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:

\[
\begin{align*}
\text{part1} &= \text{numeric operand}; \\
\text{part2} &= 0; \\
\text{if (index register} \neq 0) \\
\text{part2} &= \text{contents of indicated index register} \\
\text{total} &= \text{part1 + part2}; \\
\text{if (indirect bit} \neq 0) \\
\text{value} &= \text{contents of memory location [total]}; \\
\text{else} \\
\text{value} &= \text{total};
\end{align*}
\]

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

0100101 0 00000000000000000000000000000000
4A000000

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

INC R6

0001000 0 01100000000000000000000000000000
08600000

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

LOAD R2, 36

0000001 0 01000000000000000000000000000000
02200024

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

ADD R7, R3

0000110 0 01110011000000000000000000000000
0C730000

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

LOAD R7, R3 + 12

0000001 0 01110011000000000000000000000000
0273000C

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

ADD R4, [R3]

0000110 1 01000011000000000000000000000000
0D430000

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

STORE R2, [1234]

0000111 1 00100000000000000000000000010010
072004D2

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

STORE R2, [R5 - 375]

0000111 1 00100101111111010001001
0725FE89

Operation = 3
Indirect bit = 1
Main register = 2
Index register = 5
Numeric = -375
Execution Examples, starting from these values already in memory:

<table>
<thead>
<tr>
<th>location</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>27100</td>
<td>592</td>
</tr>
<tr>
<td>27101</td>
<td>759</td>
</tr>
<tr>
<td>27102</td>
<td>43</td>
</tr>
<tr>
<td>27103</td>
<td>27105</td>
</tr>
<tr>
<td>27104</td>
<td>2</td>
</tr>
<tr>
<td>27105</td>
<td>682</td>
</tr>
<tr>
<td>27106</td>
<td>11</td>
</tr>
<tr>
<td>27107</td>
<td>22</td>
</tr>
<tr>
<td>27108</td>
<td>33</td>
</tr>
</tbody>
</table>

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

STORE  R6, 27101
Fails to execute, as the operand does not address memory.
<table>
<thead>
<tr>
<th>opcode</th>
<th>mnemonic</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>HALT</td>
<td>the processor is halted, execution of instructions stops.</td>
</tr>
<tr>
<td>1</td>
<td>LOAD reg, operand</td>
<td>reg ← value</td>
</tr>
<tr>
<td>2</td>
<td>LOADH reg, operand</td>
<td>reg ← (reg ∧ FFFF) + (value « 16)</td>
</tr>
<tr>
<td>3</td>
<td>STORE reg, operand</td>
<td>destination ← reg</td>
</tr>
<tr>
<td>4</td>
<td>INC operand</td>
<td>destination ← value + 1</td>
</tr>
<tr>
<td>5</td>
<td>DEC operand</td>
<td>destination ← value - 1</td>
</tr>
<tr>
<td>6</td>
<td>ADD reg, operand</td>
<td>reg ← reg + value</td>
</tr>
<tr>
<td>7</td>
<td>SUB reg, operand</td>
<td>reg ← reg - value</td>
</tr>
<tr>
<td>8</td>
<td>MUL reg, operand</td>
<td>reg ← reg × value</td>
</tr>
<tr>
<td>9</td>
<td>DIV reg, operand</td>
<td>reg ← reg ÷ value</td>
</tr>
<tr>
<td>10</td>
<td>MOD reg, operand</td>
<td>reg ← reg modulo value</td>
</tr>
<tr>
<td>11</td>
<td>RSUB reg, operand</td>
<td>reg ← value - reg</td>
</tr>
<tr>
<td>12</td>
<td>RDIV reg, operand</td>
<td>reg ← value ÷ reg</td>
</tr>
<tr>
<td>13</td>
<td>RMOD reg, operand</td>
<td>reg ← value modulo reg</td>
</tr>
<tr>
<td>14</td>
<td>AND reg, operand</td>
<td>reg ← reg ∧ value</td>
</tr>
<tr>
<td>15</td>
<td>OR reg, operand</td>
<td>reg ← reg ∨ value</td>
</tr>
<tr>
<td>16</td>
<td>XOR reg, operand</td>
<td>reg ← reg ⊕ value</td>
</tr>
<tr>
<td>17</td>
<td>NOT reg, operand</td>
<td>reg ← ~ value</td>
</tr>
</tbody>
</table>
| 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\textsuperscript{th} bit of reg                                                 |
| 23     | SBIT reg, operand | value\textsuperscript{th} bit of reg ← 1                                                   |
| 24     | CBIT reg, operand | value\textsuperscript{th} bit of reg ← 0                                                   |
25 JUMP operand \( \text{PC} \leftarrow \text{value} \)

