

Midterm Exam
CSC 252
6 March 2026
Computer Science Department
University of Rochester

Instructor: Yuhao Zhu

TAs: Aarav Ahuja, Hannah Davidon, Han Yan, Junjie Zhao

Name: _____

Problem 0 (2 points):

Problem 1 (12 points):

Problem 2 (14 points):

Problem 3 (22 points):

Problem 4 (24 points):

Total (74 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 (if any).

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: _____

Have a good spring break and GOOD LUCK!!!

Problem 0: Warm-up (2 Points)

What's the most surprising thing you've learned so far?

Problem 1: Fixed-Point Arithmetics (12 points)

Part a) (2 points) Represent decimal number 67 in the hexadecimal form.

0x43

Part b) (4 points) Represent octal (base 8) number 67 in the decimal form **and** binary form.

55₁₀ and 110111₂

Part c) (6 points) Consider two signed binary numbers $A = 11110$ and $B = 00001$, both are represented using two's complement representation. Do the math below.

(2 points) What's the result of $\neg A \vee B$ ($\neg$ is bit-wise NOT and $\vee$ is bit-wise OR)?

00001

(2 points) What's the result of $A \oplus B$ ($\oplus$ is bit-wise XOR)?

11111

(2 points) How to write A in base 5?

-2₅

Problem 2: Floating-Point Arithmetics (14 points + 4 points extra credit)

Part a) (4 points)

(2 points) Give the binary representation of $11\frac{7}{8}$.

1011.111

(2 points) Write 183.875 using the normalized binary scientific notation.

$1.0110111111 \times 2^{-7}$

Part b) (4 points + 2 points extra credit) Your friend William is fine-tuning a Large Language Model. When William attempts to load the model parameters into the GPU in FP32 (IEEE 754 single-precision) format, the GPU triggers an “out of memory” error. Your friend instead loads the model parameters into the GPU using the FP16 format, which follows the principles of the IEEE-style floating point representation we discussed in the class and has the following format.

- FP16: 10 fraction bits, total 16 bits

(2 points) How many fraction bits are in FP32 (IEEE 754 single-precision)?

23

(2 points) What is the bias for FP16?

$2^{5-1} - 1 = 15$

(2 points extra credit) Why using the FP16 format instead of the FP32 format to load model weights might not trigger the “out of memory” error?

FP32 takes 32bits to be represented and FP16 takes 16 bits to be represented. Therefore, using FP16 to load model weights will save around half of the GPU memory compared to using FP32.

Part c) (6 points + 2 points extra credit) In addition to FP16, William also tries to load model parameters in the FP8 format, which follows the principles of the IEEE-style floating point representation we discussed in the class with the following format:

- FP8: 3 fraction bits, total 8bits

(2 points) What is the bias for FP8?

$$2^{4-1} - 1 = 7$$

(4 points) Each parameter in the model is a number. Given the requirement that the smallest gap between two representable numbers is at most $\frac{1}{2^{-11}}$, which of the two options (FP16 or FP8) meets the requirement? Please show your work.

This problem has a typo, “ $\frac{1}{2^{-11}}$ ” should be “ $\frac{1}{2^{11}}$ ”. We will give you credit for if your answer is one of the following:

1. Both, because the smallest positive number that FP16 can represent is $2^{-14} \times 2^{-10}$, and the smallest positive number that FP8 can represent is $2^{-6} \times 2^{-3}$.
2. FP16, because the smallest positive number that FP16 can represent is $2^{-14} \times 2^{-10}$, but the smallest positive number that FP8 can represent is $2^{-6} \times 2^{-3}$.

(2 points extra credit) William finds that the large language model loaded in the FP16 format performs better than the model loaded in the FP8 format. What might be the reason?

Compared to weights (numbers) represented in FP16, weights (numbers) represented in FP8 will be rounded more aggressively, which causes the issue of low precision after arithmetic calculation.

Problem 3: Logic Design (22 points)

Part a) (4 points)

(2 points) What is the result of a bitwise NOR operation between 01010 and 11001?

