4190.308-002: Computer Architecture
Final Exam
December 18th, 2019
Professor Jae W. Lee

Solutions

Student ID #: _____

Name: _____

This is a closed book, closed notes exam.
120 Minutes
xxx Pages
(+ 4 Appendix Pages)

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 30 minutes before 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.

- "**RISC-V Reference sheet**" is provided at the end of this exam; use it as you need.

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

Part A: Short Answers (24 points)

Question 1 (24 points)

아래 단답형 문제에 답하시오. 답을 설명할 필요는 없으며, 정답은 각각 4점 부여, (1)-(4)번 문제의 경우 오답은 4점 감점함.

(1) 파이프라인 프로세서 precise exception을 구현할 때, 여러 명령어가 동시에 exception을 발생시킬 경우 프로그램 순서상 가장 이른 (oldest) 명령어가 가장 높은 우선순위를 갖는다. (True/False)

TRUE

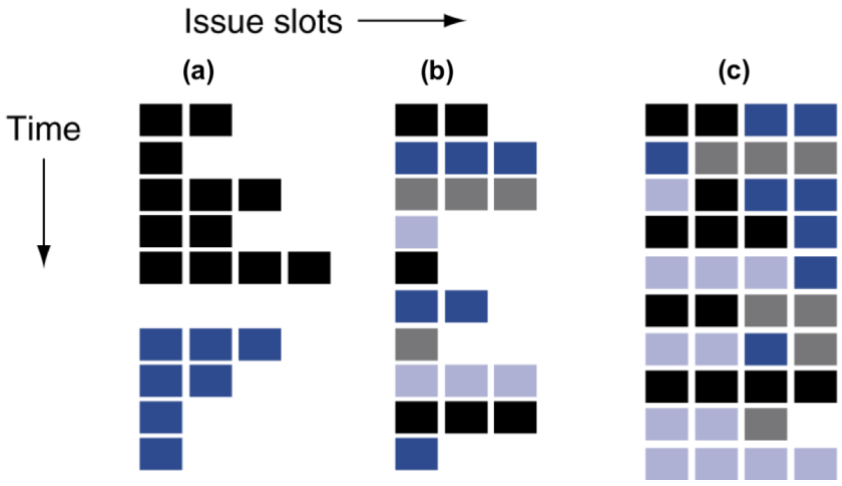
(2) Invalidation 기반 cache coherence protocol에서는 store 명령을 실행할 때, 먼저 그 block에 대한 exclusive access를 확보한다. (True/False)

TRUE

(3) Vector 아키텍처는 SIMD Extension(예: AVX-256등)과 달리 variable length vector를 지원한다. (True/False)

TRUE

(4) 아래의 그림은 3개의 다른 multi-threading 기법의 명령어 스케줄을 비교한다. 여기서 4개의 쓰레드의 명령어는 각기 다른 색깔로 표현되어 있다. (a)-(c) 각각의 기법을 무엇이라고 부르는가?



(5) (8 points) GPU 아키텍처의 특징을 간략히 요약하시오. (3 bullet 내외)

A lot of simple cores, Shared instruction stream among fragments (work items) – SIMT 모델, Hiding memory stalls by zero-latency context switching

Part B: Instruction-Level Parallelism (32 points)

Question 2 (32 points)

정훈은 성능이 불만족스러운 아래의 C코드를 어셈블리 수준에서 최적화하려고 한다.

```
long *laxpy(long *X, long *Y, long a, size_t length)
{
    for(size_t i = 0 ; i < length ; i++)
    {
        Y[i] = a * X[i] + Y[i];
    }
    return Y;
}
```

주어진 하드웨어의 사양은 다음과 같다.

- Static dual-issue RISC-V CPU
- 모든 명령어의 수행 시간은 1 사이클
- Load-use hazard를 해결하는데 추가적으로 1 사이클(bubble)이 필요

(1) 아래 어셈블리 코드는 위의 코드를 컴파일해서 얻은 결과이다. Loop body(#Loop start … end) 1 iteration의 스케줄로 표를 채우고, cycles per iteration을 구하시오. Loop iteration간 overlap은 없다고 가정한다. (필요시 부록 B의 표를 사용해도 무방함.)

```
#a0 = X, a1 = Y, a2 = a, a3 =length
AXPY:
xor t0, t0, t0    # use t0 as i
beq t0, a2, OUT

LOOP:             # Loop start
ld t1, 0(a0)      # t1 = X[i]
ld t2, 0(a1)      # t2 = Y[i]
mul t1, t1, a2    # t1 = a * X[i]
add t1, t2, t1    # t1 = Y[i] + a * X[i]
sd t1, 0(a1)      # Y[i] = t1

addi t0, t0, 1    # i += 1
addi a0, a0, 8    # X += 1
addi a1, a1, 8    # Y += 1

bne t0, a3, LOOP  # Loop end
OUT:
# Return to caller
```

	ALU/Branch	Load/Store	Cycle
LOOP:		ld t1, 0(a0)	1
		ld t2, 0(a1)	2
	mul t1, t1, a2		3
	add t1, t2, t1		4
	addi t0, t0, 1	sd t1, 0(a1)	5
	addi a0, a0, 8		6
	addi a1, a1, 8		7
	bne t0, a3, LOOP		8
			9
			10

Cycles per iteration = 8 사이클

(2) 위의 루프를 Unrolling factor=2로 loop unrolling만을 적용한 코드를 작성하고, cycles per iteration을 구하시오. 원래의 register allocation을 유지해야하며, register renaming은 불허한다.

```
# a0 = X, a1 = Y, a2 = a, a3 =length
AXPY:
xor t0, t0, t0    # use t0 as i
beq t0, a2, OUT

LOOP:             # Loop start

ld t1, 0(a0)
ld t2, 0(a1)
mul t1, t1, a2
add t1, t2, t1
sd t1, 0(a1)

ld t1, 8(a0)
ld t2, 8(a1)
mul t1, t1, a2
add t1, t2, t1
sd t1, 8(a1)

addi t0, t0, 2
addi a0, a0, 16
addi a1, a1, 16


bne t0, a3, LOOP  # Loop end
```

```
OUT:
# Return to caller.
```

