CprE 381 Homework 3

[Note: This homework gives you more practice with the MIPS assembly language. When you are asked to assemble a program, you can try running it on SPIM/MARS to confirm it works. However, make sure you have it running in Bare Machine configuration so that your machine code matches the ISA.]

1. MIPS Machine Code
   a. The following instruction (count or cnt) is not included in the MIPS instruction set:

```
cnt $t0, $t1
# The first operand is rt, the second operand is rd
# if (M[R[rt]] >= 0)
#   R[rt] = R[rt]+1, R[rd] = R[rd] + 1, PC=PC
```

If this instruction were to be implemented in the MIPS instruction set, what is the most appropriate instruction format? Explain why. Provide a sequence of MIPS instructions that performs the same operation. Ungraded, but exam-worthy: Postulate why this instruction wasn't included the MIPS ISA (there is a general philosophical reason and at least one very specific technical reason).
Since **cnt** requires two register operands it could be either an R-type or an I-type instruction (a J-type instruction has no register operands). It might be useful to use an I-type instruction with the immediate as 1 for use when incrementing the register operands, but this would not be particularly elegant.

One possible implementation:

```
        j cond
    loop:
        addi  $t0, $t0, 1
        addi  $t1, $t1, 1
    cond:
        lb    $at, 0($t0)
        slti  $at, $at, 0       # M[R[rt]] < 0
        beq   $at, $zero, loop
                                # !(M[R[rt]] < 0) =
                                (M[R[rt]] >= 0)
                                # PC = PC really means
                                keep executing the
                                functionality of the
                                instruction until the
                                condition doesn't match
                                and then allow the
                                default PC=PC+4 to move
```

execution to the next instruction

One thing to be careful of is that you can't just perform the increment instructions first since they happen conditionally, even when the instruction is first executed. An alternative approach starts with a conditional branch and then unconditionally jumps back at the end, but that requires an additional instruction to be executed for each execution of the **cnt** instruction.

The likely reason that MIPS doesn't provide such an instruction is that it is doing multiple operations at once--a conditional evaluation, a memory operation, two register increments, and a conditional non-PC+4 PC update. Each of these operations would require separate hardware in an implementation or would require a more complex microcoded hardware design. Very specifically, the **cnt** instruction would require two registers in the register file to be written (since you've already done your Lab 2, you should appreciate why this would cause a hardware issue). In either case, this could hurt performance. Another reason could possibly be that it complicates the job of a compiler to choose when to use this complex instruction versus the simpler sequences (this instruction is only useful in very limited cases).

b. Translate the following MIPS assembly into machine code providing the following for each instruction. First, identify the instruction's format. Second, provide the decimal value for each instruction field. Third, provide the hex encoding of the entire instruction. Assume `Begin` is at `0x00400abc`.

```
begin:
      nor       $t0, $s0, $a0
      addiu     $t1, $zero, 127
loop:
      lw        $t2, 64($t0)
      sltu      $t3, $s2, $t1
      beq       $t3, $zero, loop
      j         begin
```

1. nor $t0, $s0, $a0					
R-format					
Opcode (6)	rs (5)	rt (5)	rd (5)	shamt(5)	func (6)
0	$s0=16	$a0=4	$t0=8	0	27
0000 00	1 0000	0 0100	0 1000	0 0000	10 0111
0x02044027					

2. addiu $t1, $zero, 127
I-format

Opcode (6)	rs (5)	rt (5)	Imm (16)
9	$zero=0	$t1=9	127
00 1001	0 0000	0 1001	0000 0000 0111 1111
0x2409007F			

3. lw $t2, 64($t0)			
I-format			
Opcode (6)	rs (5)	rt (5)	Imm (16)
23	$t0=8	$t2=9	127
10 0011	0 1000	0 1001	0000 0000 0100 0000
0x8D0A0040			

4. sltu $t3, $s2, $t1					
I-format					
Opcode (6)	rs (5)	rt (5)	rd (5)	shamt(5)	func (6)
0	$s2=18	$t1=9	$t3=11	0	2B
00 0000	1 0010	0 1001	0 1011	0 0000	10 1011
0x0249582B					

