CS152 Computer Architecture and Engineering

ISAs, Microprogramming and Pipelining

Assigned 1/30/2019 — Problem Set #1, Version (1.1) — *Due February 11*

http://inst.eecs.berkeley.edu/~cs152/sp19

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 you are unlikely to succeed on the exam or midterms! We will distribute solutions to the problem sets on the day the problem sets are due to give you feedback. Homework assignments are due at the beginning of class on the due date. Late homework 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:

```
unsigned int power(unsigned int x, unsigned int y) {
  unsigned int result = 1;
  for ( ; y != 0; y--){
    result = result * x;
  }
  return result;
}
```

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 `%ecx` contains `y`, and register `%eax` contains `result`, and register `%ebx` contains `x`. Throughout parts (a-d), we will ignore what happens in the `done` label and return statement.

```
          movl   $1, %eax
loop:     test   %ecx,%ecx
          jz     done
          mul    %ebx
          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
`movl $imm32,` R_{DEST}	$\langle R_{DEST} \rangle$ = imm32	6 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
`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
`mul` R_{SRC}	<%eax> = <%eax> * R_{SRC} *Note: %edx and flags are used to handle the extra width of the resulting value; they are ignored here*	2 bytes

Notice that the jump instruction jle (jump if less than or equal) depends on ZF, SF, and OF, which are status flags. 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 y = 10? 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 upon entry, x1 contains `x` and x2 contains `y`; x3 should receive `result`. 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 $1, %eax`		
`test %ecx,%ecx`		
`jz done`		
`mul %ebx`		
`dec %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 y = 10 using 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 top two entries; accesses that involve other 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 A	load value at M[A]; push value onto stack
POP A	pop stack; store value to M[A]
MUL	pop two values from the stack; MULtiply them; push result onto stack
INC	pop value from top of stack; increment value by one; push result onto stack
ZERO	zero out value at top of stack
DEC	pop value from top of stack; decrement value by one; push result onto stack
BEQZ *label*	pop value from stack; if it's zero, continue at *label*; else, continue with next instruction
BNEZ *label*	pop value from stack; if it's not zero, continue at *label*; else, continue with next instruction
JUMP *label*	continue execution at location *label*

Translate the `power` 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. Assume `y` is at address 0x8000 (to fit within a 2-byte address specifier). At the end of the loop, `result` should be at the top of the stack.

How many bytes is the stack program using your translation? How many bytes of instructions need to be fetched for y = 10 using 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 powers easier, we will consider a modified architecture that has a second accumulator (the "**multiplication** accumulator") to hold the results of multiplications. Assume each instruction occupies three bytes if it takes an address or label; other instructions occupy one byte.

`load A`	Load the value at address A into the accumulator
`store C`	Store the accumulator's value into the address contained in C
`add A`	Add the value at address A to the value in the accumulator
`mul A`	Multiply the **multiplication** accumulator's value by the value at address A
`swap`	Swap the values in the multiplication and normal accumulators
`inc`	Decrement the accumulator by one
`dec`	Decrement the accumulator by one
`zero`	Zero the value in the accumulator
`beqz L`	Branch to label `L` if the accumulator holds a zero value
`bnez L`	Branch to label `L` if the accumulator does not hold a zero value

Notice that all instructions use the same accumulator, except for multiply instructions. Also note that there are no register specifiers in this example; A and L represent memory addresses. Translate the `power` loop to use this ISA. Assume that at the start, x is held at the address 0x8000 and y is held at 0x8004. You should return the final result in the **multiplication** accumulator.