	ALU/Branch	Load/Store	Cycle
LOOP:		ld t1, 0(a0)	1
		ld t2, 0(a1)	2
	mul t1, t1, a2		3
	add t1, t2, t1		4
		sd t1, 0(a1)	5
		ld t1, 8(a0)	6
		ld t2, 8(a1)	7
	mul t1, t1, a2		8
	add t1, t2, t1		9
	addi t0, t0, 2	sd t1, 8(a1)	10
	addi a0, a0, 16		11
	addi a1, a1, 16		12
	bne t0, a3, LOOP		13
			14
			15
			16

Cycles per iteration: 13 cycles/2 iterations = 6.5

(3) 2번의 코드에 Register renaming 기법을 적용하여 코드를 업데이트하고, cycles per iteration을 새로 구하시오 (Register renaming 이외의 기법은 허용되지 않음. 필요하다면 부록 B를 활용하시오).

```
#a0 = X, a1 = Y, a2 = a, a3 =length
AXPY:
xor t0, t0, t0    #Index

beq t0, a2, OUT
LOOP:
ld t1, 0(a0)
ld t2, 0(a1)

mul t1, t1, a2
ld t3, 8(a0)

add t1, t2, t1
```

```
ld t4, 8(a1)

mul t3, t3, a2
sd t1, 0(a1)

add t3, t4, t3
sd t3, 8(a1)

addi t0, t0, 2
addi a0, a0, 16
addi a1, a1, 16









bne, t0, a3, LOOP
OUT:
#Return to caller.
```

	ALU/Branch	Load/Store	Cycle
LOOP:		ld t1, 0(a0)	1
		ld t2, 0(a1)	2
	mul t1, t1, a2	ld t3, 8(a0)	3
	add t1, t2, t1	ld t4, 8(a1)	4
	mul t3, t3, a2	sd t1, 0(a1)	5
	add t3, t4, t3		6
	addi t0, t0, 2	sd t3, 8(a1)	7
	addi a0, a0, 16		8
	addi a1, a1, 16		9
	bne t0, a3, LOOP		10
			11
			12
			13
			14

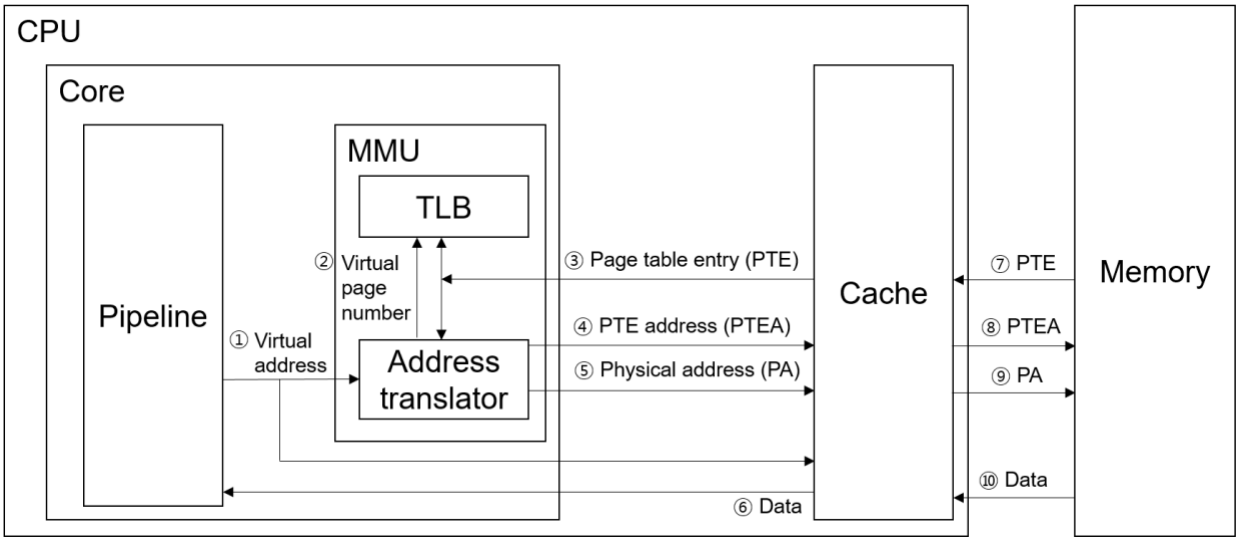
			15
			16

Cycles per iteration: 10 cycles/2 iterations = 5

Part C: Caches (44 points)

Question 3 (12 points)

다음 그림은 Virtually-addressed, physically-tagged 캐쉬 구조를 나타낸 것이다.



다음 세 가지 각각의 경우에 대해 접근 순서를 나열하시오.

(1) TLB hit, cache hit

1 -> 2 -> 3 -> 5 -> 6

(2) TLB miss, cache hit (for all memory accesses)

1 -> 2 -> 4 -> 3 -> 5 -> 6

(3) TLB miss, cache miss (for all memory accesses)

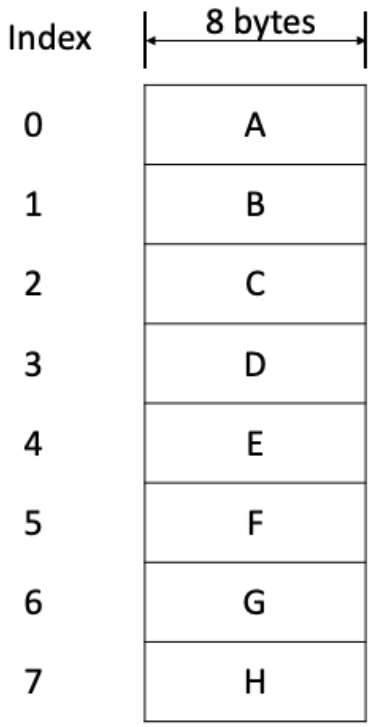
1 -> 2 -> 4 -> 8 -> 7 -> 3 -> 5 -> 9 -> 10 -> 6

