# CIS 2400 Fall 2022: Final Exam (CONVERTED TO RISC-V)

Dec 15, 2022

First Name : _Joel___________________________________

Last Name : _ Ramirez _______________________________

Penn ID : _________________________________________

Please fill in your information above, read the following pledge, and sign in the space below:

***I neither cheated myself nor helped anyone cheat on this exam. All answers on this exam are my own. Violation of this pledge can result in a failing grade.***

Sign Here : ___________________________________________

Exam Details & Instructions:

- There are 7 questions (and a short bonus 8th question) worth a total of 100 points.
- You have 120 minutes to complete this exam
- You will be provided with two pages containing references sheets. Do not put any answers on these pages, they will not be graded.
- The exam is closed book. This includes textbooks, phones, laptops, wearable devices, other electronics, and any notes outside of what is mentioned below.
- You are allowed two 8.5 x 11 inch sheets of paper (double sided) for notes.
- Any electronic or noise-making devices you do have should be turned off and put away.
- Remove all hats, headphones, and watches.

Advice:

- Remember that there are 7 questions (and a short bonus 8th question). Please budget your time so you can get to every question.
- Do not be alarmed if there seems to be more space than needed for an answer, we try to include a lot of space just in case it is needed.
- **Try to relax and take a deep breath.** Remember that we also want you to learn from this.

**Please put your initials at the top of each page in case the pages become separated.**

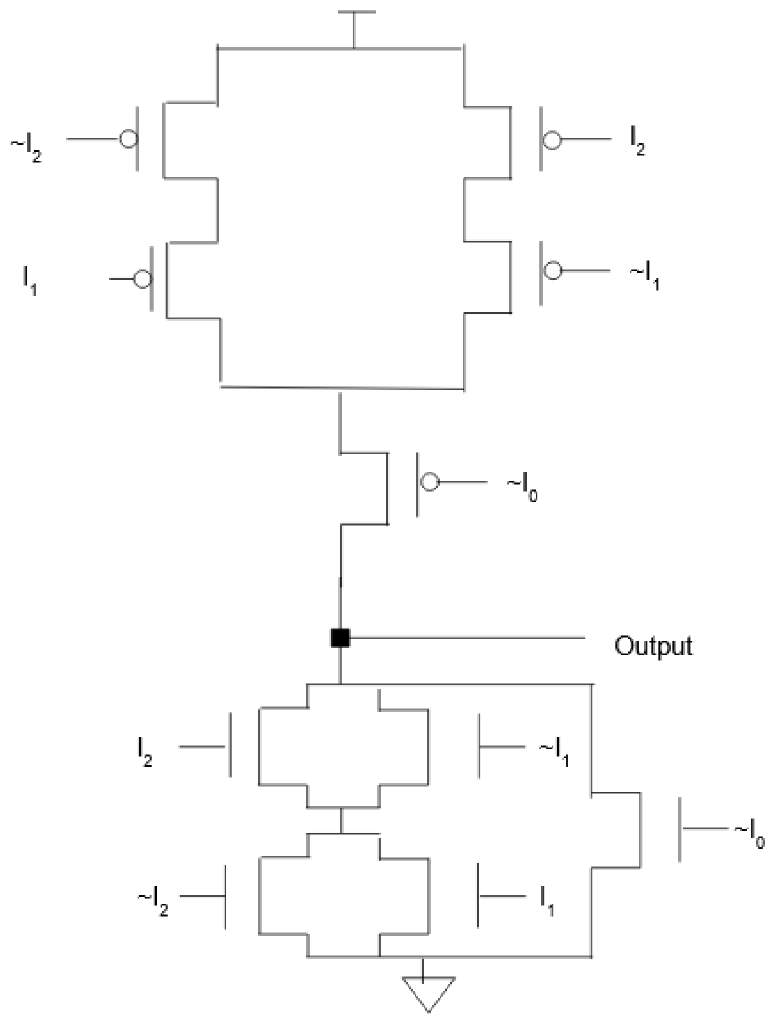
**If you need extra space, the back of the last page of this exam is blank for you as scratch space and to write answers. If you use it, please clearly indicate on that page and under the corresponding question prompt that you are using the extra page to answer that question. Please also write your full name and PennID at the top of the sheet.**

