4190.308: Computer Architecture
Final Exam
December 19th, 2017
Professor Jae W. Lee

Student ID #: _____

Name: _____

This is a closed book, closed notes exam.
120 Minutes
16 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 (24 points)

Question 1 (24 points)

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

Do not 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) Pipelining improves both the latency of an instruction and throughput. (True/False)

(2) For both DRAM and disks, access time (in second) has been scaling much more slowly than capacity (in $/MB). (True/False)

(3) Assuming the capacity and block size of a cache remain fixed, doubling the associativity doubles the number of tags in the cache. (True/False)

(4) Increasing cache block size helps exploiting spatial locality. (True/False)

(5) When you summarize performance metrics represented in ratio, *geometric mean* is used. (True/False)

(6) The performance of a GPU is not affected by the frequency of conditional branches as it runs many threads in parallel. (True/False)

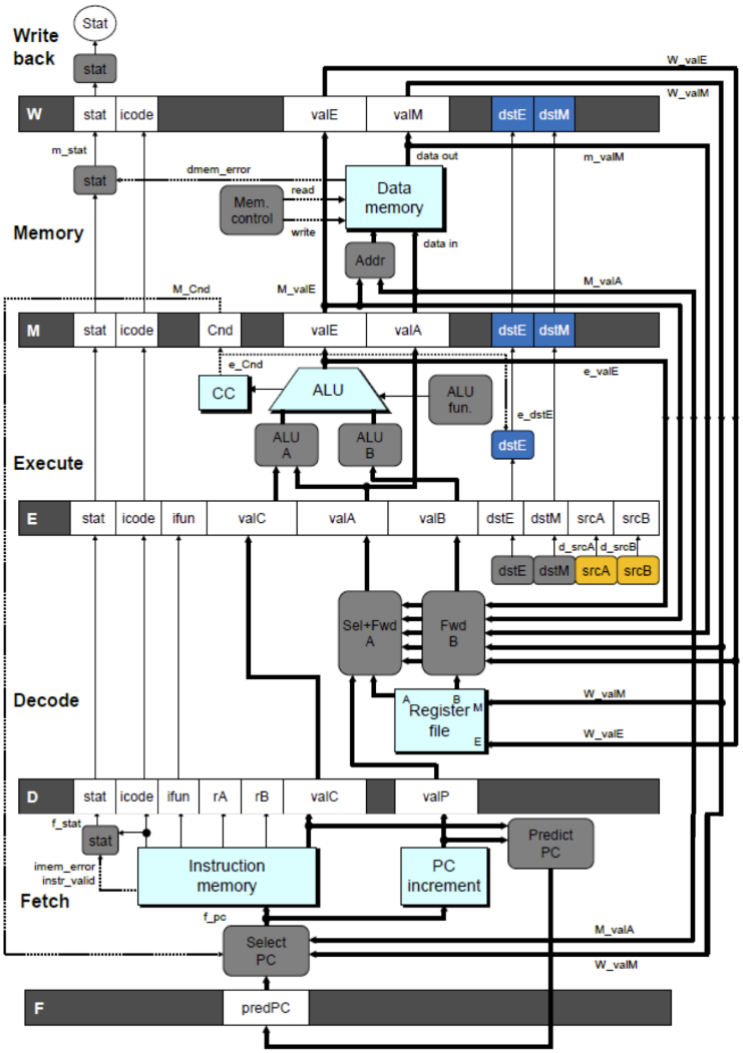
Part B: Pipelining (36 points)

Question 2 (36 points)

SNU Computers Corp. (SCC) has been selling Y86-64 compatible processors, called *s5*™ processor family. The figure below shows the pipeline of the *s5*™ processor family, which implements full forwarding logic.

