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 my 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

0100101 0 0000 0000 0000000000000000
4A00000
```

Operation = 37
Indirect bit = 0
Main register = 0
Index register = 0
Numeric = 0

```
   INC   R6

0000100 0 0110 0000 0000000000000000
0860000
```

Operation = 4
Indirect bit = 0
Main register = 6
Index register = 0
Numeric = 0

```
   LOAD  R2, 36

0000001 0 0010 0000 0000000000100100
0220024
```

Operation = 1
Indirect bit = 0
Main register = 2
Index register = 0
Numeric = 36

```
   ADD   R7, R3

0000110 0 0111 0011 0000000000000000
0C73000
```

Operation = 6
Indirect bit = 0
Main register = 7
Index register = 3
Numeric = 0

```
   LOAD  R7, R3 + 12

0000001 0 0111 0011 0000000000001100
027300C
```

Operation = 1
Indirect bit = 0
Main register = 7
Index register = 3
Numeric = 12

```
   ADD   R4, [R3]

0000110 1 0100 0011 0000000000000000
0D43000
```

Operation = 6
Indirect bit = 1
Main register = 4
Index register = 3
Numeric = 0

```
   STORE R2, [1234]

0000011 1 0010 0000 0000010011010010
07204D2
```

Operation = 3
Indirect bit = 1
Main register = 2
Index register = 0
Numeric = 1234

```
   STORE R2, [R5 - 375]

