

All registers and memory locations are 32 bits, the concept of *byte* does not apply except in the few special string-processing instructions. When characters are stored to make a string, they are packed four per memory location, with the first character of the string being in the least-significant 8 bits.

Negative numbers are represented in the two's complement format.

Floating point numbers are stored in the intel 32-bit floating format, whatever that is.

Bits are numbered from 0, the least significant, to 31 the most significant.

In numeric representations, bit 31 is the sign bit.

There are 16 regular registers, numbered from 0 to 15.

R0 is a scratch register, with slightly limited functionality

R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12 are general purpose registers

SP, the stack pointer, is encoded as register 13

FP, the frame pointer, is encoded as register 14

PC, the program counter, is encoded as register 15

The instruction format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Operation							I	Main Register				Index Register			Numeric Operand																

I is the Indirect bit.

Two's complement, range -32768 to +32767

If bits 16-19 are all zero, i.e. "Index Register" indicates R0, then no index register is used when the instruction executes. Thus it is not possible to use R0 as an index register.

In the description of an instruction, the term *reg* refers to the register indicated by bits 20 to 23 (main register), and *operand* refers to the combination of indirect bit, index register, and numeric operand as illustrated on the next two pages.

If the term *value* appears in the description, it refers to the value of the operand, which is calculated as follows:

```

part1 = numeric operand;
part2 = 0;
if (index register ≠ 0)
    part2 = contents of indicated index register
total = part1 + part2;
if (indirect bit ≠ 0)
    value = contents of memory location [total];
else
    value = total;

```

If the sequence "*reg* ← *x*" appears, it means that the content of the main register is replaced by *x*.

If the sequence "*destination* ← *x*" appears, then the operand may consist of just an index register, in which case the content of the register is replaced by *x*, otherwise the indirect bit must be set, and the content of memory location [total] is replaced by *x*.

## Assembly Examples:

RET	Operation = 37
01001010 00000000 00000000 00000000	Indirect bit = 0
4A000000	Main register = 0
	Index register = 0
	Numeric = 0
INC R6	Operation = 4
00001000 01100000 00000000 00000000	Indirect bit = 0
08600000	Main register = 6
	Index register = 0
	Numeric = 0
LOAD R2, 36	Operation = 1
00000010 00100000 00000000 00100100	Indirect bit = 0
02200024	Main register = 2
	Index register = 0
	Numeric = 36
ADD R7, R3	Operation = 6
00001100 01110011 00000000 00000000	Indirect bit = 0
0C730000	Main register = 7
	Index register = 3
	Numeric = 0
LOAD R7, R3 + 12	Operation = 1
00000010 01110011 00000000 00001100	Indirect bit = 0
0273000C	Main register = 7
	Index register = 3
	Numeric = 12
ADD R4, [R3]	Operation = 6
00001101 01000011 00000000 00000000	Indirect bit = 1
0D430000	Main register = 4
	Index register = 3
	Numeric = 0
STORE R2, [1234]	Operation = 3
00000111 00100000 00000100 11010010	Indirect bit = 1
072004D2	Main register = 2
	Index register = 0
	Numeric = 1234
STORE R2, [R5 - 375]	Operation = 3
00000111 00100101 11111110 10001001	Indirect bit = 1
0725FE89	Main register = 2
	Index register = 5
	Numeric = -375

Execution Examples, starting from these values already in memory:

<u>location</u>	<u>contents</u>
27100	592
27101	759
27102	43
27103	27105
27104	2
27105	682
27106	11
27107	22
27108	33

LOAD R2, 5	The value stored in register 2 is now 5
LOAD R3, R2+4	The value stored in register 3 is now 9
LOAD R4, 27102	The value stored in register 4 is now 27102
LOAD R5, [27100]	The value stored in register 5 is now 592
LOAD R6, [R4]	The value stored in register 6 is now 43
ADD R6, R2	The value stored in register 6 is now 48
STORE R6, [27101]	The content of memory location 27101 is changed from 759 to 48
INC R6	The value stored in register 6 is now 49
STORE R6, [R4 - 2]	The content of memory location 27100 is changed from 592 to 49
LOAD SP, 27108	The value stored in register 13 (stack pointer) is now 27108
PUSH R2	The content of memory location 27107 is changed from 22 to 5 The value stored in register 13 (stack pointer) is now 27107
PUSH [R4]	The content of memory location 27106 is changed from 11 to 43 The value stored in register 13 (stack pointer) is now 27106
POP R4	The value stored in register 4 is now 43 The value stored in register 13 (stack pointer) is now 27107
STORE R6, 27101	Fails to execute, as the operand does not address memory.