**Question 1 {15 pts}**

Your job is to design a circuit that will take as input a 3-bit value representing an unsigned integer and produces the output 1 if and only if the value is 3 or 5. For example, if I = 101 then the circuit should output 1. In your diagram $I_2$, $I_1$ and $I_0$ should indicate the 3 bits of the input where $I_2$ is the MSB and $I_0$ is the LSB. Remember that we cannot grade what we cannot read so please make your diagrams as neat and clear as possible.

**Part 1 {7 pts}:** Design a proper **CMOS** circuit which takes in all 3 bits of the input I and produces the output bit. You can assume that you also have access to negated versions of all of the input bits. Your solution must be a single CMOS circuit consisting of complementary pull-up and pull-down transistor networks. It should not involve cascading multiple CMOS circuits. Please label the inputs and outputs of your circuit clearly on your schematic.

**One Possible Solution**



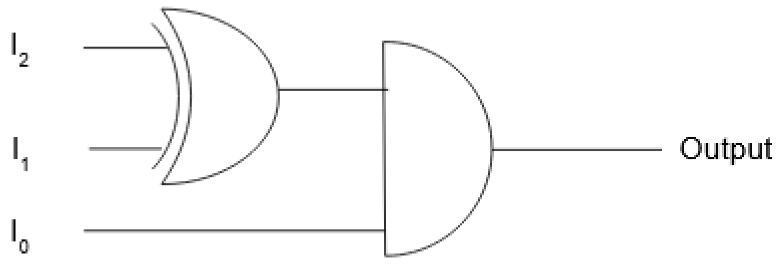
**Part 2 {8 pts}:** Design a gate level circuit that takes in the 3-bit input I and produces the correct output. In addition to producing the correct output, **your solution should only use inverters and 2-input gates.**

**Your solution should also minimize the number of gates used to make the circuit**. These include: NOT (inverter), AND, OR, XOR, NAND, NOR, XNOR.

You may use a single type of gate more than once, but **to get full credit the total number of used gates (including inverters) must be at most 3**. Partial credit will be given for correct solutions that use 4 gates, less partial credit for 5 gates, even less partial credit for 6 gates and no credit for anything that uses 7 or more gates. Note that using a PLA will not work for this question.

Please label the inputs and outputs of your circuit clearly on your schematic.

**One Possible Solution**



**Question 2 {15 pts}**
Your job is to write a RISC-V assembly subroutine that checks to see if an array is sorted.
The subroutine should take as inputs:

- a0: an address to the start of an array of integers in memory
- a1: The length of that array 'n'.

Your subroutine should check the array specified to see if it is sorted, storing 1 in a0 if it is sorted, and 0 in a0 if it is not sorted. The array is sorted if each element after the first element is greater than or equal to the previous element in the array. For example: [-12, 2, 5, 5, 7] is sorted but [3, 4, 3, 4, 5] is not sorted.

We have provided the assembler directives, some labels and some comments to start the program. Your program should execute a RET pseudo-instruction when it is finished.

Additionally, you should also note that:

- Since this is a subroutine, **<u>you can NOT modify RA, FP or SP</u>** in your program
- **<u>The data stored in the array that a0 points should not be modified</u>**
- The inputs in a0, and a1 have been set for you, you do not need to set the input values in your program
- a0 contains a valid address to an array in data memory
- a1 can be assumed to be at least 2
- a0 is where the result should be stored for this subroutine
- You are free to modify any of the registers that aren't RA, FP or SP.
- You are free to add additional labels and comments as needed

