CS 152/252A
Spring 2023 Computer Architecture Midterm 1

- You have 120 minutes. You're not expected to finish the exam. We provide a variety of questions so you can demonstrate your knowledge on the topics you know best.
- You must write your student ID on the bottom-left of every page of the exam (except this first one). You risk losing credit for any page you don't write your student ID on.
- For questions with length limits, do not use semicolons or dashes to lengthen your explanation.
- The exam is closed book, no calculator, and closed notes, other than two double-sided cheat sheet that you may reference.
- For multiple choice questions,
  - ☐ means mark **all options** that apply
  - ◯ means mark a **single choice**

First name	
Last name	
SID	
Exam Room	
Name and SID of person to the right	
Name and SID of person to the left	
Discussion TAs (or None)	

Honor code: "As a member of the UC Berkeley community, I act with honesty, integrity, and respect for others."

By signing below, I affirm that all work on this exam is my own work, and honestly reflects my own understanding of the course material. I have not referenced any outside materials (other than one double-sided cheat sheet), nor collaborated with any other human being on this exam. I understand that if the exam proctor catches me cheating on the exam, that I may face the penalty of an automatic "F" grade in this class and a referral to the Center for Student Conduct.

Signature: _____

Point Distribution

Q1.	Iron Law: Doesn't Bend or Break	15
Q2.	ISAs (oh my)	15
Q3.	Pipelining: Bypass Extravaganza	30
Q4.	Make More Cache	30
Q5.	Microding: That's So Fetch	24
Q6.	Virtual Memory + Caches	25 + 10
Q7.	(CS252 Only) Virtual Memory: Build Your Own Hierarchy	15
	Total	164

THIS PAGE IS INTENTIONALLY LEFT BLANK

Q1. [15 pts] Iron Law: Doesn't Bend or Break

(a) [5 pts] What are the tradeoffs if we add support for the following CISC instruction by turning our 5-stage RISC pipeline into a **6-stage pipeline**, with a second decode stage after the original decode stage?

The instruction:

```
MEMADD rd, rs1, rs2
```

Its functionality:

```
M[R[rd]] = R[R[rs1]] + R[R[rs2]]
```

The proposed way of implementing this instruction is the following:

- Read `R[rs1]`, `R[rs2]`, and `R[rd]` from the register file.
- Read `R[R[rs1]]` and `R[R[rs2]]` from the register file.
- Compute `R[R[rs1]] + R[R[rs2]]`.
- Store the above result into memory, i.e. into `M[R[rd]]`.

Important assumptions:

Assume the register file supports at most 3 reads at a time.

Assume the instruction is only given inputs that make it execute correctly (no edge cases or exceptions).

For each of the following terms of the Iron Law, state whether the term will **Increase, Decrease, or No Change** and explain why with **at most 2** sentences.

(i) Instructions / program

- ◯ Increase
- ◯ Decrease
- ◯ No Change

(ii) Cycles / instruction

- ◯ Increase
- ◯ Decrease
- ◯ No Change

(iii) Time / cycle

- ◯ Increase
- ◯ Decrease
- ◯ No Change

For the following subparts in this question, assume the RISC-V pipeline currently only has support for *decoding* floating point instructions but **not** actually executing them. For example, if the programmer writes a floating point instruction, the instruction will currently be equivalent to a NOP (no operation).

In this question, we'll consider the tradeoffs in how we might be able to add a floating point unit to the RISC-V pipeline in order to actually be able to execute floating point instructions.

(b) [5 pts] What are the tradeoffs if we add an **unpipelined** floating point unit to the RISC-V pipeline?

For each of the following terms of the Iron Law, state whether the term will **Increase, Decrease, or No Change** and explain why with **at most 2** sentences.

(i) Instructions / program

- ◯ Increase
- ◯ Decrease
- ◯ No Change

(ii) Cycles / instruction

- ◯ Increase
- ◯ Decrease
- ◯ No Change

(iii) Time / cycle

- ◯ Increase
- ◯ Decrease
- ◯ No Change

(c) [5 pts] What are the tradeoffs if we add a **pipelined** floating point unit to the RISC-V pipeline?

For each of the following terms of the Iron Law, state whether the term will **Increase, Decrease, or No Change** and explain why with **at most 2** sentences.

