4190.308: Computer Architecture
Midterm Exam
November 1st, 2018
Professor Jae W. Lee
SOLUTIONS

Student ID #: ____________________

Name: ____________________

This is a closed book, closed notes exam.
80 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 (16 points)

Question 1 (16 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) Today's CPUs primarily focus on reducing clock cycle time to improve performance. (True/False)

ANSWER: FALSE

(2) RISC architectures (e.g., MIPS, ARM) generally have an advantage in clock frequency over CISC architectures (e.g., x86-64). (True/False)

ANSWER: TRUE

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

ANSWER: TRUE

(4) It is required for a *callee* function to restore the original values of *caller-saved* registers before return to the caller function. (True/False)

ANSWER: FALSE

Part B: Integer and Floating-point C Puzzles (24 points)

Question 2 (24 points)

We generate random values for x, y, and z, and convert them to other types on x86-64/Linux/

```
/* Generate random values */
int x = random();  // 4-byte signed
int y = random();
int z = random();

/* Cast to other types */
unsigned ux = (unsigned) x;  // 4-byte unsigned
unsigned uy = (unsigned) y;

double dx = (double) x;  // double-precision floating point
double dy = (double) y;
double dz = (double) z;
```

For each of the following expressions mark True if it *always* holds; if not, mark False. For every correct answer, you will get 3 points; for every wrong one, you will lose 3 points (and 0 points for no answer).

Expression	True or False?
`dx*dx>=0`	T F
`(x<y)==(-x>-y)`	T F
`((x+y)<<4)+y-x==17*y+15*x`	T F
`ux-uy==-(y-x)`	T F
`(double)(float)x == dx`	T F
`dx+dy==(double)(x+y)`	T F
`dx+dy+dz==dz+dy+dx`	T F
`dx*dy*dz==dz*dy*dx`	T F

Part C: Floating-Point Numbers (24 points)

Question 3 (24 points)

To accelerate deep learning applications, Google's hardware team has introduced a new floating-point format, called *bfloat16*, which is a 16-bit floating-point representation based on the IEEE 754 standard. The most significant bit represents a sign bit. The next eight bits are the exponent with a bias of 127. The last seven bits are the fraction. The same rules of the IEEE standard apply (normalized, denormalized, representation of zero, infinity, and NaN).

Sign (1 bit)	Exponent (8 bits)	Fraction (7bit)

(1) Fill in the empty boxes in the following table. (2 point each)

Number	Decimal Representation	Binary Representation
Negative Zero	-0.0	1 00000000 0000000
121/8	15.125_{10}	0 10000010 1110010
Positive Infinity	$+\infty$	0 11111111 0000000
One	1	0 01111111 0000000
The smallest negative number	$-255 * 2^{120}$	1 11111110 1111111

(2) What is (a) the largest finite number and (b) maximum denormal number for *bfloat16*? (2 points each)

(a) $1.1111111_{(2)} * 2^{254-127} = (2^8-1) * 2^{-7} * 2^{127}$

(b) $0.1111111_{(2)} * 2^{1-127} = (2^7-1) * 2^{-7} * 2^{-126}$

(3) There is another 16-bit floating-point (FP) format called *half-precision FP*, which uses 1, 5, 10 bits for sign, exponent, and fraction, respectively. What is the benefit of *bfloat16* over *half-precision FP*? Explain briefly in a few sentences.
(Hint: Deep learning applications often use *float* type in C to represent data.)

(8 points)

Since both *bfloat16* and *float* types share the same exponent field, it is easy to convert from one format to the other. Thus, there are very little modifications to the existing single-precision FP hardware to support mixed-precision computation.

Part D: Human x86-64 Compiler (24 points)

Question 4 (24 points)

Ben Bitdiddle is writing an assembly code, `fib.s` of the original C code (`fib.c`). His code is currently incomplete as the for loop that computes the *n*-th Fibonacci number is missing. Fill in the missing loop section in `fib.s`. Assume the following register mapping: `x(%rax)`, `y(%rdx)`, `n(%rsi)`, and `i(%rdi)`. *Note that the answer should correct for all integer value n*.