**The rest of this page can be used as work space to write pseudo code or whatever you like.**
**<u>Your answer should be written on the next page.</u>**

```
    .text
    .p2align 2
    .globl SORTED
SORTED:
   # start writing code here
   # There are other possible
   # solutions
   lw  t0, 0(a0)   #int prev = a0[0]
   li  t1, 1     # int i = 1
   li  t2, 1     # bool res = true
.LOOP:
   bge t1, a1, .BEFORE_END
   slli t3, t1, 2  # multiply by 4
   add t3, a0, t3
   lw  t3, 0(t3)
   slt  t2, t3, t0 # t2 = !(t0<=t3)
   slti t2, t2, 1 # t2 = !t2
   beq t2, x0, .BEFORE_END
   mv  t0, t3
   addi t1, t1, 1
   j .LOOP
.BEFORE_END:
    mv a0, t2
END:   # end of program
    ret
```

**Question 3 {5 pts}:**

The RISC-V architecture is designed to be small with only a few supported operations. While we can do a lot with the instructions provided, there can be desires to add new instructions to the language.

Adding new instructions to the RISC-V single cycle processor requires changing the hardware to some extent (at minimum in the decoder so that it can decode new instructions). Adding more transistors and logical units to our single-cycle processor design would have costs. One of these costs is that if we ran the same assembly code on our updated single-cycle processor and on the version before the change, the updated processor would either run the program with the same speed or slower.

Please explain why adding new instructions could make our processor slower. We are looking for a specific concept/idea though it doesn't have to be named directly. Please use 4 sentences at maximum to write your answer.

**If we add more instructions then that means we will need to add gates to the decoder and likely other parts of the processor. Doing this would mean we would likely increase the longest path of gates we need to go through to execute the instruction. So if we account for gate delay, then the instruction would take longer to execute.**

**Question 4 {10 pts} (This one is probably easier in RISC-V than it was in LC4)**
Consider the Single Cycle RISC-V instruction set that we walked through in class. Suppose we wanted to add a new instruction called INCR that incremented the value of a register by 1.

Mnemonic: INCR Rd
Semantics: Rd = Rd + 1
Encoding: 0000 0000 0001 dddd d000 dddd d001 0011

The usual rules for updating the PC apply.

Indicate how each of the following control signals should be set to execute this instruction. If a control signal doesn't matter for this instruction, you must instead write an X.

**Answer: This ends up being just a pseudo instruction for ADDI rd, rd, 1**

| Control Singal Name | Value |
|---|---|
| PCMux.CTL | **3** |
| regFile.WE | **1** |
| regInputMux.CTL | **0** |
| DATA.WE | **0** |
| PCAddMux.CTL | **X** |
| ALUInputMux.CTL | **1** |
| ALU.CTL | **0** |

**Question 5 {22 pts}**
Answer the following questions about the following assembly. This code was generated based off a C function that takes in a single parameter **`arg`** and returns an integer.

```
;;;;;;;;;;;;;;;;;;;;;;;;mystery;;;;;;;;;;;;;;;;;;;;;;;
        .text
        .p2align 2
        .globl mystery(int)
mystery(int):
        addi    sp, sp, -32
        sw      ra, 28(sp)
        sw      fp, 24(sp)
        addi    fp, sp, 32
        sw      a0, -16(fp)
        lw      a0, -16(fp)
        bnez    a0, .LBB0_2
        j       .LBB0_1
.LBB0_1:
        li      a0, 0
        sw      a0, -12(fp)
        j       .LBB0_3
.LBB0_2:
        lw      a0, -16(fp)
        andi    a1, a0, 1
        sw      a1, -20(fp)  #  THIS LINE <------------------
        srli    a0, a0, 1
        call    mystery(int)
        mv      a1, a0
        lw      a0, -20(fp)  #  THIS LINE <------------------
        add     a0, a0, a1
        sw      a0, -12(fp)
        j       .LBB0_3
.LBB0_3:
        lw      a0, -12(fp)
        lw      ra, 28(sp)
        lw      fp, 24(sp)
        addi    sp, sp, 32
        ret
```

**Part 1 {5 pts}: (This one is probably harder in RISC-V than it was in LC4)**
There were two lines of comments saying `#   THIS LINE <------------------`
Those two comments are on the same line as some RISC-V Instructions. What are the purpose of those two lines of assembly? What are they doing? Please describe how they help this assembly act as a function. **Do NOT just explain the exact actions that are carried out and do not say just say "Stores a value on the stack and later loads it".** Why is the value being loaded and stored on the stack in those two separate places? What is being loaded/stored?

