4190.308: Computer Architecture
Midterm Exam
November 7th, 2017
Professor Jae W. Lee

Student ID #: ____________________

Name: ____________________

This is a closed book, closed notes exam.
90 Minutes
14 Pages
(+ 2 Pages for Appendices)

Total Score: 200 points

Notes:

- Please turn off all of your electronic devices (phones, tablets, notebooks, netbooks, and so on). A clock is available on the lecture screen.
- Please stay in the classroom until the end of the examination.

- You must not discuss the exam's contents with other students during the exam.
- You must not use any notes on papers, electronic devices, desks, or part of your body.

(This page is intentionally left blank. Feel free to use as you like.)

Part A: Short Answers (20 points)

Question 1 (20 points)

Please answer the following questions. You don't have to justify your answer—just write down your answer only.

Don't guess! You will get 4 points for each correct answer and lose 4 points for each wrong answer (but 0 point for no answer).

(1) When performing multiple *integer* additions, the order of additions does not affect the final result since addition is commutative. (True/False)

ANSWER: True

(2) The starting address of a structure in C (i.e., `struct { … }`) must be a multiple of 8. (True/False)

ANSWER: False

(3) RISC architectures generally have fewer registers than CISC architectures. (True/False)

ANSWER: False

(4) Variable `x` has a 4-byte value of `0xdeadbeef` and the address of `x` is `0x400`. Assuming a *big-endian* architecture, how is the value stored in memory? Fill the bytes at the right locations.

Address	`0x3FC`	`0x3FD`	`0x3FE`	`0x3FF`	`0x400`	`0x401`	`0x402`	`0x403`	`0x404`
Value					**0xde**	**0xad**	**0xbe**	**0xef**	

(5) Assuming `x` is an `int` variable in C, if `x >= 0`, the following inequality *always* holds: `-x <= 0`. (True/False)

ANSWER: True

Part B: Floating-Point Numbers (20 points)

Question 2 (20 points)

To accelerate deep learning applications, Company M has introduced a new floating-point format, called *m$-fp9*, which is a 9-bit floating-point representation based on the IEEE 754 standard. The most significant bit represents a sign bit. The next three bits are the exponent with an exponent bias of 3. The last five bits are the fraction. The rules are like those in the IEEE standard (normalized, denormalized, representation of zero, infinity, and NaN).

Sign (1 bit)	Exponent (3 bits)	Fraction (5 bit)

(1) Fill in the empty boxes in the following table. (1 for each)

Number	**Decimal Representation**	**Binary Representation**
Positive Zero	+0.0	0 000 00000
Negative Zero	-0.0	1 000 00000
3.25_{10}	3.25 (13/4)	0 100 10100
0.125_{10}	0.125 (1/8)	0 000 10000
One	1.0	0 011 00000
Positive Infinity	$+\infty$	0 111 00000
Negative Infinity	$-\infty$	1 111 00000
The largest number	15.75 (63/4)	0 110 11111
The smallest positive number	2^{-7} (1/128)	0 000 00001

(2) Using the *denomalized* form, (a) how many non-zero values can be represented?; (b) what is the maximum denomal number?; (c) what is the smallest positive non-zero denormal number? (3 for each)
(a) 62 (b) 31/128 (c) 1/128

Part C: Human x86-64 Compiler (46 points)

Question 3 (22 points)

Alice Hacker wrote the following C code to run it on x86-64/Linux system.

```
#include <stdio.h>
int switch_func(int x, int y, int z)
{
  int res = 1;
  switch (x) {
    case 1:
      res = y * z;
      break;
    case 3:
      res += y;
    case 4:
      res -= z;
      break;
    case 6:
    case 7:
      res = y/z;
      break;
    default:
      res = 3;
  }
  return res;
}
int main() {
  int result1 = switch_func(3, 2, 1);
  int result2 = switch_func(2, 3, 4);
  printf("Welcome to CA world\n");
  printf("Result 1 = %d\n", result1);
  printf("Result 2 = %d\n", result2);
  printf("Result 3 = %d\n", switch_func(1, 5, 7));
  return 0;
}
```