The mid-range *s5*™ processor targets the PC market with the following features:

- Standard five stage (F, D, E, M, W) pipeline
- Forwarding logic for data and control hazard
- Predict that the branches are always taken (including `call`)



SCC's mid-range *s5*™ processor pipeline

(1) We will evaluate the following test code.

```
I1: addq     $1, %r9
I2: mrmovq   (%r9), %r10
I3: popq     %r10
I4: mrmovq   (%r10), %r9
I5: addq     %r10, %r9
```

Count how many cycles will be needed to execute the test code on *s5*™ by writing out each instruction's progress through the pipeline by filling in the table below with pipeline stages (F: Fetch, D: Decode, E: Execute, M: Memory, W: Write Back).

Cycle / Instr	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
I1	**F**	**D**	**E**	**M**	**W**										
I2															
I3															
I4															
I5															

(2) Again, we will evaluate another following test code, starting from I1.

```
I1: callq    F1
I2: addq     $1, %r9
```

```
F1: mrmovq    (%rsp), %r9
F2: rmmovq    %r9, (%rsp)
F3: ret
```

Count how many cycles will be needed to execute the test code on *s5*™ by filling in the table below.

Cycle / Instr	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
I1	**F**	**D**	**E**	**M**	**W**										
F1															
F2															
F3															
I2															

(3) Following are HCL codes of the four control signals of *s5*™ related to exception handling logic. Fill in the blanks below. You may use constants of SAOK, SHLT, SADR, SINS to indicate the type of exception.

```
int f_stat = [
    imem_error : ______;
    !instr_valid : ______;
    f_icode == IHALT : ______;
    1 : ______;
];
```

```
int m_stat = [
    dmem_error : ______;
    1 : ______;
];
```

```
int Stat = [   # final status determined at writeback stage
    # SBUB in earlier stages indicates bubble
    W_stat == SBUB : ______;
    1 : ______;
];
```

```
# Should the condition codes be updated?
bool set_cc = E_icode == IOPQ &&
     # State changes only during normal operation


     ____________________________________________________;
```

Part C: Caches (42 points)

Question 3 (18 points)

Let's assume the baseline cache design can be characterized by three parameters: (# of sets, Associativity, Block size) = (S_0, E_0, B_0). Ben Bitdiddle is exploring three strategies to double the cache size. For each of the following three strategies, identify what type(s) of cache misses (among the 3C's) are reduced and explain why briefly.

(1) (S, E, B) = ($2S_0$, E_0, B_0): Doubling the number of sets

(2) (S, E, B) = (S_0, $2E_0$, B_0): Doubling the associativity

(3) (S, E, B) = (S_0, E_0, $2B_0$): Doubling the block size

Question 4 (24 points)

In this question we want to calculate the cache miss rate for a given program analytically. Assume the following parameters for the cache (data cache only):

Parameter	Value
# of sets (S)	4
Cache block size (B)	64 bytes
Associativity (E)	1 (direct-mapped)
Size of `int` data	4 bytes

For each of the following three versions of the same program, calculate *the total number of cache misses* (not average miss rate). Here are assumptions:

- The data cache is initially empty.
- Accesses to the array `A[]` are the only memory accesses.
- The declaration of `A[N*N]` generates no memory access.
- The address of `A[0]` is aligned to a multiple of 32.

(1)

```
int N = 32;
int A[N*N];

for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
    int sum = 0;
      for (k = 0; k < N; k++)
        sum += i * j * k;
      A[i * N + j] = sum;
  }
}
```

(2)

```
int N = 32;
int A[N*N];

for (k = 0; k < N; k++) {
  for (i = 0; i < N; i++) {
    int r = i * k;
      for (j = 0; j < N; j++)
        A[i * N + j] += r * j;
  }
}
```

(3)

