



Topic Notes: MIPS Programming

We spent some time looking at the MIPS Instruction Set Architecture. We will now consider how to program in MIPS assembly language.

Our arsenal of MIPS assembly instructions now include:

- `add $a, $b, $c`
- `sub $a, $b, $c`
- `addi $a, $b, n`
- `lw $a, n($b)`
- `sw $a, n($b)`
- `sll $a, $b, n`
- `srl $a, $b, n`
- `and $a, $b, $c`
- `andi $a, $b, n`
- `or $a, $b, $c`
- `ori $a, $b, n`
- `nor $a, $b, $c`
- `beq $a, $b, L`
- `bne $a, $b, L`
- `j L`

Control Structures in MIPS Assembly

Suppose we want to implement a `while` loop in MIPS assembly. Consider the high-level language code:

```
while (a!=b)
  a += i;
```

where a is stored in register \$s0, b in \$s1 and the increment i is in \$s2.

Our MIPS assembly code might look like this:

```
Loop:   beq     $s0, $s1, EndLoop      # if a==b goto EndLoop
        add     $s0, $s0, $s2        # a = a + i
        j      Loop                  # goto Loop
EndLoop:
        ... code after loop ...
```

Note that we use a beq instruction here as we want to branch in the case where we no longer wish to execute the contents of the loop.

Now consider a slightly more complicated loop:

```
while (a[i] == k)
  i += j;
```

We need to deal with an array access here. Suppose we have made the following register assignments:

The start of array a is in \$s3, i is in \$s0, k is in \$s1, and j is in \$s2. All are int values.

One way to implement this code would be:

```
LoopTop: sll     $t0, $s0, 2          # t0 = i * 4
        add     $t0, $t0, $s3        # t0 = address of a[i]
        lw      $t1, 0($t0)         # t1 = contents of a[i]
        bne    $t1, $s1, EndLoop     # if a[i] != k, goto EndLoop
        add     $s0, $s0, $s2        # i = i + j
        j      LoopTop              # jump back to the top of the loop
EndLoop:
        ... code after loop ...
```

A few notes:

- We need to multiply i by 4 to get the correct offset, since we're assuming a is an array of word-sized values.
- We might be tempted to write

```
lw      $t1, $t0($s3)
```

to access the value at an offset of `$t0` from our base register `$s3`. But that is not valid MIPS - the offset part of the `lw` and `sw` instructions needs to be a constant, not a register. The MIPS designers could have provided such an instruction (it would be R-format instead of I-format), but they chose not to.

Before we can complete our next example, we need a couple of additional instructions – reading and writing single bytes from memory.

These instructions, `lb` for load byte and `sb` for store byte, work just like the `lw` and `sw` instructions except that only single-byte values are processed.

```
lb      $a, n($b)
```

Loads the single byte at an offset of `n` from register `$b` and stores it, *sign-extended*, into register `$a`.

```
sb      $a, n($b)
```

Stores the byte in the bottom 8 bits of register `$a` into memory at an offset of `n` from register `$b`.

String Processing Examples

Armed with these instructions, we can write our next example: a string copy function like C's `strcpy`:

```
strcpy(dest, src);
```

Recall that C strings are terminated with a null (0) character.

For now, we'll just look at the main loop of this function. Assume register `$s1` holds the address of the start of the `dest` string and that `$s2` holds the address of the start of the `src` string.

Our task is to write a loop that copies characters (bytes) from the source string to the destination string.

```
LoopTop: lb      $t0, 0($s2)      # temp reg = char at $s2
          sb      $t0, 0($s1)      # char at $s1 gets temp reg
          addi   $s2, $s2, 1      # increment $s2
          addi   $s1, $s1, 1      # increment $s1
          bne   $t0, $zero, LoopTop # branch if we didn't just copy a null
```

For another example:

```

char a[11];
... put something in a ...
for (i=0; i<10; i++) {
    a[i+1] = a[i];
}

