CS152 Computer Architecture and Engineering

ISAs, Microprogramming and Pipelining

Assigned 01/27/2021 Problem Set #1, Version (1.2) *Due February 8*

http://inst.eecs.berkeley.edu/~cs152/sp21

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 **11:59pm PST** on the specified due date. Refer to Piazza for the entry code to join the CS152 Gradescope. Late submissions will not be accepted, except for extreme circumstances and with prior arrangement.

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 computes the number of bits that are set in a value (known as the *population count*). The method shown here is faster than the naïve approach of iterating through every bit.[1]

```
unsigned int popcount(unsigned int x) {
  unsigned int n;
  for (n = 0; x != 0; n++) {
    x &= x - 1; // Clear least significant 1 bit
  }
  return n;
}
```

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.

```
          movl   $0,%ecx
loop:     test   %eax,%eax
          jz     done
          mov    %eax,%ebx
          dec    %ebx
          and    %ebx,%eax
          inc    %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 <$R_{SUBSCRIPT}$>.

Instruction	Operation	Length
`movl $imm32,` R_{DEST}	<R_{DEST}> `= imm32`	6 bytes
`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

[1] This C version originates from Kernighan & Ritchie (1988), although the technique appears to have been first published by Wegner (1960) for the IBM 704.

`inc` R_{DEST}	<R_{DEST}> = <R_{DEST}> + 1	2 bytes
`dec` R_{DEST}	<R_{DEST}> = <R_{DEST}> - 1	2 bytes
`and` 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, 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

How many bytes is the program? For the above x86 assembly code, how many bytes of instructions need to be fetched if `x = 0xABCD1234`? Assuming 32-bit data values, how many bytes of data memory need to be loaded? Stored?

Problem 1.B RISC

Translate each of the x86 instructions in the following table into 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. 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 in the class website, resources page.

x86 instruction	**Label**	**RISC-V instruction sequence**
`movl   $0,%ecx`		
`test   %eax,%eax`		
`jz     done`		
`mov    %eax,%ebx`		
`dec    %ebx`		
`and    %ebx,%eax`		
`inc    %ecx`		
`jmp    loop`		
…	`done:`	…

How many bytes is the RISC-V program using your direct translation? How many bytes of RISC-V instructions need to be fetched for `x = 0xABCD1234` with your direct translation? Assuming 32-bit data values, how many bytes of data memory need to be loaded? Stored?

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*
AND	pop two values from the stack; AND them; push result onto stack
INC	pop value from top of stack; increment value by 1; push result onto stack
DEC	pop value from top of stack; decrement value by 1; push result onto stack
ZERO	zero the value at top of 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 `popcount` loop to the stack ISA. For uniformity, please use the same control flow as in parts (a) and (b). Assume that when we reach the loop, `x` is at the top of the stack. At the end of the loop, the stack should contain only `n` at the top. Assume that memory starting at address 0x8000 (to fit within a 2-byte address specifier) is available to use as temporary storage.

How many bytes is your program? How many bytes of instructions need to be fetched for `x = 0xABCD1234` with your translation? Assuming 32-bit data values, how many bytes of data memory need to be loaded? Stored? Would the number of bytes loaded and stored change if the stack could fit 8 entries in registers?

Problem 1.D Accumulator

In an accumulator ISA, one operand is implicitly a specific register (the same for all instructions), called the accumulator. 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*
AND *addr*	AND the value at *addr* with the value in the primary accumulator
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 `popcount` loop to use this ISA. Assume that `x` initially held at address 0x8000. You should return `n` in the **primary** accumulator.

How many bytes is your program? How many bytes of instructions need to be fetched for `x = 0xABCD12324` with your translation? Assuming 32-bit data values, how many bytes of data memory need to be loaded? Stored?

Problem 1.E Conclusions

In just 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.F Optimization

To get more practice with RISC-V, optimize the code from part B so that fewer dynamic instructions are executed on average and the frequency of taken branches is minimized. There are solutions more efficient than simply translating each individual x86 instruction as you did in part (b). Your solution should contain commented assembly code, a brief explanation of your optimizations, and a short analysis of the savings you obtained.

Problem 2: Microprogramming and Bus-based Architectures

In this problem, we 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 solution should be as elegant and efficient as possible with respect to the number of microinstructions used.

Problem 2.A — Implementing SUBLEQ

For this problem, you are to implement a new kind of conditional branch instruction, **Sub**tract and Branch if **L**ess Than or **E**qual to **Z**ero. The new instruction has the following format:

SUBLEQ rd, rs1, rs2

SUBLEQ[2] performs the following operation: The memory word at the address in *rs1* is subtracted from the word at the address in *rd*, and the result is stored back to the address in *rd*. Then, if the result is less than or equal to 0, it branches to the address in *rs2*.

