CS152 Computer Architecture and Engineering
Introduction to ISAs and Microprogramming

January 29, 2009

SOLUTIONS

Assigned January 30 — Problem Set #1 — *Due February 10*

http://inst.eecs.berkeley.edu/~cs152/sp09

Problem 1: CISC, RISC, and Stack: comparing ISAs

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. We ignore MIPS's branch-delay slot in this solution since it hadn't been discussed in lecture. Remember MIPS64 instructions are only 32 bits long so you need to construct a 32 bit address from 16 bit immediates. Also, since the problem specified that the value of b was already contained in R1, you could skip the lui/lw instructions entirely.

x86 instruction	label	MIPS64 instruction sequence
`Xor %edx,%edx`		`xor r4, r4, r4`
`Xor %ecx,%ecx`		`xor r3, r3, r3`

`Cmp 0x8049580,%ecx`	*loop*	`lui r6, 0x0804`
		`lw r1, 0x9580 (r6)`
		`slt r5, r3, r1`

`jl L1`		`bnez r5, L1`
`Jmp done`		`j done`
`Add %eax,%edx`	*L1*	`add r4, r4, r2`

`inc %ecx`
 	
`Jmp loop`
		`addi r3, r3, #1`
		
		`j loop`

`...`	*done:*	`...`

How many bytes is the MIPS64 program using your direct translation?
10*4 = 40 (or 8*4=32 if you leave out the lui/lw)

How many bytes of MIPS64 instructions need to be fetched for b = 10 using your direct translation.
If you get part of the address of b into a register, you don't have to repeat the lui. So there are 3 instructions in the prelude and 6 that are part of the loop (we don't need to fetch the "j" until the 11th iteration). There are four instructions in the 11th iteration. All instructions are 4 bytes. 4*(3+10*6+4) = 268. If you kept b in a reg, then it's 4(2+10*5+3) = 220.

How many bytes of data memory need to be fetched? Stored?
Fetched: 11 * 4 = 44. (or zero if you keep B in a reg)
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

How many bytes of data memory need to be fetched? Stored?
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 = 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 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 Conclusions

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

Problem 1.E Optimization

ideas:
count down to zero instead of up: then you don't need the slt prior to the branch
rearrange control flow can eliminate a jump that must get killed n-1 times.
hold b (or &b, depending on your interpretation) in a register if you haven't done it already

```
      xor r4, r4, r4
      lui r6, 0x0804          ;optional if r1 already contains b
      lw r1, 0x9580(r6)       ;optional if r1 already contains b
      jmp dec
loop: add r4, r4, r2
dec:  addiu r1, r1, #-1
      bgez r1, loop
done:                         ;this inst must be idempotent if we assume
delay slots.
```

This re-write brings dynamic code size down to 140/132 bytes; static code size to 32/24; and memory traffic down to 4/0 bytes (Notation: b in memory / b already in r1). A nop after the jmp would make this legal MIPS and still keep the loop tight.

Problem 2: Microprogramming and Bus-Based Architectures

Problem 2.A — 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.

The microcode for ADDm is straightforward.

Problem 2.B — Implementing DBNEZ Instruction

The question asked to jump to PC+4+offset. This ignores that the immediate value needs to be shifted left by 2 before it can be added to PC+4, to make sure we don't run into alignment problems. We did this because the data path given doesn't really have facilities for shifting.

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

Problem 2.C — Instruction Execution Times

Instruction	Cycles
`SUB  R3,R2,R1`	3 + 3 = 6
`SUBI R2,R1,#4`	3 + 3 = 6
`SW   R1,0(R2)`	3 + 5 = 8
`BNEZ R1,label  # (R1 == 0)`	3 + 2 = 5
`BNEZ R1,label  # (R1 != 0)`	3 + 5 = 8
`BEQZ R1,label  # (R1 == 0)`	3 + 5 = 8
`BEQZ R1,label  # (R1 != 0)`	3 + 2 = 5
`J    label`	3 + 3 = 6
`JR   R1`	3 + 2 = 5
`JAL  label`	3 + 4 = 7
`JALR R1`	3 + 4 = 7

As discussed in Lecture 4, 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. Overall, SW, BNEZ (for a taken branch), and BEQZ (for a taken branch) take the most cycles to execute (8), while BNEZ (for a not-taken branch), BEQZ (for a not-taken branch) and JR take the fewest cycles (5).

