VE370 RC6

VE370 TA Group

Lecture Recap

- Control Hazard

HW 4.21

	Instruction sequence

a.	`ADD R5,R2,R1`
	`LW R3,4(R5)`
	`LW R2,0(R2)`
	`OR R3,R5,R3`
	`SW R3,0(R5)`

b.	`LW R2,0(R1)`
	`AND R1,R2,R1`
	`LW R3,0(R2)`
	`LW R1,0(R1)`
	`SW R1,0(R2)`

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.

4.21.4

	Instruction sequence

a.	ADD R5,R2,R1
	LW R3,4(R5)
	LW R2,0(R2)
	OR R3,R5,R3
	SW R3,0(R5)

b.	LW R2,0(R1)
	AND R1,R2,R1
	LW R3,0(R2)
	LW R1,0(R1)
	SW R1,0(R2)

PCWrite
=IF/IDWrite
=~ID/EXZero

	Instruction Sequence	First Five Cycles 1 2 3 4 5	Signals

a.	ADD R5,R2,R1
	LW R3,4(R5)
	LW R2,0(R2)
	OR R3,R5,R3
	SW R3,0(R5)
		IF ID EX MEM WB
		IF ID EX MEM
		IF ID EX
		IF ID
		IF
			1: PCWrite = 1, ALUin1 = X, ALUin2 = X
			2: PCWrite = 1, ALUin1 = X, ALUin2 = X
			3: PCWrite = 1, ALUin1 = 0, ALUin2 = 0
			4: PCWrite = 1, ALUin1 = 1, ALUin2 = 0
			5: PCWrite = 1, ALUin1 = 0, ALUin2 = 0

b.	LW R2,0(R1)
	AND R1,R2,R1
	LW R3,0(R2)
	LW R1,0(R1)
	SW R1,0(R2)
		IF ID EX MEM WB
		IF ID *** EX
		IF *** ID
		IF
			1: PCWrite = 1, ALUin1 = X, ALUin2 = X
			2: PCWrite = 1, ALUin1 = X, ALUin2 = X
			3: PCWrite = 1, ALUin1 = 0, ALUin2 = 0
			4: PCWrite = 0, ALUin1 = X, ALUin2 = X
			5: PCWrite = 1, ALUin1 = 2, ALUin2 = 0

4.21.5

- Additional Inputs:
  - register Rd from the ID/EX pipeline register
  - the output number of the output register from the EX/MEM pipeline register.
- No additional outputs are needed. We can stall the pipeline using the three output signals that we already have.

Text explanation: The instruction that is currently in the ID stage needs to be stalled if it depends on a value produced by the instruction in the EX or the instruction in the MEM stage. So we need to check the destination register of these two instructions. For the instruction in the EX stage, we need to check Rd for R-type instructions and Rd for loads. For the instruction in the MEM stage, the destination register is already selected (by the Mux in the EX stage) so we need to check that register number (this is the bottommost output of the EX/MEM pipeline register). The additional inputs to the hazard detection unit are register Rd from the ID/EX pipeline register and the output number of the output register from the EX/MEM pipeline register. The Rt field from the ID/EX register is already an input of the hazard detection unit in Figure 4.60.

No additional outputs are needed. We can stall the pipeline using the three output signals that we already have.

4.22

a.

```
Label1: LW   R2,0(R2)
        BEQ  R2,R0,Label ; Taken once, then not taken
        OR   R2,R2,R3
        SW   R2,0(R5)
```

b.

```
        LW   R2,0(R1)
Label1: 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)
```

4.22.4

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?

- Dependent of former instruction -> need the instruction from previously calculated -> ID stage while previous one not finished calculating -> **STALL**!
- Similar for load strategy.

Text explanation: The hazard detection logic must detect situations when the branch depends on the result of the previous R-type instruction, or on the result of two previous loads. When the branch uses the values of its register operands in its ID stage, the R-type instruction's result is still being generated in the EX stage. Thus we must stall the processor and repeat the ID stage of the branch in the next cycle. Similarly, if the branch depends on a load that immediately precedes it, the result of the load is only generated two cycles after the branch enters the ID stage, so we must stall the branch for two cycles. Finally, if the branch depends on a load that is the second-previous instruction, the load is completing its MEM stage when the branch is in its ID stage, so we must stall the branch for one cycle. In all three cases, the hazard is a data hazard.

Note that in all three cases we assume that the values of preceding instructions are forwarded to the ID stage of the branch if possible.

