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 his own solution to the problems.

CS152 Computer Architecture and Engineering

SOLUTIONS

ISAs, Microprogramming and Pipelining

Spring 2019 — Problem Set #1 — *Due February 11*

The problem sets also provide essential background material for the quizzes. 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 at the quizzes! 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

Problem 1.A **CISC**

How many bytes is the program?
6B + 2B + 2B + 2B + 1B + 2B = 15B

For the above x86 assembly code, how many bytes of instructions need to be fetched if b = 10?
6B + 11*(2B + 2B) + 10*(2B + 1B + 2B) = 100B
The prologue will execute once, the first condition check will execute 11 times, and the body of the loop will execute 10 times.

Assuming 32-bit data values, how many bytes of data memory need to be fetched? Stored?
There are no loads or stores!
Stored: 0

Problem 1.B **RISC**

Many translations will be appropriate; here's one. Under the "direct translation" objective, it was okay to replace test with an instruction, a NOP, or nothing.

x86 instruction	label	RISC-V instruction sequence
`movl $1, %eax`		`addi x3, x0, x1`
`test %ecx,%ecx`	`loop:`	`add x0, x0, x0 # NOP, could be nothing`
`jz done`		`beq x2, x0, done`
`mul %ebx`		`mul x3, x3, x1`
`dec %ecx`		`addi x2, x2, -1`
`jmp loop`		`jal x0, loop`
`...`	`done:`	`...`

How many bytes is the RISC-V program using your direct translation?
6*4 = 24 (or 5*4=20 if you leave out the explicit NOP)

How many bytes of RISC-V instructions need to be fetched for y = 10 using your direct translation?
For an explicit NOP:

- The 4B prologue executes once
- The 8B conditional test executes 11 times
- The 12B body of the loop executes 10 times

4B + 11*8B + 10*12B = 212B

If eliminating the NOP: 168B

Assuming 32-bit data values, how many bytes of data memory need to be loaded? Stored?
There are still no loads or stores.

Problem 1.C **Stack**

```
Label    Instruction      Comment With Stack Contents
         ...              # x
         pop 0x8004       # <empty>
         push 0x8000      # y
         zero             # 0
         inc              # 1 == result
loop:    push 0x8000      # y, result
         beqz done        # result
         push 0x8004      # x, result
         mul              # result'
         push 0x8000      # y, result'
         dec              # y', result'
         pop 0x8000       # result'
         jump loop        # result'
done:    ...              # result
```

Here, the "prime" symbol is used to distinguish the $n+1^{th}$ value taken by a variable from the n^{th}.

How many bytes is your program?
8*3B + 4*1B = 28B

Using your stack translations from part (c), how many bytes of stack instructions need to be fetched for y = 10?
8B + 11*6B + 10 * 14B = 214B

Assuming 32-bit data values, how many bytes of data memory need to be fetched?

- 4B loaded in prologue (once)
- 4B loaded in loop bounds check (11 times)
- 8B loaded in loop body (10 times)
- TOTAL: 128B

Stored?

- 4B stored in prologue (once)
- 4B stored in loop body (10 times)
- TOTAL: 44B

If you could push and pop to/from a four-entry register file rather than memory (the Java virtual machine does this), what would be the resulting number of bytes fetched and stored?
There are only three variables, so almost all memory accesses could be eliminated; only the initial load from 0x8000 would remain. This would yield 4B loaded and zero stored.

Problem 1.D **Accumulator**

One possible solution:

```
        zero          # A = 0
        inc           # A = 1
        swap          # MA = 1, A = ?
        load 0x8004   # A = y
loop:   mul 0x8000    # MA = MA * x
        dec           # A--
        bnez loop     # repeat if y != 0
```

How many bytes is your program?
4*1B + 3*3B = 13B

How many bytes are fetched for y = 10?
6B prologue + 10*(7B body) = 76B

How many bytes are loaded for y = 10? Stored?
44B loaded, 0B stored.

Problem 1.E **Conclusions**

CISC < RISC < STACK for both static and dynamic code size.
(RISC ≈ CISC) < 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**

A simple optimization is loop unrolling. With some careful bookkeeping, this can reduce the number of multiplies! This implementation keeps x*x as a temporary value to cut the number of multiplies approximately in half.

```
      addi x3, x0, 1     # result = 1
      andi x6, x2, -1    # x6 = y - (y%2)
      mul  x7, x1, x1    # x7 = x*x
      beq  x6, x2, loop  # skip ahead if y is even
      add  x3, x0, x1    # set result to x if y is odd
loop: mul  x3, x3, x7    # result = result * x*x
      addi x6, x6, -2    # x6 = x6 - 2
      bge  x6, x0, loop  # repeat if x6 != 0
done: ...
```

Problem 2: Microprogramming and Bus-Based Architectures

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

Note that the microarchitectural registers in the path (IR, A, B, MA) are not part of the architectural state. Therefore, using asterisks for the load signals (ldIR, ldA, ldB, and ldMA) is acceptable as long as the correctness of your microcode is not affected—in fact, the most optimal solution will have as many "don't care" signals as possible, as this can result in slightly better implementations of the underlying digital circuits.

This microarchitecture differs from others you have seen, since memory accesses take only one cycle. This means that it is possible to access memory and perform a conditional microbranch or jump in the same micro-op, as there is no need to use the "spin" microbranch. Furthermore, since it is guaranteed that memory accesses finish in one cycle, it is possible to have ldMA be set to any value while accessing memory, as long as the address isn't reused again in the future.

