Excerpts from SPIM S20: A MIPS R2000 Simulator

"\( \frac{1}{25} \) th the performance at none of the cost"

James R. Larus

Copyright ©1990–1997 by James R. Larus
(This document may be copied without royalties, so long as this copyright notice remains on it.)

0.1 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. Op-codes 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 __start # Must be global
__start: 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:

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

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

.byte b1, ..., bn
Store the n values in successive bytes of memory.

.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.

.globl sym
Declare that symbol sym is global and can be referenced from other files.
```
<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>$a0 = integer</td>
<td></td>
</tr>
<tr>
<td>print_float</td>
<td>2</td>
<td>$f12 = float</td>
<td></td>
</tr>
<tr>
<td>print_double</td>
<td>3</td>
<td>$f12 = double</td>
<td></td>
</tr>
<tr>
<td>print_string</td>
<td>4</td>
<td>$a0 = string</td>
<td></td>
</tr>
<tr>
<td>read_int</td>
<td>5</td>
<td></td>
<td>integer (in $v0)</td>
</tr>
<tr>
<td>read_float</td>
<td>6</td>
<td></td>
<td>float (in $f0)</td>
</tr>
<tr>
<td>read_double</td>
<td>7</td>
<td></td>
<td>double (in $f0)</td>
</tr>
<tr>
<td>read_string</td>
<td>8</td>
<td>$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>
</tbody>
</table>

Table 1: System services.

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

.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.

0.2 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

li $v0, 1       # system call code for print_int
li $a0, 5       # integer to print
syscall        # print it

print_int is passed an integer and prints it on the console. print_float prints a single floating point number. print_double prints a double precision number. print_string is passed a pointer to a null-terminated string, which it writes to the console.

read_int, read_float, and read_double read an entire line of input up to and including the newline. Characters following the number are ignored. read_string has the same semantics as
the Unix library routine \texttt{fgets}. It reads up to \( n - 1 \) characters into a buffer and terminates the string with a null byte. If there are fewer characters on the current line, it reads through the newline and again null-terminates the string.

\texttt{strx} returns a pointer to a block of memory containing \( n \) additional bytes. \texttt{exit} stops a program from running.

1 \hspace{1em} 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.

1.1 \hspace{1em} 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.
<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</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.
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 from functions. Registers $t0–$t9 (8–15, 24, 25) are callee-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. Register $fp (30) is the frame pointer.\(^1\) 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.

### 1.2 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($r), which uses the sum of the immediate (integer) c and the contents of register $r 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 ± 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.

### 1.3 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.

```plaintext
abs Rdest, Rsrcc
```

Put the absolute value of the integer from register Rsrcc in register Rdest.\(^1\)

