extended to 64 bits before the operation, the result is truncated to 32 lower bits and then zero-extended to 64 bits. Highest 4 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.59 addi.l (add immediate long word)

Bits	31 26	25 21	20 16	15 0
	000010	r1	r2	immediate

Sets `R<r1> = R<r2> + immediate`. Operands are treated as signed 64-bit integer quantities. The `immediate` operand is sign-extended to 64 bits before addition.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.60 addi.ul (add immediate unsigned long word)

Bits	31 26	25 21	20 16	15 0
	001010	r1	r2	immediate

Sets `R<r1> = R<r2> + immediate`. Operands are treated as unsigned 64-bit integer quantities. The `immediate` operand is zero-extended to 64 bits before addition.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.61 subi.b (subtract immediate byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	01000	01011	op3

Sets `R<r1> = R<r2> - op3`. Lower byte of the `R<r2>` operand is treated as signed 8-bit integer quantity that is sign-extended to 64 bits before the operation, `op3` is treated as signed 6-bit integer quantity that is sign-extended to 64 bits before the operation, the result is truncated to 8 lower bits and then sign-extended to 64 bits. Highest 7 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.62 subi.ub (subtract immediate unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	01001	01011	op3

Sets `R<r1> = R<r2> - op3`. Lower byte of the `R<r2>` operand is treated as unsigned 8-bit integer quantity that is zero-extended to 64 bits before the operation, `op3` is treated as unsigned 6-bit integer quantity that is zero-extended to 64 bits before the operation, the result is truncated to 8 lower bits and then zero-extended to 64 bits. Highest 7 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.63 subi.h (subtract immediate half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	01010	01011	op3

Sets `R<r1> = R<r2> - op3`. Lower half-word of the `R<r2>` operand is treated as signed 16-bit integer quantity that is sign-extended to 64 bits before the operation, `op3` is treated as signed 6-bit integer quantity that is sign-extended to 64 bits before the operation, the result is truncated to 16 lower bits and then sign-extended to 64 bits. Highest 6 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.64 subi.uh (subtract immediate unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	01011	01011	op3

Sets `R<r1> = R<r2> - op3`. Lower half-word of the `R<r2>` operand is treated as unsigned 16-bit integer quantity that is zero-extended to 64 bits before the operation, `op3` is treated as unsigned 6-bit integer quantity that is zero-extended to 64 bits before the operation, the result is truncated to 16 lower bits and then zero-extended to 64 bits. Highest 6 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.65 subi.w (subtract immediate word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	01100	01011	op3

Sets `R<r1> = R<r2> - op3`. Lower word of the `R<r2>` operand is treated as signed 32-bit integer quantity that is sign-extended to 64 bits before the operation, `op3` is treated as signed 6-bit integer quantity that is sign-extended to 64 bits before the operation, the result is truncated to 32 lower bits and then sign-extended to 64 bits. Highest 4 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.66 subi.uw (subtract immediate unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	01101	01011	op3

Sets `R<r1> = R<r2> - op3`. Lower word of the `R<r2>` operand is treated as unsigned 32-bit integer quantity that is zero-extended to 64 bits before the operation, `op3` is treated as unsigned 6-bit integer quantity that is zero-extended to 64 bits before the operation, the result is truncated to 32 lower bits and then zero-extended to 64 bits. Highest 4 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.67 subi.l (subtract immediate long word)

Bits	31 26	25 21	20 16	15 0
	000011	r1	r2	immediate

Sets `R<r1> = R<r2> - immediate`. Operands are treated as signed 64-bit integer quantities. The `immediate` operand is sign-extended to 64 bits before addition.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.68 subi.ul (subtract immediate unsigned long word)

Bits	31 26	25 21	20 16	15 0
	001011	r1	r2	immediate

Sets `R<r1> = R<r2> - immediate`. Operands are treated as unsigned 64-bit integer quantities. The `immediate` operand is zero-extended to 64 bits before subtraction.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.69 muli.b (multiply immediate byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	10000	01011	op3

Sets `R<r1> = R<r2> * op3`. Lower byte of the `R<r2>` operand is treated as signed 8-bit integer quantity that is sign-extended to 64 bits before the operation, `op3` is treated as signed 6-bit integer quantity that is sign-extended to 64 bits before the operation, the result is truncated to 8 lower bits and then sign-extended to 64 bits. Highest 7 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.70 muli.ub (multiply immediate unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	10001	01011	op3

Sets `R<r1> = R<r2> * op3`. Lower byte of the `R<r2>` operand is treated as unsigned 8-bit integer quantity that is zero-extended to 64 bits before the operation, `op3` is treated as unsigned 6-bit integer quantity that is zero-extended to 64 bits before the operation, the result is truncated to 8 lower bits and then zero-extended to 64 bits. Highest 7 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.71 muli.h (multiply immediate half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	10010	01011	op3

Sets `R<r1> = R<r2> * op3`. Lower half-word of the `R<r2>` operand is treated as signed 16-bit integer quantity that is sign-extended to 64 bits before the operation, `op3` is treated as signed 6-bit integer quantity that is sign-extended to 64 bits before the operation, the result is truncated to 16 lower bits and then sign-extended to 64 bits. Highest 6 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.72 muli.uh (multiply immediate unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	10011	01011	op3

Sets `R<r1> = R<r2> * op3`. Lower half-word of the `R<r2>` operand is treated as unsigned 16-bit integer quantity that is zero-extended to 64 bits before the operation, `op3` is treated as unsigned 6-bit integer quantity that is zero-extended to 64 bits before the operation, the result is truncated to 16 lower bits and then zero-extended to 64 bits. Highest 6 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.73 muli.w (multiply immediate word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	10100	01011	op3

Sets `R<r1> = R<r2> * op3`. Lower word of the `R<r2>` operand is treated as signed 32-bit integer quantity that is sign-extended to 64 bits before the operation, `op3` is treated as signed 6-bit integer quantity that is sign-extended to 64 bits before the operation, the result is truncated to 32 lower bits and then sign-extended to 64 bits. Highest 4 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.74 muli.uw (multiply immediate unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	10101	01011	op3

Sets `R<r1> = R<r2> * op3`. Lower word of the `R<r2>` operand is treated as unsigned 32-bit integer quantity that is zero-extended to 64 bits before the operation, `op3` is treated as unsigned 6-bit integer quantity that is zero-extended to 64 bits before the operation, the result is truncated to 32 lower bits and then zero-extended to 64 bits. Highest 4 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.75 muli.l (multiply immediate long word)

Bits	31 26	25 21	20 16	15 0
	000100	r1	r2	immediate

Sets `R<r1> = R<r2> * immediate`. Operands are treated as signed 64-bit integer quantities. The `immediate` operand is sign-extended to 64 bits before multiplication.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.76 muli.ul (multiply immediate unsigned long word)

Bits	31 26	25 21	20 16	15 0
	001100	r1	r2	immediate

Sets `R<r1> = R<r2> * immediate`. Operands are treated as unsigned 64-bit integer quantities. The `immediate` operand is zero-extended to 64 bits before multiplication.

Sets the `O` bit of `$flags` to 1 if an integer overflow occurs during the operation.

Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and an integer overflow occurs during the operation.

6.3.77 divi.b (divide immediate byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	11000	01011	op3

Sets `R<r1> = R<r2> / op3`. Lower byte of the `R<r2>` operand is treated as signed 8-bit integer quantity that is sign-extended to 64 bits before the operation, `op3` is treated as signed 6-bit integer quantity that is sign-extended to 64 bits before the operation, the result is truncated to 8 lower bits and then sign-extended to 64 bits. Highest 7 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if `immediate=0`.

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and `immediate=0`.

6.3.78 divi.ub (divide immediate unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	11001	01011	op3

Sets `R<r1> = R<r2> / op3`. Lower byte of the `R<r2>` operand is treated as unsigned 8-bit integer quantity that is zero-extended to 64 bits before the operation, `op3` is treated as unsigned 6-bit integer quantity that is zero-extended to 64 bits before the operation, the result is truncated to 8 lower bits and then zero-extended to 64 bits. Highest 7 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if `immediate=0`.

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and `immediate=0`.

6.3.79 divi.h (divide immediate half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	11010	01011	op3

Sets `R<r1> = R<r2> + op3`. Lower half-word of the `R<r2>` operand is treated as signed 16-bit integer quantity that is sign-extended to 64 bits before the operation, `op3` is treated as signed 6-bit integer quantity that is sign-extended to 64 bits before the operation, the result is truncated to 16 lower bits and then sign-extended to 64 bits. Highest 6 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if `immediate=0`.

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and `immediate=0`.

6.3.80 divi.uh (divide immediate unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	11011	01011	op3

Sets `R<r1> = R<r2> / op3`. Lower half-word of the `R<r2>` operand is treated as unsigned 16-bit integer quantity that is zero-extended to 64 bits before the operation, `op3` is treated as unsigned 6-bit integer quantity that is zero-extended to 64 bits before the operation, the result is truncated to 16 lower bits and then zero-extended to 64 bits. Highest 6 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if `immediate=0`.

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and `immediate=0`.

6.3.81 divi.w (divide immediate word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	11100	01011	op3

Sets `R<r1> = R<r2> / op3`. Lower word of the `R<r2>` operand is treated as signed 32-bit integer quantity that is sign-extended to 64 bits before the operation, `op3` is treated as signed 6-bit integer quantity that is sign-extended to 64 bits before the operation, the result is truncated to 32 lower bits and then sign-extended to 64 bits. Highest 4 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if `immediate=0`.

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and `immediate=0`.

6.3.82 divi.uw (divide immediate unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	11101	01011	op3

Sets `R<r1> = R<r2> / op3`. Lower word of the `R<r2>` operand is treated as unsigned 32-bit integer quantity that is zero-extended to 64 bits before the operation, `op3` is treated as unsigned 6-bit integer quantity that is zero-extended to 64 bits before the operation, the result is truncated to 32 lower bits and then zero-extended to 64 bits. Highest 4 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if `immediate=0`.

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and `immediate=0`.

6.3.83 divi.l (divide immediate long word)

Bits	31 26	25 21	20 16	15 0
	000101	r1	r2	immediate

Sets `R<r1> = R<r2> / immediate`. Operands are treated as signed 64-bit integer quantities. The `immediate` operand is sign-extended to 64 bits before division.

Sets the `O` bit of `$flags` to 1 if `immediate=0`.

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and `immediate=0`.

6.3.84 divi.ul (divide immediate unsigned long word)

Bits	31 26	25 21	20 16	15 0
	001101	r1	r2	immediate

Sets `R<r1> = R<r2> / immediate`. Operands are treated as unsigned 64-bit integer quantities. The `immediate` operand is zero-extended to 64 bits before division.

Sets the `O` bit of `$flags` to 1 if `immediate=0`.

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and `immediate=0`.

6.3.85 modi.b (modulo immediate byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00110	01011	op3

Sets `R<r1> = R<r2> - (R<r2> / op3) * op3`. Lower byte of the `R<r2>` operand is treated as signed 8-bit integer quantity that is sign-extended to 64 bits before the operation, `op3` is treated as signed 6-bit integer quantity that is sign-extended to 64 bits before the operation, the result is truncated to 8 lower bits and then sign-extended to 64 bits. Highest 7 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if `immediate=0`.

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and `immediate=0`.

6.3.86 modi.ub (modulo immediate unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00111	01011	op3

Sets `R<r1> = R<r2> - (R<r2> / op3) * op3`Lower byte of the `R<r2>` operand is treated as unsigned 8-bit integer quantity that is zero-extended to 64 bits before the operation, `op3` is treated as unsigned 6-bit integer quantity that is zero-extended to 64 bits before the operation, the result is truncated to 8 lower bits and then zero-extended to 64 bits. Highest 7 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if `immediate=0`.

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and `immediate=0`.

6.3.87 modi.h (modulo immediate half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	01110	01011	op3

Sets `R<r1> = R<r2> - (R<r2> / op3) * op3`. Lower half-word of the `R<r2>` operand is treated as signed 16-bit integer quantity that is sign-extended to 64 bits before the operation, `op3` is treated as signed 6-bit integer quantity that is sign-extended to 64 bits before the operation, the result is truncated to 16 lower bits and then sign-extended to 64 bits. Highest 6 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if `immediate=0`.

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and `immediate=0`.

6.3.88 modi.uh (modulo immediate unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	01111	01011	op3

Sets `R<r1> = R<r2> - (R<r2> / op3) * op3`. Lower half-word of the `R<r2>` operand is treated as unsigned 16-bit integer quantity that is zero-extended to 64 bits before the operation, `op3` is treated as unsigned 6-bit integer quantity that is zero-extended to 64 bits before the operation, the result is truncated to 16 lower bits and then zero-extended to 64 bits. Highest 6 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if `immediate=0`.

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and `immediate=0`.

6.3.89 modi.w (modulo immediate word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	10110	01011	op3

Sets `R<r1> = R<r2> - (R<r2> / op3) * op3`. Lower word of the `R<r2>` operand is treated as signed 32-bit integer quantity that is sign-extended to 64 bits before the operation, `op3` is treated as signed 6-bit integer quantity that is sign-extended to 64 bits before the operation, the result is truncated to 32 lower bits and then sign-extended to 64 bits. Highest 4 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if `immediate=0`.

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and `immediate=0`.

6.3.90 modi.uw (modulo immediate unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	10111	01011	op3

Sets `R<r1> = R<r2> - (R<r2> / op3) * op3`. Lower word of the `R<r2>` operand is treated as unsigned 32-bit integer quantity that is zero-extended to 64 bits before the operation, `op3` is treated as unsigned 6-bit integer quantity that is zero-extended to 64 bits before the operation, the result is truncated to 32 lower bits and then zero-extended to 64 bits. Highest 4 bytes of the `R<r2>` operand is ignored.

Sets the `O` bit of `$flags` to 1 if `immediate=0`.

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and `immediate=0`.

6.3.91 modi.l (modulo immediate long word)

Bits	31 26	25 21	20 16	15 0
	000110	r1	r2	immediate

Sets `R<r1> = R<r2> - (R<r2> / immediate) * immediate`. Operands are treated as signed 64-bit integer quantities. The `immediate` operand is sign-extended to 64 bits before calculating the remainder.

Sets the `O` bit of `$flags` to 1 if `immediate=0`.

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and `immediate=0`.

6.3.92 modi.ul (modulo immediate unsigned long word)

Bits	31 26	25 21	20 16	15 0
	001110	r1	r2	immediate

Sets `R<r1> = R<r2> - (R<r2> / immediate) * immediate`. Operands are treated as unsigned 64-bit integer quantities. The `immediate` operand is zero-extended to 64 bits before calculating the remainder.

Sets the `O` bit of `$flags` to 1 if `immediate=0`.

Causes a `ZDIV` exception if an `N` bit of `$state` is 1 and `immediate=0`.

6.4 Boolean instructions

6.4.1 and.b (bitwise and byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	001000

Sets `R<r1> = R<r2> ∧ R<r3>`. Operation is performed simultaneously over the lower 8 bits of both operands and yields a 8-bit result, which is then sign-extended to 64 bits.

6.4.2 and.ub (bitwise and unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	001001

Sets `R<r1> = R<r2> ∧ R<r3>`. Operation is performed simultaneously over the lower 8 bits of both operands and yields a 8-bit result, which is then zero-extended to 64 bits.

6.4.3 and.h (bitwise and half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	001010

Sets `R<r1> = R<r2> ∧ R<r3>`. Operation is performed simultaneously over the lower 16 bits of both operands and yields a 16-bit result, which is then sign-extended to 64 bits.

6.4.4 and.uh (bitwise and unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	001011

Sets `R<r1> = R<r2> ∧ R<r3>`. Operation is performed simultaneously over the lower 16 bits of both operands and yields a 16-bit result, which is then zero-extended to 64 bits.

6.4.5 and.w (bitwise and word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	001100

Sets `R<r1> = R<r2> ∧ R<r3>`. Operation is performed simultaneously over the lower 32 bits of both operands and yields a 32-bit result, which is then sign-extended to 64 bits.

6.4.6 and.uw (bitwise and unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	001101

Sets `R<r1> = R<r2> ∧ R<r3>`. Operation is performed simultaneously over the lower 32 bits of both operands and yields a 32-bit result, which is then zero-extended to 64 bits.

6.4.7 and.l (bitwise and long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	001110

Sets `R<r1> = R<r2> ∧ R<r3>`. Operation is performed simultaneously over all 64 bits of both operands.

6.4.8 or.b (bitwise or byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	010000

Sets `R<r1> = R<r2> ∨ R<r3>`. Operation is performed simultaneously over the lower 8 bits of both operands and yields a 8-bit result, which is then sign-extended to 64 bits.

6.4.9 or.ub (bitwise or unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	010001

Sets `R<r1> = R<r2> ∨ R<r3>`. Operation is performed simultaneously over the lower 8 bits of both operands and yields a 8-bit result, which is then zero-extended to 64 bits.

6.4.10 or.h (bitwise or half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	010010

Sets `R<r1> = R<r2> ∨ R<r3>`. Operation is performed simultaneously over the lower 16 bits of both operands and yields a 16-bit result, which is then sign-extended to 64 bits.

6.4.11 or.uh (bitwise or unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	010011

Sets `R<r1> = R<r2> ∨ R<r3>`. Operation is performed simultaneously over the lower 16 bits of both operands and yields a 16-bit result, which is then zero-extended to 64 bits.

6.4.12 or.w (bitwise or word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	010100

Sets `R<r1> = R<r2> ∨ R<r3>`. Operation is performed simultaneously over the lower 32 bits of both operands and yields a 32-bit result, which is then sign-extended to 64 bits.

6.4.13 or.uw (bitwise or unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	010101

Sets `R<r1> = R<r2> ∨ R<r3>`. Operation is performed simultaneously over the lower 32 bits of both operands and yields a 32-bit result, which is then zero-extended to 64 bits.

6.4.14 or.l (bitwise or long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	010110

Sets `R<r1> = R<r2> ∨ R<r3>`. Operation is performed simultaneously over all 64 bits of both operands.

6.4.15 xor.b (bitwise exclusive or byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	011000

Sets `R<r1> = R<r2> ⊕ R<r3>`. Operation is performed simultaneously over the lower 8 bits of both operands and yields a 8-bit result, which is then sign-extended to 64 bits.

6.4.16 xor.ub (bitwise exclusive or unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	011001

Sets `R<r1> = R<r2> ⊕ R<r3>`. Operation is performed simultaneously over the lower 8 bits of both operands and yields a 8-bit result, which is then zero-extended to 64 bits.

6.4.17 xor.h (bitwise exclusive or half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	011010

Sets `R<r1> = R<r2> ⊕ R<r3>`. Operation is performed simultaneously over the lower 16 bits of both operands and yields a 16-bit result, which is then sign-extended to 64 bits.

6.4.18 xor.uh (bitwise exclusive or unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	011011

Sets `R<r1> = R<r2> ⊕ R<r3>`. Operation is performed simultaneously over the lower 16 bits of both operands and yields a 16-bit result, which is then zero-extended to 64 bits.

6.4.19 xor.w (bitwise exclusive or word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	011100

Sets `R<r1> = R<r2> ⊕ R<r3>`. Operation is performed simultaneously over the lower 32 bits of both operands and yields a 32-bit result, which is then sign-extended to 64 bits.

6.4.20 xor.uw (bitwise exclusive or unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	011101

Sets `R<r1> = R<r2> ⊕ R<r3>`. Operation is performed simultaneously over the lower 32 bits of both operands and yields a 32-bit result, which is then zero-extended to 64 bits.

6.4.21 xor.l (bitwise exclusive or long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	011110

Sets `R<r1> = R<r2>` ⊕ `R<r3>`. Operation is performed simultaneously over all 64 bits of both operands.

6.4.22 impl.b (bitwise implication byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	000111

Sets `R<r1> = R<r2>` ⇒ `R<r3>`. Operation is performed simultaneously over the lower 8 bits of both operands and yields a 8-bit result, which is then sign-extended to 64 bits.

6.4.23 impl.ub (bitwise implication unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	001111

Sets `R<r1> = R<r2>` ⇒ `R<r3>`. Operation is performed simultaneously over the lower 8 bits of both operands and yields a 8-bit result, which is then zero-extended to 64 bits.

6.4.24 impl.h (bitwise implication half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	010111

Sets `R<r1> = R<r2>` ⇒ `R<r3>`. Operation is performed simultaneously over the lower 16 bits of both operands and yields a 16-bit result, which is then sign-extended to 64 bits.

6.4.25 impl.uh (bitwise implication unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	011111

Sets `R<r1> = R<r2>` ⇒ `R<r3>`. Operation is performed simultaneously over the lower 16 bits of both operands and yields a 16-bit result, which is then zero-extended to 64 bits.

6.4.26 impl.w (bitwise implication word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	100111

Sets `R<r1> = R<r2> ⇒ R<r3>`. Operation is performed simultaneously over the lower 32 bits of both operands and yields a 32-bit result, which is then sign-extended to 64 bits.

6.4.27 impl.uw (bitwise implication unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	101111

Sets `R<r1> = R<r2> ⇒ R<r3>`. Operation is performed simultaneously over the lower 32 bits of both operands and yields a 32-bit result, which is then zero-extended to 64 bits.

6.4.28 impl.l (bitwise implication long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01010	110111

Sets `R<r1> = R<r2> ⇒ R<r3>`. Operation is performed simultaneously over all 64 bits of both operands.

6.4.29 andi.l (bitwise and immediate long word)

Bits	31 26	25 21	20 16	15 0
	011000	r1	r2	immediate

Sets `R<r1> = R<r2> ∧ immediate`. Operation is performed simultaneously over all 64 bits of both operands. The `immediate` operand is zero-extended to 64 bits before conjunction.

6.4.30 ori.l (bitwise or immediate long word)

Bits	31 26	25 21	20 16	15 0
	011001	r1	r2	immediate

Sets `R<r1> = R<r2> ∨ immediate`. Operation is performed simultaneously over all 64 bits of both operands. The `immediate` operand is zero-extended to 64 bits before disjunction.

6.4.31 xori.l (bitwise exclusive or immediate long word)

Bits	31 26	25 21	20 16	15 0
	011010	r1	r2	immediate

Sets `R<r1> = R<r2> ⊕ immediate`. Operation is performed simultaneously over all 64 bits of both operands. The `immediate` operand is zero-extended to 64 bits before disjunction.

6.4.32 impli.l (bitwise implication immediate long word)

Bits	31 26	25 21	20 16	15 0
	011011	r1	r2	immediate

Sets `R<r1> = R<r2> ⇒ immediate`. Operation is performed simultaneously over all 64 bits of both operands. The `immediate` operand is zero-extended to 64 bits before disjunction.

6.4.33 not.b (bitwise not byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	100000

Set `R<r1> =¬R<r2>`. Operation is performed simultaneously over the lower 8 bits of the operand, giving the lower 8 bits of the result, which is then sign-extended to 64 bits.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.4.34 not.ub (bitwise not unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	100001

Set `R<r1> =¬R<r2>`. Operation is performed simultaneously over the lower 8 bits of the operand, giving the lower 8 bits of the result, which is then zero-extended to 64 bits.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.4.35 not.h (bitwise not half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	100010

Set `R<r1> = ¬R<r2>`. Operation is performed simultaneously over the lower 16 bits of the operand, giving the lower 16 bits of the result, which is then sign-extended to 64 bits.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.4.36 not.uh (bitwise not unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	100011

Set `R<r1> = ¬R<r2>`. Operation is performed simultaneously over the lower 16 bits of the operand, giving the lower 16 bits of the result, which is then zero-extended to 64 bits.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.4.37 not.w (bitwise not word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	100100

Set `R<r1> = ¬R<r2>`. Operation is performed simultaneously over the lower 32 bits of the operand, giving the lower 32 bits of the result, which is then sign-extended to 64 bits.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.4.38 not.uw (bitwise not unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	100101

Set `R<r1> = ¬R<r2>`. Operation is performed simultaneously over the lower 32 bits of the operand, giving the lower 32 bits of the result, which is then zero-extended to 64 bits.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.4.39 not.l (bitwise not long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	100110

Set `R<r1> = ¬R<r2>`. Operation is performed simultaneously over all 64 bits of the operand.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.5 Shift instructions

6.5.1 shl.b (shift left byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	000000

Sets `R<r1> = R<r2> << R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower byte of the shifted value is treated as an 8-bit unsigned integer value, which is zero-extended to 64 bits before the shift. The shift result is truncated to the lower 8 bits and then sign-extended to 64 bits,

6.5.2 shl.ub (shift left unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	011000

Sets `R<r1> = R<r2> << R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower byte of the shifted value is treated as an 8-bit unsigned integer value, which is zero-extended to 64 bits before the shift. The shift result is truncated to the lower 8 bits and then zero-extended to 64 bits,

6.5.3 shl.h (shift left half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	001000

Sets `R<r1> = R<r2> << R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower half-word of the shifted value is treated as a 16-bit unsigned integer value, which is zero-extended to 64 bits before the shift. The shift result is truncated to the lower 16 bits and then sign-extended to 64 bits,

6.5.4 shl.uh (shift left unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	100000

Sets `R<r1> = R<r2> << R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower half-word of the shifted value is treated as a 16-bit unsigned integer value, which is zero-extended to 64 bits before the shift. The shift result is truncated to the lower 16 bits and then zero-extended to 64 bits,

6.5.5 shl.w (shift left word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	010000

Sets `R<r1> = R<r2> << R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower word of the shifted value is treated as a 32-bit unsigned integer value, which is zero-extended to 64 bits before the shift. The shift result is truncated to the lower 32 bits and then sign-extended to 64 bits,

6.5.6 shl.uw (shift left unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	101000

Sets `R<r1> = R<r2> << R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower word of the shifted value is treated as a 32-bit unsigned integer value, which is zero-extended to 64 bits before the shift. The shift result is truncated to the lower 32 bits and then zero-extended to 64 bits,

6.5.7 shl.l (shift left long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	110000

Sets `R<r1> = R<r2> << R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0.

