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

Assigned 1/24/2016 Problem Set #1 *Due February 5*

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

In this problem, your task is to compare four different ISAs. x86 is an extended accumulator, CISC architecture with variable-length instructions. RISC-V is a load-store, RISC architecture with fixed-length instructions (for this problem only consider the 32-bit form of its ISA). We will also look at a simple stack-based ISA and at an accumulator architecture.

Problem 1.A CISC

How many bytes is the program?
19

For the above x86 assembly code, how many bytes of instructions need to be fetched if b = 10?
4+10*(13)+10=144

Assuming 32-bit data values, how many bytes of data memory need to be fetched? Stored?
Fetched: the compare instruction accesses memory, and brings in a 4-byte word b+1 times:
4*11= 44
Stored: 0

Problem 1.B RISC

Many translations will be appropriate; here's one. Other people have used sub instead of slt. Remember (as far as we are concerned for this PS, or Lab 1, or any Quiz), RISC-V instructions are only 32 bits long so you need to construct a 32 bit address from 12-bit and 20-bit immediates. Also, since the problem specified that the value of b was already contained in x1, you could skip the lui/lw instructions entirely.

x86 instruction	**label**	**RISC-V instruction sequence**
`xor %edx,%edx`		`xor x4, x4, x4`
`xor %ecx,%ecx`		`xor x3, x3, x3`

`cmp 0x8049580,%ecx`	`loop:`	`lui x6, 0x08049`
		`lw x1, 0x580 (x6)`
		`slt x5, x3, x1`

`jl L1`		`bne x5,x0, L1`
`jmp done`		`j done`
`add %eax,%edx`	`L1:`	`add x4, x4, x2`

`inc %ecx`
 	
`jmp loop`
 	
`...`
	
	`done:`
		`addi x3, x3, #1`
		
		`j loop`
		
		`...`

How many bytes is the RISC-V program using your direct translation?
10*4 = 40 (or 8*4=32 if you leave out the lui/lw)

How many bytes of RISC-V instructions need to be fetched for b = 10 using your direct translation?
Since the value of b stays the same, you don't have to repeat the lui and lw after you load it into a register the first time. So there are 4 instructions in the prelude and 5 that are part of the loop (we don't need to fetch the "j" until the 11th iteration). There are 3 instructions in the 11th iteration. All instructions are 4 bytes. 4*(4+10*5+3) = 228. If you noted that b is in a register and didn't load from memory, then it's only 220 bytes.

Assuming 32-bit data values, how many bytes of data memory need to be fetched? Stored?
Fetched: 4 (or zero if you keep B in a register)
Stored: 0

Problem 1.C **Stack**

```
        pop a   ;m[a] <- a
        push 0  ;push a dummy value (mem[0]) onto stack so we
        zero    ;   have something to zero
        pop result  ;m[result] <- 0   (result)
        push 0   ;push a dummy value onto stack
        zero
        pop i   ;m[i] <- 0   (i)
loop:   push 0x8000 ;push b
        push i
        sub         ;b-i
        bnez L1
        goto done
L1:     push a
        push result
        add
        pop result  ; result = result+a
        push i
        inc    ; i=i+1
        pop i
        goto loop
done:
```

How many bytes is your program?
50

Using your stack translations from part (c), how many bytes of stack instructions need to be fetched for b = 10?
(5*3+2*1) + 10*(9*3+3*1)+(4*3+1) = 330

Assuming 32-bit data values, how many bytes of data memory need to be fetched?
fetched = 4*number of dynamic pushes. There are 2 in the prelude, 2 at loop that get executed 11 times, and 3 at L1 that get executed 10 times. 2+2*11+3*10=54. 54*4 bytes = 216 bytes

Stored?
stored = 4 * number of dynamic pops. 4*(3+2*10) = 92 bytes
Note that the stack-depth in this program never exceeds two words, so we don't have to worry about extra accesses for spilling.

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 four variables, so almost all memory accesses could be eliminated. If you stick to

a direct translation where you keep b in memory, then you would have to get it 11 times: 44 bytes fetched, 0 bytes stored. If you keep b in a register, too, then you only have to get it once: 4 bytes fetched, 0 bytes stored (but the code in 1.C's answer doesn't directly support this).

