CS152 Computer Architecture and Engineering

ISAs, Microprogramming, and Pipelining

Assigned *01/28/2026*	Problem Set #1, Version (1.0)	*Due February 8 @ 11:59:59PT*

https://cs152-teach.github.io/sp26-web/

The problem sets are intended to help you learn the material, and we encourage you to collaborate with other students and to ask questions in discussion sections and office hours to understand the problems. *However, each student must turn in their own solution to the problems.*

The problem sets also provide essential background material for the exam and the midterms. The problem sets will be graded primarily on an effort basis, but if you do not work through the problem sets yourself you are unlikely to succeed on the exam or midterms! We will distribute solutions to the problem set on the day after the deadline to give you feedback.

Assignments must be submitted through Gradescope by **11:59:59pm PT** on the specified due date. *Box all solutions that don't involving filling in a figure/table. Only boxed solutions and filled in figures/tables will be considered for grading.* See the course website for the policy on late submissions.

Name: _____SOLUTIONS_____

SID: _____

Collaborators (Name, SID):

_____

Problem 1: CISC, RISC, Accumulator, and Stack: Comparing ISAs

In this problem, your task is to compare four different ISAs: x86 (a CISC architecture with variable-length instructions), RISC-V (a load-store, RISC architecture with 32-bit instructions in its base form), a stack-based ISA, and an accumulator-based ISA.

Problem 1.A — CISC

Let us begin by considering the following C code, which (inefficiently) rotates the bits in a 32-bit value by `n` times.

```
unsigned int rotate(unsigned int x, unsigned int n) {
   unsigned int msb;

   while (n != 0) {
       msb = x >> 31;
       x = (x << 1) | msb;
       n--;
   }
   return x;
}
```

Using gcc and objdump on an x86 machine, we see that the above loop compiles to the following x86 instruction sequence. On entry to this code, register `%eax` contains `x` and register `%ecx` contains `n`. Throughout parts (a–d), we will ignore what happens in the `done` label and return statement.

```
loop:  test %ecx,%ecx
           jz  done
           mov %eax,%ebx
           shr $31,%ebx
           shl $1,%eax
           or  %ebx,%eax
           dec %ecx
           jmp loop
 done: ...
```

The meanings and instruction lengths of the instructions used above are given in the following table. Registers are denoted with $R_{SUBSCRIPT}$, register contents with $\langle R_{SUBSCRIPT} \rangle$.

Instruction	Operation	Length
mov R_{SRC}, R_{DEST}	$\langle R_{DEST} \rangle$ = $\langle R_{SRC} \rangle$	2 bytes
test R_{SRC1}, R_{SRC2}	temp = $\langle R_{SRC1} \rangle$ & $\langle R_{SRC2} \rangle$ Set flags based on value of temp	2 bytes
dec R_{DEST}	$\langle R_{DEST} \rangle$ = $\langle R_{DEST} \rangle$ - 1	1 byte
shl $imm8, R_{DEST}	$\langle R_{DEST} \rangle$ = $\langle R_{DEST} \rangle$ << imm8	2 bytes
shr $imm8, R_{DEST}	$\langle R_{DEST} \rangle$ = $\langle R_{DEST} \rangle$ >> imm8	2 bytes

`or` R_{SRC}, R_{DEST}	<R_{DEST}> = <R_{DEST}> | <R_{SRC}>	2 bytes
`jmp label`	jump to the address specified by `label`	2 bytes
`jz label`	if (`ZF == 1`), jump to the address specified by `label`	2 bytes

Notice that the jump instruction `jz` (jump if zero) depends on `ZF`, which is a status flag. Status flags are set by the instruction preceding the jump, based on the result of the computation. Some instructions, like the `test` instruction, perform a computation and set status flags such as `ZF`, but do not return any result. The meanings of the status flags are given in the following table:

Name	Purpose	Condition Reported
`ZF`	Zero	Result is zero

i) How many bytes is the program?
ii) For the above x86 assembly code, how many bytes of instructions need to be fetched if `x = 0x01020304` and `n = 7`?
iii) Assuming 32-bit data values, how many bytes of data memory need to be loaded? Stored?

i) Bytes in program = 7 * 2 bytes + 1 byte = 15 bytes
ii) Instruction bytes fetched = 7 * 15 bytes (loop iterations) + 4 bytes (final `test` and `jz`) = 109 bytes
iii) Data loaded/stored = 0 bytes