(i) Instructions / program

- ◯ Increase
- ◯ Decrease
- ◯ No Change

(ii) Cycles / instruction

- ◯ Increase
- ◯ Decrease
- ◯ No Change

(iii) Time / cycle

- ◯ Increase
- ◯ Decrease
- ◯ No Change

Q2. [15 pts] ISAs (oh my)

Suppose we have the following program:

```
int remainder(int a, int b) {
    while (a >= b) {
        a = a - b;
    }
    return a;
}
```

For a RISC architecture, Our RISC-V compiler generates the following assembly code:

```
loop:
    blt x1, x2, done
    sub x1, x1, x2
    j loop
done:
    ...
```

Meanwhile, our Stack architecture includes the following instructions:

add **Instruction**	**Definition**
PUSH addr	load value at addr; push value onto stack
POP addr	pop stack; store value to addr
SLT	pop two values from the stack; if the first is less than the second, push 1 onto the stack, else push 0
SUB	pop two values from the stack; subtract the second from the first; push result onto stack
BNEZ label	pop value from stack; if it's not zero, branch to label; else, continue with next instruction
JUMP label	continue execution at location label

Our stack compiler generates the following assembly code (assume *a* is initially stored at **0x8000** and *b* is initially stored at **0x8004**:

```
loop:
    PUSH 0x8004
    PUSH 0x8000
    SLT
    BNEZ done
    PUSH 0x8004
    PUSH 0x8000
    SUB
    POP 0x8000
    JUMP loop
done:
    ...
```

Assume that RISC-V instructions are 4 bytes each. Assume a Stack instruction occupies three bytes if it takes an address or label; other instructions occupy one byte. Assume data vales are 4 bytes each.

(a) [2 pts] How many bytes is the RISC program?

- ◯ 10
- ◯ 12
- ◯ 14
- ◯ 16

(b) [2 pts] How many bytes is the Stack program?

- ◯ 13
- ◯ 73
- ◯ 23
- ◯ 33

Assume a=7 and b=3.

(c) [2 pts] How many instruction bytes are fetched in the RISC program?

- ◯ 18
- ◯ 24
- ◯ 28
- ◯ 36

(d) [2 pts] How many instruction bytes are fetched in the Stack program?

- ◯ 33
- ◯ 52
- ◯ 46
- ◯ 56

(e) [2 pts] How many data bytes are loaded/stored in the RISC program?

- ◯ 0
- ◯ 16
- ◯ 32
- ◯ 40

(f) [2 pts] How many data bytes are loaded/stored in the Stack program?

- ◯ 16
- ◯ 32
- ◯ 48
- ◯ 52

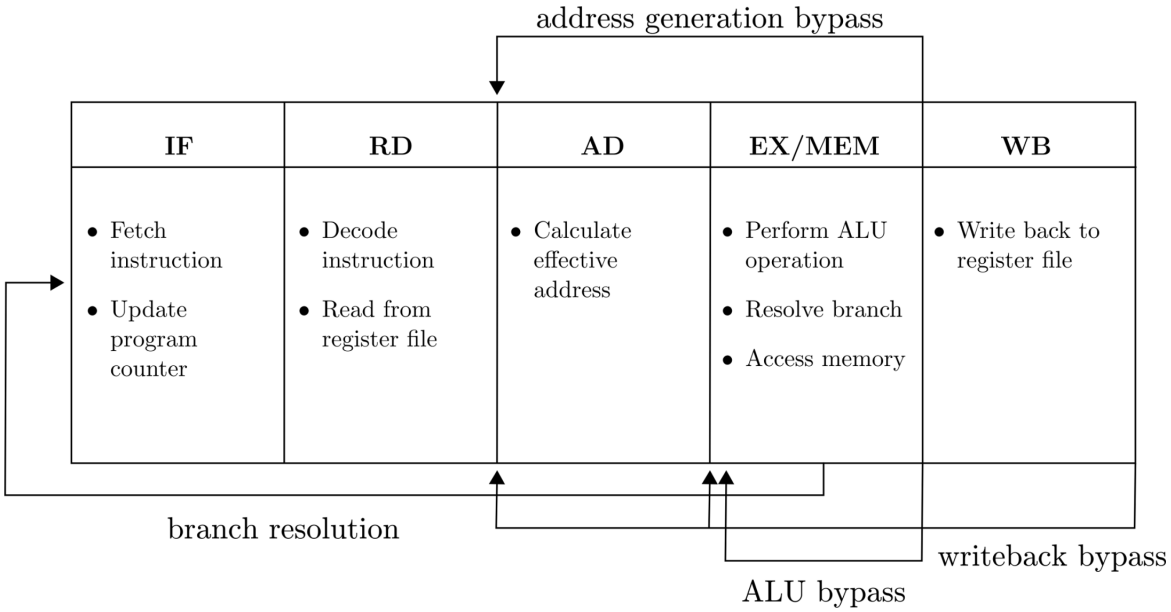
(g) [3 pts] Which ISA is better for the given implementations?

- ◯ RISC
- ◯ Stack
- ◯ Both are equally good

Use the Iron Law to justify your answer in **at most 2 sentences**.

Q3. [30 pts] Pipelining: Bypass Extravaganza

We introduce the Address Generation Interlock (AGI) pipeline below, which eliminates the load-use hazard of our more conventional Load-Use Interlock (LUI) pipeline. In turn, we deal with an address generation hazard, which warrants the address generation bypass path. In this problem, we will analyze the performance of this pipeline.