```
add Rdest, Rsrcc1, Src2
addi Rdest, Rsrcc1, Imm
addu Rdest, Rsrcc1, Src2
```

\(^1\)The MIPS compiler does not use a frame pointer, so this register is used as callee-saved register $s8.
addiu Rdest, Rsrlc, Imm  
*Addition Immediate (without overflow)*
Put the sum of the integers from register Rsrlc and Src2 (or Imm) into register Rdest.

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

div Rsrlc, Src2  
*Divide (signed)*
Divide the contents of the two registers. divu treats is 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.

divu Rdest, Rsrlc, Src2  
*Divide (unsigned)*
Put the quotient of the integers from register Rsrlc and Src2 into register Rdest. divu treats its operands as unsigned values.

mul Rdest, Rsrlc, Src2  
*Multiply (without overflow)*
Put the product of the integers from register Rsrlc and Src2 into register Rdest.

mulu Rdest, Rsrlc, Src2  
*Multiply (with overflow)*
Put the product of the integers from register Rsrlc and Src2 into register Rdest.

mulu Rdest, Rsrlc, Src2  
*Unsigned Multiply (with overflow)*
Put the product of the integers from register Rsrlc and Src2 into register Rdest.

mult Rsrlc, Src2  
*Multiply*
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.

multu Rsrlc, Src2  
*Unsigned Multiply*
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.

neg Rdest, Rsrlc  
*Negate Value (with overflow)*
Put the negative of the integer from register Rsrlc into register Rdest.

negu Rdest, Rsrlc  
*Negate Value (without overflow)*
Put the negative of the integer from register Rsrlc into register Rdest.

nor Rdest, Rsrlc, Src2  
*NOR*
Put the logical NOR of the integers from register Rsrlc and Src2 into register Rdest.

not Rdest, Rsrlc  
*NOT*†
Put the bitwise logical negation of the integer from register Rsrlc into register Rdest.

ori Rdest, Rsrlc, Imm  
*OR Immediate*
Put the logical OR of the integers from register Rsrlc and Src2 (or Imm) into register Rdest.

rem Rdest, Rsrlc, Src2  
*Remainder*†
Put the remainder from dividing the integer in register Rsrlc 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.

remu Rdest, Rsrlc, Src2  
*Unsigned Remainder*†
Put the remainder from dividing the integer in register Rsrlc 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.
rol Rdest, Rsrcl, Src2  
ror Rdest, Rsrcl, Src2  

Rotate the contents of register Rsrcl left (right) by the distance indicated by Src2 and put the result in register Rdest.

sll Rdest, Rsrcl, Src2  
sllv Rdest, Rsrcl, Rsrcl  
sra Rdest, Rsrcl, Src2  
srav Rdest, Rsrcl, Src2  
srl Rdest, Rsrcl, Src2  
srlv Rdest, Rsrcl, Src2

Shift the contents of register Rsrcl left (right) by the distance indicated by Src2 (Rsrcl2) and put the result in register Rdest.

sub Rdest, Rsrcl, Src2  
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, Imm

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

1.4 Constant-Manipulating Instructions

li Rdest, imm  

Load Immediate

Move the immediate imm into register Rdest.

lui Rdest, imm  

Load Upper Immediate

Load the lower halfword of the immediate imm into the upper halfword of register Rdest. The lower bits of the register are set to 0.

1.5 Comparison Instructions

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

seq Rdest, Rsrcl, Src2  

Set Equal

Set register Rdest to 1 if register Rsrcl equals Src2 and to 0 otherwise.

sge Rdest, Rsrcl, Src2  
sgeu Rdest, Rsrcl, Src2

Set Greater Than Equal

Set register Rdest to 1 if register Rsrcl is greater than or equal to Src2 and to 0 otherwise.

sgt Rdest, Rsrcl, Src2  
sgtu Rdest, Rsrcl, Src2

Set Greater Than

Set register Rdest to 1 if register Rsrcl is greater than Src2 and to 0 otherwise.

sle Rdest, Rsrcl, Src2  
sleu Rdest, Rsrcl, Src2

Set Less Than Equal