```

Assuming \$s0 contains the address of a, here's one way to write this:

```

                add    $s0, $zero, $zero        # i=0
ForLoop:       slti   $t1, $s0, 10            # i<10 ?
                beq    $t1, $zero, LoopEnd    # if done, branch out
                add    $t2, $s0, $s1        # t2 gets address of a[i]
                lb     $t3, 0($t2)          # t3 gets a[i]
                addi   $t2, $t2, 1          # t2 gets address of a[i+1]
                sb     $t3, 0($t2)          # a[i+1] gets t3
                addi   $s0, $s0, 1          # i++
                j      ForLoop              # back to check loop condition
LoopEnd:

```

MIPS Subroutines and Programs

You are all familiar with function/method calls in high-level languages. In assembly language, we usually refer to these as *subroutines*.

The idea is the same as a function or method call – the program branches from its sequence of instructions to execute a chunk of code elsewhere, then returns to continue where it left off.

We'll now look at how to write and call subroutines in MIPS assembly.

Special Registers and Instructions

Recall that there were several registers reserved to help support subroutine calls:

- \$a0-\$a4: argument registers – a place for the caller to place values to send to the subroutine.
- \$v0, \$v1: return value registers – a place for subroutines to return values to the caller.
- \$ra: return address register – where to resume executing the caller when the subroutine is finished.

We also have a couple of special jump instructions:

- *Jump and Link*:

```
jal address
```

This instruction puts the address of the next instruction (PC+4) into register `$ra`, then jumps to the address.

This is a J-format instruction, just like the standard jump instruction (`j`).

- *Jump to Register:*

```
jr $a
```

Jumps to the address specified in the given register.

This is an R-format instruction.

Assuming the subroutine hasn't changed the `$ra` register, this can be used to return from the subroutine.

Registers and the Stack

We said previously that the “s” registers `$s0-$s7` are the ones assigned to variables and the “t” registers `$t0-$t7` are temporary values.

This becomes important when we start looking at subroutines. The accepted convention for register use by subroutines:

- `$t0-$t7` are always available for use by a subroutine
 - if a subroutine calls another subroutine, it must assume that the called subroutine will modify `$t0-$t7`.
 - if this is a problem for the calling subroutine, it should save any values it has in `$t0-$t7` to memory and restore them after the subroutine call.
- `$s0-$s7` should be unchanged by a subroutine call
 - if a subroutine calls another subroutine, it can expect its values in `$s0-$s7` to remain upon return.
 - if the called subroutine wishes to make use of `$s0-$s7`, it should save the caller's values in any of these registers it will use in memory, then restore them before return.

Since subroutines can be called by anyone, we don't know which “s” registers, if any, are important to the caller. So we have to save these if we use them.

So where do we save values in memory when we need to save them? On the *stack*.

The stack is a section of memory dedicated to saving registers to manage subroutine calls.

We have a special register `$sp` called the *stack pointer* that indicates the *top* of the stack.

The stack grows and shrinks as registers are saved and restored during a program's execution.

If we have a subroutine that will need to make use of `$s0`, `$s1` and `$s2`, we need to do the following at the start of the subroutine's code:

```

addi    $sp, $sp, -12    # make room for 3 4-byte values
sw      $s0, 0($sp)     # push s0
sw      $s1, 4($sp)     # push s1
sw      $s2, 8($sp)     # push s2

```

Then before returning:

```

lw      $s2, 8($sp)     # pop s2
lw      $s1, 4($sp)     # pop s1
lw      $s0, 0($sp)     # pop s0
addi    $sp, $sp, 12    # restore the stack pointer

```

Note that we “pop” in the opposite order as we “push”.

