Computer Architecture
&
Organization (EEL 4768)

Lab #2

The MIPS Register Files

A register file is a small set of high-speed storage cells inside the CPU. There are special-purpose registers such as the IR and PC, and also general-purpose registers for storing operands of instructions such as add, sub, mul, etc. A CPU register can generally be accessed in a single clock cycle, whereas main memory may require dozens of CPU clock cycles to read or write.

Since there are very few registers compared to memory cells, registers also require far fewer bits to specify which register to use. This in turn allows for smaller instruction codes. For example, the MIPS processor has 32 general-purpose registers, so it takes 5 bits to specify which one to use. In contrast, the MIPS has a 4 GibiByte memory address space, so it takes 32 bits to specify which memory cell to use.

MIPS is a load-store architecture, which means that only load and store instructions can access memory. All other instructions (add, sub, mul, div, and, or, etc.) must get their operands from registers and store their results in a register. Suppose x, y, and sum are variables in a program, and we want to translate the following statement to MAL (MIPS Assembly Language):

```
sum = x + y
```

Since variables represent memory locations, the MIPS processor can only use them in load and store instructions. The values must be first loaded into CPU registers using load instructions. We can then add the values in the CPU registers using an add instruction, which must also put the result in a register. Finally, we must use a store instruction to place the result in the variable sum.

```
# sum = x + y
lw    $t0, x            # Load x from memory into a CPU register
lw    $t1, y            # Load y from memory into a CPU register
add   $t0, $t0, $t1     # Add x and y
sw    $t0, sum          # Store the result from the CPU register to memory
```

The MIPS processor has one standard register file containing thirty-two 32-bit registers for use by integer and logic instructions. These registers are called $0 through $31.

MIPS assembly language employs a *convention* for use of registers. This convention is not enforced by the assembler or the hardware, but it *must* be followed by all MIPS assembly language programmers in order to avoid unexpected behavior of modules written by different people.

Table1. MIPS Registers

Register Number	Conventional Name	Usage
$0	$zero	Hard-wired to 0
$1	$at	Reserved for pseudo-instructions
$2 - $3	$v0, $v1	Return values from functions
$4 - $7	$a0 - $a3	Arguments to functions - not preserved by subprograms

Register Number	Conventional Name	Usage
$8 - $15	$t0 - $t7	Temporary data, not preserved by subprograms
$16 - $23	$s0 - $s7	Saved registers, preserved by subprograms
$24 - $25	$t8 - $t9	More temporary registers, not preserved by subprograms
$26 - $27	$k0 - $k1	Reserved for kernel. Do not use.
$28	$gp	Global Area Pointer (base of global data segment)
$29	$sp	Stack Pointer
$30	$fp	Frame Pointer
$31	$ra	Return Address
$f0 - $f3	-	Floating point return values
$f4 - $f10	-	Temporary registers, not preserved by subprograms
$f12 - $f14	-	First two arguments to subprograms, not preserved by subprograms
$f16 - $f18	-	More temporary registers, not preserved by subprograms
$f20 - $f31	-	Saved registers, preserved by subprograms

The MIPS Instructions

For readability, instructions can be indented. A single tab is typical for opcodes and directives. For labels they can be placed in the leftmost tab field, or if they are lengthy then they can be placed on the line above to avoid excessive indentation of code, and leave room for comments to the right of the instruction.

```
 a_long_label:
 add   $t0, $t5, $s2          # $t0 = $t5 + $s2
```

By convention, the first operand is the destination for most MIPS instructions.

Integer Arithmetic and Logic Instructions

Below are some sample integer arithmetic instructions. Most of these instructions can only use CPU registers for operands (source and destination). A few instructions take one immediate value as a source operand.

```
 add     $t0, $t1, $t2          # $t0 = $t1 + $t2   Exception for signed overflow
 addu    $t0, $t1, $t2          # $t0 = $t1 + $t2   No exception
 sub     $s0, $s0, $t4          # $s0 = $s0 - $t4
 move    $t0, $a0               # $t0 = $a0
 addi    $t0, $t2,4             # $t0 = $t2 + 4
```

Integer Load and Store Instructions

Load and store instructions are the only instructions that can access memory. Their purpose is to move data between a memory location and a CPU register. They are not interchangeable with the move instruction.

