Computer Science Fundamentals II

Practice Final Exam

Spring 2020

You would normally have 3 hours to complete an exam like this, and my expectation is that it should take you only 2 hours to complete most of it. There are eight problems.

Here is a summary of the MIPS32 assembly instructions that you can use in your programs:

`and` $RD, $RS1, $RS2,	–compute the bitwise AND of two registers, storing the result in a third.
`andi` $RD, $RS, *value*	–compute the bitwise AND of a register and an immediate value.
`or` $RD, $RS1, $RS2,	–compute the bitwise OR of two registers, storing the result in a third.
`ori` $RD, $RS, *value*	–compute the bitwise OR of a register and an immediate value.
`shl` $RD, $RS, 1	–shift one bit to the left, filling the least significant bit with 0.
`shr` $RD, $RS, 1	–shift one bit to the right, filling the most significant bit with 0.
`li` $RD, *value*	–loads an immediate value into a register.
`la` $RD, *label*	–loads the address of a label into a register.
`lw` $RD, *label*	–loads a word at an labelled address into a register.
`lw` $RD, ($RS)	–loads a register from memory at an address specified in a register.
`lw` $RD, *offset*($RS)	–loads from an address specified in a register, but at an offset.
`sw` $RS, ($RD)	–stores a register into memory at an address specified in a register.
`sw` $RS, *offset*($RD)	–stores to an address specified in a register, but at an offset.
`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.
`negu` $RD, $RS	–copy the negated value of 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`

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

1. Using only the MIPS32 instructions on the prior page, **write the code for a MIPS32 function** named with the label `bitPalindrome` that, when given a 32-bit integer parameter, returns whether or not that integer's bits form a palindrome. According to the standard calling conventions of MIPS32 assembly, the integer to be checked will be passed as register `a0` and the returned value resulting from the check should be stored in register `v0`.

   For example, the 32 bit sequence `00001110100100111100100101110000` encodes the value 244566384. Because this bit sequence is the same as its reverse, then the code set register `v0` to 1 if `a0` is set to 244566384. If instead `a0` is set to 3 then the code should set `v0` to 0 since the bit sequence `00000000000000000000000000000011` is not a palindrome.

   Hint: consider using the SLL, SLR, ANDI/AND, and ORI/OR instructions to manipulate the bits of registers.

2. Here, we build a digital circuit for the "carry out" of a 2-bit addition. That is, suppose we are adding two numbers encoded as the bit pairs $x_1 : x_0$ and the bits $y_1 : y_0$. Consider the logic for c, the carry bit that results from that addition. For example, if we are adding 3 to 2, that's the addition of `11` to `10`. The result is 5, i.e. `101`. Because the result's leftmost bit is 1, then the carry c is 1. If instead we are adding 1 to 2 (that is, `01` to `10`) the result is `011` and so the carry c is 0.

   (a) **Give the truth table** for the logic of c.

   (b) **Give the boolean expression** for c.

   (c) **Draw the circuit** for c.

3. Consider building the next state logic for a finite state machine that processes a sequence of 1s and 0s. Its input bit is named b, and its state bits are named s_1 and s_0. State bit s_1 is set to 0 if the machine has seen a sequence with an even number of 1s. s_1 is set to 1 if the machine has seen a sequence with an odd number of 1s. State bit s_0 is set to 0 if the machine has seen a sequence with an even number of 0s. s_0 is set to 1 if the machine has seen a sequence with an odd number of 0s.

   Thus, initially, the machine is in the state `00` because it has seen no 1s and no 0s. And, after the machine has seen the sequence of bits `01101`, then the machine is in state `10` because it has seen three 1s (an odd number) and two 0s (an even number).. Processing `01101` it goes through the state transitions

   $$\texttt{00} \xrightarrow{0} \texttt{01} \xrightarrow{1} \texttt{11} \xrightarrow{1} \texttt{01} \xrightarrow{0} \texttt{00} \xrightarrow{1} \texttt{10}$$

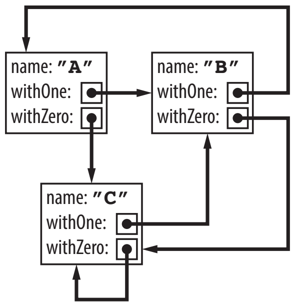
   Devise the truth table for the next state logic for this machine. When faced with the next input bit b when in state $s_1 : s_0$, the machine transitions to state $s'_1 : s'_0$. **Give the boolean expression** for s'_1 and s'_0 in terms of b, s_1, and s_0.

