Computer Science Fundamentals II
Final Exam

Spring 2020

Submit by: noon PST, Thursday, May 14th, 2020.

This 5-page exam has eight problems on its last three pages. Feel free to read these first two instruction pages carefully before you actually start this exam. Give yourself about four hours total to complete the work: about 30 minutes to get settled and to read these instructions, 3 hours to devise and write up your answers to the problems posed, and then 30 minutes to package those answers into your repo submission.

Here is a summary of the topics and techniques stressed by the problems:

- boolean expressions using AND, OR, NOT.
- two's complement encodings of integers
- basic MIPS calculations in registers
- byte load and store in MIPS
- linked list manipulation
- basic recursive algorithms
- C++ class and method definitions
- C++ inheritance
- the C++ STL's `vector` class

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 final"
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.
`shl` $RD, $RS, *positions*	–shift a register's bits left, storing the result in another.
`move` $RD, $RS	–copy a register's value to another.
`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. Suppose you have a register that holds two bits $r_1 : r_0$, and those bits are interpreted as a two's complement encoding of an integer.

   (a) What is the range of integer values that this register represents? **Give the integer interpretation of each** of the possible two bit patterns that could be held by the register.

   (b) Suppose we have a second such register with bits $s_1 : s_0$. **Give a boolean expression** for a condition *invalid* indicating that their product cannot be encoded as two's complement in a third such 2-bit register. That expression should only use the AND, OR, and NOT connectives. You need not give a truth table for your expression's design, but you should **describe the cases when the product is invalid**.

   Hint: Let me help you think about this. Suppose we instead had 4-bit registers. Two could each encode -6 and -2 and their product would be 12. A 4-bit register cannot encode this product in two's complement because 7 is the highest value a 4-bit register can encode.

2. The sequence of integer squares is 0, 1, 4, 9, 16, 25, 36, and so on. If you look at consecutive squares, they differ by 1, 3, 5, 7, 9, 11, and so on. The squares' differences are just the positive odd numbers, in increasing order. To enumerate the squares, we could use this algorithm: start by outputting 0, then add the first odd number to it and output 1. Then add the next odd number to it and output 4. Then add the next odd number to it and output 9. And so on.

   Write a MIPS program that gets an integer input n, and outputs the first n squares, one on each line, using the algorithm above. If $n = 4$, it should output 0 through 9.

   Label your code as `main` and have the last two lines read:

   ```
   li $v0,0
   jr $ra
   ```

3. Assume we have the start of a MIPS program with a label `text` of a null-terminated string of characters in memory. **Write a snippet of MIPS code** that modifies the contents of this string so that it is reversed. If, for example, the string is made up of 6 bytes whose non-null characters read "hello", then after your snippet executes they should instead read "olleh".

4. In this problem we devise a class `Meal` whose object instances have a few properties, and a client class `Eater` that eats meal object instances.

   - Every `Meal` object is either healthy or unhealthy to eat, and has a rating in how happy it makes a person when they eat it. This rating is a floating point value. In addition, when a person eats this meal, they say something aloud when they eat it. These three pieces of information are stored with every meal object instance and are given to its constructor. Meal objects have three methods `isHealthy`, `getHappiness`, and `whatToSay` that are "getters" for these three pieces of information. There are no other methods, just the constructor and these three getters.
   - An `Eater` object is either happy or unhappy, and they have a level of health. They are initially happy and they have a health of 10.0. In addition to its constructor, which takes no information as arguments, it has one method `eat` that takes a `Meal` object as its argument. When they eat a meal, they output the appropriate message about that meal. When they eat a healthy meal, their health increases by the happiness rating of the meal. When they eat an unhealthy meal, their health decreases by the happiness rating of the meal. They are happy if they've just eaten an unhealthy meal with a happiness rating above 5.0, regardless of their health. They are also happy if their health is positive.

   Write the class definitions and the code for **the methods** of these two classes.

5. Using the `Meal` class from the prior problem, let's build several classes that derive from it.

   - A `Donut` is an unhealthy meal. Every donut meal has a happiness rating of 5.0. When eaten, a person says "Mmmm... donuts."
   - A `Pasta` is a healthy meal that has a happiness rating that depends on the instance. Pastas come in different shapes and sizes, and are identified by a name like "rigatoni", or "buccatini", or "macaroni". When a person eats a `Pasta` meal instance, they say "I love rigatoni!" or "I love macaroni!", or whatever the name of the pasta is.
   - A `Pea` is a healthy meal. Every instance has a happiness rating of 0.05. When a pea meal is eaten, a person says "*shlirp*" the first time, "*shlippp*" the second time, and continues to alternate between these two when it keeps eating that same `Pea` meal instance. Its instance variable for what is said can just be the empty string, and its `whatToSay` can alternate between these two based on some additional state information held by pea meal instances.