Set register Rdest to 1 if register Rsrcl is less than or equal to Src2 and to 0 otherwise.
slt Rdest, Rs1, Src2  
Set Less Than

slt i Rdest, Rs1, Imm  
Set Less Than Immediate

slt u Rdest, Rs1, Src2  
Set Less Than Unsigned

slt iu Rdest, Rs1, Imm  
Set Less Than Unsigned Immediate

Set register Rdest to 1 if register Rs1 is less than Src2 (or Imm) and to 0 otherwise.

sne Rdest, Rs1, Src2  
Set Not Equal †

Set register Rdest to 1 if register Rs1 is not equal to Src2 and to 0 otherwise.

1.6 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.  

Branch instruction †

bczt label  
Branch Coprocessor z True

bczf label  
Branch Coprocessor z False

Conditionally branch to the instruction at the label if coprocessor z’s condition flag is true (false).

beq Rs1, Src2, label  
Branch on Equal

Conditionally branch to the instruction at the label if the contents of register Rs1 equals Src2.

beqz Rs1, label  
Branch on Equal Zero †

Conditionally branch to the instruction at the label if the contents of Rs1 equals 0.

bge Rs1, Src2, label  
Branch on Greater Than Equal †

bgeu Rs1, Src2, label  
Branch on GTE Unsigned †

Conditionally branch to the instruction at the label if the contents of register Rs1 are greater than or equal to Src2.

bgez Rs1, label  
Branch on Greater Than Equal Zero

Conditionally branch to the instruction at the label if the contents of Rs1 are greater than or equal to 0.

bgezal Rs1, label  
Branch on Greater Than Equal Zero And Link

Conditionally branch to the instruction at the label if the contents of Rs1 are greater than or equal to 0. Save the address of the next instruction in register 31.

bgt Rs1, Src2, label  
Branch on Greater Than †

bgtu Rs1, Src2, label  
Branch on Greater Than Unsigned †

Conditionally branch to the instruction at the label if the contents of register Rs1 are greater than Src2.

bgtz Rs1, label  
Branch on Greater Than Zero

Conditionally branch to the instruction at the label if the contents of Rs1 are greater than 0.
ble Rscl, Src2, label  
**Branch on Less Than Equal** †
bleu Rscl, Src2, label  
**Branch on LTE Unsigned** †
Conditionally branch to the instruction at the label if the contents of register Rscl are less than or equal to Src2.

blez Rscl, label  
**Branch on Less Than Equal Zero**
Conditionally branch to the instruction at the label if the contents of Rscl are less than or equal to 0.

bgezal Rscl, label  
**Branch on Greater Than Equal Zero And Link**
bltzal Rscl, label  
**Branch on Less Than And Link**
Conditionally branch to the instruction at the label if the contents of Rscl are greater or equal to 0 or less than 0, respectively. Save the address of the next instruction in register 31.

blt Rscl, Src2, label  
**Branch on Less Than** †
bltu Rscl, Src2, label  
**Branch on Less Than Unsigned** †
Conditionally branch to the instruction at the label if the contents of register Rscl are less than Src2.

bltz Rscl, label  
**Branch on Less Than Zero**
Conditionally branch to the instruction at the label if the contents of Rscl are less than 0.

bne Rscl, Src2, label  
**Branch on Not Equal**
Conditionally branch to the instruction at the label if the contents of register Rscl are not equal to Src2.

bnez Rscl, label  
**Branch on Not Equal Zero** †
Conditionally branch to the instruction at the label if the contents of Rscl are not equal to 0.

j label  
**Jump**
Unconditionally jump to the instruction at the label.

jal label  
**Jump and Link**
jalr Rscl  
**Jump and Link Register**
Unconditionally jump to the instruction at the label or whose address is in register Rscl. Save the address of the next instruction in register 31.

jr Rscl  
**Jump Register**
Unconditionally jump to the instruction whose address is in register Rscl.

1.7 Load Instructions

1a Rdest, address  
