1 SPIM

1.1 Assembler Syntax

Comments in assembler files begin with a sharp-sign (#). Everything from the sharp-sign to the end of the line is ignored.

Identifiers are a sequence of alphanumeric characters, underbars (_), and dots (.) that do not begin with a number. Opcodes for instructions are reserved words that are not valid identifiers. Labels are declared by putting them at the beginning of a line followed by a colon, for example:

```
.data
item: .word 1
.text
.globl main  # Must be global
main: lw $t0, item
```

Strings are enclosed in double-quotes ("."). Special characters in strings follow the C convention:

```
newline \n
tab \t
quote \" 
```

SPIM supports a subset of the assembler directives provided by the MIPS assembler:

`.align n`
Align the next datum on a 2^n byte boundary. For example, `.align 2` aligns the next value on a word boundary. `.align 0` turns off automatic alignment of `.half`, `.word`, `.float`, and `.double` directives until the next `.data` or `.kdata` directive.

`.ascii str`
Store the string in memory, but do not null-terminate it.

`.asciiz str`
Store the string in memory and null-terminate it.

`.byte b1, ..., bn`
Store the n values in successive bytes of memory.
.comm sym size
    Allocate size bytes of data segment for symbol sym.

.data <addr>
    The following data items should be stored in the data segment. If the optional argument
    addr is present, the items are stored beginning at address addr.

double d1, ..., dn
    Store the n floating point double precision numbers in successive memory locations.

.extern sym size
    Declare that the datum stored at sym is size bytes large and is a global symbol. This
directive enables the assembler to store the datum in a portion of the data segment that
is efficiently accessed via register $gp.

.float f1, ..., fn
    Store the n floating point single precision numbers in successive memory locations.

.globl sym
    Declare that symbol sym is global and can be referenced from other files.

.half h1, ..., hn
    Store the n 16-bit quantities in successive memory halfwords.

.kdata <addr>
    The following data items should be stored in the kernel data segment. If the optional
argument addr is present, the items are stored beginning at address addr.

.ktext <addr>
    The next items are put in the kernel text segment. In SPIM, these items may only be
instructions or words (see the .word directive below). If the optional argument addr is
present, the items are stored beginning at address addr.

.label sym
    Declare that symbol sym is a label.

.lcomm sym size
    Allocate size bytes for symbol sym in the portion of the data segment that can be accessed
via register $gp.

.space n
    Allocate n bytes of space in the current segment (which must be the data segment in
SPIM).

.set noat
    Permit the program to refer to the $at register explicitly, and forbid SPIM from generating
pseudoinstructions that modify $at.

.set at
    Forbid the program from referring to the $at register explicitly, and permit SPIM to
generate pseudoinstructions that modify $at (the default).
<table>
<thead>
<tr>
<th>Service</th>
<th>System Call Code</th>
<th>Arguments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_int</td>
<td>1</td>
<td>$a0 = integer</td>
<td>integer (in $v0)</td>
</tr>
<tr>
<td>print_float</td>
<td>2</td>
<td>$f12 = float</td>
<td>float (in $f0)</td>
</tr>
<tr>
<td>print_double</td>
<td>3</td>
<td>$f12 = double</td>
<td>double (in $f0)</td>
</tr>
<tr>
<td>print_string</td>
<td>4</td>
<td>$a0 = string</td>
<td></td>
</tr>
<tr>
<td>read_int</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read_float</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read_double</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read_string</td>
<td>8</td>
<td>$a0 = buffer, $a1 = length</td>
<td></td>
</tr>
<tr>
<td>sbrk</td>
<td>9</td>
<td>$a0 = amount</td>
<td>address (in $v0)</td>
</tr>
<tr>
<td>exit</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>print_character</td>
<td>11</td>
<td>$a0 = character</td>
<td>character (in $v0)</td>
</tr>
<tr>
<td>read_character</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>open</td>
<td>13</td>
<td>$a0 = filename,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$a1 = flags, $a2 = mode</td>
<td></td>
</tr>
<tr>
<td>read</td>
<td>14</td>
<td>$a0 = file descriptor,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$a1 = buffer, $a2 = count</td>
<td></td>
</tr>
<tr>
<td>write</td>
<td>15</td>
<td>$a0 = file descriptor,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$a1 = buffer, $a2 = count</td>
<td></td>
</tr>
<tr>
<td>close</td>
<td>16</td>
<td>$a0 = file descriptor</td>
<td>0 (in $v0)</td>
</tr>
<tr>
<td>exit2</td>
<td>17</td>
<td>$a0 = value</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: System services.

