Final Exam
CSC 252
4 May 2022
Computer Science Department
University of Rochester

Instructor: Yuhao Zhu
TAs: Nisarg Ujjainkar, Abhishek Tyag, Kalen Frieberg, Gunnar Hammonds, Mandar Juvekar, Zihao Lin, Vladimir Maksimovski, Yiyao (Jack) Yu

Name: _____

Problem 0 (3 points):	_____
Problem 1 (12 points):	_____
Problem 2 (18 points):	_____
Problem 3 (25 points):	_____
Problem 4 (20 points):	_____
Problem 5 (26 points)	_____
Problem 6 (26 points):	_____
Total (130 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!!!

Problem 0: Warm-up (3 Points)

Do you think 252 could be taught in high school?

Problem 1: Miscellaneous (12 points)

Part a) (3 points) Write down the sum of 32_8 and 0x32 in base 2.

01001100

Part b) (3 points) Generally, the access time of a direct-mapped cache is ____ than that of a fully associative cache that is the same total size. Answer with <, >, =, <=, or >=.

<=

Part c) (3 points) (True or False) Virtual memory has no size limitation.

False

Part d) (3 points) Consider the following C struct Person:

```
struct Person{
      long id;
      float age;
      float weight;
      float height;
      char name[20];
      char sex[7];
      struct Person * nextPerson;
};
```

What's the size of the struct Person?

56 Bytes

Problem 2: ISA (18 points)

A stack-based ISA is designed. The ISA uses a hardware stack, and instructions in this ISA manipulate this stack. Each entry on the stack is one byte long, and the memory is byte-addressable. All instructions are 8-bit long, and are classified into two categories.

R Category

Binary encoding:

OpCode<7-5 bits>	00000<4-0 bits>

Instruction list:

Instruction	**OpCode**	**Role**
`pop`	001	Remove the item at the top of the stack.
`halt`	001	Halt the processor.

I Category

Binary encoding:

OpCode<7-5 bits>	Immediate value in 2's component<4-0 bits>

Instruction list:

Instruction	**OpCode**	**Role**
`pushi`	000	Push sign-extended immediate value on the stack.
`load`	101	Let the top entry of the stack be A. Compute `Address` = A + sign-extended immediate value; Pop A from the stack; Use `Address` to load a byte from the main memory and push the byte on the stack.
`store`	100	Let the top entry of the stack be A, and the second entry of the stack be B. Compute `Address` = A + sign-extended immediate value; Pop A from the stack; Store B at the memory location `Address`; Pop B from the stack.

`loadd`	110	`Address` = sign-extended immediate value. Load a byte from the memory using `Address` and push the byte on the stack.

Part a) (12 points) Encode the following instructions in binary.

(4 points) `store -8`

10011000

(4 points) `pushi 7`

00000111

(4 points) `halt`

00100000

Part b) (6 points) You want to implement a function that pops the top value from the stack and then pushes it back on the stack twice. Memory location at address 0 is reserved for this instruction to use temporarily. Write an assembly program using the existing instructions to implement this function.

```
pushi 0
store 0
loadd 0
loadd 0
```

Problem 3: Floating-Point Arithmetic (25 points)

Suppose that the IEEE decided to add a new *n*-bit floating-point standard, with its main characteristics consistent with the other IEEE standards. This *n*-bit standard can precisely represent the value $6\frac{3}{16}$, but cannot precisely represent $11\frac{7}{16}$. The smallest **positive normalized value** that can be represented in this standard is 2^{-30}.

Part a) (3 points) Convert $6\frac{3}{16}$ to Binary Normalized Form

$1.100011 * 2^2$

Part b) (3 points) How many of the *n* bits are fraction bits?

6

Part c) (3 points) What's the bias of this standard?

31

Part d) (3 points) How many of the *n* bits are exponent bits?

6

Part e) (3 points) What is *n*?

13

Part f) (10 points) Suppose that using this new IEEE standard you perform two separate calculations, assume nearest-even rounding is used:

1. $(256 + 2\frac{1}{4}) - 256$
2. $(256 - 256) + 2\frac{1}{4}$

What would be the result of these calculations? Do they both result in equivalent mathematically precise answers? Show your math to earn partial credit.