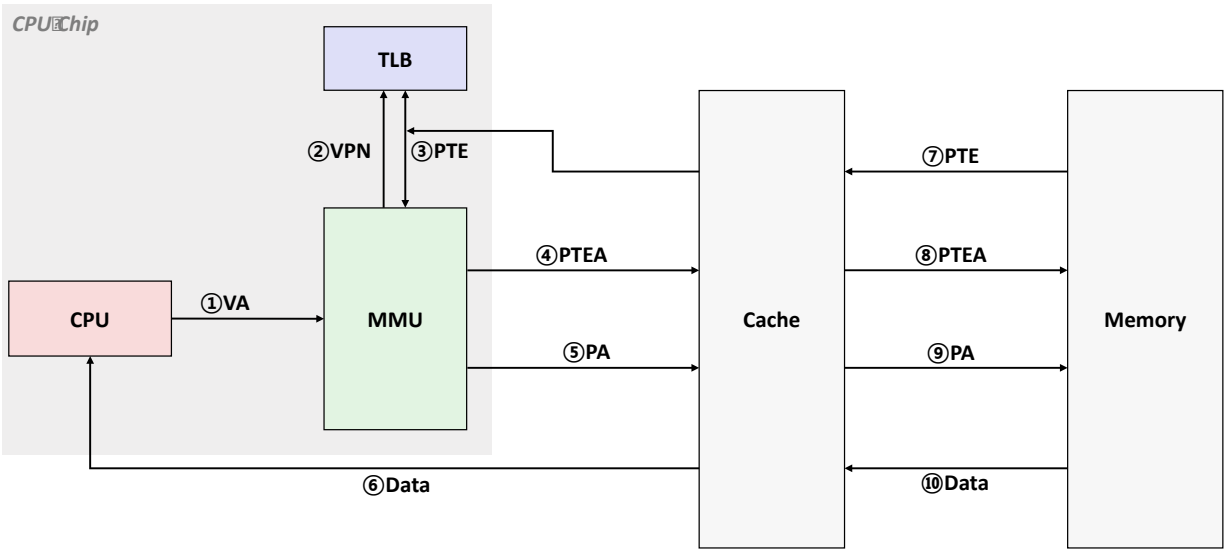
```
int N = 32;
int A[N*N];

for (j = 0; j < N; j++) {
  for (k = 0; k < N; k++) {
    int r = k * j;
      for (i = 0; i < N; i++)
        A[i * N + j] += i * r;
  }
}
```

Part D: Cache and Virtual Memory (64 points)

Question 5 (12 points)

The figure below shows the access path of a physically addressed cache integrated with virtual memory.



Express the order of an access by a sequence of numbers on the access path, assuming the following three conditions

1) TLB hit, Cache hit for PTE, Cache hit for requested instruction/data

   ① →

2) TLB Miss, Cache hit for PTE, Cache hit for requested instruction/data

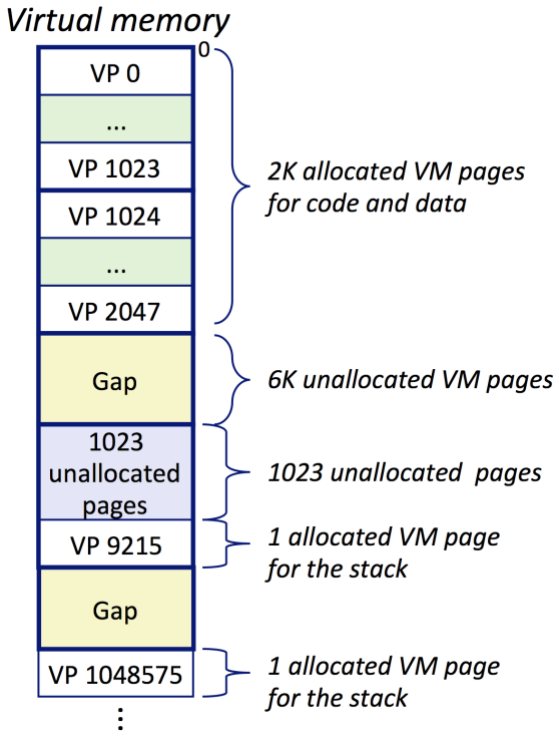
   ① →

3) TLB Miss, Cache miss for PTE, Cache miss for requested instruction/data

   ① →

Question 6 (16 points)

Let's assume that a process is using the virtual pages 0 ~ 2047 (`0x7FF`), 9215 (`0x23FF`), and 1048575 (`0xFFFFF`), as shown in the figure on the right. The address space is 32-bit, page size is 4KB, and a page table entry takes 4 bytes. Please answer the following questions.