.text <addr>

The next items are put in the user text segment. In SPIM, these items may only be instructions or words (see the .word directive below). If the optional argument addr is present, the items are stored beginning at address addr.

 .word w1, ..., wn

Store the $n$ 32-bit quantities in successive memory words.

SPIM does not distinguish various parts of the data segment (.data, .rdata, and .sdata).

1.2 System Calls

SPIM provides a small set of operating-system-like services through the system call (syscall) instruction. To request a service, a program loads the system call code (see Table 1) into register $v0 and the arguments into registers $a0...$a3 (or $f12 for floating point values). System calls that return values put their result in register $v0 (or $f0 for floating point results). For example, to print “the answer = 5”, use the commands:

```
.data
str: .asciiz "the answer = "
.text
li $v0, 4 # system call code for print_str
la $a0, str # address of string to print
syscall # print the string
            # print the string
li $v0, 1 # system call code for print_int
li $a0, 5 # integer to print
```

3
2 Description of the MIPS R2000

A MIPS processor consists of an integer processing unit (the CPU) and a collection of coprocessors that perform ancillary tasks or operate on other types of data such as floating point numbers (see Figure 1). SPIM simulates two coprocessors. Coprocessor 0 handles traps, exceptions, and the virtual memory system. SPIM simulates most of the first two and entirely omits details of
the memory system. Coprocessor 1 is the floating point unit. SPIM simulates most aspects of this unit.

2.1 CPU Registers

The MIPS (and SPIM) central processing unit contains 32 general purpose 32-bit registers that are numbered 0–31. Register \( n \) is designated by \$n. Register \$0 always contains the hardwired value 0. MIPS has established a set of conventions as to how registers should be used. These suggestions are guidelines, which are not enforced by the hardware. However a program that violates them will not work properly with other software. Table 2 lists the registers and describes their intended use.

Registers \$at (1), \$k0 (26), and \$k1 (27) are reserved for use by the assembler and operating system.

Registers \$a0–\$a3 (4–7) are used to pass the first four arguments to routines (remaining arguments are passed on the stack). Registers \$v0 and \$v1 (2, 3) are used to return values