**Load Address** †
Load computed address, not the contents of the location, into register Rdest.

1b Rdest, address  
**Load Byte**
1bu Rdest, address  
**Load Unsigned Byte**
Load the byte at address into register Rdest. The byte is sign-extended by the 1b, but not the 1bu, instruction.
ld Rddest, address
Load the 64-bit quantity at address into registers Rddest and Rddest + 1.

lh Rddest, address
Load the 16-bit quantity (halfword) at address into register Rddest. The halfword is sign-extended by the lh, but not the lhu, instruction

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

lwcz Rddest, address
Load the word at address into register Rddest of coprocessor z (0–3).

lw1 Rddest, address
Load the left (right) bytes from the word at the possibly-unaligned address into register Rddest.

ulh Rddest, address
Load the 16-bit quantity (halfword) at the possibly-unaligned address into register Rddest. The halfword is sign-extended by the ulh, but not the ulhu, instruction

uw Rddest, address
Load the 32-bit quantity (word) at the possibly-unaligned address into register Rddest.

1.8 Store Instructions

sb Rsrsc, address
Store the low byte from register Rsrsc at address.

sd Rsrsc, address
Store the 64-bit quantity in registers Rsrsc and Rsrsc + 1 at address.

sh Rsrsc, address
Store the low halfword from register Rsrsc at address.

sw Rsrsc, address
Store the word from register Rsrsc at address.

swcz Rsrsc, address
Store the word from register Rsrsc of coprocessor z at address.

swl Rsrsc, address
Store the left (right) bytes from register Rsrsc at the possibly-unaligned address.

ush Rsrsc, address
Store the low halfword from register Rsrsc at the possibly-unaligned address.

usw Rsrsc, address
Store the word from register Rsrsc at the possibly-unaligned address.
1.9 Data Movement Instructions

\texttt{move Rdest, Rsrc} \hspace{1cm} \textit{Move}\dagger

Move the contents of \texttt{Rsrc} to \texttt{Rdest}.

The multiply and divide unit produces its result in two additional registers, \texttt{hi} and \texttt{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.

\texttt{mfhi Rdest} \hspace{1cm} \textit{Move From hi}

\texttt{mflo Rdest} \hspace{1cm} \textit{Move From lo}

Move the contents of the \texttt{hi} (lo) register to register \texttt{Rdest}.

\texttt{mthi Rdest} \hspace{1cm} \textit{Move To hi}

\texttt{mtlo Rdest} \hspace{1cm} \textit{Move To lo}

Move the contents register \texttt{Rdest} to the \texttt{hi} (lo) register.

Coprocessors omitted. Floating Point Instructions Omitted.

2 Calling Convention

The calling convention described in this section is the one used by \texttt{gcc}, not the native MIPS compiler, which uses a more complex convention that is slightly faster.

Figure 2 shows a diagram of a stack frame. A frame consists of the memory between the frame pointer (\texttt{fp}), which points to the word immediately after the last argument passed on the stack, and the stack pointer (\texttt{sp}), which points to the last word in the frame. As typical of Unix systems, the stack grows down from higher memory addresses, so the frame pointer is above stack pointer.

The following steps are necessary to effect a call:

1. Pass the arguments. By convention, the first four arguments are passed in registers \texttt{$a0$–}$a3$ (though simpler compilers may choose to ignore this convention and pass all arguments via the stack). The remaining arguments are pushed on the stack.

2. Save the caller-saved registers. This includes registers \texttt{$t0$–$t9$}, if they contain live values at the call site.

3. Execute a \texttt{jal} instruction.

Within the called routine, the following steps are necessary:

1. Establish the stack frame by subtracting the frame size from the stack pointer.

