

## Exercise 4.11

In this exercise we examine in detail how an instruction is executed in a single-cycle datapath. Problems in this exercise refer to a clock cycle in which the processor fetches the following instruction word:

	Instruction word
a.	10101100011000100000000000010100
b.	00000000100000100000100000101010

**4.11.1** [5] <4.4> What are the outputs of the sign-extend and the jump “Shift left 2” unit (near the top of Figure 4.24) for this instruction word?

**4.11.2** [10] <4.4> What are the values of the ALU control unit’s inputs for this instruction?

**4.11.3** [10] <4.4> What is the new PC address after this instruction is executed? Highlight the path through which this value is determined.

The remaining problems in this exercise assume that data memory is all zeros and that the processor’s registers have the following values at the beginning of the cycle in which the above instruction word is fetched:

	R0	R1	R2	R3	R4	R5	R6	R8	R12	R31
a.	0	-1	2	-3	-4	10	6	8	2	-16
b.	0	256	-128	19	-32	13	-6	-1	16	-2

**4.11.4** [10] <4.4> For each Mux, show the values of its data output during the execution of this instruction and these register values.

**4.11.5** [10] <4.4> For the ALU and the two add units, what are their data input values?

**4.11.6** [10] <4.4> What are the values of all inputs for the “Registers” unit?

## Exercise 4.12

In this exercise, we examine how pipelining affects the clock cycle time of the processor. Problems in this exercise assume that individual stages of the datapath have the following latencies:

	IF	ID	EX	MEM	WB
a.	250ps	350ps	150ps	300ps	200ps
b.	200ps	170ps	220ps	210ps	150ps

**4.12.1** [5] <4.5> What is the clock cycle time in a pipelined and non-pipelined processor?

**4.12.2** [10] <4.5> What is the total latency of an LW instruction in a pipelined and non-pipelined processor?

**4.12.3** [10] <4.5> If we can split one stage of the pipelined datapath into two new stages, each with half the latency of the original stage, which stage would you split and what is the new clock cycle time of the processor?

The remaining problems in this exercise assume that instructions executed by the processor are broken down as follows:

	ALU	BEQ	LW	SW
a.	45%	20%	20%	15%
b.	55%	15%	15%	15%

**4.12.4** [10] <4.5> Assuming there are no stalls or hazards, what is the utilization of the data memory?

**4.12.5** [10] <4.5> Assuming there are no stalls or hazards, what is the utilization of the write-register port of the “Registers” unit?

**4.12.6** [30] <4.5> Instead of a single-cycle organization, we can use a multi-cycle organization where each instruction takes multiple cycles but one instruction finishes before another is fetched. In this organization, an instruction only goes through stages it actually needs (e.g., ST only takes 4 cycles because it does not need the WB stage). Compare clock cycle times and execution times with single-cycle, multi-cycle, and pipelined organization.

### Exercise 4.13

In this exercise, we examine how data dependences affect execution in the basic 5-stage pipeline described in Section 4.5. Problems in this exercise refer to the following sequence of instructions:

	Instruction Sequence
a.	SW R16, -100(R6) LW R4, 8(R16) ADD R5, R4, R4
b.	OR R1, R2, R3 OR R2, R1, R4 OR R1, R1, R2

**4.13.1** [10] <4.5> Indicate dependences and their type.

**4.13.2** [10] <4.5> Assume there is no forwarding in this pipelined processor. Indicate hazards and add NOP instructions to eliminate them.

**4.13.3** [10] <4.5> Assume there is full forwarding. Indicate hazards and add NOP instructions to eliminate them.

	Without Forwarding	With Full Forwarding	With ALU-ALU Forwarding Only
<b>a.</b>	250ps	300ps	290ps
<b>b.</b>	180ps	240ps	210ps

**4.13.4** [10] <4.5> What is the total execution time of this instruction sequence without forwarding and with full forwarding? What is the speedup achieved by adding full forwarding to a pipeline that had no forwarding?

**4.13.5** [10] <4.5> Add NOP instructions to this code to eliminate hazards if there is ALU-ALU forwarding only (no forwarding from the MEM to the EX stage).