(1) Fill in the blanks from the output of the program. (8, 6, 3)

```
Welcome to CA world
Result 1 = ___2____
Result 2 = ___3____
Result 3 = ___35___
```

(2) The assembly code for `switch_func()` is shown below. Fill in the jump table to make the program work correctly. The first entry is already provided as a reference. (2 for each)

```
switch_func:
  movl  $1, %eax
  cmpq  $8, %rdi
  ja    .L2
  jmp   *.L0(,%rdi,8)
.L2:
  movl  $3, %rax
  ret
.L3:
  movq  %rsi, %rax
  imulq %rdx, %rax
  ret
.L4:
  subq  %rdx, %rax
  ret
.L7:
  addq  %rsi, %rax
  jmp   .L4
.L8:
  movq  %rsi, %rax
  cqto
  idivq %rdx
  ret
```

```
# Jump Table
.section    .rodata
  .align  8
.L0:
  .quad   .L2    # x == 0
  .quad   .L3    # x == 1
  .quad   .L2    # x == 2
  .quad   .L7    # x == 3
  .quad   .L4    # x == 4
  .quad   .L2    # x == 5
  .quad   .L8    # x == 6
  .quad   .L8    # x == 7
```

Question 4 (24 points)

Ben Bitdiddle is writing an assembly code, `swap.s`, as shown below together with the original C code (`swap.c`). His code is currently incomplete as the part that swaps the values of `%rax` and `%rdx` is missing. Fill in the missing part in `swap.s` *without using any temporary storage in either register or memory*. (correct = 24, incorrect but tried using xor || and,sub = 12, else = 0)

```
/* swap.c */
# include <stdio.h>

int main () {
    int x = 3, y = 1;
    printf("x = %d, y = %d\n", x, y);  // x = 3, y = 1
    ... // swapping x and y (omitted)
    printf("x = %d, y = %d\n", x, y);  // x = 1, y = 3
    return 0;
}
```

```
# swap.s
# x in %rax, y in %rdx
.main
    pushq  %rbp
    mov    %rsp, %rbp  # initiate procedure
    mov    $3, %rax
    mov    $1, %rdx
    callq  0x400450 <printf@plt>
    # Implement the swap between %rax (=x) and %rdx (=y)
    xor    %rax, %rdx
    xor    %rdx, %rax
    xor    %rax, %rdx









    # finish the function
    callq  0x400450 <printf@plt>
    mov    $0x0, %eax
    leaveq
    ret
```

Part D: Human x86-64 De-compiler (44 points)

Question 5 (20 points)

Consider the source code below, where M and N are constants defined by `#define` (not shown).

```
int array1 [M][N];
int array2 [N][M];

int copy(int i, int j)
{
    array1[i][j] = array2[j][i];
}
```

Suppose the code above generates the following assembly code:

```
copy:
    pushq %rbp
    movq  %rsp, %rbp
    pushq %rbx
    leaq    (%rdi, %rdi, 8), %rdx
    salq    $2, %rdx
    movq  %rsi, %rax
    salq    $4, %rax
    subq   %rsi, %rax
    salq    $2, %rax
    movq array2(%rax, %rdi, 4), %rax
    movq %rax, array1(%rdx, %rsi, 4)
    popq  %rbx
    movq %rbp, %rsp
    popq  %rbp
    ret
```

What are the values of M and N? Infer those values from the assembly code.

M = 15

N = 9

Question 6 (24 points)

In the following C code the definitions of both `struct node` and function `func` are incomplete.

<pre>typedef struct node {
   __long__ x;
   __long__ y;
   struct node *prev;
   struct node *next;
   struct node *root;
} node_t;</pre>
	<pre>void func (node_t *n)
	{
	   node_t *m;
	   m = _n->next_;
	   m->y /= 16;
	   return m;
	}</pre>

The `func` function is complied into the following assembly code.

```
func:
    pushq %rbp
    movq %rsp, %rbp
    movq 24(%rax), %rax
    shrw $4, 8(%rax)
    movq %rbp, %rsp
    popq %rbp
    ret
```

Fill in the blanks in the C code above. Note that there is a unique answer.