(wrong register: per -2 points, infinity loop or no loop: 12 points)

```
/* fib.c */
int main () {
    int x = 0, y = 1, n = 5, z;
    for (int i = 0; i < n; i++) {
        z = x + y;   x = y;    y = z;
    }

    // remaining part (omitted)
    return 0;
}
```

```
# fib.s: x in %rax, y in %rdx, n in %rsi, and i in %rdi
.main
    pushq  %rbp
    movq   %rsp, %rbp  # initiate procedure
    movq   $0x0, %rax
    movq   $0x1, %rdx
    movq   $0x5, %rsi
    movq   $0x0, %rdi
    # for loop: calculate n-th Fibonacci number into z
for:
    xorq   %rcx, %rcx
loop1:
    movq   %rdx, %rcx
    addq   %rax, %rcx
    movq   %rdx, %rax
    movq   %rcx, %rdx
    cmpq   %rsi, %rdi
    incq   %rdi
    jne    loop1


    # remaining part (omitted)
    ...
    mov    $0x0, %rax
    ret
```

Part E: Human x86-64 De-compiler (42 points)

Question 5 (24 points)

The following assembly code shows the body of function `foobar()`. (2.4 points each, nearest)

```
foobar:
    xorq  %r9, %r9        # Initialize i
loop_i:
    movq  %rdi, %rax      # %rdi == arr, %rsi == n
    movq  %r9,  %rdx
loop_j:
    movq  (%rdi), %rcx
    movq  (%rax), %r8
    cmpq  %r8, %rcx
    jle   skip
    movq  %r8, (%rdi)
    movq  %rcx,(%rax)
skip:
     addq  $0x1, %rdx
     addq  $0x4, %rax
     cmpq  %rdx, %rsi
     jg    loop_j
     addq  $0x1,  %r9
     addq  $0x4, %rdi
     cmpq  %r9,  %rsi
     jne   loop_i
     retq
```

Alice Hacker has reconstructed C code from it. Fill in the blanks in the C code below.

```
void foobar(int * arr, int n)
{
    for (int i = 0; i < ___n___; i++) {

        for (int j = ___i___; j < ___n___; j++) {

            if (_arr[i]_ > _arr[j]_) {

                int temp = arr[i];

                arr[i] = arr[j];

                arr[j] = temp;
            }
        }
    }
}
```

Question 6 (18 points)

There are two data structures: array of structures (AoS) and structure of arrays (SoA).

`struct AoS {`
`   char   w;`
`   int    x;`
`   char   y;`
`   double z;`
`};`
	`struct SoA {`
	`   char   w[N];`
	`   int    x[N];`
	`   char   y[N];`
	`   double z[N];`
	`};`

These two structures are used in the `main` function below.

```
#include <stdio.h>
#include <string.h>

#define N 10

int main () {
    struct AoS cell1[N];
    struct SoA cell2;
    ...
    return 0;
}
```

(1) How many bytes are used for AoS and SoA structures? (Note that `N` is 10.)

(4 points each, similar answer: 2 points)

Aos: 240 (24 * 10), SoA: 144 (10+2 + 4*10 + 10+2 + 8*10)

(2) Change both structures to maximize space efficiency. How many bytes are saved for each structure by this change?

(structure: 6 points, 2 points each saved)

```
struct Aos { double z; int x; char w; char y;}
```

Aos: 80 saved, SoA: No saved

Part F: Procedure Calls (30 points)

Question 7 (30 points)

Here is a C program that counts the number of 1's in argument `n`. The assembly code of `popcount()` generated by x86-64/Linux gcc is shown in the right.

Answer the following questions.

```c
#include <stdio.h>

int popcount(int n){
  if(n!=0)
    return (n&0x1) + popcount(n>>1);
  else
    return 0;

}

int main(){
  unsigned int n;
  printf("n : ");
  scanf("%d",&n);
  printf("Number of 1's: %d\n",
           popcount(n)); ②

  return 0;
}
```

```
00000000004005f6 <popcount>:
4005f6:       push   %rbp
4005f7:       mov    %rsp,%rbp
4005fa:       push   %rbx
4005fb:       sub    $0x18,%rsp
4005ff:       mov    %edi,-0x14(%rbp)
400602:       cmpl   $0x0,-0x14(%rbp)
400606:       je 400620 <popcount+0x2a>
400608:       mov    -0x14(%rbp),%eax
40060b:       and    $0x1,%eax
40060e:       mov    %eax,%ebx
400610:       mov    -0x14(%rbp),%eax
400613:       sar    %eax
400615:       mov    %eax,%edi
400617:       callq  4005f6 <popcount> ①
40061c:       add    %ebx,%eax
40061e:       jmp    400625 <popcount+0x2f>
400620:       mov    $0x0,%eax
400625:       add    $0x18,%rsp
400629:       pop    %rbx
40062a:       pop    %rbp
40062b:       retq
```