6.5.8 shr.b (shift right byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	000001

Sets `R<r1> = R<r2> >> R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower byte of the shifted value is treated as an 8-bit unsigned integer value, which is zero-extended to 64 bits before the shift. The shift result is truncated to the lower 8 bits and then sign-extended to 64 bits,

6.5.9 shr.ub (shift right unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	011001

Sets `R<r1> = R<r2> >> R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower byte of the shifted value is treated as an 8-bit unsigned integer value, which is zero-extended to 64 bits before the shift. The shift result is truncated to the lower 8 bits and then zero-extended to 64 bits,

6.5.10 shr.h (shift right half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	001001

Sets `R<r1> = R<r2> >> R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower half-word of the shifted value is treated as a 16-bit unsigned integer value, which is sign-extended to 64 bits before the shift. The shift result is truncated to the lower 16 bits and then zero-extended to 64 bits,

6.5.11 shr.uh (shift right unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	100001

Sets `R<r1> = R<r2> >> R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower half-word of the shifted value is treated as a 16-bit unsigned integer value, which is zero-extended to 64 bits before the shift. The shift result is truncated to the lower 16 bits and then zero-extended to 64 bits,

6.5.12 shr.w (shift right word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	010001

Sets `R<r1> = R<r2> >> R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower word of the shifted value is treated as a 32-bit unsigned integer value, which is sign-extended to 64 bits before the shift. The shift result is truncated to the lower 32 bits and then sign-extended to 64 bits,

6.5.13 shr.uw (shift right unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	101001

Sets `R<r1> = R<r2> >> R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower word of the shifted value is treated as a 32-bit unsigned integer value, which is zero-extended to 64 bits before the shift. The shift result is truncated to the lower 32 bits and then zero-extended to 64 bits,

6.5.14 shr.l (shift right long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	110001

Sets `R<r1> = R<r2> >> R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0.

6.5.15 asl.b (arithmetic shift left byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	000010

Sets `R<r1> = R<r2> << R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. The lower 7 bits of the `R<r2>` are shifted by the required number of bits, with new bits introduced by shift set to 0. The shift result is then sign-extended to 64 bits.

6.5.16 asl.ub (arithmetic shift left unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	011010

Sets `R<r1> = R<r2> << R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. The lower 7 bits of the `R<r2>` are shifted by the required number of bits, with new bits introduced by shift set to 0. The shift result is then zero-extended to 64 bits.

6.5.17 asl.h (arithmetic shift left half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	001010

Sets `R<r1> = R<r2> << R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. The lower 15 bits of the `R<r2>` are shifted by the required number of bits, with new bits introduced by shift set to 0. The shift result is then sign-extended to 64 bits.

6.5.18 asl.uh (arithmetic shift left unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	100010

Sets `R<r1> = R<r2> << R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. The lower 15 bits of the `R<r2>` are shifted by the required number of bits, with new bits introduced by shift set to 0. The shift result is then zero-extended to 64 bits.

6.5.19 asl.w (arithmetic shift left word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	010010

Sets `R<r1> = R<r2> << R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. The lower 31 bits of the `R<r2>` are shifted by the required number of bits, with new bits introduced by shift set to 0. The shift result is then sign-extended to 64 bits.

6.5.20 asl.uw (arithmetic shift left unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	101010

Sets `R<r1> = R<r2> << R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. The lower 31 bits of the `R<r2>` are shifted by the required number of bits, with new bits introduced by shift set to 0. The shift result is then zero-extended to 64 bits.

6.5.21 asl.l (arithmetic shift left long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	110010

Sets `R<r1> = R<r2> << R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. The lower 63 bits of the `R<r2>` are shifted by the required number of bits, with new bits introduced by shift set to 0.

6.5.22 asr.b (arithmetic shift right byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	000011

Sets `R<r1> = R<r2> >> R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to the copy of the original sign bit of `R<r2>`. The lower byte of the shifted value is treated as an 8-bit signed integer value, which is sign-extended to 64 bits before the shift. The shift result is truncated to the lower 8 bits and then sign-extended to 64 bits,

6.5.23 asr.ub (arithmetic shift right unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	011011

Sets `R<r1> = R<r2> >> R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to the copy of the original sign bit of `R<r2>`. The lower byte of the shifted value is treated as an 8-bit signed integer value, which is sign-extended to 64 bits before the shift. The shift result is truncated to the lower 8 bits and then zero-extended to 64 bits,

6.5.24 asr.h (arithmetic shift right half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	001011

Sets `R<r1> = R<r2> >> R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to the copy of the original sign bit of `R<r2>`. The lower half-word of the shifted value is treated as a 16-bit signed integer value, which is sign-extended to 64 bits before the shift. The shift result is truncated to the lower 16 bits and then sign-extended to 64 bits,

6.5.25 asr.uh (arithmetic shift right unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	100011

Sets `R<r1> = R<r2> >> R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to the copy of the original sign bit of `R<r2>`. The lower half-word of the shifted value is treated as a 16-bit signed integer value, which is sign-extended to 64 bits before the shift. The shift result is truncated to the lower 16 bits and then zero-extended to 64 bits,

6.5.26 asr.w (arithmetic shift right word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	010011

Sets `R<r1> = R<r2> >> R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to the copy of the original sign bit of `R<r2>`. The lower word of the shifted value is treated as a 32-bit signed integer value, which is sign-extended to 64 bits before the shift. The shift result is truncated to the lower 32 bits and then sign-extended to 64 bits,

6.5.27 asr.uw (arithmetic shift right unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	101011

Sets `R<r1> = R<r2> >> R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to the copy of the original sign bit of `R<r2>`. The lower word of the shifted value is treated as a 32-bit signed integer value, which is sign-extended to 64 bits before the shift. The shift result is truncated to the lower 32 bits and then zero-extended to 64 bits,

6.5.28 asr.l (arithmetic shift right long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	110011

Sets `R<r1> = R<r2> >> R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to the copy of the original sign bit of `R<r2>`.

6.5.29 rol.b (rotate left byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	000100

Sets `R<r1> = R<r2> << R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. Bits are rotated, i.e. each new bit introduced at one end of the shifted value is a copy of the bit pushed out of the other end by the shift. The lower byte of the shifted value is treated as an 8-bit unsigned integer value that is rotated and then sign-extended to 64 bits.

6.5.30 rol.ub (rotate left unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	011100

Sets `R<r1> = R<r2> << R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. Bits are rotated, i.e. each new bit introduced at one end of the shifted value is a copy of the bit pushed out of the other end by the shift. The lower byte of the shifted value is treated as an 8-bit unsigned integer value that is rotated and then zero-extended to 64 bits.

6.5.31 rol.h (rotate left half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	001100

Sets `R<r1> = R<r2> << R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. Bits are rotated, i.e. each new bit introduced at one end of the shifted value is a copy of the bit pushed out of the other end by the shift. The lower half-word of the shifted value is treated as a 16-bit unsigned integer value that is rotated and then sign-extended to 64 bits.

6.5.32 rol.uh (rotate left unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	100100

Sets `R<r1> = R<r2> << R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. Bits are rotated, i.e. each new bit introduced at one end of the shifted value is a copy of the bit pushed out of the other end by the shift. The lower half-word of the shifted value is treated as a 16-bit unsigned integer value that is rotated and then zero-extended to 64 bits.

6.5.33 rol.w (rotate left word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	010100

Sets `R<r1> = R<r2> << R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. Bits are rotated, i.e. each new bit introduced at one end of the shifted value is a copy of the bit pushed out of the other end by the shift. The lower word of the shifted value is treated as a 32-bit unsigned integer value that is rotated and then sign-extended to 64 bits.

6.5.34 rol.uw (rotate left unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	101100

Sets `R<r1> = R<r2> << R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. Bits are rotated, i.e. each new bit introduced at one end of the shifted value is a copy of the bit pushed out of the other end by the shift. The lower word of the shifted value is treated as a 32-bit unsigned integer value that is rotated and then zero-extended to 64 bits.

6.5.35 rol.l (rotate left)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	110100

Sets `R<r1> = R<r2> << R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. Bits are rotated, i.e. each new bit introduced at one end of the shifted value is a copy of the bit pushed out of the other end by the shift.

6.5.36 ror.b (rotate right byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	000101

Sets `R<r1> = R<r2> >> R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. Bits are rotated, i.e. each new bit introduced at one end of the shifted value is a copy of the bit pushed out of the other end by the shift. The lower byte of the shifted value is treated as an 8-bit unsigned integer value that is rotated and then sign-extended to 64 bits.

6.5.37 ror.ub (rotate right unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	011101

Sets `R<r1> = R<r2> >> R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. Bits are rotated, i.e. each new bit introduced at one end of the shifted value is a copy of the bit pushed out of the other end by the shift. The lower byte of the shifted value is treated as an 8-bit unsigned integer value that is rotated and then zero-extended to 64 bits.

6.5.38 ror.h (rotate right half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	001101

Sets `R<r1> = R<r2> >> R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. Bits are rotated, i.e. each new bit introduced at one end of the shifted value is a copy of the bit pushed out of the other end by the shift. The lower half-word of the shifted value is treated as a 16-bit unsigned integer value that is rotated and then sign-extended to 64 bits.

6.5.39 ror.uh (rotate right unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	100101

Sets `R<r1> = R<r2> >> R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. Bits are rotated, i.e. each new bit introduced at one end of the shifted value is a copy of the bit pushed out of the other end by the shift. The lower half-word of the shifted value is treated as a 16-bit unsigned integer value that is rotated and then zero-extended to 64 bits.

6.5.40 ror.w (rotate right word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	010101

Sets `R<r1> = R<r2> >> R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. Bits are rotated, i.e. each new bit introduced at one end of the shifted value is a copy of the bit pushed out of the other end by the shift. The lower word of the shifted value is treated as a 32-bit unsigned integer value that is rotated and then sign-extended to 64 bits.

6.5.41 ror.uw (rotate right unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	101101

Sets `R<r1> = R<r2> >> R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. Bits are rotated, i.e. each new bit introduced at one end of the shifted value is a copy of the bit pushed out of the other end by the shift. The lower word of the shifted value is treated as a 32-bit unsigned integer value that is rotated and then zero-extended to 64 bits.

6.5.42 ror.l (rotate right long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01100	110101

Sets `R<r1> = R<r2> >> R<r3>`. The shift counter is treated as an unsigned integer quantity with all bits significant. Bits are rotated, i.e. each new bit introduced at one end of the shifted value is a copy of the bit pushed out of the other end by the shift.

6.5.43 shli.b (shift left immediate byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	00000	sa

Sets `R<r1> = R<r2> << sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower byte of the shifted value is treated as an 8-bit unsigned integer value, which is zero-extended to 64 bits before the shift. The shift result is truncated to the lower 8 bits and then sign-extended to 64 bits,

6.5.44 shli.ub (shift left immediate unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00001	00000	sa

Sets `R<r1> = R<r2> << sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower byte of the shifted value is treated as an 8-bit unsigned integer value, which is zero-extended to 64 bits before the shift. The shift result is truncated to the lower 8 bits and then zero-extended to 64 bits,

6.5.45 shli.h (shift left immediate half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00010	00000	sa

Sets `R<r1> = R<r2> << sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower half-word of the shifted value is treated as a 16-bit unsigned integer value, which is zero-extended to 64 bits before the shift. The shift result is truncated to the lower 16 bits and then sign-extended to 64 bits,

6.5.46 shli.uh (shift left immediate unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00011	00000	sa

Sets `R<r1> = R<r2> << sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower half-word of the shifted value is treated as a 16-bit unsigned integer value, which is zero-extended to 64 bits before the shift. The shift result is truncated to the lower 16 bits and then zero-extended to 64 bits,

6.5.47 shli.w (shift left immediate word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00100	00000	sa

Sets `R<r1> = R<r2> << sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower word of the shifted value is treated as a 32-bit unsigned integer value, which is zero-extended to 64 bits before the shift. The shift result is truncated to the lower 32 bits and then sign-extended to 64 bits,

6.5.48 shli.uw (shift left immediate unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00101	00000	sa

Sets `R<r1> = R<r2> << sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower word of the shifted value is treated as a 32-bit unsigned integer value, which is zero-extended to 64 bits before the shift. The shift result is truncated to the lower 32 bits and then zero-extended to 64 bits,

6.5.49 shli.l (shift left immediate long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00110	00000	sa

Sets `R<r1> = R<r2> << sa`. The shift counter is treated as an unsigned integer quantity. New bits introduced by shift are set to 0.

6.5.50 shri.b (shift right immediate byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	01000	00000	sa

Sets `R<r1> = R<r2> >> sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower byte of the shifted value is treated as an 8-bit unsigned integer value, which is zero-extended to 64 bits before the shift. The shift result is truncated to the lower 8 bits and then sign-extended to 64 bits,

6.5.51 shri.ub (shift right immediate unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	01001	00000	sa

Sets `R<r1> = R<r2> >> sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower byte of the shifted value is treated as an 8-bit unsigned integer value, which is zero-extended to 64 bits before the shift. The shift result is truncated to the lower 8 bits and then zero-extended to 64 bits,

6.5.52 shri.h (shift right immediate half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	01010	00000	sa

Sets `R<r1> = R<r2> >> sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower half-word of the shifted value is treated as a 16-bit unsigned integer value, which is zero-extended to 64 bits before the shift. The shift result is truncated to the lower 16 bits and then sign-extended to 64 bits,

6.5.53 shri.uh (shift right immediate unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	01011	00000	sa

Sets `R<r1> = R<r2> >> sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower half-word of the shifted value is treated as a 16-bit unsigned integer value, which is zero-extended to 64 bits before the shift. The shift result is truncated to the lower 16 bits and then zero-extended to 64 bits,

6.5.54 shri.w (shift right immediate word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	01100	00000	sa

Sets `R<r1> = R<r2> >> sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower word of the shifted value is treated as a 32-bit unsigned integer value, which is zero-extended to 64 bits before the shift. The shift result is truncated to the lower 32 bits and then sign-extended to 64 bits,

6.5.55 shri.uw (shift right immediate unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	01101	00000	sa

Sets `R<r1> = R<r2> >> sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower word of the shifted value is treated as a 32-bit unsigned integer value, which is zero-extended to 64 bits before the shift. The shift result is truncated to the lower 32 bits and then zero-extended to 64 bits,

6.5.56 shri.l (shift right immediate long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	01110	00000	sa

Sets `R<r1> = R<r2> >> sa`. The shift counter is treated as an unsigned integer quantity. New bits introduced by shift are set to 0.

6.5.57 asli.b (arithmetic shift left immediate byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	10000	00000	sa

Sets `R<r1> = R<r2> << sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. The lower 7 bits of the `R<r2>` are shifted by the required number of bits, with new bits introduced by shift set to 0. The shift result is then sign-extended to 64 bits.

6.5.58 asli.ub (arithmetic shift left immediate unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	10001	00000	sa

Sets `R<r1> = R<r2> << sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. The lower 7 bits of the `R<r2>` are shifted by the required number of bits, with new bits introduced by shift set to 0. The shift result is then zero-extended to 64 bits.

6.5.59 asli.h (arithmetic shift left immediate half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	10010	00000	sa

Sets `R<r1> = R<r2> << sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. The lower 15 bits of the `R<r2>` are shifted by the required number of bits, with new bits introduced by shift set to 0. The shift result is then sign-extended to 64 bits.

6.5.60 asli.uh (arithmetic shift left immediate unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	10011	00000	sa

Sets `R<r1> = R<r2> << sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. The lower 15 bits of the `R<r2>` are shifted by the required number of bits, with new bits introduced by shift set to 0. The shift result is then zero-extended to 64 bits.

6.5.61 asli.w (arithmetic shift left immediate word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	10100	00000	sa

Sets `R<r1> = R<r2> << sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. The lower 31 bits of the `R<r2>` are shifted by the required number of bits, with new bits introduced by shift set to 0. The shift result is then sign-extended to 64 bits.

6.5.62 asli.uw (arithmetic shift left immediate unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	10101	00000	sa

Sets `R<r1> = R<r2> << sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. The lower 31 bits of the `R<r2>` are shifted by the required number of bits, with new bits introduced by shift set to 0. The shift result is then zero-extended to 64 bits.

6.5.63 asli.l (arithmetic shift left immediate long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	10110	00000	sa

Sets `R<r1> = R<r2> << sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. The lower 63 bits of the `R<r2>` are shifted by the required number of bits, with new bits introduced by shift set to 0.

6.5.64 asri.b (arithmetic shift right immediate byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	11000	00000	sa

Sets `R<r1> = R<r2> >> sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to the copy of the extended operand's sign bit. The lower byte of the shifted value is treated as an 8-bit signed integer value, which is sign-extended to 64 bits before the shift. The shift result is truncated to the lower 8 bits and then sign-extended to 64 bits,

6.5.65 asri.ub (arithmetic shift right immediate unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	11001	00000	sa

Sets `R<r1> = R<r2> >> sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to the copy of the extended operand's sign bit. The lower byte of the shifted value is treated as an 8-bit signed integer value, which is sign-extended to 64 bits before the shift. The shift result is truncated to the lower 8 bits and then zero-extended to 64 bits,

6.5.66 asri.h (arithmetic shift right immediate half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	11010	00000	sa

Sets `R<r1> = R<r2> >> sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to the copy of the extended operand's sign bit. The lower half-word of the shifted value is treated as a 16-bit signed integer value, which is sign-extended to 64 bits before the shift. The shift result is truncated to the lower 16 bits and then sign-extended to 64 bits,

6.5.67 asri.uh (arithmetic shift right immediate unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	11011	00000	sa

Sets `R<r1> = R<r2> >> sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to the copy of the extended operand's sign bit. The lower half-word of the shifted value is treated as a 16-bit signed integer value, which is sign-extended to 64 bits before the shift. The shift result is truncated to the lower 16 bits and then zero-extended to 64 bits,

6.5.68 asri.w (arithmetic shift right immediate word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	11100	00000	sa

Sets `R<r1> = R<r2> >> sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to the copy of the extended operand's sign bit. The lower word of the shifted value is treated as a 32-bit signed integer value, which is sign-extended to 64 bits before the shift. The shift result is truncated to the lower 32 bits and then sign-extended to 64 bits,

6.5.69 asri.uw (arithmetic shift right immediate unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	11101	00000	sa

Sets `R<r1> = R<r2> >> sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to the copy of the extended operand's sign bit. The lower word of the shifted value is treated as a 32-bit signed integer value, which is sign-extended to 64 bits before the shift. The shift result is truncated to the lower 32 bits and then zero-extended to 64 bits,

6.5.70 asri.l (arithmetic shift right immediate long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	11110	00000	sa

Sets `R<r1> = R<r2> >> sa`. The shift counter is treated as an unsigned integer quantity. New bits introduced by shift are set to the copy of the original sign bit of `R<r2>`.

6.5.71 roli.b (rotate left immediate byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	00001	sa

Sets `R<r1> = R<r2> << sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. The lower 8 bits of an operand are rotated, i.e. each new bit introduced at either end of the shifted value is a copy of the bit pushed out by the shift. After rotation, these lower 8 bits are sign-extended to 64 bits and placed into `R<r1>`.

6.5.72 roli.ub (rotate left immediate unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00001	00001	sa

Sets `R<r1> = R<r2> << sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. The lower 8 bits of an operand are rotated, i.e. each new bit introduced at either end of the shifted value is a copy of the bit pushed out by the shift. After rotation, these lower 8 bits are zero-extended to 64 bits and placed into `R<r1>`.

6.5.73 roli.h (rotate left immediate half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00010	00001	sa

Sets `R<r1> = R<r2> << sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower 16 bits of an operand are rotated, i.e. each new bit introduced at either end of the shifted value is a copy of the bit pushed out by the shift. After rotation, these lower 16 bits are sign-extended to 64 bits and placed into `R<r1>`.

6.5.74 roli.uh (rotate left immediate unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00011	00001	sa

Sets `R<r1> = R<r2> << sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower 16 bits of an operand are rotated, i.e. each new bit introduced at either end of the shifted value is a copy of the bit pushed out by the shift. After rotation, these lower 16 bits are sign-extended to 64 bits and placed into `R<r1>`.

6.5.75 roli.w (rotate left immediate word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00100	00001	sa

Sets `R<r1> = R<r2> << sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower 32 bits of an operand are rotated, i.e. each new bit introduced at either end of the shifted value is a copy of the bit pushed out by the shift. After rotation, these lower 8 bits are sign-extended to 64 bits and placed into `R<r1>`.

6.5.76 roli.uw (rotate left immediate unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00101	00001	sa

Sets `R<r1> = R<r2> << sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower 32 bits of an operand are rotated, i.e. each new bit introduced at either end of the shifted value is a copy of the bit pushed out by the shift. After rotation, these lower 8 bits are sign-extended to 64 bits and placed into `R<r1>`.

6.5.77 roli.l (rotate left immediate long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00110	00001	sa

Sets `R<r1> = R<r2> << sa`. The shift counter is treated as an unsigned integer quantity. Bits are rotated, i.e. each new bit introduced at one end of the shifted value is a copy of the bit pushed out of the other end by the shift.

6.5.78 rori.b (rotate right immediate byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	01000	00001	sa

Sets `R<r1> = R<r2> >> sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. The lower 8 bits of an operand are rotated, i.e. each new bit introduced at either end of the shifted value is a copy of the bit pushed out by the shift. After rotation, these lower 8 bits are sign-extended to 64 bits and placed into `R<r1>`.

6.5.79 rori.ub (rotate right immediate unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	01001	00001	sa

Sets `R<r1> = R<r2> >> sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. The lower 8 bits of an operand are rotated, i.e. each new bit introduced at either end of the shifted value is a copy of the bit pushed out by the shift. After rotation, these lower 8 bits are zero-extended to 64 bits and placed into `R<r1>`.

6.5.80 rori.h (rotate right immediate half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	01010	00001	sa

Sets `R<r1> = R<r2> >> sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower 16 bits of an operand are rotated, i.e. each new bit introduced at either end of the shifted value is a copy of the bit pushed out by the shift. After rotation, these lower 16 bits are sign-extended to 64 bits and placed into `R<r1>`.

6.5.81 rori.uh (rotate right immediate unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	01011	00001	sa

Sets `R<r1> = R<r2> >> sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower 16 bits of an operand are rotated, i.e. each new bit introduced at either end of the shifted value is a copy of the bit pushed out by the shift. After rotation, these lower 16 bits are sign-extended to 64 bits and placed into `R<r1>`.

6.5.82 rori.w (rotate right immediate word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	01100	00001	sa

Sets `R<r1> = R<r2> >> sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower 32 bits of an operand are rotated, i.e. each new bit introduced at either end of the shifted value is a copy of the bit pushed out by the shift. After rotation, these lower 8 bits are sign-extended to 64 bits and placed into `R<r1>`.

6.5.83 rori.uw (rotate right immediate unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	01101	00001	sa

Sets `R<r1> = R<r2> >> sa`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0. The lower 32 bits of an operand are rotated, i.e. each new bit introduced at either end of the shifted value is a copy of the bit pushed out by the shift. After rotation, these lower 8 bits are sign-extended to 64 bits and placed into `R<r1>`.

6.5.84 rori.l (rotate right immediate long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	01110	00001	sa

Sets `R<r1> = R<r2> >> sa`. The shift counter is treated as an unsigned integer quantity. Bits are rotated, i.e. each new bit introduced at one end of the shifted value is a copy of the bit pushed out of the other end by the shift.

6.6 Bit manipulation instructions

6.6.1 clz (Count Leading Zeroes)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	110110

Stores the number of leading ‘0’ bits in `R<r2>` and stores the result in `R<r1>`.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.6.2 ctz (Count Tailing Zeroes)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	110111

Stores the number of tailing '0' bits in `R<r2>` and stores the result in `R<r1>`.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.6.3 clo (Count Leading Ones)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	111110

Stores the number of leading '1' bits in `R<r2>` and stores the result in `R<r1>`.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.6.4 cto (Count Tailing Ones)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	111111

Stores the number of tailing '1' bits in `R<r2>` and stores the result in `R<r1>`.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.6.5 bfe.l (bit field extract from long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	cnt	00100	sa

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	cnt	00101	sa

Extracts a bit field from `R<r2>` into `R<r1>`. The number `N` of bits to extract (i.e. the size of the bit field) is `cnt` (for the 1st `bfe` form) or `cnt+32` (for the 2nd `bfe` form). The bits `[sa..sa+N)` of `R<r2>` are treated as an `N`-bit signed integer value, which is sign-extended to 64 bits and the result placed into `R<r1>`.

Causes an `OPERAND` exception if `N = 0` or `N + sa > 64`.

6.6.6 bfe.ul (bit field extract from unsigned long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	cnt	00110	sa

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	cnt	00111	sa

Extracts a bit field from `R<r2>` into `R<r1>`. The number `N` of bits to extract (i.e. the size of the bit field) is `cnt` (for the 1st `bfeu` form) or `cnt+32` (for the 2nd `bfeu` form). The bits `[sa..sa+N)` of `R<r2>` are treated as an `N`-bit unsigned integer value, which is zero-extended to 64 bits and the result placed into `R<r1>`.

Causes an `OPERAND` exception if `N = 0` or `N + sa > 64`.

6.6.7 bfi.l (bit field inject into long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	cnt	00010	sa

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	cnt	00011	sa

Injects a bit field from `R<r2>` into `R<r1>`. The number `N` of bits to inject (i.e. the size of the bit field) is `cnt` (for the 1st `bfi` form) or `cnt+32` (for the 2nd `bfi` form). The lower `N` bits of `R<r2>` replace the bits `[sa..sa+N)` of `R<r1>`, with other bits of `R<r1>` remaining unaffected

Causes an `OPERAND` exception if `N = 0` or `N + sa > 64`.

