

4190.308: Computer Architecture
Midterm Exam
November 4th, 2016
Professor Jae W. Lee

Student ID #: _____

Name: _____

This is a closed book, closed notes exam.

120 Minutes

14 Pages

(+ 2 Appendix Pages)

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 indicate whether each of the following statements is true or false. You don't have to justify your answer—Just write down true or false.

- (1) According to the technology trends, the capacity of DRAM devices has been scaling up much faster than the speed (latency) of them.

ANSWER:

- (2) To compare two IEEE 754 floating-point numbers (except for $\pm\text{Infinity}$, and NaN), you can simply interpret them as two sign-magnitude integers and perform an integer comparison to obtain the correct result.

ANSWER:

- (3) When performing multiple floating-point additions, the order of additions does not affect the final result since addition is commutative.

ANSWER:

- (4) Unlike integers, the difference between a pair of two adjacent floating-point numbers is non-uniform.

ANSWER:

- (5) CISC architectures (e.g., x86-64) generally have an advantage in code size over RISC architectures (e.g., MIPS, ARM).

ANSWER:

Part B: Floating-Point Numbers (20 points)

Question 2 (20 points)

Consider the following 6-bit floating-point representation based on the IEEE 754 floating point format. The most significant bit represents a sign bit. The next three bits are the exponent, with an exponent bias of 3. The last two 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 (2 bit)
-----------------	----------------------	---------------------

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

Number	Decimal Representation	Binary Representation
Positive Zero	+0.0	
Negative Zero	-0.0	
0.75_{10}	0.75 (3/4)	
0.125_{10}	0.125 (1/8)	
One	1.0	
Positive Infinity	$+\infty$	
Negative Infinity	$-\infty$	
Not-a-Number	NaN	
The largest number		
The smallest positive number		

(2) Show all the possible non-zero values that are represented in the *denormalized* form.

Part C: Human x86-64 CPU (26 points)

Question 3 (12 points)

Ben Bitdiddle wrote the following C code, compiled it to x86-64 binary using gcc, and ran it. What is the program output? (*Hint*: Think about what the generated assembly code will look like.)

```
#include <stdio.h>

int main()
{
    int x = 1, y;

    if (x == 0 && x--) y = 1;
    else                y = 0;

    printf("x = %d, y = %d\n", x, y); // 1st printf

    if (x == 1 && x--) y = 1;
    else                y = 0;

    printf("x = %d, y = %d\n", x, y); // 2nd printf

    if (x == -1 && x--) y = 1;
    else                y = 0;

    printf("x = %d, y = %d\n", x, y); // 3rd printf
}
```

Question 4 (14 points)

Alice Hacker wrote the following C code to run it on x86-64/Linux system. What will be the program output? Fill in each blank with a correct value.

```
#include <stdio.h>
union {
    int i;
    short s[2];
    unsigned char c[4];
} u;

int main()
{
    int s0, s1;

    u.i = 0xbadbabe;
    s0 = (int) u.s[0];
    s1 = (int) u.s[1];

    printf("sizeof(int)=%d, sizeof(short)=%d, sizeof(char)=%d\n",
           sizeof(int), sizeof(short), sizeof(char));
    printf("sizeof(u.i)=%d\n", sizeof(u.i));
    printf("sizeof(u.s)=%d, sizeof(u.s[0])=%d\n", sizeof(u.s), sizeof(u.s[0]));
    printf("sizeof(u.c)=%d, sizeof(u.c[0])=%d\n", sizeof(u.c), sizeof(u.c[0]));
    printf("sizeof(u)=%d\n", sizeof(u));
    printf("s0=0x%x, s1=0x%x\n", s0, s1);
    printf("u.c=0x%x 0x%x 0x%x 0x%x\n", u.c[0], u.c[1], u.c[2], u.c[3]);
}
```

sizeof(int)=4, sizeof(short)=2, sizeof(char)=1

sizeof(u.i)=____

sizeof(u.s)=____, sizeof(u.s[0])=____

sizeof(u.c)=____, sizeof(u.c[0])=____

sizeof(u)=____

s0=0x____, s1=0x____

u.c=0x____ 0x____ 0x____ 0x____

Part D: Human x86-64 Compiler (38 points)

Question 5 (18 points)

The following code shows an array of a simple structure. Assume an x86-64/Linux system.

```
struct {
    int i;
    double d[2];
    char c;
    short s;
} st[2];
```

- (1) If the address of `st[0]` is `0x1000`, what is each element's address (in hexadecimal format)? Fill in the table below.

