Final Exam
CSC 252
1 May 2023
Computer Science Department
University of Rochester

Instructor: Yuhao Zhu
TAs: Yifan Zhu, Matthew Nappo, Yekai Pan, Toranosuke Ozawa, Zeyu Nie, Yumeng He, Stela Ciko, Nisarg Ujjainkar

Name: _____

Problem 0 (3 points):	_____
Problem 1 (17 points):	_____
Problem 2 (20 points):	_____
Problem 3 (24 points):	_____
Problem 4 (48 points):	_____
Problem 5 (16 points)	_____
Problem 6 (12 points)	_____
Total (140 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 180 minutes to work.

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

Signature:_____

GOOD LUCK and Have a Good Summer Break!!!

Problem 0: Warm-up (3 Points)

Are you grateful that 252 is not an elective?

Problem 1: Miscellaneous (17 points)

(3 points) Write `0xFACE` in binary.

1111 1010 1100 1110

(3 points) The `fork()` system call spawns a new thread in the parent process; True or False?

False

(3 points) `malloc()` allocates physical memory in DRAM; True or False?

False

(4 points) List two advantages of using virtual memory over physical memory.

Enable multitasking, good for safety, easy-to-use continuous addresses, enable swapping with disk, …

(4 points) What can happen if multiple threads access resources held by multiple locks?

A deadlock can occur (partial credit for race condition)

Problem 2: Floating-Point Arithmetics (20 points)

Part a) (6 points) Basic Arithmetics
(3 points) Write the result of 6+(16/64) in the normalized binary form

1.1001 * 2^2

(3 points) Compute $1.1 \times 2^{-2} \times 1.001 \times 2^{4}$. Write the result in the normalized binary form. Show your work to earn partial credit.

1.1 * 2^(-2) * 1.001 * 2^(4) = 1.1 * 1.001 * 2^(-2+4) = (1.001 + 0.1001) * 2^(-2) = 1.1011 * 2^(2)

Part b) (6 points) True or False questions.
(3 points) The IEEE 754 single precision floating point representation can be used to precisely represent all rational numbers between 0 and 1.

False

(3 points) The IEEE 754 double precision floating point representation can be used to precisely represent all real numbers between 0 and 1.

False

Part c) (4 points) Consider the following C code.

```
int x = 0x8f7;
int* pi = &x;
float* pf = (float*) pi;
```

Assume data is stored in little-endian format and `int` variables are 4 bytes aligned. Now we dereference `pf` and print its value, what will we get?

c

Explanation:
Whether we use little-endian or big-endian doesn't matter. `int` and `float` are both `4`-byte single values (but a string is not a single value as it is a variable length array of chars and this will make a trickier question), so no matter which endian we use, the order we read it is just the order we write it.

`0x8f7` is a relatively small number so bits are close to the least significant digits and its binary representation is `00 00 08 f7` (a subnormal as `float`).

Even if we read in a different endian, we get a byte-reversed order `f7 08 00 00` (a negative number near zero as `float`), NOT `7f 80 00 00` (the positive infinity as `float`).

(a) It will give a `Nan`
(b) It will give a floating point number equal to the integer `0x8f7`
(c) It will give a subnormal number
(d) There is a syntax error because we cannot cast an `int` pointer to a `float` pointer

Part d) (4 points) In most programming languages when we want to calculate 0.3 × 3.0, we won't get 9.0; instead we might get something like 0.89999999999999991. What is the most likely cause of this?

We cannot represent 0.3 or 0.9 precisely as a floating point number in binary. Rounding error for multiplication is not the key. (NOT expected to answer something like "hardware error" here because this is a very rare cause)

Problem 3: 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** and **cmp a, b** then **jge c** would compare **b** to **a** then jump to **c** if **b ≥ a.**
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.

The declaration of function `y()` is given below; its function body is intentionally incomplete. The first parameter of the function, `arr`, is the pointer to an array of 5 elements. `y()` will iterate over each element in the `arr` array exactly once and update each element if and only if a condition is met.

