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 `0xEFF01270, and $t2` holds the value `0x00000001`. What is the value of `$t2` after the following instructions?

```
          lui    $t0, 0xFFFF
          slt    $t1, $t1, $t0
          beq    $t1, $0, SOMEPLACE
          sra    $t2, $t1, 31
          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 beq does not take the branch to SOMEPLACE. We then execute the sra command and shift the value 0x0000_0001 to the right 31 which is equal to 0x0000_0000 and stores it into $t2. Finally we jump to exit and terminate the program. The last assignment to $t2 was 0x0000_0000 or 0

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

int *a	$s0
int i	$s1
int N	$s2
int *b	$s3

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

```
 Int max = a[0]
for (i=1; i<N; i++) {
   max = a[i] > max ? a[i] : max;
}
```

   Several acceptable versions exist. Below is one and I've included a sample assembly file that can be run on MARS.

```
# This code assumes a value N = 10

.data
    a: .word 0, 1, 2, 3, 4, 100, 6, 7, 8, 9

.text

la $s0, a     # $s0 = a*
addi $s1, $0, 0  # $s1 = max, this is an output
initalize to zero
addi $s2, $0, 0  # $s2 = i, temporary variable
initalize to zero
addi $s3, $0, 10 # $s3 = N, this should match the
length of "a" in data section

lw $s1, 0($s0) # int max = a[0]

addi $s2, $0, 1 # i = 1; portion of for loop

j lp_check # Begin for loop, always check condition
first before executing body

lp:
addi $s2, $s2, 1 # i++, we can skip the first value
since we initalize "max" to the first value of the
array
sll $t0, $s2, 2 # convert array offset to byte offset
add $t0, $s0, $t0 # compute address of a[i]

lw $t1, 0($t0) # get value of array at a[i]

sgt $t2, $t1, $s1    # a[i] > max
beq $t2, $0, lp_check # : max;
add $s1, $0, $t1     # ? a[i]

lp_check:
blt $s2, $s3, lp # for (i=1; i < N; i++)

# end of program
```

*In our case there is a specific number of swaps, but using the general equation of ln(n) + 0.577. In this case I am rounding to the nearest whole number.

N	# Instructions 10 insts/init + (N iterations)*8 insts/iteration + 1 extra instruction per replacement (Number of replacements)

0	10
1	10 + 1 * 8 + (0+ 0.577) = 19
10	10 + 10 * 8 + (2.3 + 0.577)= 93
100	10 + 100 * 8 + (4.605 + 0.577) = 815
1000	10 + 1000 * 8 + (6.907 + 0.577) = 8017

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 rotates a registers value left or right depending on a signed immediate value:

    ```
    rolr $t0, $t1, sign_ext_imm
    # rotates $t1 left by sign_ext_imm bits
    # (a negative immediate implie right shift)
    ```

    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 perform this type of rotation we first load our immediate into a temporary register so that we can use it in our sllv instruction.

    We first perform a shift left to compute the upper bits for the result of the rotation. We then compute the number of bits that we need to shift right (32 – the number we shifted left) in order to get the lower bits in the rotation.

    Once we have both values, we can or the result of the upper and lower values to get the final result of the rotation

    ```
    addiu $at, $zero, sign_ext_imm
    sllv $t0, $t1, $at
    subu $at, $zero, $at
    addiu $at, $at, 32
    srlv $at, $t1, $at
    or $t0, $t0, $at # combine
    ```

    There is a limitation in that you cannot shift by more than 32 bits, so this can handle rotations up to 31 bits.

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 * strrchr (char * dst, int character);
```

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

```
int i = 0;
char* last_occurence;
while (dst[i] != '\0') {
  if( dst[i] = character ){
      last_occurence = *dst[i];
  }
  i++;
}
return last_occurence;
```

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

.data

```
character: .byte ' '
dst:        .asciiz "CprE381 is lots of fun!"

.text

la $a0, dst
la $t0, character
lb $a1, 0($t0)

addi $t0, $0, 0 # int i = 0;
lb $s0, 0($a0) # dst[0]
addi $v0, $0, -1 # default return to -1 for error (character doesn't exist in string)

#base case
bne $s0, $a1, lp_cond
add $v0, $a0, $0 # if first instance matches then use this then continue checking
j lp_cond

lp: # while( dst[i] != 0 )
addi $t0, $t0, 1 # i++
add $t2, $t0, $a0 # address of dst[i]
lb $s0, 0($t2) # dst[i]
bne $s0, $a1, lp_cond # if( dst[i] = character ){
add $v0, $0, $t2 # last_occurence = *dst[i];

lp_cond:
bne $s0, $0, lp
```

b. Add code that *calls* strrchr 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, this can be accomplished by setting character = dst[0]. Then you could test a number shorter than the length to make sure the resultant string removes all the characters before the last occurrence of character. Finally, there should be a test to point to the last character.

dst	character	Expected dst result
“Cpre381 is lots of fun!”	‘o’	“of fun!“

“CprE381 is lots of fun!“	‘C’	“CprE381 is lots of fun!”
“CprE381 is lots of fun!“	‘ !‘	“ !“