Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.035, Fall 2000 Handout 18— MIPS R2000 Tuesday, October 11

The target platform for the Decaf/Espresso compiler is a simulator called SPIM. SPIM was developed by Jim Larus from the University of Wisconsin-Madison, and much of this text comes directly from his original document, which is in the directory `/mit/6.035/doc/spim-6.2/`.[1]

SPIM simulates the MIPS R2000. This document describes a subset of the instructions that the R2000 provides and that can be used by a Decaf/Espresso compiler. We restricted the instruction set to make your job easier. A later handout will describe the interface to SPIM.

1 Simulation of a Virtual Machine

The MIPS architecture, like that of most RISC computers, is difficult to program directly because of its delayed branches, delayed loads, and restricted address modes. This difficulty is tolerable since these computers were designed to be programmed in high-level languages and so present an interface designed for compilers, not programmers. A good part of the complexity results from delayed instructions. A *delayed branch* takes two cycles to execute. In the second cycle, the instruction immediately following the branch executes. This instruction can perform useful work that normally would have been done before the branch or it can be a `nop` (no operation). Similarly, *delayed loads* take two cycles so the instruction immediately following a load cannot use the value loaded from memory.

MIPS wisely chose to hide this complexity by implementing a *virtual machine* with their assembler. This virtual computer appears to have non-delayed branches and loads and a richer instruction set than the actual hardware. The assembler *reorganizes* (rearranges) instructions to fill the delay slots. It also simulates the additional, *pseudoinstructions* by generating short sequences of actual instructions.

By default, SPIM simulates the richer, virtual machine. It can also simulate the actual hardware. We will describe the virtual machine and only mention in passing features that do not belong to the actual hardware. In doing so, we are following the convention of MIPS assembly language programmers (and compilers), who routinely take advantage of the extended machine. Instructions marked with a dagger (†) are pseudoinstructions.

2 Description of the MIPS R2000

In addition to having a CPU and a memory, a MIPS processor also has a collection of coprocessors that perform ancillary tasks or operate on other types of data such as floating point numbers. However, these coprocessors will not be used in 6.035, and therefore we will ignore them in the remainder of this document.

[1] "SPIM S20: A MIPS R2000 Simulator", by James R. Larus, larus@cs.wisc.edu, Computer Sciences Department, University of Wisconsin–Madison, 1210 West Dayton Street, Madison, WI 53706, USA, 608-262-9519. Copyright 1990, 1991 by James R. Larus. This document may be copied without royalties, so long as this copyright notice remains on it.

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

Table 1: MIPS registers and the convention governing their use.

2.1 CPU Registers

The CPU contains 32 general purpose 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 1 lists the registers and describes their intended use; registers not listed should not be used by a Decaf/Espresso compiler.

Registers `$at` (1), `$k0` (26), and `$k1` (27) are reserved for use by the assembler and operating system. You shouldn't use these registers; many virtual assembly instructions use them behind the scenes.

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 caller-saved registers used for temporary quantities. The MIPS convention says that none of these registers need to be preserved across a function call; functions

need to save their values before doing a call.

Register `$sp` (29) is the stack pointer, which points to the top location on the stack. Register `$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.

Register 16–23 are callee-saved registers. For simple code generation, you may ignore them. If you want to use them, their symbolic names are $s0–$s7. Since these are callee-save registers, if a procedure wants to use one of these registers, it must save the value that was in the register, and restore that value before it returns. We recommend that you do not use these registers (it just adds one more thing to worry about), but if you do use them you must obey the calling convention that they are callee-save registers.

2.2 Byte Order

Processors can number the bytes within a word to make the byte with the lowest number either the leftmost or rightmost one. The convention used by a machine is its *byte order*. MIPS processors can operate with either *big-endian* byte order:

Byte #			
0	1	2	3

or *little-endian* byte order:

Byte #			
3	2	1	0

SPIM operates with both byte orders. SPIM's byte order is determined by the byte order of the underlying hardware running the simulator. On an Intel x86 machine, SPIM is little-endian, while on a Sun SPARC or SGI Indy or O2, SPIM is big-endian.

3 Memory Usage

The organization of memory in MIPS systems is conventional. A program's address space is composed of three parts (see Figure 1).

At the bottom of the user address space (0x400000) is the text segment, which holds the instructions for a program. (The notation 0x preceding a number means that that number is given in hexadecimal.)

Above the text segment is the data segment (starting at 0x1000000), which is divided into two parts. The static data portion contains objects whose size and address are known to the compiler and linker. Immediately above these objects is dynamic data. As a program allocates space dynamically (i.e., by `malloc`), the `sbrk` system call moves the top of the data segment up.