Problem 1.B **RISC**

Translate each of the x86 instructions in the following table into zero, one or more RISC-V instructions. Place the `loop` label where appropriate. You should use the minimum number of instructions needed to translate each x86 instruction. (You are allowed to replace *multiple* x86 instructions with a single RISC-V instructions). Assume that `x1` contains `x` upon entry, and `x2` should receive `n`. If needed, use `x4` as a condition register, and `x6`, `x7`, etc., for temporaries. You should not need to use any floating-point registers or instructions in your code. A description of the RISC-V instruction set architecture can be found on the class website resources page.

Note: It is possible to rotate a string in O(1) time without a loop. For this problem, we want you to use the more inefficient O(N) loop-based solution.

x86 instruction	Label	RISC-V instruction sequence
`test %ecx,%ecx`	`loop:`	`beq x2, x0, done`
`jz done`		
`mov $eax,%ebx`		`srli x6, x1, 31`
`shr $31,%ebx`		
`shl $1,%eax`		`slli x1, x1, 1`
`or %ebx,%eax`		`or x1, x6, x1`
`dec %ecx`		`addi x2, x2, -1`
`jmp loop`		`jal x0, loop`
…	`done:`	…

i) How many bytes is the RISC-V program using your direct translation?
ii) How many bytes of RISC-V instructions need to be fetched for `x = 0x01020304` and `n = 7` with your direct translation?
iii) Assuming 32-bit data values, how many bytes of data memory need to be loaded? Stored?

The answer for this question may be slightly different, depending on how exactly students translated their programs.

i) Bytes in program = 6 * 4 bytes = 24 bytes

ii) Instruction bytes fetched = 7 * 24 bytes (loop iterations) + 4 bytes (final `beq`) = 172 bytes

iii) Bytes loaded/stored = 0 bytes

Problem 1.C **Stack**

In a stack architecture, all operations occur on top of the stack. Only push and pop access memory, and all other instructions remove their operands from the stack and replace them with the result. The hardware implementation we will assume for this problem set uses stack registers for the topmost two entries; accesses that involve deeper stack positions (e.g., pushing or popping something when the stack has more than two entries) use an extra memory reference. Assume each instruction occupies three bytes if it takes an address or label; other instructions occupy one byte.

Instruction	Definition
PUSH *addr*	load value at *addr*; push value onto stack
POP *addr*	pop stack; store value to *addr*
OR	pop two values from the stack; OR them; push result onto stack
SHL	pop value from top of stack; shift left by 1; push result onto stack
SIGN	pop value from top of stack; shift right by 31; push result onto stack
DEC	pop value from top of stack; decrement value by 1; push result onto stack
BEQZ *label*	pop value from stack; if it's zero, branch to *label*; else, continue with next instruction
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*

Translate the `rotate` loop to the stack ISA. You are permitted to change the sequence of instructions from 1.A and 1.B. Assume that when we reach the loop, `n` is at the top of the stack and `x` is underneath it. At the end of the loop, the stack should contain only `x` at the top. Assume that byte-addressable memory starting at address 0x8000 is available to use as temporary storage. Assume that data values are 32-bits wide.

i) How many bytes is your program?
ii) How many bytes of instructions need to be fetched for `x = 0x01020304` and `n = 7` with your translation?
iii) How many bytes of data memory need to be loaded? Stored? Remember accesses that involve deeper stack positions (e.g., pushing or popping something when the stack has more than two entries) use an extra memory reference.
iv) Would the number of bytes loaded and stored change if the stack could fit 8 entries in registers?

Answers may be slightly different depending on how students translated their programs.

```
loop:  pop 0x8000 # [n,x]; n is at 0x8000
       push 0x8000 # [x]
       beqz done #[n,x]
       pop 0x8004 # [x]; x is at 0x8004
       push 0x8004 # []
```

```
            shl # [x]
            push 0x8004 # [x<<1]
            sign # [x, x<<1]
            or # [msb(x), x<<1]
            push 0x8000 # [x']
            dec # [n,x']
            jump loop # [n-1,x']
done:       …
```