4. Let's use C to devise a state machine data structure. **Define a `State` struct** that contains three attributes below. A bunch of linked `State` structs defines a machine.

   - `name`: a string that names this state of the machine.
   - `withOne`: a pointer to a `State`. This is the next state when a 1 is processed.

- `withZero`: a pointer to a `State`, the next state when it processes a 0.

For example, consider the following diagram for a state machine that has three states named A, B, and C.



If it is in state C and it processes the sequence of bits 0,1,1,1 it ends up in state B.

(a) Consider the following snippet of C code that builds three `State` structs:

```
State A {"A",nullptr,nullptr};
State B {"B",nullptr,nullptr};
State C {"C",nullptr,nullptr};
```

Complete the code so that their `withOne` and `withZero` attributes mimic the structure of the diagram on the prior page.

(b) **Write a C function** `process` that takes three parameters: a pointer to a `State` struct, a pointer to an integer array containing 0s and 1s, and a integer describing the length of that array. It should return the name of the state that the machine would end up in, having processed that array of bits in order. For example, if we feed `process` the address of `C`, an array of length four with the sequence 0,1,1,1, and the integer 4, then it would return the string `"B"`.

5. **Write a C++ function** `primes` that takes a positive integer `n` and gives back a `vector` of integers. That vector should hold the first n prime numbers. The algorithm you use should check primeness by relying on the list of primes it has generated so far. For example, when fed 6 and if it has generated the vector with 2, 3, 5, 7 so far, then it should determine that 8, 9, and 10 are not prime by only checking amongst those numbers as possible divisors. When that call completes it should return the vector with the sequence 2, 3, 5, 7, 11, 13. The function should then always start its work with a vector of size 1 containing the single item 2.

6. Consider the following short exercises:

(a) Suppose we are writing a C++ `main` that has an integer `std::vector` named `v`. Express a lambda expression that, when given an integer parameter `x`, outputs to `std::cout` a line containing the smallest index in `v` whose value is `x`. If there is no such index, it shouldn't output anything.

(b) It's common to represent a 2-dimensional table as an array of arrays. Suppose we want to have a table `t` with entries of type `double`, and `r` rows and `c` columns. Write the code that declares the type of `t`, initializes/allocates its array of rows, allocates each of its rows, and then sets each entry in the table to 0.0. When complete, the code should have `t[i][j]` be the entry at row `i` and column `j` in the table.

(c) It's also common to represent a 2-dimensional table as a single, 1-dimensional array having `r * c` total entries, by performing a calculation on the row and column indexes `i` and `j` to compute the index in that one-dimensional array that holds that entry. Suppose the array is laid out first with the entries from row 0, then the entries from row 1, and so forth, what is the index of the entry at row `i` and column `j`? Now consider instead a three-dimensional table that has a number of rows `r`, columns `c`, and levels `l`, laid out as a one-dimensional array. Devise a calculation of the index for the entry at row `i`, column `j`, and level `k`.

(d) Consider a linked list made up of these two structs:

```
struct node {
    int data;
    struct node *next;
};
struct llist {
    node *first;
};
```

Assume that x is of type `llist`. Give the C expression for the data stored in the second node of its list (you can assume the list has length two or more). Give the C boolean expression for checking whether the list has length exactly two.

(e) Draw the stack and heap for the following C program just before it returns 0:

```
int main() {
    int x {6};
    int* y = &x;
    int* z = new int;
    *y = 10;
    *z = x;
    x++;
    return 0;
}
```

(f) Describe exactly what the problem is, or what the problems are, with this code:

```
int* whichIsTheMin(int *x, int *y) {
   if (*x < *y){
       return &x;
   } else {
       return &y;
   }
}
```