26 JZER reg, operand \( \text{if } (\text{reg} = 0) \text{PC} \leftarrow \text{value} \)

27 JPOS reg, operand \( \text{if } (\text{reg} \geq 0) \text{PC} \leftarrow \text{value} \)

28 JNEG reg, operand \( \text{if } (\text{reg} < 0) \text{PC} \leftarrow \text{value} \)

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.

\begin{align*}
0 & \quad \text{JCOND EQL, operand} & & \text{if } (\text{flagZ}) \text{ PC} \leftarrow \text{value} \\
1 & \quad \text{JCOND NEQ, operand} & & \text{if } (\sim\text{flagZ}) \text{ PC} \leftarrow \text{value} \\
2 & \quad \text{JCOND LSS, operand} & & \text{if } (\text{flagN}) \text{ PC} \leftarrow \text{value} \\
3 & \quad \text{JCOND LEQ, operand} & & \text{if } (\text{flagZ} \lor \text{flagN}) \text{ PC} \leftarrow \text{value} \\
4 & \quad \text{JCOND GTR, operand} & & \text{if } (\sim\text{flagZ} \land \sim\text{flagN}) \text{ PC} \leftarrow \text{value} \\
5 & \quad \text{JCOND GEQ, operand} & & \text{if } (\sim\text{flagN}) \text{ PC} \leftarrow \text{value} \\
6 & \quad \text{JCOND ERR, operand} & & \text{if } (\text{flagE}) \text{ PC} \leftarrow \text{value}
\end{align*}

30 GETFL reg, operand \( \text{reg} \leftarrow \text{flag}[\text{value}] \)

31 SETFL reg, operand \( \text{flag}[\text{value}] \leftarrow \text{reg} \)

32 GETSR reg, operand \( \text{reg} \leftarrow \text{specialregister}[\text{value}] \)

33 SETSR reg, operand \( \text{specialregister}[\text{value}] \leftarrow \text{reg} \)

34 PUSH operand \( \text{SP} \leftarrow \text{SP} - 1 \)  
\( \text{memory}[\text{SP}] \leftarrow \text{value} \)

35 POP operand \( \text{destination} \leftarrow \text{memory}[\text{SP}] \)  
\( \text{SP} \leftarrow \text{SP} + 1 \)

36 CALL operand \( \text{SP} \leftarrow \text{SP} - 1 \)  
\( \text{memory}[\text{SP}] \leftarrow \text{PC} \)  
\( \text{PC} \leftarrow \text{value} \)

37 RET \( \text{PC} \leftarrow \text{memory}[\text{SP}] \)  
\( \text{SP} \leftarrow \text{SP} + 1 \)

38 LDCH reg, operand \( \text{value} \) is treated as a memory address. The \( \text{reg} \)th 8-bit byte (character) starting from that address in memory is loaded into \( \text{reg} \), i.e.,  
\( \text{reg} \leftarrow \text{byte}(\text{reg} \mod 4) \) of \( \text{memory}[\text{value} + \text{reg} \div 4] \)

39 STCH reg, operand \( \text{value} \) is treated as a memory address. The \( \text{reg} \)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.  
\( \text{byte}(\text{reg} \mod 4) \) of \( \text{memory}[\text{value} + \text{reg} \div 4] \) \( \leftarrow \text{R0} \)