Element	Address
<code>int i</code>	
<code>double d[0]</code>	
<code>double d[1]</code>	
<code>char c</code>	
<code>short s</code>	
<code>st[1]</code>	

- (2) Redefine the structure to have the smallest size. How many bytes are saved for this array by this optimization?

Question 6 (20 points)

Consider the following assembly code for a **for** loop in C:

```

loop:
    push %ebp
    mov  %esp,%ebp
    mov  %edi,%ecx
    mov  %esi,%edx
    xor  %eax,%eax
    cmp  %edx,%ecx
    jle  .L4
.L6:
    dec  %ecx
    inc  %edx
    inc  %eax
    cmp  %edx,%ecx
    jg   .L6
.L4:
    inc  %eax
    mov  %ebp,%esp
    pop  %ebp
    ret

```

Please de-compile this code. In other words, fill in the original C code below using the assembly code. (Note: you may only use the symbolic variable names `x`, `y`, and `result` in your code — *do not use register names!*)

```

int loop(int x, int y)
{
    int result;

    for ( _____; _____ ; result++ )
    {
        _____;
        _____;
    }
    _____;

    return result;
}

```


Part E: Procedure Calls (32 points)

Question 7 (32 points)

Here is a C program which prints the n -th term of the Fibonacci sequence. C function `fibonacci()` in the left is compiled to x86-64 assembly in the right with an x86-64/Linux GCC compiler. Answer the following questions.

<pre>#include <stdio.h> int fibonacci(int n) { if (n == 0) return 0; else if (n == 1) return 1; return fibonacci(n-1) + fibonacci(n-2); } int main() { int n; printf("n: "); scanf("%d", &n); printf("%d\n", fibonacci(n)); ② return 0; }</pre>	<pre>fibonacci: 0x400614 pushq %rbp 0x400615 movq %rsp, %rbp 0x400618 pushq %rbx 0x400619 subq \$24, %rsp 0x40061d movl %edi, -20(%rbp) 0x400620 cmpl \$0, -20(%rbp) 0x400624 jne 0x40062d 0x400626 movl \$0, %eax 0x40062b jmp 0x400658 0x40062d cmpl \$1, -20(%rbp) 0x400631 jne 0x40063a 0x400633 movl \$1, %eax 0x400638 jmp 0x400658 0x40063a movl -20(%rbp), %eax 0x40063d subl \$1, %eax 0x400640 movl %eax, %edi 0x400642 call 0x400614 0x400647 movl %eax, %ebx 0x400649 movl -20(%rbp), %eax 0x40064c subl \$2, %eax 0x40064f movl %eax, %edi 0x400651 call 0x400614 ① 0x400656 addl %ebx, %eax 0x400658 addq \$24, %rsp 0x40065c popq %rbx 0x40065d popq %rbp 0x40065f retq</pre>
--	---

(1) What is the total number of instructions executed if $n = 2$?

- (2) Assuming $n = 5$, what are the values of `%ebx`, `%eax`, and `%rip` just before ① is executed for the first time?
- (3) What will the stack snapshot look like at the program execution point in Question (2)? Fill in the empty table below. Use “???” for an unknown value.

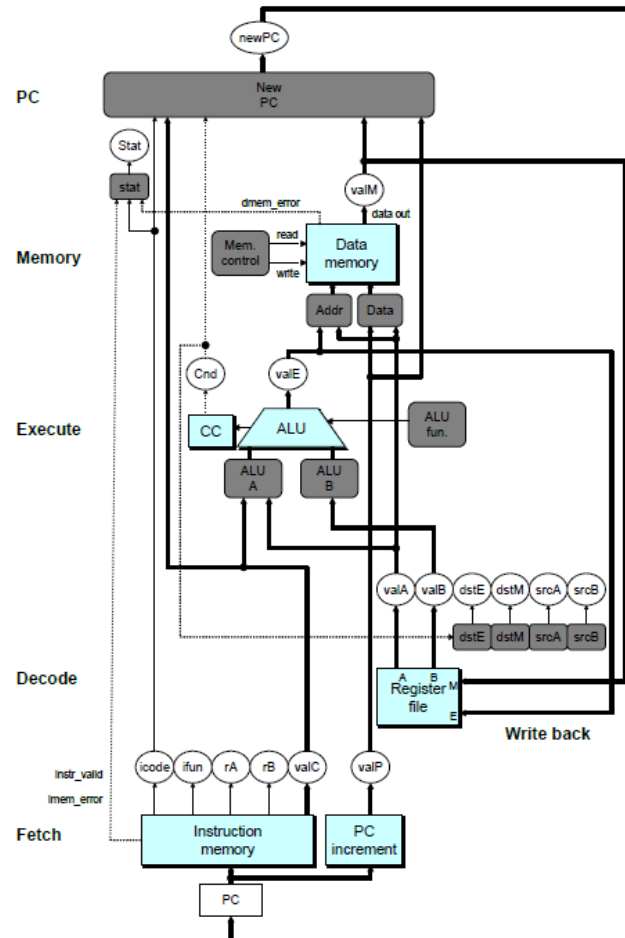
Hints:

- A. `%rsp` and `%rbp` hold `0x7fffffff360` and `0x7fffffff380`, respectively.
- B. The return address to `main` is `0x4005f2` (i.e. after all `fibonacci()` is done).
- C. Right before ②, both `%rbp` and `%rbx` hold `0x0`.

Stack Address	Value	
	Bytes 7~4	Bytes 3~0
0x7fffffff418	0x00000000	0x004005f2
0x7fffffff410	0x00000000	0x00000000
0x7fffffff408	0x00000000	0x00000000
0x7fffffff400		
0x7fffffff3f8		
0x7fffffff3f0		
0x7fffffff3e8		
0x7fffffff3e0		
0x7fffffff3d8		
0x7fffffff3d0		
0x7fffffff3c8		
0x7fffffff3c0		
0x7fffffff3b8		
0x7fffffff3b0		
0x7fffffff3a8		
0x7fffffff3a0		
0x7fffffff398		
0x7fffffff390		
0x7fffffff388	0x00000000	0x00400647
0x7fffffff380	0x00007fff	0xffffe3b0
0x7fffffff378	0x00000000	0x00000000
0x7fffffff370	???	???
0x7fffffff368		
0x7fffffff360	???	???

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

Here is an overall structure of Y86-64 sequential implementation.



Question 8 (10 points)

Using Y86-64 instruction encoding (in Appendix), fill in the boxes below.

(Note: You may or may not need all 10 bytes (boxes) for Question (2).)

(1) byte 0 1 →

Disassemble

(2) jne 0x277 →

byte 0 1 2 3 4 5 6 7 8 9

Assemble

Question 9 (20 points)

Please fill the following computation table of the Y86-64 SEQ implementation for pushq instruction. We already filled the fetch stage for you as an example. Use the following variables ONLY: valA, valB, valC, valE, valM, valP, Register value, and Memory value.