0000011 1 0010 0101 1111111010001001
0725FE89
```

Operation = 3
Indirect bit = 1
Main register = 2
Index register = 5
Numeric = -375

Execution Examples, starting from these values already in memory:

location	*contents*
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.

opcode	*mnemonic*	*action*
`0`	`HALT`	the processor is halted, execution of instructions stops.
`1`	`LOAD` *reg*, *operand*	*reg* ← *value*
`2`	`LOADH` *reg*, *operand*	*reg* ← (*reg* ∧ FFFF) + (*value* « 16) the most significant 16 bits of the register are replaced
`3`	`STORE` *reg*, *operand*	*destination* ← *reg*
`4`	`INC` *operand*	*destination* ← *value* + 1
`5`	`DEC` *operand*	*destination* ← *value* - 1
`6`	`ADD` *reg*, *operand*	*reg* ← *reg* + *value*
`7`	`SUB` *reg*, *operand*	*reg* ← *reg* - *value*
`8`	`MUL` *reg*, *operand*	*reg* ← *reg* × *value*
`9`	`DIV` *reg*, *operand*	*reg* ← *reg* ÷ *value*
`10`	`MOD` *reg*, *operand*	*reg* ← *reg* `modulo` *value*
`11`	`RSUB` *reg*, *operand*	*reg* ← *value* - *reg*
`12`	`RDIV` *reg*, *operand*	*reg* ← *value* ÷ *reg*
`13`	`RMOD` *reg*, *operand*	*reg* ← *value* `modulo` *reg*
`14`	`AND` *reg*, *operand*	*reg* ← *reg* ∧ *value*
`15`	`OR` *reg*, *operand*	*reg* ← *reg* ∨ *value*
`16`	`XOR` *reg*, *operand*	*reg* ← *reg* ⊕ *value*
`17`	`NOT` *reg*, *operand*	*reg* ← ~ *value*
`18`	`SHL` *reg*, *operand*	*flagZ* ← 1 *if most sig. (value) bits of reg all* 0*, otherwise* 0 *reg* ← *reg* « *value,* zeros being inserted at the right
`19`	`SHR` *reg*, *operand*	*flagZ* ← 1 *if least sig. (value) bits of reg all* 0*, otherwise* 0 *reg* ← *reg* » *value,* zeros being inserted at the left
`20`	`COMP` *reg*, *operand*	*flagZ* ← 1 *if reg* = *value, otherwise* 0 *flagN* ← 1 *if reg* < *value, otherwise* 0
`21`	`COMPZ` *operand*	*flagZ* ← 1 *if value* = 0*, otherwise* 0 *flagN* ← 1 *if value* < 0*, otherwise* 0
`22`	`TBIT` *reg*, *operand*	*flagZ* ← $value^{th}$ *bit of reg*
`23`	`SBIT` *reg*, *operand*	$value^{th}$ *bit of reg* ← 1
`24`	`CBIT` *reg*, *operand*	$value^{th}$ *bit of reg* ← 0

	25	`JUMP` *operand*	PC ← *value*
	26	`JZER` *reg*, *operand*	*if* (*reg* = 0) PC ← *value*
	27	`JPOS` *reg*, *operand*	*if* (*reg* ≥ 0) PC ← *value*
	28	`JNEG` *reg*, *operand*	*if* (*reg* < 0) PC ← *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`, *operand*	*if* (*flagZ*) PC ← *value*
1		`JCOND NEQ`, *operand*	*if* (~ *flagZ*) PC ← *value*
2		`JCOND LSS`, *operand*	*if* (*flagN*) PC ← *value*
3		`JCOND LEQ`, *operand*	*if* (*flagZ* ∨ *flagN*) PC ← *value*
4		`JCOND GTR`, *operand*	*if* (~*flagZ* ∧ ~*flagN*) PC ← *value*
5		`JCOND GEQ`, *operand*	*if* (~*flagN*) PC ← *value*
6		`JCOND ERR`, *operand*	*if* (*flagE*) PC ← *value*
	30	`GETFL` *reg, operand*	*reg* ← *flag*[*value*]
	31	`SETFL` *reg, operand*	*flag*[*value*] ← *reg*
	32	`GETSR` *reg, operand*	*reg* ← *specialregister*[*value*]
	33	`SETSR` *reg, operand*	*specialregister*[*value*] ← *reg*
	34	`PUSH` *operand*	SP ← SP - 1 *memory*[SP] ← *value*
	35	`POP` *operand*	*destination* ← *memory*[SP] SP ← SP + 1
	36	`CALL` *operand*	SP ← SP - 1 *memory*[SP] ← PC PC ← *value*
	37	`RET`	PC ← *memory*[SP] SP ← SP + 1
	38	`LDCH` *reg*, *operand*	*value* is treated as a memory address. The *reg*th 8-bit byte (character) starting from that address in memory is loaded into *reg*. i.e., *reg* ← byte *(reg* `modulo` 4*)* of *memory*[*value* + *reg*÷4]
	39	`STCH` *reg*, *operand*	*value* is treated as a memory address. The *reg*th 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. byte *(reg* `modulo` 4*)* of *memory*[*value* + *reg*÷4] ← R0
	40	`PERI`	Control peripheral activity: see separate documentation
	42	`FLAGSJ` *reg*, *operand*	*all flags* ← *reg*

		PC ← *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* ← *memory*[SP+1] PC ← *memory*[SP+5] FP ← *memory*[SP+6] SP ← *memory*[SP+7] R12 ← *memory*[SP+8] R11 ← *memory*[SP+9] R10 ← *memory*[SP+10] R9 ← *memory*[SP+11] R8 ← *memory*[SP+12] R7 ← *memory*[SP+13] R6 ← *memory*[SP+14] R5 ← *memory*[SP+15] R4 ← *memory*[SP+16] R3 ← *memory*[SP+17] R2 ← *memory*[SP+18] R1 ← *memory*[SP+19] R0 ← *memory*[SP+20] SP ← SP + 21
