Notes on Translating Three-Address Code to MIPS Assembly Code

Saumya Debray
Department of Computer Science
The University of Arizona, Tucson

1 Notes on the MIPS R2000

1.1 General Information

This document describes how to translate 3-address intermediate code to assembly code for the MIPS R2000 processor (as implemented by Jim Larus's SPIM simulator).

Assembly code files should end with the suffix '`.s`'. The SPIM simulator reads in assembly source files, so there is no need to translate to machine code.

Comments can be inserted in the assembler source: a comment is indicated by a '`#`', and extends to the end of the line. It is recommended that you generate comments giving three-address instructions together with your assembly code to simplify debugging.

1.2 The Stack

A stack frame has the structure shown in Figure 1. The stack grows from high addresses towards low addresses.

Two registers are relevant for stack management: the *stack pointer* `$sp` (register 29) and the *frame pointer* `$fp` (register 30). The stack pointer `$sp` points to byte 0 (the high byte) of the top of the stack, i.e., the next available word is at displacement `4($sp)`.

The return address is passed to the callee in register register 31.

1.3 General-Purpose Registers and Memory

The MIPS is a simple load/store architecture, i.e., arithmetic instructions typically operate only on registers. It has 32 general-purpose registers of 32 bits each, numbered 0 through 31. In MIPS assembly language, register i is written `$`i. The value of register 0 (`$0`) is always 0. Registers `$1`, `$26`, and `$27` are reserved for use by the assembler and the OS kernel. Registers `$29` (stack pointer, `$sp`), `$30` (frame pointer, `$fp`), and `$31` (return address register, `$ra`) are used for managing activation records and function calls/returns. The results of integer-valued functions are returned in register `$2` (`$v0`).

Memory is byte addressable in big-endian mode, with 32-bit addresses. All instructions are 32 bits long, and must be aligned.

1.4 Byte Order

The SPIM simulator follows the byte order of the underlying processor. This means that on `lectura`, it is big-endian. That is, byte 0 of a 4-byte word is the leftmost byte of that word (see Figure 1).

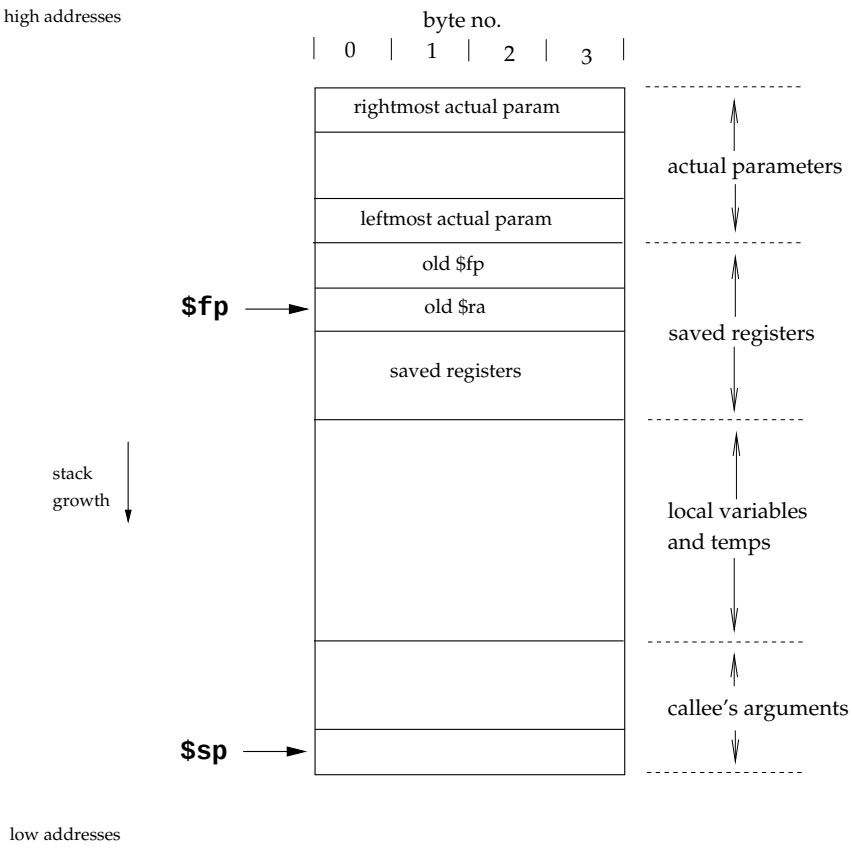

Figure 1: Structure of a stack frame

Note that this may cause programs to produce different results if you run SPIM on a little-endian machine.