Question 4 (32 points)

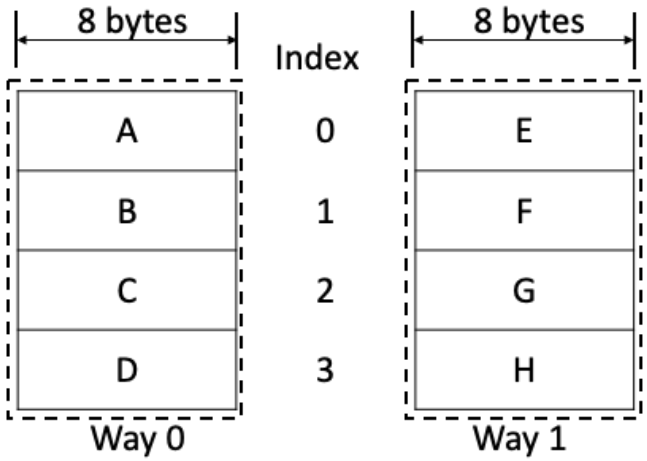
다음의 캐쉬 및 페이지 크기를 가정하시오.

- Cache block size = 8 bytes / block
- Cache size = 64 bytes (8 blocks)
- Page size = 16 bytes

(1) Virtually addressed, physically addressed 캐쉬의 동작을 두 개의 다른 구조에서 비교하고자 한다(direct-mapped cache, 2-way set-associative cache). 0x34라는 virtual (byte) address를 cache에 mapping할 때, 이 block이 할당 될 수 있는 모든 cache block을 나열하시오. 이를 위해, 아래 주어진 표를 채우시오. (9점)



(A) Direct-mapped Cache



(B) 2-way Set-associative Cache

기본: 1점
한 칸당 2점

	Virtually addressed	Physically addressed
Direct-mapped (A)	G	A, C, E, G
2-way Set-associative (B)	C, G	A, C, E, G

(2) 다음은 (1)번에 나와있는 virtually addressed physically tagged, 2-way set-associative cache 의 작동 방법에 대해 알아본다. Cache와 TLB의 initial snapshot은 아래와 같다. Replacement policy 는 Least Recently Used (LRU)를 가정하시오. (Cache에는 valid bit (V)와 tag만, TLB에는 PPN과 VPN만 표기하였다.) (16점)

Index	V	Tags (way0)	V	Tags (way1)
0	**1**	0x45	**0**	
1	**1**	0x3D	**0**	
2	**1**	0x1D	**0**	
3	**0**		**0**	

Initial cache tag states (8 blocks)

VPN	PPN	VPN	PPN
0x0	0x0A	0x10	0x6A
0x1	0x1A	0x20	0x7A
0x2	0x2A	0x30	0x8A
0x3	0x3A	0x40	0x9A
0x5	0x4A	0x50	0xAA
0x7	0x5A	0x70	0xBA

TLB states (12 entries)

다음과 같은 address sequence가 주어졌을 때 캐쉬의 마지막 상태는 어떻게 되는가? 아래 표를 채우시오. 주소는 hexadecimal로 표시 하시오.

Address sequence: 0x34 -> 0x38 -> 0x50 -> 0x54 -> 0x208 -> 0x20C -> 0x74 -> 0x54

Index	V	Tags (way0)	V	Tags (way1)
0	**1**	0x45	**0**	
1	**1**	0x3D	**0**	
2	**1**	0x2D	**1**	0x25
3	**1**	0x1D	**0**	

각 칸 당 2점

(3) Cache hit은 1 cycle, cache miss 는 16 cycle이 걸린다고 할 때, (2)에서의 8개 word의 address sequence에 대한 average memory access time(AMAT)은 얼마인가? (7점)

hit/miss 개수: 1점씩,
cycle: 5점

0x34 (hit: index 2)
-> 0x38 (miss: index 3)
-> 0x50 (miss: index 2)
-> 0x54 (hit: index 2)
-> 0x208 (hit: index 1)
-> 0x20C (hit: index 1)
-> 0x74 (miss: index 2)
-> 0x54 (hit: index 2)

Hit은 5개, Miss는 3개
Average memory access time = 1 + 3 / 8 * 16 = 7 cycles

Part D: Dependability (20 points)

Question 5 (20 points)

아래 그림은 8-bit data를 protect 하기 위한 12-bit Hamming SEC(Single-Error Correction) 코드의 인코딩을 보여준다.

Bit position	1	2	3	4	5	6	7	8	9	10	11	12
Encoded data bits	p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8

Even parity를 가정하고, 아래 물음에 답하시오.

(1) Parity bit p4가 커버하는 모든 code bit(d1-d8, p1-p4)을 나열하시오.

p4, d2, d3, d4, d8

(2) 읽어낸 12-bit 코드의 값이 0x345라고 하자. 이 코드가 전송하는 8-bit 데이터 값은 얼마인가? (d1이 MSB, d8이 LSB라 가정) 필요하면 error correction을 적용하시오.

0x345 = 0011 0100 0101

Group p1 = Odd parity (error)
Group p2 = Odd parity (error)
Group p4 = Odd parity (error)
Group p8 = Even parity (no error)

따라서, p8p4p2p1=0111, 즉 7번 bit(d4)에서 error가 발생하였다. 이를 수정하면,

0011 0110 0101 = 0x365

여기서 d1-d8 비트열을 추출하면 1011 0101이 되므로, 원래 8-bit 데이터는 **0xb5**.

Part E: Virtual Memory (32 points)

Question 6 (32 points)

최신 CPU 아키텍처는 TLB miss를 줄이기 위해 multiple page size를 지원한다. 정훈은 주어진 CPU에서 huge page와 regular page를 사용하였을 때, 성능 차이를 대해 평가하고자 한다. 평가하려는 CPU의 사양은 다음과 같다.