The table below shows one correct way to implement ADDm0 in microcode:

State	PseudoCode	Ld IR	Reg Sel	Reg Wr	en Reg	ld A	ld B	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	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
ADDm0:	MA <- R[rs1]	0	rs1	0	1	*	*	*	0	1	*	0	*	0	N	*
	A <- Mem	0	*	*	0	1	*	*	0	0	0	1	*	0	S	*
	MA <- R[rs2]	0	rs2	0	1	0	*	*	0	1	*	0	*	0	N	*
	B <- Mem	0	*	*	0	0	1	*	0	0	0	1	*	0	S	*
	MA <- R[rd]	*	rd	0	1	0	0	*	0	1	*	0	*	0	N	*
	Mem <- A+B	*	*	*	0	0	0	ADD	1	0	1	1	*	0	S	*
	ubr to fetch	*	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH0

Worksheet M1-1: Implementation of ADDm instruction

Problem 2.B **Implementing STRLEN Instruction**

The table below shows one way to implement STRLEN in microcode.

A few notes:

- LdIR is zero throughout, since we need to keep the rd register specifier around until STRLEN is complete
- The pointer and counter values bounce in and out of the 'A' register, as it enables using immediate values
- Having multiple 'ldX' signals enabled at once can help get more work done in fewer cycles, by broadcasting a value to multiple destinations

- We can ubranch on flags of an ALU operation without driving the result to the bus (saves energy)

State	PseudoCode	IdIR	Reg Sel	Reg Wr	en Reg	IdA	IdB	ALUOp	en ALU	Id MA	Mem Wr	en mem	Imm Sel	en Imm	uBr	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	N	*
	PC <- A+4; dispatch	0	PC	1	1	0	*	INC_A_4	1	*	*	0	*	0	D	*
…																
NOP0	uBr to FETCH0	*	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH0
STRLEN	A,B,MA←R[rs1]	0	rs1	0	1	1	1	*	0	1	*	0	*	0	N	*
	Rd←0	0	rd	1	1	*	0	SUB	1	0	*	0	*	0	N	*
STRLEN2	A←Mem	0	*	*	0	1	0	*	0	*	0	1	*	0	N	*
	if A==0, uBr to FETCH0, A <- R[rd]	0	rd	0	1	1	0	COPY_A	0	*	*	0	*	0	EZ	FETCH0
	RD ← A + 1	0	rd	1	1	*	0	INC_A_1	1	*	*	0	*	0	N	*
	A ← B	0	*	*	0	1	*	COPY_B	1	*	*	0	*	0	N	*
	B, MA ← A + 4, uBr J STRLEN2	0	*	*	0	*	1	INC_A_1	1	*	*	0	*	0	J	STRLEN2

Problem 2.C **Instruction Execution Times**

The answers below are derived from the microcoded processor described in the lab. It is okay if your answers differ from having been derived from the lecture notes.

Instruction	Cycles
ADD x3,x2,x1	3 + 3 = 6
ORI x2,x1,#4	3 + 3 = 6
SW 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 + 3 + 4 = 10
BEQ x1,x2,label #(x1 != x2)	3 + 4 = 7
J label	3 + 5 = 8
JAL label	3 + 5 = 8
JALR x1	3 + 5 = 8
AUIPC x1, #128	3 + 4 = 7

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

Instruction	Cycles	uInsn summary (not including fetch & dispatch)
ADD x3,x2,x1	3 + 3 = 6	1) A ← R[x1] 2) B ← R[x2] 3) R[x3] ←A+B
ORI x2,x1,#4	3 + 3 = 6	1) A ← R[x1] 2) B ← R[x2] 3) R[x3] ←A |[1] B
SW x1,0(x2)	3 + 4 = 7 or +1 = 8	1) A ← R[x2] 2) B ← Imm 3) MA ← A + B 4) M[MA] ← R[x1] 5) uBr J FETCH[2]
BNE x1,x2,label #(x1 == x2)	3 + 3 = 6	1) A ← R[x1] 2) B ← R[x2] 3) A – B; uBr NZ FETCH0; B ← Imm[4]
BNE x1,x2,label #(x1 != x2)	^ + 3 = 9	1) A ← PC 2) A ← PC – 4 3) PC ← A + B
BEQ x1,x2,label #(x1 == x2)	3 + 3 = 6	1) A ← R[x1] 2) B ← R[x2] 3) A – B; uBr EZ FETCH0; B ← Imm[4]
BEQ x1,x2,label #(x1 != x2)	^ + 3 = 9	1) A ← PC 2) A ←A - 4 3) PC ← A + B
J label	3 + 3 = 6	1) R[rd] ← PC 2) B ← Imm 3) PC ← A[3]+B
JAL label	3 + 3 = 6	Same as above
JALR x1	3 + 3 = 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 uInsns are assumed to have a uBr J back to FETCH0 unless stated otherwise.

[1] These operations were not provided in the handout, but it is reasonable to assume the can occur in a single ALU op.

[2] The wording of the question is ambiguous as to whether or not "Memory will not assert its busy signal" implies that it cannot happen in the machine vs you can assume it won't for the purposes of the cycle accounting. The prior would permit uBRing in parallel with the store. The safer, intended answer assumes the latter case and separates the uBr and the memory op.

[3] A 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 PC into A register and decrement it by four. (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.

Problem 3: 6-Stage Pipeline

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

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**

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 have to 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 write after write hazards that must be appropriately addressed by the control logic. The two instructions writing at the same time have to 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**

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**

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**

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, 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**

CISC

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 couldn't hide behind convenient software abstractions/subroutines!

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

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.

RISC

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

CISC

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. C

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: typically the front-end of the processor 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're still around for legacy reasons. But if you had a chance at a clean slate, you'd 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).

RISC

Problem 5: Iron Law of Processor Performance

		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 and I$ hit-rates.
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 MIPS 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 front-end)	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.