Computer Science Fundamentals II
MIDTERM #2

Spring 2020

Submit by: noon PST, Friday, April 24th, 2020.

This four-page exam has six problems on its last two pages. Feel free to read these first two instruction pages carefully before you actually start this exam. Give yourself about two hours total to complete the work: about 15-20 minutes to get settled and to read these instructions, 90 minutes to devise your answers to the problems posed, and then 10-15 minutes to package those answers into your repo submission.

Here is a summary of the topics examined by the problems:

- boolean expressions using AND, OR, NOT.
- truth tables
- the *sum of products* form for boolean expressions
- unsigned binary coding of non-negative integers
- two's complement coding of integers
- basic MIPS calculations in registers
- loops and conditional execution in MIPS
- byte load and store in MIPS
- calling a function in MIPS (but not designing a MIPS function)

The second page of this exam gives a summary of the MIPS32 instructions that you might need to complete this exam. If you are looking for some other one while solving a problem, you are probably doing something wrong. I also give several exam-specific MIPS code lines `INPUT`, `OUTPUT`, and `PRINT` that you can use in your MIPS programs in place of system calls.

You can use your notes and other written sources for reference, though the exam's limited time makes it difficult to do this extensively, and you really shouldn't need to. Standard academic honesty practices apply. You shouldn't consult coding solutions on the web or look up answers. If you *do* consult an on-line reference, say, one that you feel is reasonable given these guidelines, it's always good practice to tell me what you consulted, regardless.

You might be inclined to compile, run, and test your code that you are working on. I'm encouraging you not to, if only because of the limited time and because of the nature of the exam. This means that I'll be more forgiving of syntax errors, and most errors, in general. Your code should be meant to run and work, but I'm also not assuming that you're taking the time to compile, run, and test it.

The same principles apply in using LogiSim to build circuits. My exam only asks for boolean expressions and truth tables, so you aren't actually asked to give circuit diagrams. You can draw and submit circuit designs if you like, but I'm asking you instead to do the written legwork necessary to devise a circuit's design, and that design work is more important in my evaluation than an actual LogiSim circuit that you might draw and test.

Once you've put all the images and text for the exam into your repo, submit it. Recall that you have to type

```
git add the_image_or_text_file
```

for each file that you submit. And then you commit and push with the commands:

```
git commit -m "completed midterm"
git push origin master
```

And then it will all get submitted. If you run into any problems, email me.

MIPS32 coding summary and guidelines:

`li` $RD, *value*	–loads an immediate value into a register.
`la` $RD, *label*	–loads the address of a label into a register.
`lb` $RD, ($RS)	–loads a byte from memory at an address specified in a register.
`lb` $RD, *offset* ($RS)	–same, but at a constant offset from the given address.
`sb` $RS, ($RD)	–stores a byte of a register into memory at an address specified in a register.
`sb` $RS, *offset* ($RD)	–same, but at a constant offset from the given address.
`addu` $RD, $RS1, $RS2,	–add two registers, storing the sum in another.
`subu` $RD, $RS1, $RS2,	–subtract two registers, storing the difference in a third.
`addiu` $RD, $RS, *value*	–add a value to a register, storing the sum in another.
`move` $RD, $RS	–copy a register's value to another.
`jal` *label*	–jump to a labelled line storing a return address in `$ra`.
`jr` $RA	–jump to an address stored in a register.
`b` *label*	–jump to a labelled line.
`blt` $RS1, $RS2, *label*	–jump to a labelled line if one register's value is less than another.
`bltz` $RS, *label*	–jump to a labelled line if a register's value is less than zero.
`gt`, `le`, `ge`, `eq`, `ne`	–other conditions than `lt`

The registers you can access are named `$v0-v1`, `$a0-a3`, `$t0-t9`, `$s0-s7`, `$sp`, `$fp`, and `$ra`.

You can use the line

register = `INPUT`

in your MIPS code to stand in for a system call #5 like the code below for register `$t0`:

```
li $v0, 5       # $t0 = INPUT
syscall         #
move $t0, $v0   #
```

You can use the line

`OUTPUT` *register*

in your MIPS code to stand in for these system calls #1 and #4 like the code below for register `$s0`:

```
move $a0, $s0   # OUTPUT $s0
li $v0, 1       #
syscall         #
li $v0, 4       #
la $a0, eoln    #
syscall         #
```

You can assume that there is a labelled data declaration `eoln` for the null-terminated string containing the end of line character.