i) Bytes in program = 8 * 3 bytes (instructions with addresses/labels) + 4 * 1 byte (instructions *without* addresses/labels) = 28 bytes
ii) Instruction bytes fetched = 7 * 28 bytes (loop iterations) + 3 * 3 bytes (final `beqz`) = 205 bytes
iii) Bytes loaded = 7 * 4 pushes * 4 bytes (loop iterations) + 1 push * 4 bytes (final `beqz`) = 116 bytes
Bytes stored = 7 * 2 pops * 4 bytes (loop iterations) + 1 pop * 4 bytes (final `beqz`) = 60
iv) The solution above doesn't use more than two stack entries, so there would be no change if more stack entries were added. But if your solution does use more than two stack entries, then adding more registers may eliminate implicit loads.

Problem 1.D **Accumulator**

In an accumulator ISA, one of the operands and the destination are always the same. Further, this operand/destination may be explicit or implicit (in which case it is the same for all instructions). We will consider an accumulator ISA with an implicit accumulator register in this problem. To make programming easier, we will consider a modified architecture that has a secondary accumulator to hold an additional value. Assume each instruction occupies three bytes if it takes an address or label; other instructions occupy one byte. To make programming easier, we will consider a modified architecture that has a secondary accumulator to hold an additional value. Assume each instruction occupies three bytes if it takes an address or label; other instructions occupy one byte.

Instruction	**Definition**
LOAD *addr*	load value at *addr* into the primary accumulator
STORE *addr*	store the primary accumulator's value to *addr*
OR *addr*	OR the value at *addr* with the value in the primary accumulator
SHL	left-shift the value in the primary accumulator by one bit
SIGN	logical right-shift the value in the primary accumulator by 31 bits
INC	increment the primary accumulator by 1
DEC	decrement the primary accumulator by 1
SWAP	swap the values in the primary and secondary accumulators
ZERO	zero the value in the primary accumulator
BEQZ *label*	branch to *label* if the primary accumulator holds a zero value
BNEZ *label*	branch to *label* if the primary accumulator holds a non-zero value
JUMP *label*	continue execution at location *label*

Notice that all instructions operate on the primary accumulator. Also note that there are no register specifiers in this architecture; *addr* and *label* represent memory addresses. Translate the `rotate` loop to use this ISA. Assume that `x` initially held at address 0x8000, and `n` is initially held at address 0x8004. You are permitted to write temporary variables to any addresses above 0x8000. You should return `x` in the **primary** accumulator.

i) How many bytes is your program?
ii) How many bytes of instructions need to be fetched for `x = 0x01020304` and `n = 7` with your translation?
iii) Assuming 32-bit data values, how many bytes of data memory need to be loaded? Stored?

```
LOAD 0x8000 # (p: ?, s: ?) 1
SWAP # (p: x, s: ?)
LOAD 0x8004 # (p: ?, s: x) 2
```

```
loop:       BEQZ inner_done # (p: n, s: x) 3
            DEC # (p: n, s: x)
            SWAP # (p: n', s: x)

            SHL # (p: x, s: n-1)
            STORE 0x8008 # (p: x<<1, s: n') 4

            LOAD 0x8000 # (p: x<<1, s: n') 5
            SIGN # (p: x, s: n-1)

            OR 0x8008 # (p: msb(x), s: n') 6
            STORE 0x8000 # (p: x', s: n') 7

            SWAP # (p: x', s: n)
            JUMP loop # (p: n', s: x') 8

inner_done: SWAP # (p: n, s: x)
done:       …
# `inner_done` is used to ensure we don't modify the original `done` label in the Q
# adding a `SWAP` instruction to the `done` label is an equivalent solution, as long
as the SWAP instruction is accounted for in following sub parts.
```

i) Bytes in program = 8 * 3 bytes (instructions with addresses/labels) + 7 * 1 byte (instructions *without* addresses/labels) = 31 bytes
ii) Instruction bytes fetched = 7 bytes (prologue) + 7 * 23 bytes (loop iterations) + 3 bytes (final BEQZ) + 1 byte (epilogue) = 172 bytes
iii) Data bytes loaded = 2 * 4 bytes (prologue) + 7 * 2 * 4 bytes (loop iterations) = 64 bytes
Data bytes stored = 7 * 2 * 4 (loop iterations) = 56 bytes

Problem 1.E — Conclusions

In a few sentences, compare the four ISAs you have studied with respect to code size, number of instructions fetched, and data memory traffic. Which one would you choose if you were to build a specialized processor to execute the code in this program, and why?

