Mid-Term Exam

CSC 252/452 Fall 2024
University of Rochester
Oct. 9th, 2024

Instructions:

1. This exam has 13 pages including this page. Check that your copy has all the pages.
2. Do not start the exam until 3:25 pm and do not continue working after 4:40 pm. You may ask a proctor if you are unsure of the time.
3. For fairness, the instructor and proctors will not answer questions about the exam content. If an exam question is unclear to you, make assumptions and write down any assumptions with your answer.
4. **For students who are taking 452**, all problems, including those marked as extra credit, are part of the total score.
5. "I don't know" is given 15% partial credit, but you must erase everything else. This does not apply to extra credit questions.
6. Your answers to all questions must be contained in the given boxes. Use spare space to show all supporting work to earn partial credit.
7. List of problems (75 points total + 3 points extra credit):
    - Problem 0 (2 points)
    - Problem 1 (17 points)
    - Problem 2 (14 points + 3 points extra credit)
    - Problem 3 (16 points)
    - Problem 4 (16 points)
    - Problem 5 (10 points)

Your name:

Signature:

Good luck!!!

Problem 0: Warm-up (2 Points)

Select all the options that apply. Multiple selections are allowed.

- ☐ I have been to TA office hours.
- ☐ I have asked a question on Piazza.
- ☐ I have found answers to my question on Piazza.
- ☐ I like programming in Assembly.

Problem 1: Integer Arithmetic (17 points)

Part a) (4 points) Represent the decimal number 74 in hexadecimal form.

4A

Part b) (4 points) Represent the hexadecimal number 0xA8 in decimal form **and** binary form.

decimal: 168
binary: 10101000

Part c) (4 points) What are the 2's complement representations of the decimal numbers -15 **and** 23? Assume an 8-bit representation.

-15: 11110001
23: 00010111

Part d) (3 points) Assume the bit stream $b_n b_{n-1} b_{n-2} \ldots b_1 b_0$ represents a number in 2's complement format. What is the result if we perform the following operations sequentially?

1. Flip all the bits in the bit stream.
2. Add 1 to the resulting bit stream from step 1.
3. Add the bit stream from step 2 to the original bit stream.

0

Part e) (2 points) Consider two 4-bit registers `R1` and `R2`; `R1`=1100, `R2`=0100 (both values are in binary form). What are the values of the carry, overflow, zero, and sign flags after the operation "`add R1, R2`"?

carry: 1, overflow: 0, zero: 1, sign: 0

Problem 2: Floating-Point Arithmetic (14 points + 3 points extra credit)

Part a) (6 points) Consider a decimal number F = 89.50

(2 points) Put F into the normalized scientific notation (in decimal).

8.95*10^1

(2 points) Give the binary representation of F. Use as many bits as needed to precisely represent the result.

1011001.1

Part b) (8 points) Assume we are using a new 14-bit floating-point standard whose characteristics are compliant with the floating-point representations we discussed in the class. For this representation, the exponent bias is 15.

(2 points) How many bits are used for exponent and fraction, respectively?

exp: 5 bits
frac: 8 bits

(2 points) With this standard, what is the floating-point representation of 0xB?

0 10010 01100000

(2 points) What is the smallest positive number that can be represented? Write it in this 14-bit floating-point format.

0 00000 00000001

(4 points) Given two numbers, A and B, represented using this 14-bit floating-point format:

- A = 11101100001111
- B = 00001000100001

Calculate A+B and provide the result in the 14-bit floating-point format.

111101100001111 (same with A)

(3 points extra credit) Using the IEEE 754 single-precision floating-point format, what is the smallest positive **integer** that **cannot** be precisely represented? You can write the answer in mathematical expression without evaluating it.

2^24+1

Problem 3: Assembly Programming (16 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`** (signed comparison).
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. We use **Row-Major** Ordering for 2D arrays.

Part a) (11 points)

Consider the following function **`p0`**. Its C code and assembly code are both partially given. Assume that the input arguments to **`p0`** are **`x=0`**, **`y=4`**.

C code:

```
unsigned long p0 (unsigned long x, unsigned long y)
{
      unsigned long t1, t2, t3;
      t1 = x*24+7;
      t2 = ......; //intentionally hidden
      if (t1 > 10)
               t3 = t2+t1*4+3;
      else
               t3 = t1;
      return ......; //intentionally hidden

}
```

Assembly code:

```
     # some irrelevant instructions at the end omitted
     movq %rdi, -40(%rbp)
     movq %rsi, -48(%rbp)
     movq -40(%rbp), %rdx
     movq %rdx, %rax
     addq %rax, %rax
     addq %rdx, %rax
     __A__ $3, %rax
     addq $7, %rax
     movq %rax, -16(%rbp)
     movq -16(%rbp), %rax
     addq $4, %rax
     movq %rax, -8(%rbp)
     cmpq $10, -16(%rbp)
     __B__ .L2
     movq -16(%rbp), %rax
     leaq 0( , __C__, __D__), %rdx
     movq -8(%rbp), %rax
     addq %rdx, %rax
     addq $3, %rax
     movq %rax, -24(%rbp)
     jmp  .L3
.L2:
     movq -16(%rbp), %rax
     movq %rax, -24(%rbp)
.L3:
     movq -24(%rbp), %rax
     addq $3, %rax
     # some irrelevant instructions at the end omitted
     ret
```

Complete the missing pieces in the assembly code.