A First Complete Subroutine

Let's return to our string copy code:

```

LoopTop: lb      $t0, 0($s2)    # temp reg = char at $s2
          sb      $t0, 0($s1)    # char at $s1 gets temp reg
          addi    $s2, $s2, 1    # increment $s2
          addi    $s1, $s1, 1    # increment $s1
          bne    $t0, $zero, LoopTop # branch if we didn't just copy a null

```

In order to make this a subroutine, we need to get values from the subroutine argument registers and save and restore values of any “s” registers we decide to use. Our code becomes:

```

strcpy:  addi    $sp, $sp, -8    # make space for 2 words on the stack
          sw      $s2, 4($sp)    # save s2
          sw      $s1, 0($sp)    # save s1
          add     $s1, $a0, $zero # copy arg0 into s1
          add     $s2, $a1, $zero # copy arg1 into s2

LoopTop: lb      $t0, 0($s2)    # temp reg = char at $s2
          sb      $t0, 0($s1)    # char at $s1 gets temp reg
          addi    $s2, $s2, 1    # increment $s2
          addi    $s1, $s1, 1    # increment $s1

```

```

    bne    $t0, $zero, LoopTop # branch if we didn't just copy a null

    lw     $s1, 0($sp)        # restore s1
    lw     $s2, 4($sp)        # restore s2
    addi   $sp, $sp, 8        # restore sp
    jr     $ra                # return from subroutine

```

Note that we could do something simpler here: save and restore `$a0` and `$a1` and use those in place of `$s1` and `$s2` throughout.

A Recursive Example

We will develop a MIPS assembly subroutine to implement the following C function:

```

int factorial (int x) {
    if (x<1) return 1;
    return x * factorial(x-1);
}

```

Since this subroutine calls another subroutine (itself) we need to save `$ra` and any temp registers we care about before making the recursive call.

We will assume a subroutine `multiply` exists and we will use that to do our multiplication to get extra practice with subroutines.

Here is some code for this:

```

factorial:
    # make space for 2 words on the stack
    addi   $sp, $sp, -8

    # save $ra and $a0 on the stack
    sw     $a0, 4($sp)
    sw     $ra, 0($sp)

    slti   $t0, $a0, 1        # is x < 1?
    beq    $t0, $zero, L1     # if no, skip over if part

    # x >= 1, just return 1
    addi   $v0, $zero, 1      # return value is 1

    # we could restore $a0 and $ra but we know they haven't
    # changed when we take this quick exit, so let's not
    # but we still need to put $sp back the way we found it
    addi   $sp, $sp, 8
    jr     $ra                # return to caller

```

```

# here, x>=1 so we need to make a recursive call
L1:  addi   $a0, $a0, -1    # get x-1 for the parameter
     jal   factorial      # recursive call, will put answer in $v0

# We now want to set up for the multiply, but we destroyed $a0
# above, but have it on the stack, so let's load it
     lw   $a0, 4($sp)
     add  $a1, $v0, $zero # put factorial(x-1) result into $a1
     jal  multiply        # multiply $a0*$a1, put result in $v0

# $v0 is where we want our answer, so no work there
# but multiply could have changed $a0 and did change $ra
     lw   $ra, 0($sp)    # restore $ra
     lw   $a0, 4($sp)    # restore $a0
     addi $sp, $sp, 8    # restore $sp
     jr   $ra            # return to caller

```

Trace through this with the call `factorial(3)`.

Complete Programs and SPIM

We will use a simulator called SPIM to execute MIPS programs.

- SPIM reads MIPS assembly language programs
- SPIM simulates the execution of each instruction
- SPIM displays values of registers and memory
- SPIM allows breakpoints and step-by-step execution
- SPIM provides primitive I/O to allow interactive processing

Three SPIM versions are available:

- `spim` – the command-line version
- `xspim` – the X11 version (Macs, Unix workstations)
- `PCspim` – Windows version

We will grade submissions with `spim`, but you are welcome to use any version you wish when developing MIPS programs.

A complete program consists of several parts, including the MIPS assembly code we have been considering in our previous examples.

Our first complete example will show a few of the parts of a MIPS program beyond what we have seen.

