Notes on Translating Three-Address Code to MIPS Assembly Code

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

1 A Simple Three-Address Instruction Set

This document considers a simple three-address instruction set and describes its translation to MIPS assembly code:

- **Assignment and copy statements:**
  1. \( x := y \ op \ z \)
     \( \quad \ op \in \{+,-,\ast,\div\}. \)
  2. \( x := -y \)
  3. \( x := y \)

- **Conditional and unconditional jumps:**
  1. if \( x \ relop \ y \) goto \( L \)
     \( L \) is a label; \( relop \in \{\leq, <, >, \geq, ==, !=\}. \)
  2. goto \( L \)
     \( L \) is a label.
  3. label \( L \)
     \( L \) is a label; it must be unique in the program.

- **Instructions for function calls/returns:**
  1. enter \( f \)
     \( f \) is (a pointer to the symbol table entry of) a function.
  2. leave \( f \)
     \( f \) is (a pointer to the symbol table entry of) a function.
  3. param \( x \)
  4. call \( f, n \)
     \( f \) is (a pointer to the symbol table entry of) a function; \( n \) is the number of arguments being passed.
  5. return
     return to the caller.
  6. return \( x \)
     return to the caller with return value \( x \).
  7. retrieve \( x \)
     Copy the return value into \( x \).
2 Notes on the MIPS R2000

2.1 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.

2.2 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.

Figure 1: Structure of a stack frame
2.3 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 little-endian mode, with 32-bit addresses. All instructions are 32 bits long, and must be 4-byte-aligned.

2.4 Byte Order

The SPIM simulator follows the byte order of the underlying processor. This means that on \( \text{lecture} \), it is little-endian. That is, byte 0 of a 4-byte word is the rightmost (least significant) byte of that word.

3 Code Generation

3.1 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.

3.2 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:

Local variables will not map to identifiers, but will be accessed via displacements off the frame pointer register \( $fp \).

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 leading and/or trailing underscores on labels to avoid conflicts with user-defined identifiers.

3.3 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
```
For the CSC 453 project, all variables are of type `int` and therefore occupy 4 bytes each.

String constants can be defined using the `.ascii` and `.asciiz` directives, which store the strings listed in memory as a sequence of characters. String constants declared using `.ascii` are not terminated with a 0 byte, while those declared using `.asciiz` are 0-terminated. Thus, the string constant "x = %d, y = %d\n" can be defined by any of the following:

```
.ascii "x = %d, y = %d\12\0"
.ascii "x = %d, y = %d\n0"
.asciiz "x = %d, y = %d\12"
.asciiz "x = %d, y = %d\n"
```

For the CSC 453 project, string constants will be needed only in generating code for the `println()` function (see Section 4).

Finally, alignment restrictions can be enforced using the directive

```
.align n
```

which causes the next data/code to be loaded at an address divisible by $2^n$.

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

```
Source Program                  Assembler Directive
.int x, a;
       .data
       .align 2
       x: .space 4
       a: .space 4
.int y;
       y: .space 4
```

Code and data portions can be intermixed (as long as proper care is taken to align everything properly), as shown in Figure 2.

### 3.4 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.

#### 3.4.1 Accessing Globals

Global variables are accessed directly by name. To load an `int` variable `x` into register 5, we use

```
lw $5, x
```

To store register 5 into a global variable `y` we use

```
sw $5, y
```

#### 3.4.2 Accessing Locals: 1. Actual Parameters

The parameter passing convention described here is 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 $12(fp)$, and the rightmost is at $16(fp)$. Notice that, in Figure 1, the actuals are at higher addresses than $fp$. Because of this, actuals are accessed using positive offsets from $fp$.

3.4.3 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 $fp$.

3.5 Loading Constant Values into Registers

A constant value $n$ can be loaded into a register $r$ using the `li` (“load immediate”) instruction:\(^1\)

