Midterm Exam

CSC 252
25 March 2021
Computer Science Department
University of Rochester

Instructor: Yuhao Zhu
TAs: Rongcui Dong, Elana Elman, Kalen Frieberg, Sudhanshu Gupta, Yiyao (Jack) Yu, Vladimir Maksimovski, Nathan Reed, Rafaello Sanna

Name: _____

Problem 0 (2 points):	_____
Problem 1 (15 points):	_____
Problem 2 (10 points):	_____
Problem 3 (17 points):	_____
Problem 4 (13 points):	_____
Problem 5 (6 points)	_____
Problem 6 (12 points):	_____
Total (75 points):	_____
Extra Credit (10 points)	_____

Remember "**I don't know**" is given 15% partial credit, but you must erase everything else. This does not apply to extra credit questions.

Your answers to all questions must be contained in the given boxes. Use spare space to show all supporting work to earn partial credit.

You have 75 minutes to work.

Please sign the following. I have not given nor received any unauthorized help on this exam.

Signature:_____

GOOD LUCK!!!

Problem 0: Warm-up (2 Points)
Is assembly programming more fun than programming in Java?

Problem 1: Fixed-Point Arithmetics (15 points)
Part a) (4 points) Represent the decimal number 92 in binary.

Part b) (4 points) What is the decimal representation of the base 5 number 243?

Part c) (4 points) What is the 2's complement representation of the decimal value -92? Assuming an 8-bit representation. Express your answer in hexadecimal.

Part d) (3 points) If 4-bit registers R1 and R2 contain the values 1100 and 0111 respectively, what are the values of the carry, overflow, and sign flags after the operation "add R1, R2"?

Problem 2: Floating-Point Arithmetics (10 points + 4 points extra credit)

Part a) (4 points) Put $19\frac{3}{8}$ into the binary normalized form.

Part b) (6 points) The IEEE has decided to introduce a new 14-bit floating-point standard, whose main characteristics are consistent with existing floating-point number representations that we discussed in the class.

The following is the encoding of $\frac{17}{64}$ in this 14-bit standard: `00110100010000`

How many bits are used for the exponent? How many for the fraction?

Part c) (4 points extra credit)

The IEEE 754 floating-point standard states that a NaN is considered *"quiet"* if its most significant fractional bit is 1, and *"signaling"* if its most significant fractional bit is 0. Some programming languages take advantage of this feature to encode pointers. In particular, a pointer is stored as the fractional bit of a *quiet* NaN. This technique is called "NaN boxing."

NaNs used for "NaN boxing" (i.e., store pointers) always have their second most significant fractional bit set to 1, whereas NaNs that are meant to store actual NaN values set that bit to 0.

Suppose a programming language were to use NaN boxing. The language uses the IEEE 754 32-bit floating point representation. What's the maximum number of bits in a pointer that can be stored in a NaN using NaN boxing?

Suppose we wanted to store a pointer value **`0xF3CB`** in a 32-bit NaN. Write the binary encoding of this NaN value. Pad any bits that are irrelevant to this problem with zeros.

Problem 3: Logic Design (17 points)

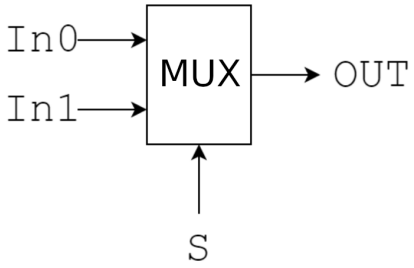
Part a) (6 points)

(3 points) What is the result of a bit-wise XOR between 0101 and 1001?

(3 points) What is the result of a bit-wise NOR between 0101 and 1001?

Part b) (6 points)

A two-input MUX selects between the two inputs (**`In0`** and **`In1`**) according to the select signal **`S`**. The diagram and truth table of a 2-input MUX are shown below. A MUX gate simply sets its output **`OUT`** to **`In0`** if **`S`** is 0 and sets **`OUT`** to **`In1`** if **`S`** is 1.