- Static code size: CISC < RISC < (Stack ~= Accumulator)
- Dynamic code size: CISC < (RISC ~= Stack ~= Accumulator)
- Data memory traffic: (CISC == RISC) < Accumulator < Stack
    - If your code is not well-matched for a stack machine, even accumulator machines can be more efficient
- We would choose CISC if we wanted to minimize bandwidth and memory storage requirements
    - Another ISA choice is also acceptable if the student provides a reasonable explanation

Problem 1.F — Optimization

To get more practice with RISC-V, optimize the code from 1.B so that fewer instructions are executed on average *and* the number of jumps and taken branches is minimized. Your solution should be more optimal than simply translating each x86 instruction. Provide comments and a brief explanation of your optimizations.

Note: It is possible to rotate a string in O(1) time without a loop. For this problem, we want you to use the more inefficient O(N) loop-based solution.

Common optimizations may include:

- Loop unrolling
    - Reduces the loop overhead
- Loop inversion: translating the while loop to a do-while loop
    - Eliminates the unconditional jump

Problem 2: Microprogramming and Bus-based Architectures

In this problem, you will explore microprogramming by writing microcode for the bus-based implementation of the RISC-V machine described in **Handout #1 (Bus-Based RISC-V Implementation)**. Read the instruction fetch microcode in **Table H1-3 of Handout #1**. Make sure that you understand how different types of data and control transfers are achieved by setting the appropriate control signals before attempting this problem.

The final solutions should be as elegant and efficient as possible with respect to the number of microinstructions used.

Problem 2.A — Implementing CIMMDIV

For this problem, you are to implement a new kind of immediate instruction, **CIMMDIV**. The new instruction has the following format:

CIMMDIV rd, rs1, imm

The value in *rs1* is *ceil* divided by imm, and the *result* is stored in *rd*.

```
rd ← rs1 / imm
```

Your CPU's ALU does **not** have support for a ceiling division operation. Fortunately, divide can also be implemented as a loop, as illustrated below:

```
unsigned int ceil_div(unsigned int x, unsigned int y) {
    unsigned int quotient = 0;
    while (x >= y) {
        x -= y;
        quotient += 1;
    }
    if (x > 0) {
        quotient += 1;
    }
    return quotient;
}
```

This loop *is* realizable with the microcode of your CPU.

Fill in Worksheet 2.A with the microcode for CIMMDIV. Use *don't cares* (*) for fields where it is safe to do so, and make sure all of your microinstructions are legal. Your code is permitted to modify *rd* and *rs1* during the execution of this instruction. Comment your code clearly. You may not need every row. Finally, make sure that the instruction fetches the next instruction (i.e., by

doing a microbranch to FETCH0 as discussed above) once the result has been saved to *rd*.Assume rd is initialized to 0.

You may want to consult the microcode found in the micro-coded processor provided in Lab 1, which can be viewed at `chipyard/generators/riscv-sodor/src/main/scala/sodor/rv32_ucode/microcode.scala` for guidance. Warning: While that microcode passes all provided assembly tests and benchmarks, no guarantees to the optimality of that code are assured, and there may still be bugs in the provided implementation.

We will accept any reasonable solution, even if it is different from the one on the next page.

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	0	PC	1	0	0	*	INC_A_4	1	*	0	0	*	0	D	*
. . .																
NOP0:	microbranch back to FETCH0	*	*	0	0	*	*	*	0	*	0	0	*	0	J	FETCH0
CIMMDIV:	A,B <- R[rs1]	0	rs1	0	1	1	1	*	0	*	0	0	*	0	N	*
	R[rd] <- A-B	0	rd	1	0	0	0	SUB	1	*	0	0	*	0	N	*
	B <- imm	0	*	0	0	*	1	*	0	*	0	0	IMMI	1	N	*
LOOP:	A <- R[rs1]	0	rs1	0	1	1	0	*	0	*	0	0	*	0	N	*
	if (A < B) goto CEIL_IF	0	*	0	0	0	0	SLTU	0	*	0	0	*	0	NZ	CEIL_IF
	R[rs1] <- A-B	0	rs1	1	0	0	0	SUB	1	*	0	0	*	0	N	*
	A <- R[rd]	0	rd	0	1	1	0	*	0	*	0	0	*	0	N	*
	R[rd] <- A+1 goto LOOP	0	rd	1	0	0	0	INC_A_1	1	*	0	0	*	0	J	LOOP
CEIL_IF:	if (A == 0) goto FETCH0	0	*	0	0	0	*	CopyA	0	*	0	0	*	0	EZ	FETCH0
	A <- R[rd]	0	rd	0	1	1	*	*	0	*	0	0	*	0	N	*
	R[rd] <- A+1	0	rd	1	0	0	*	INC_A_1	1	*	0	0	*	0	N	*
	goto FETCH0	*	*	0	0	*	*	*	0	*	0	0	*	0	J	FETCH0