1) What is the total size of a two-level page table assuming the L1 page table has 256 (=2^8) entries?

2) If a memory request misses at both TLB and the cache, how many memory references are necessary to fulfill this request? Explain briefly what each reference is for. Assume the PTE for the requested address is valid.

Question 7 (36 points)

For the rest of this question, please assume the following

- Page size = 4096 bytes
- 20-bit virtual address
- 16-bit physical address
- 4-way set associative TLB with 16 entries
- 2-way set associative, physically addressed cache with 64 bytes/block

We ask you to follow step-by-step operations of a 2-way physically addressed cache.

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	B	1	07	6	0	28	3	1	01	F	0
1	31	0	1	12	3	0	07	E	1	0B	1	1
2	2A	A	0	11	1	1	1F	8	1	07	5	1
3	07	3	1	3F	F	0	10	D	1	32	0	0

Initial TLB state (16 entries)

Index	Tag	Valid	Tag	Valid
0	5A	1	-	0
1	11	1	-	0
2	3A	1	-	0
3	0E	1	-	0
4	63	1	-	0
5	35	1	-	0
6	70	1	-	0
7	45	1	-	0

Initial cache state (16 entries, only tag and valid bits are shown)

(1) Assume we access the cache with virtual address `0xC5D70`. Please fill out the blank in **hexadecimal number**.
(*Note*: You should write '`X`' if the value would be not specified.)

VPN: _____ PPN: _____

TLBI: _____ CO (cache offset): _____

TLBT: _____ CI (cache index): _____

TLB hit? (Y/N) _____ CT (cache tag): _____

Cache hit? (Y/N) _____

(2) Assuming the Least Recently Used (LRU) replacement policy, what will be the final cache state after accessing the following address sequence? Fill out the table below with the new cache state.

Address sequence (all in *virtual* address)**:**

`0x0C40C` → `0x7E040` → `0x46CC4` → `0x7EBD8`
→ `0x1FBF4` → `0x1D180` → `0x2D1A0` → `0xA0400`

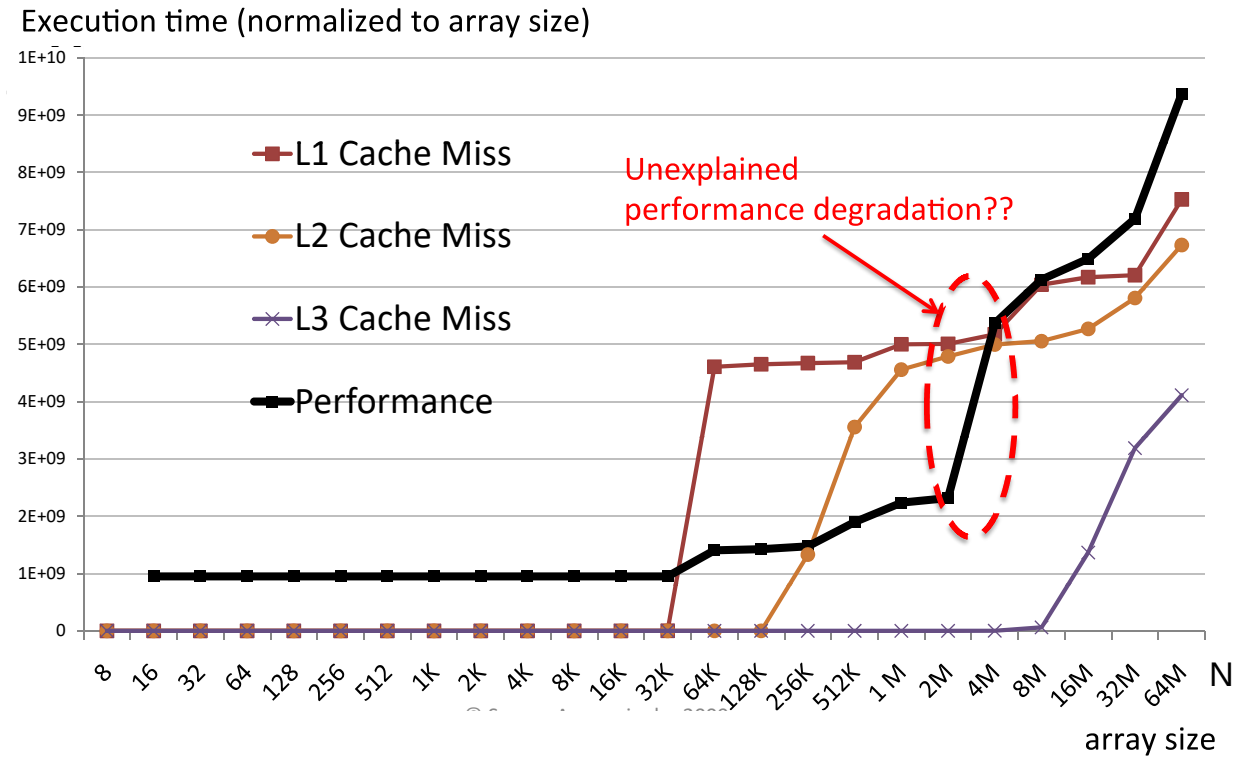
Index	Tag	Valid	Tag	Valid
0				
1				
2				
3				
4				
5				
6				
7				