Parts (a) through (c) in this problem will be analyzing the following code snippet applied on the AGI pipeline. Register **s0** holds the address of an array whose elements are addresses that point randomly to other elements of the array. **Assume that the register file does not support reading the new value of a register in the same cycle that it is being written, and that none of the load operations access memory outside of the array bounds.**

```
      add  t2, x0, s0
      addi t1, x0, 0
      addi t0, x0, 100    # Runs for 100 iterations
LOOP: lw   t2, 0(t2)
      lw   t2, 0(t2)
      addi t1, t1, 1
      bne  t0, t1, LOOP
```

(a) [2 pts] What does the code snippet above seem to be doing?

- ◯ Vector addition
- ◯ Computing the sum of array elements
- ◯ Pointer chasing
- ◯ Initializing a C struct

(b) Analyze the code on the AGI pipeline. Assume that **the address generation bypass has not been added** to the pipeline for this part. We will be working with an **always-taken branch predictor**. Assume memory accesses take one cycle.

(i) [6 pts] Fill in the pipeline diagram on the following page for **TWO iterations of the loop**, assuming that the branch condition evaluates to true both times.

(ii) [3 pts] What is the CPI (cycles per instruction) for the **100 iterations of the loop** on the AGI pipeline? (Do not include the initial `add` and `addi` instructions.)

SID: _____

	Instructions	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19				
1	add t2, x0, s0	F	D	A	XM	W																		
2	addi t1, x0, 0		F	D	A	XM	W																	
3	addi t0, x0, 100			F	D	A	XM	W																
4																								
5																								
6																								
7																								
8																								
9																								
10																								
11																								
12																								
13																								
14																								
15																								
16																								
17																								
18																								

(c) We now add the address generation bypass path to the pipeline. We still use an always-taken branch predictor, and we assume memory accesses take one cycle.

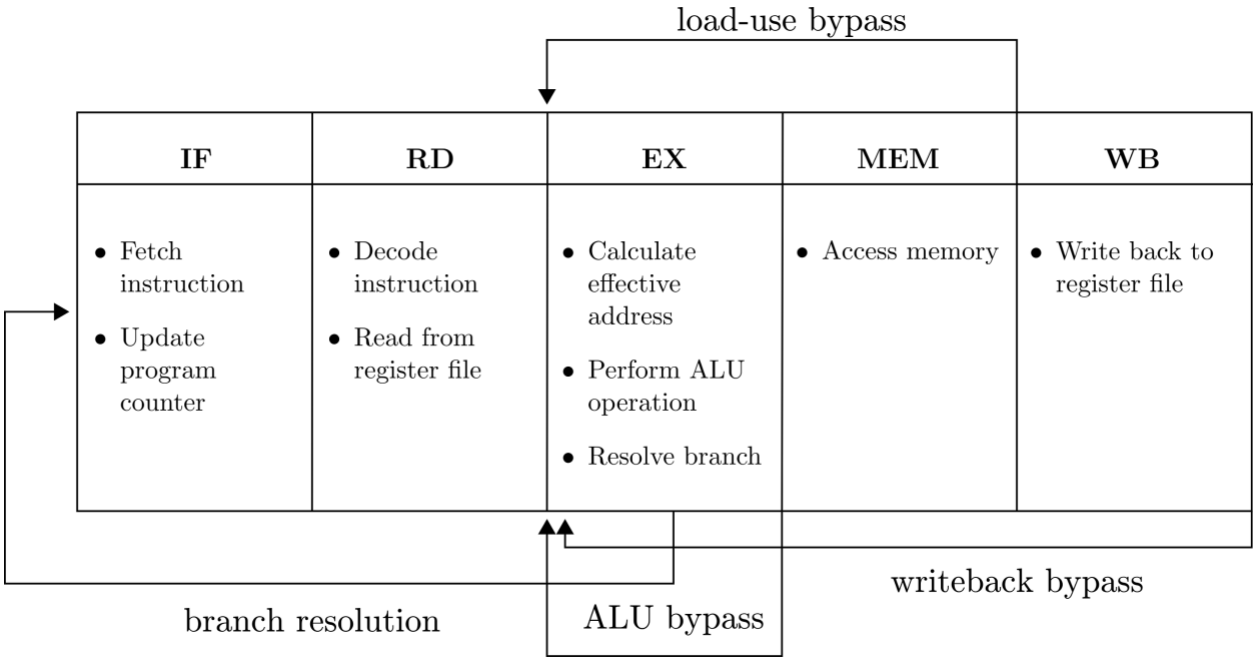
(i) [2 pts] What hazard(s) does the address generation bypass path resolve?

