CS152/252A
Fall 2026

ISAs, Microprogramming, and Pipelining
Problem Set #1

Due: September 13 @ 11:59:59 PM PST

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. For Problems 1 and 2, upload your completed PDF as usual. **Problems 3, 4, and 5 must be answered directly in the Gradescope free-response fields; do not include answers to Problems 3, 4, or 5 in your uploaded PDF.** For the PDF portion, box all solutions that don't involve 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: _____

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 1.1 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 Problems 1.1–1.4, 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 `<R_SUBSCRIPT>`.

(Continued on the next page)

Instruction	Operation	Length
`mov R_SRC, R_DEST`	`<R_DEST> = <R_SRC>`	2 bytes
`test R_SRC1, R_SRC2`	`temp = <R_SRC1> & <R_SRC2>` Set flags based on value of `temp`	2 bytes
`dec R_DEST`	`<R_DEST> = <R_DEST> - 1`	1 byte
`shl $imm8, R_DEST`	`<R_DEST> = <R_DEST> << imm8`	2 bytes
`shr $imm8, R_DEST`	`<R_DEST> = <R_DEST> >> 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?

Problem 1.2 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 instruction.) 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` `jz done`		
`mov %eax,%ebx` `shr $31,%ebx`		
`shl $1,%eax`		
`or %ebx,%eax`		
`dec %ecx`		
`jmp loop`		
`...`		`...`

(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?

Problem 1.3 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 1.1 and 1.2. 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 stack registers?

Problem 1.4 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.

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?

Problem 1.5 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?

Problem 1.6 Optimization

To get more practice with RISC-V, optimize the code from 1.2 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.

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 2.1 — Implementing CCOMPACT

For this problem, you are to implement a new instruction, `CCOMPACT`. The instruction has the following format:

```
CCOMPACT rd, rs1, imm
```

`rs1` contains a pointer to the beginning of a source array, `rd` contains a pointer to the beginning of a destination array, and `imm` specifies the number of 32-bit elements in the source array.

`CCOMPACT` copies every nonzero element from the source array into the destination array while preserving the original order. Zero-valued elements are skipped.

For example, if the source array contains

```
[3, 0, 1, 0, 4, 2]
```

and `imm = 6`, then after executing `CCOMPACT`, the destination array should begin with

```
[3, 1, 4, 2]
```

The instruction can be described by the following pseudocode:

```
unsigned int count = imm;

while (count != 0) {
    unsigned int x = *rs1;

    if (x != 0) {
        *rd = x;
        rd += 4;
    }

    rs1 += 4;
    count -= 1;
}
```

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

Worksheet 2.A

PROBLEMS 3, 4, and 5 ARE ONLINE-ONLY

Submit all answers directly in the Gradescope free-response fields.

Do not include answers to Problems 3, 4, or 5 in your uploaded PDF.

Problem 3: 6-Stage Pipeline

Submission note: Problems 3 and 4 must be answered directly in the Gradescope free-response fields. Do not include answers to Problems 3 or 4 in your uploaded PDF.

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.*

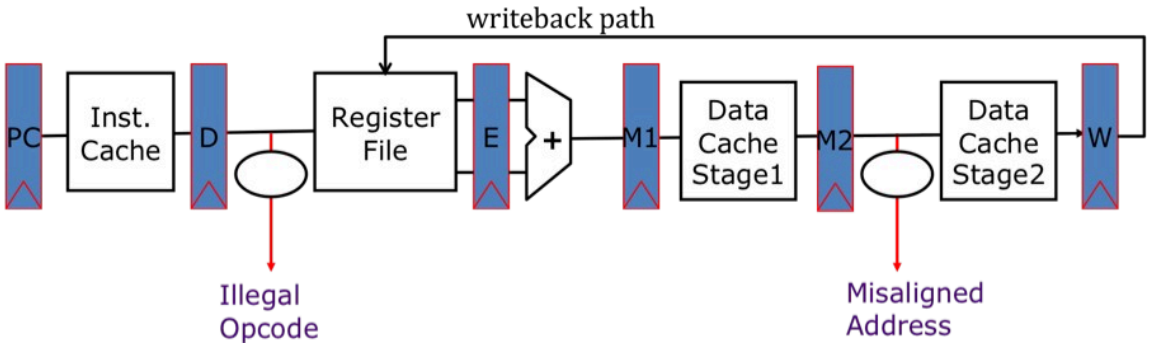

Figure 1-A. 6-stage pipeline. For clarity, bypass paths are not shown. Handout 1 (RV32I 5-Stage Pipeline Diagram) shows the full pipeline diagram.

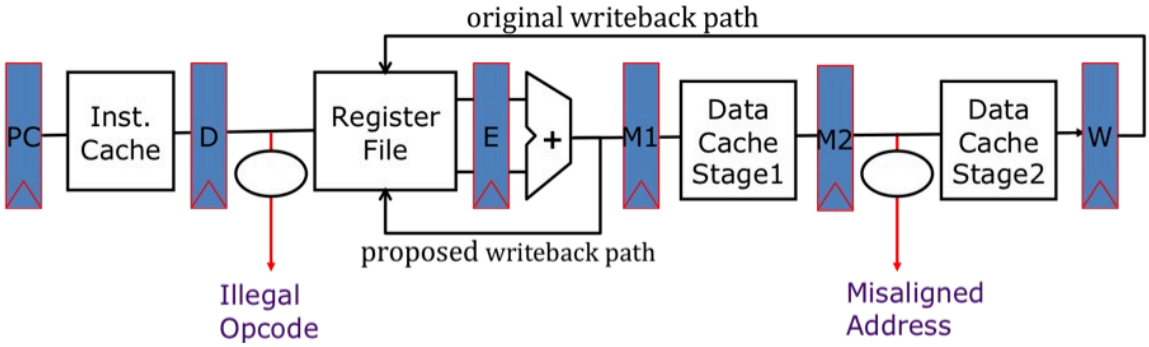

Figure 1-B. 6-stage pipeline with proposed additional write port.

Problem 3.1 — 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.

Problem 3.2 — 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?

Problem 3.3 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.

Problem 3.4 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?

Problem 3.5 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.

Problem 4: Iron Law of Processor Performance

Problem 4.1 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 be 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)	You expose a division operation in the ISA and add the unpipelined division functionality to the ALU, instead of using a loop-based division algorithm.				
B)	You design a compiler that keeps hot values in registers to minimize touching memory.				

		Instructions / Program	**Cycles / Instruction**	**Seconds / Cycle**	**Overall Performance**
C)	You add a specialized N-cycle floating-point unit parallel to your single-cycle ALU in a 5-stage RISC-V pipeline. Assume the pipeline can handle completing instructions (writing to register file) out of program order while maintaining correctness. Also assume there are no faults or exceptions that occur. Compare the performance to a 5-stage RISC-V pipeline that has the same N-cycle floating-point operations built into the ALU where all other operations take 1 cycle.				
D)	For a given CISC ISA, changing the implementation of the microarchitecture 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 RISC-V-like instructions vs. much finer μops.				

Problem 4.2 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 Problem 4.1) 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**
1)		
2)		
3)		

Problem 5: Feedback

(Required)

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