6.6.8 brev.b (bit reversal in byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	101000

Reverses positions of 8 lower bits of `R<r2>` (i.e. bit 0 is swapped with bit 7, bit 1 with bit 6, etc.), sign-extends the reversed bits to 64 bits and placed the result into `R<r1>`.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.6.9 brev.ub (bit reversal in unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	101001

Reverses positions of 8 lower bits of `R<r2>` (i.e. bit 0 is swapped with bit 7, bit 1 with bit 6, etc.), zero-extends the reversed bits to 64 bits and placed the result into `R<r1>`.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.6.10 brev.h (bit reversal in half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	101010

Reverses positions of 16 lower bits of `R<r2>` (i.e. bit 0 is swapped with bit 15, bit 1 with bit 14, etc.), sign-extends the reversed bits to 64 bits and placed the result into `R<r1>`.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.6.11 brev.uh (bit reversal in unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	101011

Reverses positions of 16 lower bits of `R<r2>` (i.e. bit 0 is swapped with bit 15, bit 1 with bit 14, etc.), zero-extends the reversed bits to 64 bits and placed the result into `R<r1>`.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.6.12 brev.w (bit reversal in word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	101100

Reverses positions of 32 lower bits of `R<r2>` (i.e. bit 0 is swapped with bit 31, bit 1 with bit 30, etc.), sign-extends the reversed bits to 64 bits and placed the result into `R<r1>`.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.6.13 brev.uw (bit reversal in unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	101101

Reverses positions of 32 lower bits of `R<r2>` (i.e. bit 0 is swapped with bit 31, bit 1 with bit 30, etc.), zero-extends the reversed bits to 64 bits and placed the result into `R<r1>`.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.6.14 brev.l (bit reversal in long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	101110

Reverses positions of all bits of `R<r2>` (i.e. bit 0 is swapped with bit 63, bit 1 with bit 62, etc.) and placed the result into `R<r1>`.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.7 Data type conversion instructions

6.7.1 cvt.bl (convert byte to long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	000001

Set `R<r1>` = lower 8 bits of `R<r2>`. The operand is sign-extended to 64 bits before assignment.

Sets the `O` bit of `$flags` to 1 if one of the bits 8..63 of an operand is not equal to bit 7.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.
Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and one of the bits 8..63 of an operand is not equal to bit 7.

6.7.2 cvt.ubl (convert unsigned byte to long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	000010

Set `R<r1>` = lower 8 bits of `R<r2>`. The operand is zero-extended to 64 bits before assignment.

Sets the `O` bit of `$flags` to 1 if one of the bits 8..63 of an operand is not 0.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.
Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and one of the bits 8..63 of an operand is not 0.

6.7.3 cvt.hl (convert half-word to long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	000011

Set `R<r1>` = lower 16 bits of `R<r2>`. The operand is sign-extended to 64 bits before assignment.

Sets the `O` bit of `$flags` to 1 if one of the bits 16..63 of an operand is not equal to bit 15.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.
Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and one of the bits 16..63 of an operand is not equal to bit 15.

6.7.4 cvt.uhl (convert unsigned half-word to long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	000100

Set `R<r1>` = lower 16 bits of `R<r2>`. The operand is zero-extended to 64 bits before assignment.

Sets the `O` bit of `$flags` to 1 if one of the bits 16..63 of an operand is not 0.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.
Causes an `IOVERFLOW` exception if an `O` bit of `$state` is 1 and one of the bits 16..63 of an operand is not 0.

6.7.5 cvt.wl (convert word to long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	000101

Set `R<r1>` = lower 32 bits of `R<r2>`. The operand is sign-extended to 64 bits before assignment.

Sets the `O` bit of `$flags` to 1 if one of the bits 32..63 of an operand is not equal to bit 31.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

Causes an IOVERFLOW exception if an O bit of $state is 1 and one of the bits 32..63 of an operand is not equal to bit 31.

6.7.6 cvt.uwl (convert unsigned word to long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01001	000110

Set R<r1> = lower 32 bits of R<r2>. The operand is zero-extended to 64 bits before assignment.

Sets the O bit of $flags to 1 if one of the bits 32..63 of an operand is not 0.

Causes an OPCODE exception if bits 11..15 of the instructions are not 0.
Causes an IOVERFLOW exception if an O bit of $state is 1 and one of the bits 32..63 of an operand is not 0.

6.8 Comparison instructions

6.8.1 seq.l (set equal long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	110000

If R<r2> = R<r3>, then set R<r1> = 1, otherwise then set R<r1> = 0. Operands are compared as signed integer quantities.

6.8.2 sne.l (set not equal long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	110001

If R<r2> ≠ R<r3>, then set R<r1> = 1, otherwise then set R<r1> = 0. Operands are compared as signed integer quantities.

6.8.3 slt.l (set less than long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	110010

If R<r2> < R<r3>, then set R<r1> = 1, otherwise then set R<r1> = 0. Operands are compared as signed integer quantities.

6.8.4 sle.l (set less than or equal long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	110011

If `R<r2>` ≤ `R<r3>`, then set `R<r1> = 1`, otherwise then set `R<r1> = 0`. Operands are compared as signed integer quantities.

6.8.5 sgt.l (set greater than long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	110100

If `R<r2>` > `R<r3>`, then set `R<r1> = 1`, otherwise then set `R<r1> = 0`. Operands are compared as signed integer quantities.

6.8.6 sge.l (set greater than or equal)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	110101

If `R<r2>` ≥ `R<r3>`, then set `R<r1> = 1`, otherwise then set `R<r1> = 0`. Operands are compared as signed integer quantities.

6.8.7 slt.ul (set less than unsigned long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	111010

If `R<r2>` < `R<r3>`, then set `R<r1> = 1`, otherwise then set `R<r1> = 0`. Operands are compared as unsigned integer quantities.

6.8.8 sle.ul (set less than or equal unsigned long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	111011

If `R<r2>` ≤ `R<r3>`, then set `R<r1> = 1`, otherwise then set `R<r1> = 0`. Operands are compared as unsigned integer quantities.

6.8.9 sgt.ul (set greater than unsigned long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	111100

If `R<r2>` > `R<r3>`, then set `R<r1> = 1`, otherwise then set `R<r1> = 0`. Operands are compared as unsigned integer quantities.

6.8.10 sge.ul (set greater than or equal unsigned long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01001	111101

If `R<r2>` ≥ `R<r3>`, then set `R<r1> = 1`, otherwise then set `R<r1> = 0`. Operands are compared as unsigned integer quantities.

6.8.11 seqi.l (set equal immediate long word)

Bits	31 26	25 21	20 16	15 0
	010000	r1	r2	immediate

If `R<r2>` = `immediate`, then set `R<r1> = 1`, otherwise then set `R<r1> = 0`. Operands are compared as signed integer quantities. The `immediate` operand is sign-extended to 64 bits before being compared.

6.8.12 snei.l (set not equal immediate long word)

Bits	31 26	25 21	20 16	15 0
	010001	r1	r2	immediate

If `R<r2>` ≠ `immediate`, then set `R<r1> = 1`, otherwise then set `R<r1> = 0`. Operands are compared as signed integer quantities. The `immediate` operand is sign-extended to 64 bits before being compared.

6.8.13 slti.l (set less than immediate long word)

Bits	31 26	25 21	20 16	15 0
	010010	r1	r2	immediate

If `R<r2>` < `immediate`, then set `R<r1> = 1`, otherwise then set `R<r1> = 0`. Operands are compared as signed integer quantities. The `immediate` operand is sign-extended to 64 bits before being compared.

6.8.14 slei.l (set less than or equal immediate long word)

Bits	31 26	25 21	20 16	15 0
	010011	r1	r2	immediate

If `R<r2>` ≤ `immediate`, then set `R<r1> = 1`, otherwise then set `R<r1> = 0`. Operands are compared as signed integer quantities. The `immediate` operand is sign-extended to 64 bits before being compared.

6.8.15 sgti.l (set greater than immediate long word)

Bits	31 26	25 21	20 16	15 0
	010100	r1	r2	immediate

If `R<r2>` > `immediate`, then set `R<r1> = 1`, otherwise then set `R<r1> = 0`. Operands are compared as signed integer quantities. The `immediate` operand is sign-extended to 64 bits before being compared.

6.8.16 sgei.l (set greater than or equal immediate long word)

Bits	31 26	25 21	20 16	15 0
	010101	r1	r2	immediate

If `R<r2>` ≥ `immediate`, then set `R<r1> = 1`, otherwise then set `R<r1> = 0`. Operands are compared as signed integer quantities. The `immediate` operand is sign-extended to 64 bits before being compared.

6.8.17 slti.ul (set less than immediate unsigned long word)

Bits	31 26	25 21	20 16	15 0
	010110	r1	r2	immediate

If `R<r2>` < `immediate`, then set `R<r1> = 1`, otherwise then set `R<r1> = 0`. Operands are compared as unsigned integer quantities. The `immediate` operand is zero-extended to 64 bits before being compared.

6.8.18 slei.ul (set less than or equal immediate unsigned long word)

Bits	31 26	25 21	20 16	15 0
	010111	r1	r2	immediate

If `R<r2>` ≤ `immediate`, then set `R<r1> = 1`, otherwise then set `R<r1> = 0`. Operands are compared as unsigned integer quantities. The `immediate` operand is zero-extended to 64 bits before being compared.

6.8.19 sgti.ul (set greater than immediate unsigned long word)

Bits	31 26	25 21	20 16	15 0
	011110	r1	r2	immediate

If `R<r2>` > `immediate`, then set `R<r1> = 1`, otherwise then set `R<r1> = 0`. Operands are compared as unsigned integer quantities. The `immediate` operand is zero-extended to 64 bits before being compared.

6.8.20 sgei.ul (set greater than or equal immediate unsigned long word)

Bits	31 26	25 21	20 16	15 0
	011111	r1	r2	immediate

If `R<r2>` ≥ `immediate`, then set `R<r1> = 1`, otherwise then set `R<r1> = 0`. Operands are compared as unsigned integer quantities. The `immediate` operand is zero-extended to 64 bits before being compared.

6.9 Load/store instructions

6.9.1 l.b (load byte)

Bits	31 26	25 21	20 16	15 0
	100000	r1	r2	immediate

Loads the byte from the memory address `addr` into `R<r1>`. The byte is sign-extended to 64 bits in the process of loading.

6.9.2 l.ub (load unsigned byte)

Bits	31 26	25 21	20 16	15 0
	100001	r1	r2	immediate

Loads the byte from the memory address `addr` into `R<r1>`. The byte is zero-extended to 64 bits in the process of loading.

6.9.3 l.h (load half-word)

Bits	31 26	25 21	20 16	15 0
	100010	r1	r2	immediate

Loads the 2-byte integer value from the memory address `addr` into `R<r1>`. The value is sign-extended to 64 bits in the process of loading.

Depending on the contents of the bit B of $state, the value can be loaded in either big-endian or little-endian format.

6.9.4 l.uh (load unsigned half-word)

Bits	31 26	25 21	20 16	15 0
	100011	r1	r2	immediate

Loads the 2-byte integer value from the memory address addr into R<r1>. The value is zero-extended to 64 bits in the process of loading.

Depending on the contents of the bit B of $state, the value can be loaded in either big-endian or little-endian format.

6.9.5 l.w (load word)

Bits	31 26	25 21	20 16	15 0
	100100	r1	r2	immediate

Loads the 4-byte integer value from the memory address addr into R<r1>. The value is sign-extended to 64 bits in the process of loading.

Depending on the contents of the bit B of $state, the value can be loaded in either big-endian or little-endian format.

6.9.6 l.uw (load unsigned word)

Bits	31 26	25 21	20 16	15 0
	100101	r1	r2	immediate

Loads the 4-byte integer value from the memory address addr into R<r1>. The value is zero-extended to 64 bits in the process of loading.

Depending on the contents of the bit B of $state, the value can be loaded in either big-endian or little-endian format.

6.9.7 l.l (load long word)

Bits	31 26	25 21	20 16	15 0
	100110	r1	r2	immediate

Loads the 8-byte integer value from the memory address addr into R<r1>.

Depending on the contents of the bit B of $state, the value can be loaded in either big-endian or little-endian format.

6.9.8 xchg (exchange)

Bits	31 26	25 21	20 16	15 0
	100111	r1	r2	immediate

Exchanges the 8-byte integer value from the memory address `addr` with `R<r1>`.

Depending on the contents of the bit `B` of `$state`, the value can be moved in either big-endian or little-endian format.

The memory bus is locked for the duration of operation; so the exchange is executed as an atomic action.

6.9.9 s.b (store byte)

Bits	31 26	25 21	20 16	15 0
	101000	r1	r2	immediate

Stores the lower byte of `R<r1>` into the memory at address `addr`.

6.9.10 s.h (store half-word)

Bits	31 26	25 21	20 16	15 0
	101001	r1	r2	immediate

Stores the lower 2 bytes of `R<r1>` into the memory at address `addr`.

Depending on the contents of the bit `B` of `$state`, the value can be stored in either big-endian or little-endian format.

6.9.11 s.w (store word)

Bits	31 26	25 21	20 16	15 0
	101010	r1	r2	immediate

Stores the lower 4 bytes of `R<r1>` into the memory at address `addr`.

Depending on the contents of the bit `B` of `$state`, the value can be stored in either big-endian or little-endian format.

6.9.12 s.l (store long word)

Bits	31 26	25 21	20 16	15 0
	101011	r1	r2	immediate

Stores the `R<r1>` into the memory at address `addr`.

Depending on the contents of the bit B of $state, the value can be stored in either big-endian or little-endian format.

6.9.13 lir (Load IP-relative)

Bits	31 26	25 21	20 0
	001001	r1	immediate

Calculates the address $ip + (immediate << 2) (immediate is sign-extended before the shift) and loads the 8-byte value at this address into <r1>. The address must be a multiple of 8.

In Virtual mode, this instruction requires EXECUTE rather than READ permission for the memory area where the value is loaded from. This reflects upon the main purpose of this instruction – loading 64-bit constants; specifically addresses of static symbols.

6.9.14 ldm (Load Multiple)

Bits	31 26	25 21	20 0
	011100	r1	immediate

Loads up to 21 registers from memory at consecutive addresses starting from the address R<r1> and incrementing by 8 for each loaded register. Depending on the contents of the bit B of $state, register values can be loaded in either big-endian or little-endian format.

The 21 bits of the immediate operand each correspond to one general purpose register in the following order (from lower to higher bits):

- $a0..$a3 (bits 0..3)
- $s0..$s12 (bits 4..16)
- $gp (bit 17)
- $fp (bit 18)
- $dp (bit 19)
- $ra (bit 20)

Registers are loaded from memory starting from those corresponding to lower bits of the register mask. If all registers are loaded successfully, R<r1> is incremented by off.

The main purpose of this instruction is to save registers into an activation stack upon procedure entry. However, it can also be used to temporarily save multiple registers within a procedure body if the base register other than $sp is used.

Causes an OPERAND exception if r1 designates one of the loaded registers or $ip.

6.9.15 stm (Store Multiple)

Bits	31 26	25 21	20 0
	011101	r1	immediate

Stores up to 21 registers into memory at consecutive addresses starting from the address `R<r1> - off` (where `off = 8 * <number of registers stored>`) and incrementing by 8 for each stored register. Depending on the contents of the bit `B` of `$state`, register values can be stored in either big-endian or little-endian format.

The 21 bits of the `immediate` operand each correspond to one general purpose register in the following order (from lower to higher bits):

- `$a0`..`$a3` (bits 0..3)
- `$s0`..`$s12` (bits 4..16)
- `$gp` (bit 17)
- `$fp` (bit 18)
- `$dp` (bit 19)
- `$ra` (bit 20)

Registers are stored to memory starting from those corresponding to lower bits of the register mask. If all registers are stored successfully, `R<r1>` is decremented by `off`.

The main purpose of this instruction is to save registers into an activation stack upon procedure entry. However, it can also be used to temporarily save multiple registers within a procedure body if the base register other than `$sp` is used.

Causes an `OPERAND` exception if `r1` designates one of the stored registers or `$ip`.

6.10 Flow control instructions

6.10.1 j (Jump)

Bits	31 26	25 0
	000111	target

Sets `$ip = jaddr`.

6.10.2 jal (Jump And Link)

Bits	31 26	25 0
	001111	target

Sets `$ra = $ip`; then sets `$ip = jaddr`.

6.10.3 jr (Jump Register)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	00000	00000	01001	111000

Set `$ip` = `R<r1>`.

Causes an `OPCODE` exception if bits 11..20 of the instructions are not 0.

6.10.4 jalr (Jump And Link Register)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	00000	00000	01001	111001

Sets `$ra = $ip`; Set `$ip` = `R<r1>`.

Causes an `OPCODE` exception if bits 11..20 of the instructions are not 0.

6.10.5 beq.l (branch on equal long word)

Bits	31 26	25 21	20 16	15 0
	110000	r1	r2	immediate

If `R<r1> = R<r2>`, then sets `$ip = baddr`; otherwise has no effect.

6.10.6 bne.l (branch on not equal long word)

Bits	31 26	25 21	20 16	15 0
	110001	r1	r2	immediate

If `R<r1> ≠ R<r2>`, then sets `$ip = baddr`; otherwise has no effect.

6.10.7 blt.l (branch on less than long word)

Bits	31 26	25 21	20 16	15 0
	110010	r1	r2	immediate

If `R<r1> < R<r2>`, then sets `$ip = baddr`; otherwise has no effect. Operands are compared as signed integer quantities.

6.10.8 ble.l (branch on less than or equal long word)

Bits	31 26	25 21	20 16	15 0
	110011	r1	r2	immediate

If `R<r1> ≤ R<r2>`, then sets `$ip = baddr`; otherwise has no effect. Operands are compared as signed integer quantities.

6.10.9 bgt.l (branch on greater than long word)

Bits	31 26	25 21	20 16	15 0
	110100	r1	r2	immediate

If `R<r1> > R<r2>`, then sets `$ip = baddr`; otherwise has no effect. Operands are compared as signed integer quantities.

6.10.10 bge.l (branch on greater than or equal long word)

Bits	31 26	25 21	20 16	15 0
	110101	r1	r2	immediate

If `R<r1> ≥ R<r2>`, then sets `$ip = baddr`; otherwise has no effect. Operands are compared as signed integer quantities.

6.10.11 blt.ul (branch on less than unsigned long word)

Bits	31 26	25 21	20 16	15 0
	110110	r1	r2	immediate

If `R<r1> < R<r2>`, then sets `$ip = baddr`; otherwise has no effect. Operands are compared as unsigned integer quantities.

6.10.12 ble.ul (branch on less than or equal unsigned long word)

Bits	31 26	25 21	20 16	15 0
	110111	r1	r2	immediate

If `R<r1> ≤ R<r2>`, then sets `$ip = baddr`; otherwise has no effect. Operands are compared as unsigned integer quantities.

6.10.13 bgt.ul (branch on greater than unsigned long word)

Bits	31 26	25 21	20 16	15 0
	111110	r1	r2	immediate

If `R<r1> > R<r2>`, then sets `$ip = baddr`; otherwise has no effect. Operands are compared as unsigned integer quantities.

6.10.14 bge.ul (branch on greater than or equal unsigned long word)

Bits	31 26	25 21	20 16	15 0
	111111	r1	r2	immediate

If `R<r1> ≥ R<r2>`, then sets `$ip = baddr`; otherwise has no effect. Operands are compared as unsigned integer quantities.

6.10.15 beqi.l (branch on equal immediate long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	imm5	10000	00001	imm6

If `R<r1> = imm5`, then sets `$ip = baddr`; otherwise has no effect. The `imm5` immediate operand is treated as a signed integer, which is sign-extended to 64 bits before comparison. The `imm6` field specifies a signed 6-bit branch displacement; the branch target is calculated as `$ip + (imm6 << 2).`

6.10.16 bnei.l (branch on not equal immediate long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	imm5	10001	00001	imm6

If `R<r1> ≠ imm5`, then sets `$ip = baddr`; otherwise has no effect. The `imm5` immediate operand is treated as a signed integer, which is sign-extended to 64 bits before comparison. The `imm6` field specifies a signed 6-bit branch displacement; the branch target is calculated as `$ip + (imm6 << 2).`

6.10.17 blti.l (branch on less than immediate long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	imm5	10010	00001	imm6

If `R<r1> < imm5`, then sets `$ip = baddr`; otherwise has no effect. The `imm5` immediate operand is treated as a signed integer, which is sign-extended to 64 bits before comparison. The `imm6` field specifies a signed 6-bit branch displacement; the branch target is calculated as `$ip + (imm6 << 2).`

6.10.18 blei.l (branch on less than or equal immediate long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	imm5	10011	00001	imm6

If `R<r1> ≤ imm5`, then sets `$ip = baddr`; otherwise has no effect. The `imm5` immediate operand is treated as a signed integer, which is sign-extended to 64 bits before comparison. The `imm6` field specifies a signed 6-bit branch displacement; the branch target is calculated as `$ip + (imm6 << 2)`.

6.10.19 bgti.l (branch on greater than immediate long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	imm5	10100	00001	imm6

If `R<r1> > imm5`, then sets `$ip = baddr`; otherwise has no effect. The `imm5` immediate operand is treated as a signed integer, which is sign-extended to 64 bits before comparison. The `imm6` field specifies a signed 6-bit branch displacement; the branch target is calculated as `$ip + (imm6 << 2)`.

6.10.20 bgei.l (branch on greater than or equal immediate long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	imm5	10101	00001	imm6

If `R<r1> ≥ imm5`, then sets `$ip = baddr`; otherwise has no effect. The `imm5` immediate operand is treated as a signed integer, which is sign-extended to 64 bits before comparison. The `imm6` field specifies a signed 6-bit branch displacement; the branch target is calculated as `$ip + (imm6 << 2)`.

6.10.21 blti.ul (branch on less than immediate unsigned long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	imm5	11010	00001	imm6

If `R<r1> < imm5`, then sets `$ip = baddr`; otherwise has no effect. The `imm5` immediate operand is treated as an unsigned integer, which is zero-extended to 64 bits before comparison, operands are then compared as unsigned values. The `imm6` field specifies a signed 6-bit branch displacement; the branch target is calculated as `$ip + (imm6 << 2)`.

6.10.22 blei.ul (branch on less than or equal immediate unsigned long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	imm5	11011	00001	imm6

If `R<r1> ≤ imm5`, then sets `$ip = baddr`; otherwise has no effect. The `imm5` immediate operand is treated as an unsigned integer, which is zero-extended to 64 bits before comparison, operands are then compared as unsigned values. The `imm6` field specifies a signed 6-bit branch displacement; the branch target is calculated as `$ip + (imm6 << 2)`.

6.10.23 bgti.ul (branch on greater than immediate unsigned long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	imm5	11100	00001	imm6

If `R<r1> > imm5`, then sets `$ip = baddr`; otherwise has no effect. The `imm5` immediate operand is treated as an unsigned integer, which is zero-extended 64 bits before comparison, operands are then compared as unsigned values. The `imm6` field specifies a signed 6-bit branch displacement; the branch target is calculated as `$ip + (imm6 << 2)`.

6.10.24 bgei.ul (branch on greater than or equal immediate unsigned long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	imm5	11101	00001	imm6

If `R<r1> ≥ imm5`, then sets `$ip = baddr`; otherwise has no effect. The `imm5` immediate operand is treated as an unsigned integer, which is zero-extended to 64 bits before comparison, operands are then compared as unsigned values. The `imm6` field specifies a signed 6-bit branch displacement; the branch target is calculated as `$ip + (imm6 << 2)`.

6.11 TLB control instructions

6.11.1 iitlb (invalidate Instruction TLB) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	00000	00000	00000	01000	001000

Invalidates all instruction TLB entries. If the processor has a single TLB for both instruction and data, invalidating instruction TLB implicitly invalidates data TLB.

Causes an `OPCODE` exception if bits 11..25 of the instructions are not 0.

6.11.2 idtlb (invalidate Data TLB) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	00000	00000	00000	01000	001001

Invalidates all data TLB entries. If the processor has a single TLB for both instruction and data, invalidating data TLB implicitly invalidates instruction TLB.

Causes an `OPCODE` exception if bits 11..25 of the instructions are not 0.

6.11.3 iitlbe (invalidate Instruction TLB entry) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	00000	00000	01000	001010

If an instruction TLB has an entry such that the address `R<r1>` belongs to the virtual page described by the entry, that entry is invalidated; otherwise the instruction has no effect.

If the processor has a single TLB for both instruction and data, invalidating instruction TLB entry implicitly invalidates data TLB entry.

Causes an `OPCODE` exception if bits 11..20 of the instructions are not 0.

6.11.4 idtlbe (invalidate data TLB entry) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	00000	00000	01000	001011

If a data TLB has an entry such that the address `R<r1>` belongs to the virtual page described by the entry, that entry is invalidated; otherwise the instruction has no effect.

If the processor has a single TLB for both instruction and data, invalidating data TLB entry implicitly invalidates instruction TLB entry.

Causes an `OPCODE` exception if bits 11..20 of the instructions are not 0.

6.11.5 iitlbc (invalidate Instruction TLB for current process) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	00000	00000	00000	01000	001100

Invalidates all instruction TLB entries which are active (bit `A` set to 1) and whose `CID` matches the `CID` of the `$state` register. If the processor has a single TLB for both instruction and data, invalidating instruction TLB implicitly invalidates data TLB.

Causes an `OPCODE` exception if bits 11..25 of the instructions are not 0.

6.11.6 idtlbc (invalidate Data TLB for current process) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	00000	00000	00000	01000	001101

Invalidates all data TLB entries which are active (bit `A` set to 1) and whose `CID` matches the `CID` of the `$state` register. If the processor has a single TLB for both instruction and data, invalidating data TLB implicitly invalidates instruction TLB.

Causes an `OPCODE` exception if bits 11..25 of the instructions are not 0.

6.11.7 iitlbec (invalidate Instruction TLB entry for current process) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	00000	00000	01000	001110

If an instruction TLB has an entry such that the address `R<r1>` belongs to the virtual page described by the entry and whose `CID` matches the `CID` of the `$state` register, that entry is invalidated; otherwise the instruction has no effect.

If the processor has a single TLB for both instruction and data, invalidating instruction TLB entry implicitly invalidates data TLB entry.

Causes an `OPCODE` exception if bits 11..20 of the instructions are not 0.

6.11.8 idtlbec (invalidate data TLB entry for current process) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	00000	00000	01000	001111

If a data TLB has an entry such that the address `R<r1>` belongs to the virtual page described by the entry and whose `CID` matches the `CID` of the `$state` register, that entry is invalidated; otherwise the instruction has no effect.

If the processor has a single TLB for both instruction and data, invalidating data TLB entry implicitly invalidates instruction TLB entry.

Causes an `OPCODE` exception if bits 11..20 of the instructions are not 0.

6.12 Cache control instructions

6.12.1 imb (Instruction Memory Barrier) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	00000	00000	00000	01000	010000

This instruction ensures that subsequent instructions are fetched from memory. Specifically:

- The instruction cache is invalidated. If a single cache is used for both instruction and data, flushing the instruction cache implicitly causes data cache to be flushed as well, which, in turn, may cause some deferred data to be written to memory.
- The instruction prefetch buffer is cleared.
- The instruction TLB is invalidated. If the processor has a single TLB for both instruction and data, invalidating instruction TLB implicitly invalidates data TLB.

