CS152 Quiz 1 Answer Guide
Distributed 2/25/2008

Problem Q.1: Microprogramming Bus-Based Architectures

[28 points]

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), which we have included at the end of this quiz for your reference. 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.

You are to implement in microcode a double indirect addressing mode, as described below. In this addressing mode, the source register contains a pointer to a location in memory whose value is a pointer to the location in memory whose value is to be loaded. The instruction has the following format:

LWmm r_d, r_s

LWmm performs the following operation:

$r_d \leftarrow M[\, M[\, r_s \,]\,]$

Fill in Worksheet Q1-1 with the microcode for LWmm. 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 in the Handout).

State	PseudoCode	ld IR	Reg Sel	Reg W	en Reg	ld A	ld B	ALUOp	en ALU	ld MA	Mem W	en Mem	Ex Sel	en Imm	μB r	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
LWMM0:	MA <- R[rs]	*	rs	0	1	*	*	*	0	1	*	0	*	0	N	*
	MA <- Mem	*	*	*	0	*	*	*	0	1	0	1	*	0	N	*
	R[rd] <- Mem; ubranch back to fetch	*	rd	1	1	*	*	*	0	*	0	1	*	0	J	FETCH0

Worksheet Q1-1

Problem Q2: Dual ALU Pipeline [33 points]

Problem Q2.A **ALU Usage**

	ALU1 or ALU2?
`add r1, r2, r3`	**ALU1**
`lw r4, 0(r1)`	
`add r5, r4, r6`	**ALU2**
`add r7, r5, r8`	**ALU2**
`add r1, r2, r3`	**ALU1**
`lw r4, 0(r1)`	
`add r5, r1, r6`	**ALU1**

The following timeline diagrams the execution of the instructions, with the stage where each instruction produces its result highlighted in bold, and the bypassing between instructions shown by arrows.

add_1	IF	ID	**EX1**	EX2	WB						
lw_1		IF	ID	EX1	**MEM**	WB					
add_2			IF	ID	EX1	**EX2**	WB				
add_3				IF	ID	EX1	**EX2**	WB			
add_4					IF	ID	**EX1**	EX2	WB		
lw_2						IF	ID	EX1	**MEM**	WB	
add_5							IF	ID	**EX1**	EX2	WB

The pipeline is initially idle, so the first add reads its operands from the register file in ID and uses ALU1. The second add uses the result of the lw which is not available by the end of ID; therefore the add uses ALU2, and the load data is bypassed to it at the end of EX1. The third add uses the result of the second, so its data is not available by the end of ID; it also uses ALU2, allowing the data to be bypassed to it at the end of EX1. The fourth add has no dependencies on the previous instructions; it reads its operands from the register file in ID and uses ALU1. The fifth add uses the result of the fourth add. This value is bypassed to it at the end of ID from EX2/MEM, and it uses ALU1.

	stall? (yes/no)	explanation
`add` **`r1,`** `r2, r3` `lw r4, 0(`**`r1`**`)`	**No**	**The add (in EX1) uses ALU1 and bypasses its result to the LW (in ID).**

`lw` **`r1,`** `0(r2)`
`add r3,` **`r1,`** `r4`
`lw r5, 0(`**`r1`**`)`
	No	**The first LW (in EX2/MEM) bypasses its result to the add (in EX1) which will use ALU2, and also to the second LW (in ID).**

`lw` **`r1,`** `0(r2)` `lw r3, 0(`**`r1`**`)`	**Yes**	**The result of the first LW (in EX1) is not available in time for the second LW (in ID), so the second LW must stall.**
`lw` **`r1,`** `0(r2)` `sw` **`r1,`** `0(r3)`	**No**	**The LW (in EX2/MEM) bypasses its result to the SW (in EX1) in time for it to store the data in EX2/MEM.**

`lw` **`r1,`** `0(r2)`
`add` **`r3, r1,`** `r4`
`sw r5, 0(`**`r3`**`)`
	Yes	**The LW (in EX2/MEM) bypasses its result to the add (in EX1) which will use ALU2. But, the result of the add (in EX1) is not available in time for the SW (in ID), so the SW must stall.**

`lw` **`r1,`** `0(r2)` `add r3,` **`r1,`** `r4`	**No**	**The LW (in EX2/MEM) bypasses its result to the add (in EX1) which will use ALU2.**

Note that the base address operand for both LW and SW must be available by the end of ID, but the data operand for SW must only be available by the end of EX1.

Problem Q3: Processor Design (Short Yes/No Questions) [10 points]

The following questions describe two variants of a processor which are otherwise identical. In each case, circle "**Yes**" if the variants might generate different results from the same compiled program, and circle "**No**" otherwise. You must also briefly explain your reasoning. Ignore differences in the time each machine takes to execute the program.

Problem Q3.A **Interlock vs. Bypassing**

No. Data dependencies are preserved with either interlocks or bypassing, so the processors always generate the same results. Bypassing improves performance by eliminating stalls.

Problem Q3.B **Delay Slot**

Yes. The instruction following a taken branch is executed on processor A, but killed on processor B; so, the processors can generate different results.

Problem Q3.C **Structural Hazard**

No. Both processors retrieve the same data values. There is only a performance difference because processor A must stall an instruction fetch to allow a load instruction to access memory.

Problem Q3.D **Microcode size**

No. A wide variety of possible microded machines can implement the same user-level ISA semantics and generate the same results for all programs.

Problem Q3.E **Register Size**

Either answer, depending on assumptions about microcode & ISA changes.
No: With appropriate microcode, both machines could generate identical results for a 32-bit ISA. Also, machine A could implement a 64-bit ISA using two 32-bit registers for each 64-bit value and carefully handling overflow conditions.
Yes: Assuming microcode was literally unchanged, the machines would generate different results due to the different overflow properties of 32-bit and 64-bit registers. For example, if a value is shifted left, bits are lost using 32-bit registers that are retained with 64-bit registers.

Problem Q.4: Iron Law of Processor Performance (Short Answer) [8 points]

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** to receive credit.

	Instructions / Program	Cycles / Instruction	Seconds / Cycle	Reasoning?
Combining two pipeline stages	No effect. No change is made to the ISA, so the program remains the same.	Decrease. Fewer possible pipeline hazards between instructions.	May increase. If combined stage makes critical path longer, cycle time may have to increase.	
Removing a complex instruction	May increase. If program used this instruction, the compiler will have to replace it with several simple ones	May decrease. If complex instruction took more cycles than the others, overall CPI will decrease	May decrease. Complicated hardware may be removed as well	
Running the machine at a higher clock frequency	No effect. ISA is unchanged	No effect. Pipeline is unchanged	Decrease. Clock frequency increase means each cycle takes fewer seconds	
Using a better compiler	Usually decrease as improved compiler will generate more concise code, but could increase if more, simpler instructions reduced hazards	May decrease as better compiler scheduling can avoid hazards from load-use delay slots, and branch delay slots.	No effect. Underlying hardware is unchanged.	