Part E: Procedure Calls (30 points)

Question 7 (30 points)

Here is a C function (`foo`) vulnerable to a buffer overflow attack, which is one of the most common security threats for today's computers. This function has a security hole as it does not check the length of the string. Here are some notes about the function.

9

- Function `gets(buf)` receives a character string from the standard input (keyboard) until the user hits the <enter> key and store it to `buf`. A null character (`'\0'`) is automatically appended at the end of the string to form a valid C string.
- Characters '`0`' through '`9`' have ASCII codes `0x30` through `0x39`.
- C strings are null-terminated (i.e., terminated by a character with value `0x00`).

```
void foo(int x){
  int a[3];
  char buf[4];
  a[0] = x;
  a[1] = 0xBFFFFF2D;
  gets(buf);
  printf("a[0] = 0x%x, a[1] = 0x%x, buf = %s\n", a[0], a[1], buf);
}
```

```
080485d0 <foo>:
80485d0: 55                   pushq   %rbp
80485d1: 48 89 e5             movq    %rsp,%rbp
80485d4: 48 83 ec 10          subq    $0x10,%rsp
80485d8: 53                   pushq   %rbx
80485d9: 8b 45 08             movl    0x8(%ebp),%eax
80485dc: 89 45 f8             movl    %eax,0xfffffff4(%ebp)
80485df: c7 45 f4 f3 f2 f1 f0 movl    $0xbfffff2d,0xfffffff8(%ebp)
80485e6: 8d 5d f0             leal    0xfffffff0(%ebp),%ebx
80485e9: 53                   pushl   %ebx
80485ea: e8 b7 fe ff ff       callq   80484a4 <_init+0x54> # gets
80485ef: 53                   pushq   %rbx
80485f0: 8b 45 f8             movl    0xfffffff8(%ebp),%eax
80485f3: 50                   pushl   %eax
80485f4: 8b 45 f4             movl    0xfffffff4(%ebp),%eax
80485f7: 50                   pushl   %eax
80485f8: 68 ec 90 04 08       pushl   $0x80490ec
80485fd: e8 94 fe ff ff       callq   8048494 <_init+0x44> # printf
8048602: 8b 5d ec             movl    0xffffffec(%ebp),%ebx
8048605: 48 89 ec             movq    %rbp,%rsp
8048608: 5d                   popq    %rbp
8048609: c3                   ret
804860a: 90                   nop
```

(1) Fill in the following table with the locations of the program values. Express them as decimal offsets (positive or negative) relative to register `%rbp`. (1 for each)

Program Value	Decimal Offset
`a`	-12
`a[2]`	-4
`x`	8
`buf`	-16
`buf[3]`	-13
Saved value of register `%rbx`	-24

(2) Consider the case where procedure `foo` is called with argument `x` as equal to `0xA1A2A3A4`, and we type "`0123456789`" when `gets` is invoked. Fill in the following table indicating whether each of the program values is corrupted or not by calling `gets`. (2 for each)

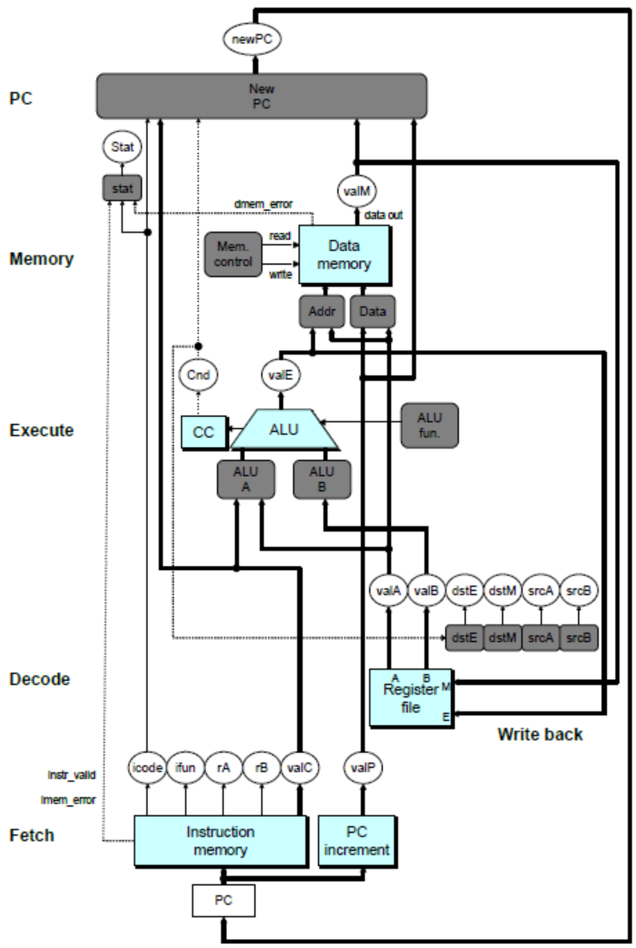
Program Value	Corrupted? (Y/N)
`a[0]`	Y
`a[1]`	Y
`a[2]`	N
`x`	N
Saved value of register `%rbp`	N
Saved value of register `%rbx`	N

