CS152 Computer Architecture and Engineering

Solution

ISAs, Microprogramming and Pipelining
Problem Set #1, Version (1.2)

Assigned 01/27/2021 *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.

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?

- How many bytes in program: 6 + 7*2 = 20
- How many instruction bytes fetched:
  - Number of loop iterations is determined by how many bits are set to 1 in `x` (15 for `x = 0xABCD1234`)
  - 6 in prologue (1 time)
  - 4 in loop condition (16 times)
  - 10 in loop body (15 times)
  - 6 + 16*4 + 15*10 = 220
- How many data bytes loaded/stored: 0 (no loads or stores)

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`		`addi x2, x0, 0`
`test    %eax,%eax`	`loop:`	`addi x0, x0, 0 # nop (can be omitted)`
`jz      done`		`beq x1, x0, done`
`mov     %eax,%ebx`		`addi x6, x1, 0 # mv`
`dec     %ebx`		`addi x6, x6, -1`
`and     %ebx,%eax`		`and x1, x1, x6`
`inc     %ecx`		`addi x2, x2, 1`
`jmp     loop`		`jal x0, 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?

- How many bytes in program: 8*4 = 32 (or 7*4 = 28 if omitting the explicit NOP)
- How many instruction bytes fetched:
  - 4 in prologue (1 time)
  - 8 (or 4 without NOP) in loop condition (16 times)
  - 20 in loop body (15 times)
  - With explicit NOP: 4 + 16*8 + 15*20 = 432 bytes
  - Without explicit NOP: 4 + 16*4 + 15*20 = 368 bytes
- How many data bytes loaded/stored: 0 (no loads or stores)

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?

```
Label      Instruction           Comment
           …                     # x
           pop 0x8000            # <empty>
           zero                  # n=0
loop:      push 0x8000           # n, x
           beqz done             # n
           inc                   # n'
           push 0x8000           # n', x
           push 0x8000           # n', x, x
```

```
            dec                     # n’, x, x-1
            and                     # n’, x’
            pop 0x8000              # n’
            jump loop               # n’
done:       …
```

Note: The x’ and n’ symbols denote (i+1)-th values in the sequence. The leftmost entry in each comment represents the top of the stack.

- How many bytes in program: 4*1 + 7*3 = 25
- How many instruction bytes fetched:
  - 4 in prologue (1 time)
  - 6 in loop condition (16 times)
  - 15 in loop body (15 times)
  - 4 + 16*6 + 15*15 = 325
- How many data bytes loaded:
  - 4 in loop condition (16 times)
  - 8 explicitly in loop body (15 times)
  - 4 implicitly in loop body, as n’ is reloaded into the bottom stack register after the AND instruction shortens the stack to two entries again (15 times)
  - 16*4 + 15*8 + 15*4 = 244
- How many data bytes stored:
  - 4 in prologue (1 time)
  - 4 explicitly in loop body (15 times)
  - 4 implicitly in loop body, as n’ is spilled to memory when the stack grows to more than two entries (15 times)
  - 4 + 15*4 + 15*4 = 124
- With more stack registers, it is possible to avoid spilling `n` to memory, saving one implicit load and one store per iteration for 15*4 = 60 fewer bytes loaded and 60 fewer bytes stored. However, pushing `x` onto the stack twice still requires using memory accesses. (Actual stack machines typically have a stack manipulation instruction to duplicate the topmost entry.)

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?

```
Label       Instruction           Comment
            zero                  # A = 0
            swap                  # A = ?, B = 0
            load 0x8000           # A = x
            beqz done             # Skip if x == 0
loop:       dec                   # A = x-1
            and 0x8000            # A = x & (x-1)
            store 0x8000          # x' = A
            swap                  # A = n, B = x'
            inc                   # A = n++
            swap                  # A = x', B = n'
            bnez loop             # Loop while x != 0
done:       swap                  # A = n
```

…

- How many bytes in program: 7 + 5*3 = 22
- How many instruction bytes fetched:
    - 8 in prologue (1 time)
    - 13 in loop body (15 times)
    - 1 in epilogue (1 time)
    - 8 + 15*13 + 1 = 204
- How many data bytes loaded:
    - 4 in prologue (1 time)
    - 4 in loop body (15 times)
    - 4 + 15*4 = 64
- How many data bytes stored: 4*15 = 60

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?

- CISC < Accumulator < Stack < RISC for both static and dynamic code size
  - RISC uses fewer instructions but has worse encoding density
- (RISC ≈ CISC) < Accumulator < Stack for data memory traffic
- If the code is not very well-matched for a stack machine, even accumulator machines can be more efficient.

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.

