Exercise Class

Fall, 2017

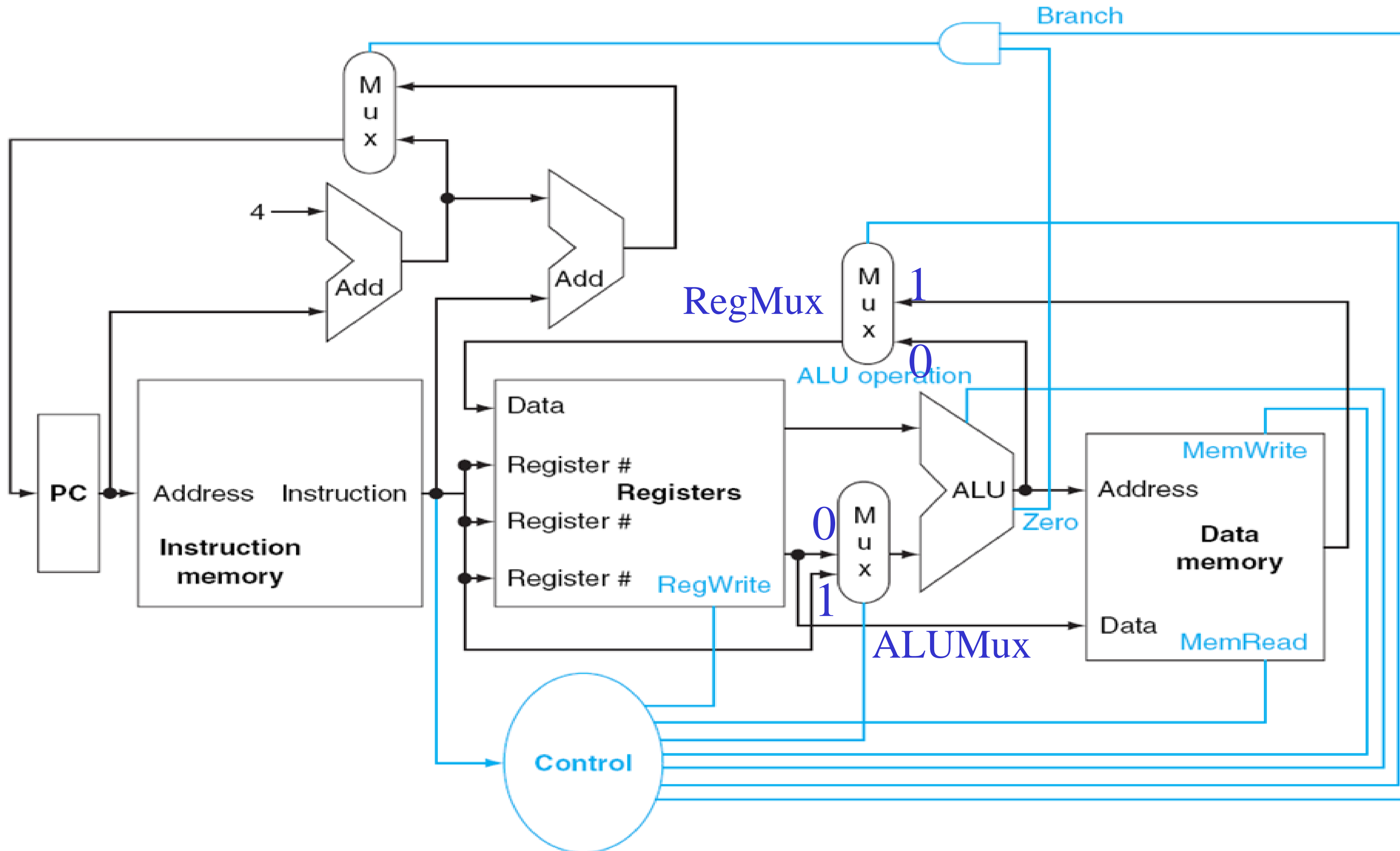
Attention, please!

- Midterm Exam is scheduled on Monday Oct. 23, 2017 in class.

- We will have a Midterm Preview class on Oct. 18, 2017.

Different instructions utilize different hardware blocks in the basic single-cycle implementation. The next three problems in this question refer to the following instruction:

	Instruction	Interpretation
a.	add R_d , R_s , R_t	$Reg[R_d]=Reg[R_s]+Reg[R_t]$
b.	lw R_t , Offs(R_s)	$Reg[R_t]=Mem[Reg[R_s]+Offs]$



Question 1- (1)

What are the values of control signals generated by the control in the figure on last slide for this instruction? (hint: just write "Add" for "addition operation" for "ALUOp")

	RegWrite	MemRead	ALUMux	MemWrite	ALUOp	RegMux	Branch
a.							
b.							

ALUMux is the control signal that controls the Mux at the ALU input, 0 (Reg) selects the output of the register file and 1 (Imm) selects the immediate from the instruction word as the second input to the ALU.

RegMux is the control signal that controls the Mux at the Data input to the register file, 0 (ALU) selects the output of the ALU and 1 (Mem) selects the output of memory.