- ☐ Structural
- ☐ Data
- ☐ Control

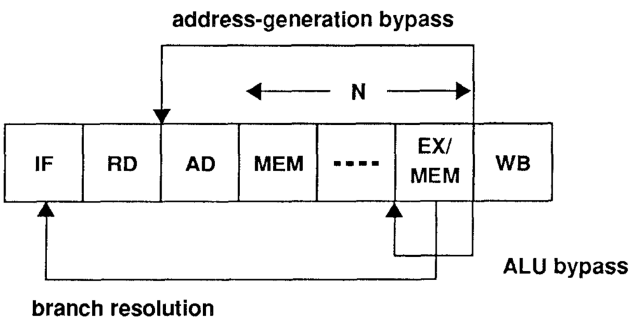
(ii) [4 pts] What is the CPI (cycles per instruction) for the **100 iterations of the loop** on the AGI pipeline after adding the address generation bypass path? (Do not include the initial `add` and `addi` instructions.) For your convenience, we have provided another pipeline diagram two pages ahead as scratch space, but it will not be graded.

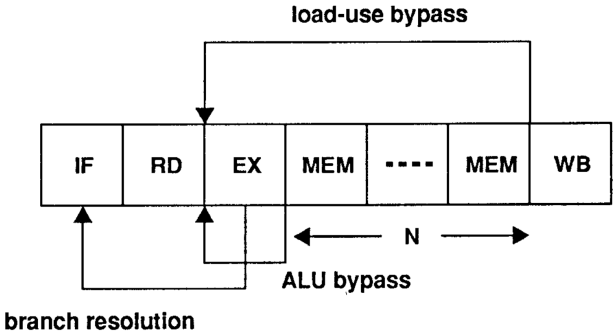
(d) [5 pts] In which of the following independent scenario(s) might we prefer the AGI pipeline over our conventional LUI pipeline? Assume that if our data cache memory accesses take N cycles, then we pipeline the MEM stage N times.

Give a **maximum total three sentence justification** on your selection(s). Below is a diagram of the LUI pipeline. Answer options continue on the next page.



Below are diagrams for the AGI and LUI pipelines respectively that illustrates a multiple cycle (thus, pipelined) MEM stage.





☐ With perfect branch prediction

☐ When we need to consume less area with our design

☐ When data cache memory accesses take multiple cycles, paired with good branch prediction

☐ None of the above

(e) Consider the following code snippet:

```
lw t1, 0(t2)
addi t2, t2, 0x100
```

(i) [3 pts] Suppose that the *lw* instruction raises a memory address exception, while the *addi* instruction raises an integer overflow exception. For which of the following pipeline designs would this code snippet cause an imprecise exception to occur? Select all that apply.

Assume that exceptions are handled immediately when they are raised and if two exceptions are raised simultaneously, then the exception corresponding to the earlier instruction is prioritized.

☐ A single cycle non-pipelined machine

☐ The AGI pipeline above

☐ A modified LUI pipeline with the MEM stage split into M1 and M2 stages (memory writes and exceptions occur in M2)

☐ None of the above

(ii) [5 pts] Which of the following hardware modifications would add precise exception support to the design(s) that raised imprecise exceptions? Select all that apply. Assume that all modifications you select will be implemented in conjunction.

Give a **maximum total two sentence justification** on how your selected modification(s) would resolve the imprecise exception(s).

☐ Stall throwing ALU-related exceptions by one cycle

☐ Hold exception flags in pipeline until commit point

☐ Add an additional read port to the regfile and to memory

☐ None of the above: all pipelines support precise exceptions for this code snippet

SID: _____

	Instructions	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19				
1	add t2, x0, s0	F	D	A	XM	W																		
2	addi t1, x0, 0		F	D	A	XM	W																	
3	addi t0, x0, 100			F	D	A	XM	W																
4																								
5																								
6																								
7																								
8																								
9																								
10																								
11																								
12																								
13																								
14																								
15																								
16																								
17																								
18																								

Q4. [30 pts] Make More Cache

(a) [3 pts] Given a 2-way set associative 256 byte L1 cache that has an 8-bit address and 32 byte cache lines in a byte-addressable system, how many bits would correspond to the tag, index, and offset for this cache?

(i) Tag

(ii) Index

(iii) Offset

(b) [8 pts] Like you did in Homework 2, fill in the following table with the **hexadecimal tags** that populate each cache line. We will not grade the contents of the table and will only grade your answer to part (i) below the table.

Assume an initially empty 2-way set-associative L1 cache with a FIFO replacement policy, independent from the cache discussed in Part (a). Assume that we fill Way 0 before Way 1. The address with respect to the L1 cache has the following division: **3 bit tag, 1 bit index, and 4 bit offset**.

Cache Access Table					
Address	Set 0, Way 0	Set 0, Way 1	Set 1, Way 0	Set 1, Way 1	Hit (Y/N)
`0b1000 1000`	(A)				
`0b0000 0000`					
`0b1000 0010`					(B)
`0b0101 0010`					
`0b0001 0000`		(F)			
`0b0100 0000`	(H)			(C)	
`0b1000 1101`		(D)			
`0b0010 0000`					(G)
`0b0011 0000`			(E)		

(i) What are the hexadecimal tags or hit values (in Y/N) for the cells labeled as A-H?