2. Save the callee-saved registers in the frame. Register \texttt{fp} is always saved. Register \texttt{ra} needs to be saved if the routine itself makes calls. Any of the registers \texttt{$s0$–$s7$} that are used by the callee need to be saved.

3. Establish the frame pointer by adding the stack frame size - 4 to the address in \texttt{sp}.

Finally, to return from a call, a function places the returned value into \texttt{\$r0} and executes the following steps:
Figure 2: Layout of a stack frame. The frame pointer points just below the last argument passed on the stack. The stack pointer points to the last word in the frame.
1. Restore any callee-saved registers that were saved upon entry (including the frame pointer $fp$).

2. Pop the stack frame by adding the frame size to $sp$.

3. Return by jumping to the address in register $ra$.

## 3 Other CSI333 Reference Material

### 3.1 ASCII Table

<table>
<thead>
<tr>
<th>Oct</th>
<th>Dec Hex</th>
<th>Char</th>
<th>Oct</th>
<th>Dec Hex</th>
<th>Char</th>
<th>Oct</th>
<th>Dec Hex</th>
<th>Char</th>
<th>Oct</th>
<th>Dec Hex</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0 00</td>
<td>NUL</td>
<td>040</td>
<td>32 20</td>
<td>SPACE</td>
<td>140</td>
<td>96 60</td>
<td>'</td>
<td>001</td>
<td>1 01</td>
<td>SOH</td>
</tr>
<tr>
<td>002</td>
<td>2 02</td>
<td>STX</td>
<td>042</td>
<td>34 22</td>
<td>'</td>
<td>102</td>
<td>66 42</td>
<td>B</td>
<td>103</td>
<td>67 43</td>
<td>C</td>
</tr>
<tr>
<td>004</td>
<td>4 04</td>
<td>EOT</td>
<td>044</td>
<td>36 24</td>
<td>$</td>
<td>104</td>
<td>68 44</td>
<td>D</td>
<td>105</td>
<td>69 45</td>
<td>E</td>
</tr>
<tr>
<td>006</td>
<td>6 06</td>
<td>ACK</td>
<td>046</td>
<td>38 26</td>
<td>&amp;</td>
<td>106</td>
<td>70 46</td>
<td>F</td>
<td>147</td>
<td>103 67</td>
<td>g</td>
</tr>
<tr>
<td>007</td>
<td>7 07</td>
<td>BEL</td>
<td>047</td>
<td>39 27</td>
<td>'</td>
<td>107</td>
<td>71 47</td>
<td>G</td>
<td>151</td>
<td>105 69</td>
<td>i</td>
</tr>
<tr>
<td>010</td>
<td>8 08</td>
<td>BS</td>
<td>050</td>
<td>40 28</td>
<td>(</td>
<td>110</td>
<td>72 48</td>
<td>H</td>
<td>153</td>
<td>107 6B</td>
<td>k</td>
</tr>
<tr>
<td>011</td>
<td>9 09</td>
<td>HT</td>
<td>051</td>
<td>41 29</td>
<td>)</td>
<td>111</td>
<td>73 49</td>
<td>I</td>
<td>155</td>
<td>109 6D</td>
<td>m</td>
</tr>
<tr>
<td>012</td>
<td>10 0A</td>
<td>LF</td>
<td>052</td>
<td>42 2A</td>
<td>*</td>
<td>112</td>
<td>74 4A</td>
<td>J</td>
<td>157</td>
<td>111 6F</td>
<td>o</td>
</tr>
<tr>
<td>013</td>
<td>11 0B</td>
<td>VT</td>
<td>053</td>
<td>43 2B</td>
<td>+</td>
<td>113</td>
<td>75 4B</td>
<td>K</td>
<td>158</td>
<td>112 71</td>
<td>q</td>
</tr>
<tr>
<td>014</td>
<td>12 0C</td>
<td>FF</td>
<td>054</td>
<td>44 2C</td>
<td>,</td>
<td>114</td>
<td>76 4C</td>
<td>L</td>
<td>162</td>
<td>114 73</td>
<td>s</td>
</tr>
<tr>
<td>015</td>
<td>13 0D</td>
<td>CR</td>
<td>055</td>
<td>45 2D</td>
<td>-</td>
<td>115</td>
<td>77 4D</td>
<td>M</td>
<td>159</td>
<td>116 75</td>
<td>u</td>
</tr>
<tr>
<td>016</td>
<td>14 0E</td>
<td>SO</td>
<td>056</td>
<td>46 2E</td>
<td>.</td>
<td>116</td>
<td>78 4E</td>
<td>N</td>
<td>165</td>
<td>118 77</td>
<td>w</td>
</tr>
<tr>
<td>017</td>
<td>15 0F</td>
<td>SI</td>
<td>057</td>
<td>47 2F</td>
<td>/</td>
<td>117</td>
<td>79 4F</td>
<td>O</td>
<td>167</td>
<td>120 79</td>
<td>y</td>
</tr>
<tr>
<td>020</td>
<td>16 10</td>
<td>DLE</td>
<td>060</td>
<td>48 30</td>
<td>0</td>
<td>120</td>
<td>80 50</td>
<td>P</td>
<td>170</td>
<td>122 80</td>
<td>a</td>
</tr>
<tr>
<td>021</td>
<td>17 11</td>
<td>DC1</td>
<td>061</td>
<td>49 31</td>
<td>1</td>
<td>121</td>
<td>81 51</td>
<td>Q</td>
<td>172</td>
<td>123 82</td>
<td>r</td>
</tr>
<tr>
<td>022</td>
<td>18 12</td>
<td>DC2</td>
<td>062</td>
<td>50 32</td>
<td>2</td>
<td>122</td>
<td>82 52</td>
<td>R</td>