State	PseudoCode	Ld IR	Reg Sel	Reg Wr	en Reg	ld A	ld B	ALUOp	en ALU	ld MA	Mem Wr	en Mem	Ex Sel	en Imm	μBr	Next State
FETCH0:	MA <- PC; A <- PC	0	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	*	*	INC_A_4	1	*	*	0	*	0	D	*
. . .																
NOP0:	microbranch Back to FETCH0	0	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH0
ADDm0:	MA <- R[rs]	0	rs	0	1	*	*	*	0	1	*	0	*	0	N	*
	A <- Mem	0	*	*	0	1	*	*	0	*	0	1	*	0	N	*
	MA <- R[rt]	0	rt	0	1	0	*	*	0	1	*	0	*	0	N	*
	B <- Mem	0	*	*	0	0	1	*	0	*	0	1	*	0	N	*
	MA <- R[rd]	*	rd	0	1	0	0	*	0	1	*	0	*	0	N	*
	Mem <- A+B; fetch	*	*	*	0	*	*	ADD	1	*	1	1	*	0	J	FETCH0

Worksheet M1-1: Implementation of ADDm instruction

State	PseudoCode	ld IR	Reg Sel	Reg Wr	en Reg	ld A	ld B	ALUOp	en ALU	Ld MA	Mem Wr	en Mem	Ex 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	N	*
	PC <- A+4; B <- A+4	0	PC	1	1	*	1	INC_A_4	1	*	*	0	*	0	D	*
. . .																
NOP0:	microbranch back to FETCH0	*	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH0
DBNEZ:	**A <- rs**	**0**	**rs**	**0**	**1**	**1**	**0**	*****	**0**	*****	*****	**0**	*****	**0**	**N**	*****
	rs <- A – 1 **µB to FETCH0 if zero**	**0**	**rs**	**1**	**1**	*****	**0**	**DEC_A_1**	**1**	*****	*****	**0**	*****	**0**	**Z**	**FETCH0**
	A <- sExt16(IR)	*****	*****	*****	**0**	**1**	**0**	*****	**0**	*****	*****	**0**	**sExt16**	**1**	**N**	*****
	PC <- A+B **jump to FETCH0**	*****	**PC**	**1**	**1**	*****	*****	**ADD**	**1**	*****	*****	**0**	*****	**0**	**J**	**FETCH0**

Worksheet M1-2: Implementation of DBNEZ Instruction

Problem 2.D — Exponentiation

In the given code, 'm' and 'n' are always nonnegative integers. Therefore, we don't have to worry about the cases where 'i' is larger than 'n' or 'j' is larger than 'm'. Also, for this problem, 0 raised to any power is just 0, while any nonzero value raised to the 0th power is 1. Note that the pseudocode that is given returns a value of 0 when 0 is raised to the 0th power. However, the actual `pow()` function in the standard C library returns a value of 1 for this case. We present the solution that implements the pseudocode given in the problem rather than C's `pow()` function.

```
#
# R5: temp, R6: j
#

            ADD   R3, R0, R0       ; put 0 in result
            BEQZ  R1, _END_I       ; if m is 0, end
            ADDI  R3, R0, #1       ; put 1 in result
            BEQZ  R2, _END_I       ; if n is 0, the loop is over; we set
                                   ; i equal to n and count down to 0—since
                                   ; R2 does not have to be preserved, we
                                   ; use it for i
            SUBI  R5, R1, #1       ; temp = m - 1
            BEQZ  R6, _END_I       ; if m is 1, the result will be 1,
                                   ; so end the program
_START_I:
            ADD   R5, R0, R3       ; temp = result
            SUBI  R6, R1, #1       ; j = m - 1 (the number of times to
                                   ; execute the second loop)
_START_J:
            ADD   R3, R3, R5       ; result += temp
            SUBI  R6, R6, #1       ; j--
            BNEZ  R6, _START_J     ; Re-execute loop until j reaches 0
_END_J:
            SUBI  R2, R2, #1       ; i--
            BNEZ  R2, _START_I     ; Re-execute loop until i reaches 0
_END_I:
```

To compute the number of instructions and cycles to execute this code, let us consider subsets of the code.

Code	# of instructions	# of cycles
`ADD R3, R0, R0` `BEQZ R1, _END_I`	2	$6\times1 + 8\times1 = 14$ (m = 0) $6\times1 + 5\times1 = 11$ (m > 0)
`ADDI R3, R0, #1` `BEQZ R2, _END_I`	2 (if m > 0)	$6\times1 + 8\times1 = 14$ (n = 0) $6\times1 + 5\times1 = 11$ (n > 0)
`SUBI R5, R1, #1` `BEQZ R6, _END_I`	2 (if m > 0 and n > 0)	$6\times1 + 8\times1 = 14$ (m =1) $6\times1 + 5\times1 = 11$ (m > 1)

`_START_I:`
`ADD R5, R0, R3`
`SUBI R6, R1, #1`
	2n (if m > 1 and n > 0)	$(6\times2)\times n = 12n$