[2]The MIPS compiler does not use a frame pointer, so this register is used as callee-saved register `$s8`.

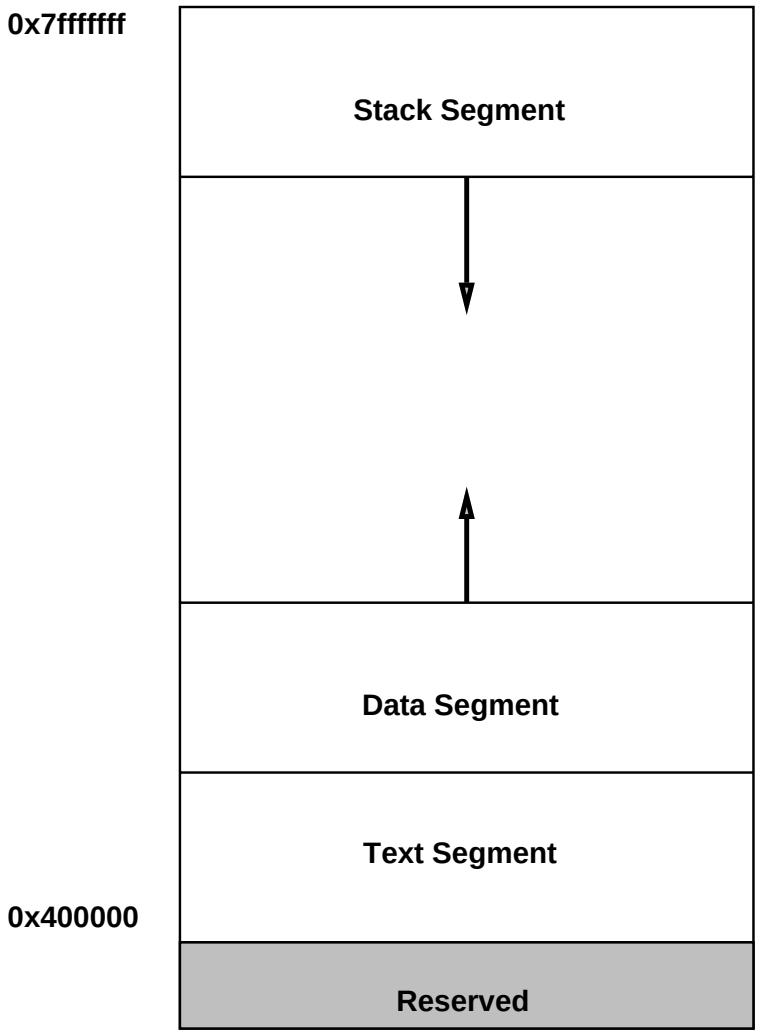

Figure 1: Layout of memory.

The program stack resides at the top of the address space (0x7FFFFFFF). It grows down, towards the data segment.

Your program doesn't need to know these specific addresses, however. The assembler or the simulator can deal with symbolic labels, and will start your program with the correct value of `$sp`. Branches are relative in MIPS code, and jumps to absolute targets are fixed up at link time (or when SPIM reads in the program) to point to the correct address. This means that SPIM can load your program at any address and it will function properly. This will become important later on for running in "native SGI mode."

4 Calling Convention

This section describes the traditional MIPS calling convention, now known as *o32*. Newer SGI compilers implement the *n32* calling convention which has some important differences. Your compiler **must** generate code that uses the o32 standard when doing callouts to external functions (see section 5.3). You're free to use whatever convention you like for internal calls since you control both the caller and callee code, but we suggest you implement o32 or something similar.

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

The MIPS conventions also require that `$sp` be aligned at all times. Under o32, `$sp` must always be a multiple of 8, and under n32 it must be a multiple of 16. If your compiler needs to change `$sp`, it should always do so by a multiple of 8, adding some padding to the stack frame if necessary, to maintain this alignment restriction. The convention also specifies which registers are saved across the function call, described in section 2.1.

The following steps are necessary when making a call:

1. Pass the arguments. By convention, arguments are passed on the stack. The first argument is lowest in memory (top of stack) and the last argument is highest in memory (bottom of stack). You should allocate space for at least four arguments (16 bytes on the stack), even if there are actually fewer than four arguments. Now the callee can access the nth argument by reading $(n * 4)(\$fp)$; that computation does not involve the number of arguments passed (which is important for procedures that take a variable number of arguments, such as `printf()` in C). Furthermore, the convention says that the first four arguments are actually passed in registers `$a0` – `$a3`. You must still reserve space on the stack for these arguments, but the callee will ignore the value on the stack (feel free to leave it uninitialized) and will use the register instead.