- 36-bit virtual address space, 32-bit physical address space
- 8-byte Page Table Entry (PTE)
- 1-level page table covering the entire virtual address space
- 4KB regular page, 4MB huge page
- Data-TLB (D-TLB) holds 64 PTEs with FIFO replacement policy.
- No consideration for Instruction-TLB (I-TLB).

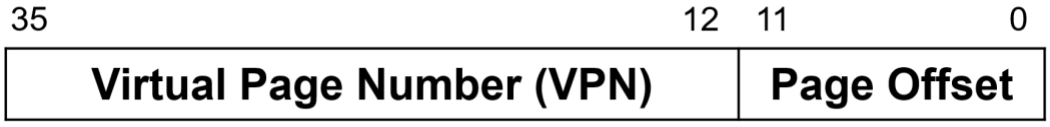
성능을 평가하기 위해 사용하는 프로그램은 다음과 같다.

```
char A[1048576]; // 1MB array
char B[1048576]; // 1MB array
char C[1048576]; // 1MB array

for(int i=0; i<1048576; i++)
  C[i] = A[i] + B[i];
```

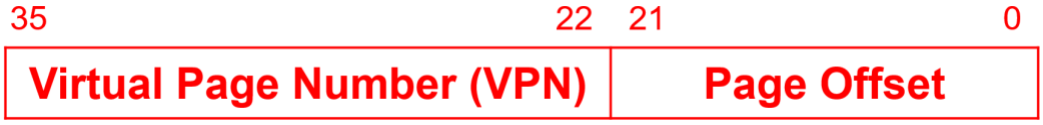
A, B, C는 physical memory상의 연속된 공간에 존재하는 것으로 가정한다. 즉, 다음의 두 가지 virtual-to-physical address mapping을 비교한다.

- 4KB page: A, B, C는 768개의 4KB page를 사용
- 4MB page: A, B, C는 1개의 4MB page를 사용

(1) 다음은 4KB 페이지에 대해 virtual address breakdown을 보여준다.



같은 방식으로 4MB 페이지에 대한 virtual address breakdown을 보이시오.



(2) Page Table Overhead (PTO) 는 다음과 같이 정의한다.

$$PTO = \frac{Physical\ memory\ that\ is\ allocated\ to\ page\ tables}{Physical\ memory\ that\ is\ allocated\ to\ data\ pages}$$

위의 프로그램에서 4MB 페이지를 사용할 때와 4KB 페이지를 사용할 때의 PTO를 각각 구하시오.

4KB: 42.3, 4MB: 1/32

(3) Page Fragmentation Overhead (PFO) 는 다음과 같이 정의한다.

$$PFO = \frac{Physical\ memory\ that\ is\ allocated\ to\ data\ pages\ but\ is\ never\ ac}{Physical\ memory\ that\ is\ allocated\ to\ data\ pages\ and\ is\ acces}$$

위의 프로그램에서 4MB 페이지를 사용할 때와 4KB 페이지를 사용할 때의 PFO를 각각 구하시오.

4KB: 0, 4MB: 1/3

(4) 주어진 프로그램에서 4MB 페이지를 사용할 때와 4KB 페이지를 사용할 때의 TLB miss 횟수를 구하시오.

4KB: 768, 4MB: 1

(5) 주어진 프로그램에서 4MB 페이지를 사용할 때와 4KB 페이지를 사용할 때의 성능 차이와 가장 가까운 것을 고르고, 그 이유에 대해 서술하시오.

a)1.01배 b) 10배 c) 1,000배 d) 1,000,000배

a) 메모리 접근 4K번에 TLB 미스 한번 정도 일어나므로 1.01배는 충분히 reasonable함

Part F: Parallel Architectures (48 points)

Question 7 (12 points)

윤호는 병렬 프로그램 X를 작성하였다. 이 프로그램 실행시간중 병렬로 처리되는 시간의 비율은 40%이다. (P = 0.4)

(1) 윤호는 컴퓨터구조 수업을 들었기 때문에, 프로그램 X의 병렬성은 매우 효율적이다. 즉, N개의 코어를 사용할 때, N배의 ideal speedup을 자랑한다. 이때, 2-코어와 16-코어에서 전체 프로그램의 speedup은 각각 얼마인가? (8점)

$$Speedup_{4-core} = \frac{1}{0.6 + \frac{0.4}{2}} = 1.25$$

$$Speedup_{16-core} = \frac{1}{0.6 + \frac{0.4}{16}} = 1.6$$

(2) 프로그램 X가 병렬처리를 통해 달성할 수 있는 speedup의 upper bound (Amdahl's Limit)는 얼마인가? (4점)

$$\frac{1}{0.6} = 1.67$$

Question 8 (16 points)

64명의 학생의 시험점수가 val[] array에 저장되어 있을 때, 최고득점(Top-1)을 찾는 병렬 프로그램을 작성하고자 한다. 다음 물음에 답하여라.

(1) 더 빠른 계산을 위해 64-프로세서 병렬처리를 도입한다. val[] array는 shared memory에 할당되어 있음을 가정하고, 실행을 $O(\log_2 N)$ 복잡도(N=64)로 마칠 수 있는 다음 병렬 코드를 완성하시오. 단, half, Pn은 각 프로세서의 local variable이며, Pn은 Processor ID(0-63) 값을 가지고 있다. (13점)

병렬 6점
나머지 2점씩

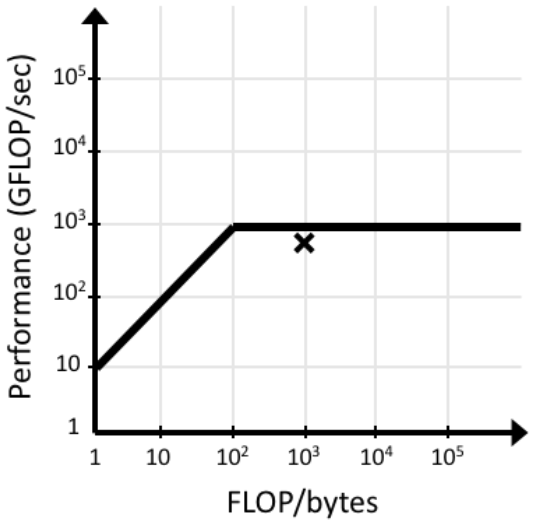
```
half = 64;
do
  synch();

  half = half/2; /* dividing line on who compares */
  if (Pn < half)
    if (val[Pn] < val[Pn+half]) val[Pn] = val[Pn+half];




while (half > 1);
```

(2) 만약 (1)번 코드에서 synch()함수의 역할은 무엇이며, 이 함수가 없을 경우 어떤 문제가 발생하는가? (3점)

synch()는 barrier sync primitive. half = 2의 0번째 index가 실행된 후에 half = 4의 뒤에 연산이 실행될 수도 있다.