S	OUT
0	In0
1	In1

We can construct other basic logic gates using only MUXes. For instance, we can construct a 2-input OR gate that computes **`A OR B`** using a two-input MUX by setting its inputs as follows:

- `In0 = B`
- `In1 = constant value of 1`
- `S = A`

This will give us **`A OR B`** at **`OUT`**.

(3 points) Construct a NOT gate that computes **`!A`** using only one two-input MUX. **`A`** is the input signal to the NOT gate. Specify what **`In0`**, **`In1`**, and **`S`** should be in this MUX.

In0 is:

In1 is:

S is:

(3 points) Construct a two-input AND gate that computes **A AND B** using only one two-input MUX. **A** and **B** are the input signals to the AND gate. Specify what **In0**, **In1**, and **S** should be in this MUX.

In0 is:

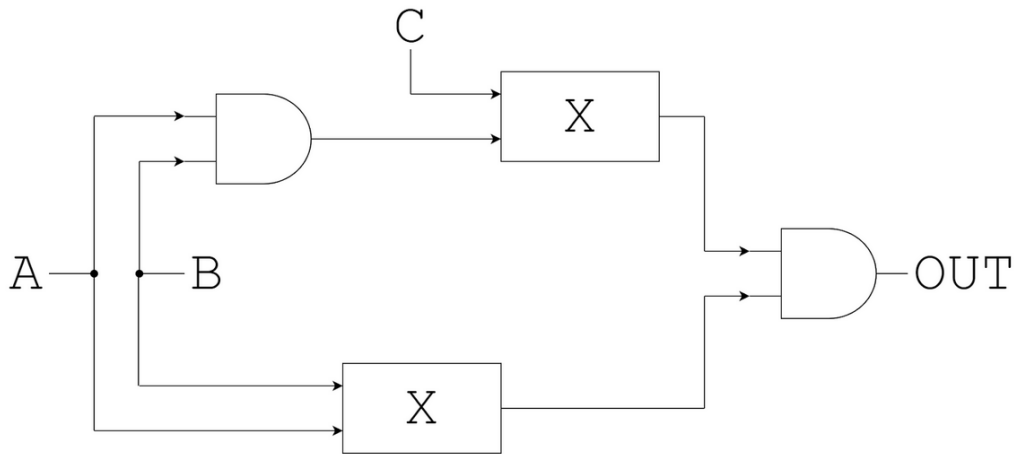
In1 is:

S is:

Part c) (5 points)

(3 points) The combinational circuit shown below takes in three 1-bit inputs: **A**, **B**, and **C,** and produces one 1-bit output: **Out**. The relationship between **A**, **B**, **C,** and **Out** is shown in the accompanying truth table.

This circuit contains two identical but unknown gates X. What is gate X so that the circuit matches the given truth table?



A	B	C	Out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

X is:

(2 points) Assuming the delay of each gate is 1ps, what is the delay of the entire circuit?

Problem 4: Assembly Programming I (13 points)

Conventions:

1. For this section, the assembly shown uses the syntax **`opcode src, dst`** for instructions with two arguments where **`src`** is the source argument and **`dst`** is the destination argument. For example, this means that **`mov a, b`** moves the value **`a`** into **`b`** and **`sub a, b`** computes the value (**`b`** - **`a`**) and stores it in **`b`**. All C code is compiled on a 64-bit machine, where arrays grow towards higher addresses.
2. Also, for functions that take two arguments, the first argument is stored in **`%rdi`** and the second is stored in **`%rsi`** at the time the function is called. The return value of this function is stored in **`%eax`** at the time the function returns.

Consider the following code:

```
typedef struct {
      char netid[12];
      char *name;
      int sid;
      double gpa;
} Student;

void print_ten_students() {
  Student[10] students;

  print_gpa(students[0].gpa);
  print_name(students[3].name);
  print_studentid(students[3].sid);
}
```

Assume that the beginning address of **`students`** is stored in **`%rbx`**. Also assume that the compiler doesn't reorder struct fields.