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero</td>
<td>0</td>
<td>Constant 0</td>
</tr>
<tr>
<td>at</td>
<td>1</td>
<td>Reserved for assembler</td>
</tr>
<tr>
<td>v0</td>
<td>2</td>
<td>Expression evaluation and results of a function</td>
</tr>
<tr>
<td>v1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>a0</td>
<td>4</td>
<td>Argument 1</td>
</tr>
<tr>
<td>a1</td>
<td>5</td>
<td>Argument 2</td>
</tr>
<tr>
<td>a2</td>
<td>6</td>
<td>Argument 3</td>
</tr>
<tr>
<td>a3</td>
<td>7</td>
<td>Argument 4</td>
</tr>
<tr>
<td>t0</td>
<td>8</td>
<td>Temporary (not preserved across call)</td>
</tr>
<tr>
<td>t1</td>
<td>9</td>
<td>Temporary (not preserved across call)</td>
</tr>
<tr>
<td>t2</td>
<td>10</td>
<td>Temporary (not preserved across call)</td>
</tr>
<tr>
<td>t3</td>
<td>11</td>
<td>Temporary (not preserved across call)</td>
</tr>
<tr>
<td>t4</td>
<td>12</td>
<td>Temporary (not preserved across call)</td>
</tr>
<tr>
<td>t5</td>
<td>13</td>
<td>Temporary (not preserved across call)</td>
</tr>
<tr>
<td>t6</td>
<td>14</td>
<td>Temporary (not preserved across call)</td>
</tr>
<tr>
<td>t7</td>
<td>15</td>
<td>Temporary (not preserved across call)</td>
</tr>
<tr>
<td>s0</td>
<td>16</td>
<td>Saved temporary (preserved across call)</td>
</tr>
<tr>
<td>s1</td>
<td>17</td>
<td>Saved temporary (preserved across call)</td>
</tr>
<tr>
<td>s2</td>
<td>18</td>
<td>Saved temporary (preserved across call)</td>
</tr>
<tr>
<td>s3</td>
<td>19</td>
<td>Saved temporary (preserved across call)</td>
</tr>
<tr>
<td>s4</td>
<td>20</td>
<td>Saved temporary (preserved across call)</td>
</tr>
<tr>
<td>s5</td>
<td>21</td>
<td>Saved temporary (preserved across call)</td>
</tr>
<tr>
<td>s6</td>
<td>22</td>
<td>Saved temporary (preserved across call)</td>
</tr>
<tr>
<td>s7</td>
<td>23</td>
<td>Saved temporary (preserved across call)</td>
</tr>
<tr>
<td>t8</td>
<td>24</td>
<td>Temporary (not preserved across call)</td>
</tr>
<tr>
<td>t9</td>
<td>25</td>
<td>Temporary (not preserved across call)</td>
</tr>
<tr>
<td>k0</td>
<td>26</td>
<td>Reserved for OS kernel</td>
</tr>
<tr>
<td>k1</td>
<td>27</td>
<td>Reserved for OS kernel</td>
</tr>
<tr>
<td>gp</td>
<td>28</td>
<td>Pointer to global area</td>
</tr>
<tr>
<td>sp</td>
<td>29</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>fp or s8</td>
<td>30</td>
<td>Frame pointer</td>
</tr>
<tr>
<td>ra</td>
<td>31</td>
<td>Return address (used by function call)</td>
</tr>
</tbody>
</table>

Table 2: MIPS registers and the convention governing their use.
from functions. Registers $t0–t9 (8–15, 24, 25) are caller-saved registers used for temporary quantities that do not need to be preserved across calls. Registers $s0–s7 (16–23) are callee-saved registers that hold long-lived values that should be preserved across calls.

Register $sp (29) is the stack pointer, which points to the last location in use on the stack.\(^1\) Register $fp (30) is the frame pointer.\(^2\) Register $ra (31) is written with the return address for a call by the jal instruction.

Register $gp (28) is a global pointer that points into the middle of a 64K block of memory in the heap that holds constants and global variables. The objects in this heap can be quickly accessed with a single load or store instruction.

### 2.2 Addressing Modes

MIPS is a load/store architecture, which means that only load and store instructions access memory. Computation instructions operate only on values in registers. The bare machine provides only one memory addressing mode: c(rx), which uses the sum of the immediate (integer) c and the contents of register rx as the address. The virtual machine provides the following addressing modes for load and store instructions:

<table>
<thead>
<tr>
<th>Format</th>
<th>Address Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(register)</td>
<td>contents of register</td>
</tr>
<tr>
<td>imm</td>
<td>immediate</td>
</tr>
<tr>
<td>imm (register)</td>
<td>immediate + contents of register</td>
</tr>
<tr>
<td>symbol</td>
<td>address of symbol</td>
</tr>
<tr>
<td>symbol ± imm</td>
<td>address of symbol + or − immediate</td>
</tr>
<tr>
<td>symbol (register)</td>
<td>address of symbol + contents of register</td>
</tr>
<tr>
<td>symbol ± imm (register)</td>
<td>(address of symbol + or − immediate) + contents of register</td>
</tr>
</tbody>
</table>

Most load and store instructions operate only on aligned data. A quantity is aligned if its memory address is a multiple of its size in bytes. Therefore, a halfword object must be stored at even addresses and a full word object must be stored at addresses that are a multiple of 4. However, MIPS provides some instructions for manipulating unaligned data.