Problem 3: 6-Stage Pipeline

In this problem, we consider a modification to the fully bypassed 5-stage RISC-V processor pipeline originally presented in Lecture 3 and further expanded on in Handout #1 (RV32I 5-Stage Pipeline Diagram). Our new processor has a data cache with a two-cycle latency. To accommodate this cache, the memory stage is pipelined into two stages, M1 and M2, as shown in Figure 1-A. Additional bypasses are added to keep the pipeline fully bypassed.

Suppose we are implementing this 6-stage pipeline in a technology in which register file ports are inexpensive, but bypasses are costly. We wish to reduce cost by removing some of the bypass paths, but without increasing CPI. The proposal is for all integer arithmetic instructions to write their results to the register file at the end of the Execute stage, rather than waiting until the Writeback stage. A second register file write port is added for this purpose. Remember that register file writes occur on each rising clock edge, and values can be read in the next clock cycle. The proposed change is shown in Figure 1-B.

In this problem, assume that the only exceptions that can occur in this pipeline are illegal opcodes (detected in the Decode stage) and invalid memory address (detected at the start of the **M2** stage). Additionally, assume that the control logic is optimized to stall only when necessary. ***You may ignore branch and jump instructions in this problem.***

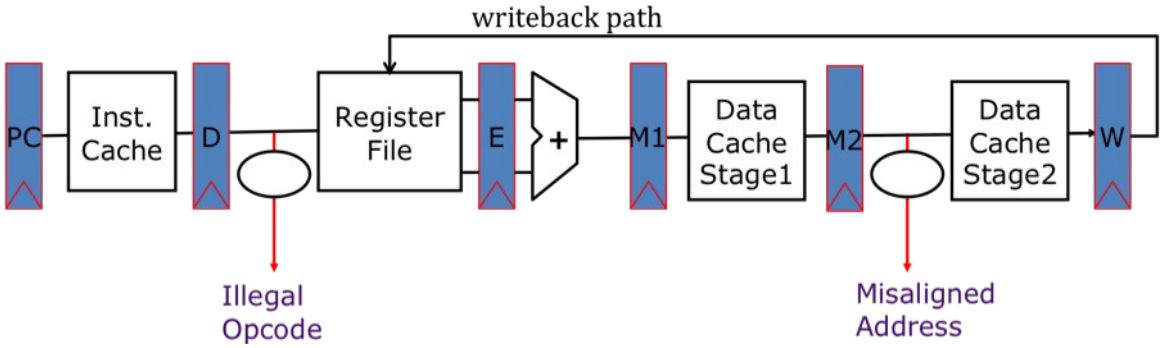

Figure 1-A. 6-stage pipeline. For clarity, bypass paths are not shown. Handout #1 (RV32I 5-Stage Pipeline Diagram) shows the full pipeline diagram.