Some simple optimizations are:

- Loop inversion (transforming the `while` into a `do-while` loop) to eliminate the unconditional jump
- Loop unrolling to approximately halve the number of taken branches as well as the number of `addi` instructions used to increment `n`

Label	**Instruction**	**Comment**
	`addi x2, x0, 0`	`# n = 0`
	`beq x1, x0, done`	`# Skip if x == 0`
`loop:`	`addi x6, x1, -1`	`# x-1`
	`and x1, x1, x6`	`# x &= x-1`
	`beq x1, x0, tail`	`# Break if x == 0`
	`addi x6, x1, -1`	`# x-1`
	`and x1, x1, x6`	`# x &= x-1`
	`addi x2, x2, 2`	`# n += 2`
	`bne x1, x0, loop`	`# Loop while x != 0`
`done:`	…	
`tail:`	`addi x2, x2, 1`	`# Handle case of odd n`
	`jal x0, done`	

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:	MA <- R[rs1]	0	rs1	0	1	*	*	*	0	1	0	0	*	0	N	
	B <- Mem	0	*	0	0	*	1	*	0	0	0	1	*	0	S	
	MA <- R[rd]	0	rd	0	1	*	0	*	0	1	0	0	*	0	N	
	A <- Mem	0	*	0	0	1	0	*	0	0	0	1	*	0	S	
	B <- A - B	0	0	0	1	*	1	SUB	1	0	0	0	*	0	N	
	Mem, A <- B	0	*	0	0	1	0	COPY_B	1	0	1	0	*	0	S	
	A <- A - B (== 0)	0	*	0	0	1	0	SUB	1	*	0	0	*	0	N	
	if (A < B) uBr to FETCH0; A <- R[rs2]	0	rs2	0	1	1	*	SLT	0	*	0	0	*	0	NZ	FETCH0
	PC <- A; uBr to FETCH0	*	PC	1	0	*	*	COPY_A	1	*	0	0	*	0	J	FETCH0

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.

The problem description is ambiguous as to whether the NUL terminator is included or excluded from the count when *rs2* = 0, so both approaches were accepted. The reference solution follows the latter interpretation.

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
STRCHRCT 0:	A, B, MA <- R[rs1]	0	rs1	0	1	1	1	*	0	1	0	0	*	0	N	*
	R[rd] <- A - B (== 0)	0	rd	1	0	*	*	SUB	1	0	0	0	*	0	N	*
STRCHRCT 1:	A <- Mem	0	*	0	0	1	0	*	0	0	0	1	*	0	S	*
	B <- R[rs2]; if (A == 0) uBr to FETCH0	0	rs2	0	1	0	1	COPY_A	0	*	0	0	*	0	EZ	FETCH0
	if (A - B != 0) uBr to STRCHRCT2; A <- R[rd]	0	rd	0	1	1	*	SUB	0	*	0	0	*	0	NZ	STRCHRCT2
	R[rd] <- A + 1	0	rd	1	0	*	*	INC_A_1	1	*	0	0	*	0	N	*
STRCHRCT 2:	A <- R[rs1]	0	rs1	0	1	1	*	*	0	*	0	0	*	0	N	
	MA, R[rs1] <- A + 4; uBr to STRCHRCT1	0	rs1	1	0	*	*	INC_A_4	1	1	0	0	*	0	J	STRCHRCT1

Worksheet 2.B

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.

The answers below are derived from the microcoded processor described in the lab.

Instruction	Cycles
`SUB x3,x2,x1`	3 + 3 = 6
`ANDI x2,x1,#4`	3 + 3 = 6
`LW x1,0(x2)`	3 + 5 = 8
`BNE x1,x2,label #(x1 == x2)`	3 + 4 = 7
`BNE x1,x2,label #(x1 != x2)`	3 + 3 + 4 = 10
`BEQ x1,x2,label #(x1 != x2)`	3 + 4 = 7
`BEQ x1,x2,label #(x1 != x2)`	3 + 3 + 4 = 10
`J label`	3 + 6 = 9
`JAL label`	3 + 6 = 9
`JALR x1`	3 + 6 = 9
`AUIPC x1, #128`	3 + 4 = 7

The answers below are derived from the microcoded processor from the handout.