(A) (B) (C) (D) (E)

(F) (G) (H)

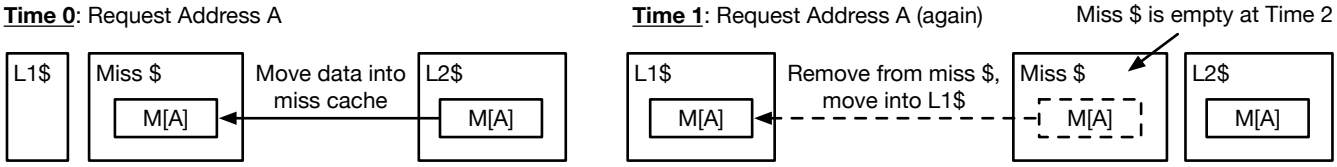
(c) [2 pts] Determine the average memory access time (AMAT) of the following program description. Assume the hit time of the L1 cache is 4ns and the L2 access time is 20ns. Of your 10 memory accesses, 2 are L1 hits. Assume all L1 misses are resolved in the L2.

Only your answer in the small box will be graded; you may use the bigger box on the next page for work, but it will not be graded.

(d) [4 pts] Now let's consider the effect of adding a miss cache to our cache hierarchy. A miss cache, similar to a victim cache, is a small, fully associative cache that is placed between the L1 and L2 caches. When a cache line is first requested, it is placed in the miss cache instead of the L1 cache. If the L1 cache misses on the line again while it is still present in the miss cache, it is moved into the L1 cache.

When an address is requested, the processor first checks the L1 cache for the line, then the miss cache, and then, the L2 cache.

Additionally, the L1 cache and miss cache are *exclusive*. That is, you will not find the same cache line present in both the miss cache and the L1 cache — a line may exist in at most one cache at a time.



Select all that apply for what is true about a miss cache.

☐ A miss cache naturally adheres to the following replacement policies: MRU (most recently used), when moving data to the L1, and FIFO (first-in-first-out), when evicting due to full capacity.

☐ A miss cache is always helpful, but we may not always add it to our design because it consumes area.

☐ A miss cache aims to prevent L1 cache pollution by filtering out data that is less spatially local.

☐ A miss cache aims to prevent L1 cache pollution by filtering out data that is less temporally local.

(e) [9 pts] We will now analyze the same access pattern from part (b) with the addition of a miss cache, which is explained in part (c).

Again, fill in the following table with the **hexadecimal tags** that populate each cache line. We will not grade the contents of the table and will only grade your answer to part (i) below the table.

Assume an initially empty 2-way set-associative L1 cache with a FIFO replacement policy. Assume that we fill Way 0 before Way 1. The address with respect to the L1 cache has the following division: 3 bit tag, 1 bit index, and 4 bit offset. Assume an initially empty fully associative miss cache, also with a FIFO replacement policy. The address with respect to the miss cache has the following division: 4 bit tag and 4 bit offset.

Cache Access Table								
Address	L1 Set 0 Way 0	L1 Set 0 Way 1	L1 Set 1 Way 0	L1 Set 1 Way 1	Hit (Y/N)	MC Way 0	MC Way 1	Hit (Y/N)
`0b1000 1000`								(F)
`0b0000 0000`							(A)	
`0b1000 0010`					(I)			(B)
`0b0101 0010`						(G)		
`0b0001 0000`								
`0b0100 0000`	(C)							
`0b1000 1101`					(D)	(H)		
`0b0010 0000`								
`0b0011 0000`						(E)		

(i) What are the hexadecimal tags or hit values (in Y/N) for the cells labeled as A-H?

(A) (B) (C) (D) (E)

(F) (G) (H) (I)

(f) [4 pts] Determine the average memory access time (AMAT) of the following program description using this miss cache-based system. Once again, assume the hit time of the L1 cache is 4ns, and the L2 access time is 20ns. Additionally, assume the miss cache access time is 10ns. Of your 10 memory accesses, 2 are L1 hits. Another 2 are resolved in the miss cache. Assume all L1 and miss cache misses are resolved in the L2.

Only your answer in the small box will be graded; you may use the bigger box for work, but it will not be graded.

Q5. [24 pts] Microding: That's So Fetch

Skip Registering provides conditional execution without branches and control dependencies. Each instruction is associated with a third register, `rs3`, called a **skip register**. If the skip register evaluates to zero, the instruction executes normally. If the skip register evaluates to a non-zero value, the instruction is treated as a no-op.

We extend the RISC-V ISA to add skip registering as follows. We allow any of the 32 RISC-V register (`x0` through `x31`) to be used as a skip register. We also change the encoding of RISC-V instructions to allow them to be skip registered on the value of any of the 32 RISC-V registers.

We denote skip registered instructions in assembly by prefixing them with the skip register in parenthesis before the instruction. For example:

```
(x0) li x1, 1
```

denotes that the `li` instruction is skip registered on the register `x0`. Since `x0` always evaluates to `0`, the `li` instruction will be executed and `x1` will be set to `1`. If, after the execution of the previous instruction, the following instruction is executed:

```
(x1) li x1, 2
```

The value of `x1` will remain unchanged at `1` because the skip register (`x1`) evaluates to a non-zero value and the instruction is treated as a no-op.