Question 9 (20 points)

어느 회사에서 딥러닝 가속을 위한 프로세서인 NPUv1 (Neural Processing Unit version 1)에 대한 Roofline 분석을 아래와 같이 수행하였다. 해당 NPU에서 딥러닝 프로그램을 실행하였을 때, 그래프에서 X로 표시된 위치 (10^3FLOP/byes)에 위치함을 확인하고, 이를 바탕으로 새로운 NPUv2를 제작하려고 한다. 이때 NPUv1과 NPUv2의 스펙은 아래 표와 같이 가정하고, 물음에 답하시오.



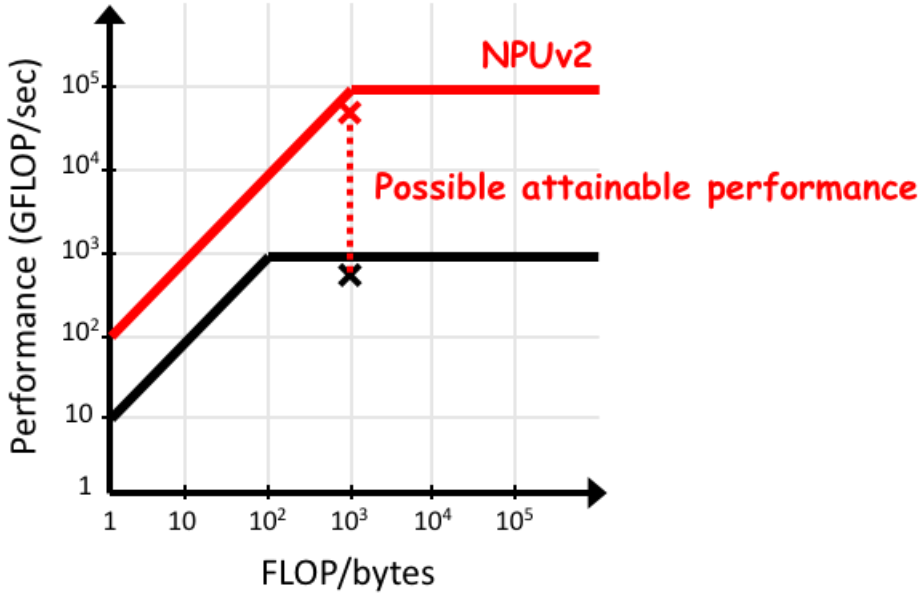
Roofline 그래프, 표시된 선은 NPUv1 의 Roofline을 나타냄.

	Frequency	# of PEs	DRAM bandwidth
NPUv1	500 MHz	2000	10 GB/s
NPUv2	2.5 GHz	?	100 GB/s

(1) 주어진 NPUv2 스펙에서 주어진 딥러닝 응용의 최적 성능을 달성하기 위해 몇개의 Processing Elements (PE)가 필요한가? NPUv2에서 모델이 사용하는 off-chip memory 접근 패턴이 NPUv1과 동일하고, 1개의 PE는 매 clock cycle 당 1개의 부동소수점 연산(FLOP)을 수행할 수 있음을 가정하시오. (10점)