Causes an `OPCODE` exception if bits 11..25 of the instructions are not 0.

6.12.2 dmb (Data Memory Barrier) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	00000	00000	00000	01000	010001

This instruction ensures that all memory accesses performed up to this point have had their effects. Specifically:

- The data cache is flushed and invalidated. This may cause some deferred data to be written to memory. If a single cache is used for both instruction and data, flushing the data cache implicitly causes instruction cache to be flushed as well.
- The data TLB is invalidated. If the processor has a single TLB for both instruction and data, invalidating data TLB implicitly invalidates instruction TLB.

Causes an `OPCODE` exception if bits 11..25 of the instructions are not 0.

6.12.3 imbc (Instruction Memory Barrier for current process) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	00000	00000	00000	01000	010100

This instruction ensures that subsequent instructions for the current process are fetched from memory. Specifically:

- All cache lines of the instruction cache that can be used by the current process are invalidated. If a single cache is used for both instruction and data, flushing instruction cache entries implicitly causes data cache entries to be flushed as well, which, in turn, may cause some deferred data to be written to memory.
- The instruction prefetch buffer is cleared.
- All instruction TLB entries that can be used by the current process are invalidated. If the processor has a single TLB for both instructions and data, invalidating instruction TLB entries implicitly invalidates data TLB entries.

Causes an `OPCODE` exception if bits 11..25 of the instructions are not 0.

6.12.4 dmbc (Data Memory Barrier for current process) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	00000	00000	00000	01000	010101

This instruction ensures that all memory accesses performed up to this point by the current process have had their effects. Specifically:

- All cache lines of the data cache that can be used by the current process are flushed. This may cause some deferred data to be written to memory. If a single cache is used for both instructions and data, flushing data cache lines implicitly causes instruction cache to be flushed as well.
- All data TLB entries that can be used by the current process are invalidated. If the processor has a single TLB for both instructions and data, invalidating data TLB entries implicitly invalidates instruction TLB entries.

Causes an `OPCODE` exception if bits 11..25 of the instructions are not 0.

6.13 I/O instructions

6.13.1 in.b (input byte) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01000	101000

Using the lower 16 bits of `R<r1>` as an I/O address, reads a byte from the specified I/O port, sign-extends it to 64 bits and places the result in `R<r2>`. If there is no I/O port at the specified I/O address, the value 00_{16} is always read immediately; otherwise the instruction stalls until a byte is received from the I/O port.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.13.2 in.ub (input unsigned byte) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01000	110000

Using the lower 16 bits of `R<r1>` as an I/O address, reads a byte from the specified I/O port, zero-extends it to 64 bits and places the result in `R<r2>`. If there is no I/O port at the specified I/O address, the value $\texttt{00}_{16}$ is always read immediately; otherwise the instruction stalls until a byte is received from the I/O port.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.13.3 in.h (input half-word) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01000	101001

Using the lower 16 bits of `R<r1>` as an I/O address, reads a half-word from the specified I/O port, sign-extends it to 64 bits and places the result in `R<r2>`. If there is no I/O port at the specified I/O address, the value $\texttt{0000}_{16}$ is always read immediately; otherwise the instruction stalls until a half-word is received from the I/O port.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.13.4 in.uh (input unsigned half-word) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01000	110001

Using the lower 16 bits of `R<r1>` as an I/O address, reads a half-word from the specified I/O port, zero-extends it to 64 bits and places the result in `R<r2>`. If there is no I/O port at the specified I/O address, the value $\texttt{0000}_{16}$ is always read immediately; otherwise the instruction stalls until a half-word is received from the I/O port.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.13.5 in.w (input word) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01000	101010

Using the lower 16 bits of `R<r1>` as an I/O address, reads a word from the specified I/O port, sign-extends it to 64 bits and places the result in `R<r2>`. If there is no I/O port at the specified I/O address, the value $\texttt{00000000}_{16}$ is always read immediately; otherwise the instruction stalls until a word is received from the I/O port.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.13.6 in.uw (input unsigned word) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01000	110010

Using the lower 16 bits of `R<r1>` as an I/O address, reads a word from the specified I/O port, zero-extends it to 64 bits and places the result in `R<r2>`. If there is no I/O port at the specified I/O address, the value 00000000_{16} is always read immediately; otherwise the instruction stalls until a word is received from the I/O port.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.13.7 in.l (input long word) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01000	101011

Using the lower 16 bits of `R<r1>` as an I/O address, reads a long word from the specified I/O port and places the result in `R<r2>`. If there is no I/O port at the specified I/O address, the value 0000000000000000_{16} is always read immediately; otherwise the instruction stalls until a long word is received from the I/O port.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.13.8 out.b (output byte) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01000	101100

Using the lower 16 bits of `R<r1>` as an I/O address, writes the lower byte of `R<r2>` to the specified I/O port. If there is no I/O port at the specified I/O address, the half-word written is lost; otherwise the processor stalls until the port acknowledges the receipt of the byte.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.13.9 out.h (output half-word) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01000	101101

Using the lower 16 bits of `R<r1>` as an I/O address, writes the lower 16 bits of `R<r2>` to the specified I/O port. If there is no I/O port at the specified I/O address, the value written there is lost; otherwise the processor stalls until the port acknowledges the receipt of the half-word.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.13.10 out.w (output word) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01000	101110

Using the lower 16 bits of `R<r1>` as an I/O address, writes the lower 32 bits of `R<r2>` to the specified I/O port. If there is no I/O port at the specified I/O address, the word written there is lost; otherwise the processor stalls until the port acknowledges the receipt of the word.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.13.11 out.l (output long word) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01000	101111

Using the lower 16 bits of `R<r1>` as an I/O address, writes the value of `R<r2>` to the specified I/O port. If there is no I/O port at the specified I/O address, the long word written there is lost; otherwise the processor stalls until the port acknowledges the receipt of the long word.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.13.12 tstp (Test Port status) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01000	100000

Using the lower 16 bits of `R<r1>` as an I/O address, tests the current state of the I/O port at that address. The contents of `R<r2>` is then modified as follows:

- Bit 0 (lowest) is set to 1 if there is an I/O port at the specified I/O address, or to 0 if there is no I/O port at the specified I/O address.
- Bit 1 is set to 1 if there is a pending interrupt in the I/O port at the specified I/O address, or to 0 if there is no pending interrupt there.
- Bit 2 is set to 1 if I/O interrupts are currently enabled in the I/O port at the specified address, or to 0 if they are disabled.
- Bits 3..15 and 32..63 are set to 0.
- If there is a pending interrupt in the port, bits 16..31 are set to the interrupt status code pending on the port and the interrupt is released (i.e. no longer pending on the I/O port). If there is no pending interrupt there, bits 16..31 are set to 0.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.13.13 setp (Set Port status) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	01000	100001

Using the lower 16 bits of `R<r1>` as an I/O address, sets the current state of the I/O port at that address. The contents of `R<r2>` is used to determine how the port state shall be set:

- If bit 1 is 1, the interrupt is made pending on the I/O port. Bits 16..31 of the `R<r2>` are used as an interrupt status code. If this bit is 0, the instruction has no effect on whether or not an interrupt is pending on the I/O port.
- If bit 2 is 1, I/O interrupts are enabled in the I/O port at the specified address. If this bit is 0, I/O interrupts are disabled.
- Bits 0, 3..15 and 32..63 are ignored.

Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

6.14 Miscellaneous instructions

6.14.1 halt (Halt processor) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	00000	00000	00000	01000	011001

This instruction causes the processor to halt immediately by setting the `W` bit of its `$state` register to 0. The halted processor executes no instructions, but can process enabled interrupts.

Causes an `OPCODE` exception if bits 11..25 of the instructions are not 0.

6.14.2 iret (Return from Interrupt Handler) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	inum	00000	00000	01000	011000

The instruction transfers control from the interrupt handler back to the interrupted code. The following steps are performed:

1. `$ip = $isaveip.<inum>` (restore instruction pointer)
2. `$state = $isavestate.<inum>` (restore processor state)

Causes an `OPCODE` exception if bits 11..20 of the instructions are not 0.
Causes `OPERAND` exception if `inum` is out of range `[0..5]`.

6.14.3 getfl (Get Flags)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	00000	00000	01100	111000

Sets `R<r1> = $flags`.

Causes an `OPCODE` exception if bits 11..20 of the instructions are not 0.

6.14.4 setfl (Set Flags)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	00000	00000	01100	111001

Sets `$flags = R<r1>`.

Causes an `OPCODE` exception if bits 11..20 of the instructions are not 0.

6.14.5 rstfl (Reset Flags)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	00000	00000	00000	01100	111010

Sets `$flags = 0x0000000000000000`.

Causes an `OPCODE` exception if bits 11..25 of the instructions are not 0.

6.14.6 svc (Supervisor Call)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	00000	00000	00000	01100	111101

Causes the `SVC` interrupt.

Causes an `OPCODE` exception if bits 11..25 of the instructions are not 0.

6.14.7 brk (break)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	brk	00000	00000	01100	111110

Causes the `PROGRAM` interrupt. The exception code stored in the `$isc.prg` is `0x0000000000000017` + ((`brk`) `<< 8`).

Causes an `OPCODE` exception if bits 11..20 of the instructions are not 0.

6.14.8 cpuid (CPU Identification)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	00000	00000	01000	011010

Stores the information about identification and features of the executing processor core into `R<r1>`. This information has the following format:

Bits	63 56	55 48
	processor	core

Bits	47	46	45 40	39 32
	P	C	AV	-

Bits	31 16
	-

Bits	15 7	6	5	4	3	2	1	0
	-	M	V	R	U	D	F	B

Individual fields have the following meanings:

- Processor – the 8-bit identifier of the processor to which the executing processor core belongs.
- Core – the 8-bit identifier of the calling core within its processor.
- AV (Architecture Version) – this 6-bit field contains a numeric identifier of the Cereon ISA version implemented by the processor. Currently this value can only be 1, which means Cereon ISA version 1 (as described in this document).
- P (primary Processor) – this bit is 1 if the processor to which the executing core belongs to a processor that is hardwired as a primary processor; 0 otherwise.
- C (primary Core) – this bit is 1 if the executing core is hardwired as a primary core within its processor; 0 otherwise.
- M (performance Monitoring) – when this bit is 1, the processor has a performance monitoring feature; otherwise the processor does not have it.
- V (Virtual memory) – when this bit is 1, the processor has a virtual memory feature; otherwise the processor does not have it. Note that virtual memory feature and protected memory feature are mutually exclusive – a processor can have none or either, but not both.
- R (pRotected memory) – when this bit is 1, the processor has a protected memory feature; otherwise the processor does not have it. Note that virtual memory feature and protected memory feature are mutually exclusive – a processor can have none or either, but not both.
- U (Unaligned operands) – when this bit is 1, the processor has the unaligned operands feature; otherwise the processor does not have it. Unaligned operands feature, when available, allows loading and storing multi-byte values from/to memory at an address that is not a multiple of the value's size. Note

that this feature, if present, does not apply to instructions, region tables or virtual page tables, which must always be naturally aligned.

- D (Debug) – when this bit is 1, the processor has the debug feature; otherwise the processor does not have it. Debug feature, when available, provides debug registers, instructions that examine and modify contents of debug registers, and exceptions caused by debug registers.
- F (Floating point) – when this bit is 1, the processor has the floating point feature; otherwise the processor does not have it. Floating point feature, when available, allows floating point loads, stores, compares and arithmetic.
- B (Base) – when this bit is 1, the processor implements the Base instruction set. In the current version (1) of the Cereon architecture, this bit is always 1.

Causes an `OPCODE` exception if bits 11..20 of the instructions are not 0.

6.14.9 sigp (Signal Processor) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	01000	011011

Using the lower 16 bits of `R<r1>` as a core ID, sends the external signal to a processor core with the specified ID. The type of the external interrupt is `SIGNAL`; the lower 32 bits of the `R<r2>` are used as an external interrupt subcode. The register `R<r3>` is set to one of the following values:

- `0` – if the external signal has been accepted by the destination core. The accepted signal may not necessarily cause an immediate `EXTERNAL` interrupt in the destination core; however, one is guaranteed to occur eventually.
- `1` – if the destination core does not exist.
- `2` – if the destination core exists but cannot accept the external signal because another external signal, accepted previously, is already pending there.

Causes a `MASKED` exception if signalling to itself and the `EXTERNAL` interrupt is disabled.

6.14.10 nop (No Operation)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	00000	00000	00000	01001	000111

This instruction has no effect other than to occupy processor for one cycle.

Causes an `OPCODE` exception if bits 11..25 of the instructions are not 0.

7 The floating point feature

This section describes instructions provided as a part of the Cereon Floating point feature. This feature provides an ability to perform basic floating point operations in hardware, thus significantly increasing the speed of programs which require a large number of floating point calculations to be made.

7.1 Data movement instructions

7.1.1 mov.d (move double precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	000000

Sets `F<r1> = F<r2>`.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.

7.1.2 li.d (load immediate double precision floating point)

Bits	31 26	25 21	20 0
	001000	r1	immediate

Loads the reduced-precision floating point `immediate` value, extended to double precision format, into `F<r1>`.

Causes an `OPCODE` exception if the floating point feature is not installed.

7.2 Arithmetic instructions

7.2.1 add.f (add single precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	10000	010000

Sets `F<r1> = F<r2> + F<r3>`. Both operands are converted to single precision before operation, which is also carried out with single precision, yielding a single precision result that is then expanded to double precision before being stored.

Sets the `R` bit of `$flags` to 1 if either operand is a signalling NaN.
Sets the `E` bit of `$flags` to 1 if a floating point overflow occurs during the operation.
Sets the `U` bit of `$flags` to 1 if a floating point underflow occurs during the operation.
Sets the `I` bit of `$flags` to 1 if an inexact result is produced.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and either operand is a signalling NaN.
Causes an `FOVERFLOW` exception if an `E` bit of `$state` is 1 and a floating point overflow occurs during the operation.
Causes an `FUNDERFLOW` exception if an `U` bit of `$state` is 1 and a floating point underflow occurs during the operation.
Causes an `FINEXACT` exception if an `I` bit of `$state` is 1 and an inexact result is produced.

7.2.2 add.d (add double precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	10000	001000

Sets `F<r1> = F<r2> + F<r3>`.

Sets the `R` bit of `$flags` to 1 if either operand is a signalling NaN.
Sets the `E` bit of `$flags` to 1 if a floating point overflow occurs during the operation.
Sets the `U` bit of `$flags` to 1 if a floating point underflow occurs during the operation.
Sets the `I` bit of `$flags` to 1 if an inexact result is produced.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and either operand is a signalling NaN.
Causes an `FOVERFLOW` exception if an `E` bit of `$state` is 1 and a floating point overflow occurs during the operation.
Causes an `FUNDERFLOW` exception if an `U` bit of `$state` is 1 and a floating point underflow occurs during the operation.
Causes an `FINEXACT` exception if an `I` bit of `$state` is 1 and an inexact result is produced.

7.2.3 sub.f (subtract single precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	10000	010001

Sets `F<r1> = F<r2> - F<r3>`. Both operands are converted to single precision before operation, which is also carried out with single precision, yielding a single precision result that is then expanded to double precision before being stored.

Sets the `R` bit of `$flags` to 1 if either operand is a signalling NaN.
Sets the `E` bit of `$flags` to 1 if a floating point overflow occurs during the operation.
Sets the `U` bit of `$flags` to 1 if a floating point underflow occurs during the operation.
Sets the `I` bit of `$flags` to 1 if an inexact result is produced.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and either operand is a signalling NaN.
Causes an `FOVERFLOW` exception if an `E` bit of `$state` is 1 and a floating point overflow occurs during the operation.
Causes an `FUNDERFLOW` exception if an `U` bit of `$state` is 1 and a floating point underflow occurs during the operation.
Causes an `FINEXACT` exception if an `I` bit of `$state` is 1 and an inexact result is produced.

7.2.4 sub.d (subtract double precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	10000	001001

Sets `F<r1> = F<r2> - F<r3>`.

Sets the `R` bit of `$flags` to 1 if either operand is a signalling NaN.
Sets the `E` bit of `$flags` to 1 if a floating point overflow occurs during the operation.
Sets the `U` bit of `$flags` to 1 if a floating point underflow occurs during the operation.
Sets the `I` bit of `$flags` to 1 if an inexact result is produced.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and either operand is a signalling NaN.
Causes an `FOVERFLOW` exception if an `E` bit of `$state` is 1 and a floating point overflow occurs during the operation.
Causes an `FUNDERFLOW` exception if an `U` bit of `$state` is 1 and a floating point underflow occurs during the operation.
Causes an `FINEXACT` exception if an `I` bit of `$state` is 1 and an inexact result is produced.

7.2.5 mul.f (multiply single precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	10000	010010

Sets `F<r1> = F<r2> * F<r3>`. Both operands are converted to single precision before operation, which is also carried out with single precision, yielding a single precision result that is then expanded to double precision before being stored.

Sets the `R` bit of `$flags` to 1 if either operand is a signalling NaN.
Sets the `E` bit of `$flags` to 1 if a floating point overflow occurs during the operation.
Sets the `U` bit of `$flags` to 1 if a floating point underflow occurs during the operation.

Sets the I bit of $flags to 1 if an inexact result is produced.

Causes an OPCODE exception if the floating point feature is not available.
Causes an FOPERAND exception if an R bit of $state is 1 and either operand is a signalling NaN.
Causes an FOVERFLOW exception if an E bit of $state is 1 and a floating point overflow occurs during the operation.
Causes an FUNDERFLOW exception if an U bit of $state is 1 and a floating point underflow occurs during the operation.
Causes an FINEXACT exception if an I bit of $state is 1 and an inexact result is produced.

7.2.6 mul.d (multiply double precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	10000	001010

Sets F<r1> = F<r2> * F<r3>.

Sets the R bit of $flags to 1 if either operand is a signalling NaN.
Sets the E bit of $flags to 1 if a floating point overflow occurs during the operation.
Sets the U bit of $flags to 1 if a floating point underflow occurs during the operation.
Sets the I bit of $flags to 1 if an inexact result is produced.

Causes an OPCODE exception if the floating point feature is not available.
Causes an FOPERAND exception if an R bit of $state is 1 and either operand is a signalling NaN.
Causes an FOVERFLOW exception if an E bit of $state is 1 and a floating point overflow occurs during the operation.
Causes an FUNDERFLOW exception if an U bit of $state is 1 and a floating point underflow occurs during the operation.
Causes an FINEXACT exception if an I bit of $state is 1 and an inexact result is produced.

7.2.7 div.f (divide single precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	10000	010011

Sets F<r1> = F<r2> / F<r3>. Both operands are converted to single precision before operation, which is also carried out with single precision, yielding a single precision result that is then expanded to double precision before being stored.

Sets the Z bit of $flags to 1 if F<r3> is 0.
Sets the R bit of $flags to 1 if either operand is a signalling NaN.
Sets the E bit of $flags to 1 if a floating point overflow occurs during the operation.

Sets the `U` bit of `$flags` to 1 if a floating point underflow occurs during the operation.
Sets the `I` bit of `$flags` to 1 if an inexact result is produced.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes a `FZDIV` exception if a `Z` bit of `$state` is 1 and `F<r3>` is 0.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and either operand is a signalling NaN.
Causes an `FOVERFLOW` exception if an `E` bit of `$state` is 1 and a floating point overflow occurs during the operation.
Causes an `FUNDERFLOW` exception if an `U` bit of `$state` is 1 and a floating point underflow occurs during the operation.
Causes an `FINEXACT` exception if an `I` bit of `$state` is 1 and an inexact result is produced.

7.2.8 div.d (divide double precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	10000	001011

Sets `F<r1> = F<r2> / F<r3>`.

Sets the `Z` bit of `$flags` to 1 if `F<r3>` is 0.
Sets the `R` bit of `$flags` to 1 if either operand is a signalling NaN.
Sets the `E` bit of `$flags` to 1 if a floating point overflow occurs during the operation.
Sets the `U` bit of `$flags` to 1 if a floating point underflow occurs during the operation.
Sets the `I` bit of `$flags` to 1 if an inexact result is produced.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes a `FZDIV` exception if a `Z` bit of `$state` is 1 and `F<r3>` is 0.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and either operand is a signalling NaN.
Causes an `FOVERFLOW` exception if an `E` bit of `$state` is 1 and a floating point overflow occurs during the operation.
Causes an `FUNDERFLOW` exception if an `U` bit of `$state` is 1 and a floating point underflow occurs during the operation.
Causes an `FINEXACT` exception if an `I` bit of `$state` is 1 and an inexact result is produced.

7.2.9 abs.f (absolute value single precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	010100

Set `F<r1> = |F<r2>|`. Operand is converted to single precision before operation, which is also carried out with single precision, yielding a single precision result that is then expanded to double precision before being stored.

Sets the `R` bit of `$flags` to 1 if either operand is a signalling NaN.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and an operand is a signalling NaN.

7.2.10 abs.d (absolute value double precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	001100

Set `F<r1> = |F<r2>|`.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and an operand is a signalling NaN.

7.2.11 neg.f (negate single precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	010101

Set `F<r1> = -F<r2>`. Operand is converted to single precision before operation, which is also carried out with single precision, yielding a single precision result that is then expanded to double precision before being stored.

Sets the `R` bit of `$flags` to 1 if either operand is a signalling NaN.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and an operand is a signalling NaN.

7.2.12 neg.d (negate double precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	001101

Set `F<r1> = -F<r2>`.

Sets the `R` bit of `$flags` to 1 if either operand is a signalling NaN.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and an operand is a signalling NaN.

7.2.13 sqrt.f (square root single precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	010110

Set `F<r1> = sqrt(F<r2>)`. Operand is converted to single precision before operation, which is also carried out with single precision, yielding a single precision result that is then expanded to double precision before being stored.

Sets the `R` bit of `$flags` to 1 if an operand is a signalling NaN or negative.
Sets the `I` bit of `$flags` to 1 if an inexact result is produced.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.
Causes an `FOPERAND` exception if an operand is negative.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and an operand is a signalling NaN.
Causes an `FINEXACT` exception if an `I` bit of `$state` is 1 and an inexact result is produced.

7.2.14 sqrt.d (square root double precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	001110

Set `F<r1> = sqrt(F<r2>).`

Sets the `R` bit of `$flags` to 1 if an operand is a signalling NaN or negative.
Sets the `I` bit of `$flags` to 1 if an inexact result is produced.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.
Causes an `FOPERAND` exception if an operand is negative.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and an operand is a signalling NaN.
Causes an `FINEXACT` exception if an `I` bit of `$state` is 1 and an inexact result is produced.

7.3 Data type conversion instructions

7.3.1 cvt.fb (Convert single precision floating point to byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	100000

Set R<r1> = F<r2>. The operand is converted to single precision before being further converted to an integer. The result is a 8-bit signed integer, which is then sign-extended to 64 bits. If the operand cannot be represented as an 8-bit signed integer value exactly, the closest representable 8-bit signed integer value results.

Sets the R bit of $flags to 1 if an operand is a NaN, whether signalling or quiet.
Sets the E bit of $flags to 1 if the integer part of the floating point value is too large to be represented as an integer exactly.

Causes an OPCODE exception if the floating point feature is not available.
Causes an FOPERAND exception if an R bit of $state is 1 and an operand is a NaN, whether signalling or quiet.
Causes an FOVERFLOW exception if an E bit of $state is 1 and the integer part of the floating point value is too large to be represented as an integer exactly.

7.3.2 cvt.fub (Convert single precision floating point to unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	100100

Set R<r1> = F<r2>. The operand is converted to single precision before being further converted to an integer. The result is a 8-bit unsigned integer, which is then zero-extended to 64 bits. If the operand cannot be represented as an 8-bit unsigned integer value exactly, the closest representable 8-bit unsigned integer value results.

Sets the R bit of $flags to 1 if an operand is a NaN, whether signalling or quiet.
Sets the E bit of $flags to 1 if the integer part of the floating point value is too large to be represented as an integer exactly.

Causes an OPCODE exception if the floating point feature is not available.
Causes an FOPERAND exception if an R bit of $state is 1 and an operand is a NaN, whether signalling or quiet.
Causes an FOVERFLOW exception if an E bit of $state is 1 and the integer part of the floating point value is too large to be represented as an integer exactly.

7.3.3 cvt.fh (Convert single precision floating point to half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	100001

Set `R<r1>` = `F<r2>`. The operand is converted to single precision before being further converted to an integer. The result is a 16-bit signed integer, which is then sign-extended to 64 bits. If the operand cannot be represented as a 16-bit signed integer value exactly, the closest representable 16-bit signed integer value results.

Sets the `R` bit of `$flags` to 1 if an operand is a NaN, whether signalling or quiet.
Sets the `E` bit of `$flags` to 1 if the integer part of the floating point value is too large to be represented as an integer exactly.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and an operand is a NaN, whether signalling or quiet.
Causes an `FOVERFLOW` exception if an `E` bit of `$state` is 1 and the integer part of the floating point value is too large to be represented as an integer exactly.

7.3.4 cvt.fuh (Convert single precision floating point to unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	100101

Set `R<r1>` = `F<r2>`. The operand is converted to single precision before being further converted to an integer. The result is a 16-bit unsigned integer, which is then zero-extended to 64 bits. If the operand cannot be represented as a 16-bit unsigned integer value exactly, the closest representable 16-bit unsigned integer value results.

Sets the `R` bit of `$flags` to 1 if an operand is a NaN, whether signalling or quiet.
Sets the `E` bit of `$flags` to 1 if the integer part of the floating point value is too large to be represented as an integer exactly.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and an operand is a NaN, whether signalling or quiet.
Causes an `FOVERFLOW` exception if an `E` bit of `$state` is 1 and the integer part of the floating point value is too large to be represented as an integer exactly.

7.3.5 cvt.fw (Convert single precision floating point to word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	100010

Set R<r1> = F<r2>. The operand is converted to single precision before being further converted to an integer. The result is a 32-bit signed integer, which is then sign-extended to 64 bits. If the operand cannot be represented as a 32-bit signed integer value exactly, the closest representable 32-bit signed integer value results.