`_START_J:`
`ADD R3, R3, R5`
`SUBI R6, R6, #1`
`BNEZ R6, _START_J`
	3n(m-1)
	(if m > 1 and n > 0)
		$(6\times2 + 5\times1)\times n + (6\times2 + 8\times1) \times(m-2)\times n = 17n + 20n(m-2)$

`_END_J:`
`SUBI R2, R2, #1`
`BNEZ R2, _START_I`
	2n (if m > 1 and n > 0)	$(6 + 8)\times n - 2 = 14n-2$

From the above table, we can complete the table given in the problem.

m,n	Instructions	Cycles
0, 1	2	14
1, 0	4	25
2, 2	20	117
3, 4	46	283
M, N (m = 0)	2	13
M, N (m > 0, n = 0)	4	24
M, N (m = 1, n > 0)	6	35
M, N (m > 1, n > 0)	3n(m-1)+4n+6	19n(m-2)+42n+31

Problem 3: A 5-Stage Pipeline with an Additional Adder

Problem 3.A — Elimination of a hazard

The new datapath was trying to eliminate the hazard that occurs when a load instruction is immediately followed by an ALU instruction that requires the value that was loaded. In the original datapath, a pipeline interlock (stall) is needed for this type of instruction sequence, an example of which is shown below. In Ben's datapath, this load-use interlock is not required because the data from the load instruction can be immediately forwarded to the ALU.

```
LW R1, 0(R3)
ADDI R1, R1, #5
```

Problem 3.B — New Hazard

The new hazard occurs when the result of an ALU operation is needed to calculate the address of a load or store instruction.

```
ADDI R1, R1, #5
LW R3, 3(R1)
```

Problem 3.C — Comparison

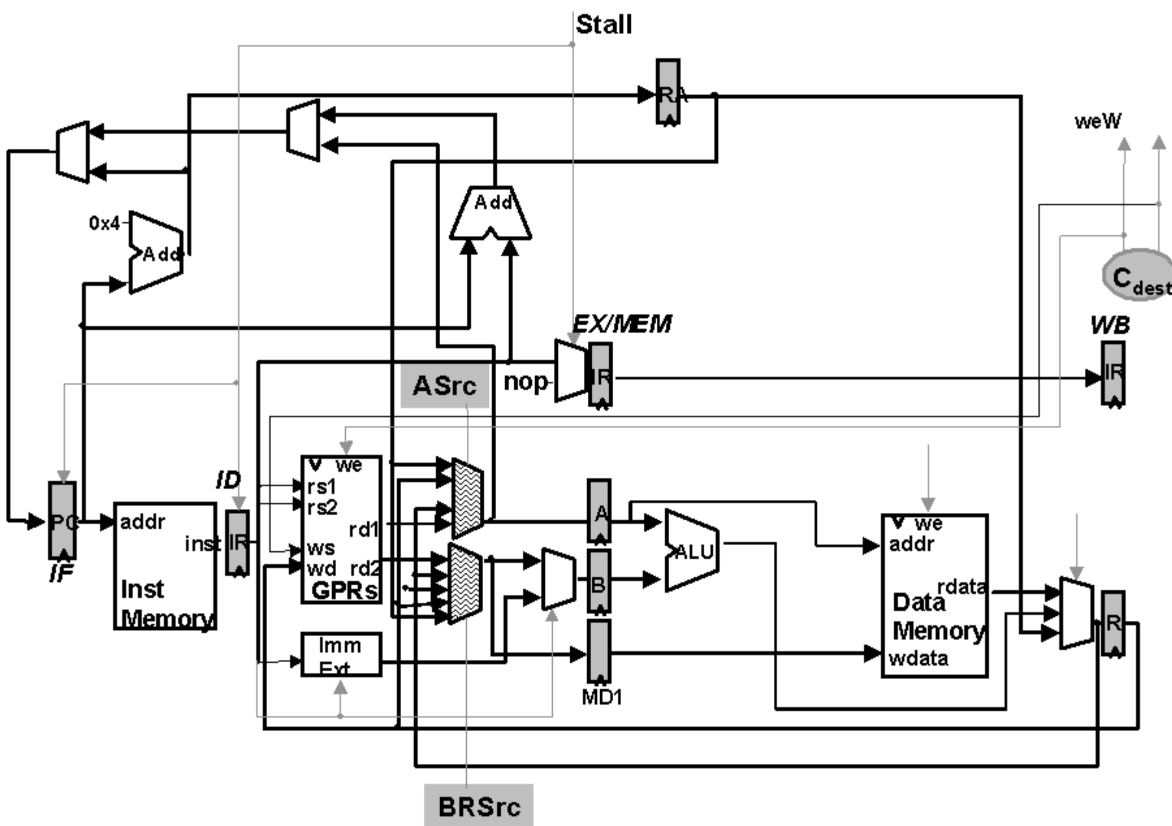
Now an address-generation interlock is needed for the LW instruction in the sequence in 3.B. Note that this new hazard affects both load and store instructions, while the original hazard only affected load instructions. This is a disadvantage of the modified pipeline. Also, the new datapath requires more hardware (another adder) than the original datapath. However, the load-use hazard illustrated in Problem 3.A has been eliminated. If we examine the behavior of typical programs, we will see that the percentage of load instructions resulting in the load-use interlock from Problem 3.A is higher than the percentage of all loads and stores resulting in the address-generation interlock from Problem 3.B. This is because many address calculations are based on values that change infrequently (e.g. the stack pointer does not change while a procedure is being executed). If a base address register has not been recently changed, then there will be no address-generation interlock. By contrast, when a load is issued, the load value is usually required within a few cycles, so a load-use interlock is much more likely. Whether performance is better on the original pipeline or on the modified pipeline will depend on the specific program.