100000 = 2.5 * 40000
따라서 40000개의 PEs 필점
답 틀리면 0점

(2) 위 그래프에 NPUv2의 Roofline 그래프를 포개어 그리시오.(10점)
선 당 5 점



RISC-V Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

RISC-V Reference Data ①

RV64I BASE INTEGER INSTRUCTIONS, in alphabetical order

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
add,addw	R	ADD (Word)	R[rd] = R[rs1] + R[rs2]	1)
addi,addiw	I	ADD Immediate (Word)	R[rd] = R[rs1] + imm	1)
and	R	AND	R[rd] = R[rs1] & R[rs2]	
andi	I	AND Immediate	R[rd] = R[rs1] & imm	
auipc	U	Add Upper Immediate to PC	R[rd] = PC + {imm, 12'b0}	
beq	SB	Branch EQual	if(R[rs1]==R[rs2) PC=PC+{imm,1b'0}	
bge	SB	Branch Greater than or Equal	if(R[rs1]>=R[rs2) PC=PC+{imm,1b'0}	
bgeu	SB	Branch ≥ Unsigned	if(R[rs1]>=R[rs2) PC=PC+{imm,1b'0}	2)
blt	SB	Branch Less Than	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	
bltu	SB	Branch Less Than Unsigned	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	2)
bne	SB	Branch Not Equal	if(R[rs1]!=R[rs2) PC=PC+{imm,1b'0}	
csrrc	I	Cont./Stat.RegRead&Clear	R[rd] = CSR;CSR = CSR & ~R[rs1]	
csrrci	I	Cont./Stat.RegRead&Clear Imm	R[rd] = CSR;CSR = CSR & ~imm	
csrrs	I	Cont./Stat.RegRead&Set	R[rd] = CSR; CSR = CSR | R[rs1]	
csrrsi	I	Cont./Stat.RegRead&Set Imm	R[rd] = CSR; CSR = CSR | imm	
csrrw	I	Cont./Stat.RegRead&Write	R[rd] = CSR; CSR = R[rs1]	
csrrwi	I	Cont./Stat.Reg Read&Write Imm	R[rd] = CSR; CSR = imm	
ebreak	I	Environment BREAK	Transfer control to debugger	
ecall	I	Environment CALL	Transfer control to operating system	
fence	I	Synch thread	Synchronizes threads	
fence.i	I	Synch Instr & Data	Synchronizes writes to instruction stream	
jal	UJ	Jump & Link	R[rd] = PC+4; PC = PC + {imm,1b'0}	
jalr	I	Jump & Link Register	R[rd] = PC+4; PC = R[rs1]+imm	3)
lb	I	Load Byte	R[rd] = {56'bM[](7),M[R[rs1]+imm](7:0)}	4)
lbu	I	Load Byte Unsigned	R[rd] = {56'b0,M[R[rs1]+imm](7:0)}	
ld	I	Load Doubleword	R[rd] = M[R[rs1]+imm](63:0)	
lh	I	Load Halfword	R[rd] = {48'bM[](15),M[R[rs1]+imm](15:0)}	4)
lhu	I	Load Halfword Unsigned	R[rd] = {48'b0,M[R[rs1]+imm](15:0)}	
lui	U	Load Upper Immediate	R[rd] = {32'b\<imm<31>, imm, 12'b0}	
lw	I	Load Word	R[rd] = {32'bM[](31),M[R[rs1]+imm](31:0)}	4)
lwu	I	Load Word Unsigned	R[rd] = {32'b0,M[R[rs1]+imm](31:0)}	
or	R	OR	R[rd] = R[rs1] | R[rs2]	
ori	I	OR Immediate	R[rd] = R[rs1] | imm	
sb	S	Store Byte	M[R[rs1]+imm](7:0) = R[rs2](7:0)	
sd	S	Store Doubleword	M[R[rs1]+imm](63:0) = R[rs2](63:0)	
sh	S	Store Halfword	M[R[rs1]+imm](15:0) = R[rs2](15:0)	
sll,sllw	R	Shift Left (Word)	R[rd] = R[rs1] << R[rs2]	1)
slli,slliw	I	Shift Left Immediate (Word)	R[rd] = R[rs1] << imm	1)
slt	R	Set Less Than	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	
slti	I	Set Less Than Immediate	R[rd] = (R[rs1] < imm) ? 1 : 0	
sltiu	I	Set < Immediate Unsigned	R[rd] = (R[rs1] < imm) ? 1 : 0	2)
sltu	R	Set Less Than Unsigned	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	2)
sra,sraw	R	Shift Right Arithmetic (Word)	R[rd] = R[rs1] >> R[rs2]	1,5)
srai,sraiw	I	Shift Right Arith Imm (Word)	R[rd] = R[rs1] >> imm	1,5)
srl,srlw	R	Shift Right (Word)	R[rd] = R[rs1] >> R[rs2]	1)
srli,srliw	I	Shift Right Immediate (Word)	R[rd] = R[rs1] >> imm	1)
sub,subw	R	SUBtract (Word)	R[rd] = R[rs1] – R[rs2]	1)
sw	S	Store Word	M[R[rs1]+imm](31:0) = R[rs2](31:0)	
xor	R	XOR	R[rd] = R[rs1] ^ R[rs2]	
xori	I	XOR Immediate	R[rd] = R[rs1] ^ imm	

Notes: 1) The Word version only operates on the rightmost 32 bits of a 64-bit registers
2) Operation assumes unsigned integers (instead of 2's complement)
3) The least significant bit of the branch address in jalr is set to 0
4) (signed) Load instructions extend the sign bit of data to fill the 64-bit register
5) Replicates the sign bit to fill in the leftmost bits of the result during right shift
6) Multiply with one operand signed and one unsigned
7) The Single version does a single-precision operation using the rightmost 32 bits of a 64-bit F register
8) Classify writes a 10-bit mask to show which properties are true (e.g., –inf, -0,+0, +inf, denorm, ...)
9) Atomic memory operation; nothing else can interpose itself between the read and the write of the memory location
The immediate field is sign-extended in RISC-V

ARITHMETIC CORE INSTRUCTION SET ②

RV64M Multiply Extension

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
mul,mulw	R	MULtiply (Word)	R[rd] = (R[rs1] * R[rs2])(63:0)	1)
mulh	R	MULtiply High	R[rd] = (R[rs1] * R[rs2])(127:64)	
mulhu	R	MULtiply High Unsigned	R[rd] = (R[rs1] * R[rs2])(127:64)	2)
mulhsu	R	MULtiply upper Half Sign/Uns	R[rd] = (R[rs1] * R[rs2])(127:64)	6)
div,divw	R	DIVide (Word)	R[rd] = (R[rs1] / R[rs2])	1)
divu	R	DIVide Unsigned	R[rd] = (R[rs1] / R[rs2])	2)
rem,remw	R	REMainder (Word)	R[rd] = (R[rs1] % R[rs2])	1)
remu,remuw	R	REMainder Unsigned (Word)	R[rd] = (R[rs1] % R[rs2])	1,2)