<td>174</td>
<td>125 84</td>
<td>t</td>
</tr>
<tr>
<td>023</td>
<td>19 13</td>
<td>DC3</td>
<td>063</td>
<td>51 33</td>
<td>3</td>
<td>123</td>
<td>83 53</td>
<td>S</td>
<td>176</td>
<td>127 86</td>
<td>v</td>
</tr>
<tr>
<td>024</td>
<td>20 14</td>
<td>DC4</td>
<td>064</td>
<td>52 34</td>
<td>4</td>
<td>124</td>
<td>84 54</td>
<td>T</td>
<td>178</td>
<td>130 88</td>
<td>x</td>
</tr>
<tr>
<td>025</td>
<td>21 15</td>
<td>NAK</td>
<td>065</td>
<td>53 35</td>
<td>5</td>
<td>125</td>
<td>85 55</td>
<td>U</td>
<td>180</td>
<td>132 90</td>
<td>z</td>
</tr>
<tr>
<td>026</td>
<td>22 16</td>
<td>SYN</td>
<td>066</td>
<td>54 36</td>
<td>6</td>
<td>126</td>
<td>86 56</td>
<td>V</td>
<td>182</td>
<td>134 92</td>
<td>}</td>
</tr>
<tr>
<td>027</td>
<td>23 17</td>
<td>ETB</td>
<td>067</td>
<td>55 37</td>
<td>7</td>
<td>127</td>
<td>87 57</td>
<td>W</td>
<td>184</td>
<td>136 94</td>
<td>\</td>
</tr>
<tr>
<td>030</td>
<td>24 18</td>
<td>CAN</td>
<td>070</td>
<td>56 38</td>
<td>8</td>
<td>130</td>
<td>88 58</td>
<td>X</td>
<td>186</td>
<td>138 96</td>
<td>'</td>
</tr>
<tr>
<td>031</td>
<td>25 19</td>
<td>EM</td>
<td>071</td>
<td>57 39</td>
<td>9</td>
<td>131</td>
<td>89 59</td>
<td>Y</td>
<td>188</td>
<td>140 98</td>
<td>_</td>
</tr>
<tr>
<td>032</td>
<td>26 1A</td>
<td>SUB</td>
<td>072</td>
<td>58 3A</td>
<td>:</td>
<td>132</td>
<td>90 6A</td>
<td>Z</td>
<td>190</td>
<td>142 100</td>
<td>a</td>
</tr>
<tr>
<td>033</td>
<td>27 1B</td>
<td>ESC</td>
<td>073</td>
<td>59 3B</td>
<td>;</td>
<td>133</td>
<td>91 6B</td>
<td>[</td>
<td>192</td>
<td>144 102</td>
<td>c</td>
</tr>
<tr>
<td>034</td>
<td>28 1C</td>
<td>FS</td>
<td>074</td>
<td>60 3C</td>
<td>&lt;</td>
<td>134</td>
<td>92 6C</td>
<td>\</td>
<td>194</td>
<td>146 104</td>
<td>e</td>
</tr>
<tr>
<td>035</td>
<td>29 1D</td>
<td>GS</td>
<td>075</td>
<td>61 3D</td>
<td>=</td>
<td>135</td>
<td>93 6D</td>
<td>i</td>
<td>196</td>
<td>148 106</td>
<td>f</td>
</tr>
<tr>
<td>036</td>
<td>30 1E</td>
<td>RS</td>
<td>076</td>
<td>62 3E</td>
<td>&gt;</td>
<td>136</td>
<td>94 6E</td>
<td>~</td>
<td>198</td>
<td>150 108</td>
<td>h</td>
</tr>
<tr>
<td>037</td>
<td>31 1F</td>
<td>US</td>
<td>077</td>
<td>63 3F</td>
<td>?</td>
<td>137</td>
<td>95 6F</td>
<td>_</td>
<td>200</td>
<td>152 110</td>
<td>k</td>
</tr>
</tbody>
</table>
3.2 Hex Digit Table

<table>
<thead>
<tr>
<th>H</th>
<th>Bin</th>
<th>Dec</th>
<th>H</th>
<th>Bin</th>
<th>Dec</th>
<th>H</th>
<th>Bin</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>4</td>
<td>0100</td>
<td>4</td>
<td>8</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
<td>5</td>
<td>0101</td>
<td>5</td>
<td>9</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
<td>6</td>
<td>0110</td>
<td>6</td>
<td>A</td>
<td>1010</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
<td>7</td>
<td>0111</td>
<td>7</td>
<td>B</td>
<td>1011</td>
<td>11</td>
</tr>
</tbody>
</table>

3.3 Powers of 2

\[
\begin{array}{c|c|c}
2^0 & 1 & 1 \\
2^1 & 2 & 10 \\
2^2 & 4 & 100 \\
2^3 & 8 & 1000 \\
2^4 & 16 & 10000 \\
2^5 & 32 & 100000 \\
2^6 & 64 & 1000000 \\
2^7 & 128 & 10000000 \\
2^8 & 256 & 100000000 \\
2^9 & 512 & 1000000000 \\
2^{10} & 1024 & 1K \\
2^{11} & 2048 & 2K \\
2^{12} & 4096 & 4K \\
2^{13} & 8192 & 8K \\
2^{14} & 16384 & 16K \\
2^{15} & 32768 & 32K \\
2^{16} & 65536 & 64K \\
2^{17} & 131072 & 128K \\
2^{18} & 262144 & 256K \\
2^{19} & 524288 & 512K \\
2^{20} & 1048576 & 1Meg \\
2^{21} & 2097152 & 2M \\
2^{22} & 4194304 & 4M \\
2^{23} & 8388608 & 8M \\
2^{24} & 16777216 & 16M \\
2^{25} & 33554432 & 32M \\
2^{26} & 67108864 & 64M \\
2^{27} & 134217728 & 128M \\
2^{28} & 268435456 & 256M \\
2^{29} & 536870912 & 512M \\
2^{30} & 1073741824 & 1Gig \\
2^{31} & 2147483648 & 2G \\
2^{32} & 4294967296 & 4G \\
2^{33} & 8589934592 & 8G \\
2^{34} & 17179869184 & 16G \\
\end{array}
\]