Problem 1.D Accumulator

```
        zero accumulator         Zero the accumulator
        load B subtractor        Load variable b into the subtractor

loop:   dec subtractor           Decrement the subtractor
        add A accumulator        Add the value of variable A into the accumulator
        bnez loop subtractor     Branch if the subtractor is non zero
        goto done
done:
```

The above is just one way of doing it by using both the accumulator and subtractor. There is a way to solve this problem with using just one of the two, by storing and loading values from memory at every loop iteration, similar to the stack architecture. Both solutions are fine.

How many bytes is your program?
17

Can the same program be implemented with just one accumulator (i.e., no subtractor)?
There are two ways to answer this. If we don't load and store values to and from memory at every iteration the answer is no. By the nature of this ISA, the moment we load one variable, say B, we cannot store another variable which is necessary to do the increments.
However, similar to the stack architecture, we can make this work with one accumulator. It just takes more loads and stores.

If not, how would you extend this ISA to implement this program with just one accumulator?
If the answer to the above was no, we can simply add a multiply instruction, or add memory-memory instructions (instructions that can operate on two memory addresses). However, the latter would change the nature of the ISA.

Problem 1.E Conclusions

CISC < RISC < STACK for both static and dynamic code size.
(RISC ≈ CISC) < STACK for data memory traffic

Problem 1.F Optimization

Most optimizations revolve around the elimination of unnecessary control flow. Also, the load can be hoisted out of the loop.

```
        lui x6, 0x08049          ;optional if x1 already contains b
        lw x1, 0x580(x6)         ;optional if x1 already contains b
        xor x4, x4, x4
        blt x1,x0, done
loop:   addi x1, x1, -1
        add x4, x4, x2
        bgtz x1, loop
done:
```

The optimization here is to decrement a counter until it reaches zero, instead of incrementing an initially zero count until it reaches b. We can also omit the unconditional jumps if we just put a conditional branch at the very end. This re-write brings dynamic code size down to 136 bytes; static code size to 28; and memory traffic down to 4 bytes.

Problem 2: Microprogramming and Bus-Based Architectures

Problem 2.A Implementing Memory-to-Memory Add

Worksheet M1-1 shows one way to implement ADDm in microcode.

Note that to maintain "clean" behavior of your microcode, no registers in the register file should change their value during execution (unless they are written to). This does not refer to the registers in the datapath (IR, A, B, MA). Thus, using asterisks for the load signals (ldIR, ldA, ldB, and ldMA) is acceptable as long as the correctness of your microcode is not affected (and in fact, should be done for full optimality). Also note the ubr to FETCH0 must be contained on its own line, since you can't "spin" on the same micro-code line if memory is still busy OR jump to FETCH0 if memory is not busy. S is either "spin on same micro-code line (upc)" or go to upc+1.

When performing a memory access, you could be "spinning" for many cycles, waiting for memory to become "not busy". In that time, you must keep all inputs to the memory system constant: thus, ldMA must be 0, because you don't want the memory address to change while accessing memory! Likewise, in "Mem <- A+B", ldA and ldB must also be set to 0, so that the data being sent to memory stays constant. To phrase this is another way, we have no idea when the memory system latches in our inputs.

Finally, note the cleverness of ldA being "0" on FETCH2. On entering an instruction, A always equals PC+4. This saves a cycle if we dispatch to a jump instruction, which first loads PC+4 into the ALU (or the RDNPC instruction, which loads PC+4 into rd).

The microcode for ADDm is straightforward.

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 STRCPY Instruction

Worksheet M1-2 shows one way to implement STRCPY in microcode.

A few notes:
-LdIR is zero for all uops because we keep needing to read the actual values of Rs, Rd which are stored in the IR register
-ldMA is kept at 0 when performing a memory operation because memory operations are multi-cycle and thus you need to hold the memory address constant (this logic also applies to ldA,ldB when used as sources for memory).