### 2.3 Arithmetic and Logical Instructions

In all instructions below, Src2 can either be a register or an immediate value (a 16 bit integer). The immediate forms of the instructions are only included for reference. The assembler will translate the more general form of an instruction (e.g., add) into the immediate form (e.g., addi) if the second argument is constant.

\[
\begin{align*}
\text{abs Rdest, Rs}r & \quad \text{Absolute Value} \uparrow \\
\text{addu Rdest, Rs}r1, \text{Src}2 & \quad \text{Addition} \\
\text{addiu Rdest, Rs}r1, \text{Imm} & \quad \text{Addition Immediate}
\end{align*}
\]

Put the absolute value of the integer from register Rsr in register Rdest.

Put the sum of the integers from register Rsr1 and Src2 (or Imm) into register Rdest.

\(^1\)In earlier version of SPIM, $sp was documented as pointing at the first free word on the stack (not the last word of the stack frame). Recent MIPS documents have made it clear that this was an error. Both conventions work equally well, but we choose to follow the real system.

\(^2\)The MIPS compiler does not use a frame pointer, so this register is used as callee-saved register $s8.
and Rdest, Rsrc1, Src2  
and Rdest, Rsrc1, Imm  

Put the logical AND of the integers from register Rsrcl and Src2 (or Imm) into register Rdest.

div Rsrcl, Src2  
divu Rsrcl, Src2  

Divide the contents of the two registers. divu treats the operands as unsigned values. Leave the quotient in register lo and the remainder in register hi. Note that if an operand is negative, the remainder is unspecified by the MIPS architecture and depends on the conventions of the machine on which SPIM is run.

div Rdest, Rsrcl, Src2  
divu Rdest, Rsrcl, Src2  

Put the quotient of the integers from register Rsrcl and Src2 into register Rdest. divu treats the operands as unsigned values.

mul Rdest, Rsrcl,Src2  
mulou Rdest, Rsrcl, Src2  

Put the product of the integers from register Rsrcl and Src2 into register Rdest.

mul Rsrcl , Src2  
multu Rsrcl, Src2  

Multiply the contents of the two registers. Leave the low-order word of the product in register lo and the high-word in register hi.

negu Rdest, Rsrcl  

Put the negative of the integer from register Rsrcl into register Rdest.

nor Rdest, Rsrcl, Src2  
nor Rdest, Rsrcl, Src2  

Put the logical NOR of the integers from register Rsrcl and Src2 into register Rdest.

not Rdest, Rsrcl  

Put the bitwise logical negation of the integer from register Rsrcl into register Rdest.

or Rdest, Rsrcl, Src2  
or Rdest, Rsrcl, Src2  

Put the logical OR of the integers from register Rsrcl and Src2 (or Imm) into register Rdest.

rem Rdest, Rsrcl, Src2  
remu Rdest, Rsrcl, Src2  

Put the remainder from dividing the integer in register Rsrcl by the integer in Src2 into register Rdest. Note that if an operand is negative, the remainder is unspecified by the MIPS architecture and depends on the conventions of the machine on which SPIM is run.

subu Rdest, Rsrcl, Src2  

Put the difference of the integers from register Rsrcl and Src2 into register Rdest.

xor Rdest, Rsrcl, Src2  
xori Rdest, Rsrcl, Src2  

Put the logical XOR of the integers from register Rsrcl and Src2 (or Imm) into register Rdest.

7
2.4 Constant-Manipulating Instructions

li Rdest, imm
Move the immediate imm into register Rdest.