(1) What is the total number of instructions executed if `n`=12 (`0x1100`)? (3 points)

92 (20 * 4 + 12)

(2) Assuming n=7, what are the values of `%ebx`, `%eax`, and `%rip` when ① is reached for the first time? (1 point each)

`%ebx = 1`

`%eax = 3`

`%rip = 400617`

(3) What will the stack snapshot look like when ① is reached *for the second time*? Fill in the empty table below. Use "???" for an unknown value.
(-3 point for each 2 wrong blanks. ex. if you have 5 blanks wrong, you get -6 points
ex. if you have 6 blanks wrong, you get -9 points)

Hints:

A. `%rsp` and `%rbp` hold `0x7fffffffe120` and `0x7fffffffe140`, respectively.

B. The return address to main() is `0x400672` (i.e. after all `popcount()` is done).

Stack Address	Value	
	Bytes 7~4	Bytes 3~0
0x7fffffffe178	0x00000000	0x00400672
0x7fffffffe170	0x00007fff	0xffffe190
0x7fffffffe168	0x00000000	0x00000000
0x7fffffffe160	???	???
0x7fffffffe158	0x00000007	???
0x7fffffffe150	???	???
0x7fffffffe148	0x00000000	0x0040061c
0x7fffffffe140	0x00007fff	0xffffe170
0x7fffffffe138	0x00000000	0x00000001
0x7fffffffe130	???	???
0x7fffffffe128	0x00000003	???
0x7fffffffe120	0x00000000	0x00000000

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

Question 8 (20 points)

(1) Using Y86-64 instruction encoding (in Appendix), fill in the box below.
(Note : You may or may not need all 10 bytes(boxes))
((a) 3 and (b) 2 points)

(a)

Byte	0	1	2	3	4	5	6	7	8	9
	50	15	f8	ff	ff	ff	ff	ff	ff	ff

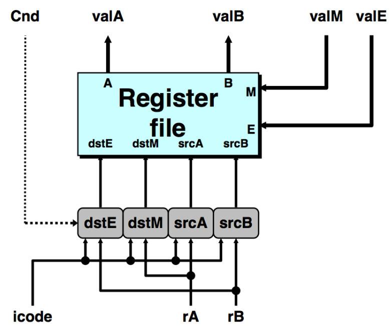
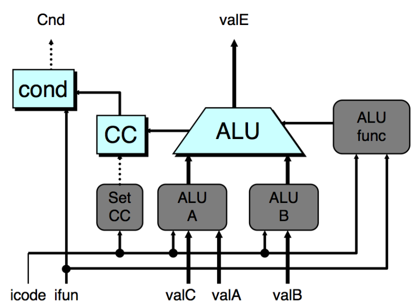
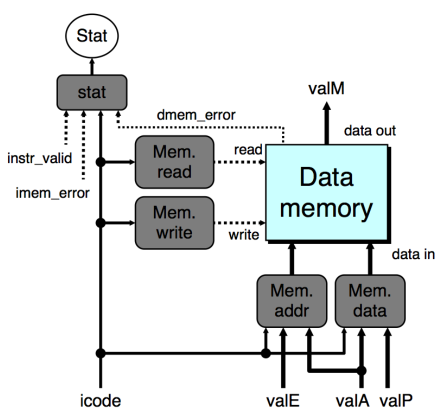
→ Disassemble → **`mrmovq -8(%rbp), %rcx`**

(b) `cmovl %rax, %rcx` → Assemble →

byte	0	1	2	3	4	5	6	7	8	9
	22	**01**								

(2) We provide a table of all constants used in HCL below. Complete the HCL code on the next page with correct constants for the Y86-64 SEQ implementation.
(5/7 point for each right answer -> round up to the nearest num.
We handled 1 oversubscribed answer as 1 wrong answer.
Ex. if you have t right answer and k oversubscribed answer, you have t-k right answer in total)