Sets the R bit of $flags to 1 if an operand is a NaN, whether signalling or quiet.
Sets the E bit of $flags to 1 if the integer part of the floating point value is too large to be represented as an integer exactly.

Causes an OPCODE exception if the floating point feature is not available.
Causes an FOPERAND exception if an R bit of $state is 1 and an operand is a NaN, whether signalling or quiet.
Causes an FOVERFLOW exception if an E bit of $state is 1 and the integer part of the floating point value is too large to be represented as an integer exactly.

7.3.6 cvt.fuw (Convert single precision floating point to unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	100110

Set R<r1> = F<r2>. The operand is converted to single precision before being further converted to an integer. The result is a 32-bit unsigned integer, which is then zero-extended to 64 bits. If the operand cannot be represented as a 32-bit unsigned integer value exactly, the closest representable 32-bit unsigned integer value results.

Sets the R bit of $flags to 1 if an operand is a NaN, whether signalling or quiet.
Sets the E bit of $flags to 1 if the integer part of the floating point value is too large to be represented as an integer exactly.

Causes an OPCODE exception if the floating point feature is not available.
Causes an FOPERAND exception if an R bit of $state is 1 and an operand is a NaN, whether signalling or quiet.
Causes an FOVERFLOW exception if an E bit of $state is 1 and the integer part of the floating point value is too large to be represented as an integer exactly.

7.3.7 cvt.fl (Convert single precision floating point to long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	100011

Set R<r1> = F<r2>. The operand is converted to single precision before being further converted to an integer. The result is a signed integer. If the operand cannot be represented as a 64-bit signed integer value exactly, the closest representable 64-bit signed integer value results.

Sets the R bit of $flags to 1 if an operand is a NaN, whether signalling or quiet.

Sets the E bit of `$flags` to 1 if the integer part of the floating point value is too large to be represented as an integer exactly.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and an operand is a NaN, whether signalling or quiet.
Causes an `FOVERFLOW` exception if an `E` bit of `$state` is 1 and the integer part of the floating point value is too large to be represented as an integer exactly.

7.3.8 cvt.ful (convert single precision floating point to unsigned long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	100111

Set `R<r1>` = `F<r2>`. The operand is converted to single precision before being further converted to an integer. The result is an unsigned integer. If the operand cannot be represented as a 64-bit unsigned integer value exactly, the closest representable 64-bit unsigned integer value results.

Sets the `R` bit of `$flags` to 1 if an operand is a NaN, whether signalling or quiet.
Sets the `E` bit of `$flags` to 1 if the integer part of the floating point value is too large to be represented as an integer exactly.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and an operand is a NaN, whether signalling or quiet.
Causes an `FOVERFLOW` exception if an `E` bit of `$state` is 1 and the integer part of the floating point value is too large to be represented as an integer exactly.

7.3.9 cvt.db (Convert single precision floating point to byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	110000

Set `R<r1>` = `F<r2>`. The result is a 8-bit signed integer, which is then sign-extended to 64 bits. If the operand cannot be represented as an 8-bit signed integer value exactly, the closest representable 8-bit signed integer value results.

Sets the `R` bit of `$flags` to 1 if an operand is a NaN, whether signalling or quiet.
Sets the `E` bit of `$flags` to 1 if the integer part of the floating point value is too large to be represented as an integer exactly.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and an operand is a NaN, whether signalling or quiet.
Causes an `FOVERFLOW` exception if an `E` bit of `$state` is 1 and the integer part of the floating point value is too large to be represented as an integer exactly.

7.3.10 cvt.dub (Convert double precision floating point to unsigned byte)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	110100

Set `R<r1>` = `F<r2>`. The result is a 8-bit unsigned integer, which is then zero-extended to 64 bits. If the operand cannot be represented as an 8-bit unsigned integer value exactly, the closest representable 8-bit unsigned integer value results.

Sets the `R` bit of `$flags` to 1 if an operand is a NaN, whether signalling or quiet.
Sets the `E` bit of `$flags` to 1 if the integer part of the floating point value is too large to be represented as an integer exactly.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and an operand is a NaN, whether signalling or quiet.
Causes an `FOVERFLOW` exception if an `E` bit of `$state` is 1 and the integer part of the floating point value is too large to be represented as an integer exactly.

7.3.11 cvt.dh (Convert double precision floating point to half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	110001

Set `R<r1>` = `F<r2>`. The result is a 16-bit signed integer, which is then sign-extended to 64 bits. If the operand cannot be represented as a 16-bit signed integer value exactly, the closest representable 16-bit signed integer value results.

Sets the `R` bit of `$flags` to 1 if an operand is a NaN, whether signalling or quiet.
Sets the `E` bit of `$flags` to 1 if the integer part of the floating point value is too large to be represented as an integer exactly.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and an operand is a NaN, whether signalling or quiet.
Causes an `FOVERFLOW` exception if an `E` bit of `$state` is 1 and the integer part of the floating point value is too large to be represented as an integer exactly.

7.3.12 cvt.duh (Convert double precision floating point to unsigned half-word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	110101

Set `R<r1>` = `F<r2>`. The result is a 16-bit unsigned integer, which is then zero-extended to 64 bits. If the operand cannot be represented as a 16-bit unsigned integer value exactly, the closest representable 16-bit unsigned integer value results.

Sets the `R` bit of `$flags` to 1 if an operand is a NaN, whether signalling or quiet. Sets the `E` bit of `$flags` to 1 if the integer part of the floating point value is too large to be represented as an integer exactly.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and an operand is a NaN, whether signalling or quiet.
Causes an `FOVERFLOW` exception if an `E` bit of `$state` is 1 and the integer part of the floating point value is too large to be represented as an integer exactly.

7.3.13 cvt.dw (Convert double precision floating point to word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	110010

Set `R<r1>` = `F<r2>`. The result is a 32-bit signed integer, which is then sign-extended to 64 bits. If the operand cannot be represented as a 32-bit signed integer value exactly, the closest representable 32-bit signed integer value results.

Sets the `R` bit of `$flags` to 1 if an operand is a NaN, whether signalling or quiet. Sets the `E` bit of `$flags` to 1 if the integer part of the floating point value is too large to be represented as an integer exactly.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and an operand is a NaN, whether signalling or quiet.
Causes an `FOVERFLOW` exception if an `E` bit of `$state` is 1 and the integer part of the floating point value is too large to be represented as an integer exactly.

7.3.14 cvt.duw (Convert double precision floating point to unsigned word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	110110

Set `R<r1>` = `F<r2>`. The result is a 32-bit unsigned integer, which is then zero-extended to 64 bits. If the operand cannot be represented as a 32-bit unsigned integer value exactly, the closest representable 32-bit unsigned integer value results.

Sets the `R` bit of `$flags` to 1 if an operand is a NaN, whether signalling or quiet. Sets the `E` bit of `$flags` to 1 if the integer part of the floating point value is too large to be represented as an integer exactly.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and an operand is a NaN, whether signalling or quiet.
Causes an `FOVERFLOW` exception if an `E` bit of `$state` is 1 and the integer part of the floating point value is too large to be represented as an integer exactly.

7.3.15 cvt.dl (Convert double precision floating point to long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	110011

Set `R<r1>` = `F<r2>`. The result is a signed integer. If the operand cannot be represented as a 64-bit signed integer value exactly, the closest representable 64-bit signed integer value results.

Sets the `R` bit of `$flags` to 1 if an operand is a NaN, whether signalling or quiet.
Sets the `E` bit of `$flags` to 1 if the integer part of the floating point value is too large to be represented as an integer exactly.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and an operand is a NaN, whether signalling or quiet.
Causes an `FOVERFLOW` exception if an `E` bit of `$state` is 1 and the integer part of the floating point value is too large to be represented as an integer exactly.

7.3.16 cvt.dul (convert double precision floating point to unsigned long word)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	110111

Set `R<r1>` = `F<r2>`. The result is an unsigned integer. If the operand cannot be represented as a 64-bit unsigned integer value exactly, the closest representable 64-bit unsigned integer value results.

Sets the `R` bit of `$flags` to 1 if an operand is a NaN, whether signalling or quiet.
Sets the `E` bit of `$flags` to 1 if the integer part of the floating point value is too large to be represented as an integer exactly.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and an operand is a NaN, whether signalling or quiet.
Causes an `FOVERFLOW` exception if an `E` bit of `$state` is 1 and the integer part of the floating point value is too large to be represented as an integer exactly.

7.3.17 cvt.bf (convert byte to single precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	101000

Set `F<r1>` = `R<r2>`. The lower 8 bits of an operand are treated as a signed integer, that is then converted to a single precision floating point value, which is then expanded to double precision. The higher 56 bits of an operand are ignored for conversion purposes.

Causes an `OPCODE` exception if the floating point feature is not available.

7.3.18 cvt.ubf (convert unsigned byte to single precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	101100

Set `F<r1>` = `R<r2>`. The lower 8 bits of an operand are treated as an unsigned integer, that is then converted to a single precision floating point value, which is then expanded to double precision. The higher 56 bits of an operand are ignored for conversion purposes.

Causes an `OPCODE` exception if the floating point feature is not available.

7.3.19 cvt.hf (convert half-word to single precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	101001

Set `F<r1>` = `R<r2>`. The lower 16 bits of an operand are treated as a signed integer, that is then converted to a single precision floating point value, which is then expanded to double precision. The higher 48 bits of an operand are ignored for conversion purposes.

Causes an `OPCODE` exception if the floating point feature is not available.

7.3.20 cvt.uhf (convert unsigned half-word to single precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	101101

Set F<r1> = R<r2>. The lower 16 bits of an operand are treated as an unsigned integer, that is then converted to a single precision floating point value, which is then expanded to double precision. The higher 48 bits of an operand are ignored for conversion purposes.

Causes an OPCODE exception if the floating point feature is not available.

7.3.21 cvt.wf (convert word to single precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	101010

Set F<r1> = R<r2>. The lower 32 bits of an operand are treated as a signed integer, that is then converted to a single precision floating point value, which is then expanded to double precision. The higher 32 bits of an operand are ignored for conversion purposes. If an operand cannot be represented exactly as a single precision floating point value, the closest representable value results.

Sets the I bit of $flags to 1 if an inexact result is produced.

Causes an OPCODE exception if the floating point feature is not available.
Causes an FINEXACT exception if an I bit of $state is 1 and an inexact result is produced.

7.3.22 cvt.uwf (convert unsigned word to single precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	101110

Set F<r1> = R<r2>. The lower 32 bits of an operand are treated as an unsigned integer, that is then converted to a single precision floating point value, which is then expanded to double precision. The higher 32 bits of an operand are ignored for conversion purposes. If an operand cannot be represented exactly as a single precision floating point value, the closest representable value results.

Sets the I bit of $flags to 1 if an inexact result is produced.

Causes an OPCODE exception if the floating point feature is not available.
Causes an FINEXACT exception if an I bit of $state is 1 and an inexact result is produced.

7.3.23 cvt.lf (convert long word to single precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	101011

Set `F<r1>` = `R<r2>`. The operand is a signed integer. It is converted to a single precision floating point value, which is then expanded to double precision. If an operand cannot be represented exactly as a single precision floating point value, the closest representable value results.

Sets the `I` bit of `$flags` to 1 if an inexact result is produced.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FINEXACT` exception if an `I` bit of `$state` is 1 and an inexact result is produced.

7.3.24 cvt.ulf (convert unsigned long word to single precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	101111

Set `F<r1>` = `R<r2>`. The operand is an unsigned integer. It is converted to a single precision floating point value, which is then expanded to double precision. If an operand cannot be represented exactly as a single precision floating point value, the closest representable value results.

Sets the `I` bit of `$flags` to 1 if an inexact result is produced.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FINEXACT` exception if an `I` bit of `$state` is 1 and an inexact result is produced.

7.3.25 cvt.bd (convert byte to double precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	111000

Set `F<r1>` = `R<r2>`. The lower 8 bits of an operand are treated as a signed integer, that is then converted. The higher 56 bits of an operand are ignored for conversion purposes.

Causes an `OPCODE` exception if the floating point feature is not available.

7.3.26 cvt.ubd (convert unsigned byte to double precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	111100

Set `F<r1>` = `R<r2>`. The lower 8 bits of an operand are treated as an unsigned integer, that is then converted. The higher 56 bits of an operand are ignored for conversion purposes.

Causes an `OPCODE` exception if the floating point feature is not available.

7.3.27 cvt.hd (convert half-word to double precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	111001

Set `F<r1>` = `R<r2>`. The lower 16 bits of an operand are treated as a signed integer, that is then converted. The higher 48 bits of an operand are ignored for conversion purposes.

Causes an `OPCODE` exception if the floating point feature is not available.

7.3.28 cvt.uhd (convert unsigned half-word to double precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	111101

Set `F<r1>` = `R<r2>`. The lower 16 bits of an operand are treated as an unsigned integer, that is then converted. The higher 48 bits of an operand are ignored for conversion purposes.

Causes an `OPCODE` exception if the floating point feature is not available.

7.3.29 cvt.wd (convert word to double precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	111010

Set `F<r1>` = `R<r2>`. The lower 32 bits of an operand are treated as a signed integer, that is then converted. The higher 32 bits of an operand are ignored for conversion purposes.

Causes an `OPCODE` exception if the floating point feature is not available.

7.3.30 cvt.uwd (convert unsigned word to double precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	111110

Set `F<r1>` = `R<r2>`. The lower 32 bits of an operand are treated as an unsigned integer, that is then converted. The higher 32 bits of an operand are ignored for conversion purposes.

Causes an `OPCODE` exception if the floating point feature is not available.

7.3.31 cvt.ld (convert long word to double precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	111011

Set `F<r1>` = `R<r2>`. The operand is a signed integer. If an operand cannot be represented exactly as a floating point value, the closest representable value results.

Sets the `I` bit of `$flags` to 1 if an inexact result is produced.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FINEXACT` exception if an `I` bit of `$state` is 1 and an inexact result is produced.

7.3.32 cvt.uld (convert unsigned long word to double precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	111111

Set `F<r1>` = `R<r2>`. The operand is an unsigned integer. If an operand cannot be represented exactly as a floating point value, the closest representable value results.

Sets the `I` bit of `$flags` to 1 if an inexact result is produced.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FINEXACT` exception if an `I` bit of `$state` is 1 and an inexact result is produced.

7.3.33 cvt.df (Convert double precision floating point to single precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10000	000001

Set `F<r1>` = `F<r2>` rounded to nearest 32-bit floating point value. The result value is the 64-bit floating point value closest to the operand that can be represented in 32-bit floating point format without any loss of magnitude or precision.

Sets the `R` bit of `$flags` to 1 if either operand is a signalling NaN.
Sets the `E` bit of `$flags` to 1 if a floating point overflow occurs during the operation.
Sets the `U` bit of `$flags` to 1 if a floating point underflow occurs during the operation.
Sets the `I` bit of `$flags` to 1 if an inexact result is produced.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and either operand is a signalling NaN.
Causes an `FOVERFLOW` exception if an `E` bit of `$state` is 1 and a floating point overflow occurs during the operation.
Causes an `FUNDERFLOW` exception if an `U` bit of `$state` is 1 and a floating point underflow occurs during the operation.
Causes an `FINEXACT` exception if an `I` bit of `$state` is 1 and an inexact result is produced.

7.4 Comparison instructions

7.4.1 seq.d (set equal double precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	10000	011000

If `F<r2>` = `F<r3>`, then set `R<r1> = 1`, otherwise then set `R<r1> = 0`. If either operand is a NaN (whether signalling or quiet) and an `R` bit of `$state` is 0, `R<r1>` is set to 0.

Sets the `R` bit of `$flags` to 1 if either operand is a NaN, whether signalling or quiet.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and either operand is a NaN, whether signalling or quiet.

7.4.2 sne.d (set not equal double precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	10000	011001

If `F<r2>` ≠ `F<r3>`, then set `R<r1> = 1`, otherwise then set `R<r1> = 0`. If either operand is a NaN (whether signalling or quiet) and an `R` bit of `$state` is 0, `R<r1>` is set to 0.

Sets the `R` bit of `$flags` to 1 if either operand is a NaN, whether signalling or quiet.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and either operand is a NaN, whether signalling or quiet.

7.4.3 slt.d (set less than double precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	10000	011010

If `F<r2>` < `F<r3>`, then set `R<r1> = 1`, otherwise then set `R<r1> = 0`. If either operand is a NaN (whether signalling or quiet) and an `R` bit of `$state` is 0, `R<r1>` is set to 0.

Sets the `R` bit of `$flags` to 1 if either operand is a NaN, whether signalling or quiet.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and either operand is a NaN, whether signalling or quiet.

7.4.4 sle.d (set less than or equal double precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	10000	011011

If `F<r2>` ≤ `F<r3>`, then set `R<r1> = 1`, otherwise then set `R<r1> = 0`. If either operand is a NaN (whether signalling or quiet) and an `R` bit of `$state` is 0, `R<r1>` is set to 0.

Sets the `R` bit of `$flags` to 1 if either operand is a NaN, whether signalling or quiet.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and either operand is a NaN, whether signalling or quiet.

7.4.5 sgt.d (set greater than double precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	10000	011100

If `F<r2>` > `F<r3>`, then set `R<r1> = 1`, otherwise then set `R<r1> = 0`. If either operand is a NaN (whether signalling or quiet) and an `R` bit of `$state` is 0, `R<r1>` is set to 0.

Sets the `R` bit of `$flags` to 1 if either operand is a NaN, whether signalling or quiet.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and either operand is a NaN, whether signalling or quiet.

7.4.6 sge.d (set greater than or equal double precision floating point)

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	r3	10000	011101

If `F<r2>` ≥ `F<r3>`, then set `R<r1> = 1`, otherwise then set `R<r1> = 0`. If either operand is a NaN (whether signalling or quiet) and an `R` bit of `$state` is 0, `R<r1>` is set to 0.

Sets the `R` bit of `$flags` to 1 if either operand is a NaN, whether signalling or quiet.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and either operand is a NaN, whether signalling or quiet.

7.5 Load/store instructions

7.5.1 l.f (load single precision floating point)

Bits	31 26	25 21	20 16	15 0
	101100	r1	r2	immediate

Loads the 4-byte floating point value from the memory address `addr`, converts it into the 64-bit floating point value and stores the result in `F<r1>`. Depending on the contents of the bit `B` of `$state`, the value can be loaded in either big-endian or little-endian format.

Causes an `OPCODE` exception if the floating point feature is not available.

7.5.2 l.d (load double precision floating point)

Bits	31 26	25 21	20 16	15 0
	101101	r1	r2	immediate

Loads the 8-byte floating point value from the memory address `addr` into `F<r1>`. Depending on the contents of the bit `B` of `$state`, the value can be loaded in either big-endian or little-endian format.

Causes an `OPCODE` exception if the floating point feature is not available.

7.5.3 s.f (store single precision floating point)

Bits	31 26	25 21	20 16	15 0
	101110	r1	r2	immediate

Converts the `F<r1>` into a 4-byte floating point value and stores this value into the memory at address `addr`. Depending on the contents of the bit `B` of `$state`, the value can be stored in either big-endian or little-endian format.

Causes an `OPCODE` exception if the floating point feature is not available.

7.5.4 s.d (store double precision floating point)

Bits	31 26	25 21	20 16	15 0
	101111	r1	r2	immediate

Stores `F<r1>` into the memory at address `addr`. Depending on the contents of the bit `B` of `$state`, the value can be stored in either big-endian or little-endian format.

Causes an `OPCODE` exception if the floating point feature is not available.

7.6 Flow control instructions

7.6.1 beq.d (branch on equal double precision floating point)

Bits	31 26	25 21	20 16	15 0
	111000	r1	r2	immediate

If `F<r1> = F<r2>`, then sets `$ip = baddr`; otherwise has no effect. If either operand is a NaN (whether signalling or quiet) and an `R` bit of `$state` is 0, the instruction has no effect.

Sets the `R` bit of `$flags` to 1 if either operand is a NaN, whether signalling or quiet.

Causes an `OPCODE` exception if the floating point feature is not available.

Causes an FOPERAND exception if an R bit of $state is 1 and either operand is a NaN, whether signalling or quiet.

7.6.2 bne.d (branch on not equal double precision floating point)

Bits	31 26	25 21	20 16	15 0
	111001	r1	r2	immediate

If F<r1> ≠ F<r2>, then sets $ip = baddr; otherwise has no effect. If either operand is a NaN (whether signalling or quiet) and an R bit of $state is 0, the instruction has no effect.

Sets the R bit of $flags to 1 if either operand is a NaN, whether signalling or quiet.

Causes an OPCODE exception if the floating point feature is not available.
Causes an FOPERAND exception if an R bit of $state is 1 and either operand is a NaN, whether signalling or quiet.

7.6.3 blt.d (branch on less than double precision floating point)

Bits	31 26	25 21	20 16	15 0
	111010	r1	r2	immediate

If F<r1> < F<r2>, then sets $ip = baddr; otherwise has no effect. If either operand is a NaN (whether signalling or quiet) and an R bit of $state is 0, the instruction has no effect.

Sets the R bit of $flags to 1 if either operand is a NaN, whether signalling or quiet.

Causes an OPCODE exception if the floating point feature is not available.
Causes an FOPERAND exception if an R bit of $state is 1 and either operand is a NaN, whether signalling or quiet.

7.6.4 ble.d (branch on less than or equal double precision floating point)

Bits	31 26	25 21	20 16	15 0
	111011	r1	r2	immediate

If F<r1> ≤ F<r2>, then sets $ip = baddr; otherwise has no effect. If either operand is a NaN (whether signalling or quiet) and an R bit of $state is 0, the instruction has no effect.

Sets the R bit of $flags to 1 if either operand is a NaN, whether signalling or quiet.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and either operand is a NaN, whether signalling or quiet.

7.6.5 bgt.d (branch on greater than double precision floating point)

Bits	31 26	25 21	20 16	15 0
	111100	r1	r2	immediate

If `F<r1> > F<r2>`, then sets `$ip = baddr`; otherwise has no effect. If either operand is a NaN (whether signalling or quiet) and an `R` bit of `$state` is 0, the instruction has no effect.

Sets the `R` bit of `$flags` to 1 if either operand is a NaN, whether signalling or quiet.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and either operand is a NaN, whether signalling or quiet.

7.6.6 bge.d (branch on greater than or equal double precision floating point)

Bits	31 26	25 21	20 16	15 0
	111101	r1	r2	immediate

If `F<r1> ≥ F<r2>`, then sets `$ip = baddr`; otherwise has no effect. If either operand is a NaN (whether signalling or quiet) and an `R` bit of `$state` is 0, the instruction has no effect.

Sets the `R` bit of `$flags` to 1 if either operand is a NaN, whether signalling or quiet.

Causes an `OPCODE` exception if the floating point feature is not available.
Causes an `FOPERAND` exception if an `R` bit of `$state` is 1 and either operand is a NaN, whether signalling or quiet.

8 The debug feature

This section describes instructions and facilities provided as a part of the Cereon Debug feature. It provides support for hardware-level debugging of both user and system code.

8.1 Debug events

A debug event is a condition that can occur during an execution of an instruction. The Cereon Debug feature allows tracking up to 16 different conditions at the same time.

When the corresponding condition occurs during an execution of an instruction, the fact is remembered until the instruction execution finishes successfully. Three possibilities can arise:

- An execution of an instruction can trigger an exception after one or more debug events have been recorded, they are all discarded immediately and an exception occurs normally.
- An instruction execution finishes successfully and `PROGRAM` interrupts are enabled. The debug event causes a `PROGRAM` interrupt with an interrupt status code referring to the debug event that caused it. If there were several debug events recorded for the instruction, the one with a highest priority is processed and the other ones are discarded.
- An instruction execution finishes successfully and `PROGRAM` interrupts are disabled. All debug events recorded for the instruction are discarded.

8.2 Debug registers

The 32 64-bit debug registers `d0..d31` are logically divided into 16 consecutive groups of 2 (i.e. `d0/d1`, `d2/d3`, etc.) Each of these groups can either be unused, or it can contain a definition of a single debug event, thus making it possible to track up to 16 different debug events at the same time.

The priority of a debug event depends on which pair of debug registers it is specified in. Debug events specified in debug registers with smaller numbers have higher priorities.

8.2.1 Debug event specification

The exact form in which a debug event is specified in a debug register pair is dependent on the type of the debug event. However, the highest 6 bits of the debug register with an even number always have the same contents:

Bits	63	62	61 58	57 0
	K	U	type	Specific to debug event type

The meaning of individual fields within the upper 6 bits of an even-numbered debug register is explained below.

8.2.1.1 K (Kernel mode)

When this bit is 1, the debug event can occur in Kernel mode; otherwise, the debug event will not occur in Kernel mode even if the condition it describes occurs during instruction execution.

8.2.1.2 U (User mode)

When this bit is 1, the debug event can occur in User mode; otherwise, the debug event will not occur in User mode even if the condition it describes occurs during instruction execution.

8.2.1.3 Type

These 4 bits represent a type of the debug event. Altogether there can be 16 different debug event types, although only one is defined in this version of the Cereon architecture.

8.2.2 Memory Access debug event

The Memory Access debug event occurs when a specific area of memory is accessed in a specific mode. It can be used to set up both variable watchpoints and code breakpoints.

The specification of a Memory Access debug event uses the debug register pair in the following way:

Bits	63	62	61 58	57 55	54	53	52	51 48	47 0
(even)	K	U	0000	-	R	W	X	scale	size

Bits	63 0
(odd)	address

The meaning of individual fields within the debug register pair is explained below.

8.2.2.1 Address

Specifies the start address of the memory area being monitored.

8.2.2.2 Scale and Size

The value `size << scale` (same as $\texttt{size*2}^{\texttt{scale}}$) specifies the size, in bytes, of the memory area being watched. If the value $\texttt{address+size*2}^{\texttt{scale}}$ wraps around, the debug event specification is invalid and a debug event never occurs.

8.2.2.3 X (eXecute)

If this bit is 1, an attempt to execute an instruction from the memory area under watch causes a debug event to occur; otherwise executing instructions from the memory area under watch does not cause the debug event.