**4.13.6** [10] <4.5> What is the total execution time of this instruction sequence with only ALU-ALU forwarding? What is the speedup over a no-forwarding pipeline?

## Exercise 4.14

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
<b>a.</b>	<pre> SW R16,12(R6) LW R16,8(R6) BEQ R5,R4,Label ; Assume R5 != R4 ADD R5,R1,R4 SLT R5,R15,R4 </pre>
<b>b.</b>	<pre> SW R2,0(R3) OR R1,R2,R3 BEQ R2,R0,Label ; Assume R2 == R0 OR R2,R2,R0 Label: ADD R1,R4,R3 </pre>

**4.14.1** [10] <4.5> 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 5-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?

**4.14.2** [20] <4.5> For this problem, assume that all branches are perfectly predicted (this eliminates all control hazards) and that no delay slots are used. If we change load/store instructions to use a register (without an offset) as the address, these instructions no longer need to use the ALU. As a result, MEM and EX stages can be overlapped and the pipeline has only 4 stages. Change this code to accommodate this changed ISA. Assuming this change does not affect clock cycle time, what speedup is achieved in this instruction sequence?

**4.14.3** [10] <4.5> Assuming stall-on-branch and no delay slots, what speedup is achieved on this code if branch outcomes are determined in the ID stage, relative to the execution where branch outcomes are determined in the EX stage?

The remaining problems in this exercise assume that individual pipeline stages have the following latencies:

	IF	ID	EX	MEM	WB
a.	200ps	120ps	150ps	190ps	100ps
b.	150ps	200ps	200ps	200ps	100ps

**4.14.4** [10] <4.5> Given these pipeline stage latencies, repeat the speedup calculation from 4.14.2, but take into account the (possible) change in clock cycle time. When EX and MEM are done in a single stage, most of their work can be done in parallel. As a result, the resulting EX/MEM stage has a latency that is the larger of the original two, plus 20ps needed for the work that could not be done in parallel.

**4.14.5** [10] <4.5> Given these pipeline stage latencies, repeat the speedup calculation from 4.14.3, taking into account the (possible) change in clock cycle time. Assume that the latency ID stage increases by 50% and the latency of the EX stage decreases by 10ps when branch outcome resolution is moved from EX to ID.

**4.14.6** [10] <4.5> Assuming stall-on-branch and no delay slots, what is the new clock cycle time and execution time of this instruction sequence if BEQ address

computation is moved to the MEM stage? What is the speedup from this change? Assume that the latency of the EX stage is reduced by 20ps and the latency of the MEM stage is unchanged when branch outcome resolution is moved from EX to MEM.

### Exercise 4.15

In this exercise, we examine how the ISA affects pipeline design. Problems in this exercise refer to the following new instruction:

<b>a.</b>	ADDM Rd, Rt+Offs(Rs)	Rd=Rt+Mem[Offs+Rs]
<b>b.</b>	BEQM Rd, Rt, Offs(Rs)	if Rt=Mem[Offs+Rs] then PC = Rd

**4.15.1** [20] <4.5> What must be changed in the pipelined datapath to add this instruction to the MIPS ISA?

**4.15.2** [10] <4.5> Which new control signals must be added to your pipeline from 4.15.1?

**4.15.3** [20] <4.5, 4.13> Does support for this instruction introduce any new hazards? Are stalls due to existing hazards made worse?

**4.15.4** [10] <4.5, 4.13> Give an example of where this instruction might be useful and a sequence of existing MIPS instructions that are replaced by this instruction.

**4.15.5** [10] <4.5, 4.11, 4.13> If this instruction already exists in a legacy ISA, explain how it would be executed in a modern processor like AMD Barcelona.

The last problem in this exercise assumes that each use of the new instruction replaces the given number of original instructions, that the replacement can be made once in the given number of original instructions, and that each time the new instruction is executed the given number of extra stall cycles is added to the program's execution time:

	Replaces	Once in every	Extra Stall Cycles
<b>a.</b>	2	30	2
<b>b.</b>	3	40	1

**4.15.6** [10] <4.5> What is the speedup achieved by adding this new instruction? In your calculation, assume that the CPI of the original program (without the new instruction) is 1.