1.5 Using the SPIM Simulator

At this time, the SPIM simulator can be invoked on `lectura` by executing

```
/usr/local/bin/spim
```

The simulator will respond with the prompt '`(spim)`', at which point various commands may be executed as described in the SPIM user manual. Alternatively, you can use the X-window interface provided via the command `/usr/local/bin/xspim`.

A typical interactive session might proceed as follows:

(1) Compile the source program into a MIPS assembly file, say `prog.s`.

(2) Invoke SPIM, as described above.

(3) Load and execute the program:

```
(spim) read "prog.s"
(spim) run
```

The `run` command, by default, causes execution of your program to start at label `main`. To exit the simulator, type `quit` or `^D`.

You can also "batch" the execution of a file, say `prog.s`, via the command `spim -file prog.s`.

2 Code Generation

2.1 Data and Text Segments

A set of data declarations must be preceded by the line

```
.data
```

A section of code (i.e., assembly instructions) must be preceded by

```
.text
```

Figure 2 gives an example of the use of these directives.

2.2 Identifiers and Labels

A global identifier *id* in the source program will translate to essentially the same identifier *id* in the assembly code generated, though to avoid inadvertent conflicts with SPIM opcodes, it's recommended that you add an underscore '`_`' in front of each source identifier:

> **Helpful Hint:** Append an underscore '`_`' in front of each source identifier before writing out assembly code. Thus, a source code identifier '`x`' is written out as '`_x`' in the MIPS code you generate. This avoids inadvertent name collisions between source program identifiers and MIPS opcodes like '`b`' and '`j`'.

Local variables will not map to identifiers, but will be accessed via displacements off the frame pointer.

A label is simply an identifier.

Keep in mind that while you will be compiling your program a function at a time, the simulator will see all the labels and identifiers generated in the assembly code output. For this reason, you should be careful to generate labels such that (*i*) no two compiler-generated labels will ever be in conflict; and (*ii*) a compiler-generated label will be unlikely to conflict with an identifier from the user's program. For example, you might consider using a global counter for your labels, so that distinct labels use distinct counter values; and have a leading and trailing pair of underscores on labels—e.g., produce labels such as '`__012__`'—to avoid conflicts with user-defined identifiers.

2.3 Assembler Directives

Space for globals can be generated one identifier at a time. An identifier *id* that occupies *n* bytes of storage is allocated as

id `:  .space` *n*

Alignment restrictions can be enforced using the directive

`.align` *n*

which causes the next data/code to be loaded at an address divisible by 2^n. An `.align` directive is necessary when transitioning from declarations for one or more `char` global variables to a declaration for an `int` global variable.

Example: Consider the following source program fragment, which declares several global variables, with the corresponding assembler directives:

Source Program	Assembler Directive	
`char u, v, w;`	`.data`	
	`u:  .space 1`	
	`v:  .space 1`	
	`w:  .space 1`	
`int x, a;`		
	`.align 2`	(the next variable must be 4-byte-aligned)
	`x:  .space 4`	
	`a:  .space 4`	
`char y;`		
	`y:  .space 1`	

Code and data portions can be intermixed (as long as proper care is taken to align everything properly), as shown in Figure 2. However, avoid splitting the code for a single function into multiple `.text` portions, as this can sometimes cause problems.

2.4 Size Conversions

To load a 1-byte `char` variable at address *addr* into a 32-bit (sign-extended) value in register *reg*, use the instruction '`lb` *reg*`,` *addr*'. To store a 32-bit value in register *reg* into a 1-byte `char` variable at address *addr*, use the instruction '`sb` *addr*`,` *reg*'.