A value of X is a "don't care" (does not matter if signal is 0 or 1)

Question 1- (2)

Which resources (blocks) perform a useful function for this instruction?

a.	
b.	

Question 1- (3)

1. Which resources (blocks) produce outputs, but their outputs are not used for this instruction?
2. Which resources produce no outputs for this instruction?

	Outputs that are not used	No outputs
a.		
b.		

Solution

1 The values of the signals are as follows:

	RegWrite	MemRead	ALUMux	MemWrite	ALUOp	RegMux	Branch
a.	1	0	0 (Reg)	0	Add	0 (ALU)	0
b.	1	1	1 (Imm)	0	Add	1 (Mem)	0

ALUMux is the control signal that controls the Mux at the ALU input, 0 (Reg) selects the output of the register file and 1 (Imm) selects the immediate from the instruction word as the second input to the ALU.

RegMux is the control signal that controls the Mux at the Data input to the register file, 0 (ALU) selects the output of the ALU and 1 (Mem) selects the output of memory.

A value of X is a “don’t care” (does not matter if signal is 0 or 1)

2 Resources performing a useful function for this instruction are:

a.	All except Data Memory and branch Add unit
b.	All except branch Add unit and second read port of the Registers

3

	Outputs that are not used	No outputs
a.	Branch Add	Data Memory
b.	Branch Add, second read port of Registers	None (all units produce outputs)

Question 2

In this exercise, we examine how data dependences affect execution in the basic five-stage pipeline

please indicate data dependences and their type for each of the following two sequences of instructions. Note that one example of dependency in code segment a “RAW on $1 from I1 to I3” was already given. Please follow the format of this example dependency and write all other existing data dependencies in code segment “a” and code segment “b”.

a.	I1: lw $1,40($6) I2: add $6,$2,$2 I3: sw $6,50($1)	RAW on $1 from I1 to I3
b.	I1: lw $5,-16($5) I2: sw $5,-16($5) I3: add $5,$5,$5	

Question 2 –(2)

Assume there is no forwarding in this pipelined processor. Indicate hazards and add *nop* instructions to eliminate them

a.	lw $1,40($6)	
b.	lw $5,-16($5)	

Question 2 – (3)

Assume there is full forwarding. Indicate hazards and add *nop* instructions to eliminate them

	Instruction sequence	
a.		
b.		

Solution

1

	Instruction sequence	Dependences

a.	`I1: lw $1,40($6)`
	`I2: add $6,$2,$2`
	`I3: sw $6,50($1)`
		RAW on $1 from I1 to I3
		RAW on $6 from I2 to I3
		WAR on $6 from I1 to I2

b.	`I1: lw $5,-16($5)`
	`I2: sw $5,-16($5)`
	`I3: add $5,$5,$5`
		RAW on $5 from I1 to I2 and I3
		WAR on $5 from I1 and I2 to I3
		WAW on $5 from I1 to I3

2 In the basic five-stage pipeline WAR and WAW dependences do not cause any hazards. Without forwarding, any RAW dependence between an instruction and the next two instructions (if register read happens in the second half of the clock cycle and the register write happens in the first half). The code that eliminates these hazards by inserting `nop` instructions is:

Solution

	Instruction sequence	

a.	lw $1,40($6)
	add $6,$2,$2
	nop
	nop
	sw $6,50($1)
		Delay I3 to avoid RAW hazard on $1 from I1

b.	lw $5,-16($5)
	nop
	nop
	sw $5,-16($5)
	add $5,$5,$5
		Delay I2 to avoid RAW hazard on $5 from I1
		
		Note: no RAW hazard from on $5 from I1 now

3 With full forwarding, an ALU instruction can forward a value to EX stage of the next instruction without a hazard. However, a load cannot forward to the EX stage of the next instruction (by can to the instruction after that). The code that eliminates these hazards by inserting `nop` instructions is:

	Instruction sequence	

a.	lw $1,40($6)
	add $6,$2,$2
	sw $6,50($1)
		No RAW hazard on $1 from I1 (forwarded)

b.	lw $5,-16($5)
	nop
	sw $5,-16($5)
	add $5,$5,$5
		Delay I2 to avoid RAW hazard on $5 from I1
		Value for $5 is forwarded from I2 now
		Note: no RAW hazard from on $5 from I1 now

Question 3

In this exercise, we examine how resource hazards, control hazards, and ISA design can affect pipelined execution. Problems in this exercise refer to the following fragment of MIPS code:

	Instruction sequence
a.	lw $1, 40 ($6) beq $2, $0, Label ; Assume $2 == $0 sw $6, 50 ($2) Label: add $2, $3, $4 sw $3, 50 ($4)
b.	lw $5, -16 ($5) sw $4, -16 ($4) lw $3, -20 ($4) beq $2, $0, Label ; Assume $2 != $0 add $5, $1, $4

For this problem, assume that all branches are perfectly predicted (this eliminates all control hazards) and that no delay slots are used. If we only have one memory (for both instructions and data), there is a structural hazard every time we need to fetch an instruction in the same cycle in which another instruction accesses data. To guarantee forward progress, this hazard must always be resolved in favor of the instruction that accesses data. What is the total execution time of this instruction sequence in the five-stage pipeline that only has one memory? We have seen that data hazards can be eliminated by adding *nops* to the code. Can you do the same with this structural hazard? Why?