2.5 Comparison Instructions

In all instructions below, Src2 can either be a register or an immediate value (a 16 bit integer).

seq Rdest, Rsrc1, Src2
Set register Rdest to 1 if register Rsrc1 equals Src2 and to 0 otherwise.

sge Rdest, Rsrc1, Src2
Set Greater Than Equal

sgeu Rdest, Rsrc1, Src2
Set Greater Than Equal Unsigned

sgt Rdest, Rsrc1, Src2
Set Greater Than

sgtu Rdest, Rsrc1, Src2
Set Greater Than Unsigned

sle Rdest, Rsrc1, Src2
Set Less Than Equal

sleu Rdest, Rsrc1, Src2
Set Less Than Equal Unsigned

slt Rdest, Rsrc1, Src2
Set Less Than

slti Rdest, Rsrc1, Imm
Set Less Than Immediate

sltu Rdest, Rsrc1, Src2
Set Less Than Unsigned

sltiu Rdest, Rsrc1, Imm
Set Less Than Unsigned Immediate

sne Rdest, Rsrc1, Src2
Set Not Equal

2.6 Branch and Jump Instructions

In all instructions below, Src2 can either be a register or an immediate value (integer). Branch instructions use a signed 16-bit offset field; hence they can jump $2^{15} - 1$ instructions (not bytes) forward or $2^{15}$ instructions backwards. The jump instruction contains a 26 bit address field.

b label
Unconditionally branch to the instruction at the label.

beq Rsrc1, Src2, label
Conditionally branch to the instruction at the label if the contents of register Rsrc1 equals Src2.

beqz Rsrc, label
Conditionally branch to the instruction at the label if the contents of Rsrc equals 0.

bge Rsrc1, Src2, label
Branch on Greater Than Equal

bgeu Rsrc1, Src2, label
Branch on GTE Unsigned
Conditionally branch to the instruction at the label if the contents of register \texttt{Rsrc1} are greater than or equal to \texttt{Src2}.

\begin{flushleft}
\textbf{bgez Rsrc, label} \quad \textit{Branch on Greater Than Equal Zero}
\end{flushleft}

Conditionally branch to the instruction at the label if the contents of \texttt{Rsrc} are greater than or equal to 0.

\begin{flushleft}
\textbf{bgezal Rsrc, label} \quad \textit{Branch on Greater Than Equal Zero And Link}
\end{flushleft}

Conditionally branch to the instruction at the label if the contents of \texttt{Rsrc} are greater than or equal to 0. Save the address of the next instruction in register 31.

\begin{flushleft}
\textbf{bgt Rsrc1, Src2, label} \quad \textit{Branch on Greater Than} \hfill \dagger
\textbf{bgtu Rsrc1, Src2, label} \quad \textit{Branch on Greater Than Unsigned} \hfill \dagger
\end{flushleft}

Conditionally branch to the instruction at the label if the contents of register \texttt{Rsrc1} are greater than \texttt{Src2}.

\begin{flushleft}
\textbf{bgtz Rsrc, label} \quad \textit{Branch on Greater Than Zero}
\end{flushleft}

Conditionally branch to the instruction at the label if the contents of \texttt{Rsrc} are greater than 0.

\begin{flushleft}
\textbf{ble Rsrc1, Src2, label} \quad \textit{Branch on Less Than Equal} \hfill \dagger
\textbf{bleu Rsrc1, Src2, label} \quad \textit{Branch on LTE Unsigned} \hfill \dagger
\end{flushleft}

Conditionally branch to the instruction at the label if the contents of register \texttt{Rsrc1} are less than or equal to \texttt{Src2}.

\begin{flushleft}
\textbf{blez Rsrc, label} \quad \textit{Branch on Less Than Equal Zero}
\end{flushleft}

Conditionally branch to the instruction at the label if the contents of \texttt{Rsrc} are less than or equal to 0.

\begin{flushleft}
\textbf{bgezal Rsrc, label} \quad \textit{Branch on Greater Than Equal Zero And Link}
\textbf{bltzal Rsrc, label} \quad \textit{Branch on Less Than And Link}
\end{flushleft}

Conditionally branch to the instruction at the label if the contents of \texttt{Rsrc} are greater or equal to 0 or less than 0, respectively. Save the address of the next instruction in register 31.

\begin{flushleft}
\textbf{blt Rsrc1, Src2, label} \quad \textit{Branch on Less Than} \hfill \dagger
\textbf{bltu Rsrc1, Src2, label} \quad \textit{Branch on Less Than Unsigned} \hfill \dagger
\end{flushleft}