<u>opcode</u>	<u>mnemonic</u>	<u>action</u>
0	HALT	the processor is halted, execution of instructions stops.
1	LOAD <i>reg, operand</i>	$reg \leftarrow value$
2	LOADH <i>reg, operand</i>	$reg \leftarrow (reg \wedge FFFF) + (value \ll 16)$ the most significant 16 bits of the register are replaced
3	STORE <i>reg, operand</i>	$destination \leftarrow reg$
4	INC <i>operand</i>	$destination \leftarrow value + 1$
5	DEC <i>operand</i>	$destination \leftarrow value - 1$
6	ADD <i>reg, operand</i>	$reg \leftarrow reg + value$
7	SUB <i>reg, operand</i>	$reg \leftarrow reg - value$
8	MUL <i>reg, operand</i>	$reg \leftarrow reg \times value$
9	DIV <i>reg, operand</i>	$reg \leftarrow reg \div value$
10	MOD <i>reg, operand</i>	$reg \leftarrow reg \text{ modulo } value$
11	RSUB <i>reg, operand</i>	$reg \leftarrow value - reg$
12	RDIV <i>reg, operand</i>	$reg \leftarrow value \div reg$
13	RMOD <i>reg, operand</i>	$reg \leftarrow value \text{ modulo } reg$
14	AND <i>reg, operand</i>	$reg \leftarrow reg \wedge value$
15	OR <i>reg, operand</i>	$reg \leftarrow reg \vee value$
16	XOR <i>reg, operand</i>	$reg \leftarrow reg \oplus value$
17	NOT <i>reg, operand</i>	$reg \leftarrow \sim value$
18	SHL <i>reg, operand</i>	$flagZ \leftarrow 1$ if most sig. (value) bits of reg all 0, otherwise 0 $reg \leftarrow reg \ll value$ , zeros being inserted at the right
19	SHR <i>reg, operand</i>	$flagZ \leftarrow 1$ if least sig. (value) bits of reg all 0, otherwise 0 $reg \leftarrow reg \gg value$ , zeros being inserted at the left
20	COMP <i>reg, operand</i>	$flagZ \leftarrow 1$ if $reg = value$ , otherwise 0 $flagN \leftarrow 1$ if $reg < value$ , otherwise 0
21	COMPZ <i>operand</i>	$flagZ \leftarrow 1$ if $value = 0$ , otherwise 0 $flagN \leftarrow 1$ if $value < 0$ , otherwise 0
22	TBIT <i>reg, operand</i>	$flagZ \leftarrow value^{th} \text{ bit of } reg$
23	SBIT <i>reg, operand</i>	$value^{th} \text{ bit of } reg \leftarrow 1$
24	CBIT <i>reg, operand</i>	$value^{th} \text{ bit of } reg \leftarrow 0$

25	JUMP <i>operand</i>	$PC \leftarrow value$
26	JZER <i>reg, operand</i>	<i>if</i> ( <i>reg</i> = 0 ) $PC \leftarrow value$
27	JPOS <i>reg, operand</i>	<i>if</i> ( <i>reg</i> $\geq$ 0 ) $PC \leftarrow value$
28	JNEG <i>reg, operand</i>	<i>if</i> ( <i>reg</i> < 0 ) $PC \leftarrow value$
29	JCOND	Note that no main register is used with the JCOND instruction. Instead, its 4 bits are used to encode one of the seven condition tests shown here.
0	JCOND EQL, <i>operand</i>	<i>if</i> ( <i>flagZ</i> ) $PC \leftarrow value$
1	JCOND NEQ, <i>operand</i>	<i>if</i> ( $\sim flagZ$ ) $PC \leftarrow value$
2	JCOND LSS, <i>operand</i>	<i>if</i> ( <i>flagN</i> ) $PC \leftarrow value$
3	JCOND LEQ, <i>operand</i>	<i>if</i> ( <i>flagZ</i> $\vee$ <i>flagN</i> ) $PC \leftarrow value$
4	JCOND GTR, <i>operand</i>	<i>if</i> ( $\sim flagZ \wedge \sim flagN$ ) $PC \leftarrow value$
5	JCOND GEQ, <i>operand</i>	<i>if</i> ( $\sim flagN$ ) $PC \leftarrow value$
6	JCOND ERR, <i>operand</i>	<i>if</i> ( <i>flagE</i> ) $PC \leftarrow value$
30	GETFL <i>reg, operand</i>	$reg \leftarrow flag[value]$
31	SETFL <i>reg, operand</i>	$flag[value] \leftarrow reg$
32	GETSR <i>reg, operand</i>	$reg \leftarrow specialregister[value]$
33	SETSR <i>reg, operand</i>	$specialregister[value] \leftarrow reg$
34	PUSH <i>operand</i>	$SP \leftarrow SP - 1$ $memory[SP] \leftarrow value$
35	POP <i>operand</i>	$destination \leftarrow memory[SP]$ $SP \leftarrow SP + 1$
36	CALL <i>operand</i>	$SP \leftarrow SP - 1$ $memory[SP] \leftarrow PC$ $PC \leftarrow value$
37	RET	$PC \leftarrow memory[SP]$ $SP \leftarrow SP + 1$
38	LDCH <i>reg, operand</i>	<i>value</i> is treated as a memory address. The <i>reg</i> <sup>th</sup> 8-bit byte (character) starting from that address in memory is loaded into <i>reg</i> . i.e., $reg \leftarrow \text{byte}(reg \bmod 4) \text{ of } memory[value + reg \div 4]$
39	STCH <i>reg, operand</i>	<i>value</i> is treated as a memory address. The <i>reg</i> <sup>th</sup> 8-bit byte (character) starting from that address is replaced by the value of register 0 without modifying the other 24 bits of that word. $\text{byte}(reg \bmod 4) \text{ of } memory[value + reg \div 4] \leftarrow R0$
40	PERI	Control peripheral activity: see separate documentation
42	FLAGSJ <i>reg, operand</i>	$all\ flags \leftarrow reg$

