MC97 Basic Run-time Environment (for Labs)

March 22, 1996; Max Hailperin

This handout covers just the basics, temporarily assuming the only type is `int` and that no procedure takes more than four arguments. It also leaves room for optimization. (You should start thinking about extensions and optimizations.) This handout describes a set of conventions that are compatible with the SPIM simulator and with MIPS conventions regarding register usage. However, this handout does *not* cover all of the MIPS standard conventions, and code generated as described here will work only under SPIM and not (unless you do a little touch-up work) on the real live MIPS machines we have (the SGIs). Anyone who is interested is encouraged to look at the on-line books on the SGIs: *MIPSpro Assembly Language Programmer's Guide*, *MIPS Compiling and Performance Tuning Guide*, and *MIPSpro Compiling, Debugging, and Performance Tuning Guide*. You are also welcome to talk with me about these issues, of course.

Framework The assembly language file your compiler generates should consist of a chunk of assembly language code for each global variable definition and for each procedure definition. The sections below describe these chunks of code. There is no "prologue" or "epilogue" needed at the beginning or end of the file; it can simply begin with the first chunk of code and end after the last chunk.

Global variable definitions A global variable declaration indicates that a global variable defined in another file is usable in the current file as well; it doesn't actually allocate any memory space for the variable. A global variable definition, on the other hand, actually allocates the space. In C, a global variable declaration looks like `extern int x;` while a global variable definition looks like `int x;`. For the declaration, no assembly code is needed, while for the definition, the chunk of code you emit would be:

```
        .data
        .globl  x
x:      .word   0
        .text
```

(The 0 is the initial value, which is conventionally 0 unless there is an explicit initializer on the definition.)

Procedure definitions A procedure definition results in a chunk of code being output that consists of a "procedure prologue" at the beginning, then code for each of the statements in the procedure, and then a "procedure epilogue" at the end. You should generate the same format of code for all procedure definitions, including `main`. The format described here is slightly different than the one we have used in prior labs for `main`, because before we were "cheating" and getting away with it, while now we will do it right.

For a procedure called foo with no local variables (including no arguments), the prologue would look like this:

```
        .globl  foo
foo:
        subu    $sp, 80
        sw      $ra, 76($sp)
```

If the procedure had any local variables (including arguments), at the end there would be one additional `sw` line for each local variable declared. Suppose there are two local variables, stored in registers `$s0` and `$s1`. Then the extra `sw` lines could be:

```
        sw $s0, 72($sp)
        sw $s1, 68($sp)
```

In general, the offsets in the `sw` decrease by 4 each time.

Once the `sw` instructions have saved the caller's values from these registers, any arguments the procedure received should be moved from the registers they arrived in, which are `$a0`, `$a1`, etc., into the appropriate `$s` registers.

After the procedure prologue, there is a chunk of code for each instruction. These should be familiar to you from prior labs, with the exception of the code for procedure calls, return statements, and global variable accesses and assignments. These three special cases are described in seperate subsections below.

Finally, the procedure ends with a procedure epilogue. At the end of the hypothetical `foo` procedure with no local variables (including no arguments), the epilogue would look like:

```
foo$ret:
        lw      $ra, 76($sp)
        addu    $sp, 80
        j       $ra
```

If `foo` instead had used registers `$s0` and `$s1` for local variables, then the epilogue would look like:

```
foo$ret:
        lw      $ra, 76($sp)
        lw      $s0, 72($sp)
        lw      $s1, 68($sp)
        addu    $sp, 80
        j       $ra
```

In general, an `lw` instruction needs to be included to correspond with each of the `sw` instructions in the prologue. This restores the calling procedure's values to these registers.

Return statements To compile a return statement where there is no value to return (in a procedure declared to return "void"), you would just emit the instruction

```
j       foo$ret
```

assuming that the return statement occurs in the procedure named `foo`.

If, on the other hand, the return statement includes an expression whose value should be returned, then that value needs to first be computed and placed into `$v0`. Then the jump to the epilogue (i.e., the `j foo$ret` instruction) can be done as above.

Procedure calls When you are compiling the code for a statement and encounter a procedure call, there are four things that you need to generate:

1. The code the compute the argument values.
2. The code to save to the stack any temporary registers that contain values you'll need after the call returns. (If the procedure call is an entire statement, then there won't be any temporary registers to save. However, if the procedure call appears as part of an expression, for example in the statement `x = y + z + f(x);`, then there may be temporary registers to save, such as the one holding the sum of `x` and `y` in this example.)
3. The procedure call itself.
4. The code to restore from the stack any saved temporary registers.

If the procedure call produces a return value, you will also need to move that from `$v0` to somewhere else.

To compute the argument values, it is best if you first compute the value of each argument expression into a temporary register (optionally skipping over any expressions that are constant or just a single variable), and then once all have been evaluated, move the values into `$a0`, `$a1`, ... This avoids complications when a procedure call appears in the middle of an argument to another procedure call, as in `f(x+y, g(2), 3)`.

For each temporary register that needs saving, you would emit an `sw` operation. For example, if `$t0` and `$t1` need saving, you'd emit:

```
sw      $t0, 0($sp)
sw      $t1, 4($sp)
```

The offsets from the stack pointer here start at 0 and work up by fours. The order in which you save the registers is unimportant.

For the procedure call itself, supposing that you were calling our favorite procedure, `foo`, you'd emit:

```
jal     foo
```

Restoring the temporary registers comes next, and it would be just like saving them except with `lw` in place of `sw`.

Global variable usage To make use of a global variable's value in an expression, it needs to be loaded into a register, using the `lw` operation. For example, given a global variable `x`, you might do

```
lw      $t0, x
```

Conversely, to assign a new value to a global variable, you would do something like

```
sw      $t0, x
```