00100

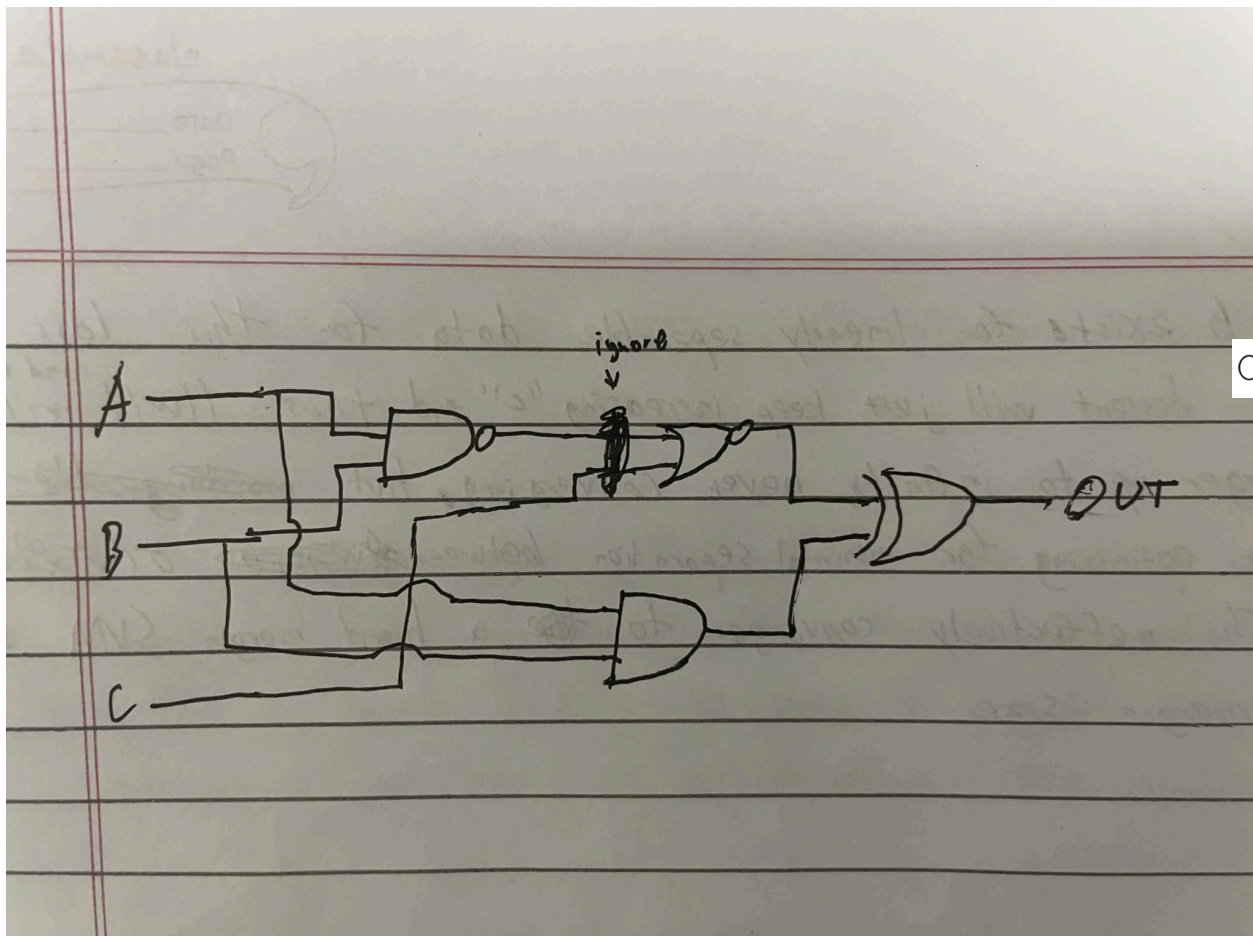
(2 points) What is the result of a bitwise XOR operation between 10011 and 11110?