RV64F and RV64D Floating-Point Extensions

fld,flw	I	Load (Word)	F[rd] = M[R[rs1]+imm]	1)
fsd,fsw	S	Store (Word)	M[R[rs1]+imm] = F[rd]	1)
fadd.s,fadd.d	R	ADD	F[rd] = F[rs1] + F[rs2]	7)
fsub.s,fsub.d	R	SUBtract	F[rd] = F[rs1] – F[rs2]	7)
fmul.s,fmul.d	R	MULtiply	F[rd] = F[rs1] * F[rs2]	7)
fdiv.s,fdiv.d	R	DIVide	F[rd] = F[rs1] / F[rs2]	7)
fsqrt.s,fsqrt.d	R	SQuare RooT	F[rd] = sqrt(F[rs1])	7)
fmadd.s,fmadd.d	R	Multiply-ADD	F[rd] = F[rs1] * F[rs2] + F[rs3]	7)
fmsub.s,fmsub.d	R	Multiply-SUBtract	F[rd] = F[rs1] * F[rs2] - F[rs3]	7)
fnmadd.s,fnmadd.d	R	Negative Multiply-ADD	F[rd] = –(F[rs1] * F[rs2] + F[rs3])	7)
fnmsub.s,fnmsub.d	R	Negative Multiply-SUBtract	F[rd] = –(F[rs1] * F[rs2] – F[rs3])	7)
fsgnj.s,fsgnj.d	R	SiGN source	F[rd] = { F[rs2]<63>,F[rs1]<62:0>}	7)
fsgnjn.s,fsgnjn.d	R	Negative SiGN source	F[rd] = { (–F[rs2]<63>), F[rs1]<62:0>}	7)
fsgnjx.s,fsgnjx.d	R	Xor SiGN source	F[rd] = {F[rs2]<63>^F[rs1]<63>, F[rs1]<62:0>}	7)
fmin.s,fmin.d	R	MINimum	F[rd] = (F[rs1] < F[rs2]) ? F[rs1] : F[rs2]	7)
fmax.s,fmax.d	R	MAXimum	F[rd] = (F[rs1] > F[rs2]) ? F[rs1] : F[rs2]	7)
feq.s,feq.d	R	Compare Float EQual	R[rd] = (F[rs1]== F[rs2]) ? 1 : 0	7)
flt.s,flt.d	R	Compare Float Less Than	R[rd] = (F[rs1]< F[rs2]) ? 1 : 0	7)
fle.s,fle.d	R	Compare Float Less than or =	R[rd] = (F[rs1]<= F[rs2]) ? 1 : 0	7)
fclass.s,fclass.d	R	Classify Type	R[rd] = class(F[rs1])	7,8)
fmv.s.x,fmv.d.x	R	Move from Integer	F[rd] = R[rs1]	7)
fmv.x.s,fmv.x.d	R	Move to Integer	R[rd] = F[rs1]	7)
fcvt.s.d	R	Convert to SP from DP	F[rd] = single(F[rs1])	
fcvt.d.s	R	Convert to DP from SP	F[rd] = double(F[rs1])	
fcvt.s.w,fcvt.d.w	R	Convert from 32b Integer	F[rd] = float(R[rs1](31:0))	7)
fcvt.s.l,fcvt.d.l	R	Convert from 64b Integer	F[rd] = float(R[rs1](63:0))	7)
fcvt.s.wu,fcvt.d.wu	R	Convert from 32b Int Unsigned	F[rd] = float(R[rs1](31:0))	2,7)
fcvt.s.lu,fcvt.d.lu	R	Convert from 64b Int Unsigned	F[rd] = float(R[rs1](63:0))	2,7)
fcvt.w.s,fcvt.w.d	R	Convert to 32b Integer	R[rd](31:0) = integer(F[rs1])	7)
fcvt.l.s,fcvt.l.d	R	Convert to 64b Integer	R[rd](63:0) = integer(F[rs1])	7)
fcvt.wu.s,fcvt.wu.d	R	Convert to 32b Int Unsigned	R[rd](31:0) = integer(F[rs1])	2,7)
fcvt.lu.s,fcvt.lu.d	R	Convert to 64b Int Unsigned	R[rd](63:0) = integer(F[rs1])	2,7)

RV64A Atomtic Extension

amoadd.w,amoadd.d	R	ADD	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] + R[rs2]	9)
amoand.w,amoand.d	R	AND	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] & R[rs2]	9)
amomax.w,amomax.d	R	MAXimum	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amomaxu.w,amomaxu.d	R	MAXimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amomin.w,amomin.d	R	MINimum	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amominu.w,amominu.d	R	MINimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amoor.w,amoor.d	R	OR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] | R[rs2]	9)
amoswap.w,amoswap.d	R	SWAP	R[rd] = M[R[rs1]], M[R[rs1]] = R[rs2]	9)
amoxor.w,amoxor.d	R	XOR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] ^ R[rs2]	9)
lr.w,lr.d	R	Load Reserved	R[rd] = M[R[rs1]], reservation on M[R[rs1]]	
sc.w,sc.d	R	Store Conditional	if reserved, M[R[rs1]] = R[rs2], R[rd] = 0; else R[rd] = 1	

CORE INSTRUCTION FORMATS

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	Opcode
I	imm[11:0]			rs1	funct3	rd	Opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

Appendix A: RISC-V Reference Sheet (Page 2)

PSEUDO INSTRUCTIONS ③

MNEMONIC	NAME	DESCRIPTION	USES
beqz	Branch = zero	if(R[rs1]==0) PC=PC+{imm,1b'0}	beq
bnez	Branch ≠ zero	if(R[rs1]!=0) PC=PC+{imm,1b'0}	bne
fabs.s,fabs.d	Absolute Value	F[rd] = (F[rs1]< 0) ? –F[rs1] : F[rs1]	fsgnx
fmv.s,fmv.d	FP Move	F[rd] = F[rs1]	fsgnj
fneg.s,fneg.d	FP negate	F[rd] = –F[rs1]	fsgnjn
j	Jump	PC = {imm,1b'0}	jal
jr	Jump register	PC = R[rs1]	jalr
la	Load address	R[rd] = address	auipc
li	Load imm	R[rd] = imm	addi
mv	Move	R[rd] = R[rs1]	addi
neg	Negate	R[rd] = –R[rs1]	sub
nop	No operation	R[0] = R[0]	addi
not	Not	R[rd] = ~R[rs1]	xori
ret	Return	PC = R[1]	jalr
seqz	Set = zero	R[rd] = (R[rs1]== 0) ? 1 : 0	sltiu
snez	Set ≠ zero	R[rd] = (R[rs1]!= 0) ? 1 : 0	sltu