2.5 Accessing Memory

The way in which a memory location is accessed depends on whether it is a global or a local; and if a local, then whether or not it is a formal parameter.

```
int x;              .data
                    x:  .space 4
char y, z           y:  .space 1
                    z:  .space 1
                    .align 2
                    .text
foo()               foo:
{
...                 ( code for foo )
}

                    .data
int a[10];          a:  .space 40

                    .text
bar()               bar:
{
...                 ( code for bar )
}
```

Figure 2: An Example of Code Layout for a Program

2.5.1 Accessing Globals

A scalar global variable can be accessed directly by name, e.g., to load an `int` variable `x` into register 5, we can use

```
lw $5, x
```

2.5.2 Accessing Locals: 1. Actual Parameters

The parameter passing convention described here is considerably simpler (but not as efficient) as that described in the SPIM manual. Here, all parameters are passed on the stack, and the n^{th} parameter to a function ($n \geq 1$, going from left to right) can be accessed from within the called function as k`($fp)`, where $k = 4n + 4$. For example, given a function with three parameters, the leftmost is at `8($fp)`, the middle parameter is at `16($fp)`, and the rightmost is at `20($fp)`. Notice that, in Figure 1, the actuals are at higher addresses than `$fp`. Because of this, actuals are accessed using positive offsets from `$fp`.

2.5.3 Accessing Locals: 2. Non-Parameter Variables

To access a local variable into a register, use the appropriate displacement off the frame pointer `$fp`: a non-parameter local variable at displacement k from the frame pointer is accessed as '`-k($fp)`'. Notice that, in Figure 1, the non-parameter local variables are below the frame pointer, i.e., at lower addresses than `$fp`. Because of this, such variables are accessed using negative offsets from `$fp`.

2.6 Loading Constant Values into Registers

A constant value n that is at most 16 bits wide (i.e., is less than $2^{16} = 65,536$) can be loaded into a register r using the `li` ("load immediate") instruction:[1]

```
li r, n
```

[1]Strictly speaking, '`li`' is a pseudo-instruction: the actual MIPS hardware doesn't have this instruction; the assembler translates the instruction '`li` r`, $0,` n' to '`addi` r`, $0,` n'.

A constant n that occupies between 16 and 32 bits wide can be loaded into a register r using the pair of instructions

```
lui r, n16^hi
ori r, n16^lo
```

where n_{16}^{hi} denotes the high 16 bits of n and n_{16}^{lo} denotes the low 16 bits of n. For example, suppose $r \equiv \$8$ and $n \equiv$ `0xaabbccdd`, then $n_{16}^{hi} \equiv$ `0xaabb` and $n_{16}^{lo} \equiv$ `0xccdd`, and the necessary instruction sequence is:

```
lui $8, 0xaabb
ori $8, 0xccdd
```

2.7 Arithmetic Operations

Arithmetic operations are performed on registers. Shown below is a simple translation scheme (the SPIM manual discusses instructions that are able to use immediate operands that are not more than 16 bits wide: this optimization can result in somewhat more efficient code, but complicates the code generation process somewhat):

`x := y` *op* `z`	*load* `y` *into* reg_1 *load* `z` *into* reg_2 *opc* reg_3, reg_1, reg_2 *store* reg_3 *into* `x`
`x := -y`	*load* `y` *into* reg_1 `neg` reg_2, reg_1 *store* reg_2 *into* `x`

where, for *op* $\in$ {`+, -, *, /`}, *opc* is, respectively, `add`, `sub`, `mul`, and `div`.

Note that multiplication by powers of 2 can be done using a `sll` (shift-left) instruction rather than the more expensive `mul` instruction.

2.8 Conditional and Unconditional Jumps

Unconditional and conditional control transfers can be implemented as follows:

`goto L`	`j L`	the offset of `L` is at most $\pm 2^{26}$ bytes
`if x` *op* `y goto L`	*load* `x` *into* reg_1 *load* `y` *into* reg_2 `b`*cc* reg_1, reg_2, `L`	 the offset of `L` is about $\pm 2^{15}$ instructions

where the condition codes are given by the following:

op	*cc*	*op*	*cc*	*op*	*cc*
`<=`	`le`	`<`	`lt`	`!=`	`ne`
`==`	`eq`	`>`	`gt`	`>=`	`ge`

2.9 Procedures

As with most RISC processors, the MIPS R2000 passes the first few (actually, four) arguments in a procedure call in registers; remaining arguments, if any, are passed on the stack, with the frame pointer `$fp` pointing to the word immediately after the last argument passed on the stack.

For simplicity, we'll adopt a simpler parameter passing convention where all arguments are passed on the stack (if you want you can implement the more efficient scheme described above: the changes necessary to the assembly code described below aren't too hard to figure out). We'll also adopt a convention slightly different from that described in the SPIM manual, and have the `$fp` register point at the leftmost actual parameter on the stack.