Consider the following assembly code. Before executing the code, `%rdi` contains a pointer to an array [17, 7, 10, 8, 15] and `%rsi` contains the value 10.

C code:

```
void y(long* arr, long b){
      …
}
```

Assembly code:

```
<irrelevant code omitted>
13          movq    %rdi, -24(%rbp)
14          movq    %rsi, -32(%rbp)
15          movq    $0, -8(%rbp)
16          jmp     .L2
17    .L4:
18          movq    -8(%rbp), %rax
19          leaq    0(,%rax,8), %rdx
20          movq    -24(%rbp), %rax
21          addq    %rdx, %rax
22          movq    (%rax), %rax
23          movq    %rax, -16(%rbp)
24          movq    -16(%rbp), %rax
```

```
25          cmpq    -32(%rbp), %rax
26          jge     .L3
27          movq    -32(%rbp), %rax
28          movq    %rax, -16(%rbp)
29    .L3:
30           _A_    $1, -8(%rbp)
31    .L2:
32          cmpq    $_B_, -8(%rbp)
33          jle     .L4
34          ret
```

(4 points) Which line between line 18 and line 30 is responsible for computing the offset of each array element in `arr`?

19

(3 points) At the first iteration of the loop (hint: that's when the value stored in `-8(%rbp)` is `0`) , what is stored in `%rax` at the end of line 22?

17

(3 points) What kind of programming structure is used in this function?

a. Do-while
b. While/for
c. Switch statement
d. None of the above

(6 points) Fill in **A** and **B** with the proper instruction so that the function behaves as desired.

A:

addq

B:

4

(4 points) Describe briefly what the condition is when updating each array element.

If element<b, changes element to b.

(4 points) What are the updated array values at the end of this function execution?

[17,10,10,10,15]

Problem 4: Microarchitecture/ISA (48 points)

You are working at Intel and are on the team for designing a new microarchitecture. Your manager gives you components for the standard five-stage pipeline: (F)etch, (D)ecode, (E)xecute, (M)emory and (W)riteback stages, with the same functionality as discussed in the class. The pipelined processor specification you manager give you is as follows:

- (F)etch, (D)ecode, (E)xecute stages take 15 ns
- The (M)emory stage takes 100 ns.
- The (W)riteback stage takes 135 ns
- After each stage there is a pipeline register which has a delay of 15 ns.

Part a) (**12 points**)
(3 points) What is the order of the 5 pipeline stages in a typical processor?

(F)etch, (D)ecode, (E)xecute, (M)emory, (W)riteback

(**3 points**) What is the shortest possible clock period for the specification your manager gives you?

150 ns

(**3 points**) Assuming no stalls or control dependencies of any kind, using the clock frequency you suggested, and that all the stages are occupied with instructions, how many instructions can this processor finish in 750 ns?

5

(**3 points**) At the end of which stage is the branch target resolved?

(E)xecute

Part b) (20 points)
Now you want to design an ISA for this machine assuming the following:

- 5 bit address space
- Memory is byte addressable
- Addresses are physical addresses (i.e., no virtual memory)
- Each instruction is 1-byte long; instructions can be padded with 0 at the end if needed
- 4 general purpose registers encoded as:

Register	Binary
`r0`	00
`r1`	01
`r2`	10
`r3`	11

- Three opcodes encoded as:

Name	Opcode	Behavior
`cmp`	001	Performs bitwise AND on two operands
`jif`	010	Conditional jump
`nop`	011	A no-op instruction

(**8 points**) Encode the following program in binary, assuming the instructions start at an absolute address of 0.

```
0: cmp r0 r1
1: jif 3
2: nop
3: nop
```

0: 0010 0010
1: 0100 0011 (or 0101 1000)
2: 0110 0000
3: 0110 0000

(**4 points**) How many cycles are expected to be lost when a branch is mispredicted? Write your explanation to earn partial credit.