How many bytes is your program? How many bytes of instructions need to be fetched for y = 10 using your translation? Assuming 32-bit values, how many bytes are loaded when y = 10? 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 it can be expressed in fewer instructions. 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 paragraph that explains 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.

In order to further simplify this problem, *ignore* the busy signal, and assume that the memory is as fast as the register file.

The final solution should be elegant and efficient (e.g. number of new states needed, amount of new hardware added).

Problem 2.A — Implementing Memory-to-Memory Add

For this problem, you are to implement a new memory-memory add operation. The new instruction has the following format:

ADDm r_d, r_s, r_t

ADDm performs the following operation:

M[r_d] <= M[r_s] + M[r_t]

Fill in Worksheet 2.A with the microcode for ADDm. 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 as long as you do it neatly. Your code should exhibit "clean" behavior and not modify any registers (except r_d) in the course of executing the instruction.

Finally, make sure that the instruction fetches the next instruction (i.e., by doing a microbranch to FETCH0 as discussed above).

You may want to consult the micro-code found in the micro-coded processor provided in Lab1, which can be viewed at ${LAB1ROOT}/src/rv32_ucode/micrcode.scala for guidance. Warning: While that micro-code passes all provided assembly tests and benchmarks, no guarantees to the optimality of the code can be assured, and there may still be bugs in the provided implementation.

State	PseudoCode	ldIR	Reg Sel	Reg Wr	en Reg	ldA	ldB	ALUOp	en ALU	ld MA	Mem Wr	en Mem	Ex Sel	en Imm	uBr	Next State
FETCH0:	MA <- PC; A <- PC	0	PC	0	1	1	*	*	0	1	*	0	*	0	N	*
	IR <- Mem	1	*	*	0	0	*	*	0	0	0	1	*	0	N	*
	PC <- A+4	0	PC	1	1	0	*	INC_A_4	1	*	*	0	*	0	D	*
. . .																
NOP0:	microbranch back to FETCH0	0	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH0
ADDM0:																

In this question we ask you to implement a useful string instruction, *string length* (STRLEN). This instruction uses the same encoding as the other arithmetic instructions (R-type) on RISC-V (note rs2 should always be 00000 as there's only one source operand):

rd	rs1	rs2	func	opcode
5 bits	5 bits	5 bits	10 bits	7 bits

The STRLEN instruction measures the length of a string stored in memory (**M[Rs1]**), and returns the result in **Rd.**

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"). The length of the string is equal to the number of 4-byte words that precede the null character (a string consisting of only the null character has length 0).

Your task is to fill out Worksheet 2.B for STRLEN 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	N	*
	IR <- Mem	1	*	*	0	0	*	*	0	*	0	1	*	0	S	*
	PC <- A+4; dispatch	0	PC	1	1	0	*	INC_A_4	1	*	*	0	*	0	D	*
. . .																
NOP0:	microbranch back to FETCH0	*	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH0
STRCPY:																

Problem 2.C **Instruction Execution Times**

How many cycles does it take to execute the following instructions in the microcoded RISC-V machine? Use the states and control points from RISC-V-Controller-2 in Lecture 2 (or Lab 1, in ${LAB1ROOT}/src/rv32_ucode/micrcode.scala) and assume Memory will not assert its busy signal.

Instruction	Cycles
`SUB  x3,x2,x1`	
`ORI  x2,x1,#4`	
`SW   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 4. 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.***

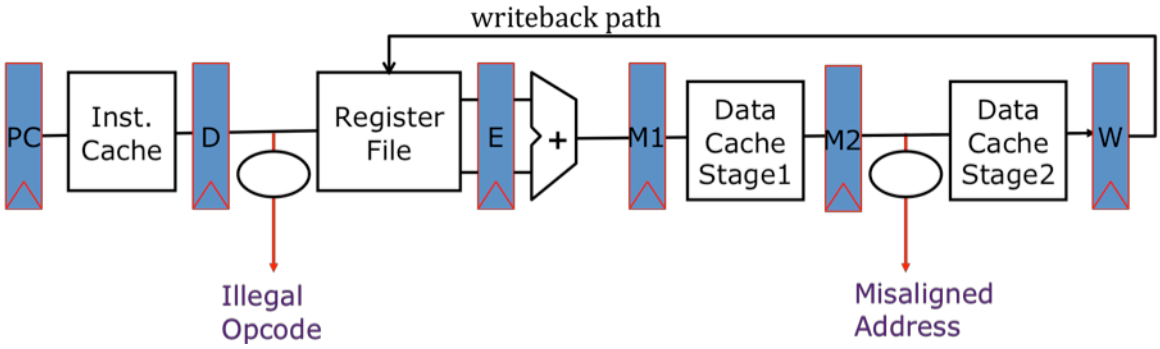

Figure 1-A. 6-stage pipeline. For clarity, bypass paths are not shown.

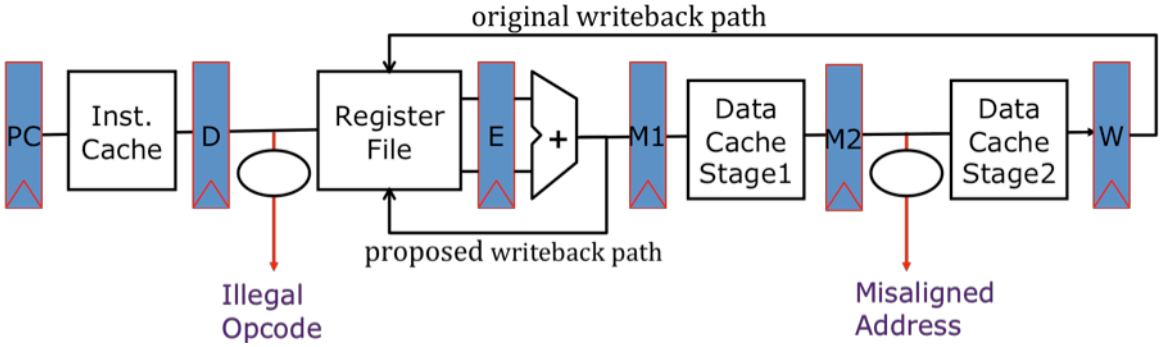

Figure 1-B. 6-stage pipeline with proposed additional write port.

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, circle 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 (say, you're 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 2019 (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				

f)	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)				
g)	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 front-end)				