2. Save the caller-saved registers. This includes registers `$t0`–`$t9`, if they contain live values at the call site. The caller-saved registers are stored in the caller's activation frame: in other words, they are not pushed on the stack following the arguments, but stored somewhere higher in the stack.

3. Execute a `jal` instruction (see Section 7).

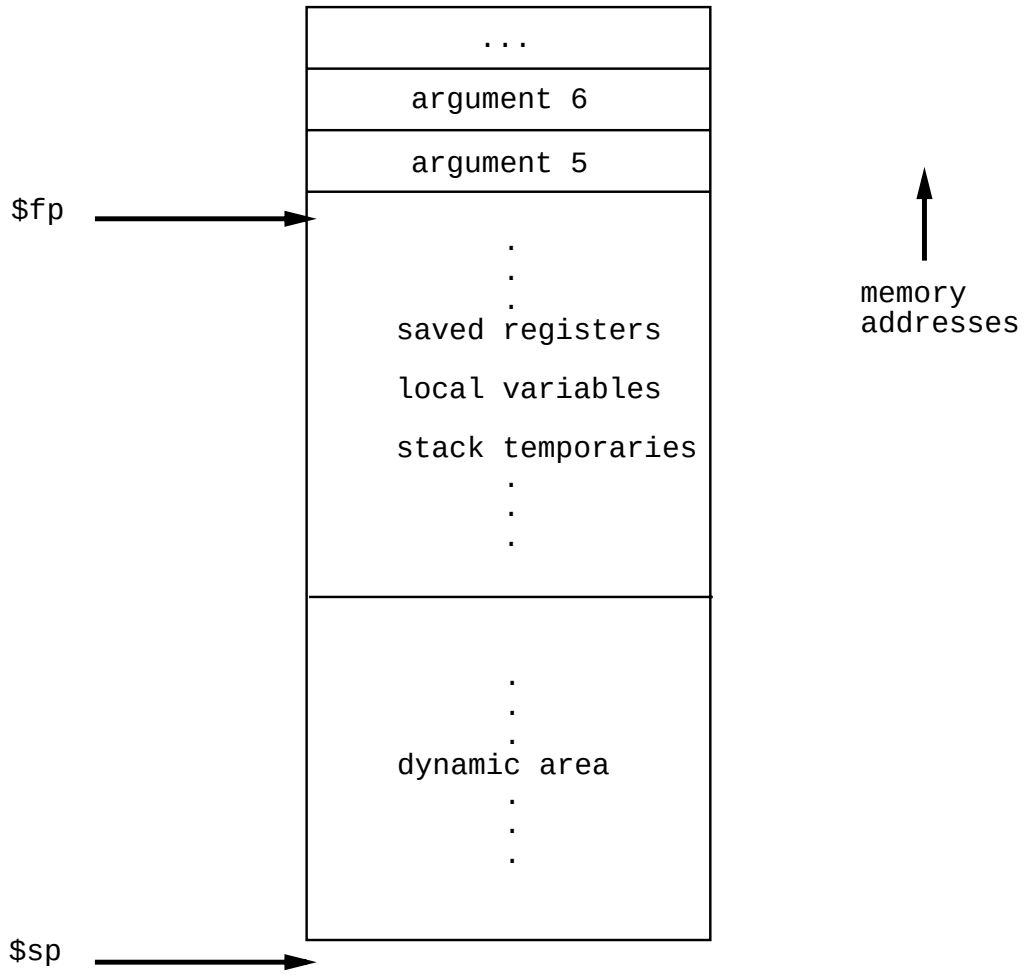

Figure 2: Layout of a stack frame. The frame pointer points just below the last argument passed on the stack. The stack pointer points to the first word after the frame.

Within the called routine, the following steps are necessary:

1. Establish the stack frame by subtracting the frame size from the stack pointer. If the called routine calls another procedure, it will save its registers in its stack frame.
2. Save the callee-saved registers in the frame. Register `$fp` is always saved. Register `$ra` needs to be saved if the routine itself makes calls.
3. Establish the frame pointer by adding the stack frame size to the address in `$sp`.

To return from a call, a function places the returned value into `$v0` and executes the following steps:

1. Restore any callee-saved registers that were saved upon entry.
2. Pop the stack frame by adding the frame size to `$sp`.
3. Return by jumping to the address in register `$ra`.

Finally, when the callee returns, the caller must pop any arguments that were pushed on the stack.