## Exercise 4.16

The first three problems in this exercise refer to the following MIPS instruction:

Instruction	
a.	SW R16,-100(R6)
b.	OR R2,R1,R0

**4.16.1** [5] <4.6> As this instruction executes, what is kept in each register located between two pipeline stages?

**4.16.2** [5] <4.6> Which registers need to be read, and which registers are actually read?

**4.16.3** [5] <4.6> What does this instruction do in the EX and MEM stages?

The remaining three problems in this exercise refer to the following loop. Assume that perfect branch prediction is used (no stalls due to control hazards), that there are no delay slots, and that the pipeline has full forwarding support. Also assume that many iterations of this loop are executed before the loop exits.

Loop	
a.	Loop: ADD R1,R2,R1 LW R2,0(R1) LW R2,16(R2) SLT R1,R2,R4 BEQ R1,R9,Loop
b.	Loop: LW R1,0(R1) AND R1,R1,R2 LW R1,0(R1) LW R1,0(R1) BEQ R1,R0,Loop

**4.16.4** [10] <4.6> Show a pipeline execution diagram for the third iteration of this loop, from the cycle in which we fetch the first instruction of that iteration up to (but not including) the cycle in which we can fetch the first instruction of the next iteration. Show all instructions that are in the pipeline during these cycles (not just those from the third iteration).

**4.16.5** [10] <4.6> How often (as a percentage of all cycles) do we have a cycle in which all five pipeline stages are doing useful work?

**4.16.6** [10] <4.6> At the start of the cycle in which we fetch the first instruction of the third iteration of this loop, what is stored in the IF/ID register?

## Exercise 4.17

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

	ADD	BEQ	LW	SW
a.	40%	30%	25%	5%
b.	60%	10%	20%	10%

**4.17.1** [5] <4.6> 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?

**4.17.2** [5] <4.6> Assuming there are no stalls, how often (percentage of all cycles) do we actually need to use all three register ports (two reads and a write) in the same cycle?

**4.17.3** [5] <4.6> Assuming there are no stalls, how often (percentage of all cycles) do we use the data memory?

Each pipeline stage in [Figure 4.33](#) has some latency. Additionally, pipelining introduces registers between stages ([Figure 4.35](#)), 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.	200ps	120ps	150ps	190ps	100ps	15ps
b.	150ps	200ps	200ps	200ps	100ps	15ps

**4.17.4** [5] <4.6> Assuming there are no stalls, what is the speedup achieved by pipelining a single-cycle datapath?

**4.17.5** [10] <4.6> We can convert all load/store instructions into register-based (no offset) and put the memory access in parallel with the ALU. What is the clock cycle time if this is done in the single-cycle and in the pipelined datapath? Assume that the latency of the new EX/MEM stage is equal to the longer of their latencies.

**4.17.6** [10] <4.6> The change in 4.17.5 requires many existing LW/SW instructions to be converted into two-instruction sequences. If this is needed for 50% of these instructions, what is the overall speedup achieved by changing from the 5-stage pipeline to the 4-stage pipeline where EX and MEM are done in parallel?

## Exercise 4.18

The first three problems in this exercise refer to the execution of the following instruction in the pipelined datapath from [Figure 4.51](#), and assume the following clock cycle time, ALU latency, and Mux latency:

	Instruction	Clock Cycle Time	ALU Latency	Mux Latency
a.	LW R1, 32(R2)	50ps	30ps	15ps
b.	OR R1, R5, R6	200ps	170ps	25ps

**4.18.1** [10] <4.6> For each stage of the pipeline, what are the values of the control signals asserted by this instruction in that pipeline stage?

**4.18.2** [10] <4.6, 4.7> How much time does the control unit have to generate the ALUSrc control signal? Compare this to a single-cycle organization.

**4.18.3** What is the value of the PCSrc signal for this instruction? This signal is generated early in the MEM stage (only a single AND gate). What would be a reason in favor of doing this in the EX stage? What is the reason against doing it in the EX stage?

The remaining problems in this exercise refer to the following signals from [Figure 4.48](#):

	Signal 1	Signal 2
a.	ALUSrc	PCSrc
b.	Branch	RegWrite

**4.18.4** [5] <4.6> For each of these signals, identify the pipeline stage in which it is generated and the stage in which it is used.