$1.00000000 * 2^8 +$

$1.001 * 2^1 =$

$1.00000000 * 2^8 +$

$0.0000001001 * 2^8 =$

$1.000000 * 2^8$: Mantissa can only be 6 bits long, round using nearest even

The first calculation's result would be the imprecise value 0, the second calculations result would be the precise value $2\frac{1}{4}$.

This is because when performing the operation $(2\frac{1}{4} + 256)$, the 6 fraction bits used in the standard don't allow for enough precision to properly represent the exact value $258\frac{1}{4}$, so this operation will resolve to the imprecise value 256. This is not an issue in the second calculation as $(256 - 256)$ resolves to 0 which can be added to $2\frac{1}{4}$ without a loss of precision.

Problem 4: Cache (20 points)

For all the questions in this problem, assume that we are using a 12-bit machine with a byte-addressable memory and a direct-mapped cache. The cache can hold up to 16 cache lines.

Part a) (3 points) How many bits do you need for the set index?

4

Part b) (17 points) The following sequence of 9 memory accesses generates the hits/misses shown. Some miss/hit entries are intentionally left blank. The cache is initially empty. Note that the addresses are written in binary with spaces added between each 4 bits for readability. These are not necessarily the tag/index/offset boundaries.

#	Address	Hit/Miss
1	1101 1111 0000	Miss
2	0000 1101 1111	Miss
3	1101 0111 0101	Miss
4	0000 1101 1100	Hit
5	1101 1111 0011	Miss
6	1111 0111 0010	Miss
7	1101 1111 0000	Miss
8	0000 1101 1101	Hit
9	1111 0111 0100	Miss

(3 points) What is the number of tag bits?

5

(3 points) What is the number of offset bits?

3

(3 points) What is the size of each cache line (ignore the valid bit, dirty bit, and tag bits etc.)? Show the formula you used to calculate this, and the value you get from it.

size of cache line = associativity * block size = 1 * 2^3 Bytes = 8 Bytes

(8 points) Fill the miss/hit for each of the blank entries.

Problem 5: Assembly Programming (26 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. For functions that take an argument, the argument is stored in **`%rdi`** at the time the function is called. The return value of this function is stored in **`%rax`** at the time the function returns.

Consider the assembly of a C function `void foobar(int *x)` which takes a single `int` pointer parameter `x`.

```
0000000000001159 <foobar>:
     1159:      mov   (%rdi),%eax
     115b:      inc   %eax
     115d:      mov   %eax,(%rdi)
     115f:      lea   0x2ecf,%rdi
     1166:      cmp   $0x1,%eax
     1169:      je    1177 <foobar+0x1e>
     116b:      cmp   $0x2,%eax
     116e:      jne   117c <foobar+0x23>
     1170:      lea   0x2eb4,%rdi
     1177:      call  1030 <puts@plt>
     117c:      ret
```

The addresses **`0x2ecf`** and **`0x2eb4`** contain the beginning of character strings **`"foo"`** and **`"bar"`** respectively. **`puts()`** is a standard C function that prints a character string given the string start address as a parameter.

Part a) (9 points)

(3 points) When the initial value of **`*x`** is 0, what is printed by the program?

foo

(3 points) When the initial value of **`*x`** is 1, what is printed by the program?

bar

(3 points) The `call` on line `1177` is replaced with a `jmp`. Would this program still run correctly? Explain your answer.

Yes. foobar() immediately returns after calling put() and does not put anything on its stack frame. Using jmp makes puts() return from foobar()'s stack frame cleanly.

Part b) (17 points)

Assuming that there are two threads, each of which executes the same `foobar()` function with the same exact parameter `x`. `*x` is initially set to `0`. For the sake of this problem, assume that the instructions inside the `puts()` function are always executed as an atomic unit (they are either executed together without interruption or not executed at all).

(4 points) What are all the possible values of `*x` after both threads finish execution of `foobar()`?

