Homework Number 1

This homework is based on the introductory lecture and focuses primarily on instruction set architectures and their use in computer organization.

The intent of these exercises is to get you comfortable with the MIPS assembly language and get you thinking "past the syntax". Solving these problems will necessarily involve reading significant chunks of Chapter 3 from the text.

Textbook Problems

- 3.2 : a problem asking you to read MIPS assembly code and describe what it does
  ANS:
  The code finds the most frequent word existing in an array, and returns
  $v1 = <most frequent word in the array>
  $v0 = <number of times the word is present>

- 3.10: implementation of pseudo instructions using real MIPS instructions
  ANS:

Pseudo Instruction	***Meaning***	***Solution***
move $t5,$t3	*$t5=$t3*	*add $t5,$t3,$zero*
clear $t5	*$t5=0*	*add $t5,$zero,$zero*
li $t5,small	*$t5= small*	*addi $t5,$zero,small*
li $t5,big	*$t5 = big*	*lui $t5, upper_half(big)* *ori $t5, lower_half(big)*
lw $t5, big($t3)	*$t5 = mem[$t3+big]*	*li $at,big* *add $at,$at,$t3* *lw $t5,0($at)*
addi $t5,$t3,big	*$t5=$t3+big*	*li $at, big* *add $t5,$t3,$at*
beq $t5,small,l	*if ($t5==small)goto l*	*li $at, small* *beq $t5,$at,l*
beq $t5,big,l	*if($t5==big)goto l*	*li $at,big* *beq $t5,$at,l*
ble $t5,$t3,l	*if ($t5<=$t3)goto l*	*slt $at,$t3,$t5* *beq $at,$zero,l*
bgt $t5,$t3,l	*if($t5>$t3)goto l*	*slt $at,$t3,$t5* *bne $at,$zero,l*
bge $t5,$t3,l	*if($t5>=$t3)goto l*	*slt $at,$t5,$t3* *beq $at,$zero,l*

- 3.12: how would one do long branches in the MIPS architecture?
  ANS:

  <u>*One possible solution*</u>
  ```
  Here: bne $t1,$t2, skip
            j there
  skip:
  …… <some memory distance away>
  there:
  ```

  <u>*Another solution could use the pseudo-instruction li*</u>
  ```
  Here: bne $t1,$t2, skip
            li $t3, far
            jr $t3
  skip:
  …… <some memory distance away>
  far:
  ```

- 3.29 and 3.30 on page 206 of the text (in the "In more depth" section) are some of the neatest problems in this chapter. They describe (and use) a single instruction machine, viz., a machine with a total of *one* instruction to do various tasks. Besides attempting 3.29 and 3.30, speculate on various other *standard* tasks/instructions that you may be able to do/not_do with such a machine.

Additionally try some of these

1. Switch statements are a C-programming construct that is not available at the MIPS instruction code level.

   ```
   Switch (n) {
   Case '0': f = a; break;
   Case '1': f = b; break;
   Case '2': f = c; break;
   }
   ```

   Assume that $s0-$s2 contains a-c, $s3 contains n.
   Assume the caller wants the answer in $v0.
   a) Convert this code-snippet into MIPS assembly language.
   b) When would you implement this using a Jump Address table ? (Page 129 of text)

2. A finite impulse response (FIR) filter in signal processing, with N taps, is usually represented with the following piece of code:

   ```
   int fir(const int *w,const int *d)
   {
   int sum=0;
   for(i=0;i<N;i++) sum += w[i]*d[i];
   return sum;
   }
   ```

Assume that caller puts w in $a0, d in $a1
Assume N is a constant

a) Write MIPS assembly code that implements the FIR filter.
b) If you could add a few instruction sets from the PowerPC ISA (Page 175 of the text) to the MIPS instruction set, would that change your implementation ? why ? and how ? write pseudo-code to describe the implementation ?
c) Would this change if the filter was using bytes instead of integers ?

3. The recursive procedure for calculating factorials was described in the class notes (see slides). Walk down the code for this function and make sure you undertand the *nested-ness* of the code. Keep special track of the stack pointer and the return address for the various nested routines.
   a) Now modify this routine, to take function arguments from the stack instead of the register $a0.
   b) Further modify the routine, to return the results on the stack, rather than in $v0.
   c) Explain what happens if the “nested routine” uses more (or less) arguments than the “caller” was expecting. Viz., if the caller is putting 2 arguments on the stack and the callee is picking up 3 from the stack, explain how the return address for the routine might get clobbered?