With these changes, we can implement conditional execution without branches. For example, consider the C code:

```
if (x1 == 0) {  // x2 != 0
    x3 = x4;
} else {        // x2 == 0
    x3 = x5;
}
```

where the values of C variables `x1`, `x2`, `x3`, `x4`, `x5` are stored in the RISC-V registers `x1`, `x2`, `x3`, `x4`, `x5`. With skip registering, we can express this code in assembly without a branch instruction, and using `x2` as an intermediate:

```
(x0) sltiu x2, x1, 1  // x2 = 1 if x1 == 0, else x2 = 0
(x1) mov x3, x4       // executes if x1 == 0
(x2) mov x3, x5       // executes if x1 != 0 (i.e. x2 == 0)
```

(a) What Does This Skippy Code Do?

Given the following block of skip-registered RISC-V code:

```
(x0) li x1, 1
(x0) sltiu x2, x1, 1  // x2 = 1 if x1 == 0, else x2 = 0
(x0) li x5, 0
(x0) li x6, 0
(x1) addi x6, x6, 4
(x2) addi x6, x6, 2
# Comment A
(x0) add x5, x5, x6
(x1) li x7, 0
(x2) li x7, 5
# Comment B
```

Answer the following questions:

(i) [2 pts] At `Comment A`, what is the value of `x5`?

(ii) [2 pts] At `Comment A`, what is the value of `x6`?

(iii) [2 pts] At `Comment B`, what is the value of `x5`?

(iv) [2 pts] At `Comment B`, what is the value of `x7`?

(b) [12 pts] **Microcoded Implementation**

To implement skip registering, we must update `FETCH0` to ensure instructions with non-zero valued skip registers are not run, while instructions with *zero* valued skip registers *are* run.

Implement this new fetch instruction starting at `FETCH0` in the microcode table provided on page 19 of the exam booklet.

Important: Your microcode implementation should only use the **optimal, minimal** amount of lines possible to achieve this skip-registering fetch functionality.

Helpful notes:

- As a reminder, all instructions under this skip registering scheme will be associated with an additional register `rs3`. Although this change is not reflected in Handout 1, the `RegSel` mux has been extended to include `rs3`, enabling this new instruction argument to be accessed from the bits of an instruction just as how we can access `rd`, `rs1`, and `rs2` when applicable.
- You may also find it helpful to draw inspiration from the original, non-skip register enabled implementation of `FETCH0` included on page 19 of the exam booklet.
- Focus first on making your implementation *correct*. *Then*, focus on making your correct implementation *optimal*.

SID: _____

Old Microcode Implementation of FETCH0 for a Non-Skip Registered System

State	Pseudocode	IR Ld	Reg Sel	Reg Wr	Reg En	A Ld	B Ld	ALUOp	ALU En	MA Ld	Mem Wr	Mem En	Imm Sel	Imm En	μBr	Next State
FETCH0:	MA ← PC; A ← PC	*	PC	0	1	1	*	*	0	1	0	0	*	0	N	*
	IR ← Mem	1	*	0	0	0	*	*	0	0	0	1	*	0	S	*
	PC ← A+4 Dispatch to inst.	0	PC	1	0	0	*	INC_A_4	1	*	0	0	*	0	D	*

New Microcode Implementation of FETCH0 for a Skip Registered System

State	Pseudocode	IR Ld	Reg Sel	Reg Wr	Reg En	A Ld	B Ld	ALUOp	ALU En	MA Ld	Mem Wr	Mem En	Imm Sel	Imm En	μBr	Next State
FETCH0:																

(c) Can we do better?

(i) [4 pts] Why might skip registering be more useful for a very deeply pipelined CPU instead of a microcoded CPU? You may assume the microcoded CPU uses the implementation presented in class. Limit your answer to 2 sentences (do not use semicolons or dashes to lengthen your response).

Q6. [25 + 10 pts] Virtual Memory + Caches

You are given a VIPT (virtually-indexed, physically-tagged) cache implementation (4-way set associative cache) with the following address partitioning for both the cache input and virtual address.

tag bits [31:11]	index bits [10:6]	line offset [5:0]

Table 4: Cache Address Partitioning

VPN bits [31:9]	page offset [8:0]

Table 5: Virtual Address Partitioning

Below you are given a simplified VIPT cache implementation and corresponding TLB that is accessed in parallel. Remember that in a VIPT cache, the tag check is completed using the PPN (physical page number) obtained from a TLB/page-table-walk.