Instruction	Cycles	Summary (not including fetch and dispatch)
`SUB x3,x2,x1`	3 + 3 = 6	1) A ← R[x2]; 2) B ← R[x1]; 3) R[x3] ← A - B
`ANDI x2,x1,#4`	3 + 3 = 6	1) A ← R[x1]; 2) B ← Imm; 3) R[x2] ← A &[1] B
`LW x1,0(x2)`	3 + 4 = 7 or 3 + 5 = 8	1) A ← R[x2]; 2) B ← Imm; 3) MA ← A + B; 4) R[x1] ← Mem; 5) µBr J FETCH0[2]
`BNE x1,x2,label #(x1 == x2)`	3 + 3 = 6	1) A ← R[x1]; 2) B ← R[x2]; 3) A - B; µBr EZ FETCH0; B ← Imm[4]
`BNE x1,x2,label #(x1 != x2)`	3 + 3 + 4 = 9	4) A ← PC; 5) A ← A - 4; 6) PC ← A + B
`BEQ x1,x2,label #(x1 != x2)`	3 + 3 = 6	1) A ← R[x1]; 2) B ← R[x2]; 3) A - B; µBr NZ FETCH0; B ← Imm[4]
`BEQ x1,x2,label #(x1 == x2)`	3 + 3 + 4 = 9	4) A ← PC; 5) A ← A - 4; 6) PC ← A + B
`J label`	3 + 3 = 6	1) R[rd][5] ← PC; 2) B ← Imm; 3) PC ← A[3] + B
`JAL label`	3 + 3 = 6	Same as above
`JALR x1`	3 + 4 = 7	1) R[rd] ← PC; 2) A ← R[x1]; 3) B ← Imm; 4) PC ← A + B
`AUIPC x1, #128`	3 + 2 = 5	1) B ← Imm; 2) R[x1] ← A[3] +[1] B

[0] Terminal microinstructions are assumed to have a μBr J back to FETCH0 unless stated otherwise.
[1] These operations were not provided in the handout, but it is reasonable to assume they can occur in a single ALU op.
[2] The wording of the question is ambiguous as to whether "memory does not assert its busy signal" implies that the machine has guaranteed single-cycle memory versus you can exclude stall cycles for purposes of the cycle accounting. The former would permit the μBr to occur in parallel with the load. The safer, intended answer assumes the latter case and separates the μBr and the memory op.
[3] The A register contains PC after fetch, whereas PC is speculatively set to PC+4. Thus, we reuse A to speed up AUIPC and JAL instructions, eliding the need to load the PC into A and decrement it by 4. (Conversely, this cannot be avoided in taken conditional branches.)
[4] Speculatively load the branch offset into B to shave a cycle off a taken conditional branch. This will just be discarded in fetch if the branch is not taken.
[5] J is encoded as JAL with *rd*=x0.

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

Most cycles: Taken branch (BEQ, BNE)
Fewest cycles: Arithmetic operations (SUB, ANDI, etc.)

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.***

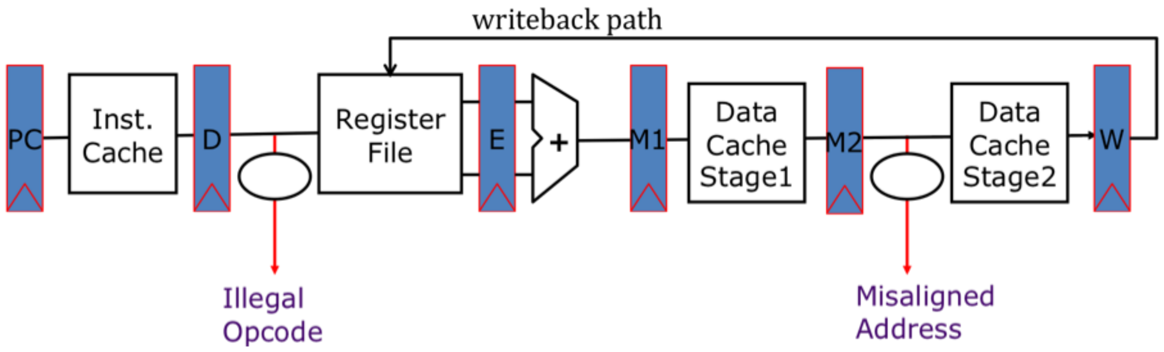

Figure 1-A. 6-stage pipeline. For clarity, bypass paths are not shown.

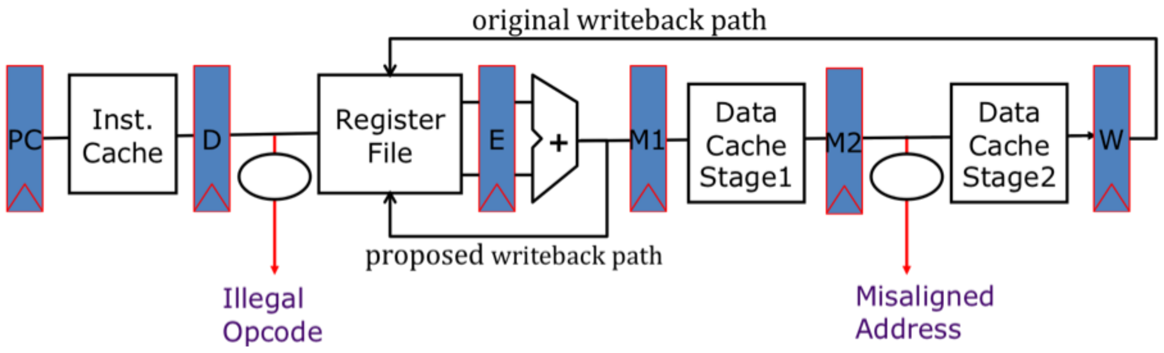

Figure 1-B. 6-stage pipeline with proposed additional write port.