Question 3 – (1)

Note that in the pipelined execution that you will show below, use *** to represent a stall when an instruction cannot be fetched because a load or store instruction is using the memory in that cycle. Cycles are represented from left to right, and for each instruction we show the pipeline stage it is in during that cycle:

	Instruction	Pipeline stage	Cycles

a.	lw $1,40($6)
	beq $2,$0,Lbl
	add $2,$3,$4
	sw $3,50($4)
		IF ID EX MEM WB	

b.	lw $5,-16($5)
	sw $4,-16($4)
	lw $3,-20($4)
	beq $2,$0,Lbl
	add $5,$1,$4
		IF ID EX MEM WB	

Solution

1 In the pipelined execution shown below, *** represents a stall when an instruction cannot be fetched because a load or store instruction is using the memory in that cycle. Cycles are represented from left to right, and for each instruction we show the pipeline stage it is in during that cycle:

	Instruction	Pipeline stage	Cycles

a.	`lw $1,40($6)`
	`beq $2,$0,Lbl`
	`add $2,$3,$4`
	`sw $3,50($4)`
		`IF ID EX MEM WB`
		`   IF ED EX MEM WB`
		`      IF ID EX MEM WB`
		`         *** IF ID EX MEM WB`
			9

b.	`lw $5,-16($5)`
	`sw $4,-16($4)`
	`lw $3,-20($4)`
	`beq $2,$0,Lbl`
	`add $5,$1,$4`
		`IF ID EX MEM WB`
		`   IF ED EX MEM WB`
		`      IF ID EX MEM WB`
		`         *** *** *** IF ID EX MEM WB`
		`                        IF ID EX MEM WB`
			12

We can not add `nops` to the code to eliminate this hazard—`nops` need to be fetched just like any other instructions, so this hazard must be addressed with a hardware hazard detection unit in the processor.

Question 4

Problem in this exercise assumes that instructions executed by a pipelined processor are broken down as follows:

	ADD	**BEQ**	**LW**	**SW**
a.	50%	25%	15%	10%
b.	30%	15%	35%	20%

Question 4 – (1)

- Assuming there are no stalls and that 60% of all conditional branches are taken, in what percentage of clock cycles does the branch adder in the EX stage generate a value that is actually used?

1 Of all these instructions, the value produced by this adder is actually used only by a `beq` instruction when the branch is taken. We have:

a.	15% (60% of 25%)
b.	9% (60% of 15%)

Question 4 – (2)

- Assuming there are no stalls, how often (percentage of all cycles) do we use the data memory?

Of these instructions, only `lw` and `sw` use the data memory. We have:

a.	25% (15% + 10%)
b.	55% (35% + 20%)

Question 4 – (3)

- Each pipeline stage has some latency. Additionally, pipelining introduces registers between stages, and each of these adds an additional latency. The remaining problems in this exercise assume the following latencies for logic within each pipeline stage and for each register between two stages

	IF	**ID**	**EX**	**MEM**	**WB**	**Pipeline register**
a.	100 ps	120 ps	90 ps	130 ps	60 ps	10 ps
b.	180 ps	100 ps	170 ps	220 ps	60 ps	10 ps

- Assuming there are no stalls, what is the speed-up achieved by pipelining a single-cycle datapath?

The clock cycle time of a single-cycle is the sum of all latencies for the logic of all five stages. The clock cycle time of a pipelined datapath is the maximum latency of the five stage logic latencies, plus the latency of a pipeline register that keeps the results of each stage for the next stage. We have:

	Single-cycle	Pipelined	Speed-up
a.	500ps	140ps	3.57
b.	730ps	230ps	3.17

Question 5

- The importance of having a good branch predictor depends on how often conditional branches are executed. Together with branch predictor accuracy, this will determine how much is spent stalling due to mispredicted branches. In this exercise, assume that the breakdown of dynamic instructions into various instruction categories is as follows:

	R-Type	**beq**	**jmp**	**low**	**sw**
a.	50%	15%	10%	15%	10%
b.	30%	10%	5%	35%	20%

Also, assume the following branch predictor accuracies:

	Always-taken	**Always not-taken**	**2-bit**
a.	40%	60%	80%
b.	60%	40%	95%

Question 5

Stall cycles due to mispredicted branes increase the CPI. What is the extra CPI due to mispredicted branches with the always-taken predictor? Assume that branch outcomes are determined in the EX stage, that there are no data hazards, and that no delay slots are used.

Each branch that is not correctly predicted by the always-taken predictor will cause 3 stall cycles, so we have:

	Extra CPI
a.	$3 \times (1 - 0.40) \times 0.15 = 0.27$
b.	$3 \times (1 - 0.60) \times 0.10 = 0.12$