2.9.1 Entering a Procedure

On entering a procedure, it is necessary to update the stack and frame pointers, and save the old frame pointer and the return address. For this, we will use the intermediate code instruction

enter f

where f is (a pointer to the symbol table entry of) the procedure being entered. We use the symbol table entry for f to determine the number of bytes n required for its stack frame (and possibly auxiliary information such as any registers that we may want to save on entry to the procedure). The sequence of actions on entry to a procedure are:

1. Set up the frame pointer.
2. Allocate the stack frame by subtracting the size of the stack frame from `$sp`. Since we know that the space occupied by local storage is n bytes, this works out to subtracting n from `$sp`.
3. Save any registers that need to be saved.
   - In general, you'll need to save `$fp` and `$ra`.
   - Additionally, if your compiler is doing any sort of register allocation or code optimization that values to be put into callee-saved registers with the expectation that these values will survive across procedure calls, then the callee must save any callee-saved registers it uses at entry, and restore these before returning to its caller (since the project in CSC 453 doesn't do any register allocation, there are no additional registers to be saved/restored in CSC 453, and this part can be ignored).
4. Allocate space for the rest of the stack frame (locals and temps).

It simplifies things to have the first two words in the area for saved registers be reserved for `$fp` and `$ra`; in this case, assuming that `$sp` is pointing at the topmost word on the stack, i.e., the leftmost actual parameter, it's simplest to first save `$fp` and `$ra`; then set up the frame pointer; and finally update `$sp` to allocate the stack frame. The resulting assembly code is:

```
la $sp, -8($sp)   # allocate space for old $fp and $ra
sw $fp, 4($sp)    # save old $fp
sw $ra, 0($sp)    # save return address
la $fp, 0($sp)    # set up frame pointer
la $sp, −n($sp)   # allocate stack frame:  n = space for locals/temps, in bytes
```

2.9.2 Calling a Procedure

For C programs, actual parameters are pushed from right to left. The relevant three address instructions translate as follows:

`param x` (`x` an `int` or `char`)	*load* `x` *into* reg_1
	`la $sp, -4($sp)`
	`sw` reg_1`, 0($sp)`

The callee does not pop the actual parameters off the stack on return, so this has to be done by the caller. To handle this, we use a three-address instruction

`call` *p, n*

where p is a procedure name and n is the number of arguments. This will translate as follows:

call `p, n`	`jal p` `la $sp,` k`($sp)`

where $k = 4n$ is the number of bytes occupied by the actual parameters.

2.9.3 Return from a Procedure

The return value of a function is put into register `$v0` by the callee. The relevant instructions therefore translate as follows:

`leave` *f*	*restore callee-saved registers, if any* (ignore for CSC 453)	
`return`	`la $sp, 0($fp)`	(deallocate locals)
	`lw $ra, 0($sp)`	(restore return address)
	`lw $fp, 4($sp)`	(restore frame pointer)
	`la $sp, 8($sp)`	(restore stack pointer)
	`jr $ra`	(return)
`return x`	*load* `x` *into* `$v0`	
	`la $sp, 0($fp)`	(deallocate locals)
	`lw $ra, 0($sp)`	(restore return address)
	`lw $fp, 4($sp)`	(restore frame pointer)
	`la $sp, 8($sp)`	(restore stack pointer)
	`jr $ra`	(return)
`retrieve x`	*store* `$v0` *into* `x`	

3 Printing Out Values

The SPIM simulator provides a number of system calls for printing out values of different types: each system call can only deal with printing a value of one particular type. Accordingly, we'll assume that values can be printed out using the following functions, which will be treated specially during code generation:

`println(`n`)` : prints out the integer n.

In order to use this function in a program, it must be declared as follows:

```
extern void println(int n);
```

The code that needs to be generated for a call to this function is described in Section 1.5 of the SPIM manual. The material below describes how to integrate these calls with the parameter passing convention used in this document.

Recall that with the convention we're using for parameter passing, (i) all parameters are passed on the stack; and (ii) the stack pointer points at the last word on the stack that is in use. Since the print routines each take just a single argument, this means that this argument is pushed on top of the stack, and the stack pointer is left pointing at it. Since the SPIM system calls expect the argument in register `$a0`, we need to load it from the stack. Thus, the generated code is as follows:

`println(`n`)` : called with integer n pushed on the stack:

```
.data
__newline:  .asciiz "\n"
.align 2

.text
println:
   li $v0, 1
   lw $a0, 0($sp)
   syscall
   li $v0, 4
   la $a0, __newline
   syscall
   jr $ra
```