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.

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 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?

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: 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**

CISC ISAs provided more complex, higher-level instructions such as string manipulation instructions and special addressing modes convenient for indexing tables (say for your company's payroll application). Two example CISC instructions: "DBcc: Test Condition, Decrement, and Branch" and "CMP2: Compare Register against Upper and Lower Bounds". This made life easy if you stared at assembly all day and could not hide behind convenient software abstractions/subroutines!

OR

CISC **RISC**

A streamlined RISC ISA is far simpler for an assembly programmer to fully understand and reason about than all the idiosyncrasies that CISC ISAs tend to have, such as the variety of complex instructions for narrow use cases and the myriad addressing modes.

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**

Compilers had difficulty targeting CISC ISAs in part because the complicated instructions have many difficult and hard to analyze side-effects. A load-store/register-register RISC ISA which limits side-effects to a single register or memory location per instruction is relatively easy for a compiler to understand, analyze, and schedule code for.

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**

When instruction fetch takes 10x longer than a CPU logic operation, you are going to want to push as much compute as you can into each instruction! Certain especially complex CISC instructions can encode tens, even hundreds of equivalent RISC instructions. For example, a CISC instruction which performs a single expensive, multi-cycle string routine in hardware would be considerably faster than even an optimized RISC implementation that would need a loop with a series of loads, stores, and arithmetic instructions in the loop body.

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**

Because RISC instructions tend to have simple, easy to analyze side-effects, they lend themselves more readily to pipelined micro-architectures which dynamically check for dependencies between instructions and interlock or bypass when dependencies arise. And because little work needs to be performed in each stage, the pipeline can be clocked at very high frequencies.

This advantage is evident in modern micro-architectures of old CISC ISAs: The frontend of the processor typically has a decoder which translates CISC instructions (e.g., x86 instructions) into RISC "micro-ops", which a high-performance pipeline can then dynamically schedule for maximum performance.

For these CISC architectures such as x86 and IBM S/360, they are still around for legacy reasons. But if you had a chance at a clean slate, you would probably prefer a clean RISC implementation with a direct translation to the micro-architecture instead of using area and power on a CISC decoder front-end (not to mention the additional complexity forced on your memory system to handle the odd CISC addressing modes).

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	Increase: NOPs must be inserted when the branch delay slot cannot be usefully filled	Decrease: Some control hazards are eliminated; also additional NOPs execute quickly because they have no data hazards.	No effect: will not meaningfully change the pipeline. ALSO ACCEPT: Decrease because no branch kill	Ambiguous: Depends on the program and how often the delay slot can be filled with useful work
b)	Adding a complex instruction	Decrease: if the added instruction can replace a sequence of instructions.	Increase: implementing the instruction can means adding stages or making stages have more complex control logic.	Increase: more control logic and interlocks will often increase the critical path. ALSO ACCEPT: No effect	Ambiguous: if the program can take advantage of the new instruction, it can be worth the cost. This is a hard decision for an ISA designer to make!
c)	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 means 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 a

d)	Improving memory access speed	No effect: since instructions make no assumption about memory speed.	Decrease: programs will spend less time stalled waiting for memory	Decrease: if memory access is on the critical path or memory was 1 cycle. ALSO ACCEPT: No effect: if memory is pipelined and just takes less cycles.	Improve: improving memory access time will increase performance of the whole system.
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)	No effect: The actual **number** of instructions is unchanged.	Decrease: since code size has shrunk, there will be fewer instruction cache (I$) misses and less time spent waiting to fetch	Increase: decode becomes more complex with more formats, and instruction fetch has to deal with misalignment.	Ambiguous: the main advantage is smaller code size, which can improve I$ hit rates and save on fetch energy (get more instructions per fetch). However, the more complex decode can offset these gains.
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)	No effect: Since the ISA is not changing, the binary does not change, and thus there is no change to Inst/Program.	Decrease: Microcoded machines take several clock cycles to execute an instruction, while the RISC pipeline should have a CPI near 1 (thanks to pipelining).	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.	The decrease in CPI from pipeline far outweighs any critical path overhead of hardwired control logic.