CS152 Computer Architecture and Engineering

January 26, 2011

ISAs, Microprogramming and Pipelining

Assigned January 26 Problem Set #1 *Due February 9*

http://inst.eecs.berkeley.edu/~cs152/sp11

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.

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.

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

In this problem, your task is to compare three different ISAs. x86 is an extended accumulator, CISC architecture with variable-length instructions. MIPS64 is a load-store, RISC architecture with fixed-length instructions. We will also look at a simple stack-based ISA.

Problem 1.A CISC

Let us begin by considering the following C code:

```
int b;  //a global variable

void multiplyByB(int a){
  int i, result;
  for(i = 0; i<b; i++){
    result=result+a;
  }
}
```

Using gcc and objdump on a Pentium III, we see that the above loop compiles to the following x86 instruction sequence. (On entry to this code, register `%ecx` contains `i`, and register `%edx` contains `result`, and register `%eax` contains `a`. `b` is stored in memory at location `0x8049580`)

```
            xor    %edx,%edx
            xor    %ecx,%ecx
 loop:      cmp    0x8049580,%ecx
            jl     L1
            jmp    done
 L1:        add    %eax,%edx
            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 $\langle R_{SUBSCRIPT} \rangle$.

Instruction	Operation	Length
add R_{DEST}, R_{SRC}	$R_{SRC} \leftarrow \langle R_{SRC} \rangle + \langle R_{DST} \rangle$	2 bytes
cmp imm32, R_{SRC2}	Temp $\leftarrow \langle R_{SRC2} \rangle$ - MEM[imm32]	6 bytes
inc R_{DEST}	$R_{DEST} \leftarrow \langle R_{DEST} \rangle + 1$	1 byte
jmp label	jump to the address specified by `label`	2 bytes
jl label	if (SF≠OF) jump to the address specified by `label`	2 bytes
xor R_{DEST}, R_{SRC}	$R_{DEST} \leftarrow R_{DEST} \otimes R_{SRC}$	2 bytes

Notice that the jump instruction `jl` (jump if less than) depends on SF and OF, which are status flags. Status flags, also known as condition codes, are analogous to the condition register used in the MIPS architecture. Status flags are set by the instruction preceding the jump, based on the result of the computation. Some instructions, like the `cmp` 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
OF	Overflow	Result exceeds positive or negative limit of number range
SF	Sign	Result is negative (less than zero)

How many bytes is the program? For the above x86 assembly code, how many bytes of instructions need to be fetched if b = 10? Assuming 32-bit data values, how many bytes of data memory need to be fetched? Stored?

Problem 1.B RISC

Translate each of the x86 instructions in the following table into one or more MIPS64 instructions. Place the L1 and loop labels where appropriate. You should use the minimum number of instructions needed to translate each x86 instruction. Assume that upon entry, R1 contains `b`, R2 contains `a`, R3 contains `i`. R4 should receive `result`. If needed, use R5 as a condition register, and R6, R7, etc., for temporaries. You should not need to use any floating-point registers or instructions in your code. A description of the MIPS64 instruction set architecture can be found in Appendix B of Hennessy & Patterson.

x86 instruction		label	MIPS64 instruction sequence
xor	%edx,%edx		
xor	%ecx,%ecx		
cmp	0x8049580,%ecx		
jl	L1		
jmp	done		
add	%eax,%edx		
inc	%ecx		
jmp	loop		
...		*done:*	...

How many bytes is the MIPS64 program using your direct translation? How many bytes of MIPS64 instructions need to be fetched for b = 10 using your direct translation? Assuming 32-bit data values, how many bytes of data memory need to be fetched? 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. The table below gives a subset of a simple stack-style instruction set. Assume each opcode is a single byte. Labels, constants, and addresses require two bytes.