		$PC \leftarrow value$
43	WAIT	CPU idles until interrupted
44	PAUSE	CPU idles for approximately 50mS, unless interrupted
45	BREAK	Enter CPU single-stepping mode
46	IRET	$all\ flags \leftarrow memory[SP]$ $PC \leftarrow memory[SP+4]$ $FP \leftarrow memory[SP+5]$ $SP \leftarrow memory[SP+6]$ $R12 \leftarrow memory[SP+7]$ $R11 \leftarrow memory[SP+8]$ $R10 \leftarrow memory[SP+9]$ $R9 \leftarrow memory[SP+10]$ $R8 \leftarrow memory[SP+11]$ $R7 \leftarrow memory[SP+12]$ $R6 \leftarrow memory[SP+13]$ $R5 \leftarrow memory[SP+14]$ $R4 \leftarrow memory[SP+15]$ $R3 \leftarrow memory[SP+16]$ $R2 \leftarrow memory[SP+17]$ $R1 \leftarrow memory[SP+18]$ $R0 \leftarrow memory[SP+19]$ $SP \leftarrow SP + 20$
47	SYSCALL <i>reg, code</i>	$memory[SP-1] \leftarrow R0$ $memory[SP-2] \leftarrow R1$ $memory[SP-3] \leftarrow R2$ $memory[SP-4] \leftarrow R3$ $memory[SP-5] \leftarrow R4$ $memory[SP-6] \leftarrow R5$ $memory[SP-7] \leftarrow R6$ $memory[SP-8] \leftarrow R7$ $memory[SP-9] \leftarrow R8$ $memory[SP-10] \leftarrow R9$ $memory[SP-11] \leftarrow R10$ $memory[SP-12] \leftarrow R11$ $memory[SP-13] \leftarrow R12$ $memory[SP-14] \leftarrow SP$ $memory[SP-15] \leftarrow FP$ $memory[SP-16] \leftarrow PC$ $memory[SP-17] \leftarrow reg$ $memory[SP-18] \leftarrow main\ register\ number$ $memory[SP-19] \leftarrow code$ $memory[SP-20] \leftarrow all\ flags$ $SP \leftarrow SP - 20$ $PC \leftarrow memory[specialregister[CGBR] + code]$ $flagSys \leftarrow 1$
48	ATAS <i>reg, operand</i>	$reg \leftarrow value ; destination \leftarrow 1$