(3) Assuming the same input as in (2) what will be the program output? (4 for each)

A. `a[0]` (hexadecimal): __________0x37363534__________
B. `a[1]` (hexadecimal): __________0xBF003938__________
C. `buf` (ASCII): ____________0123456789_____________

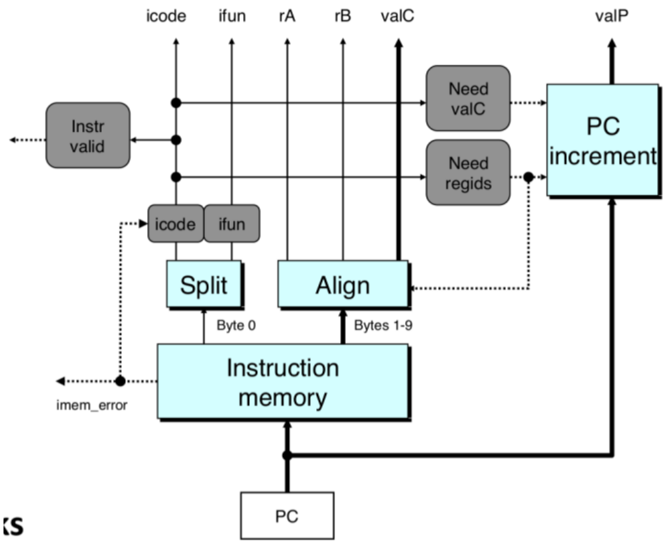
Part F: Y86-64 SEQ implementation (40 points)

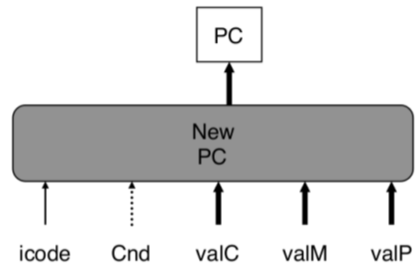
The following figure shows the overall structure of Y86-64 sequential (SEQ) implementation.



Question 8 (18 points)

The following figure shows the fetch and PC update stage of the Y86-64 SEQ implementation.





(1) Write down an HCL code for the signal `need_regids`.

```
bool need_regids =
Icode in {IRRMOVQ, IIRMOVQ, IMRMOVQ,
IRMMOVQ,
IOPQ, IPUSHQ, IPOPQ }
;
```

(1) Write down an HCL code for the signal `need_valC`.

```
bool need_valC =
Icode in { IIRMOVQ, IMRMOVQ, IRMMOVQ,
IJXX, ICALL }
;
```

(3) Write down an HCL code for the signal `new_pc`.