```
M[rd] ← M[rd] – M[rs1]
if (M[rd] ≤ 0)
     branch to rs2
```

Fill in Worksheet 2.A with the microcode for SUBLEQ. Use *don't cares* (*) for fields where it is safe to use don't cares. Study the hardware description well, and make sure all your microinstructions are legal.

Please comment your code clearly. If the pseudo-code for a line does not fit in the space provided, or if you have additional comments, you may write in the margins so long as you do it neatly. Your code should exhibit "clean" behavior and not modify *rd*, *rs1*, *rs2*, or other general-purpose architectural registers while executing the instruction.

Finally, make sure that the instruction fetches the next instruction (i.e., by doing a microbranch to FETCH0 as discussed above) if it does not branch to *rs2*.

You may want to consult the microcode found in the micro-coded processor provided in Lab1, which can be viewed at `lab1/generators/riscv-sodor/src/main/scala/rv32_ucode/microcode.scala` for guidance. <u>Warning</u>: 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.

[2] SUBLEQ is of some theoretical interest since it is an example of a "one-instruction set computer" – a computer which implements only a single instruction but is nevertheless Turing-complete (given infinite memory and time).

State	PseudoCode	ldIR	Reg Sel	Reg Wr	en Reg	ldA	ldB	ALUOp	en ALU	ld MA	Mem Wr	en Mem	Imm Sel	en Imm	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
SUBLEQ0:																

Worksheet 2.A

Problem 2.B Implementing Character Count

In this question we ask you to implement a useful string instruction to count the occurrences of a given character in a string (STRCHRCT). This instruction has the same format as other arithmetic (R-type) instructions in RISC-V:

STRCHRCT rd, rs1, rs2

The STRCHRCT instruction takes a pointer to a string in memory (*rs1*) and a character to match against (*rs2*), and it returns in register *rd* the number of times that the given character appears in the string. Your code is permitted to modify register *rs1* during the execution of this instruction.

For this problem, think of a string as an array of *4-byte* words (each character consisting of a single 4-byte word) with the last element being zero (the string is "null terminated").

Your task is to fill out Worksheet 2.B for STRCHRCT instruction. You should try to optimize your implementation for the minimal number of cycles necessary and for which signals can be set to don't-cares.

State	PseudoCode	ldIR	Reg Sel	Reg Wr	en Reg	ldA	ldB	ALUOp	en ALU	ld MA	Mem Wr	en Mem	Imm Sel	en Imm	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
STRCHRCT0:																

Problem 2.C **Instruction Execution Times**

How many cycles does it take to execute the following instructions on the microcoded RISC-V implementation? Use the states and control signals from Handout #1 (or Lab 1, in `lab1/generators/riscv-sodor/src/main/scala/rv32_ucode/microcode.scala`) and assume that memory does not assert its busy signal.