40 PERI Control peripheral activity: see separate documentation

42 FLAGSJ reg, operand \( \text{all flags} \leftarrow \text{reg} \)
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 
\( \text{all flags} \leftarrow \text{memory}[\text{SP}+1] \)
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 \text{reg}, \text{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] ← \text{reg}
memory[SP-18] ← \text{main register number}
memory[SP-19] ← \text{code}
memory[SP-20] ← \text{all flags}
memory[SP-21] ← 40
SP ← SP - 21
PC ← memory[specialregister[CGBR] + \text{code}]
flagSys ← 1
48 ATAS \[\text{reg, operand}\] \[\text{reg} \leftarrow \text{value}; \text{destination} \leftarrow 1\]
performed indivisibly, ignoring interrupts

49 PHLOAD \[\text{reg, operand}\] \[\text{reg} \leftarrow \text{physicalmemory}[\text{value}]\]

50 PHSTORE \[\text{reg, operand}\] \[\text{physicalmemory}[\text{value}] \leftarrow \text{reg}\]

51 VTRAN \[\text{reg, operand}\] \[\text{reg} \leftarrow \text{physical address for virtual address value}\]

52 MOVE \[\text{reg, reg2}\] while \(\text{R0} > 0\) repeat
\{
\text{memory}[\text{reg2}] \leftarrow \text{memory}[\text{reg}]
\text{reg2} \leftarrow \text{reg2} + 1
\text{reg} \leftarrow \text{reg} + 1
\text{R0} \leftarrow \text{R0} - 1
\}

53 FADD \[\text{reg, operand}\] \text{floating point: } \text{reg} \leftarrow \text{reg} + \text{value}

54 FSUB \[\text{reg, operand}\] \text{floating point: } \text{reg} \leftarrow \text{reg} - \text{value}

55 FMUL \[\text{reg, operand}\] \text{floating point: } \text{reg} \leftarrow \text{reg} \times \text{value}

56 FDIV \[\text{reg, operand}\] \text{floating point: } \text{reg} \leftarrow \text{reg} \div \text{value}

57 FCOMP \[\text{reg, operand}\] \text{floating point:}
\[\text{flagZ} \leftarrow 1 \text{ if } \text{reg} = \text{value}, \text{ otherwise } 0\]
\[\text{flagN} \leftarrow 1 \text{ if } \text{reg} < \text{value}, \text{ otherwise } 0\]

58 FCOMPZ \[\text{reg, operand}\] \text{floating point:}
\[\text{flagZ} \leftarrow 1 \text{ if } \text{reg} = 0, \text{ otherwise } 0\]
\[\text{flagN} \leftarrow 1 \text{ if } \text{reg} < 0, \text{ otherwise } 0\]

59 FIX \[\text{reg, operand}\] \[\text{reg} \leftarrow \text{(int)value}, \text{ value interpreted as floating point}\]

60 FRND \[\text{reg, operand}\] \[\text{reg} \leftarrow \text{(float)(closest int to value)}, \text{ both floating point}\]

61 FLOAT \[\text{reg, operand}\] \[\text{reg} \leftarrow \text{(float)value}, \text{ value interpreted as an integer}\]

62 FLOG \[\text{reg, operand}\] \text{floating point:}
\[\text{reg} \leftarrow \text{natural log(reg)}, \text{ if } \text{value} = 0\]
\[\text{reg} \leftarrow \text{log base value(reg)}, \text{ otherwise}\]

63 FEXP \[\text{reg, operand}\] \text{floating point:}
\[\text{reg} \leftarrow \text{e to power(reg)}, \text{ if } \text{value} = 0\]
\[\text{reg} \leftarrow \text{value to power(reg)}, \text{ otherwise}\]

64 FFO \[\text{reg, operand}\] \[\text{reg} \leftarrow \text{number of bits to right of first 1 in value}\]
if \(\text{value} = 0\): \[\text{reg} \leftarrow -1, \text{flagZ} \leftarrow 1, \text{flagN} \leftarrow 1\]