Note that o32 only specifies where the arguments go on the stack, where the return value is found, and which registers are preserved across calls. It says nothing about how a procedure saves temporaries and registers on the stack. For example, because Decaf and Espresso have no facility similar to `alloca()`[3], it is possible to follow o32 without using `$fp` at all.

5 The Assembler

5.1 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          \"
```

[3] `alloca()` allows the programmer to allocate dynamic amounts of memory on the stack at run-time, and pretty much forces the need for some sort of frame pointer to keep track of everything.

5.2 Assembler Directives

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 `.word` directives until the next `.data` directive.

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

`.data`
: The following data items should be stored in the data segment.

`.globl sym`
: Declare that symbol `sym` is global and can be referenced from other files. You'll want to declare your `main` procedure as global, at the very least.

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

`.text`
: 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).

`.word w1, ..., wn`
: Store the n 32-bit quantities in successive memory words.

5.3 Callouts

SPIM provides several operating-system-like services through the system call instruction (`syscall`). With one exception, however, your compilers should not generate code that uses the system calls. Instead, we have defined a "callout" facility that allows your code to call externally-defined functions. Your compilers will use `callout` to do console and file I/O and memory allocation in place of syscalls. The `callout` statement in Decaf and Espresso will also map to a `callout` assembly instruction, allowing programs to use external procedures.

Callouts are accessed using the `callout` assembly language instruction.[4] The format for `callout` is:

```
callout "functionname" num_args
```

where functionname is the name of the external function, and num_args is the number of arguments passed to that function. The arguments should be in the same locations as if callout were a function call. These locations are part of the calling convention, described above. The `callout` instruction

[4] `callout` isn't a real MIPS instruction. We've modified SPIM to do The Right Thing when it sees a callout. If the callout is to a function in the standard C library it will find it, otherwise you need to specify the library to SPIM on startup. If you compile your program to run natively, then the ASM toolkit will rewrite your callout as a normal call (`jal`), and the linker will either find the callpoint or signal an error at link time. More on this in the SPIM handout.

Service	System Call Code	Arguments	Result
print_int	1	`$a0` = integer	
print_string	4	`$a0` = string	
read_int	5		integer (in `$v0`)
read_string	8	`$a0` = buffer, `$a1` = length	
exit	10		

Table 2: System services.

makes an external call to the specified function, and places the return value (if any) into `$v0`. Note that functions that have no explicit return value may still modify `$v0`, and callouts can modify any of the caller-saved registers.

Callouts replace most of the syscalls used in previous years. If your compiler needs to allocate memory, it should load the desired number of bytes into `$a0` and call `malloc` with 1 argument. `malloc` will return a pointer to the new memory in `$v0`. The memory is uninitialized, so you'll need to explicitly clear memory if it needs to contain 0's. The syscalls for reading and writing strings and integers to the console are also no longer supported. Your compiler should instead use `printf` and other C functions for input and output. The staff has provided a compatibility library with `writestring()`, `writeint()`, `writechar()`, `readint()`, and `readchar()`, but feel free to use the more powerful C I/O routines.

SPIM can simulate callout instructions (see the handout on SPIM for more information), but there are a couple of limitations due to the way SPIM simulates MIPS instructions. First, programs using callout cannot allocate more than approximately 1 MB of memory. This is not a problem for Decaf but particularly ambitious Espresso programs may reach this limit. Additionally, if external functions return pointers to allocated memory, your code can not directly access this memory. It may pass the pointer back to another external function. The best example of such a function is `malloc()`, but SPIM contains an internal version of `malloc()` so that your programs can allocate memory. The handout on SPIM contains information on how to work around these limitations. Neither of these limitations affects running native code on SGI machines.

5.4 System Calls

As explained above, you shouldn't need to use syscalls, but we include information on them for completeness. You will need to use the `exit` call to end your program. To request a service, a program loads the system call code (see Table 2) into register `$v0` and the arguments into registers `$a0`...`$a3`. System calls that return values put their result in register `$v0`. 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_string` is passed a pointer to a null-terminated string, which it writes to the console. `read_int` reads an entire line of input up to and including the newline. Characters following the number are ignored. `exit` stops a program from running.

6 Assembler Instructions

MIPS is a load/store architecture, which means that only load and store instructions access memory. Computation instructions operate only on values in registers.

We begin by describing the instructions that allow data to be moved between memory and registers. Then we describe the remaining instructions.

6.1 Load and Store Instructions

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:

Format	Address Computation
(register)	contents of register
imm	immediate
imm (register)	immediate + contents of register
symbol	address of symbol

Note that the first two are variants of the third form: (register) is the same as 0(register) and imm is the same as imm(r0). The fourth form allows the machine location associated with a symbol to be used in a load or store to access the contents of that location.

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. Although MIPS provides some instructions for manipulating unaligned data, we will not use unaligned data in 6.035 and therefore we ignore these instructions.

There are the following load instructions:

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

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

In addition, a constant can be stored in a register:

`li Rdest, imm` *Load Immediate* †
Move the immediate `imm` into register `Rdest`.

Information is moved from a register to memory by:

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

Information can be moved from one register to another by

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

6.2 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.

Many of these instructions are "with overflow". This means that the hardware will detect overflow and raise an exception. In 6.035 we will not generate code that handles such exceptions. Instead if an exception occurs, SPIM will halt execution of the program with an error message.

`add Rdest, Rsrc1, Src2` *Addition (with overflow)*
Put the sum of the integers from register `Rsrc1` and `Src2` (or `Imm`) into register `Rdest`.

`mulo Rdest, Rsrc1, Src2` *Multiply (with overflow)* †
Put the product of the integers from register `Rsrc1` and `Src2` into register `Rdest`.

`sll Rdest, Rsrc1, Src2` *Shift Left Logical*
`srl Rdest, Rsrc1, Src2` *Shift Right Logical*
Shift the contents of register `Rsrc1` left (right) by the distance indicated by `Src2` (`Rsrc2`) and put the result in register `Rdest`.

`sub Rdest, Rsrc1, Src2` *Subtract (with overflow)*
Put the difference of the integers from register `Rsrc1` and `Src2` into register `Rdest`.

6.3 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 Equal* †
Set register `Rdest` to 1 if register `Rsrc1` equals `Src2` and to be 0 otherwise.

`sge Rdest, Rsrc1, Src2` *Set Greater Than Equal* †
Set register `Rdest` to 1 if register `Rsrc1` is greater than or equal to `Src2` and to 0 otherwise.

`sgt Rdest, Rsrc1, Src2` *Set Greater Than* †
Set register `Rdest` to 1 if register `Rsrc1` is greater than `Src2` and to 0 otherwise.

`sle Rdest, Rsrc1, Src2` *Set Less Than Equal* †
Set register `Rdest` to 1 if register `Rsrc1` is less than or equal to `Src2` and to 0 otherwise.

`slt Rdest, Rsrc1, Src2` *Set Less Than*
Set register `Rdest` to 1 if register `Rsrc1` is less than `Src2` (or `Imm`) and to 0 otherwise.

`sne Rdest, Rsrc1, Src2` *Set Not Equal* †
Set register `Rdest` to 1 if register `Rsrc1` is not equal to `Src2` and to 0 otherwise.

7 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` *Branch instruction* †
Unconditionally branch to the instruction at the label.

`beq Rsrc1, Src2, label` *Branch on Equal*
Conditionally branch to the instruction at the label if the contents of register `Rsrc1` equals `Src2`.

`bge Rsrc1, Src2, label` *Branch on Greater Than Equal* †
Conditionally branch to the instruction at the label if the contents of register `Rsrc1` are greater than or equal to `Src2`.

`bgt Rsrc1, Src2, label` *Branch on Greater Than* †
Conditionally branch to the instruction at the label if the contents of register `Rsrc1` are greater than `Src2`.

`ble Rsrc1, Src2, label` *Branch on Less Than Equal* †
Conditionally branch to the instruction at the label if the contents of register `Rsrc1` are less than or equal to `Src2`.

`blt Rsrc1, Src2, label` *Branch on Less Than* †
Conditionally branch to the instruction at the label if the contents of register `Rsrc1` are less than `Src2`.

`bne Rsrc1, Src2, label` *Branch on Not Equal*
Conditionally branch to the instruction at the label if the contents of register `Rsrc1` are not equal to `Src2`.

`j label` *Jump*

Unconditionally jump to the instruction at the label.

`jal label` *Jump and Link*

Unconditionally jump to the instruction at the label. Save the address of the next instruction in register 31.

`jal Rsrc` *Jump and Link*

Unconditionally jump to the instruction whose address is in register `Rsrc`. Save the address of the next instruction in register 31.

`jr Rsrc` *Jump Register*

Unconditionally jump to the instruction whose address is in register `Rsrc`.

7.1 Exception and Trap Instructions

`syscall` *System Call*

Register `$v0` contains the number of the system call (see Table 2) provided by SPIM.

`callout "function_name" num_args` *External Function Call*

Registers `$a0` through `$a3` contain the first four arguments, and the rest are placed on the stack. Does an external call to `function_name`. After the call, `$v0` contains the return value.