**4.18.5** [5] <4.6> For which MIPS instruction(s) are both of these signals set to 1?

**4.18.6** [10] <4.6> One of these signals goes back through the pipeline. Which signal is it? Is this a time-travel paradox? Explain.

## Exercise 4.19

This exercise is intended to help you understand the cost/complexity/performance trade-offs of forwarding in a pipelined processor. Problems in this exercise refer to pipelined datapaths from [Figure 4.45](#). These problems assume that, of all the instructions executed in a processor, the following fraction of these instructions



have a particular type of RAW data dependence. The type of RAW data dependence is identified by the stage that produces the result (EX or MEM) and the instruction that consumes the result (1<sup>st</sup> instruction that follows the one that produces the result, 2<sup>nd</sup> instruction that follows, or both). We assume that the register write is done in the first half of the clock cycle and that register reads are done in the second half of the cycle, so “EX to 3<sup>rd</sup>” and “MEM to 3<sup>rd</sup>” dependences are not counted because they cannot result in data hazards. Also, assume that the CPI of the processor is 1 if there are no data hazards.

	EX to 1 <sup>st</sup> Only	MEM to 1 <sup>st</sup> Only	EX to 2 <sup>nd</sup> only	MEM to 2 <sup>nd</sup> Only	EX to 1 <sup>st</sup> and MEM to 2 <sup>nd</sup>	Other RAW Dependences
<b>a.</b>	5%	20%	5%	10%	10%	10%
<b>b.</b>	20%	10%	15%	10%	5%	0%

**4.19.1** [10] <4.7> If we use no forwarding, what fraction of cycles are we stalling due to data hazards?

**4.19.2** [5] <4.7> If we use full forwarding (forward all results that can be forwarded), what fraction of cycles are we stalling due to data hazards?

**4.19.3** [10] <4.7> Let us assume that we cannot afford to have three-input Muxes that are needed for full forwarding. We have to decide if it is better to forward only from the EX/MEM pipeline register (next-cycle forwarding) or only from the MEM/WB pipeline register (two-cycle forwarding). Which of the two options results in fewer data stall cycles?

The remaining three problems in this exercise refer to the following latencies for individual pipeline stages. For the EX stage, latencies are given separately for a processor without forwarding and for a processor with different kinds of forwarding.

	IF	ID	EX (no FW)	EX (full FW)	EX (FW from EX/MEM only)	EX (FW from MEM/WB only)	MEM	WB
<b>a.</b>	150ps	100ps	120ps	150ps	140ps	130ps	120ps	100ps
<b>b.</b>	300ps	200ps	300ps	350ps	330ps	320ps	290ps	100ps

**4.19.4** [10] <4.7> For the given hazard probabilities and pipeline stage latencies, what is the speedup achieved by adding full forwarding to a pipeline that had no forwarding?

**4.19.5** [10] <4.7> What would be the additional speedup (relative to a processor with forwarding) if we added time-travel forwarding that eliminates all data

hazards? Assume that the yet-to-be-invented time-travel circuitry adds 100ps to the latency of the full-forwarding EX stage.

**4.19.6** [20] <4.7> Repeat 4.19.3 but this time determine which of the two options results in shorter time per instruction.

## Exercise 4.20

Problems in this exercise refer to the following instruction sequences:

Instruction Sequence	
<b>a.</b>	ADD R1,R2,R1 LW R2,0(R1) LW R1,4(R1) OR R3,R1,R2
<b>b.</b>	LW R1,0(R1) AND R1,R1,R2 LW R2,0(R1) LW R1,0(R3)

**4.20.1** [5] <4.7> Find all data dependences in this instruction sequence.

**4.20.2** [10] <4.7> Find all hazards in this instruction sequence for a 5-stage pipeline with and then without forwarding.

**4.20.3** [10] <4.7> To reduce clock cycle time, we are considering a split of the MEM stage into two stages. Repeat 4.20.2 for this 6-stage pipeline.