You can use the line

`PRINT` *label*

in your MIPS code to stand in for code like this system call #4 using a string labelled with `eoln`:

```
li $v0, 4       # PRINT eoln
la $a0, eoln    #
syscall         #
```

1. Consider a 3-bit sequence given as $s_2 : s_1 : s_0$. A palindrome is a sequence that reads the same forwards and backwards. For example the bit sequence `1 : 0 : 1` is a palindrome. **Give a truth table** for a boolean function that indicates whether $s_2 : s_1 : s_0$ is a palindrome. Now **give a boolean expression** in terms of s_2, s_1, and s_0, one that uses only the AND, OR, and NOT connectives, for that 3-bit palindrome boolean function.

2. Below is the start of some MIPS32 source code for a program that defines two labels that place null-terminated ASCII strings in memory. One is just a period, and the other is the end-of-line character. It then reads two integers from the console into registers `$t0` and `$t1` using system call #5.

```
    .data
dot:  .asciiz "."
eoln: .asciiz "\n"
    .text
    .global main
main:
    li $v0, 5       # $t0 = INPUT
    syscall         #
    move $t0, $v0   #
    li $v0, 5       # $t1 = INPUT
    syscall         #
    move $t1, $v0   #
```

   Complete the program so that it outputs a rectangle of periods, one that is `$t0` rows tall and `$t1` columns wide, to the console. For example, if the user enters 3 then 5, the code would output these three lines:

```
.....
.....
.....
```

   To make your code writing read more easily, use the words `PRINT dot` in place of the instructions that output the string referenced by label `dot`, and use the word `PRINT eoln` in place of the instructions that output the end-of-line string.

   Don't forget that `main` needs to return 0.

3. Consider two 2-bit sequences $x_1 : x_0$ and $y_1 : y_0$ that each encode a 2-bit unsigned integer value. Let's name the integers they code x and y. **Give a boolean expression** as *a sum of products* for the condition when the sum $x+y$ is an integer multiple of 4. The boolean variables in that expression are the bits x_1, x_0, y_1, y_0, and the expression should use only the AND, OR, and NOT connectives. Furthermore, it should be an OR (a sum) of expressions that are an AND (a product) of these variables or their negation with NOT.

   It would be helpful to me if you **tell me for which values** of those bit sequences you think the expression should be 1, but you don't have to give the full truth table.

4. Let $r_{k-1} : \cdots : r_2 : r_1 : r_0$ be the bits of a k-bit register that gives the two's complement encoding of some integer. Assume that k is an even number.

   (a) Suppose I tell you that the first $k/2$ bits are `0`. What does that tell you about the integer value held by that register? **What's the range** of possible values it could hold? If it helps, tell me your answer assuming k is 8. This would mean that the leftmost 4 bits are `0000`.

   (b) Suppose I tell you that the first $k/2$ bits are `1`. What does that tell you about the integer value held by that register? **What's the range** of possible values it could hold? If it helps, tell me your answer assuming k is 8. This would mean that the leftmost 4 bits are `1111`.

   (c) Suppose $k = 8$ and suppose that the register holds a value in the range -4 to 3, inclusive. (That is, it's not smaller than -4 and it's not larger than 3.) **Give a boolean expression** on the bits r_i for the condition that the register's value is in that range. That expression should use only AND, OR, and NOT.

5. **Write a snippet** of MIPS32 code that assumes that register `$a0` holds an address of the first character of a null-terminated string in memory. You can assume that the string is an upper-case letter, followed by two periods, followed by another upper-case letter, followed by two periods, and so on. The last two characters before the null-terminating byte are periods. For example, that memory might contain this character sequence:

   ```
   M..I..P..S..
   ```

   which means that it is 12 characters long just before the null byte at the end.

   The snippet of code should complete having converted the contents of memory so that it instead contains the same uppercase letters, but with the period after each replaced by the character's lowercase letter. So, if the code were run on the above string in memory it would end with the string:

   ```
   Mm.Ii.Pp.Ss.
   ```

   This is the same 12 bytes in memory, just before the null character, but edited as shown.

   Recall that 97 is the ASCII code for `a` and 65 is the ASCII code for `A`.

6. **Write a snippet** of MIPS code that, when it starts running, contains a positive value in register `$s1`. Have your code output all the positive divisors of that value. Your code can use the instruction line `OUTPUT $s0` in place of the code that outputs the value `$s0` on a single line using system calls #1 and #4.

   Your code *must* **rely on a MIPS function** labelled with `divides`. To use this function, you must put two values into registers `$a0` and `$a1` and then call the function `jal divides`. When `divides` returns, the register `$v0` will contain a 1 if `$a0` is an integer multiple of `$a1` (i.e. `$a1` evenly divides `$a0`). It will contain a 0 if `$a1` is not a divisor of `$a0`.

   Remember that, by MIPS calling conventions, the values of the registers `$s0` and `$s1` will be preserved by that call to `divides`.

   When `$s0` holds 6 at the start of your code, your code should output

   ```
   1
   2
   3
   6
   ```