```mips
li r, n
```

3.6 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):

\(^1\)Strictly speaking, ‘li’ is a pseudo-instruction: the actual MIPS hardware doesn’t have this instruction; the assembler translates the instruction ‘li r, n’ to ‘addi r, $0, n’.
\[
\begin{align*}
x := y & \text{ op } z \\
\text{load } y & \text{ into } \text{reg}_1 \\
\text{load } z & \text{ into } \text{reg}_2 \\
\text{opc } & \text{reg}_3, \text{reg}_1, \text{reg}_2 \\
\text{store } \text{reg}_3 & \text{ into } x \\
x := -y & \\
\text{load } y & \text{ into } \text{reg}_1 \\
\text{neg } & \text{reg}_2, \text{reg}_1 \\
\text{store } \text{reg}_2 & \text{ into } x 
\end{align*}
\]

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

### 3.7 Conditional and Unconditional Jumps

Unconditional and conditional control transfers can be implemented as follows:

\[
\begin{align*}
goto & \text{ L} \\
\text{j} & \text{ L} \quad \text{the offset of L is at most } \pm 2^{20} \text{ bytes} \\
\text{if } x & \text{ op } y \text{ goto } \text{L} \\
\text{load } x & \text{ into } \text{reg}_1 \\
\text{load } y & \text{ into } \text{reg}_2 \\
\text{bcc } & \text{reg}_1, \text{reg}_2, \text{L} \quad \text{the offset of L is about } \pm 2^{15} \text{ instructions}
\end{align*}
\]

where the condition codes are given by the following:

\[
\begin{array}{cccccc}
op & \text{cc} & \text{op} & \text{cc} & \text{op} & \text{cc} \\
\leq & \text{le} & < & \text{lt} & != & \text{ne} \\
== & \text{eq} & > & \text{gt} & >= & \text{ge}
\end{array}
\]

### 3.8 Functions

As with most RISC processors, the MIPS R2000 passes the first few (actually, four) arguments in a function 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.

#### 3.8.1 Entering a Function

On entering a function, 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

\[
\text{enter } f
\]

where \( f \) is (a pointer to the symbol table entry of) the function being entered. We use the symbol table entry for \( f \) to determine the number of bytes \( n \) required for its stack frame. The sequence of actions on entry to a function is:

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. For the CSC 453 project, you need to save \( \$fp \) and \( \$ra \).
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:

\[
\begin{align*}
&\text{la } \$sp, -8(\$sp) \quad \# \text{ allocate space for old } \$fp \text{ and } \$ra \\
&\text{sw } \$fp, 4(\$sp) \quad \# \text{ save old } \$fp \\
&\text{sw } \$ra, 0(\$sp) \quad \# \text{ save return address} \\
&\text{la } \$fp, 0(\$sp) \quad \# \text{ set up frame pointer} \\
&\text{la } \$sp, -n(\$sp) \quad \# \text{ allocate stack frame: } n = \text{ space for locals/temps, in bytes}
\end{align*}
\]

3.8.2 Calling a Function

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

\[
\begin{align*}
&\text{param } x \quad \boxed{\text{load } x \text{ into reg}_1} \\
&\quad \text{la } \$sp, -4(\$sp) \\
&\quad \text{sw } \text{reg}_1, 0(\$sp)
\end{align*}
\]

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

\[
\begin{align*}
&\text{call } p, n
\end{align*}
\]

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

\[
\begin{align*}
&\text{call } p, n \quad \boxed{\text{jal } p} \\
&\quad \text{la } \$sp, k(\$sp)
\end{align*}
\]

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

3.8.3 Return from a Function

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

\[
\begin{array}{l|l|l}
\hline
\text{leave } f & \text{restore callee-saved registers, if any (ignore for CSC 453)} \\
\hline
\text{return} & \text{la } \$sp, 0(\$fp) & (\text{deallocate locals}) \\
& \text{lw } \$ra, 0(\$sp) & (\text{restore return address}) \\
& \text{lw } \$fp, 4(\$sp) & (\text{restore frame pointer}) \\
& \text{la } \$sp, 8(\$sp) & (\text{restore stack pointer}) \\
& \text{jr } \$ra & (\text{return}) \\
\hline
\text{return } x & \text{load } x \text{ into } v0 \\
& \text{la } \$sp, 0(\$fp) & (\text{deallocate locals}) \\
& \text{lw } \$ra, 0(\$sp) & (\text{restore return address}) \\
& \text{lw } \$fp, 4(\$sp) & (\text{restore frame pointer}) \\
& \text{la } \$sp, 8(\$sp) & (\text{restore stack pointer}) \\
& \text{jr } \$ra & (\text{return}) \\
\hline
\text{retrieve } x & \text{store } v0 \text{ into } x
\end{array}
\]
4 Printing Out Values

For the purposes of this project we will assume the existence of a special function, \texttt{println}(x), that can be used to print out an integer value. For example, an input program might be:

```c
int main() {
    int x;
    x = 123;
    println(x);
}
```

When the MIPS code generated for this program is executed, it should print out

```
123
```

To accommodate this, in addition to the code generated for the input program, your compiler should generate the MIPS instructions for \texttt{println()} given below.

Recall that with the convention we’re using for parameter passing, \textit{(i)} all parameters are passed on the stack; and \textit{(ii)} the stack pointer points at the last word on the stack that is in use. Since \texttt{println()} takes one 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 \texttt{$a0$}, we need to load it from the stack.

The code you should generate is as follows:

\texttt{println(n) : called with integer n pushed on the stack:}

```
.align 2
.data
_nl: .asciiz "\n"
.align 2
.text
println:
    li $v0, 1
    lw $a0, 0(sp)
    syscall
    li $v0, 4
    la $a0, _nl
    syscall
    jr $ra
```

5 Some Helpful Hints

1. For each three-address instruction that is processed, writing out the three-address instruction as a comment alongside the generated MIPS code can help simplify debugging.

2. 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’.

If you do this, the function \texttt{main} will get renamed to \texttt{\_main}. Since SPIM begins execution at the label \texttt{main}, you should therefore add the following line to the code you generate:

```
main : j \_main
```
Appendix: Using the SPIM Simulator

1. Basic usage

At this time, the SPIM simulator can be invoked on your computer 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. Debugging

SPIM has some simple debugging facilities, including setting breakpoints and single-stepping through code, that can be very helpful for identifying problems in the assembly code generated by your compiler. See Section 1.2.1 (Terminal Interface) of the SPIM User Manual.