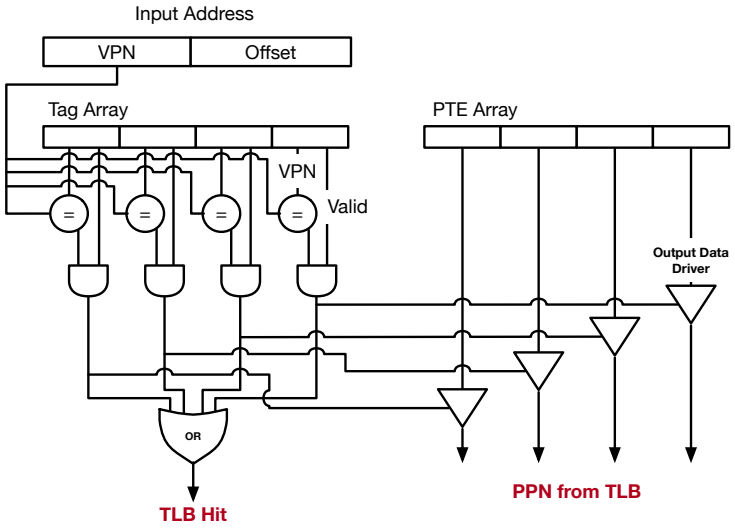


Figure 1: 4-way Fully Set-Associative TLB Implementation

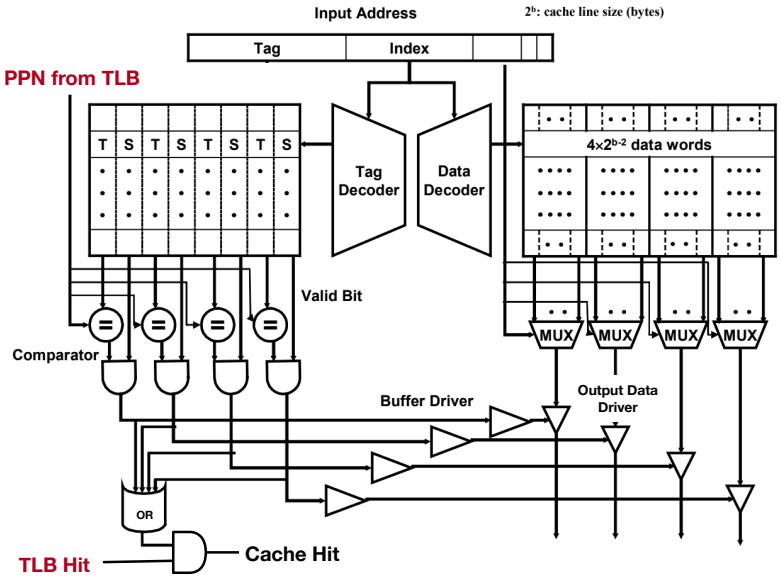


Figure 2: 4-way Set-Associative VIPT Cache Implementation

(a) [11 pts] For simplicity, assume that the TLB always hits, and that there are no page faults (PPN is always valid within a PTE (page table entry)). As seen in the diagrams, the TLB access happens in parallel with the cache lookup, and the "TLB Hit" signal is later used to determine whether the cache entry is valid. Given the block delay expressions for individual components given below and the diagrams of the VIPT cache and TLB, answer the following questions. Keep in mind that the ceil(X) operation returns the smallest integer value that is greater than or equal to X.

Component	Delay equation (ps)
Decoder	30 × (num index bits) + 80
Memory array	30 x ceil($\log_2$(num sets)) + 30 x ceil($\log_2$(num bits per set)) + 100
Comparator	30 x (num tag bits) + 70
N-to-1 MUX	50 x ceil($\log_2$(N)) + 100
Output-data/buffer driver	180
OR Gate	50
AND Gate	40

(i) [3 pts] For the longest path from the VPN tag check to the output data in the data cache.
What are the components along the path, along with the delay of the path in ps?

(ii) [3 pts] For the longest path from the Input Address index to the output data in the data cache.
What are the components along the path, along with the delay of the path in ps?

(iii) [3 pts] For the longest path from the VPN tag check to the cache hit signal in the data cache.
What are the components along the path, along with the delay of the path in ps?

(iv) [2 pts] Of the three paths given, which is the critical path? Other paths do not need to be considered for the critical path in this question, even though the list given above is not a comprehensive list of all possible paths.

- ◯ (i) VPN tag check -> Output data
- ◯ (ii) Cache idx -> Output data
- ◯ (iii) VPN tag check -> Cache hit

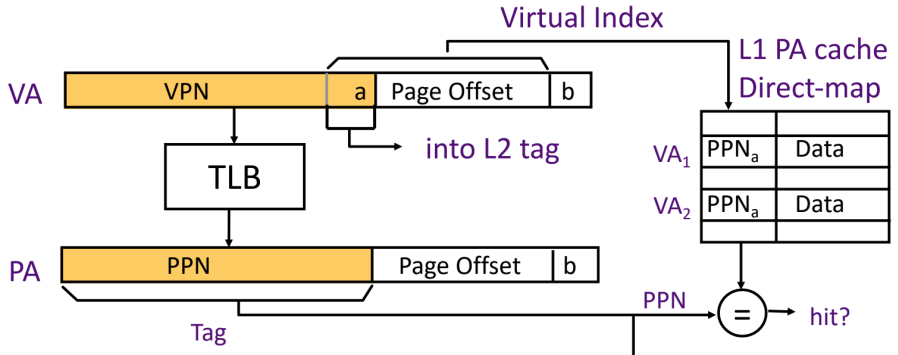
(b) [4 pts] Does this implementation of a VIPT cache with a parallel TLB have aliasing issues? If so, what is causing the aliasing? If not, explain why aliasing would not be an issue. You may write **at most 2 sentences** of explanation.