(2 points) A:

salq

(2 points) B:

jbe

(2 points) C:

%rax

(2 points) D:

4

(3 points) What is the return value of `p0`?

10

Part b) (5 points)

Consider the following assembly code for a mystery function in C. `movq` is move quadword (64 bits), `movl` is move longword (32 bits).

```
movq -4(%rdi), %rdx
movq -8(%rsi), %rax
movq %rax, (%rsi)
addq $4, %rdx
movl %edx, -12(%rdi)
ret
```

Suppose that the left table on the next page shows the state of the memory before this function is called. Suppose that the registers are set to `%rdi = 0x224c` and `%rsi = 0x2248`.

Memory status before the function.

Address	Data
0x2240	0xaa
0x2241	0x4
0x2242	0xc
0x2243	0xb0
0x2244	0x1
0x2245	0x5
0x2246	0xf
0x2247	0x9
0x2248	0x10
0x2249	0x20
0x224a	0x30
0x224b	0x0
0x224c	0x6
0x224d	0xe
0x224e	0x4
0x224f	0x3

Memory status after the function.

Address	Data
0x2240	0x14
0x2241	0x20
0x2242	0x30
0x2243	0x0
0x2244	0x1
0x2245	0x5
0x2246	0xf
0x2247	0x9
0x2248	0xaa
0x2249	0x4
0x224a	0xc
0x224b	0xb0
0x224c	0x1
0x224d	0x5
0x224e	0xf
0x224f	0x9

In the right table above, fill in the state of the memory after the function is called.

Problem 4: Data Structure (16 points)

Conventions: same with the ones in Problem 3.

Consider the following C code. `CPU` is a structure to represent the critical features of a CPU.

```
struct CPU {
    char cache[2][3];
    double clock_speed_GHz;
    int cores;
    char *name;
};

struct CPU CPU_List[10];
```

(4 points) Without data alignment, what will be the value printed when we run `printf("%d\n", sizeof(struct CPU));`?

26

(4 points) If the start of `CPU_List[0]` is stored at `-0x30(%rbp)`, where in memory is `CPU_List[0].cache[1][2]` stored?

-0x2b(%rbp)

(4 points) With proper data alignment, assume `CPU_List = 0x7fff0008`, what is the value of `CPU_List+1`?

0x7fff0028

(4 points) How to reorder the members in the structure to be more space efficient? Explain your answer?

```
struct CPU{
   double clock_speed_GHz;
   char* name;
   int cores;
   char cache[2][3];
}
```

Problem 5: ISA (10 points)

The designers of Y86-64 are considering adding the `leaq` instruction to their ISA:

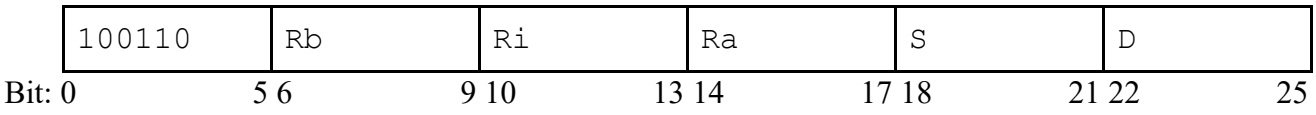
```
leaq D(Rb, Ri, S), Ra
```

This instruction calculates the effective address `Rb+Ri*S+D`, and stores the result in `Ra`. Here, `Rb`, `Ri`, and `Ra` are registers, while `S` and `D` are both 4-bit constants. For example, the following instruction calculates `%rax+%rdx*4+12`, and stores the result in `%rcx`:

```
leaq $12(%rax, %rdx, 4), %rcx
```

As we learned in class, `D(Rb, Ri, S)` represents the complete address mode. The elements `Rb`, `Ri`, `S`, and `D` could be omitted if not needed.

The Y86-64 designers proposed the following encoding format for this instruction:



The entire `leaq` instruction is 26 bits long. Bits 0-5 are used for the opcode, bits 6-9 are used for `Rb`, bits 10-13 are used for `Ri`, bits 14-17 are used for `Ra`, bits 18-21 are used for `S`, bits 22-25 are used for `D`.

Assume that the registers in Y86-64 are encoded using the 4-bit values shown in the table below:

Register	Encoding	Register	Encoding
%rax	0000	%r8	1000
%rbx	0001	%r9	1001
%rcx	0010	%r10	1010
%rdx	0011	%r11	1011
%rsi	0100	%r12	1100
%rdi	0101	%r13	1101
%rsp	0110	%r14	1110
%rbp	0111	No-register	1111

(3 points) Give the assembly form of the instruction encoded as `10011000100100001101001111`.

leaq 15(%rcx, %rsi, 4), %rdx

(3 points) When using the `leaq` instruction to compute the effective address of `4(%rax)` and store it in `%rdx`, what is the complete encoding of this instruction?

100110 0000 1111 0011 0001 0100

(4 points) Compared to fixed-length instruction encoding, what are the advantages and disadvantages of variable-length instruction encoding?

Advantages: good scalability, easy to extend, shorter length for simple instructions.
Disadvantages: harder to decode