*performed indivisibly, ignoring interrupts*

49	PHLOAD <i>reg, operand</i>	$reg \leftarrow \text{physicalmemory}[\text{value}]$
50	PHSTORE <i>reg, operand</i>	$\text{physicalmemory}[\text{value}] \leftarrow reg$
51	VTRAN <i>reg, operand</i>	$reg \leftarrow \text{physical address for virtual address value}$
52	MOVE <i>reg, reg2</i>	while $R0 > 0$ repeat { $\text{memory}[reg2] \leftarrow \text{memory}[reg]$ $reg2 \leftarrow reg2 + 1$ $reg \leftarrow reg + 1$ $R0 \leftarrow R0 - 1$ }
53	FADD <i>reg, operand</i>	<i>floating point:</i> $reg \leftarrow reg + \text{value}$
54	FSUB <i>reg, operand</i>	<i>floating point:</i> $reg \leftarrow reg - \text{value}$
55	FMUL <i>reg, operand</i>	<i>floating point:</i> $reg \leftarrow reg \times \text{value}$
56	FDIV <i>reg, operand</i>	<i>floating point:</i> $reg \leftarrow reg \div \text{value}$
57	FCOMP <i>reg, operand</i>	<i>floating point:</i> $\text{flagZ} \leftarrow 1$ if $reg = \text{value}$ , otherwise 0 $\text{flagN} \leftarrow 1$ if $reg < \text{value}$ , otherwise 0
58	FCOMPZ <i>reg, operand</i>	<i>floating point:</i> $\text{flagZ} \leftarrow 1$ if $reg = 0$ , otherwise 0 $\text{flagN} \leftarrow 1$ if $reg < 0$ , otherwise 0
59	FIX <i>reg, operand</i>	$reg \leftarrow (\text{int})\text{value}$ , <i>value</i> interpreted as floating point
60	FRND <i>reg, operand</i>	$reg \leftarrow (\text{float})(\text{closest int to value})$ , <i>both floating point</i>
61	FLOAT <i>reg, operand</i>	$reg \leftarrow (\text{float})\text{value}$ , <i>value</i> interpreted as an integer
62	FLOG <i>reg, operand</i>	<i>floating point:</i> $reg \leftarrow \text{natural log}(reg)$ , if $\text{value} = 0$ $reg \leftarrow \text{log base value}(reg)$ , otherwise
63	FEXP <i>reg, operand</i>	<i>floating point:</i> $reg \leftarrow e \text{ to power}(reg)$ , if $\text{value} = 0$ $reg \leftarrow \text{value to power}(reg)$ , otherwise
64	FFO <i>reg, operand</i>	$reg \leftarrow \text{number of bits to right of first 1 in value}$ if $\text{value} = 0$ : $reg \leftarrow -1$ , $\text{flagZ} \leftarrow 1$ , $\text{flagN} \leftarrow 1$
65	FLZ <i>reg, operand</i>	$reg \leftarrow \text{number of bits to right of last 0 in value}$ if $\text{value} = -1$ : $reg \leftarrow -1$ , $\text{flagZ} \leftarrow 1$ , $\text{flagN} \leftarrow 1$
66	RAND <i>reg</i>	$reg \leftarrow \text{random positive number}$
67	TRACE <i>reg, operand</i>	display PC, <i>reg</i> , and <i>value</i> on console
68	TYPE <i>operand</i>	send single character <i>value</i> to controlling teletype

69	INCH <i>operand</i>	<i>destination</i> $\leftarrow$ one character code from controlling keyboard or -1 if none available
70	ANDN <i>reg, operand</i>	$reg \leftarrow reg \wedge \sim value$
71	ORN <i>reg, operand</i>	$reg \leftarrow reg \vee \sim value$
72	NEG <i>reg, operand</i>	$reg \leftarrow - value$
73	FNEG <i>reg, operand</i>	$reg \leftarrow - value$ , <i>value</i> interpreted as floating point
74	ROTL <i>reg, operand</i>	<i>reg</i> is shifted <i>value</i> bits left, with the bits lost at the left being reinserted at the right.
75	ROTR <i>reg, operand</i>	<i>reg</i> is shifted <i>value</i> bits right, with the bits lost at the right being reinserted at the left.
76	ASR <i>reg, operand</i>	$flagZ \leftarrow 1$ if least sig. ( <i>value</i> ) bits of <i>reg</i> all 0, otherwise 0 $reg \leftarrow reg \gg value$ , the sign bit being duplicated at the left
77	EXBR <i>reg, operand</i>	$R0 \leftarrow$ bit range described by <i>reg</i> from <i>value</i> , with the most significant bit of the range giving the sign.
78	EXBRV <i>reg, operand</i>	$R0 \leftarrow$ bit range described by <i>reg</i> of <i>value</i> , with the most significant bit of the range giving the sign.
79	DPBR <i>reg, operand</i>	bit range described by <i>reg</i> from <i>value</i> $\leftarrow R0$ .
80	DPBRV <i>reg, operand</i>	bit range described by <i>reg</i> of <i>value</i> $\leftarrow R0$ .
81	ADJS <i>reg, operand</i>	the bit range selector in <i>reg</i> is advanced by <i>value</i> positions, taking into account the range size and the requirement for ranges not to span two words. <i>value</i> may be negative.
82	UEXBR <i>reg, operand</i>	$R0 \leftarrow$ bit range described by <i>reg</i> from <i>value</i> , unsigned.
83	UEXBRV <i>reg, operand</i>	$R0 \leftarrow$ bit range described by <i>reg</i> of <i>value</i> , unsigned.
84	UCOMP <i>reg, operand</i>	$flagZ \leftarrow 1$ if $reg = value$ , otherwise 0 $flagN \leftarrow 1$ if $reg < value$ , otherwise 0, an unsigned comparison
85	UMUL <i>reg, operand</i>	$reg \leftarrow reg \times value$ , unsigned
86	UDIV <i>reg, operand</i>	$reg \leftarrow reg \div value$ , unsigned
87	UMOD <i>reg, operand</i>	$reg \leftarrow reg \bmod value$ , unsigned