8.2.2.4 W (Write)

If this bit is 1, an attempt to write to the memory area under watch causes a debug event to occur; otherwise writing to the memory area under watch does not cause the debug event.

8.2.2.5 R (Read)

If this bit is 1, an attempt to read from the memory area under watch causes a debug event to occur; otherwise reading from the memory area under watch does not cause the debug event.

8.2.3 Disabling debug events

Any pair of debug registers which has zeroes in the two highest bits of an even-numbered debug register specifies a debug event that cannot occur in either Kernel or User mode. Such pair of debug registers describes a disabled debug event.

8.3 Data movement instructions

8.3.1 mov.gr (move debug register to general purpose register) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10001	000000

Sets `R<r1> = D<r2>`.

Causes an `OPCODE` exception if the debug feature is not available.
Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.
Causes a `PRIVILEGED` exception if the `K` bit of `$state` is 0.

8.3.2 mov.rg (move general purpose register to debug register) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10001	000001

Sets `D<r1> = R<r2>`.

Causes an `OPCODE` exception if the debug feature is not available.
Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.
Causes a `PRIVILEGED` exception if the `K` bit of `$state` is 0.

9 The performance monitoring feature

The performance monitoring feature, if present, provides hardware support for the high-precision profiling of the running code. This is achieved by counting various events that occur during program execution (such as instruction completions, cache hits and misses, etc.)

Note that the rudimentary support for the program profiling is always present in the form of the `$cc` (Cycle Counter) control register; the performance monitoring feature extends this support by being able to answer questions other than "how many cycles did the program take?"

9.1 Program events

A program event is a condition that may occur while a program is being executed. The performance monitoring feature keeps counts of how many times each program event has occurred; these counts are incremented as the program runs.

Currently the following program events are counted:

Event name	Event description
`INSTRUCTION`	An instruction has finished execution.
`MOV`	A data movement instruction has finished execution.
`ALU`	An integer computational instruction has finished execution.
`FPU`	A floating-point computational instruction has finished execution.
`LOAD`	A load instruction has finished execution.
`STORE`	A store instruction has finished execution.
`JUMP`	An unconditional jump instruction has finished execution.
`BRANCH`	A conditional branch instruction has finished execution.
`IO`	An I/O instruction has finished execution.
`CONTROL`	A control instruction has finished execution.
`ICACHEHIT`	An I-Cache hit occurred while fetching an instruction.
`ICACHEMISS`	An I-Cache miss occurred while fetching an instruction.
`DCACHERHIT`	A D-Cache hit occurred while loading a data item from memory.
`DCACHERMISS`	A D-Cache miss occurred while loading a data item from memory.
`DCACHEWHIT`	A D-Cache hit occurred while storing a data item to memory.
`DCACHEWMISS`	A D-Cache miss occurred while storing a data item to memory.
`ITLBHIT`	An I-TLB hit occurred while translating an instructuion virtual

	address.
ITLBMISS	An I-TLB miss occurred while translating an instructuion virtual address.
DTLBHIT	A D-TLB hit occurred while translating a data virtual address.
DTLBMISS	A D-TLB miss occurred while translating a data virtual address.
JPREDICT	An unconditional jump was predicted correctly.
JMISPREDICT	An unconditional jump was predicted incorrectly.
BPREDICT	A conditional branch was predicted correctly.
BMISPREDICT	A conditional branch was predicted incorrectly.
TMINTERRUPT	A TIMER interrupt occurred.
IOINTERRUPT	An IO interrupt occurred.
SVCINTERRUPT	An SVC interrupt occurred.
PRGINTERRUPT	A PROGRAM interrupt occurred.
EXTINTERRUPT	An EXTERNAL interrupt occurred.
HWINTERRUPT	A HARDWARE interrupt occurred.

9.2 Performance monitoring registers

Altogeher there are 32 performance monitoring registers m0..m31. Their usage is summarized in the following table:

Register	Symbolic name	Usage convention
m0	$mflags	Performance monitoring flags.
m1	$cnt.instruction	INSTRUCTION event counter.
m2	$cnt.mov	MOV event counter.
m3	$cnt.alu	ALU event counter.
m4	$cnt.fpu	FPU event counter.
m5	$cnt.load	LOAD event counter.
m6	$cnt.store	STORE event counter.
m7	$cnt.jump	JUMP event counter.
m8	$cnt.branch	BRANCH event counter.
m9	$cnt.io	IO event counter.
m10	$cnt.control	CONTROL event counter.
m11	$cnt.icachehit	ICACHEHIT event counter.
m12	$cnt.icachemiss	ICACHEMISS event counter.
m13	$cnt.dcacherhit	DCACHERHIT event counter.
m14	$cnt.dcachermiss	DCACHERMISS event counter.
m15	$cnt.dcachewhit	DCACHEWHIT event counter.
m16	$cnt.dcachewmiss	DCACHEWMISS event counter.
m17	$cnt.itlbhit	ITLBHIT event counter.

`m18`	`$cnt.itlbmiss`	`ITLBMISS` event counter.
`m19`	`$cnt.dtlbhit`	`DTLBHIT` event counter.
`m20`	`$cnt.dtlbmiss`	`DTLBMISS` event counter.
`m21`	`$cnt.jpredict`	`JPREDICT` event counter.
`m22`	`$cnt.jmispredict`	`JMISPREDICT` event counter.
`m23`	`$cnt.bpredict`	`BPREDICT` event counter.
`m24`	`$cnt.bmispredict`	`BMISPREDICT` event counter.
`m25`	`$cnt.tminterrupt`	`TMINTERRUPT` event counter.
`m26`	`$cnt.iointerrupt`	`IOINTERRUPT` event counter.
`m27`	`$cnt.svcinterrupt`	`SVCINTERRUPT` event counter.
`m28`	`$cnt.prginterrupt`	`PRGINTERRUPT` event counter.
`m29`	`$cnt.extinterrupt`	`EXTINTERRUPT` event counter.
`m30`	`$cnt.hwinterrupt`	`HWINTERRUPT` event counter.
`m31`	-	Reserved for future use

9.2.1 The $mflags register

While other registers act as counters, the `$mflags` register is a bit mask that specifies which of the counters are active. Individual bits within the $mflags register have the following meaning:

- Bits 0 and 32 are ignored.
- Bits 1..31 specify which counters are active in User mode. If bit N (1 ≤ N ≤ 31) is set to 1, then the register `m<N>` is incremented if the corresponding program event occurs in User mode; otherwise corresponding program events occurring in User mode are ignored.
- Bits 33..63 specify which counters are active in Kernel mode. If bit N (33 ≤ N ≤ 63) is set to 1, then the register `m<N-32>` is incremented if the corresponding program event occurs in Kernel mode; otherwise corresponding program events occurring in Kernel mode are ignored.

This scheme allows independent counting of events in User and Kernel modes, as well as starting and freezing any number of counters with a single instruction.

9.3 Data movement instructions

9.3.1 mov.mr (move performance monitoring register to general purpose register) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10010	000000

Sets `R<r1> = M<r2>`.

Causes an `OPCODE` exception if the performance monitoring feature is not available.
Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.
Causes a `PRIVILEGED` exception if the `K` bit of `$state` is 0.

9.3.2 mov.rm (move general purpose register to performance monitoring register) [privileged]

Bits	31 26	25 21	20 16	15 11	10 6	5 0
	000001	r1	r2	00000	10010	000001

Sets `M<r1> = R<r2>`.

Causes an `OPCODE` exception if the performance monitoring feature is not available.
Causes an `OPCODE` exception if bits 11..15 of the instructions are not 0.
Causes a `PRIVILEGED` exception if the `K` bit of `$state` is 0.

10 The protected memory feature

Cereon protected memory feature, when present, provides segmented virtual memory with 64-bit address space.

When a processor equipped with the protected memory feature runs in virtual mode, 64-bit memory addresses used by instructions do not refer to physical memory addresses. Instead, they are translated into physical memory addresses using combination of hardware and software.

10.1 Protected memory concepts

The following hardware elements participate in virtual address translation:

- Region table.
- Region table pointer.

10.1.1 Region table

A region table is a hidden table used by the protected memory manager to perform virtual address translation. This table contains 8 entries, each describing a single continuous region of memory. A program can access memory in any of these regions but not any other memory; in addition regions can be restricted to perform only a specific type of access (e.g. “execute only”, or “read/write, but not execute”) or to rescrict access to a specific mode (e.g. “this region can be accessed in kernel mode only).

10.1.2 Region table pointer

When a value is assigned to a control regisrter `$pth`, it is assumed that this value is a real address of the region table residing in memory. Upon such assignment, the region table is loaded from memory into the hidden registers of the memory protection unit and, as soon as processor switches to a virtual mode, starts governing all memory accesses.

In order to reduce the cost of a context switch, the memory protection unit is permitted to cache the region tables loaded from memory internally. In this case, assigning a new value to the `$pth` register may result in the region table being loaded from the memory protection unit’s cache instead of from memory. This internal region table cache acts as a sort of TLB; so a memory barrier instruction is necessary in order to ensure that a region table is actually loaded from memory and not from the cache.

10.1.3 Preparing region table for use

In order to set up a region table, it must first be prepared in memory, and then its real address loaded into `$pth`.

A region table appears in memory as a sequence of 8 consecutive entries, each entry consisting of 3 consecutive long words in the current processor’s byte order and describing a single region of memory. The total size of a region table in memory is always 8 * 3 * 8 = 192 bytes; the region table must be aligned at a 8-byte boundary regardless of whether a processor has unaligned operands feature or not.

A single 24-byte entry of a region table has the following format:

Bits	63 3	2	1	0
Base+0	virtual base	Rk	Wk	Xk

Bits	63 3	2	1	0
Base+8	real base	Ru	Wu	Xu

Bits	63 0
Base+16	size

Individual fields within a region table entry have the following meaning:

- Virtual base – this is the higher 61 bits of a virtual address where the region described by this entry starts in a process' virtual address space. The implied lower 3 bits of this address are always 0; therefore all regions are always 8-byte aligned.
- Size – this value is one less than the size of the region. Therefore, the region covers the `[<virtual base> .. <virtual base> + <size>]` range of virtual addresses.
- Real base – this is the higer 61 bits of a real address where the region starts in real memory. As for the virtual address, the lower 3 bits of a real region base address are assumed to be 0.
- `Rk`, `Wk` and `Xk` – these bits are set to 1 iff a processor can, correspondingly, read from, write to or execute code from the corresponding region when in Kernel mode.
- `Ru`, `Wu` and `Xu` – these bits are set to 1 iff a processor can, correspondingly, read from, write to or execute code from the corresponding region when in User mode.

10.2 Virtual address translation

When a virtual address `va` is translated into a physical address, the following steps are performed:

1. All 8 of the region table entries held in the hidden registers of a memory protection unit are checked in parallel to determine if `va` is in range `[<virtual base> .. <virtual base> + <size>)` for each of them.
2. If none of the entries match, an `IADDRESS` or `DADDRESS` exception occurs, depending on whether instruction or data address was being translated.
3. Otherwise, the 3 access control bits of all matching entries are ORed together, forming an access control mask `ACM`. Depending on whether processor is in kernel or user mode, either kernel-mode (`Rk`, `Wk` and `Xk`) or user-mode (`Ru`, `Wu` and `Xu`) access control bits are used.

4. `ACM` is checked to see if the required memory access is permitted. If not, an `IACCESS` or `DACCESS` exception occurs, depending on whether instruction or data address was being translated.
5. Otherwise, the real memory address is calculated as `va - <virtual base> + <real base>`. If more than one region table entry matches the virtual address `va`, and calculating real address for these matching entries yields different values, the result of address translation is undefined and an `IADDRESS` or `DADDRESS` exception occurs, depending on whether instruction or data address was being translated.

10.3 Aliasing

The memory protection unit specifically permits several disjoint virtual memory regions to refer to the same physical memory area. When this is the case, the same memory will appear to have the corresponding number of identical “copies” within a process address space.

11 The virtual memory feature

Cereon virtual memory feature, when present, provides paged virtual memory with 64-bit address space.

When a processor equipped with the virtual memory feature runs in virtual mode, 64-bit memory addresses used by instructions do not refer to physical memory addresses. Instead, they are translated into physical memory addresses using combination of hardware and software.

11.1 Virtual memory concepts

The following hardware elements participate in virtual address translation:

- Page table format descriptor.
- Page table pointer.
- Translation-lookaside buffer (TLB).
- Current context ID (`CID`) in `$state` register.

11.1.1 Page table format descriptor

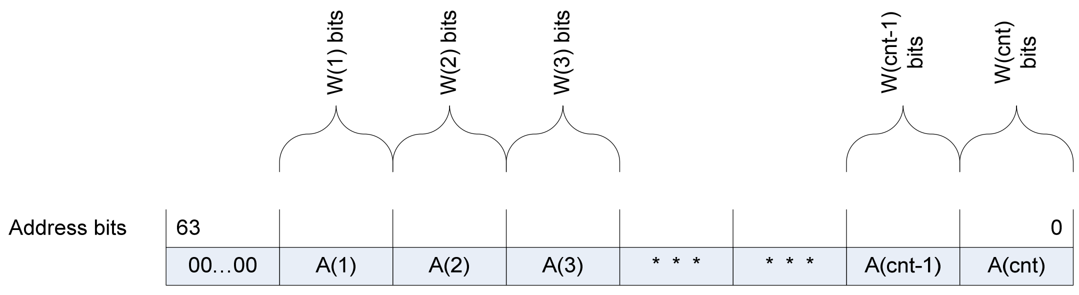
A special 64-bit value called a Page Table Format Descriptor specifies the format of the page table used by the processor. This Page Table Format Descriptor has the following layout:

bits	63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0
	cnt	W(1)	W(2)	W(3)	W(4)	W(5)	W(6)	W(7)

Its fields have the following meanings:

- `cnt` – The number of fields in the virtual address. This value must be in range 2..7.
- `w(1)`..`w(7)` – Address field width specifications. The field `w(k)` specifies how many address bits correspond to the k_{th} field in the virtual address. Only elements `w(1)`..`w(cnt)` are used, since the virtual address only has `cnt` fields. It is always the case that `w(1)+w(2)+`…`+w(cnt-1)+w(cnt)<=64`. Also, it is always the case that `w(cnt)>=8`.

When a virtual address is translated, it is first broken into a number of fields, as shown below:



There are `cnt` fields altogether in the virtual address, their length in bits specified by corresponding width values in the page table format descriptor. If `w(1)+w(2)+…+w(cnt-1)+w(cnt)<64`, then virtual address contains less than 64 significant bits (as is indeed the case in the example above). In this case the higher (unused) bits of a virtual address must all be zero, or an address translation exception occurs.

11.1.2 Page table header pointer

A 64-bit control register `$pth` specifies the physical address of the 16-byte data structure known as Page Table Header in a physical memory; the Page Table Header, in turn, specifies the page table format and the address of level 1 page table used by the processor. In multi-level page tables each level `k` of the page table contains $2^{w(k)}$ entries.

A Page Table Header must be placed at a physical memory address that is a multiple of 8, regardless of whether an Unaligned feature is available or not. When this address is loaded into a `$pth` control register:

- The 64-bit value at address `$pth` is assumed to point to the level 1 page table.
- The 64-bit value at address `$pth+8` is assumed to be a page table format descriptor.

When the `$pth` register is assigned a value, the Page Table Header is read from memory and stored in hidden processor registers, which are then used by virtual address translation mechanism.

11.1.3 Page table structure

For each level `1<=k<=cnt-2`, every entry in level `k` page table specifies the physical address of level `k+1` page table or 0 if the next level page table does not exist.

At the level before last, entries in the level `cnt-2` page table are physical memory addresses of page descriptor tables. Each page descriptor table is a table containing $2^{w(cnt-1)}$ page descriptors, 8 bytes each. A page descriptor has the following format:

Bits	63 8	7	6	5	4	3	2	1	0
	base	A	D	U	L	K	R	W	X

Individual fields within the lower byte of a page descriptor have the following meaning:

11.1.3.1 base

The upper 56 bits of the physical address of the corresponding page in the physical memory. The lower 8 bits of the page address are always assumed to be 0; therefore, a page always starts at a 256-byte boundary.

11.1.3.2 A (accessed)

This bit is set to 1 whenever a virtual page is accessed. Paging software will usually set this bit to 0 when page is loaded into the physical memory and can later determine if the page has been used by examining this bit – if there were any accesses to the page the bit `A` will be 1.

11.1.3.3 D (dirty)

This bit is set to 1 whenever a virtual page is modified. Paging software will usually set this bit to 0 when page is loaded into the physical memory and can later determine if the page has been modified by examining this bit – if there were any write accesses to the page the bit `A` will be 1.

11.1.3.4 U (used)

This bit is 1 if the corresponding page exists, 0 if it does not.

11.1.3.5 L (loaded)

This bit is 1 if the corresponding page is currently in physical memory, 0 otherwise.

11.1.3.6 K (kernel)

This bit is 1 if this page can only be used in Kernel mode, 0 if it is available in User mode.

11.1.3.7 R (read)

This bit is 1 if data can be read from this page, 0 otherwise.

11.1.3.8 W (write)

This bit is 1 if data can be written to this page, 0 otherwise.

11.1.3.9 X (execute)

This bit is 1 if instructions can be executed from this page, 0 otherwise.

11.2 Virtual address translation

When a virtual address `va` is translated into a physical address, the following steps are performed:

1. The virtual address `va` is broken into individual fields `a(1)..a(cnt)` according to the current page table format descriptor.
2. The 64-bit value `pa` is set to the page table pointer.
3. For each `k` from `1` to `cnt-2` steps 4..6 are performed
4. The 64-bit value `pp` is loaded from the physical memory address `pa+8*a(k)`. If the loading fails, the virtual address translation fails as well and a `PAGETABLE` exception occurs.
5. If `pp=0`, then virtual address translation fails and an address exception (either `IADDRESS` or `DADDRESS` depending on whether instruction or data virtual address is being translated) occurs.
6. Otherwise (i.e. `pp<>0`) assign `pp` to `pa` and continue the loop from step 4 if `k<cnt-2`.

7. Load the 64-bit page descriptor `pd` from the physical memory address `pa+8*a(cnt-1)`. If the loading fails, the virtual address translation fails as well and a `PAGETABLE` exception occurs.
8. If bit `U` of `pd` is `0`, the page does not exist; the virtual address translation fails and an address exception (either `IADDRESS` or `DADDRESS` depending on whether instruction or data virtual address is being translated) occurs.
9. If bit `L` of `pd` is `0`, the page is not currently in memory. This causes either `IPAGEFAULT` or `DPAGEFAULT` exception, depending on whether instruction or data virtual address is being translated.
10. If bit `L` of `pd` is `1`, the page is currently in memory. Bits `K`, `R`, `W` and `X` of `pd` are used to check if a memory access should be allowed. If not, the virtual address translation fails and an access exception (either `IACCESS` or `DACCESS` depending on whether instruction or data virtual address is being translated) occurs.
11. The lowest `w(cnt)` bits of `pd` are replaced with `a(cnt)` and the result is then used as a physical memory address.

Although the virtual address translation rules may seem unwieldy, in reality virtual address translation are invariably assisted by Translation Lookaside Buffers (TLBs) with hit rates close to 100%, so the whole virtual address translation process has to be performed very infrequently. The main goal of the added flexibility was to allow fine-tuning virtual memory mechanisms to suit a concrete process (for example, different processes can have different page table structures as well as different page sizes, all in order to further reduce TLB miss rates. A process that uses a 64K code segment, a 64K data segment and a 64K stack segment may be set up with 3 64K virtual pages in its page table, in which case there will be no TLB misses at all).

11.3 Accessing virtual memory across page boundaries

Unless an Unaligned Operand feature is available, all instructions which load/store data from/to memory require the memory address (i.e. physical memory address in Real mode or virtual memory address in Virtual mode) to be a multiple of a data element size (i.e. a 16-bit integer value can be loaded from/stored to an even address, a full 64-bit integer value or a 64-bit IEEE floating point value require memory address to be a multiple of 8, etc.). A violation of this requirement causes a `DALIGN` exception.

Similarly, an attempt to execute an instruction that does not start at a 4-byte boundary causes an `IALIGN` exception. Note that, unlike data alignment, instruction alignment is not subject to relaxation if an Unaligned Operand feature is available.

This, together with the virtual address translation algorithm, ensures that a program running in a Virtual mode never accesses memory across virtual page boundaries. Also, it is always the case that a virtual address which is a multiple of 2, 4 or 8 is always translated into a physical address which is a multiple of the same value.

In the presence of an Unaligned Operand feature, the processor is responsible for breaking an unaligned memory access into a series of smaller aligned memory accesses.

12 Interrupts and exceptions

Cereon supports 6 interrupt types, summarized in the table below.

Interrupt no.	Name	Type	Source
`0`	`TIMER`	Asynchronous	Interrupt timer
`1`	`IO`	Asynchronous	I/O controllers
`2`	`SVC`	Synchronous	User program
`3`	`PROGRAM`	Synchronous	Program
`4`	`EXTERNAL`	Asynchronous	External source
`5`	`HARDWARE`	Asynchronous	Hardware fault

These interrupts are described in detail below.

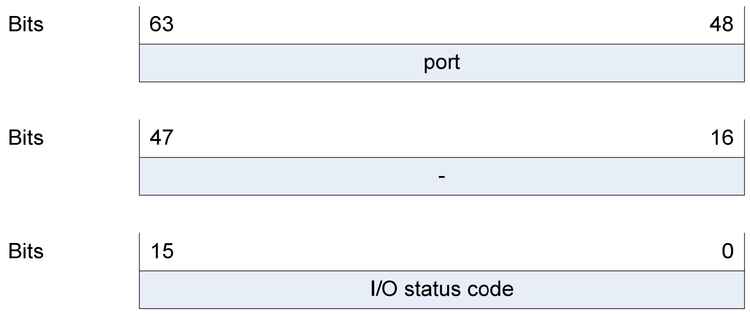
12.1 TIMER

This interrupt is generated when `$itc` reaches zero.

12.2 IO

The IO interrupt occurs when an I/O controller sends an interrupt signal to one of I/O ports is uses.

The associated Interrupt Status Code saved in the `$isc.io` register has the following format:



The meaning of individual fields within the `$isc.io` register is explained below.

12.2.1 Port

This field stores the 16-bit address of an I/O port where the `IO` interrupt has originated.

12.2.2 I/O status code

This field stores the 16-bit status code associated with the `IO` interrupt. This status code is set by whatever component has triggered the `IO` interrupt and can be used to determine its cause.

12.3 SVC

This interrupt is generated when a SVC instruction is executed.

The instruction pointer saved in `$isaveip.svc` when the `SVC` interrupt occurs refers to the instruction just after the `svc` instruction that has caused the SVC interrupt.

12.4 PROGRAM

This interrupt is generated when an instruction cannot be executed for some reason. Since this interrupt is always generated during an execution of an erroneous instruction (i.e. when the `$ip` has already been advanced to the next instruction), the `$isaveip.prg` register in an interrupt handler actually refers to the instruction which *follows* the one that has caused the `PROGRAM` interrupt.

The associated Interrupt Status Code saved in the `$isc.prg` register is a 64-bit value describing the cause of the `PROGRAM` interrupt (also known as an exception code). The list of exception codes and situations that lead to these exceptions is given in Appendix B to this document.

12.5 EXTERNAL

This interrupt is generated when an external signal arrives. Specifically, this occurs when a processor sends a signal to itself or another processor.

The associated Interrupt Status Code saved in the `$isc.prg` register is a 64-bit value describing the cause of the `EXTERNAL` interrupt. The higher 16 bits of this value always represent the external interrupt type, while the lower 48 bits may contain additional information about the cause of an external interrupt. The format and interpretation of this additional information is specific to a given external interrupt type. The list of external interrupt types and the rules for interpreting the additional interrupt information for each of these types is given in Appendix C to this document.

12.6 HARDWARE

This interrupt is generated when a hardware fault occurs.

The associated Interrupt Status Code saved in the `$isc.hw` register is a 64-bit value describing the cause of the `HARDWARE` interrupt. The list of hardware error codes and situations that lead to these errors being detected is given in Appendix D to this document.

12.7 Interrupt handlers

For each of the 6 supported interrupt types `<t>`, 4 control registers are used for handling interrupts of that type.

When an interrupt `<t>` occurs, the following steps are performed:

1. `$isaveip.<t> = $ip` (save instruction pointer).
2. `$isavestate.<t> = $state` (save processor state).

3. `$isc.<t> =` interrupt status code if one is available (this step is skipped for TIMER and SVC interrupts, as these have no associated status codes).
4. `$state = $eihstate.<t>` (change processor state to handle an interrupt).
5. `$ip = $iha.<t>` (jump to interrupt handler).

When returning from an interrupt `<t>` handler, the following steps are performed:

1. `$ip = $isaveip.<t>` (restore instruction pointer).
2. `$state = $isavestate.<t>` (restore processor state).

12.8 Interrupt masking

Bits 26..31 of the processor state `$state` contain interrupt mask for 6 supported interrupts (bit 26<=N<=31 corresponds to interrupt N-26). If the bit is 1, the corresponding interrupt is enabled; otherwise it is masked.

The following table summarizes the interrupt processing rules with regard to interrupt masking:

Interrupt	Reaction if masked
`TIMER`	The interrupt is postponed until such time as it becomes enabled.
`IO`	The interrupt is left pending until some processor core within the same processor is ready to handle it (i.e. it enables `IO` interrupt), some processor core within the same processor handles it by polling or some DMA channel within the same processor has started to handle it. Note, that the I/O port that has originated the interrupt will not be able to perform any further I/O until the IO interrupt is handled or the I/O port is reset.
`SVC`	The `MASKED` exception occurs.
`PROGRAM`	The processor is halted.
`EXTERNAL`	The behaviour depends on the type of an external interrupt. Generally, processors just ignore external interrupts when these are disabled. However, a sender of the external signal may choose to stall until the external signal has been accepted by the destination processor core.
`HARDWARE`	The processor is halted.

12.9 Synchronous and asynchronous interrupts

Synchronous interrupts are always the result of execution of some instruction. Asynchronous interrupts can occur at any time, as they originate from sources external to the processor.

Cereon delays processing of asynchronous interrupts until the beginning of the next instruction cycle.