   Define these subclasses of `Meal` with appropriate **constructors** for each of them, overriding methods and adding instance variables if necessary. You can assume that we've tagged methods in `Meal` as `virtual`, and labelled instance variables `protected`, even if you didn't do so when you wrote your answer for the problem just above.

6. Consider the following C++ definition of a struct that serves as a linked list node for a sequence of digits:

```
struct DigitNode {
  int digit;
  struct DigitNode* next;
};
```

   Write each of the four methods of the following class:

```
class DigitSequence {
private:
  DigitNode* digits;
public:
  DigitSequence(void);
  ~DigitSequence(void);
  void increment(void);
  void decrement(void);
};
```

   It represents a sequence of digits of a non-negative integer as a linked list. The first node referenced by the pointer `digits` contains the least significant digit. The last node contains the most significant digit. The number 537 would be represented as a linked list with three nodes, the first containing 7, the second containing 3, and the last containing 5. The single-digit integers 0 through 9 have just one linked list node with that digit.

   The first method `DigitSequence` is the constructor. It should build a linked list containing the single digit 0.

   The second method `increment` is a method that adds one to the number coded by the digit sequence. If that number happens to contain a sequence of 9 digits, for example 9999, then `increment` will have to change all the 9s to 0s and add a new node containing the digit 1 to the end of the linked list.

   The third method `decrement` subtracts one to the number coded by the digit sequence, if the number is greater than 0. If it is 0, nothing is changed. If the digit sequence encodes a positive power of 10, for example 10000, then `decrement` will have to change all the 0s to 9s and remove the node at the end containing the digit 1. It should `delete` that node.

   The fourth method `DigitSequence` is the destructor. It should give all the nodes that it has allocated back to the heap using `delete`.

7. **Write the code for an additional method** `DigitSequence :: output`, that outputs the digits of the number in most- to least-significant order. **You only get full credit if** this method doesn't modify the contents of the list and doesn't create any extra data structures and heap objects. *Hint:* To meet this full spec, you'll probably need to write a recursive helper function. You'll get reasonable partial credit for code that works but doesn't meet the "no modification, no extra data structures" specification, but you will not get full credit.

8. Here, we build three algorithms that operate on sequences of integers represented as a `std :: vector`.

   (a) **Write a function** `splitInto` that takes three arguments, each being a vector of integers. The first is named `array` and is passed by value. The second named `evens` is passed by reference. The third is named `odds` and is also passed by reference. You can assume that `array` has a size of at least two, and that `evens` and `odds` each have size 0. Upon completion of `splitInto`, `evens` should contain the sequence of items of `array` at even positions (item 0, item 2, item 4, etc.) and `odds` should contain the sequence of items of `array` at odd positions (item 1, item 3, item 5, etc.).

   (b) **Write a function** `merge` that takes two arguments, each being a vector of integers, named `array1` and `array2`, and passed by value. You can assume that `array1` and `array2` each have a size of at least one and are sorted. The function should return a new vector containing the items of both, and in sorted order. For example, if the first contains the sequence 0,3,4,5,18 and the second contains the sequence 2,4,6,7,11 then it should give back a new vector with the sequence 0,2,3,4,4,5,6,7,11,18.

   (c) Now **write a recursive function** `mergeSort` that uses these two functions. It takes an integer vector named `array` and is passed by value. It should return back an integer vector with the same size and storing the same integer values, but in sorted order. It should do this by following this algorithm:

   > *If the vector it is given has size less than two, then that array is already sorted, so it can just return that array back. If the size is two or greater, then it splits the vector into two vectors, the even indexed items and the odd indexed items, with* `splitInto`. *It then calls* `mergeSort` *on each of those vectors individually, thus making two different recursive calls. It gets back two sorted vectors, one with each call. It then merges these two vectors into a single vector using* `merge` *and returns that sorted result.*

   When given a vector containing 18,7,4,6,5,4,3,2,0,11, it should split them into 18,4,5,3,0 and 7,6,4,2,11, sort each of them making two new vectors with 0,3,4,5,18 and 2,4,6,7,11, then merge them into a new vector with 0,2,3,4,4,5,6,7,11,18.