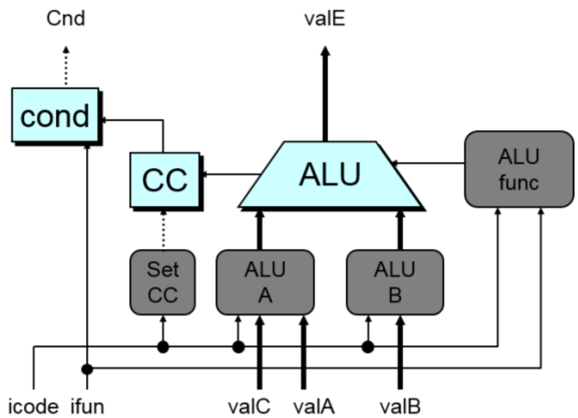
```
word new_pc =
[ icode == ICALL : valC;
icode == IJXX & Cnd : valC;
icode == IRET : valM;
1 : valP;]
;
```

Question 9 (22 points)

We would like to add `neg` instruction to the Y86-64 sequential implementation, which behaves just the same as the one in x86-64.

	icode:fn		rA:rB	
`neg rA`	`C = INEG`	`0`	`rA`	`F`

How should the control signals be modified in the Execute stage? Write down your code for the following four signals: `SetCC`, `ALUA`, `ALUB`, `ALUfunc`. We provide you with the original code for your reference.



<table>
<tr><th></th><th>Original code</th><th>Your code</th></tr>
<tr><td>ALU
A</td><td><pre>word aluA = [
  icode in {IRRMOVQ, IOPQ} : valA;
  icode in {IIRMOVQ, IRMMOVQ,
               IMRMOVQ} : valC;
  icode in {ICALL, IPUSHQ} : -8;
  icode in {IRET, IPOPQ} : 8;
];</pre></td><td><pre>word aluA = [
  icode in {IRRMOVQ, IOPQ, INEGQ} : valA;
  icode in {IIRMOVQ, IRMMOVQ,
               IMRMOVQ} : valC;
  icode in {ICALL, IPUSHQ} : -8;
  icode in {IRET, IPOPQ} : 8;
];</pre></td></tr>
<tr><td>ALU
B</td><td><pre>word aluB = [
  icode in {IRMMOVQ, IMRMOVQ,
              IOPQ, ICALL, IPUSHQ,
              IRET, IPOPQ} : valB;
  icode in {IRRMOVQ, IIRMOVQ} : 0;
];</pre></td><td><pre>word aluB = [
  icode in {IRMMOVQ, IMRMOVQ,
              IOPQ, ICALL, IPUSHQ,
              IRET, IPOPQ} : valB;
  icode in {IRRMOVQ, IIRMOVQ, INEGQ} : 0;
];</pre></td></tr>
<tr><td>ALU
func</td><td><pre>word alufun = [
  icode == IOPQ : ifun;
  1 : ALUADD;
];</pre></td><td><pre>word alufun = [
  icode == IOPQ : ifun;
  icode == INEGQ : ALUSUB;
  1 : ALUADD;
];</pre></td></tr>
</table>

Set CC	`bool set_cc = icode in {IOPQ};`	`bool set_cc = icode in {IOPQ, INEGQ};`

Appendix A: X86-64 assembly

Common instructions

`mov` `src, dst`	dst = src
`movsbl` `src, dst`	byte to int, sign-extend
`movzbl` `src, dst`	byte to int, zero-fill
`lea` `addr, dst`	dst = addr
`add` `src, dst`	dst += src
`sub` `src, dst`	dst -= src
`imul` `src, dst`	dst *= src
`neg` `dst`	dst = -dst (arith inverse)
`sal` `count, dst`	dst <<= count
`sar` `count, dst`	dst >>= count (arith shift)
`shr` `count, dst`	dst >>= count (logical shift)
`and` `src, dst`	dst &= src
`or` `src, dst`	dst |= src
`xor` `src, dst`	dst ^= src
`not` `dst`	dst = ~dst (bitwise inverse)
`cmp` `a, b`	b-a, set flags
`test` `a, b`	a&b, set flags
`jmp` `label`	jump to label (unconditional)
`je` `label`	jump equal ZF=1
`jne` `label`	jump not equal ZF=0
`js` `label`	jump negative SF=1
`jns` `label`	jump not negative SF=0
`jg` `label`	jump > (signed) ZF=0 and SF=OF
`jge` `label`	jump >= (signed) SF=OF
`jl` `label`	jump < (signed) SF!=OF
`jle` `label`	jump <= (signed) ZF=1 or SF!=OF
`ja` `label`	jump > (unsigned) CF=0 and ZF=0
`jb` `label`	jump < (unsigned) CF=1
`push` `src`	add to top of stack Mem[--%rsp] = src
`pop` `dst`	remove top from stack dst = Mem[%rsp++]
`call` `fn`	push %rip, jmp to fn
`ret`	pop %rip