See: `/cs/terescoj/shared/cs2500/examples/spim-simple/simple.s`

Things to notice about this example:

- Comments in MIPS assembly are any part of a line after a # character.
- We can define variables in the `.data` portion of the program.
- Here, we define three word-sized variables with initial values. The names are the labels, the initial values are given after the `.word` directive. These labels are *local labels*, meaning that they exist only within this assembly source file.
- We also define an uninitialized word-sized variable with the `.space` directive and 4 for the number of bytes of space to allocate.
- The program is defined in the `.text` portion – the “code section”.
- We include `.align 2` to make sure our code starts on a word boundary. (Note that `.align n` forces the next defined item to align on a 2^n -byte boundary.)
- `.globl main` makes the `main` label an *external*, or *global label*, which allows the `main` subroutine to be callable from outside of this file (in this case, by the simulator).
- The program itself is defined starting at the `main:` label.
- Note that we can use `lw` and `sw` to load registers from and store registers to named memory locations.
- Our first example of SPIM’s primitive I/O is this `syscall`. `syscall` can perform one of several functions, as determined by the value in `$v0`. Here, we have a 1 in `$v0`, which specifies `print_int`. The int to print must be placed in `$a0`.
- Finally, when `main` completes, we return from the subroutine with `jr $ra`.

To run this at the command line:

```
spim simple.s
```

This should print our answer.

We can learn much more about the execution of the program running SPIM in interactive mode.

Run SPIM without any command-line parameters to get the (`spim`) prompt, and type `help` to see the options available.

Some things to notice when stepping through our program:

- Execution starts at 0x00400000 by default, which is the code that sets up the call to `main`.
- After a few steps, `main` gets invoked with a `jal` call.
- When we get to `main`, we find that we are executing a `lui` instruction rather than the expected `lw`. Why?
 - The `lw` instruction requires a base register and offset:
`lw $t0, offset($base)`
 - The assembler allows us to write `lw $t0, num1` (which constitutes a different addressing mode) and inserts appropriate instructions to perform a load from a labelled memory location.
 - It computes the appropriate address, loads it into the reserved assembler register `$at`, then issues a `lw` instruction in the machine's memory addressing mode.

Another simple example:

See: `/cs/terescoj/shared/cs2500/examples/spim-simple/hello.c`

A few things to note:

- We can define a string constant as part of our data segment with an `.asciiz` directive.
- The `la` is a pseudoinstruction that directs the assembler to load the given register with the address of the given label.
- A `syscall` with `$v0` set to 4 is the `print_string` operation, and will print the null-terminated string starting at the address in `$a0`.

Next, we look at a complete version of the factorial program.

See: `/cs/terescoj/shared/cs2500/examples/spim-factorial/factorial.s`

- The `factorial` subroutine is identical to what we looked at last time.
- We fill in the missing `multiply` routine with the one you did for the lecture assignment, changed to use `$a0` and `$a1` as the parameters and to put the answer into `$v0`.
- The `main` subroutine will use two “s” registers and will call subroutines, so we begin by pushing `$s0`, `$s1`, and `$ra` onto the stack. Note that SPIM initialized `$sp` for us appropriately.

- We print a prompt string with the `print_string` syscall and then read in an int from the keyboard with the `read_int` syscall. Note: a complete list of syscall codes is in Figure B.9.1.
 - After the call to `factorial`, we use a series of syscalls to print the answer.
 - Finally, we pop the stack and return from `main`.
-

MIPS Pseudoinstructions

We have mentioned the idea of *pseudoinstructions* a few times. These are “instructions” that exist in MIPS assembly that don’t exist in machine language.

- The pseudoinstructions map to one or more MIPS machine instructions.
- These exist for convenience of the programmer (human or compiler).

Here are a few common MIPS pseudoinstructions:

- `move $a $b` – move (copy) contents of register `$b` to register `$a`.
This is assembled into `add $s0, $s1, $zero`.
- `blt $a, $b, L` – *branch on less than* – if `$a` is less than `$b`, branch to label `L`.
This is assembled into two machine instructions:

```
slt $at, $a, $b
bne $at, L
```

Note the use of the assembler reserved register `$at`.

- `li` – load immediate
- `la` – load address
- `sgt, sle, sge` – set on ...
- `bge, bgt, ble, blt` – conditional branches