1 or 2

(4 points) What are all the possible strings printed by this multi-threaded program?

foobar, barfoo, foofoo

(3 points) Suppose the first three instructions (`1159` to `115d`) could be replaced with a single atomic instruction called `csc252`. Would this guarantee that the program always ends with `x` containing the value `2`? Show you work to earn partial credit.

x will always be 2 in the end. It is impossible for both threads to write back 1 to *x.

(3 points) Would this guarantee that the program always prints the same string no matter how many times you run it? Show you work to earn partial credit.

The program output can be either “foobar” or “barfoo” since the puts() can still be executed in any order.

(3 points) Assuming that the two threads execute on two different processors, each with a separate cache. What is something that the processor designers have to pay attention to in order to correctly implement the `csc252` instruction?

The csc252 instruction must ensure changes made to *x that are stored in cache must also be visible in memory (cache coherence).

Problem 6: Virtual Memory (26 points)

Assume a virtual memory system that has the following characteristics:

1. The virtual address space is 32 KB and is byte addressable
2. Physical memory size is 8 KB and is byte addressable
3. Page size is 128 Bytes
4. One level page table, where each page table entry contains a valid bit, a dirty bit, and the physical page number
5. PTBR is 0x3ADC
6. There is a data TLB that stores only the last page table entry

The format of a PTE is as shown below. MSB is the valid bit followed by the dirty bit. Last few bits are the physical page number (PPN).

valid<1 bit>	dirty<1 bit>	PPN<n bits>

Part a) (3 points) What are the number of physical and virtual pages?

8KB/128B = 64 = 2^6
32KB/128B = 256 = 2^8

Part b) (3 points) What is the total size of the page table?

256 * (6+1+1) bits = 256 B

Consider the following C program:

```
void fibbo(int a[64]) {
      a[0] = 0
      a[1] = 1
      for (int i=2; i < 64; i++) {
            a[i] = a[i-2] + a[i-1];
      }
 }
```

Suppose that the virtual address of the array '`a`' is 0x0400. Assume the data TLB is empty when the code starts execution. The table below shows a part of the main memory before the code executes.

Address	Data
3ADC	B8
3ADD	3A
3ADE	CD
3ADF	78
3AE4	F9
3CDC	B6
3DDC	4F
3EDC	F0

Part c) (4 points) How many pages does array '`a`' occupy?

2

Part d) (4 points) To read `a[1]`, what virtual page number(s) is(are) accessed?

0x8

Part e) (8 points) What physical memory addresses are accessed when reading `a[1]`?

First physical memory access is for the PTE. The address is PTBR + VPN = 0x3AE4 Then we access the actual data. The PTE (at 0x3AE4) has the data 0xF9, i.e., 1111 1001. So the physical page number is the last 6 bits, which are 111001. The page offset of the virtual address at 0x404 is 0000100, so the two concatenated gives us the physical address 1110010000100, which is 0x1C84. Since a[1] takes 4 bytes, accessing the data will reference four physical addresses: 0x1C84, 0x1C85, 0x1C86, 0x1C87.

Part f) (4 points) How many data TLB misses will occur in the execution of the program? Assume the access order of the line "`a[i] = a[i-2] + a[i-1];`" is `a[i-2]`, `a[i-1]`, `a[i]` with no other accesses in between.

There are two possible assumptions one could make about the machine. 1. The CPU register file can store 2 or more 32 bit integers. In this case the answer would be 2. A[0] -> 1 miss

A[32] = A[31]+A[30] -> 1 miss to access A[32]

2. The CPU register file can store only one 32 bit integer. In this case, there will be one memory access per iteration of the loop. The correct answer with this assumption would be 4.

A[0] -> 1 miss

A[32] = A[31]+A[30] -> 1 miss to access A[32]

A[33] = a[32]+A[31] -> 2 misses

The last instruction will want to access A[31], which is not in the register. Accessing A[31] will thus result in a TLB miss because it resides in the first page. This memory access results in the previous PTE being evicted from the TLB, which now contains the PTE for the first page. Accessing A[33] again results in a TLB miss because it resides in the second page.