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	*	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
STRCPY:	MA ← R[rs1]; A ← R[rs1]	0	Rs1	0	1	1	*	*	0	1	*	0	*	0	N	
	B ← Mem	0	*	*	0	0	1	*	0	0	0	1	*	0	S	
	MA ← R[rd]	0	Rd	0	1	0	0	*	0	1	*	0	*	0	N	
	Mem ← B	0	*	*	0	0	0	COPY_B	1	0	1	1	*	0	S	
	If (B == 0) uBr to FETCH0	0	*	*	0	0	*	*	0	*	*	0	*	0	Z	FETCH0
	R[rs1] ← A + 4	0	Rs1	1	1	*	*	INC_A_4	1	*	*	0	*	0	N	
	A ← R[rd]	0	Rd	0	1	1	*	*	0	*	*	0	*	0	N	
	R[rd] ← A + 4; J to STRCPY	0	Rd	1	1	*	*	INC_A_4	1	*	*	0	*	0	J	STRCPY

Problem 2.C Instruction Execution Times

How many cycles does it take to execute the following instructions in the microcoded RICV-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
`ADD  x3,x2,x1`	$3+3=6$
`ADDI 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`	$3+5=8$

As discussed in Lecture 2, instruction execution includes the number of cycles needed to fetch the instruction. The lecture notes used 4 cycles for the fetch phase, while Worksheet 1 shows that this phase can actually be implemented in 3 cycles —either answer was fine. The above table uses 3 cycles for the fetch phase.

The above answers are derived from the micro-coded processor provided in Lab 1. It is okay if your answers differ from having been derived from the lecture notes.

Overall, BNE (for a taken branch), and BEQ (for a taken branch) take the most cycles to execute (10), while arithmetic functions such as ADD and ADDI take the fewest cycles (6).

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. (Recall that the RF is read in the ID stage, and when after the first instruction has written back, it is in M1, so the third instruction is in ID.)

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.

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 ALU op 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, even though preceding 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 one-third of dynamic instructions. Assuming that the instruction following a load or store is an ALU op two-thirds 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 reading an instruction's source operands in the ID stage, also read the destination register, rd. If an early writeback occurs before a preceding exception was detected, then the old value of rd is preserved in the EX/M1 pipeline register and can be restored to the register file, maintaining precise state.

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**

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 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! For example, a CISC instruction which performs expensive, multi-cycle floating point routines in hardware is FAR faster than a software floating point subroutine that requires perhaps dozens of expensive instruction fetches.

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: doesn't change pipeline Decrease: branch_kill signal is no longer needed	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. No effect: if it is unusable.	Increase: if implementing the instruction means adding or re-using stages. No effect: if the number of cycles is kept constant but it just lengthens the logic in one stage.	Increase: since more logic and thus longer critical path. No effect: if it is implemented by more or re-used stages but each stage gets no longer.	Ambiguous: if the program can take advantage of the new instruction, it can mitigate the costs of implementing it. 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 stalls, and likely be difficult 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: if access to Memory is pipelined (>1 cycle) since it will now take less cycles. No effect: if memory access is done in a single cycle.	Decrease: if memory access is on the critical path or memory was 1 cycle. No effect: if memory is pipelined and just takes less cycles.	Improve: improving memory access time, at least by these Iron Law metrics, will increase performance of the whole system (unless you chose “no effect” for everything). Of course, there could be other secondary costs of improving mem. access speeds, like having to use smaller caches, but I’m getting carried away here.
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: because you are replacing 32b instructions with equivalent 16b versions, it saves on code space, but it leaves the Inst/ Program count unchanged	No effect: you are simple executing equivalent 16b versions of regular 32b instructions. Both appear identical to the pipeline. decrease: since code size has shrunk, I$ hits will increase and thus less cycles will be spent fetching instructions	Increase: decode may increase this since the instruction format is more complex (and you have to deal with figuring out where the instruction boundaries are) No effect: if this fits within the cycle time, since this makes no change to the pipeline and only increases the decode stage (or perhaps adds another stage to the front-end).	Ambiguous: the main advantage is smaller code size, whichcan improve I$ hit rates and save on fetch energy (get more instructions per fetch). This can improve performance (or at least energy), however the more complex decode could also counteract 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: because 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. Increase: the RISC pipeline introduces longer control paths and adds bypasses, which are likely to be on the critical path.	Increase: it should be far easier to pipeline RISC uops once the CISC instructions Have been decoded/translated, leading to a higher performance machine (see modern x86 machines).