01101

Part b) (12 points)

(4 points) Draw a logic circuit for the following expression:

$$\text{OUT} = \text{NOT}(\text{NOT}(\text{A AND B}) \text{ OR } \text{C}) \text{ XOR } (\text{A AND B})$$

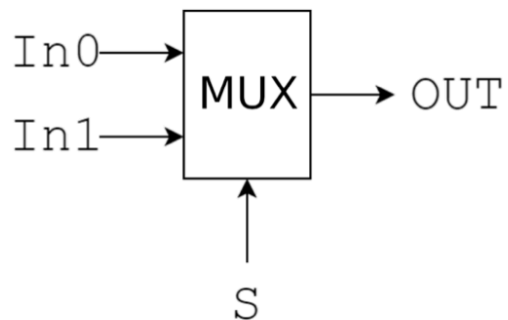


(8 points) Given the circuit you just drew, complete the following truth table. Additional columns are provided to show your partial work (for partial credit).

A	B	C	OUT				
0	0	0	0				
0	0	1	0				
0	1	0	0				
0	1	1	0				
1	0	0	0				
1	0	1	0				
1	1	0	0				
1	1	1	1				

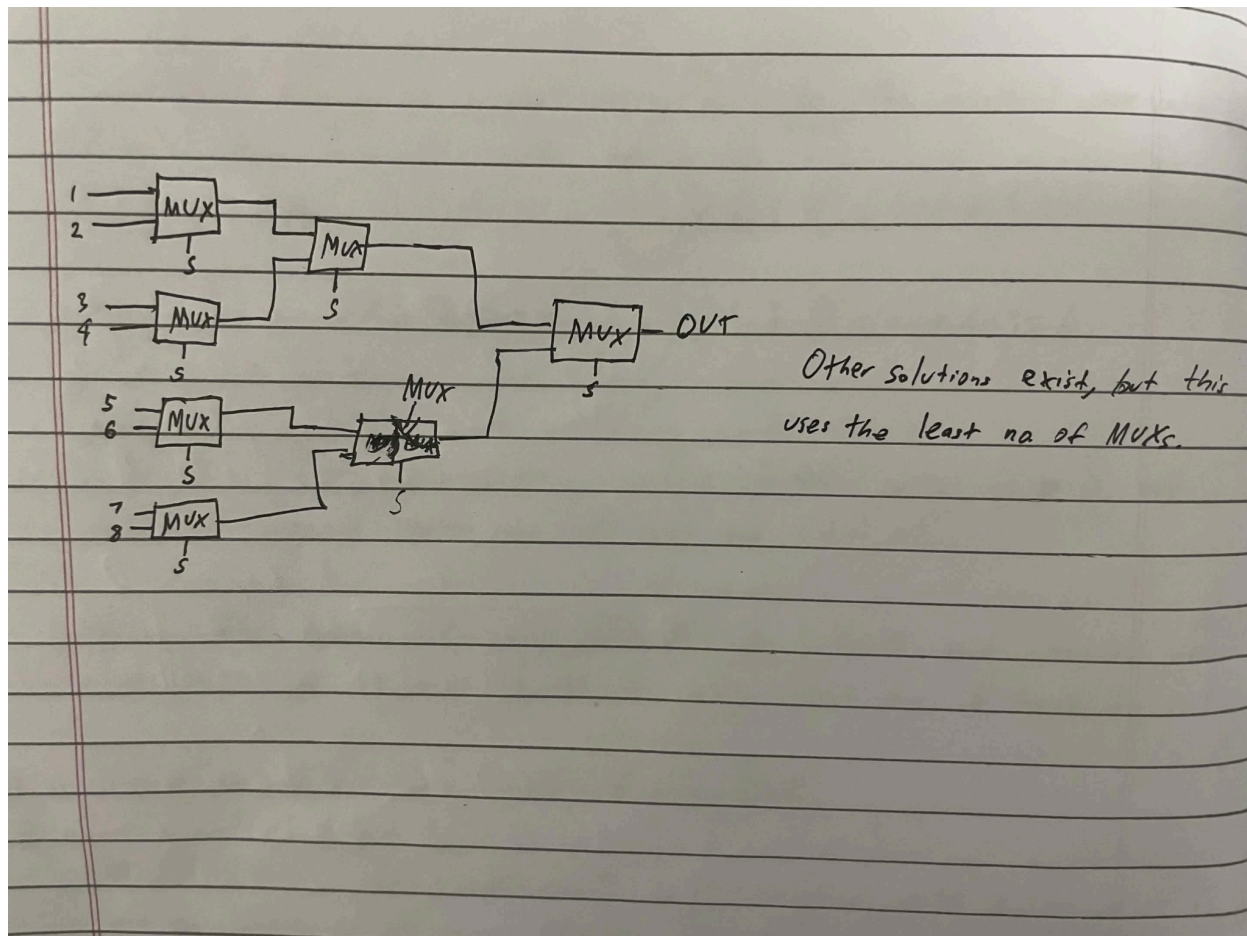
Part c) (6 points)

Below is a two-input MUX that selects the output OUT to In0 if S is 0 and IN1 if S is 1.