Example instruction	Meaning
PUSH A	push M[A] onto stack
POP A	pop stack and place popped value in M[A]
ADD	pop two values from the stack; ADD them; push result onto stack
SUB	pop two values from the stack; SUBtract top value from the 2nd; push result onto stack
ZERO	zeroes out the value at top of stack
INC	pop value from top of stack; increments value by one push new value back on the 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
GOTO *label*	continue execution at location *label*

Translate the `multiplyByB` 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, `a` is the only thing on the stack. Assume `b` is still at address 0x8000 (to fit within a 2 byte address specifier).

How many bytes is your program? Using your stack translations from part (c), how many bytes of stack instructions need to be fetched for b = 10? Assuming 32-bit data values, how many bytes of data memory need to be fetched? Stored? 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?

Problem 1.D Conclusions

In just a few sentences, compare the three ISAs you have studied with respect to code size, number of instructions fetched, and data memory traffic.

Problem 1.E Optimization

To get more practice with MIPS64, 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 MIPS machine described in Handout #1 (Bus-Based MIPS 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:

$$M[r_d] \leftarrow M[r_s] + M[r_t]$$

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).

State	PseudoCode	IdIR	Reg Sel	Reg Wr	en Reg	IdA	IdB	ALUOp	en ALU	Id 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	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:																

Worksheet 2.A

Problem 2.B — Implementing MOVN Instruction

MOVN stands for Move Conditional on Not Zero. This instruction uses the same encoding as the other arithmetic instructions (R-type) on MIPS:

opcode	rs	rt	rd	...	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

The MOVN instruction has the following format:

MOVN r_d, r_s, r_t

And it performs following operation:

$r_d \leftarrow (r_t)\ ?\ r_s : r_d$

If the value in GPR r_t is not equal to zero, then the contents of GPR r_s are placed into GPR r_d.

Your task is to fill out Worksheet 2.B for MOVN 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. You do not have to worry about the busy signal.

State	PseudoCode	ldIR	Reg Sel	Reg Wr	en Reg	ldA	ldB	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; dispatch	0	PC	1	1	*	*	INC_A_4	1	*	*	0	*	0	D	*
. . .																
NOP0:	microbranch back to FETCH0	*	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH0
MOVN:																

Worksheet 2.B

Problem 2.C **Instruction Execution Times**

How many cycles does it take to execute the following instructions in the microcoded MIPS machine? Use the states and control points from MIPS-Controller-2 in Lecture 2 and assume Memory will not assert its busy signal.

Instruction	Cycles
`SUB  R3,R2,R1`	
`SUBI R2,R1,#4`	
`SW   R1,0(R2)`	
`BEQZ R1,label  # (R1 == 0)`	
`BNEZ R1,label  # (R1 != 0)`	
`J    label`	
`JR   R1`	
`JAL  label`	
`JALR R1`	

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 MIPS 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.***

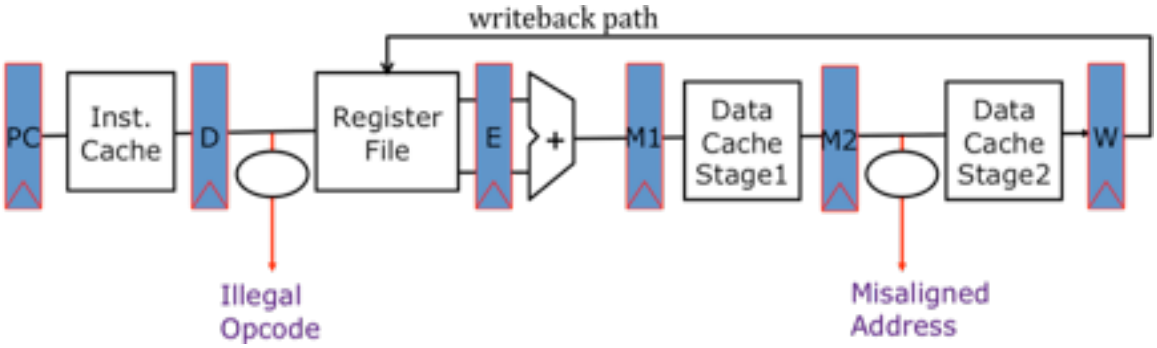

Figure 1-A. 6-stage pipeline. For clarity, bypass paths are not shown.

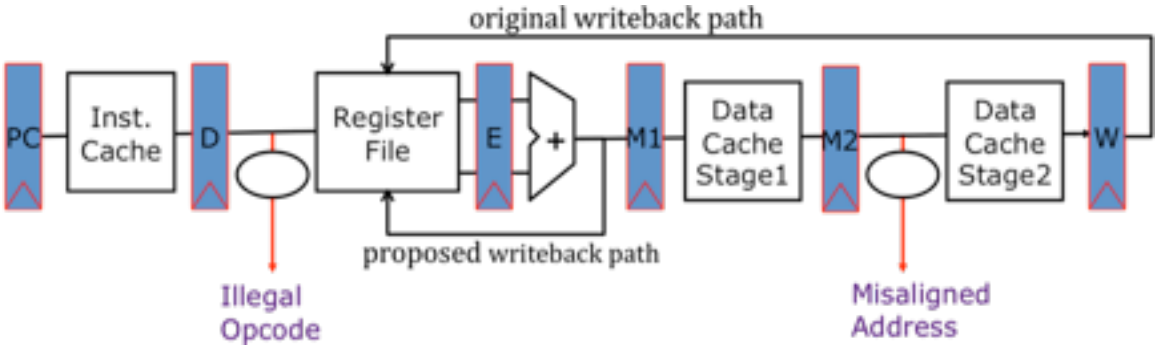

Figure 1-B. 6-stage pipeline with proposed additional write port.

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**

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.

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 2011 (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				

e)	Adding 16-bit versions of the most common instructions in MIPS (normally 32-bits in length) to the ISA (i.e., make MIPS a variable length ISA)				
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)				