pushq rA

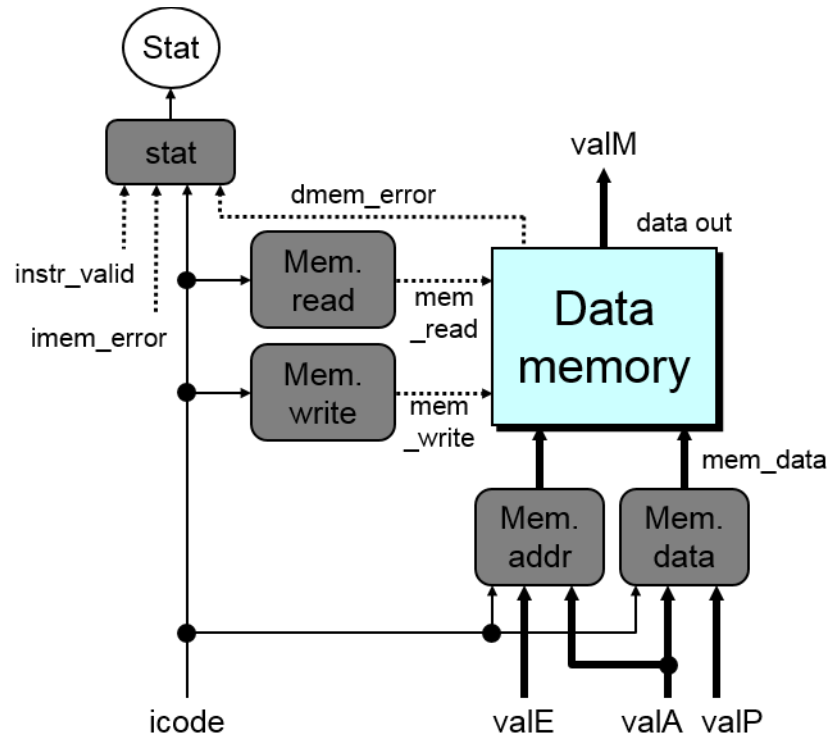
A	:	0
rA	:	F

(Notes – Use the following notations: Concatenation: “:”
 Assignment: “←”
 Register value: “R[*registerName*]”
 Memory value: “M_{size}[*memoryAddress*]”)

Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC}+2$
Decode	
Execute	
Memory	
Write back	
PC update	

Question 10 (14 points)

The following figure shows the memory stage of the Y86-64 SEQ implementation.



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

```
bool mem_write = 
```

```
;
```

(2) Write down an HCL code for the signal `mem_data`.

```
word mem_data = 
```

```
;
```

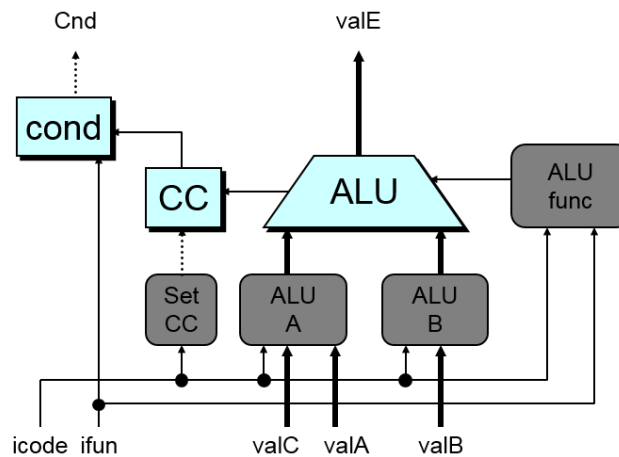
Question 11 (20 points)

We'd like to add `test` instruction to the Y86-64 sequential implementation;

`test rA, rB`

icode:fn	rA:rB
C = ITEST	0 rA rB

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.



	Original code	Your code
ALU A	<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>	
ALU B	<pre>word aluB = [icode in {IRMMOVQ, IMRMOVQ, IOPQ, ICALL, PUSHQ, IRET, IPOPQ} : valB; icode in {IRRMovQ, IIRMOVQ} : 0;];</pre>	
ALU func	<pre>word alufun = [icode == IOPQ : ifun; 1 : ALUADD;];</pre>	
Set CC	<pre>bool set_cc = icode in {IOPQ};</pre>	

Appendix A: Y86-64 (Instruction Set)

Instruction	icode:fn	rA:rB
	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
mrmmovq 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

Appendix B: 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	src, 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 flag
test	a, b	a & b, set flag
jmp	label	jump to label(unconditional)
je	label	ZF equal/zero
jne	label	~ZF not equal/zero
js	label	SF negative
jns	label	~SF nonnegative
jg	label	~(SF^OF)&~ZF greater(signed)
jge	label	~(SF^OF) greater or equal(signed)
jl	label	(SF^OF) less(signed)
jle	label	(SF^OF) ZF less or equal(signed)
ja	label	~CF&~ZF above(unsigned)
jb	label	CF below(unsigned)
push	src	add to top of stack Mem[--%rsp] = src
pop	dst	remove top from stack dst = Mem[%rsp++]
call	fn	push %rip, jump to fn
ret		pop %rip

Instruction suffixes

b	byte
w	word; 2 bytes
l	double word; 4 bytes
q	quad word; 8 bytes

Suffix is elided when can be inferred from operands. e.g. %rax implies q, %eax implies l.

Condition codes / flags

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

Registers

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

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

Direct: `mov (%rax), dst`
source read from Mem[%R]

Indirect displacement:
`mov 8(%rax), dst`

D(%R), D is displacement
source read from Mem[%R+D]

Indirect scaled-index:
`mov 8(%rsp,%rcx,4), dst`

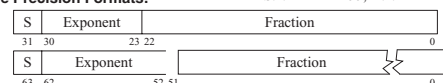
D(%RB, %RI, S)
source read from Mem[%RB+D+%RI*S]

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.

IEEE Single Precision and Double Precision Formats:



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