Instruction	Cycles
`SUB  x3,x2,x1`	
`ANDI  x2,x1,#4`	
`LW   x1,0(x2)`	
`BNE  x1,x2,label  #(x1 == x2)`	
`BNE  x1,x2,label  #(x1 != x2)`	
`BEQ  x1,x2,label  #(x1 != x2)`	
`BEQ  x1,x2,label  #(x1 == x2)`	
`J    label`	
`JAL  label`	
`JALR x1`	
`AUIPC x1, #128`	

Which instruction takes the most cycles to execute? Which instruction takes the fewest cycles to execute?

Problem 3: 6-Stage Pipeline

In this problem, we consider a modification to the fully bypassed 5-stage RISC-V processor pipeline presented in Lecture 3. 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.***

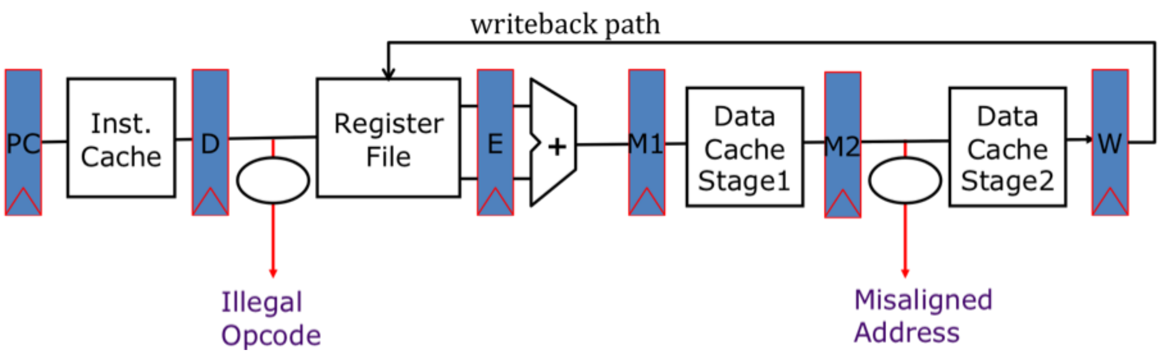

Figure 1-A. 6-stage pipeline. For clarity, bypass paths are not shown.

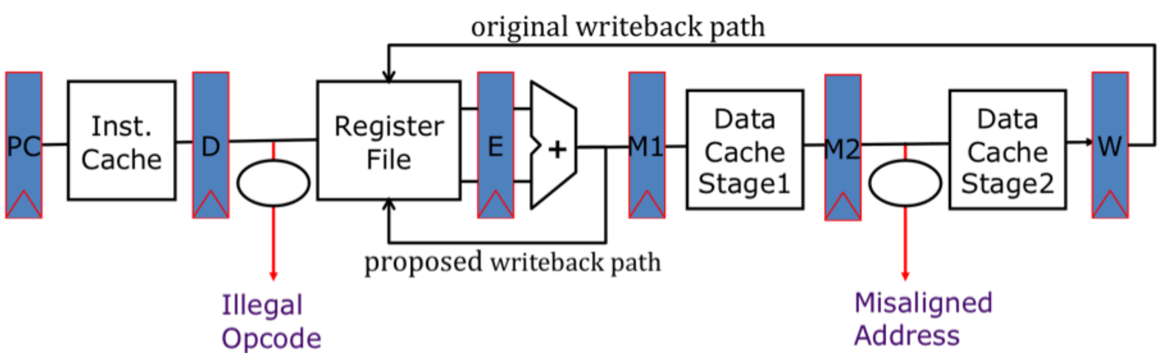

Figure 1-B. 6-stage pipeline with proposed additional write port.

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.

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?

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.

Problem 3.D — Precise Exceptions: Implemented using a 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.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.

Problem 4: CISC vs RISC

For each of the following questions, select either *CISC* or *RISC*, depending on which ISA you feel would be best suited for the situation described. Also, briefly *explain your reasoning*.

Problem 4.A	**Lack of Good Compilers I**

Assume that compiler technology is poor, and therefore your users are far more apt to write all of their code in assembly. A _____ ISA would be best appreciated by these programmers.

CISC **RISC**

Problem 4.B	**Lack of Good Compilers II**

You desire to make compilers better at targeting your *yet-to-be-designed* machine. Therefore, you choose a _____ ISA, as it would be easiest for a compiler to target, thus allowing your users to write code in higher-level languages like C and Fortran and raise their productivity.

CISC **RISC**

Problem 4.C **Fast Logic, Slow Memory**

Assume that CPU logic is fast, *very* fast, while instruction fetch accesses are at least 10x slower (suppose you are the lead architect of the “709”). Which ISA style do you choose as a best match for the hardware’s limitations?

CISC **RISC**

Problem 4.D **Higher Performance(?)**

Starting with a clean slate in the year 2021 (area/logic/memory is cheap), you think that a _____ ISA that would lend itself best to a very high performance processor (e.g., high frequency, highly pipelined).

CISC **RISC**

Problem 5: Iron Law of Processor Performance

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.

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 branch delay slot				
b)	Adding a complex instruction				
c)	Reduce number of registers in the ISA				
d)	Improving memory access speed				

e)	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)				
f)	For a given CISC ISA, changing the implementation of the micro-architecture from a microcoded engine to a RISC pipeline (with a CISC-to-RISC decoder on the frontend)				