5. beq $t3, $zero, loop			
I-format			
Opcode (6)	rs (5)	rt (5)	Imm (16)
4	$t3=11	$zero=0	-2
00 0100	0 1011	0 0000	1111 1111 1111 1110
0X1160FFFE			

8. j begin	
J-format	
Opcode (6)	Target address (26)
2	Bits 27:2 of 0x00400abc
0000 10	00 0001 0000 0000 0010 1010 1111
0x081002AF	

c. We've discussed in class that you cannot load an arbitrary 32 bit integer (e.g., 0xFEED2050) using a single instruction. Look up the **lui** instruction n (e.g., on the green sheet from your textbook) and provide a two-instruction sequence that loads 0xFEED2050 into $t0. Then, assuming that **lui** is not supported by the ISA, provide a valid three-instruction sequence that loads 0xFEED2050 into

$t0. Translate these into MIPS machine code providing the same steps as part 1.b.

```
lui $t0, 0xFEED
ori $t0, $t0, 0x2050
```

lui $t0, 0xFEED			
I-format			
Opcode (6)	rs (5)	rt (5)	Imm (16)
15	Unused=0	$t0=8	65261
0011 11	00 000	0 1000	1111 1110 1110 1101
0x3C08FEED			

ori $t0, $t0, 0x2050			
I-format			
Opcode (6)	rs (5)	rt (5)	Imm (16)
13	$t0=8	$t0=8	8272
0011 01	01 000	0 1000	0010 0000 0101 0000
0x35082050			

```
ori $t0, $zero, 0xFEED
sll $t0, $t0, 16
ori $t0, $t0, 0x2050
```

ori $t0, $zero, 0xFEED			
I-format			
Opcode (6)	rs (5)	rt (5)	Imm (16)
13	$zero=0	$t0=8	65261
0011 01	00 000	0 1000	1111 1110 1110 1101
0x3408FEED			

sll $t0, $t0, 16					
R-format					
Opcode (6)	rs (5)	rt (5)	rd (5)	shamt(5)	func (6)
0	Unused=0	$t0=8	$t0=8	16	0
0000 00	00 000	0 1000	0100 0	100 00	00 0000
0x00084400					

ori $t0, $t0, 0x3210			
I-format			
Opcode (6)	rs (5)	rt (5)	Imm (16)
13	$t0=8	$t0=8	8272
0011 01	01 000	0 1000	0010 0000 0101 0000
0x35082050			

2. MIPS Programming with procedures *[I suggest you actually simulate this program using the provided version of MARS to confirm that they work. Do not simply copy these from online examples or from the result of a compiler.* ***YOU MUST COMMENT WHAT YOU ARE DOING****.]*
    a. Write a MIPS program that iteratively (i.e., using a for loop) calculates the Fibonacci number, F_N, for an inputted number, N. Have N be an integer entered by a user and print F_N to the console. *[See MARS lecture companion files for an example of how to read an integer in MARS and print an output.]*
    See prob2a.s for one solution.

    b. Write a second MIPS program that recursively calculates F_N (i.e., using a procedure that calls itself with an updated argument). Make sure to follow the convention presented in lecture. Specifically, use the appropriate saved vs temporary registers, argument passing registers, return value registers, and a basic stack frame with appropriate alignment.
    See prob2b.s for one solution (there are many ways to do this).

    c. How many instructions (i.e., dynamic instructions) were executed in your two different programs? Briefly show your calculations. *[MARS has a tool that can count instructions, which I suggest you use to verify that your hand calculations are reasonably close.]*

    For the iterative version, there six instructions in the case of N=1. Every additional iteration requires an additional six instructions. So the total instruction count is 6*N for N>1.

    For the recursive version, there are 3 instructions to setup and call the initial function, 4 instructions executed in the base case, and 12 instructions to execute each non-base call. So # insts = 7+12*(N-2). For example, N=10 executes the original call, 1 base case call (N=2), and 8 non-base case calls.