(10 points) The following table contains source code lines from **`print_ten_students()`** and the corresponding assembly snippets. The letters A, B, C, D, and E represent blanks in the assembly code. Fill in the blanks with appropriate instructions or hexadecimal offsets so that the function call succeeds.

C function	Assembly (Fill in the blanks)
`print_gpa(students[0].gpa);`	`movq  __A__(%rbx),%rdi` `callq  40118e <print_gpa>`
`print_netid(students[3].netid);`	`__B__  (%rbx,0x3,__C__),%rdi` `callq  401136 <print_netid>`

`print_studentid(students[3].sid);`	`lea  (%rbx,0x3,__D__),%rdi`
	`movq  __E__(%rdi),%edi`
	`callq  40116c <print_studentid>`

A:

B:

C:

D:

E:

(3 points) How to reorder fields in the struct to be more space efficient? Explain your answer.

Problem 5: Assembly Programming II (6 points + 2 points extra credit)

The assembly code of a function is shown below, along with the relevant part of the memory. For **all questions in this part**, assume `%rdi = 0x7ffffffee3f0` at the beginning of the code.

The **`jns label`** is a jump instruction that jumps to **`label`** if and only if the sign bit is set to 0. **The same assembly programming conventions in the previous problem still apply.**

Assembly Code	Data (memory position: 8-byte value)

`.L2:`
`  testq  %rdi, %rdi`
`  je  .L6`
`  cmpq  %rsi, (%rdi)`
`  jns  .L4`
`  movq  0x10(%rdi), %rdi`
`  jmp  .L2`
`.L4:`
`  je  .L7`
`  movq  0x8(%rdi), %rdi`
`  jmp  .L2`
`.L6:`
`  xorl  %eax, %eax`
`  ret`
`.L7:`
`  movl  $0x1, %eax`
`  ret`
	`0x7ffffffee3f0:  0x000000000000000c`
	`0x7ffffffee3f8:  0x00007ffffffee408`
	`0x7ffffffee400:  0x0000000000000000`
	`0x7ffffffee408:  0xfffffffffffffff9`
	`0x7ffffffee410:  0x00007ffffffee438`
	`0x7ffffffee418:  0x00007ffffffee420`
	`0x7ffffffee420:  0x0000000000000000`
	`0x7ffffffee428:  0x0000000000000000`
	`0x7ffffffee430:  0x0000000000000000`
	`0x7ffffffee438:  0xfffffffffffffff4`
	`0x7ffffffee440:  0x0000000000000000`
	`0x7ffffffee448:  0x00007ffffffee3f0`

(2 points) When the function is called with argument `%rsi = 0`, does the program terminate? If so, what is the value stored in `%eax` after the function returns?

(2 points) For what values of `%rsi` does this procedure run into an infinite loop? Include the previous question's `%rsi`, if you found the program not to terminate under that `%rsi`. Write all `%rsi` values in **decimal**, and in **increasing order**.

(2 points) How would you fix all infinite loop cases by modifying **exactly one** 8-byte value in memory (must be aligned)? You can modify that 8-byte data to whatever you want. The function output should stay the same for `%rsi` values for which the function was working correctly. In other words, for all `%rsi`, if the function returned before the modification, the function should return the same value before and after the modification.

(2 points extra credit) What kind of data structure does this function most likely manipulate? Be specific: answering "array" gets no points.

Problem 6: ISA + Microarchitecture (12 points + 4 points extra credit)

Part a) (12 points)
Consider the following x86 assembly code fragment:

```
mov $0x02, %rax
nop
nop
.L1:
cmpq %rax, $0x01
jle .L2
sub $0x01, %rax
nop
nop
cmpq %rax, $0x01
jge .L1
.L2:
xchg %rdi, %rdi
add $0x99, %rsi
mov %rsi, %rax
ret
```

(3 points) How many cycles does it take to execute this code on a single-cycle, sequential machine?

We will now execute the program on two CPUs that are pipelined differently. Assume that both CPUs have a simple 1-bit branch predictor and that the branch predictor is initialized to predict "taken". The 1-bit branch predictor works by storing a single bit somewhere in the CPU. All the jumps in the program (no matter where they occur) are **predicted by this one single bit**.