47	`SYSCALL` *reg*, *code*	*memory*[SP-1] ← R0 *memory*[SP-2] ← R1 *memory*[SP-3] ← R2 *memory*[SP-4] ← R3 *memory*[SP-5] ← R4 *memory*[SP-6] ← R5 *memory*[SP-7] ← R6 *memory*[SP-8] ← R7 *memory*[SP-9] ← R8 *memory*[SP-10] ← R9 *memory*[SP-11] ← R10 *memory*[SP-12] ← R11 *memory*[SP-13] ← R12 *memory*[SP-14] ← SP *memory*[SP-15] ← FP *memory*[SP-16] ← PC *memory*[SP-17] ← *reg* *memory*[SP-18] ← *main register number* *memory*[SP-19] ← *code* *memory*[SP-20] ← *all flags* *memory*[SP-21] ← 40 SP ← SP - 21 PC ← *memory*[*specialregister*[CGBR] + *code*] *flagSys* ← 1

48	`ATAS` *reg*, *operand*	*reg* ← *value ; destination* ← 1 *performed indivisibly, ignoring interrupts*
49	`PHLOAD` *reg*, *operand*	*reg* ← *physicalmemory*[*value*]
50	`PHSTORE` *reg*, *operand*	*physicalmemory*[*value*] ← *reg*
51	`VTRAN` *reg*, *operand*	*reg* ← physical address for virtual address *value*
52	`MOVE` *reg*, *reg2*	while R0 > 0 repeat { *memory*[*reg2*] ← *memory*[reg] *reg2* ← *reg2* + 1 *reg* ← *reg* + 1 R0 ← R0 - 1 }
53	`FADD` *reg*, *operand*	*floating point: reg* ← *reg* + *value*
54	`FSUB` *reg*, *operand*	*floating point: reg* ← *reg* - *value*
55	`FMUL` *reg*, *operand*	*floating point: reg* ← *reg* × *value*
56	`FDIV` *reg*, *operand*	*floating point: reg* ← *reg* ÷ *value*
57	`FCOMP` *reg*, *operand*	*floating point:* *flagZ* ← 1 *if reg* = *value, otherwise* 0 *flagN* ← 1 *if reg* < *value, otherwise* 0
58	`FCOMPZ` *reg*, *operand*	*floating point:* *flagZ* ← 1 *if reg* = 0*, otherwise* 0 *flagN* ← 1 *if reg* < 0*, otherwise* 0
59	`FIX` *reg*, *operand*	*reg* ← (int)*value*, *value* interpreted as floating point
60	`FRND` *reg*, *operand*	*reg* ← (float)(closest int to *value*), *both floating point*
61	`FLOAT` *reg*, *operand*	*reg* ← (float)*value*, *value* interpreted as an integer
62	`FLOG` *reg*, *operand*	*floating point:* *reg* ← *natural log(reg), if value* = 0 *reg* ← *log base value(reg), otherwise*
63	`FEXP` *reg*, *operand*	*floating point:* *reg* ← *e to power(reg), if value* = 0 *reg* ←*value to power(reg), otherwise*
64	`FFO` *reg*, *operand*	*reg* ← number of bits to right of first 1 in *value* if *value* = 0: *reg* ← -1, *flagZ* ← 1, *flagN* ← 1
65	`FLZ` *reg*, *operand*	*reg* ← number of bits to right of last 0 in *value* if *value* = -1: *reg* ← -1, *flagZ* ← 1, *flagN* ← 1
66	`RAND` *reg*	*reg* ← random positive number

67	`TRACE` *reg, operand*	display PC*, reg,* and *value* on console
68	`TYPE` *operand*	send single character *value* to controlling teletype
69	`INCH` *operand*	*destination* ← one character code from controlling keyboard or `-1` if none available
70	`ANDN` *reg*, *operand*	*reg* ← *reg* ∧ ~ *value*
71	`ORN` *reg*, *operand*	*reg* ← *reg* ∨ ~ *value*
72	`NEG` *reg*, *operand*	*reg* ← - *value*
73	`FNEG` *reg*, *operand*	*reg* ← - *value*, *value* interpreted as floating point
74	`ROTL` *reg*, *operand*	*reg* is shifted *value* bits left, with the bits lost at the left being reinserted at the right.
75	`ROTR` *reg*, *operand*	*reg* is shifted *value* bits right, with the bits lost at the right being reinserted at the left.
76	`ASR` *reg*, *operand*	*flagZ* ← 1 *if least sig. (value) bits of reg all* 0*, otherwise* 0 *reg* ← *reg* » *value,* the sign bit being duplicated at the left
77	`EXBR` *reg*, *operand*	R0 ← bit range described by *reg* from *value,* with the most significant bit of the range giving the sign.
78	`EXBRV` *reg*, *operand*	R0 ← bit range described by *reg* of *value,* with the most significant bit of the range giving the sign.
79	`DPBR` *reg*, *operand*	bit range described by *reg* from *value* ← R0.
80	`DPBRV` *reg*, *operand*	bit range described by *reg* of *value* ← R0.
81	`ADJS` *reg*, *operand*	the bit range selector in *reg* is advanced by *value* positions, taking into account the range size and the requirement for ranges not to span two words. *value* may be negative.
82	`UEXBR` *reg*, *operand*	R0 ← bit range described by *reg* from *value,* unsigned.
83	`UEXBRV` *reg*, *operand*	R0 ← bit range described by *reg* of *value,* unsigned.
84	`UCOMP` *reg*, *operand*	*flagZ* ← 1 *if reg* = *value, otherwise* 0 *flagN* ← 1 *if reg* < *value, otherwise* 0, an unsigned comparison
85	`UMUL` *reg*, *operand*	*reg* ← *reg* × *value*, unsigned
86	`UDIV` *reg*, *operand*	*reg* ← *reg* ÷ *value*, unsigned
87	`UMOD` *reg*, *operand*	*reg* ← *reg* `modulo` *value*, unsigned
88	`CLRPP` *operand*	page containing physical address *value* all set to zero
89	`FILL` *reg*, *operand*	while R0 > 0 repeat { *memory*[*reg2*] ← *value*

$reg \leftarrow reg + 1$

$R0 \leftarrow R0 - 1$ }