2

(**4 points**) Assuming full pipelining, that there are no branch mispredictions, and that all jump instructions are not taken, how many cycles will it take to execute these instructions repeated 10 times? Show your math to earn partial credit.

4 + 40 = 44

(**4 points**) Assuming EVERY branch is mispredicted and that all jump instructions are not taken, how many cycles will it take to execute these instructions repeated 10 times? Show your math to earn partial credit.

4 + 40 + 20 = 64

Part c) (16 points)

Now you want to optimize the processor microarchitecture. You are told that you can split any of the stages (except the execute stage) into two stages, and each new stage will be half its original delay. These new stages cannot be split further. ANY number of consecutive stages can also be combined together and the delay of a so-combined stage is the sum of the constituting stages.

Reminder:

- When you split a stage into two pipeline stages, a new pipeline register must be inserted between the two new stages
- When you combine multiple stages into one stage, the pipeline stages between the constituting stages are no longer needed.

(**4 points**) Suppose you combine the first 4 stages. What is the shortest possible clock period after doing so? Show your math to earn partial credit.

160 ns

(**4 points**) Assuming no stalls or control dependencies of any kind and all the stages are occupied with instructions, which stage(s) do you need to combine or split so you can maximize the instructions executed per second? Write your explanation to earn partial credit.

Split the (M)emory and (W)riteback stages.

(**8 points**) On top of the decisions you made for your last question, which stage(s) should you split or combine if you are also concerned with minimizing the time lost from branch mispredictions? Write your explanation to earn partial credit.

Combine the (F)etch, (D)ecode, (E)xecute stages.

Problem 5: Cache (16 points)

For all the questions in this problem, assume that we are using a 16-bit machine with a byte-addressable memory and a N-way set-associative LRU cache (for some unknown N). The cache can hold up to 16 cache lines. Each cache line is 32 bytes (256 bits). There are 8 sets in total.

(**3 points**) How many bits do you need for the offset?

log(32) = 5

(**3 points**) How many bits do you need for the set?

log(8) = 3

(**2 points**) How many cachelines are there in each set?

2

(**8 points**) The following sequence of memory accesses generates the hits/misses as shown. Some miss/hit entries are intentionally left blank for you to figure out. The cache is initially empty. Note that addresses are written in binary with spaces added between each 4 bits for readability — these splitting points are not necessarily the tag/index/offset boundaries.

Fill in the blanks.

#	Address	Hit/Miss
1	`1100 1111 0000 0000`	Miss
2	`1101 1101 0010 0000`	Miss
3	`1101 1100 0010 0000`	Miss
4	`1100 1101 0001 1011`	Miss
5	`1100 1111 0000 0011`	Hit
6	`1100 1111 1001 0001`	Miss

7	1100 1101 0000 1111	Hit
8	1101 1100 0000 0000	Miss
9	1100 1111 0001 1111	Miss
10	1111 1111 0010 0000	Miss
11	1101 1101 0010 0000	Miss
12	0101 1001 0011 0100	Miss
13	1101 1101 0010 0100	Hit

Problem 6: Virtual Memory (12 points + 4 points extra credit)

Assume a byte addressable memory with the following characteristics:

1. Size of the virtual memory is 16 MB (1 MB = 2^{20} B)
2. Size of the physical memory is 4 MB
3. Page size is 256 Bytes
4. It uses one-level page table

Format of the PTE is shown below:

Metadata <2 bits>	PPN <n bits>

(**3 points**) How many bits do you need to represent the virtual page number (VPN)

16 bits

(**3 points**) How many bits do you need to represent the physical page number (PPN)

14 bits

(**3 points**) How many PTEs can you store in a page?

256/2=128

(**3 points**) How many pages does the page table occupy?

2^16/2^7 = 512

(**4 points extra credit**) Now we add a TLB to the machine above. The TLB has 16 entries and is direct-mapped. Which bits in the virtual address are used to index the TLB?