Problem 3.D — Datapath Improvement

If we eliminated the displacement addressing mode from the MIPS ISA and only supported register indirect addressing, then we would no longer need to compute an effective address for loads and stores. We could improve the datapath by eliminating the AC (effective address calculation) stage from Ben's modified pipeline, resulting in the following stages:

IF	ID	EX/MEM	WB
Instruction fetch	Instruction decode and register fetch	Execution of ALU operations or memory access	Write-back to register file

A diagram showing the new pipeline is given below.



This new datapath does not have either of the hazards from Ben's original or modified pipelines. Thus, bubbles would not need to be inserted into the pipeline regardless of the instruction sequence, improving instruction throughput. As a side note, the latency of a single instruction has also been reduced since there are now only 4 stages instead of 5. Although this does not improve performance in the steady state, a fewer number of stages does help in that fewer pipeline registers and bypass paths are required. However, this instruction set is limited in that it only supports register indirect addressing. This means that displacement addressing would have to be synthesized from simpler instructions (see Problem 3.E).

Problem 3.E Displacement Addressing Synthesizing

Programmers could synthesize a displacement load/store instruction using the ADDi instruction, a scratch register, and the register indirect load/store instruction. For example, to synthesize the following instruction with displacement addressing:

```
LW R1, 4(R2)
```

we could use the following equivalent instruction sequence, where R3 is a temporary register:

```
ADDI R3, R2, #4
LW R1, (R3)
```

The same programs could be written as before using this technique. However, using this limited ISA may increase the number of instructions in the program as compared to the original ISA.

Problem 3.F Jumps and Branches

If Ben uses the ALU to resolve conditional branches in both his original pipeline and his modified pipeline shown in Problem M3.A, then there will be an additional cycle of branch delay in the new datapath because the ALU is now one stage later in the pipeline. If we don't worry about duplicating logic, then we can put a comparator in any stage of the pipeline (except Instruction Fetch, as the register file has not yet been read in this stage) in order to resolve conditional branches. The table shown below compares each possible placement of the comparator.

Comparator In Stage	Number of Branch Delay Cycles	Additional Stall Condition	Change in Clock Period
WB	4	None	Will remain unchanged since comparator is simpler than ALU operation so it cannot be the critical path.
EX/MEM	3	None	Will remain unchanged since comparator is simpler than ALU operation so it cannot be the critical path.
AC	2	1 cycle stall when the ALU output or result of a load is used for the branch	Will remain unchanged since comparator is simpler than ALU operation so it cannot be the critical path.
ID	1	2 cycle stall when the ALU output or result of a load is used for the branch	Will likely **increase** the clock period since it now could be on the critical path (get register value + comparator)

Obviously placing the comparator in the Write-Back stage makes no sense since this doesn't provide an advantage over placing the comparator in the Execute/Memory stage, and in fact increases the number of branch delay cycles by 1. Placing the comparator in the Address Calculation stage instead of the Execute/Memory stage reduces the number of branch delay cycles by 1, but introduces a potential stall condition. Since the branch delay affects all branches, while the stall condition would only affect some of the branches, placing the comparator in the Address Calculation stage is to be preferred over the Execute/Memory stage. Finally, the comparator could be placed in the Instruction Decode stage. If this doesn't lengthen the critical path, then this would be the best placement, as the number of branch delay cycles is reduced to 1. However, if it does lengthen the critical path—and it likely will—then the increased cycle time would probably not be worth the reduction in the branch delay, as now *all* instructions will run more slowly.