CS152 Computer Architecture and Engineering

January 25, 2010

ISAs, Microprogramming and Pipelining
Problem Set #1

Assigned January 26

Due February 11

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 $<R_{SUBSCRIPT}>$.

Instruction	Operation	Length
add R_{DEST}, R_{SRC}	$R_{SRC} \leftarrow <R_{SRC}> + <R_{DST}>$	2 bytes
cmp imm32, R_{SRC2}	Temp $\leftarrow <R_{SRC2}> - MEM[imm32]$	6 bytes
inc R_{DEST}	$R_{DEST} \leftarrow <R_{DEST}> + 1$	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:

$$\text{ADDm } r_d, r_s, r_t$$

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	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	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

DBNEZ stands for Decrease Branch Not Equal Zero. This instruction uses the same encoding as conditional branch instructions (I-Format) on MIPS:

opcode	rs	…	offset
6 bits	5 bits	5 bits	16 bits

DBNEZ decrements register **rs** by 1, writes the result back to **rs**, and branches to (**PC+4**)+**offset**, if the result in **rs** is not equal to 0. Offset is sign extended to allow for backward branches. This instruction can be used for efficiently implementing loops.

Your task is to fill out Worksheet 2.B for DBNEZ 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.

(Note that the microcode for the fetch stage has changed slightly from the one in the Problem 2.A, to allow for more efficient implementation of some instructions.)

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 4 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?

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; 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:																

Worksheet 2.B

Problem 3: Mem-ALU Pipeline

In this problem, we consider further modifications to the fully bypassed 5-stage MIPS processor pipeline presented in Lecture 4. We will re-order the stages so the Execute (ALU) stage comes after the Memory stage. After this change we will only support register-indirect addressing. This change will let us use the contents of memory as one of the operands for arithmetic operations. For example, something like the CAdd instruction could be implemented:

$$\text{CAdd} \ \ r_d, r_{s0}, r_{s1}$$

CAdd performs the following operation:

$$r_d \leftarrow M[r_{s0}] + r_{s1}$$

In this problem, assume that the control logic is optimized to stall only when necessary, and that the pipeline is fully bypassed. *Ignore control-transfer instructions (jumps and branches).*

Besides enabling the CAdd instruction, this pipeline modification will change which hazards exist in the pipeline. We want to compare the pipeline from lecture (old) to this modified pipeline with the ALU after memory (new). Assume both the old and new pipelines are fully bypassed with correct control logic. *For each problem below, give a sample instruction sequence to clarify your explanation.*

Problem 3.A   **Elimination of a Hazard**

Give a minimal sequence of MIPS instructions that would cause a pipeline bubble in the original datapath, but not in the new datapath.

Problem 3.B   **New Hazard**

Give a minimal sequence of MIPS instructions that would cause a pipeline bubble in the new datapath, but not in the original datapath.

Problem 3.C   **Comparison**

Compare the advantages and disadvantages of the new datapath. Which one would you recommend? Justify your choice.

Problem 3.D   **Memory Addressing Modes**

As an architect you desire full compatibility with the old ISA, including support for register-immediate indirect addressing (e.g., LW r2, 8(r3)). How would you modify the new pipeline (and associated control logic) to support this efficiently?

Problem 3.E   **Precise Exceptions**

Describe a problem that might arise when implementing precise exceptions and propose a simple solution.

Problem 4: ISA Visibility

The following questions describe two variants of a processor that are otherwise identical. In each case, state whether or not the difference is visible to software written in the ISA. Briefly explain your reasoning. *Ignore differences in performance.*

Problem 4.A	Pipeline Depth

Pipelined processor A has more stages than pipelined Processor B, but both have full bypassing.

Problem 4.B	Control Type

Processor A uses microcoded control while Processor B uses hardwired control.

Problem 4.C	CISC/RISC

Processor A is considered to be a CISC machine while Processor B is a RISC machine.

Problem 4.D	Microcode Type

Machine A has very vertical microcode while machine B has a more horizontal microcode.

Problem 4.E	Stack Depth

Stack machine A has more physical registers to implement its stack than stack machine B.

Problem 4.F	Delay Slot

Processor A has a branch delay slot, while Processor B does not.

Problem 5: Iron Law of Processor Performance

Mark whether the following modifications will cause each of the categories to **increase, decrease**, or whether the modification will have **no effect**. Explain your reasoning.

	Instructions / Program	Cycles / Instruction	Seconds / Cycle
Dividing up a pipeline stage into two stages			
Adding a complex instruction			
Reducing the number of bypass paths			
Improving memory access speed			