S	OUT
0	In0
1	In1

(4 points) How would you combine these MUXs to form an eight-input MUX? Draw your implementation below.



(2 points) Using the strategy above, how many two-input MUXs are necessary to form a 2^n -input MUX?

$$2^{n-1}$$

Problem 4: Assembly Programming (24 points)

Conventions:

1. For this section, the assembly shown uses the AT&T/GAS 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**.
2. All C code is compiled on a **64-bit machine**, where arrays grow toward higher addresses.
3. We use the x86 calling convention. That is, for functions that take two arguments, the first argument is stored in **%edi (%rdi)** and the second is stored in **%esi (%rsi)** at the time the function is called; the return value of a function is stored in **%eax (%rax)** at the time the function returns.
4. We use the **Little Endian** byte order when storing multi-byte variables in memory.

Part a) (14 points)

The following is the C definition for struct Data:

```
struct Data {  
    char character;  
    int *num1;  
    int num2;  
    int num_matrix[2][2];  
};
```

(2 points) What is the size of struct Data, assuming proper alignment?

40 bytes

(2 points) Assuming the address of struct Data my_data is stored in %rbp, write the assembly code to move my_data.num2 to register %rax.

```
mov -0x10(%rbp) %rax
```

(2 points) What is a reordering of the fields in struct Data that would allow it to be maximally space-efficient?

character, num_matrix, num2, num1

Consider the following function `gcd` and a snippet of the assembly code that implements the C function.

```
int gcd(int a, int b) {  
1      if (a == b) return a;  
2      else if (a > b) return gcd(a-b, b);  
3      else return gcd(a, b-a);  
}
```

```
0000000000400446 <gcd>:  
40044e:    mov     %edi,-0x4(%rbp)  
400451:    mov     %esi,-0x8(%rbp)  
...  
400461:    mov     -0x4(%rbp),%eax  
400464:    cmp     A,%eax  
400467:    B      40047d <gcd+0x37>  
400469:    mov     -0x4(%rbp),%eax  
40046c:    sub     -0x8(%rbp),%eax  
40046f:    mov     -0x8(%rbp),%edx  
400472:    mov     C,%esi  
400474:    mov     %eax,%edi  
400476:    call    400446 <gcd>  
40047b:    jmp     400491 <gcd+0x4b>  
...  
400491:    leave  
400492:    ret
```

Your Task: Fill in the missing instructions A, B, and C to complete the assembly code.

(2 points) A:

-0x8(%rbp)

(2 points) B:

jle

(2 points) C:

%edx

(2 point) What line of C code does the `call` instruction in 400476 implement?

2

Part b) (10 points)

Consider the following assembly code:

```
400600 <g>:  
400600: leaq 8(%rdi), %rax  
400604: retq  
  
400610 <f>:  
400610: callq 400600 <g>  
400615: addq $2, %rax  
400619: retq
```

Suppose `f` is called with the following initial register state:

- `%rdi = 10`
- `%rsp = 0x100`

Assume that the memory content at and below address `0x100` is initially unknown.

(2 points) Immediately after executing the instruction

```
400610: callq 400600 <g>
```

but before executing the first instruction of `g`, what is the value of `%rsp`?

0x0f8

(2 points) Immediately after executing the instruction

```
400610: callq 400600 <g>
```

but before executing the first instruction of `g`, what 8-byte value is stored at memory location `(%rsp)`?

0x00000000000400615

(2 points) After `g` finishes execution and returns to `f` (i.e., right before executing `addq $2, %rax`), what is the value of `%rax`?

18 (0x12)

(2 points) What value does the function `f` ultimately return?

20

(2 points) Write an equivalent C implementation of the assembly code above.

```
long f(long x) {  
    return (x + 8) + 2;  
}
```