Name	Value (Hex)	Meaning
`INOP`	`0`	Code for `nop` instruction
`IHALT`	`1`	Code for `halt` instruction
`IRRMOVL`	`2`	Code for `rrmovl` instruction
`IIRMOVL`	`3`	Code for `irmovl` instruction
`IRMMOVL`	`4`	Code for `rmmovl` instruction
`IMRMOVL`	`5`	Code for `mrmovl` instruction
`IOPL`	`6`	Code for integer operation instructions
`IJXX`	`7`	Code for jump instructions
`ICALL`	`8`	Code for `call` instruction
`IRET`	`9`	Code for `ret` instruction
`IPUSHL`	`A`	Code for `pushl` instruction
`IPOPL`	`B`	Code for `popl` instruction
`FNONE`	`0`	Default function code
`RESP`	`4`	Register ID for **`%rsp`**
`RNONE`	`F`	Indicates no register file access
`ALUADD`	`0`	Function for addition operation
`SAOK`	`1`	Status code for normal operation
`SADR`	`2`	Status code for address exception
`SINS`	`3`	Status code for illegal instruction exception
`SHLT`	`4`	Status code for `halt`

Decode		int srcA = [icode in { **IRRMOVQ, IRMMOVQ,** **IOPQ, IPUSHQ** } : rA; icode in { **IPOPQ, IRET}:** RRSP; 1 : **RNONE**; # Don't need register];
Execute		int aluA = [icode in { IRRMOVQ, IOPQ } : valA; icode in { **IIRMOVQ, IRMMOVQ,** **IMRMOVQ** } : valC; icode in { **ICALL, IPUSHQ** } : -8; icode in { **IRET, IPOPQ** } : 8; # Other instructions don't need ALU]; int alufun = [icode == **IOPQ** : ifun; 1 : **ALUADD;**];
Memory		int Stat = [imem_error || dmem_error: **SADR;** !instr_valid: **SINS;** icode == IHALT : **SHLT;** 1 : **SAOK;**]; bool mem_read = icode in { **IMRMOVQ, IPOPQ, IRET**};

Question 9 (20 points)

We would like to add `addmem` instruction to the Y86-64 sequential implementation. Addmem instruction takes 2 operands, one from register and the other from the memory, and performs an addition. In other words, `addmem D(rB), rA` computes `rA<-rA+Mem[rB+D]`. Answer the following questions. (10 points each)

	icode:fn		rA:rB		D(8byte)
addmem rA,D(rB)	C = ADDMEM	0	rA	rB	D

(1) Describe the additional hardware (datapath) required to implement this instruction in one paragraph.

Two major changes:

- Include an extra adder to add the two operands after the M stage
- Register file takes an extra writeback path from this adder

(2) What is the impact of adding this instruction on the CPU cycle time? Would it be increased or unchanged? Again, explain in one paragraph.

This instruction builds on a load (`mrmove`) instruction, which is the critical path of the original Y86-64 SEQ design. Since it adds a delay of extra addition after the load, it is likely to increase the length of the critical path, and hence the CPU cycle time.

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

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

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

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

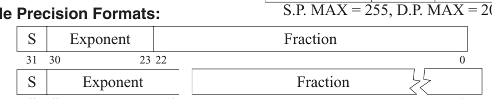
4

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 (byte 0)	fn (byte 0)	rA (byte 1)	rB (byte 1)	bytes 2–9
`halt`	0 = IHALT	0			
`nop`	1 = INOP	0			
`cmovXX rA, rB`	2 = IRRMOVQ	fn	rA	rB	
`rrmovq`		0			
`cmovle`		1			
`cmovl`		2			
`cmove`		3			
`cmovne`		4			
`cmovge`		5			
`cmovg`		6			
`irmovq V, rB`	3 = IIRMOVQ	0	F	rB	V (to byte 9)
`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 (bytes 1–8)		
`jmp`		0			
`jle`		1			
`jl`		2			
`je`		3			
`jne`		4			
`jge`		5			
`jg`		6			
`call Dest`	8 = ICALL	0	Dest (bytes 1–8)		
`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