The remaining three problems in this exercise assume that, before any of the above is executed, all values in data memory are zeroes and that registers R0 through R3 have the following initial values:

	R0	R1	R2	R3
<b>a.</b>	0	-1	31	1500
<b>b.</b>	0	4	63	3000

**4.20.4** [5] <4.7> Which value is the first one to be forwarded and what is the value it overrides?

**4.20.5** [10] <4.7> If we assume forwarding will be implemented when we design the hazard detection unit, but then we forget to actually implement forwarding, what are the final register values after this instruction sequence?

**4.20.6** [10] <4.7> For the design described in 4.20.5, add NOPs to this instruction sequence to ensure correct execution in spite of missing support for forwarding.

### Exercise 4.21

This exercise is intended to help you understand the relationship between forwarding, hazard detection, and ISA design. Problems in this exercise refer to the following sequences of instructions, and assume that it is executed on a 5-stage pipelined datapath:

Instruction sequence	
<b>a.</b>	ADD R5, R2, R1 LW R3, 4(R5) LW R2, 0(R2) OR R3, R5, R3 SW R3, 0(R5)
<b>b.</b>	LW R2, 0(R1) AND R1, R2, R1 LW R3, 0(R2) LW R1, 0(R1) SW R1, 0(R2)

**4.21.1** [5] <4.7> If there is no forwarding or hazard detection, insert NOPs to ensure correct execution.

**4.21.2** [10] <4.7> Repeat 4.21.1 but now use NOPs only when a hazard cannot be avoided by changing or rearranging these instructions. You can assume register R7 can be used to hold temporary values in your modified code.

**4.21.3** [10] <4.7> If the processor has forwarding, but we forgot to implement the hazard detection unit, what happens when this code executes?

**4.21.4** [20] <4.7> If there is forwarding, for the first five cycles during the execution of this code, specify which signals are asserted in each cycle by hazard detection and forwarding units in [Figure 4.60](#).

**4.21.5** [10] <4.7> If there is no forwarding, what new inputs and output signals do we need for the hazard detection unit in [Figure 4.60](#)? Using this instruction sequence as an example, explain why each signal is needed.

**4.21.6** [20] <4.7> For the new hazard detection unit from 4.21.5, specify which output signals it asserts in each of the first five cycles during the execution of this code.

## Exercise 4.22

This exercise is intended to help you understand the relationship between delay slots, control hazards, and branch execution in a pipelined processor. In this exercise, we assume that the following MIPS code is executed on a pipelined processor with a 5-stage pipeline, full forwarding, and a predict-taken branch predictor:

<b>a.</b>	<pre>Label1: LW  R2,0(R2)         BEQ R2,R0,Label ; Taken once, then not taken         OR  R2,R2,R3         SW  R2,0(R5)</pre>
<b>b.</b>	<pre>Label1: LW  R2,0(R1)         BEQ R2,R0,Label2 ; Not taken once, then taken         LW  R3,0(R2)         BEQ R3,R0,Label1 ; Taken         ADD R1,R3,R1 Label2: SW  R1,0(R2)</pre>

**4.22.1** [10] <4.8> Draw the pipeline execution diagram for this code, assuming there are no delay slots and that branches execute in the EX stage.

**4.22.2** [10] <4.8> Repeat 4.22.1, but assume that delay slots are used. In the given code, the instruction that follows the branch is now the delay slot instruction for that branch.

**4.22.3** [20] <4.8> One way to move the branch resolution one stage earlier is to not need an ALU operation in conditional branches. The branch instructions would be “BEZ Rd, Label” and “BNEZ Rd, Label”, and it would branch if the register has and does not have a zero value, respectively. Change this code to use these branch instructions instead of BEQ. You can assume that register R8 is available for you to use as a temporary register, and that an SEQ (set if equal) R-type instruction can be used.

Section 4.8 describes how the severity of control hazards can be reduced by moving branch execution into the ID stage. This approach involves a dedicated comparator in the ID stage, as shown in Figure 4.62. However, this approach potentially adds to the latency of the ID stage, and requires additional forwarding logic and hazard detection.

**4.22.4** [10] <4.8> Using the first branch instruction in the given code as an example, describe the hazard detection logic needed to support branch execution in the ID stage as in Figure 4.62. Which type of hazard is this new logic supposed to detect?