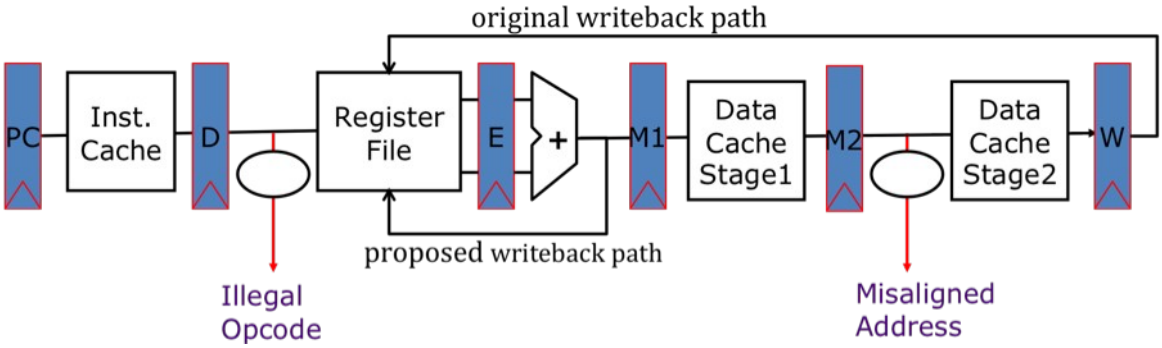

Figure 1-B. 6-stage pipeline with proposed additional write port.

Problem 3.A **Hazards: Second Write Port**

The second write port allows some bypass paths to be removed without adding stalls in the decode stage. Explain how the second write port improves performance by eliminating such stalls and give a short code sequence that would have required an interlock to execute correctly with only a single write port and with the same bypass paths removed.

The second write port improves performance by resolving some RAW hazards earlier than they would be if ALU operations had to wait until writeback to provide their results to subsequent dependent instructions. It would help with the following instruction sequence:

```
add x1, x2, x3
add x4, x5, x6
add x7, x1, x9
```

The important insight is that the second write port cannot resolve data hazards for immediately back-to-back instructions. An arithmetic instruction in the EX stage writes back as it leaves the EX stage; therefore, the bypass path is necessary if the next instruction has a RAW dependency and is allowed to leave the ID stage.

Problem 3.B **Hazards: Bypasses Removed and New Hazards**

After the second write port is added, which bypass paths can be removed in this new pipeline without introducing additional stalls? List each removed bypass individually. Are any new hazards added to the pipeline due to the earlier writeback of arithmetic instructions?

The bypass path from the end of M1 to the end of ID can be removed. (Credit was also given for the bypass path from the beginning of M2 to the beginning of EX, since these are equivalent.) Additionally, ALU results no longer must be bypassed from the end of M2 or the end of WB, but these bypass paths are still used to forward load results to earlier stages.

There are multiple potential WAW hazards that must be appropriately addressed by the control logic. The two instructions writing at the same time must be appropriately prioritized. Also, if an arithmetic instruction is in M1 and a load with the same destination register is in M2, the write of the earlier load can clobber the result of the older instruction, leading to an incorrect architectural state. The control logic needs to be modified to handle these situations by suppressing the writes of older instructions when they conflict with the writes of newer instructions.

Problem 3.C **Precise Exceptions**

Without further modifications, this pipeline may not support precise exceptions. Briefly explain why and provide a minimal code sequence that will result in an imprecise exception.

Illegal address exceptions are not detected until the start of the M2 stage. Since writebacks can occur at the end of the EX stage, it is possible for an arithmetic instruction following a memory access to an illegal address to have written its value back before the exception is detected, resulting in an imprecise exception. For example:

```
lw x1, -1(x0) # address -1 is misaligned
add x2, x3, x4 # x2 will be overwritten, but last instruction
has faulted!
```

Problem 3.D **Precise Exceptions: Implemented using an Interlock**

Describe how precise exceptions can be implemented by adding a new interlock. Provide a minimal code sequence that would engage this interlock. Qualitatively, what is the performance impact of this solution?

Stall any ALU op in the ID stage if the instruction in the EX stage is a load or a store. The instruction sequence above engages this interlock. Loads and stores account for about 1/3 of dynamic instructions. Assuming that the instruction following a load or store is an arithmetic instruction 2/3 of the time, and ignoring the existing load-use delay, this solution will increase the CPI by (1/3)*(2/3) = 2/9. However, only a qualitative explanation was necessary for credit.

Problem 3.E **Precise Exceptions: Implemented using an Extra Read Port**

Suppose you are additionally given the budget to add a new register file *read* port. Propose an alternative solution to implement precise exceptions in this pipeline without requiring any new interlocks.

In addition to writing an arithmetic instruction's destination register in the EX stage, also read its previous value and carry it down the pipeline. If an early writeback occurs before a preceding exception was detected, then the old value of *rd* is preserved in the M1 pipeline register and can be restored to the register file, maintaining precise state.