13 I/O Subsystem

The connection between Cereon and the outside world consists of several layers:

- I/O ports, which provide the processor with means of communicating to the outside world.
- I/O controllers, which connect to I/O ports and manage I/O devices.
- I/O devices, which perform actual I/O operations.

13.1 Overview

This section provides a general overview of the Cereon I/O subsystem.

13.1.1 I/O ports

Each processor has a 16-bit I/O address space, which is independent of the main memory address space. The I/O address space provides 65536 independently addressable locations, any of which can be either unused or can have an I/O port attached there.

In a multiprocessor machine each processor has an independent set of I/O ports. It is, however, possible to connect I/O ports of different processors (or, indeed, different I/O ports of the same processor) to each other via a dedicated adapter (also known as port-to-port adapter), thus providing an additional datapath between these ports that bypasses the memory bus.

In a multi-core processors, there is a single I/O address space per processor, shared by all cores. All I/O instructions issued by any of these cores for a specific I/O port lock that I/O port for the duration of the I/O instruction execution, so two or more cores belonging to the same processor cannot perform I/O at the same I/O port at the same time.

When an I/O port is read from or written to, the size of the value being read or written (i.e. byte, half-word, word or long word) is made available to the I/O port in question. Some I/O ports may allow only values of a specific size to be read or written (e.g. byte ports of older I/O controllers will typically ignore all attempt to read or write multi-byte values), while other I/O ports may be able to handle values of different sizes.

13.1.2 I/O controllers

An I/O controller is a component that talks to a processor through one or more ports and, in doing so, manages some number of I/O devices. Each specific type of I/O device (or a closely related family of I/O device types) usually needs its own controller.

Depending on the model, an I/O controller can provide one or more ports, each port occupying one slot in the processor's I/O address space.

13.1.3 DMA channels

A DMA channel is a hardware component that allows data to be transferred between memory and I/O devices without the participation of the main processor core. This allows I/O to be performed in parallel with other work.

The exact number of available DMA channels is dependent on the model of processor; some smaller versions of Cereon processors may come entirely without any DMA channels. When DMA channels are present, each DMA channel presents an alternative datapath between the I/O ports of a specific processor and main memory bus. In a multiprocessor machine each processor has an independent set of DMA channels.

Like processor cores, DMA channels lock I/O ports they are currently accessing. Unlike processor cores, DMA channels can be instructed to perform a burst transfer of a sequence of bytes from or to a specific I/O port; in which case the I/O port remains locked for the duration of the burst.

From a technical point of view, a DMA channel is a simple peripheral processor that can execute DMA programs written in its own DMA instruction set. This instruction set is not related to the general Cereon instruction set understood by main processor cores, but is instead geared towards data transfer between memory and I/O ports.

A typical sequence of actions during DMA-assisted I/O is following:

- At some point during program execution, the processor decides to perform an I/O.
- The processor initiates the I/O by talking to an appropriate I/O controller through the I/O port where that controller is attached.
- The actual data transfer must occur some time after the I/O has been initiated. Depending on the I/O controller involved in the I/O, the point when the data transfer must commence can be immediately after I/O has been initiated, some known time after the I/O has been initiated, or a controller may generate an I/O interrupt to signal the processor that it is ready to transfer data. In all cases, processor can perform other tasks until it’s time to transfer data.
- The processor then selects an unused DMA channel and programs it to oversee the data transfer. DMA channels are programmed by means of dedicated DMA controller which to all purposes looks much like any other I/O controller – the only difference being that devices managed by the DMA controller are DMA channels.
- Once the DMA controller has been programmed and started, processor can go and do some other work. The DMA controller will transfer data between memory and I/O ports as specified by a dedicated DMA program. DMA programs can vary from a simple transfer of several bytes to complicated transfer of data from/to different memory areas with full error handling along the way.
- Once the DMA channel has finished (either because all the required data has been transferred and the DMA program has ended, or because an I/O error has been detected), the DMA channel raises an I/O interrupt of its own. This

signals the end of data transfer to the processor and allows it to take whatever actions are necessary to finalize the I/O (for example, to handle an I/O error).

All data transfer performed by DMA channels uses I/O ports of the processor to which these DMA channels belong. Therefore, from the point of view of an I/O controller attached to these I/O ports, it is impossible to tell whether data read from or written to a specific I/O port were read/written by a main processor or by a DMA channel impersonating that processor for I/O purposes. This significantly simplifies I/O port and controller logic.

13.2 I/O ports

Each Cereon processor has a 16-bit I/O address space, shared by all processor cores. Any of these addresses can have an actual I/O port behind it (attached I/O address), but in a typical processor most of I/O addresses will not have a corresponding I/O port (detached I/O addresses).

13.2.1 I/O port state

Unlike ports found in mainstream workstations, Cereon I/O ports keep information about their internal state. This information is:

- A 1-bit flag indicating whether the I/O port is allowed to interrupt processor cores (1) or not (0). When this flag is 0, I/O interrupts from the I/O port in question are not dispatched to running processor cores even if the latter are prepared to handle an `IO` interrupt. In this mode, the only way to detect and handle an I/O interrupt from the I/O port is for processor core or DMA channel to poll the I/O port for interrupt.
- A 1-bit flag indicating whether an I/O interrupt is currently pending in the I/O port (1) or not (0). If an I/O interrupt is pending in the I/O port, no data transmission through that I/O port is possible, any write to the I/O port is ignored and any read from the I/O port returns 0.
- A 16-bit interrupt status code, which identifies the cause of an I/O interrupt. When an I/O interrupt is handled (as a result of either `IO` interrupt occurring in a processor core, or I/O port being polled for interrupts), the handler can examine the I/O port’s interrupt status code to determine what caused an I/O interrupt.

Only one interrupt can be pending in any single I/O port at any given time. An attempt to raise another IO interrupt at the same I/O port causes the new I/O interrupt to become pending and the old one to be lost.

An attempt to obtain state information for an I/O address which does not have an I/O port behind it always returns the all-zero state (I/O interrupts disabled, no I/O interrupt pending, interrupt status code 0). An attempt to modify status information for such an I/O address is ignored.

13.2.2 I/O port assignment

The assignment of addresses to I/O ports is arbitrary and depends on the machine where Cereon processor is being used. The only exception to this rule is in that I/O addresses `0000`$_{16}$..`00FF`$_{16}$ (i.e. the lowest 256 I/O addresses) are reserved for the devices internal to the processor rather than external to it (such as DMA controller).

13.3 DMA channels

As already mentioned, a DMA channel is a small special-purpose processor core housed within the processor alongside main processor cores. Unlike these main processor cores, a DMA channel cannot perform the full set of operations on data; however, the facilities a DMA channel provides are sufficient to start and maintain data transfer between main memory and I/O ports, perform simple error checking and effect simple I/O control logic if necessary.

13.3.1 Programmer's model

13.3.1.1 Registers

The DMA channel has 6 64-bit registers:

- The `$ip` (instruction pointer) register holds the address of a channel instruction to be executed next.
- The `$state` register holds the current state of a DMA channel.
- The 4 general purpose registers `r0`..`r3` can be used to hold memory addresses, I/O port addresses, data transferred between memory and I/O devices, or any other information. DMA channels always use physical memory addresses, whether or not some, or all, of the main processor cores are in Real or Virtual mode.

Note that, although names of DMA registers are the same as names of some general purpose or control registers, no confusion arises; each register name is interpreted depending whether it occurs in a "normal" instruction or in a DMA channel instruction.

13.3.1.2 Accessing registers

Registers of any single DMA channel are mapped onto a 8-byte area within processor's I/O address space. This mapping is performed as follows (offsets are given in bytes, starting from the lowest I/O address):

Offset	0	1	2	3	4	5	6	7
	-		$ip	$state	r0	r1	r2	r3

A single processor can host up to 4 independent DMA channels, whose I/O areas are, respectively 0..7, 8..15, 16..23 and 24..31. So, for example, to access the register `r0` of DMA channel 1, the long word shall be read from or written to I/O port 12.

Other that being assigned to DMA channel registers, the I/O ports in range 0..31 are no different from other I/O ports. In particular, I/O interrupts can be raised on any of these I/O ports, which is a standard method of signalling to processor core(s) that a DMA channel has finished its work or has detected an I/O error.

If one of the DMA channels 0..3 is not installed, its `$state` register always reads $\texttt{0000000000000000}_{16}$.

13.3.1.3 DMA channel state

The `$state` register of a DMA channel has the following format:

Bits	63 2	2	1	0
	-	W	B	P

The meaning of individual fields within the `$state` register is explained below.

13.3.1.3.1 P (Present)

This bit is hardwired to 1 for each existing DMA channel. Its presence allows the processor core(s) to determine whether a specific DMA channel 0..3 exists by reading from the corresponding `$state` register and checking whether its value is zero (DMA channel does not exist) or nonzero (DMA channel exists).

13.3.1.3.2 B (Big-endian)

When this bit is 1, the DMA channel uses the big-endian byte ordering when transferring data to and from memory; otherwise the little-endian byte ordering is used. These bits can be hardwired if a byte order of a DMA channel cannot be changed; in this case the byte order of the main processor core(s) is usually hardwired to the same value.

13.3.1.3.3 W (Working)

When this bit is 1, the DMA is in Working mode; otherwise it is in Idle mode. When in Working mode, DMA channel fetches and executes channel instructions. When in Idle mode, DMA channel does nothing.

This bit can be not only read (to determine whether a DMA channel is currently busy) but written to as well. Changing this bit from 0 to 1 causes the DMA channel to start executing a channel program from the current `$ip` address, which is a standard way of starting a DMA-assisted I/O. Changing this bit from 1 to 0 causes the DMA channel to stop immediately, which is a standard way of terminating a DMA-assisted I/O before its completion.

13.3.2 Channel programs

A channel program is a sequence of channel instructions that can be executed by a DMA channel. Each channel instruction is always 4 bytes long and must be aligned at a 4-byte boundary. When fetched by a DMA channel, the instruction is read as a single 32-bit value in either big-endian or little-endian mode, depending on the current byte order used by the DMA channel.

13.3.2.1 dma.stop (DMA stop)

Bits	31 24	23 0
	00000000	-

Sets the `W` bit of the DMA `$state` register to 0, effectively stopping the DMA channel. Then raises a DMA interrupt to signal the fact to the processor core(s). See the “DMA interrupts” section for more information about DMA interrupts.

13.3.2.2 dma.t.b (DMA transfer bytes)

Bits	31 24	23 20	19 16	15 0
	00000001	dst	src	count

This instruction performs a burst transfer of `count` bytes from the source `src` to destination `dst`.

Both `src` and `dst` are 4-bit fields that specify operand type 0..3 in the upper 2 bits and a register number 0..3 in the lower 2 bits. The register number always refers to one of the DMA channel’s registers `r0`..`r3`.

13.3.2.2.1 00_2 – I/O port

If an operand type is 00_2, the lower 16 bits of the specified register refer to a byte I/O port. When such operand is used as a source, a byte is read from the corresponding I/O port. When such operand is used as a destination, a byte is written to the corresponding I/O port.

13.3.2.2.2 01_2 – Memory

If an operand type is 01_2, the specified register refers to a physical memory address. When such operand is used as a source, a byte is read from that physical memory address. When such operand is used as a destination, a byte is written to that physical memory address.

13.3.2.2.3 10_2 – Memory post-increment

If an operand type is 10_2, the specified register refers to a physical memory address and is incremented by 1 after each memory access. When such operand is used as a source, a byte is read from that physical memory address and the DMA register is incremented by 1. When such operand is used as a destination, a byte is written to that physical memory address and the DMA register is incremented by 1.

13.3.2.2.4 11_2 – Memory pre-decrement

If an operand type is 11_2, the specified register refers to a physical memory address and is decremented by 1 before each memory access. When such operand is used as a source, the DMA register is decremented by 1 and a byte is read from that physical memory address. When such operand is used as a destination, the DMA register is decremented by 1 and a byte is written to that physical memory address.

13.3.2.2.5 Operand type combinations

The most common combination of operand types is when one of them is an I/O port and another is a post-incremented memory address. In this configuration, data is transferred from a continuous memory area to an I/O port or from an I/O port to a continuous memory area.

Nevertheless, any combination of operand types is permitted. For example, having both operands to refer to I/O ports allows transferring data from one I/O device to another without having to store the data in memory. Similarly, having both operands to refer to a post-incremented (or pre-decremented) memory addresses will cause the DMA channel to copy one memory area into another (in either forward or reverse order) without any drain on main processor core resources.

A special case of operand type combination is when both operands are post-incremented and/or pre-decremented and they both refer to the same DMA register. In this case the side effects caused by the source operand are guaranteed to occur before side effects caused by the destination operand. For example, if destination is a post-incremented register `r0` and source is a pre-decremented register `r0`, then `r0` will be decremented before reading a byte, then post-decremented after writing a byte, which has the effect of copying a byte from memory into itself.

13.3.2.3 dma.t.h (DMA transfer half-words)

Bits	31 24	23 20	19 16	15 0
	00000010	dst	src	count

This instruction performs a burst transfer of `count` half-words from the source `src` to destination `dst`.

Both `src` and `dst` are 4-bit fields that specify operand type 0..3 in the upper 2 bits and a register number 0..3 in the lower 2 bits. The rules for the interpretation of operand types are the same as those used by `dma.t.b` instruction, except post-increment and pre-decrement operand types cause the address register to be incremented or decremented by 2 (a size of a half-word) instead of 1.

13.3.2.4 dma.t.w (DMA transfer words)

Bits	31 24	23 20	19 16	15 0
	00000011	dst	src	count

This instruction performs a burst transfer of `count` words from the source `src` to destination `dst`.

Both `src` and `dst` are 4-bit fields that specify operand type 0..3 in the upper 2 bits and a register number 0..3 in the lower 2 bits. The rules for the interpretation of operand types are the same as those used by `dma.t.b` instruction, except post-increment and pre-decrement operand types cause the address register to be incremented or decremented by 4 (a size of a word) instead of 1.

13.3.2.5 dma.t.l (DMA transfer long words)

Bits	31 24	23 20	19 16	15 0
	00000100	dst	src	count

This instruction performs a burst transfer of `count` long words from the source `src` to destination `dst`.

Both `src` and `dst` are 4-bit fields that specify operand type 0..3 in the upper 2 bits and a register number 0..3 in the lower 2 bits. The rules for the interpretation of operand types are the same as those used by `dma.t.b` instruction, except post-increment and pre-decrement operand types cause the address register to be incremented or decremented by 8 (a size of a word) instead of 1.

13.3.2.6 dma.vt.b (DMA variable-length transfer bytes)

Bits	31 24	23 20	19 16	15 14	13 0
	00000101	dst	src	len	00000000000000

This instruction is similar to `dma.t.b` except the number of bytes to transfer is specified as a contents of a DMA register `len` (0..3) instead of a constant. Unlike `dma.t.b`, the `dma.vt.b` instruction allows to specify a transfer of more than 2^{16} bytes in one instruction.

13.3.2.7 dma.vt.h (DMA variable-length transfer half-words)

Bits	31 24	23 20	19 16	15 14	13 0
	00000110	dst	src	len	00000000000000

This instruction is similar to `dma.th` except the number of half-words to transfer is specified as a contents of a DMA register `len` (0..3) instead of a constant. Unlike `dma.t.h`, the `dma.vt.h` instruction allows to specify a transfer of more than 2^{16} half-words in one instruction.

13.3.2.8 dma.vt.w (DMA variable-length transfer words)

Bits	31 24	23 20	19 16	15 14	13 0
	00000111	dst	src	len	00000000000000

This instruction is similar to `dma.tw` except the number of words to transfer is specified as a contents of a DMA register `len` (0..3) instead of a constant. Unlike `dma.t.w`, the `dma.vt.w` instruction allows to specify a transfer of more than 2^{16} words in one instruction.

13.3.2.9 dma.vt.l (DMA variable-length transfer long words)

Bits	31 24	23 20	19 16	15 14	13 0
	00001000	dst	src	len	00000000000000

This instruction is similar to `dma.t.l` except the number of long words to transfer is specified as a contents of a DMA register `len` (0..3) instead of a constant. Unlike dma.t.l, the dma.vt.l instruction allows to specify a transfer of more than 2^{16} long words in one instruction.

13.3.2.10 dma.l.b (DMA load byte)

Bits	31 24	23 22	21 18	17 0
	00001001	dst	src	000000000000000000

This instruction reads a single byte from the source `src`, sign-extends it to 64 bits and places the result into the DMA register `dst`.

The 4-bit field `src` specifies the operand type and DMA register using the same rules as `dma.t.b`.

13.3.2.11 dma.l.ub (DMA load unsigned byte)

Bits	31 24	23 22	21 18	17 0
	00001010	dst	src	000000000000000000

This instruction reads a single byte from the source `src`, zero-extends it to 64 bits and places the result into the DMA register `dst`.

The 4-bit field `src` specifies the operand type and DMA register using the same rules as `dma.t.b`.

13.3.2.12 dma.l.h (DMA load half-word)

Bits	31 24	23 22	21 18	17 0
	00001011	dst	src	000000000000000000

This instruction reads a half-word from the source `src`, sign-extends it to 64 bits and places the result into the DMA register `dst`.

The 4-bit field `src` specifies the operand type and DMA register using the same rules as `dma.t.h`.

13.3.2.13 dma.l.uh (DMA load unsigned half-word)

Bits	31 24	23 22	21 18	17 0
	00001100	dst	src	000000000000000000

This instruction reads a half-word from the source `src`, zero-extends it to 64 bits and places the result into the DMA register `dst`.

The 4-bit field `src` specifies the operand type and DMA register using the same rules as `dma.t.h`.

13.3.2.14 dma.l.w (DMA load word)

Bits	31 24	23 22	21 18	17 0
	00001101	dst	src	000000000000000000

This instruction reads a word from the source `src`, sign-extends it to 64 bits and places the result into the DMA register `dst`.

The 4-bit field `src` specifies the operand type and DMA register using the same rules as `dma.t.w`.

13.3.2.15 dma.l.uw (DMA load unsigned word)

Bits	31 24	23 22	21 18	17 0
	00001110	dst	src	000000000000000000

This instruction reads a word from the source `src`, zero-extends it to 64 bits and places the result into the DMA register `dst`.

The 4-bit field `src` specifies the operand type and DMA register using the same rules as `dma.t.w`.

13.3.2.16 dma.l.l (DMA load long word)

Bits	31 24	23 22	21 18	17 0
	00001111	dst	src	000000000000000000

This instruction reads a long word from the source `src` into the DMA register `dst`.

The 4-bit field `src` specifies the operand type and DMA register using the same rules as `dma.t.l`.

13.3.2.17 dma.s.b (DMA store byte)

Bits	31 24	23 20	19 18	17 0
	00010000	dst	src	000000000000000000

This instruction stores the lower byte of a DMA register `src` to the destination `dst`.

The 4-bit field `dst` specifies the operand type and DMA register using the same rules as `dma.t.b`.

13.3.2.18 dma.s.h (DMA store half-word)

Bits	31 24	23 20	19 18	17 0
	00010001	dst	src	000000000000000000

This instruction stores the lower half-word of a DMA register `src` to the destination `dst`.

The 4-bit field `dst` specifies the operand type and DMA register using the same rules as `dma.t.h`.

13.3.2.19 dma.s.w (DMA store word)

Bits	31 24	23 20	19 18	17 0
	00010010	dst	src	000000000000000000

This instruction stores the lower word of a DMA register `src` to the destination `dst`.

The 4-bit field `dst` specifies the operand type and DMA register using the same rules as `dma.t.w`.

13.3.2.20 dma.s.l (DMA store long word)

Bits	31 24	23 20	19 18	17 0
	00010011	dst	src	000000000000000000

This instruction stores the DMA register `src` to the destination `dst`.

The 4-bit field `dst` specifies the operand type and DMA register using the same rules as `dma.t.l`.

13.3.2.21 dma.li.l (DMA load immediate long word)

Bits	31 24	23 22	21 0
	00010100	dst	immediate

This instruction sign-extends the `immediate` value to 64 bits and places the result into the DMA register `dst`.

13.3.2.22 dma.add.l (DMA add long word)

Bits	31 24	23 22	21 20	19 18	17 0
	00010101	dst	op1	op2	000000000000000000

This instruction adds the contents of DMA registers `op1` and `op2` and places the result into DMA register `dst`. An overflow, if one occurs, is ignored with a wrap-around.

13.3.2.23 dma.sub.l (DMA subtract long word)

Bits	31 24	23 22	21 20	19 18	17 0
	00010110	dst	op1	op2	000000000000000000

This instruction subtracts the contents of DMA register `op2` from DMA register `op1` and places the result into DMA register `dst`. An overflow, if one occurs, is ignored with a wrap-around.

13.3.2.24 dma.and.l (DMA bitwise and long word)

Bits	31 24	23 22	21 20	19 18	17 0
	00010111	dst	op1	op2	000000000000000000

This instruction performs a bitwise logical AND over the contents of DMA registers `op1` and `op2` and places the result into DMA register `dst`.

13.3.2.25 dma.or.l (DMA bitwise or long word)

Bits	31 24	23 22	21 20	19 18	17 0
	00011000	dst	op1	op2	000000000000000000

This instruction performs a bitwise logical OR over the contents of DMA registers `op1` and `op2` and places the result into DMA register `dst`.

13.3.2.26 dma.xor.l (DMA bitwise exclusive or long word)

Bits	31 24	23 22	21 20	19 18	17 0
	00011001	dst	op1	op2	-

This instruction performs a bitwise logical Exclusive Or over the contents of DMA registers `op1` and `op2` and places the result into DMA register `dst`.

13.3.2.27 dma.addi.l (DMA add immediate long word)

Bits	31 24	23 22	21 20	19 0
	00011010	dst	op1	immediate

This instruction adds the contents of DMA register `op1` and an `immediate` value and places the result into DMA register `dst`. An overflow, if one occurs, is ignored with a wrap-around. The `immediate` value is sign-extended to 64 bits before addition.

13.3.2.28 dma.subi.l (DMA subtract immediate long word)

Bits	31 24	23 22	21 20	19 0
	00011011	dst	op1	immediate

This instruction subtracts the `immediate` value from DMA register `op1` and places the result into DMA register `dst`. An overflow, if one occurs, is ignored with a wrap-around. The `immediate` value is sign-extended to 64 bits before subtraction.

13.3.2.29 dma.andi.l (DMA and immediate long word)

Bits	31 24	23 22	21 20	19 0
	00011100	dst	op1	immediate

This instruction performs a bitwise logical AND over the contents of DMA register `op1` and `immediate` value and places the result into DMA register `dst`. The `immediate` value is zero-extended to 64 bits before conjunction.

13.3.2.30 dma.ori.l (DMA or immediate long word)

Bits	31 24	23 22	21 20	19 0
	00011101	dst	op1	immediate

This instruction performs a bitwise logical OR over the contents of DMA register `op1` and `immediate` value and places the result into DMA register `dst`. The `immediate` value is zero-extended to 64 bits before disjunction.

13.3.2.31 dma.xori.l (DMA exclusive or immediate long word)

Bits	31 24	23 22	21 20	19 0
	00011110	dst	op1	immediate

This instruction performs a bitwise logical Exclusive OR over the contents of DMA register `op1` and `immediate` value and places the result into DMA register `dst`. The `immediate` value is zero-extended to 64 bits before exclusive disjunction.

13.3.2.32 dma.neg.l (DMA negate long word)

Bits	31 24	23 22	21 20	19 0
	00011111	dst	op1	00000000000000000000

This instruction stores the 2's complement of the contents of DMA register `op1` and into DMA register `dst`.

13.3.2.33 dma.not.l (DMA not long word)

Bits	31 24	23 22	21 20	19 0
	00100000	dst	op1	00000000000000000000

This instruction stores the 1's complement of the contents of DMA register `op1` and into DMA register `dst`.

13.3.2.34 dma.tstp (DMA Test Port status)

Bits	31 24	23 22	21 20	19 0
	00100001	dst	port	00000000000000000000

Using the lower 16 bits of DMA register `port` as an I/O address, tests the current state of the I/O port at that address. The contents of DMA register `dst` is then modified as follows:

- Bit 0 (lowest) is set to 1 is there is an I/O port at the specified I/O address, or to 0 if there is no I/O port at the specified I/O address.
- Bit 1 is set to 1 if there is a pending interrupt in the I/O port at the specified I/O address, or to 0 if there is no pending interrupt there.
- Bit 2 is set to a if I/O interrupts are currently enabled in the I/O port at the specified address, or to 0 if they are disabled.
- Bits 3..15 and 32..63 are set to 0.
- If there is a pending interrupt in the port, bits 16..31 are set to the interrupt status code pending on the port and the interrupt is released (i.e. no longer pending on the I/O port). If there is no pending interrupt there, bits 16..31 are set to 0.

13.3.2.35 dma.setp (DMA Set Port status)

Bits	31 24	23 22	21 20	19 0
	00100010	src	port	00000000000000000000

Using the lower 16 bits of DMA register `port` as an I/O address, sets the current state of the I/O port at that address. The contents of DMA register `src` is used to determine how the port state shall be set:

- If bit 1 is 1, the interrupt is made pending on the I/O port. Bits 16..31 of the DMA register `src` are used as an interrupt status code. If this bit is 0, the instruction has no effect on whether or not an interrupt is pending on the I/O port.
- If bit 2 is 1, I/O interrupts are enabled in the I/O port at the specified address. If this bit is 0, I/O interrupts are disabled.
- Bits 0, 3..15 and 32..63 are ignored.

13.3.2.36 dma.j (DMA Jump)

Bits	31 24	23 0
	00100011	offset

This instruction adds the value (`offset << 2`) to the DMA register `$ip`, effectively causing an unconditional jump. The `offset` is sign-extended to 64 bits before addition.

13.3.2.37 dma.beq.l (DMA branch on equal long word)

Bits	31 24	23 22	21 20	19 0
	00100100	op1	op2	offset

This instruction compares the values of DMA registers `op1` and `op2` and, if they are found to be equal, adds the value (`offset << 2`) to the DMA register `$ip`. The `offset` is sign-extended to 64 bits before addition.

13.3.2.38 dma.bne.l (DMA branch on not equal long word)

Bits	31 24	23 22	21 20	19 0
	00100101	op1	op2	offset

This instruction compares the values of DMA registers `op1` and `op2` and, if they are found to be not equal, adds the value (`offset << 32` to the DMA register `$ip`. The `offset` is sign-extended to 64 bits before addition.