Conditionally branch to the instruction at the label if the contents of register \texttt{Rsrc1} are less than \texttt{Src2}.

\begin{flushleft}
\textbf{bltz Rsrc, label} \quad \textit{Branch on Less Than Zero}
\end{flushleft}

Conditionally branch to the instruction at the label if the contents of \texttt{Rsrc} are less than 0.

\begin{flushleft}
\textbf{bne Rsrc1, Src2, label} \quad \textit{Branch on Not Equal}
\end{flushleft}

Conditionally branch to the instruction at the label if the contents of register \texttt{Rsrc1} are not equal to \texttt{Src2}.

\begin{flushleft}
\textbf{bnez Rsrc, label} \quad \textit{Branch on Not Equal Zero} \hfill \dagger
\end{flushleft}

Conditionally branch to the instruction at the label if the contents of \texttt{Rsrc} are not equal to 0.

\begin{flushleft}
\textbf{j label} \quad \textit{Jump}
\end{flushleft}

Unconditionally jump to the instruction at the label.
2.7 Load Instructions

`la Rdest, address`  
Load computed `address`, not the contents of the location, into register `Rdest`.  

`lw Rdest, address`  
Load the 32-bit quantity (word) at `address` into register `Rdest`.  

`lwc z Rdest, address`  
Load the word at `address` into register `Rdest` of coprocessor `z` (0–3).  

`lw l Rdest, address`  
`lw r Rdest, address`  
Load the left (right) bytes from the word at the possibly-unaligned `address` into register `Rdest`.  

2.8 Store Instructions

`sw Rsrc, address`  
Store the word from register `Rsrc` at `address`.  

`swc z Rsrc, address`  
Store the word from register `Rsrc` of coprocessor `z` at `address`.  

`sw l Rsrc, address`  
`sw r Rsrc, address`  
Store the left (right) bytes from register `Rsrc` at the possibly-unaligned `address`.  

2.9 Data Movement Instructions

`move Rdest, Rsrc`  
Move the contents of `Rsrc` to `Rdest`.  

The multiply and divide unit produces its result in two additional registers, `hi` and `lo`. These instructions move values to and from these registers. The multiply, divide, and remainder instructions described above are pseudoinstructions that make it appear as if this unit operates on the general registers and detect error conditions such as divide by zero or overflow.

`mfhi Rdest`  
`mflo Rdest`  
Move the contents of the `hi` (lo) register to register `Rdest`.  

`mthi Rdest`  
`mtlo Rdest`  
Move the contents register `Rdest` to the `hi` (lo) register.  

Coprocessors have their own register sets. These instructions move values between these registers and the CPU’s registers.
mfcz Rdest, CPsrc
Move the contents of coprocessor z's register CPsrc to CPU register Rdest.

mfc1.d Rdest, FRsrc1
Move Double From Coprocessor 1
Move the contents of floating point registers FRsrc1 and FRsrc1 + 1 to CPU registers Rdest and Rdest + 1.

mtcz Rsrpc, CPdest
Move To Coprocessor z
Move the contents of CPU register Rsrpc to coprocessor z's register CPdest.

2.10 Floating Point Instructions

The MIPS has a floating point coprocessor (numbered 1) that operates on single precision (32-bit) and double precision (64-bit) floating point numbers. This coprocessor has its own registers, which are numbered $f0–f31$. Because these registers are only 32-bits wide, two of them are required to hold doubles. To simplify matters, floating point operations only use even-numbered registers—including instructions that operate on single floats.

Values are moved in or out of these registers a word (32-bits) at a time by lwc1, swc1, mtc1, and mfc1 instructions described above or by the l.s, l.d, s.s, and s.d pseudoinstructions described below. The flag set by floating point comparison operations is read by the CPU with its bc1t and bc1f instructions.