(3) What will be the cache hit rate for (2)?

Part E: Performance (34 points)

Question 8 (14 points)

Alice Hacker ran a loop with array accesses as she varied the size of the array, and the figure below shows the results. She observed execution time per array element was increased significantly whenever the array failed to fit in L1/L2/L3 caches, respectively. However, she found an unexplained performance discontinuity when the array size was between 2M and 4M (as circled below). Can you explain why in one paragraph?



Question 9 (20 points)

You are the chief architect of *sPhone™ 11*, your company's next-generation smartphone. The CPU of *sPhone™ 11* will adopt a cool new feature, called *dynamic frequency scaling (DFS)*, where the CPU can select one of the two operating modes with different frequency settings: high-performance mode and low-power mode. In this way the smartphone can maximize battery life while maintaining good performance. You decided to characterize the two operation modes using a killer messaging app, titled *Papaya Talk™*. The two operation modes are assumed to have the same CPIs and instruction counts but different clock rates (and operating voltages), and the performance of the app is analyzed as follows:

Instruction Type	Instruction count (millions)	Clock cycles per instr (CPI)	Clock rate (GHz)	
			High Perf	Low Power
Arithmetic & Logic	10	2	1.6	1.0
Load & Store	8	6		
Branch	5	4		
Miscellaneous	2	4		

(1) What is the average CPI for this app?

(2) What are the CPU times of this app in the two operating modes?
(Note: Be sure to include time units.)

(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

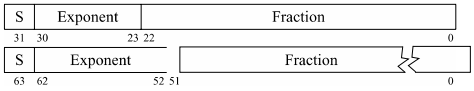
Suffix is elided when can be inferred from operands
e.g. operand `%rax` implies q, `%eax` implies `l`, and so on

IEEE 754 FLOATING-POINT STANDARD

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

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

IEEE Single Precision and Double Precision Formats:



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

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

	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					9
rmmovq rA, D(rB)	4 = IRMMOVQ	0	rA	rB	D					9
mrmovq D(rB), rA	5 = IMRMOVQ	0	rA	rB	D					9
OPq rA, rB	6 = IOPQ	fn	rA	rB						
addq		0								
subq		1								
andq		2								
xorq		3								
jXX Dest	7 = IJXX	fn	Dest						8	
jmp		0								
jle		1								
jl		2								
je		3								
jne		4								
jge		5								
jg		6								
call Dest	8 = ICALL	0	Dest						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