**These two assembly instructions store and load a temporary value from a register onto the stack. The function has to do this in case the register gets overwritten when we call the function. From the perspective of a programmer a variable or temporary computation in a function should not change, but if we don't save that register in our stack frame, it could change when we call a function.**

**Part 2 {15 pts}:** The Assembly above fits into the following code skeleton. Fill in the missing parts below. Multiple blank lines have been given in some spots, and it is possible you may not need to fill in every line to get the correct answer.

```
int mystery(unsigned int arg) {
  if (arg == 0) {__________________
  __return 0;______________________
  }________________________________
  int temp = arg & 1;______________
  _________________________________
  _________________________________
  _________________________________
  _________________________________
  return temp + mystery(arg >> 1);
}
```

**Part 3 {2 pts}:** At a high-level, what does this function appear to be doing? Please justify your answer in 4 or fewer sentences.

**This function takes in an unsigned integer and recursively counts the number of 1 bits that are in the input argument.**

**Question 6 {20 pts}**

Complete the following function:

```
// Creates a copy of the array `arr` but all instances
// of `x` are removed. The new array should be dynamically
// allocated and returned through `output`.
//
// Arguments:
// - arr: the input array, whose contents should not be  //
modified. You can assume this pointer is valid. // - len: the
length of the input array. You can assume
//       len is >= 1.
// - x: The value that we are filtering on. There should be //
no elements with the value `x` in the output array.
// - output: an output parameter to "return" the resulting //
array. You can assume this pointer is valid.
// Return Value:
// - The length of the newly allocated array
// - Negative one (-1) on error

int filter(int * arr, int len, int x, int** output) {
    // this is just one possible solution
    // some people also just malloc'd len * sizeof(int)
    // instead of pre-calculating the length of the output array

    int new_len = 0;
    for (int i = 0; i < len; i++) {
        if (x != arr[i]) {
            new_len += 1;
        }
    }
    int* res = malloc(new_len * sizeof(int));
    if (res == NULL) {
        return -1;
    }
    int current_index = 0;
    for (int i = 0; i < len; i++) {
        if (x != arr[i]) {
            res[current_index] = arr[i];
            current_index += 1;
        }
    }
    *output = res;
    return new_len;
}
```

**Question 7 {12 pts }**
Consider the following two C functions that perform the same task:

```
int factorial_iter(int n) {
  int res = 1;
  int i;
  for (i = 2; i <= n; i++) {
     res = res * i;
  }
  return res;
}

int factorial_recursive(int n) {
  if (n <= 1) {
     return 1;
  }
  return n * factorial_recursive(n - 1);
}
```

When both functions are compiled, each function ends up being around 20 instructions if you include the prologue and the epilogue. Despite this, one of these is slower to execute for larger inputs than the other.

Consider the case where a user wants to calculate 10 factorial, which implementation would execute faster? That is, which one would complete the task in the shortest amount of time. **Please justify your answer in 4 or fewer sentences.**

Note: You do not need to calculate an exact number of instructions that are executed by each implementation to get full credit.

**The iterative one is likely faster. This is because for a larger value (like 10) we need to execute the prologue and epilogue for every recursive call, whereas the extra instructions for maintaining the loop is likely less than doing the prologue and epilogue over and over again.**

**Question 8 {1 pt; all exam submissions receive this point}**
Put anything you'd like the course staff to see here. This can be nothing, a piece of art, a message to members of the staff, anything you like! If you can't come up with anything, then a suggestion would be to write down your favorite aspect(s)/topic(s) of the course.

😊