Instruction suffixes

`b` byte
`w` word (2 bytes)
`l` long/doubleword (4 bytes)
`q` quadword (8 bytes)

Condition flags

ZF	Zero flag
SF	Sign flag
CF	Carry flag
OF	Overflow flag

Suffix is elided when can be inferred from operands
e.g. operand `%rax` implies `q`, `%eax` implies `l`, and so on

IEEE 754 FLOATING-POINT STANDARD

$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

where Single Precision Bias = 127,
Double Precision Bias = 1023.

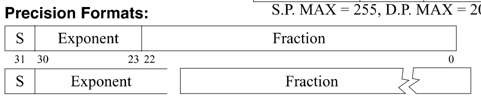
4

IEEE 754 Symbols

Exponent	Fraction	Object
0	0	± 0
0	≠0	± Denorm
1 to MAX - 1	anything	± Fl. Pt. Num.
MAX	0	±∞
MAX	≠0	NaN

S.P. MAX = 255, D.P. MAX = 2047

IEEE Single Precision and Double Precision Formats:



Registers

`%rip`	Instruction pointer
`%rsp`	Stack pointer
`%rax`	Return value
`%rdi`	1st argument
`%rsi`	2nd argument
`%rdx`	3rd argument
`%rcx`	4th argument
`%r8`	5th argument
`%r9`	6th argument
`%r10,%r11`	Caller-saved
`%rbx,%rbp,` **`%r12…%15`**	Callee-saved

Addressing modes

Example source operands to **`mov`**

Immediate

`mov $0x5, dst`
`$val`
source is constant value

Register

`mov %rax, dst`
`%R`
R is register
source in %R register

Direct

`mov 0x4033d0, dst`
`0xaddr`
source read from Mem[0xaddr]

Indirect

`mov (%rax), dst`
`(%R)`
R is register
source read from Mem[%R]

Indirect displacement

`mov 8(%rax), dst`
`D(%R)`
R is register
D is displacement
source read from Mem[%R + D]

Indirect scaled-index

`mov 8(%rsp, %rcx, 4), dst`
`D(%RB,%RI,S)`
RB is register for base
RI is register for index (0 if empty)
D is displacement (0 if empty)
S is scale 1, 2, 4 or 8 (1 if empty)
source read from
Mem[%RB + D + S*%RI]

** Originally from Stanford CS107; modified for SNU CSE 4190.308*

Appendix B: Y86-64 (Instruction Set)

Instruction icode:fn rA:rB

Instruction	byte 0		1		2 3 4 5 6 7 8 9
`halt`	0 = IHALT	0			
`nop`	1 = INOP	0			
`cmovXX rA, rB`	2 = IRRMOVQ	fn			
`rrmovq`		0			
`cmovle`		1			
`cmovl`		2			
`cmove`		3			
`cmovne`		4			
`cmovge`		5			
`cmovg`		6			
`irmovq V, rB`	3 = IIRMOVQ	0	F	rB	V
`rmmovq rA, D(rB)`	4 = IRMMOVQ	0	rA	rB	D
`mrmovq D(rB), rA`	5 = IMRMOVQ	0	rA	rB	D
`OPq rA, rB`	6 = IOPQ	fn	rA	rB	
`addq`		0			
`subq`		1			
`andq`		2			
`xorq`		3			
`jXX Dest`	7 = IJXX	fn	Dest		
`jmp`		0			
`jle`		1			
`jl`		2			
`je`		3			
`jne`		4			
`jge`		5			
`jg`		6			
`call Dest`	8 = ICALL	0	Dest		
`ret`	9 = IRET	0			
`pushq rA`	A = IPUSHQ	0	rA	F	
`popq rA`	B = IPOPQ	0	rA	F	

Register encoding

0	1	2	3	4	5	6	7
%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
8	9	A	B	C	D	E	F
%r8	%r9	%r10	%r11	%r12	%r13	%r14	No register