4.22.5

4.22.5 For 4.22.1 we have already shown the pipeline execution diagram for the case when branches are executed in the EX stage. The following is the pipeline diagram when branches are executed in the ID stage, including new stalls due to data dependences described for 4.22.4:

	Executed Instructions	Pipeline Cycles 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a.	LW R2,0(R2)	IF	ID	EX	MEM	WB										
	BEQ R2,R0,Label (T)		IF	***	***	ID	EX	MEB	WB							
	LW R2,0(R2)					IF	ID	EX	MEB	WB						
	BEQ R2,R0,Label (NT)						IF	***	***	ID	EX	MEB	WB			
	OR R2,R2,R3									IF	ID	EX	MEB	WB		
	SW R2,0(R5)										IF	ID	EX	MEB	WB	
b.	LW R2,0(R1)	IF	ID	EX	MEM	WB										
	BEQ R2,R0,Label2 (NT)		IF	***	***	ID	EX	MEM	WB							
	LW R3,0(R2)						IF	ID	EX	MEB	WB					
	BEQ R3,R0,Label1 (T)							IF	***	***	ID	EX	MEB	WB		
	BEQ R2,R0,Label2 (T)										IF	ID	EX	MEM	WB	
	SW R1,0(R2)											IF	ID	EX	MEB	WB

Now the speedup can be computed as:

a.	14/14 = 1
b.	14/15 = 0.93

4.23

	R-Type	BEQ	JMP	LW	SW
a.	40%	25%	5%	25%	5%
b.	60%	8%	2%	20%	10%

Also, assume the following branch predictor accuracies:

	Always-Taken	Always-Not-Taken	2-Bit
a.	45%	55%	85%
b.	65%	35%	98%

4.23.1 [10] <4.8> Stall cycles due to mispredicted branches 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.

Answer:

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

	Extra CPI
a.	$3 \times (1 - 0.45) \times 0.25 = 0.41$
b.	$3 \times (1 - 0.65) \times 0.08 = 0.08$

4.24

Exercise 4.24

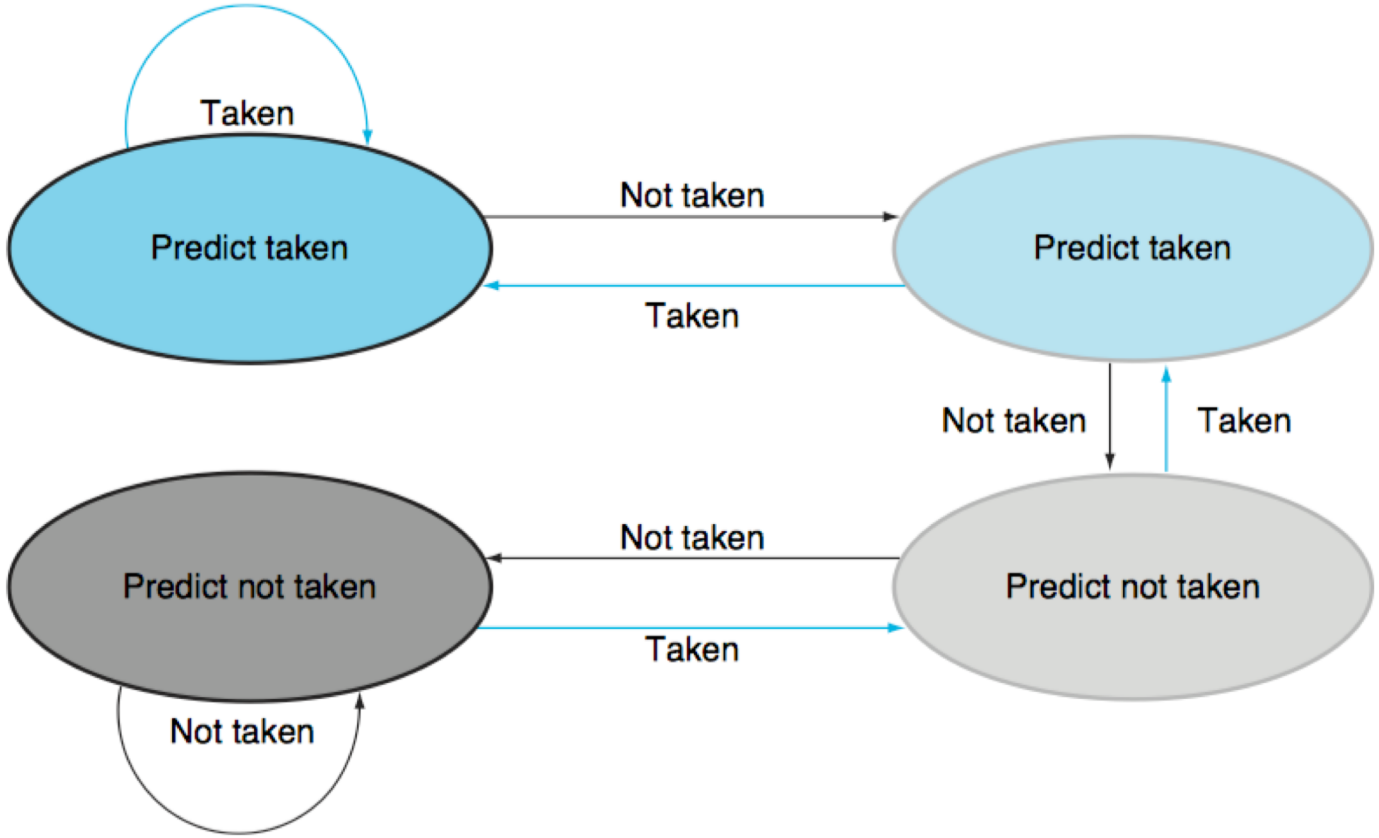
This exercise examines the accuracy of various branch predictors for the following repeating pattern (e.g., in a loop) of branch outcomes:

	Branch Outcomes
a.	T, T, NT, NT
b.	T, NT, T, T, NT

4.24.1 [5] <4.8> What is the accuracy of always-taken and always-not-taken predictors for this sequence of branch outcomes?

4.24.2 [5] <4.8> What is the accuracy of the two-bit predictor for the first 4 branches in this pattern, assuming that the predictor starts off in the bottom left state from Figure 4.63 (predict not taken)?

4.24


Taken
Predict taken
Not taken
Taken
Predict taken
Not taken
Taken
Predict not taken
Not taken
Taken
Predict not taken
Not taken

4.24

Solution 4.24

4.24.1

	Always Taken	Always Not-taken
a.	2/4 = 50%	2/4 = 50%
b.	3/5 = 60%	2/5 = 40%

4.24.2

	Outcomes	Predictor Value at Time of Prediction	Correct or Incorrect	Accuracy
a.	T, T, NT, NT	0,1,2,1	I,I,I,C	25%
b.	T, NT, T, T	0,1,0,1	I,C,I,I	25%

4.26

Exercise 4.26

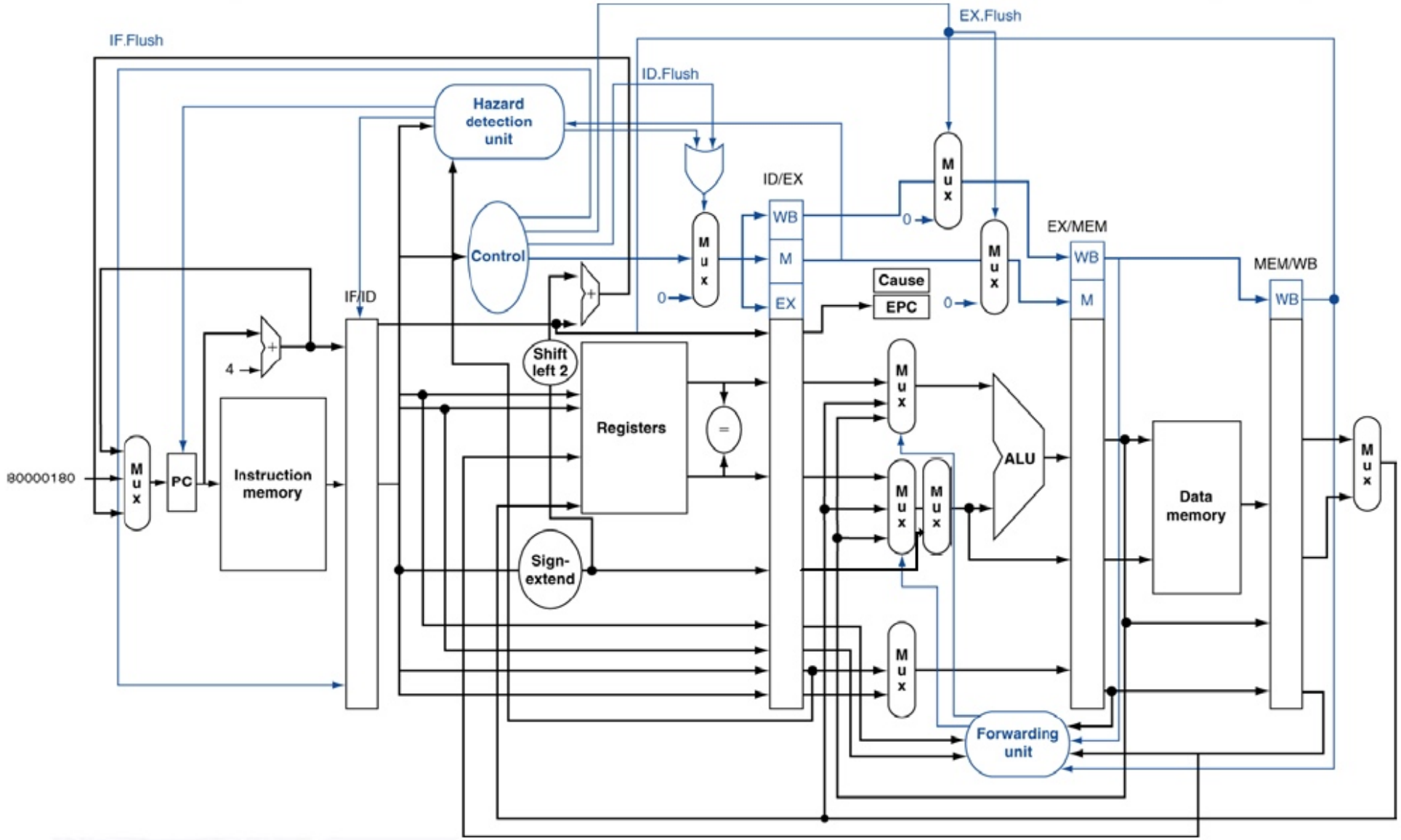
This exercise explores how exception handling affects control unit design and processor clock cycle time. The first three problems in this exercise refer to the following MIPS instruction that triggers an exception:

	Instruction	Exception
a.	`BNE R1,R2,Label`	Invalid target address
b.	`SUB R2,R4,R5`	Arithmetic overflow

4.26.1 [10] <4.9> For each stage of the pipeline, determine the values of exception-related control signals from Figure 4.66 as this instruction passes through that pipeline stage.

4.26.2 [5] <4.9> Some of the control signals generated in the ID stage are stored into the ID/EX pipeline register, and some go directly into the EX stage. Explain why, using this instruction as an example.

4.26


IF.Flush
ID.Flush
EX.Flush
Hazard detection unit
Control
ID/EX
WB
M
EX
EX/MEM
MEM/WB
Cause
EPC
IF/ID
Shift left 2
Registers
Instruction memory
PC
80000180
4
Sign-extend
ALU
Data memory
Forwarding unit
Mux
0

Solution 4.26

4.26.1 All exception-related signals are 0 in all stages, except the one in which the exception is detected. For that stage, we show values of Flush signals for various stages, and also the value of the signal that controls the Mux that supplies the PC value.

	Stage	Signals
a.	ID	IF.Flush = ID.Flush = 1, PCSel = Exc
b.	EX	IF.Flush = ID.Flush = EX.Flush = 1, PCSel = Exc

4.26.2 The signals stored in the ID/EX stage are needed to execute the instruction if there are no exceptions. Figure 4.66 does not show it, but exception conditions from various stages are also supplied as inputs to the Control unit. The signal that goes directly to EX is EX.Flush and it is based on these exception condition inputs, not on the opcode of the instruction that is in the ID stage. In particular, the EX.Flush signal becomes 1 when the instruction in the EX stage triggers an exception and must be prevented from completing.

4.27

Exercise 4.27

This exercise examines how exception handling interacts with branch and load/store instructions. Problems in this exercise refer to the following branch instruction and the corresponding delay slot instruction:

	Branch and Delay Slot
a.	`BEQ R5,R4,Label` `SLT R5,R15,R4`
b.	`BEQ R1,R0,Label` `LW R1,0(R1)`

The remaining three problems in this exercise also refer to the following store instruction:

	Store Instruction
a.	`SW R5,-40(R15)`
b.	`SW R1,0(R1)`

4.27.4 [10] <4.9> What happens if the branch is taken, the instruction at "Label" is an invalid instruction, the first instruction of the exception handler is the SW instruction given above, and this store accesses an invalid data address?

4.27.4

- The processor cancels the store instruction and other instructions (from the “Invalid instruction” exception handler) fetched after it,
- then begins fetching instructions from the invalid data address handler. A major problem here is that the new exception sets the EPC to the instruction address in the “Invalid instruction” handler, overwriting the EPC value that was already there (address for continuing the program). If the invalid data address handler repairs the problem and attempts to continue the program, the “Invalid instruction” handler will be executed. However, if it manages to repair the problem and wants to continue the program, the EPC is incorrect (it was overwritten before it could be saved). This is the reason why exception handlers must be written carefully to avoid triggering exceptions themselves, at least until they have safely saved the EPC.