In all instructions below, FRdest, FRsrc1, FRsrc2, and FRsrc are floating point registers (e.g., $f2$).

abs.d FRdest, FRsrc
Floating Point Absolute Value Double
Compute the absolute value of the floating float double in register FRsrc and put it in register FRdest.

add.d FRdest, FRsrc1, FRsrc2
Floating Point Addition Double
Compute the sum of the floating float doubles in registers FRsrc1 and FRsrc2 and put it in register FRdest.

c.eq.d FRsrc1, FRsrc2
Compare Equal Double
Compare the floating point double in register FRsrc1 against the one in FRsrc2 and set the floating point condition flag true if they are equal.

c.le.d FRsrc1, FRsrc2
Compare Less Than Equal Double
Compare the floating point double in register FRsrc1 against the one in FRsrc2 and set the floating point condition flag true if the first is less than or equal to the second.

c.lt.d FRsrc1, FRsrc2
Compare Less Than Double
Compare the floating point double in register FRsrc1 against the one in FRsrc2 and set the condition flag true if the first is less than the second.
cvt.d.s FRdest, FRsrc  Convert Single to Double
        Convert the single precision floating point number or integer in register FRsrc to a double precision number and put it in register FRdest.

cvt.d.w FRdest, FRsrc  Convert Integer to Double
        Convert the single precision floating point number or integer in register FRsrc to a double precision number and put it in register FRdest.

cvt.s.d FRdest, FRsrc  Convert Double to Single
        Convert the double precision floating point number or integer in register FRsrc to a single precision number and put it in register FRdest.

cvt.s.w FRdest, FRsrc  Convert Integer to Single
        Convert the double precision floating point number or integer in register FRsrc to a single precision number and put it in register FRdest.

cvt.w.d FRdest, FRsrc  Convert Double to Integer
        Convert the double or single precision floating point number in register FRsrc to an integer and put it in register FRdest.

cvt.w.s FRdest, FRsrc  Convert Single to Integer
        Convert the double or single precision floating point number in register FRsrc to an integer and put it in register FRdest.

div.d FRdest, FRsrc1, FRsrc2  Floating Point Divide Double
        Compute the quotient of the floating double (singles) in registers FRsrc1 and FRsrc2 and put it in register FRdest.

div.s FRdest, FRsrc1, FRsrc2  Floating Point Divide Single
        Compute the quotient of the floating double (singles) in registers FRsrc1 and FRsrc2 and put it in register FRdest.

l.d FRdest, address  Load Floating Point Double †
        Load the floating double (single) at address into register FRdest.

l.s FRdest, address  Load Floating Point Single †
        Load the floating single (single) at address into register FRdest.

mov.d FRdest, FRsrc  Move Floating Point Double
        Move the floating double (single) from register FRsrc to register FRdest.

mov.s FRdest, FRsrc  Move Floating Point Single
        Move the floating single (single) from register FRsrc to register FRdest.

mul.d FRdest, FRsrc1, FRsrc2  Floating Point Multiply Double
        Compute the product of the floating double (singles) in registers FRsrc1 and FRsrc2 and put it in register FRdest.

mul.s FRdest, FRsrc1, FRsrc2  Floating Point Multiply Single
        Compute the product of the floating double (singles) in registers FRsrc1 and FRsrc2 and put it in register FRdest.

neg.d FRdest, FRsrc  Negate Double
        Negate the floating point double (single) in register FRsrc and put it in register FRdest.

neg.s FRdest, FRsrc  Negate Single
        Negate the floating point single (single) in register FRsrc and put it in register FRdest.

s.d FRdest, address  Store Floating Point Double †
        Store the floating double (single) in register FRdest at address.

s.s FRdest, address  Store Floating Point Single †
        Store the floating single (single) in register FRdest at address.

sub.d FRdest, FRsrc1, FRsrc2  Floating Point Subtract Double
        Compute the difference of the floating double (singles) in registers FRsrc1 and FRsrc2 and put it in register FRdest.

sub.s FRdest, FRsrc1, FRsrc2  Floating Point Subtract Single
        Compute the difference of the floating double (singles) in registers FRsrc1 and FRsrc2 and put it in register FRdest.