CprE 381 Homework 2

[Note: This homework gives you practice with MIPS assembly language. When you are asked to provide a program, try running it on MARS to confirm it works. MARS can also check your understanding of code tracing, etc. Plus you get more practice with the tools.]
[I've provided assembly programs that correspond to the solutions presented here. Feel free to play around with these programs to test out alternative solutions, etc.]

1. MIPS Control Flow
   a. Assume `$t0` holds the value `0x0000FFFF`, `$t1` holds the value `0x80000000`, and `$t2` holds the value `0x00000001`. What is the value of `$t2` after the following instructions?

```
        lui   $t0, 0xFFFF
        slt   $t1, $t1, $t0
        bne   $t1, $0, SOMEPLACE
        addiu $t2, $t1, -1
        j EXIT
SOMEPLACE:
        add $t2, $t2, $t1
EXIT:
```

$t0 becomes 0xFFFF_0000 after lui. We check if $t1 < $t0, which is true (t0 is less negative), and set $t1 to 1. 1 != 0 so bne takes the branch to SOMEPLACE. In someplace we add 1 ($t1) to 1 ($t2) resulting in a final $t2 value of 0x0000_0002 or 2

   b. Translate the following C-style loop into MIPS assembly, assuming the following variables to register mappings:

int *a	$s0
int i	$s2
int N	$s3
int *b	$s4

All variables are 4 byte words. How many instructions are executed if N=1, N=10, N=100, N=1000?

```
for (i=1; i<N; i++) {
   a[i] = (b[i] / 8) + 11;   // Do NOT use div
                             // instruction
}
```

Several acceptable versions exist. Below is one and I've included a sample assembly file that can be run on MARS. Ensure students use sra instead of srl

```
lw  $s1, 0($s0)      # a[0]
ori $s2, $zero, 1    # i=1
```

```
j cond                  # get to i<N evaluation
loop:
sll $t3, $s2, 2         # i*4 to load words
addu $t4, $t3, $s0      # &a[i]
addu $t5, $t3, $s4      # &b[i]
lw $t5, 0($t5)          # b[i]
sra $t5, $t5, 3         # b[i] / 16
addiu $t5, $t5, 11      # b[i] + 11
sw $t5, 0($t4)          # store b[i]into a[i]
addiu $s2, $s2, 1       # i++
cond:
slt $t6, $s2, $s3       # $t0 is 1 if i < N
bne $t6, $zero, loop # execute loop body when i < N
```

N	# Instructions 5 insts/init + (N iterations)*10insts/iteration
0	5
1	5 + 1 * 10 = 16
10	5 + 10 * 10 = 106
100	5 + 100 * 10 = 1006
1000	5 + 1000 * 10 = 10006

c. Write MIPS assembly for the following switch statement. Assume `score` is in `$a0` and `grade` is in `$v0`. Do NOT use a jump table as a compiler might, but rather use conditional branches.

```
if (score >= 90) {
    grade = 'A';
} else if (score >= 80) {
    grade = 'B';
} else if (score >= 70) {
    grade = 'C';
} else if (score >= 60) {
    grade = 'D';
} else {
    grade = 'F';
}
```

The general process for each conditional:
score >= X
!(score < X)
slti $t0, $a0, X # $t0 == 0 when score >= X
beq $t0, $0, caseX

This is one possible implementation:

```
slti $t0, $a0, 90
```

```
beq $t0, $0, casea
slti $t0, $a0, 80
beq $t0, $0, caseb
slti $t0, $a0, 70
beq $t0, $0, casec
slti $t0, $a0, 60
beq $t0, $0, cased
j casef
casea:
addi $a1, $0, 65          # grade = 'A'
j exit
caseb:
addi $a1, $0, 66          # grade = 'B'
j exit
casec:
addi $a1, $0, 67          # grade = 'C'
j exit
cased:
addi $a1, $0, 68          # grade = 'D'
j exit
casef:
addi  $a1, $0, 70          # grade = 'F'
exit:
```

2. MIPS Assembly Language Design
   a. P&H(2.21) <§2.6>. Provide a minimal set of MIPS instructions that may be used to implement the following pseudoinstruction *[We've talked about pseudoinstructions before with **mov**. Effectively they are assembly instructions that aren't actually in the ISA of hardware, so the assembler has to translate them into another machine instruction or series of machine instructions.]*:

      ```
      not $t1, $t2     # bit-wise invert
      ```

      The following is one solution; there may be other valid solutions:

      ```
      nor $t1, $t2, $t2
      ```

   b. You are tasked with adding a new pseudoinstruction that takes the two's complement of a register's value:

      ```
      ror $t0, $t1, imm   // rotates $t1 imm bits right
      ```

      Give a minimal implementation. Should this instruction produce any exceptions (i.e., can it produce any errors)? If so, what are they and does your implementation handle them?

      To rotate bits to the other side, we can combine the results of shifting right by imm bits and shifting left by N-imm bits

      ```
      srl $t2, $t1, imm #shift imm bits right
      sll $t3, $t1, 32-imm #shift 32-imm bits left
      or $t0, $t2, $t3 #combine
      ```

These instructions do not produce errors, so there should be no need for handling that.

c. Why does MIPS not have **add** `label_dst,label_src1, label_src2,` instructions in its ISA (there are at least two reasons for this)? *[Read this instruction's function as M[label_dst] = M[label_src1] + M[label_src2].]* Provide a concrete technical justification—you should have ideas both from lab and lecture.
Philosophically, MIPS avoids performing multiple basic functions within a single instruction in order to simplify hardware. Specifically, in this instruction a datapath would have to load two values from two arbitrary memory locations, perform an addition, and then store the result to an arbitrary memory location. First, you have to determine how to generate three 32-bit arbitrary memory addresses from a single 32-bit instruction (remember, MIPS has a fixed instruction width) which would be impossible without additional instructions and limitations on the relative offsets the three operands are from a base address. Second, you would either require a memory module with two read ports or would have to sequentially use the memory module to perform the two source loads. The result of the loads would then also need to be fed back into the adder (i.e., ALU), increasing the number of paths needed in your datapath. These last issues become even more significant once you consider pipelining – a technique at the heart of the original MIPS philosophy.

3. MIPS Programming *[I suggest you actually run these programs to confirm that they work. Use MARS for MIPS runtime simulation.]*
   a. Write a simple (you do not need to optimize—just use the direct implementation) C code snippet that implements the strcat function from string.h (https://www.cplusplus.com/reference/cstring/strcat/):

```
char * strncpy (char * dst, const char * src, size_t num);
```

There are many flavors of possible solution. This is one:

```
int i = 0;
while (i < num && src[i] != '\0') {
  dst[i] = src[i];
  i++;
}
while (i < num) {
  dst[i] = '\0';
  i++;
}
return dst;
```

Translate your answer to part a into MIPS assembly. Assume that `str` is in `$a0` and `character` is in `$a1`. All other variables in your C code should be initialized within your code.

I've included one reasonable implementation in prob3b.s.

b. Add code that *calls* strncpy and prints the result for testing purposes. Provide *three* reasonable test cases for your MIPS assembly (inputs and expected outputs) and justify why you have included each one.
First, there should be a generic test case that copies the exact length of the string + null character. Then you could test a number shorter than the length to make sure that the copied string does not contain the null character. Finally, there should be a test to copy several characters greater than the length to test the null character padding.

src	num	Expected dst result
"Cpre381 is lots of fun!"	25	"CprE381 is lots of fun!"
"CprE381 is lots of fun!"	10	"CprE381 is"
"CprE381 is lots of fun!"	30	"CprE381 is lots of fun!"