OPCODES IN NUMERICAL ORDER BY OPCODE

MNEMONIC	FMT	OPCODE	FUNCT3	FUNCT7 OR IMM	HEXADECIMAL
lb	I	0000011	000		03/0
lh	I	0000011	001		03/1
lw	I	0000011	010		03/2
ld	I	0000011	011		03/3
lbu	I	0000011	100		03/4
lhu	I	0000011	101		03/5
lwu	I	0000011	110		03/6
fence	I	0001111	000		0F/0
fence.i	I	0001111	001		0F/1
addi	I	0010011	000		13/0
slli	I	0010011	001	0000000	13/1/00
slti	I	0010011	010		13/2
sltiu	I	0010011	011		13/3
xori	I	0010011	100		13/4
srli	I	0010011	101	0000000	13/5/00
srai	I	0010011	101	0100000	13/5/20
ori	I	0010011	110		13/6
andi	I	0010011	111		13/7
auipc	U	0010111			17
addiw	I	0011011	000		1B/0
slliw	I	0011011	001	0000000	1B/1/00
srliw	I	0011011	101	0000000	1B/5/00
sraiw	I	0011011	101	0100000	1B/5/20
sb	S	0100011	000		23/0
sh	S	0100011	001		23/1
sw	S	0100011	010		23/2
sd	S	0100011	011		23/3
add	R	0110011	000	0000000	33/0/00
sub	R	0110011	000	0100000	33/0/20
sll	R	0110011	001	0000000	33/1/00
slt	R	0110011	010	0000000	33/2/00
sltu	R	0110011	011	0000000	33/3/00
xor	R	0110011	100	0000000	33/4/00
srl	R	0110011	101	0000000	33/5/00
sra	R	0110011	101	0100000	33/5/20
or	R	0110011	110	0000000	33/6/00
and	R	0110011	111	0000000	33/7/00
lui	U	0110111			37
addw	R	0111011	000	0000000	3B/0/00
subw	R	0111011	000	0100000	3B/0/20
sllw	R	0111011	001	0000000	3B/1/00
srlw	R	0111011	101	0000000	3B/5/00
sraw	R	0111011	101	0100000	3B/5/20
beq	SB	1100011	000		63/0
bne	SB	1100011	001		63/1
blt	SB	1100011	100		63/4
bge	SB	1100011	101		63/5
bltu	SB	1100011	110		63/6
bgeu	SB	1100011	111		63/7
jalr	I	1100111	000		67/0
jal	UJ	1101111			6F
ecall	I	1110011	000	000000000000	73/0/000
ebreak	I	1110011	000	000000000001	73/0/001
CSRRW	I	1110011	001		73/1
CSRRS	I	1110011	010		73/2
CSRRC	I	1110011	011		73/3
CSRRWI	I	1110011	101		73/5
CSRRSI	I	1110011	110		73/6
CSRRCI	I	1110011	111		73/7

REGISTER NAME, USE, CALLING CONVENTION ④

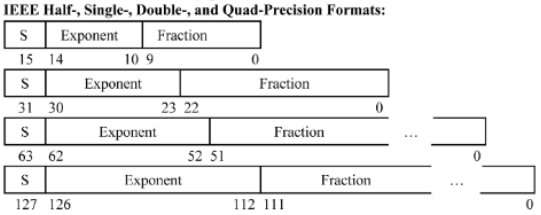
REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-f7	ft0-ft7	FP Temporaries	Caller
f8-f9	fs0-fs1	FP Saved registers	Callee
f10-f11	fa0-fa1	FP Function arguments/Return values	Caller
f12-f17	fa2-fa7	FP Function arguments	Caller
f18-f27	fs2-fs11	FP Saved registers	Callee
f28-f31	ft8-ft11	R[rd] = R[rs1] + R[rs2]	Caller

IEEE 754 FLOATING-POINT STANDARD

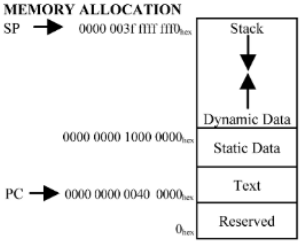
$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

where Half-Precision Bias = 15, Single-Precision Bias = 127, Double-Precision Bias = 1023, Quad-Precision Bias = 16383

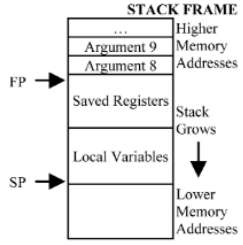
IEEE Half-, Single-, Double-, and Quad-Precision Formats:



MEMORY ALLOCATION



STACK FRAME



SIZE PREFIXES AND SYMBOLS

SIZE	PREFIX	SYMBOL	SIZE	PREFIX	SYMBOL
10^3	Kilo-	K	2^{10}	Kibi-	Ki
10^6	Mega-	M	2^{20}	Mebi-	Mi
10^9	Giga-	G	2^{30}	Gibi-	Gi
10^{12}	Tera-	T	2^{40}	Tebi-	Ti
10^{15}	Peta-	P	2^{50}	Pebi-	Pi
10^{18}	Exa-	E	2^{60}	Exbi-	Ei
10^{21}	Zetta-	Z	2^{70}	Zebi-	Zi
10^{24}	Yotta-	Y	2^{80}	Yobi-	Yi
10^{-3}	milli-	m	10^{-15}	femto-	f
10^{-6}	micro-	μ	10^{-18}	atto-	a
10^{-9}	nano-	n	10^{-21}	zepto-	z
10^{-12}	pico-	p	10^{-24}	yocto-	y

RISC-V Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

Appendix B: Execution result table for question 2

a)

	ALU/Branch	Load/Store	Cycle
LOOP:		ld t1, 0(a0)	1
			2
			3
			4
			5
			6
			7
			8
			9
			10

b)

	ALU/Branch	Load/Store	Cycle
LOOP:		ld t1, 0(a0)	1
			2
			3
			4
			5
			6
			7
			8
			9
			10
			11
			12
			13
			14

			15
			16

c)

	ALU/Branch	Load/Store	Cycle
LOOP:		ld t1, 0(a0)	1
			2
			3
			4
			5
			6
			7
			8
			9
			10
			11
			12
			13
			14
			15
			16

Appendix

Appendix