Note: It is better to read the previous value **as late as possible**, otherwise this read of *rd* might need an extra bypass path for the following instruction sequence:

```
ld x1, 0(x8)
ld x2, -1(x8) # misaligned
addi x1, x1, 4
```

This also depends on the interlocks used to resolve the WAW hazard mentioned in 3.B.

Problem 4: Iron Law of Processor Performance

Problem 4.A — Would it help?

Mark whether the following modifications will cause each of the *first three* categories to **increase, decrease**, or whether the modification will have **no effect**. Explain your reasoning within the box provided. Please state any reasonable assumptions you might make.

For the final column "Overall Performance", mark whether the following modifications **increase**, **decrease**, have **no effect**, or whether the modification will have an **ambiguous** effect. Explain your reasoning. If the modification has an **ambiguous** effect, describe the tradeoff in which it would be a beneficial modification or in which it would a detrimental modification (i.e., as an engineer would you suggest using the modification or not and why?).

		Instructions / Program	Cycles / Instruction	Seconds / Cycle	Overall Performance
a)	Adding a tightly coupled accelerator	Decrease: Entire operations (such as matrix multiplication) can be replaced with a single instruction.	Decrease: Accelerators are designed to complete tasks in less cycles than a traditional datapath.	Ambiguous: This depends on how the accelerator is implemented. For example, accelerator logic may contribute to the critical path, or it may be completely in parallel with the rest of the design.	Increase: If the accelerator is for a common operation, dedicated hardware will outperform a general purpose datapath, improving CPI.
b)	Reduce number of registers in the ISA	Increase: Values will more frequently be spilled to the stack, increasing number of loads and stores.	Increase: More loads followed by dependent instructions will cause more stalls. Memory latency is hard to schedule around.	Decrease: Fewer registers lead to shorter register file access time.	Ambiguous: If the program uses few registers and thus spills rarely to memory, the faster reg. access times may win out. Also, your instructions may be able to be shorter, improving amongst other things code density.

c)	Adding 16-bit versions of the most common instructions in RISC-V (normally 32 bits in length) to the ISA (i.e., make RISC-V a variable-length ISA)	No effect: The actual **number** of instructions is unchanged.	Decrease: Since code size has shrunk, there will be fewer instruction cache (I$) misses and less time spent waiting to fetch.	Increase: Decode becomes more complex with more formats, and instruction fetch must deal with misalignment.	Ambiguous: The main advantage is smaller code size, which can improve I$ hit rates and save on fetch energy (get more instructions per fetch). However, the more complex decode can offset these gains.
d)	For a given CISC ISA, changing the implementation of the micro-architecture from a bus-based datapath with a microcode engine (similar to Problem 2) to a pipelined RISC datapath with a CISC-to-RISC decoder on the frontend. The CISC-to-RISC translation results in each CISC instruction being translated to 1 or more coarser RISCV-like instructions v. much finer µops.	No effect: Since the ISA is not changing, the binary does not change, and thus there is no change in Inst/Program.	Decrease: Microcoded machines take several clock cycles to execute an instruction, while the RISC pipeline should have a CPI near I (thanks to pipelining and translation to much coarser RISCV-like instructions)	No effect: The amount of work done in one pipeline stage and one microcode cycle are about the same. ALSO ACCEPT: Increase: The RISC pipeline introduces longer control paths and adds bypasses, which are likely to be on the critical path.	Improve: The decrease in CPI from the RISC pipeline far outweighs any critical path overhead of hardwired control logic.

Problem 4. B **Design a World Class CPU**

Imagine you are designing a world class CPU, and you want to optimize CPI. Provide *three* unique (as in not from part A) optimizations you would make to your datapath, and explain how it benefits CPI. Feel free to be creative! Just provide a justification for your choice.

	Optimization	Impact
a)	Branch prediction	Instead of stalling, your datapath can speculate on where a branching instruction will go. If implemented effectively, this will dramatically improve CPI.
b)	Multi-level caching	Data can be fetched from the cache instead of retrieving from memory. This is much faster as caches are smaller (faster lookup) and physically closer to the CPU (speed of light / networking complexity). This means lower CPI as data can be written and read faster.
c)	Multithreading	When you can't run an instruction from a given thread because resources are taken, you can run an instruction from a different thread, improving CPI since there's less stalling.

Feedback (Required)

How long did this assignment take you? Feel free to include any other feedback you have!