13.3.2.39 dma.blt.l (DMA branch on less than)

Bits	31 24	23 22	21 20	19 0
	00100110	op1	op2	offset

This instruction compares the values of DMA registers `op1` and `op2` and, if the first is less than the second, adds the value (`offset << 2`) to the DMA register `$ip`. The `offset` is sign-extended to 64 bits before addition. Operands are compared as signed integer values.

13.3.2.40 dma.ble.l (DMA branch on less than or equal long word)

Bits	31 24	23 22	21 20	19 0
	00100111	op1	op2	offset

This instruction compares the values of DMA registers `op1` and `op2` and, if the first is less than or equal to the second, adds the value (`offset << 2`) to the DMA register `$ip`. The `offset` is sign-extended to 64 bits before addition. Operands are compared as signed integer values.

13.3.2.41 dma.bgt.l (DMA branch on greater than long word)

Bits	31 24	23 22	21 20	19 0
	00101000	op1	op2	offset

This instruction compares the values of DMA registers `op1` and `op2` and, if the first is greater than the second, adds the value (`offset << 2`) to the DMA register `$ip`. The `offset` is sign-extended to 64 bits before addition. Operands are compared as signed integer values.

13.3.2.42 dma.bge.l (DMA branch on greater than or equal long word)

Bits	31 24	23 22	21 20	19 0
	00101001	op1	op2	offset

This instruction compares the values of DMA registers `op1` and `op2` and, if the first is greater than or equal to the second, adds the value (`offset << 2`) to the DMA register `$ip`. The `offset` is sign-extended to 64 bits before addition. Operands are compared as signed integer values.

13.3.2.43 dma.blt.ul (DMA branch on less than unsigned long word)

Bits	31 24	23 22	21 20	19 0
	00101010	op1	op2	offset

This instruction compares the values of DMA registers `op1` and `op2` and, if the first is less than the second, adds the value (`offset << 2`) to the DMA register `$ip`. The `offset` is sign-extended to 64 bits before addition. Operands are compared as unsigned integer values.

13.3.2.44 dma.ble.ul (DMA branch on less than or equal unsigned long word)

Bits	31 24	23 22	21 20	19 0
	00101011	op1	op2	offset

This instruction compares the values of DMA registers `op1` and `op2` and, if the first is less than or equal to the second, adds the value (`offset << 2`) to the DMA register `$ip`. The `offset` is sign-extended to 64 bits before addition. Operands are compared as unsigned integer values.

13.3.2.45 dma.bgt.ul (DMA branch on greater than unsigned long word)

Bits	31 24	23 22	21 20	19 0
	00101100	op1	op2	offset

This instruction compares the values of DMA registers `op1` and `op2` and, if the first is greater than the second, adds the value (`offset << 2`) to the DMA register `$ip`. The `offset` is sign-extended to 64 bits before addition. Operands are compared as unsigned integer values.

13.3.2.46 dma.bge.ul (DMA branch on greater than or equal unsigned long word)

Bits	31 24	23 22	21 20	19 0
	00101101	op1	op2	offset

This instruction compares the values of DMA registers `op1` and `op2` and, if the first is greater than or equal to the second, adds the value (`offset << 2`) to the DMA register `$ip`. The `offset` is sign-extended to 64 bits before addition. Operands are compared as unsigned integer values.

13.3.2.47 dma.shl.l (DMA shift left long word)

Bits	31 24	23 22	21 20	19 18	17 0
	00101110	dst	op1	op2	000000000000000000

This instruction shifts the contents of DMA register `op1` left by the number of bits specified by the DMA register `op2` and places the result into DMA register `dst`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0.

13.3.2.48 dma.shr.l (DMA shift right long word)

Bits	31 24	23 22	21 20	19 18	17 0
	00101111	dst	op1	op2	000000000000000000

This instruction shifts the contents of DMA register `op1` right by the number of bits specified by the DMA register `op2` and places the result into DMA register `dst`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0.

13.3.2.49 dma.asl.l (DMA arithmetic shift left long word)

Bits	31 24	23 22	21 20	19 18	17 0
	00110000	dst	op1	op2	000000000000000000

This instruction shifts the contents of the lower 63 bits of the DMA register `op1` left by the number of bits specified by the DMA register `op2` and places the result into DMA register `dst`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift left are set to 0.

13.3.2.50 dma.asr.l (DMA arithmetic shift right long word)

Bits	31 24	23 22	21 20	19 18	17 0
	00110001	dst	op1	op2	000000000000000000

This instruction shifts the contents of DMA register `op1` right by the number of bits specified by the DMA register `op2` and places the result into DMA register `dst`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift right are set to the copy of the original sign bit of DMA register `op1`.

13.3.2.51 dma.rol.l (DMA rotate left long word)

Bits	31 24	23 22	21 20	19 18	17 0
	00110010	dst	op1	op2	-

This instruction rotates the contents of DMA register `op1` left by the number of bits specified by the DMA register `op2` and places the result into DMA register `dst`. The shift counter is treated as an unsigned integer quantity with all bits significant; negative shift counter causes rotation in the opposite direction. Bits are rotated, i.e. each new bit introduced at one end of the shifted value is a copy of the bit pushed out of the other end by the shift.

13.3.2.52 dma.ror.l (DMA rotate right long word)

Bits	31 24	23 22	21 20	19 18	17 0
	00110011	dst	op1	op2	-

This instruction rotates the contents of DMA register `op1` right by the number of bits specified by the DMA register `op2` and places the result into DMA register `dst`. The shift counter is treated as an unsigned integer quantity with all bits significant; negative shift counter causes rotation in the opposite direction. Bits are rotated, i.e. each new bit introduced at one end of the shifted value is a copy of the bit pushed out of the other end by the shift.

13.3.2.53 dma.shli.l (DMA shift left long word immediate)

Bits	31 24	23 22	21 20	19 6	5 0
	00110100	dst	op1	00000000000000	imm

This instruction shifts the contents of DMA register `op1` left by the number of bits specified by the `imm` and places the result into DMA register `dst`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0.

13.3.2.54 dma.shri.l (DMA shift right long word immediate)

Bits	31 24	23 22	21 20	19 6	5 0
	00110101	dst	op1	00000000000000	imm

This instruction shifts the contents of DMA register `op1` right by the number of bits specified by the `imm` and places the result into DMA register `dst`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift are set to 0.

13.3.2.55 dma.asli.l (DMA arithmetic shift left long word immediate)

Bits	31 24	23 22	21 20	19 6	5 0
	00110110	dst	op1	00000000000000	imm

This instruction shifts the contents of the lower 63 bits of the DMA register `op1` left by the number of bits specified by the `imm` and places the result into DMA register `dst`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift left are set to 0.

13.3.2.56 dma.asri.l (DMA arithmetic shift right long word immediate)

Bits	31 24	23 22	21 20	19 6	5 0
	00110111	dst	op1	00000000000000	imm

This instruction shifts the contents of DMA register `op1` right by the number of bits specified by the `imm` and places the result into DMA register `dst`. The shift counter is treated as an unsigned integer quantity with all bits significant. New bits introduced by shift right are set to the copy of the original sign bit of DMA register `op1`.

13.3.2.57 dma.roli.l (DMA rotate left long word immediate)

Bits	31 24	23 22	21 20	19 6	5 0
	00111000	dst	op1	00000000000000	imm

This instruction rotates the contents of DMA register `op1` left by the number of bits specified by the `imm` and places the result into DMA register `dst`. The shift counter is treated as an unsigned integer quantity with all bits significant; negative shift counter causes rotation in the opposite direction. Bits are rotated, i.e. each new bit introduced at one end of the shifted value is a copy of the bit pushed out of the other end by the shift.

13.3.2.58 dma.rori.l (DMA rotate right long word immediate)

Bits	31 24	23 22	21 20	19 6	5 0
	00111001	dst	op1	00000000000000	imm

This instruction rotates the contents of DMA register `op1` right by the number of bits specified by the `imm` places the result into DMA register `dst`. The shift counter is treated as an unsigned integer quantity with all bits significant; negative shift counter causes rotation in the opposite direction. Bits are rotated, i.e. each new bit introduced at one end of the shifted value is a copy of the bit pushed out of the other end by the shift.

13.3.3 DMA interrupts

A DMA interrupt is an I/O interrupt on an I/O port assigned to a DMA channel that signals an unusual or erroneous condition detected during execution of a channel program. DMA interrupts for a DMA channel are always raised on the lowest-address I/O port assigned to that DMA channel (i.e. I/O port 0 for DMA channel 0, I/O port 8 for DMA channel 1, etc.)

The interrupt status code of a DMA interrupt reflects upon the cause of the DMA interrupt. By convention, the interrupt status code 0 means “channel program completed successfully”, while other interrupt status codes signify a premature termination of a channel program due to some error.

The full list of errors that can occur during channel program execution, along with associated interrupt status codes, can be found in an Appendix E to this document.

14 The on-chip memory

The On-Chip Memory (henceforth abbreviated as OCM) provides an ability to assign one or more regions within a physical 64-bit memory address space to a high-performance integrated memory. From the program's point of view, these regions can be accessed just like any "normal" memory; however, OCM offers a much smaller worst-case latency (such as 1 or 2 CPU cycles against the 6+ bus cycles of the "main" memory), as it operates at CPU core speed and not at a memory bus speed (typically, OCM memory will be embedded into the processor chip; hence the name "On-Chip Memory"). In a typical real-time system, OCM will contain interrupt handlers as well as time-critical code and data, thus reducing the memory latency in using these artefacts.

In effect, assigning an OCM memory block to a specific region within a processor's address space means that all accesses (including instruction fetches, as well as data loads and stores) made to this region will access OCM instead of the "real" memory. The price to pay for this illusion is in that an OCM block cannot, whether wholly or partially, overlap physical memory addresses where "real" memory is available. As a consequence, each byte of memory that exists is either in cache or OCM, but never both.

Note that, by definition, OCM lives "in parallel" with the caches (if ones are provided by the particular Cereon model), so OCM is never cached, in order to make memory access times more predictable.

In a multi-core configuration, there is one OCM per processor, shared by all cores. In a multi-processor configuration, each processor has its own OCM. The latter means that shared data (in particular, lock variables used by the `xchg` instruction) shall never be placed into OCM.

14.1 OCM and virtual mode

The OCM memory, if used, is always mapped into continuous regions of a physical address space. Among other things, this means that a Cereon core equipped with either protected memory or virtual memory feature will always perform a virtual address translation and will then access the OCM only if the translated real address falls within an OCM range.

This strategy allows the same OCM block (containing, for example, critical data or interrupt handlers) to be shared between virtual address spaces of multiple processes, serving all of them.

14.2 OCM controller

An OCM controller is an on-chip component that allows both examining the available OCM configuration and mapping OCM blocks into the physical address space. To the processor, it looks just like any other I/O controller (specifically, it must be accessed through I/O ports provided by the processor's I/O space). However, being an internal device to a chip, the OCM controller occupies the predefined internal range within the processor's I/O space (namely, I/O ports 32..56); individual I/O ports within this range are assigned to the following registers of the OCM controller:

Port	Register	Size	Direction	Usage
32	`$banksize`	Long word	Read only	Available OCM size
33	`$start0`	Long word	Read/write	Region 0 start
34	`$offset0`	Long word	Read/write	Region 0 offset
35	`$size0`	Long word	Read/write	Region 0 offset
36	`$start1`	Long word	Read/write	Region 1 start
37	`$offset1`	Long word	Read/write	Region 1 offset
38	`$size1`	Long word	Read/write	Region 1 size
39	`$start2`	Long word	Read/write	Region 2 start
40	`$offset2`	Long word	Read/write	Region 2 offset
41	`$size2`	Long word	Read/write	Region 2 size
42	`$start3`	Long word	Read/write	Region 3 start
43	`$offset3`	Long word	Read/write	Region 3 offset
44	`$size3`	Long word	Read/write	Region 3 size
45	`$start4`	Long word	Read/write	Region 4 start
46	`$offset4`	Long word	Read/write	Region 4 offset
47	`$size4`	Long word	Read/write	Region 4 size
48	`$start5`	Long word	Read/write	Region 5 start
49	`$offset5`	Long word	Read/write	Region 5 offset
50	`$size5`	Long word	Read/write	Region 5 size
51	`$start6`	Long word	Read/write	Region 6 start
52	`$offset6`	Long word	Read/write	Region 6 offset
53	`$size6`	Long word	Read/write	Region 6 size
54	`$start7`	Long word	Read/write	Region 7 start
55	`$offset7`	Long word	Read/write	Region 7 offset
56	`$size7`	Long word	Read/write	Region 7 size

Since I/O operations are not permitted in User mode, only trusted system code (running in Kernel mode) can examine or modify the current OCM configuration (including its presense). However, once mapped into a physical address space, OCM can be used to contain both system and user code and/or data.

14.3 OCM concepts

This section outlines the main concepts used by the OCM.

14.3.1 OCM bank

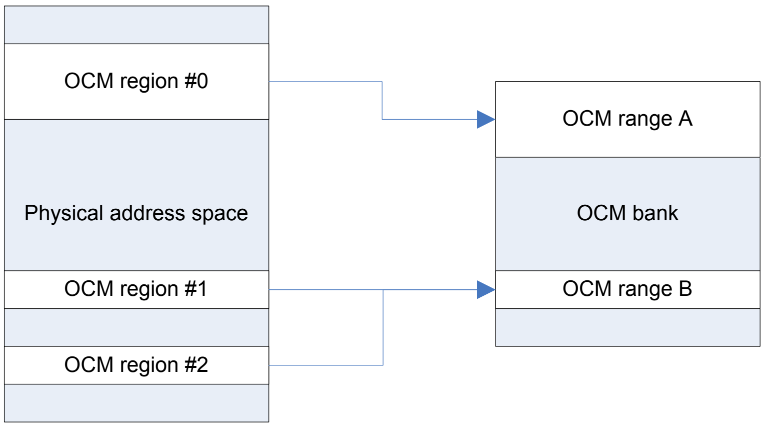
An OCM bank is a total OCM memory available to the processor.

To determine the size of the OCM bank, a core must read the value of the `$banksize` register (by reading a long word from I/O port 32). The value of 0 means the processor containing the core in question is not equipped with OCM; any other value gives the size, in bytes, of the available OCM.

14.3.2 OCM regions

An OCM region is a continuous range of physical address space where OCM memory has been mapped. Any attempts to read or write from the OCM region will cause reading or writing of the OCM bank. It is possible to define and use up to 8 OCM regions (provided, that is, that the processor is equipped with OCM).

A single OCM region acts as a “window” into the OCM memory:



As illustrated, each OCM region, if defined, is mapped into a continuous area within the OCM bank. The size of that area, naturally, is the same as the size of the region. Note also that it is possible for different regions to be mapped into OCM bank ranges that partially or fully overlap.

The net effect achieved by OCM regions is very similar to that provided by the Protected Memory feature, except the latter provides a region-based mapping from virtual address space to physical address space of an equal size, whereas the former provides a region-based mapping from physical address space to OCM address space of much smaller size.

14.3.3 OCM and memory protection

Unlike Protected Memory feature, which it closely resembles, OCM itself does not provide any memory protection features. This is only natural, given that OCM is a fast stand-in for physical memory, which has no built-in protection.

Therefore, protection of OCM memory is achieved by the same higher-level mechanisms that are used for protecting the “real” memory – when a processor core operating in Virtual mode performs virtual address translation that would have resulted in a physical address within an OCM region, access rights are checked as usual, resulting in either `IACCESS` or `DACCESS` exception if necessary.

14.3.4 OCM and data alignment

All OCM regions must obey the following restrictions:

- The start address of an OCM region must be a multiple of 8.
- The start offset of an OCM range within an OCM bank must be a multiple of 8.
- The size of an OCM region must be a multiple of 8.

The above restrictions effectively mean that all naturally aligned accesses to an OCM region become naturally aligned accesses to an OCM bank.

14.4 OCM operation

For each OCM region `N` (`0 ≤ N ≤ 7`), there are three OCM controller registers describing that region:

- `$startN` provides the physical address where the region starts.
- `$offsetN` specifies the offset within the OCM bank where the actual OCM memory starts.
- `$sizeN` provides the region size.

The net effect is in that when a processor tries to access memory address `A` in range `[$startN .. $startN + $sizeN)` (for some `N`, `0 ≤ N ≤ 7`), the OCM memory at offset `$offsetN + (A – $startN)` is accessed instead.

All of these registers can be read and written. Reading from these registers allows determining the current OCM configuration; writing to them remaps the corresponding OCM region.

14.4.1 Region control registers

The following diagram illustrates the format of the region control registers for a single region `N`:

	Bits	63	2 0
$startN		8 highest bits of the physical address	000

	Bits	63	2 0
$offsetN		8 highest bits of the OCM bank offset	000

	Bits	63	2 0
$sizeN		8 highest bits of the region’s size	000

Note that the start address, OCM offset and size of a region are all multiples of 8. Any attempt to assign to one of these registers a value whose three lower bits are not all zero results in ignoring these three nonzero lower bits and assigning the rest of the value.

14.4.2 Address overflows and wraparounds

When translating a physical memory address `A` that falls within some OCM region N into an OCM bank offset `B` (using the formula `B = $offsetN + (A - $startN)`), the following situations are possible:

- An integer overflow occurs. The translation formula above assumes that all operands are 64-bit unsigned integers and detects overflow accordingly.
- The resulting OCM bank offset `B` is beyond OCM bank boundaries.

If either (or both) of these situations occur during translation, an `IADDRESS` or `DADDRESS` exception is raised, depending on whether the translation was performed for an instruction fetch or data load/store.

15 Bootstrapping

When a Cereon machine is turned on, the following steps are performed:

1. The `W` flag of each DMA channel's `$state` register is set to 0.
2. All registers of all processors are set to 0, with the exception of registers explicitly specified below as being set to something else.
3. In every Cereon system, there is exactly one processor set up as a primary processor. If the system has more than one processor, all remaining processors are set up as secondary processors. This processor type setup is hardwired and cannot be changed. If the primary processor has more than one core, one of these cores is hardwired as a primary core.
4. For each processor, whether primary or secondary, the special bootstrap IP value is hardwired. This value is copied to `$ip` register.
5. For each processor, the `K` flag of `$state` if set to 1.
6. For each processor, the `B` flag of `$state` is set to reflect the default byte ordering. Whether this flag can be changed later or not depends on the processor model.
7. For the primary core of the primary processor, bit 31 of `$state` is set to 1, thus allowing `HARDWARE` interrupts.
8. For all secondary processor cores, bits 30 and 31 of `$state` is set to 1, thus allowing `EXTERNAL` and `HARDWARE` interrupts.
9. For all secondary processor cores, `$ip` is copied to `$iha.ext` and `$state` is copied to `$ihstate.ext`. After the copy, the `W` flag of `$ihstate.ext` is set to 1.
10. For the primary processor, the `W` flag of `$state` is set to 1. This effectively starts the primary processor.

In less technical terms, the actions above ensure that:

- All DMA channels of all processors are made idle.
- The primary processor is set up to run in Real Kernel mode, starting execution at its bootstrap address with all interrupts disabled (except `HARDWARE` interrupts).
- All secondary processors are set up to be initially not working (i.e. halted). However, they all have `EXTERNAL` and `HARDWARE` interrupts enabled, and the `EXTERNAL` interrupt handler is set to bootstrap address executed in Real Kernel mode.
- The last action of the bootstrap sequence starts the primary processor. Normally, this will cause it to execute bootstrap code from ROM or EPROM. If the operating system supports multiprocessing, the primary processor will eventually send a signal to all secondary processors, causing them to enter their `EXTERNAL` interrupt handlers in Real Kernel mode with all interrupts (except the `HARDWARE` interrupt) disabled.

16 Appendix A: GNU Free Documentation License

- **L.** Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- **M.** Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- **N.** Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- **O.** Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

17 Appendix B: Exception codes

The table below summarizes supported exception codes and describes situations when these exceptions occur.

Exception code (hex)	Exception name	Occurs when
`0000000000000001`	`ZDIV`	An attempt is made to perform an integer division by zero and an `N` flag of `$state` is set.
`0000000000000002`	`IADDRESS`	The exception is caused by an attempt to fetch an instruction from an invalid address. In real mode, this is any address where memory does not exist. In virtual mode, this is any virtual address for which there is no corresponding virtual page.
`0000000000000003`	`DADDRESS`	The exception is caused by an attempt to load/store data from/to an invalid address. In real mode, this is any address where memory does not exist. In virtual mode, this is any virtual address for which there is no corresponding virtual page.
`0000000000000004`	`IACCESS`	The exception is caused by an attempt to fetch an instruction from the virtual address that is valid in itself (mapped), but does not have the `EXECUTE` access right. The exception never occurs in real mode, since real mode offers no memory protection.
`0000000000000005`	`DACCESS`	The exception is caused by an attempt to load/store data from/to the virtual address that is valid in itself (mapped), but does not have the corresponding `READ` or `WRITE` access right. The exception also occurs in real mode, if an attempt is made to write to a ROM..
`0000000000000006`	`IALIGN`	The exception is caused by an attempt to fetch an instruction which is not naturally aligned.
`0000000000000007`	`DALIGN`	The exception is caused by an attempt to load/store data which is not naturally aligned.
`0000000000000008`	`OPCODE`	The exception is generated as a result of an attempt to execute an instruction

		with an invalid operation and/or function code.
`0000000000000009`	`OPERAND`	The exception is generated when an instruction otherwise valid uses invalid operands.
`000000000000000A`	`PRIVILEGED`	The exception is generated by an attempt to execute a privileged instruction when the processor is in User mode.
`000000000000000B`	`PAGETABLE`	The exception is generated when a page table access fails during virtual address translation. The exception never occurs in real mode.
`000000000000000C` ... `FFFFFFFFFFFFFF0C`	`IPAGEFAULT`	The exception is generated when an instruction virtual address is translated for which the virtual page is not currently in memory. The upper 56 bits of the exception code contain the upper 56 bits of an instruction address that could not be translated due to a page fault. The exception never occurs in real mode.
`000000000000000D` ... `FFFFFFFFFFFFFF0D`	`DPAGEFAULT`	The exception is generated when a data virtual address is translated for which the virtual page is not currently in memory. The upper 56 bits of the exception code contain the upper 56 bits of a data address that could not be translated due to a page fault. The exception never occurs in real mode.
`000000000000000E`	`TRAP`	This exception occurs at the beginning of an instruction cycle if a T (trap) flag is set.
`000000000000000F`	`MASKED`	This exception occurs if a processor attempts to execute the `SVC` instruction when the `SVC` interrupt is masked.
`0000000000000010`	`IOVERFLOW`	This exception occurs if an integer arithmetic overflow was detected and an `O` flag of `$state` is set.
`0000000000000011`	`FOPERAND`	This exception occurs if a floating point operation was offered an invalid operand and an `R` flag of `$state` is set.
`0000000000000012`	`FZDIV`	This exception occurs if a floating point division by zero has occurred and

		an Z flag of $state is set.
0000000000000013	FOVERFLOW	This exception occurs if an floating point arithmetic overflow was detected and an E flag of $state is set.
0000000000000014	FUNDERFLOW	This exception occurs if an floating point arithmetic underflow was detected and an U flag of $state is set.
0000000000000015	FINEXACT	This exception occurs if an floating point operation has produced an inexact result and an I flag of $state is set.
0000000000000016 ... 0000000000000F16	DEBUG0 .. DEBUG15	One of the debug events (0..15) has occurred. The interrupt handler can analyze the exception code to determine which one it was. The bits 8..11 of the exception code contain the number of the debug breakpoint 0..15 that caused this exception.
0000000000000017 ... 0000000000001F17	BREAK0 .. BREAK31	Caused by execution of a brk instruction. The interrupt handler can analyze the exception code to determine the cause of the break. The bits 8..12 of the exception code contain the break code 0..31 that caused this exception.
FFFFFFFFFFFFFFFF	UNKNOWN	A program error could not be categorized.

All exception codes not explicitly mentioned in the above table are reserved for future use.

18 Appendix C: External interrupt types

The table below summarizes supported `EXTERNAL` interrupt types codes and describes situations when these interrupts occur.

Code (hex)	Interrupt type name	Occurs when	$isc.hw
`0001`	`SIGNAL`	A processor sends a signal to either itself or another processor.	Bits 63 48 0000000000000001 Bits 47 32 sender Bits 31 0 subcode • Sender – the 16-bit ID of the processor core that has sent the signal. • Subcode – the 32 lower bits of a register specified in a `sigp` instruction.
`0002`	`RESET`	A processor is reset.	Bits 63 48 0000000000000010 Bits 47 0 -

All external interrupt types not explicitly mentioned in the above table are reserved for future use.

19 Appendix D: Hardware error codes

The table below summarizes supported hardware codes and describes situations when these errors occur.

Error code (hex)	Error name	Occurs when
`0000000000000001`	`PROCESSOR`	A fault has been detected in processor logic.
`0000000000000002`	`MEMORY`	A fault has been detected while accessing the memory.
`0000000000000003`	`BUS`	A main bus fault has been detected.
`0000000000000004`	`IO`	A fault has been detected in the I/O port (hardware faults in I/O controllers and/or I/O devices are reported via `IO` interrupts).
`0000000000000005`	`TIMER`	An interrupt timer fault has been detected.
`FFFFFFFFFFFFFFFF`	`UNKNOWN`	A hardware fault has occurred which could not be categorized exactly.

All hardware error codes not explicitly mentioned in the above table are reserved for future use.

20 Appendix E: DMA interrupt status codes

The table below summarizes supported DMA interrupt status codes and describes situations when these DMA interrupts occur.

Interrupt status code (hex)	Occurs when
`0000`	The channel program executes the `dma.stop` instruction.
`0001`	DMA channel attempts to fetch a channel instruction from an invalid address, where memory does not exist.
`0002`	DMA channel attempts to load/store data from/to an invalid address where memory does not exist.
`0003`	DMA channel attempts to fetch an instruction which is not naturally aligned.
`0004`	DMA channel attempts to load/store data which is not naturally aligned.
`0005`	DMA channel attempts to execute a channel instruction with an invalid operation code.
`FFFF`	A DMA channel error could not be categorized.

All DMA interrupt status codes not explicitly mentioned in the above table are reserved for future use.