(c) Regardless of your answer to part b, assume that the cache does indeed have aliasing issues. To solve them, the overall cache organization can be modified.

(i) [6 pts] Assuming fixed page size and fixed overall cache size, what modifications can be done to the cache and TLB organization to ensure there are no aliasing issues? You may write **at most 3 sentences** of explanation.

(ii) [4 pts] What is the estimated impact on critical path? You may write **at most 2 sentences** of explanation.

(d) [10 pts] **(CS252 Only)** Another mechanism to solve aliasing is by adding an inclusive L2 cache that checks for colliding virtual addresses sharing the same physical tag. The following diagram describes a case where two virtual addresses collide taken from lecture.



- Suppose VA1 and VA2 both map to PA and VA1 is already in L1, L2 (VA1 ≠ VA2)
- After VA2 is resolved to PA (miss in L1), a collision will be detected in L2.
- VA1 will be purged from L1 and L2 (using a + page offset), and VA2 will be loaded ⇒ *no aliasing !*

(i) [2 pts] How many sets can potentially have an aliased cache line (assuming the same cache implementation given above)? Assume we are using the same cache organization as what was described in part a.

(ii) [2 pts] Using the eviction mechanism above, is it possible for there to be more than one alias (for example 3 alias conflicts) in the cache?

- ◯ Yes
- ◯ No

(iii) [4 pts] Assuming you can read any number of sets from the tag array (unlimited read ports for the tag array), what is the minimum amount of comparators needed to make Figures 1 and 2 evict the previous aliased cache line? Assume that this eviction can happen in a cycle(s) separate from the main cache miss/hit. You may write **at most 4 sentences** of explanation.

(iv) [2 pts] If the tag array is now single-ported (i.e. only one set can be read at a time), what modifications can be made to Figures 1 and 2 for eviction? You may write **at most 4 sentences** of explanation.

Q7. [15 pts] (CS252 Only) Virtual Memory: Build Your Own Hierarchy

As the newest memory hierarchy designer at Pear, your initial task is to reduce the amount of memory requests that the pearPhone operating system runs on the new PearOne system-on-chip.

Initially your boss hands you the following incomplete virtual specification.

Description	Value
Virtual Address Bits	24 bits
Page Size	4 Kilobytes
Page Table Levels	2 Levels
Physical Address Bits	32 bits
Addressability	Byte-addressable
Page Table Entry Size	4 Bytes

(a) Given the information by your boss, fill out the following. Assume the address is split into the following sections ordered from the MSB to LSB: p0 and p1 (indexes into the page tables), and the page offset. Assume that the virtual page number (VPN) bits are evenly split into all p* indexes (i.e. p0 + p1 = VPN).

(i) [1 pt] p0 bits in virtual address

(ii) [1 pt] p1 bits in virtual address

(iii) [1 pt] Page offset bits in virtual address

(iv) [2 pts] How many pages does it take to store all contents of all page tables with only 1 page mapped to physical memory (paged in).

After hearing about the RISC-V Sv39 virtual memory scheme using 3-levels of page tables, your boss decides to ignore prior implementation and provides the following address partitioning for a virtual memory address. Similar to before, the VPN is comprised of all p* indexes (i.e. p0 + p1 + p2 = VPN).

p0 bits [23:18]	p1 bits [17:12]	p2 bits [11:6]	page offset [5:0]

Table 6: New Virtual Address Partitioning

(b) [2 pts] How many pages does it take to store all contents of all pages tables with this new scheme and only 1 page mapped to physical memory (paged in)?

(c) [4 pts] Your boss gives you a memory trace from PearOS that accesses 18 consecutive memory locations with 1KB stride starting from address 0x0 in the following manner: 0x0, 0x400, 0x800, 0xC00 ... 0x4000. How many memory accesses are done assuming there is no TLB and no page faults?

(d) [4 pts] This scheme provides support for three types of pages: normal pages, kilo-pages, and big-pages. The size of each is determined by the combination of the p* index bits and page byte offset bits in the following manner.

Page Type	Address Partitions To Combine
Normal Page	Page offset bits
Kilo-page	p2 + page offset bits
Big-page	p1 + p2 + page offset bits

(i) Given the same access pattern as before (18 consecutive memory locations with a stride of 1KB - 0x0, 0x400, 0x800, ... 0x4000), what is the minimum number of memory accesses needed with no TLB? Make sure to also minimize internal fragmentation.