The first CPU has a 5-stage pipeline similar to the one discussed in class with (F)etch, (D)ecode, (E)xecute, (M)emory, and (W)riteback stages. With this CPU, the jump outcome, i.e., whether the jump will be taken or not, is not known until the jump instruction itself finishes the E stage.

The second CPU has a 3-stage pipeline with Fetch (F), Decode (D), and Memory/Execute (MX) stages. With this CPU, the jump outcome is not known until the jump instruction completes the D stage.

Hints:

- Recall how a 1-bit branch predictor works: it simply stores whether the last jump was taken or not taken and uses that information to predict whether a future jump will be taken or not taken (e.g., if the last jump was taken, we predict that a future jump will be taken). It is updated as soon as we know if a jump will actually be taken or not.
- If a jump is mispredicted, the correct next instruction will enter Fetch in the cycle after the jump outcome is known.
- A CPU, with branch prediction, will always fetch the next instruction (that the CPU predicts to be the correct next instruction) the cycle after it fetches the jump instruction.

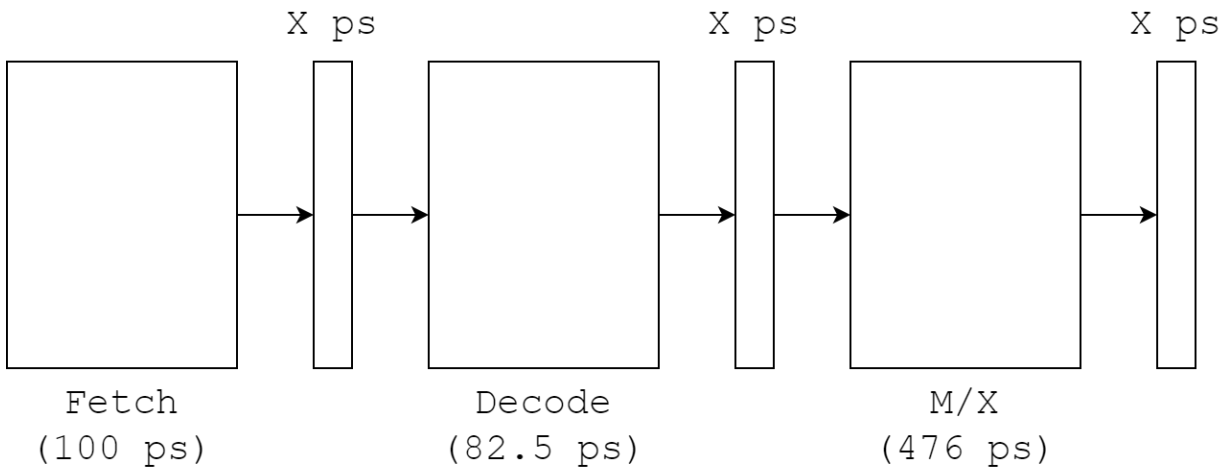
(3 points) During the execution of the code fragment, how many instructions get fetched as a result of a misprediction for the 5-stage pipeline? How many get fetched as a result of misprediction for the 3-stage pipeline?

(3 points) How many cycles does each CPU lose/waste every time it mispredicts?

(3 points) Assume the first CPU is running at 1GHz (cycle time 1 ns) and the second CPU is running at 600MHz (cycle time 1.6666... ns). Which CPU executes the code fragment faster (in less time, from fetching the first instruction to finishing the last instruction)? Explain.

Part b) (4 points extra credit)

Consider a 3-stage pipeline with Fetch (F), Decode (D), and Memory/Execute (MX) stages. The diagram is given below.



The three stages take 100 ps, 82.5 ps, and 476 ps, respectively. The fastest clock frequency this pipeline is capable of is 2.0 GHz. How long must it take to latch data into each of the 3 intermediate pipeline registers? Answer in picoseconds. A picosecond (ps) is 10^{-12} second.