65 FLZ \[\text{reg, operand}\] \[\text{reg} \leftarrow \text{number of bits to right of last 0 in value}\]
if \(\text{value} = -1\): \[\text{reg} \leftarrow -1, \text{flagZ} \leftarrow 1, \text{flagN} \leftarrow 1\]

66 RAND \[\text{reg}\] \[\text{reg} \leftarrow \text{random positive number} \]
<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>67</td>
<td>TRACE reg, operand</td>
<td>display PC, reg, and value on console</td>
</tr>
<tr>
<td>68</td>
<td>TYPE operand</td>
<td>send single character value to controlling teletype</td>
</tr>
<tr>
<td>69</td>
<td>INCH operand</td>
<td>destination ← one character code from controlling keyboard or -1 if none available</td>
</tr>
<tr>
<td>70</td>
<td>ANDN reg, operand</td>
<td>reg ← reg ∧ ~ value</td>
</tr>
<tr>
<td>71</td>
<td>ORN reg, operand</td>
<td>reg ← reg ∨ ~ value</td>
</tr>
<tr>
<td>72</td>
<td>NEG reg, operand</td>
<td>reg ← - value</td>
</tr>
<tr>
<td>73</td>
<td>FNEG reg, operand</td>
<td>reg ← - value, value interpreted as floating point</td>
</tr>
<tr>
<td>74</td>
<td>ROTL reg, operand</td>
<td>reg is shifted value bits left, with the bits lost at the left being reinserted at the right.</td>
</tr>
<tr>
<td>75</td>
<td>ROTR reg, operand</td>
<td>reg is shifted value bits right, with the bits lost at the right being reinserted at the left.</td>
</tr>
<tr>
<td>76</td>
<td>ASR reg, operand</td>
<td>flagZ ← 1 if least sig. (value) bits of reg all 0, otherwise 0 reg ← reg &gt;&gt; value, the sign bit being duplicated at the left</td>
</tr>
<tr>
<td>77</td>
<td>EXBR reg, operand</td>
<td>R0 ← bit range described by reg from value, with the most significant bit of the range giving the sign.</td>
</tr>
<tr>
<td>78</td>
<td>EXBRV reg, operand</td>
<td>R0 ← bit range described by reg of value, with the most significant bit of the range giving the sign.</td>
</tr>
<tr>
<td>79</td>
<td>DPBR reg, operand</td>
<td>bit range described by reg from value ← R0.</td>
</tr>
<tr>
<td>80</td>
<td>DPBRV reg, operand</td>
<td>bit range described by reg of value ← R0.</td>
</tr>
<tr>
<td>81</td>
<td>ADJS reg, operand</td>
<td>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.</td>
</tr>
<tr>
<td>82</td>
<td>UEXBR reg, operand</td>
<td>R0 ← bit range described by reg from value, unsigned.</td>
</tr>
<tr>
<td>83</td>
<td>UEXBRV reg, operand</td>
<td>R0 ← bit range described by reg of value, unsigned.</td>
</tr>
<tr>
<td>84</td>
<td>UCOMP reg, operand</td>
<td>flagZ ← 1 if reg = value, otherwise 0 flagN ← 1 if reg &lt; value, otherwise 0, an unsigned comparison</td>
</tr>
<tr>
<td>85</td>
<td>UMUL reg, operand</td>
<td>reg ← reg × value, unsigned</td>
</tr>
<tr>
<td>86</td>
<td>UDIV reg, operand</td>
<td>reg ← reg ÷ value, unsigned</td>
</tr>
<tr>
<td>87</td>
<td>UMOD reg, operand</td>
<td>reg ← reg modulo value, unsigned</td>
</tr>
<tr>
<td>88</td>
<td>CLRPP operand</td>
<td>page containing physical address value all set to zero</td>
</tr>
<tr>
<td>89</td>
<td>FILL reg, operand</td>
<td>while R0 &gt; 0 repeat</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ memory[reg2] ← value }</td>
</tr>
</tbody>
</table>
\text{reg} \leftarrow \text{reg} + 1
\text{R0} \leftarrow \text{R0} - 1 \}$