Load and store instructions take one register operand and one memory address or immediate operand.

```
lw     $t0, label          # Load 32-bit word at label to $t0
li     $t0, value          # Load 32-bit constant to $t0
la     $t0, label          # Load ADDRESS of label
sw     $t0, label          # The only instruction that has destination last!
```

Jump and Branch Instructions

Jump and branch instructions take an address within a .text segment as the target address. Conditional branch instructions also take two register operands to compare.

```
j      label               # Unconditional jump
beq    $t0, $t4, label     # Branch if $t0 == $t4
blt    $s4, $a0, label     # Branch if $s4 < $a0
bgt    $t0, $t1, t0_bigger # If $t0 > $t1, branch to t0_bigger
b      endif               # branch to endif
```

Exercise 1:

Overview

The primary objective of these exercise is to increase your understanding of the fundamental instructions that are used to find the maximum of 2 integers. You will write a program that explores the issue of implementing conditional execution in MIPS assembler language. The actual program that we will write will read two numbers from the user, and print out the larger of the two. One possible algorithm for this program is exactly the same as the one used by add2.asm, except that we're computing the maximum rather than the sum of two numbers.

Guideline: Find the maximum of 2 integers

1. Input the two integers from the keyboard. We'll need two registers to hold these two numbers. We can use $t0 and $t1 for them.
    a. Obtain the first number from user and place into $t0.
    b. Obtain the second number from user and place into $t1.
2. Find the maximum number between $t0 and $t1
    a. Identify conditions
        i. If $t0 > $t1, branch to t0_bigger,
        ii. otherwise, copy $t1 into $t2.
        iii. then branch to endif
        iv. t0_bigger: copy $t0 into $t2
3. Display the maximum value found
4. Exit

```
# Navid Khoshavi -- 09/07/2018
# larger.asm-- prints the larger of two numbers specified
#  at runtime by the user.
#  Registers used:
#  $t0 - used to hold the first number.
#  $t1 - used to hold the second number.
#  $t2 - used to store the larger of $t1 and $t2.
#  $v0 - syscall parameter and return value.
#  $a0 - syscall parameter.
main:
 ## Get first number from user, put into $t0.
 |--------------------------------------|
 |Put your code here                    |
 |--------------------------------------|
 ## Get second number from user, put into $t1.
 |--------------------------------------|
 |Put your code here                    |
 |--------------------------------------|
 ## put the larger of $t0 and $t1 into $t2.
```

```
|-----------------------------------------------|
|# If $t0 > $t1, branch to t0_bigger,           |
|# otherwise, copy $t1 into $t2.                |
|# and then branch to endif                     |
 t0_bigger:
|# copy $t0 into $t2                            |
 endif:
|-----------------------------------------------|

## Print out $t2.
|---------------------------------------|
|Put your code here                     |
|---------------------------------------|

li $v0, 10                  # syscall code 10 is for exit.
syscall                     # make the syscall.
# end of larger.asm.
```

While-loop

We'll translate a while-loop, then a for-loop. You can do the do-while loop as an exercise (and you should!). Here's a generic while-loop:

```
while ( <cond> )
{
<while-body>
}
```

This is how it's translated to an if-statement with goto's and a label.

```
L1: if ( <cond> )
{
<while-body>
goto L1 ;
}
```

Here's an example with real code:

```
while ( i < k )
{
       k++ ;
       i = i * 2 ;
}
```

Then translate it to if-statements with goto's.

```
L1: if ( i < k )
 {
       k++ ;
       i = i * 2 ;
       goto L1 ;
 }
```

This can be straightforward to convert to MIPS. Below is some pseudocode whereby $r1 stores i, $r2 stores j, and $r3 stores k. Thus, before executing in MARS you will need to substitute some real MIPS registers that you've learned instead of $r1, $r2, and $r3; also recall that EXIT is a label that you will need to provide. So, give it a try!

```
L1:   bge  $r1, $r2, EXIT      # branch if ! ( i < j )
addi $r3, $r3, 1               # k++
add  $r1, $r1, $r1             # i = i * 2
j    L1                        # jump back to top of loop EXIT:
```

We used the pseudo-instruction bge for convenience. Also, rather than use the multiply instruction (which requires using two new registers, HI and LO), we multiplied by 2 by adding the value to itself.

Translating for-loops

To translate a for-loop, we'll only go part way, and translate it to if-statements and goto's. Your challenge is to do the rest on your own. Here's a generic for-loop:

```
for ( <init> ; <cond> ; <update> )
 {  <for-body>  }
```

This is how it's translated to an if-statement with goto's.

```
<init>L1:
if (<inverse cond>) then goto L2:
<for-body>
<update index>
goto L1 ;
L2:
```

Translating nested loops

On your own later, or in the next lab, you can layer the same rules for translating for-loops, and realize that the <for-body> may itself contain a for-loop, as nested loops do not need to be submitted in your portfolio today.

Summary

Translating loops is fairly straightforward, because you can translate them to if-statements with goto's. Congratulations, you've reduced the problem of translating loops to translating if-statements!

Exercise 2:

2-a) Write a program which increments from 0 to 15 and display results in Decimal on the console.

2-b) Modify above program to increment from 0 to 15 and display results in Hexadecimal on the console.