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:

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 [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.

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 [10] <4.8> For the given code, what is the speedup achieved by moving branch execution into the ID stage? Explain your answer. In your speedup calculation, assume that the additional comparison in the ID stage does not affect clock cycle time.

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

Exercise 4.23

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 time 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	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.

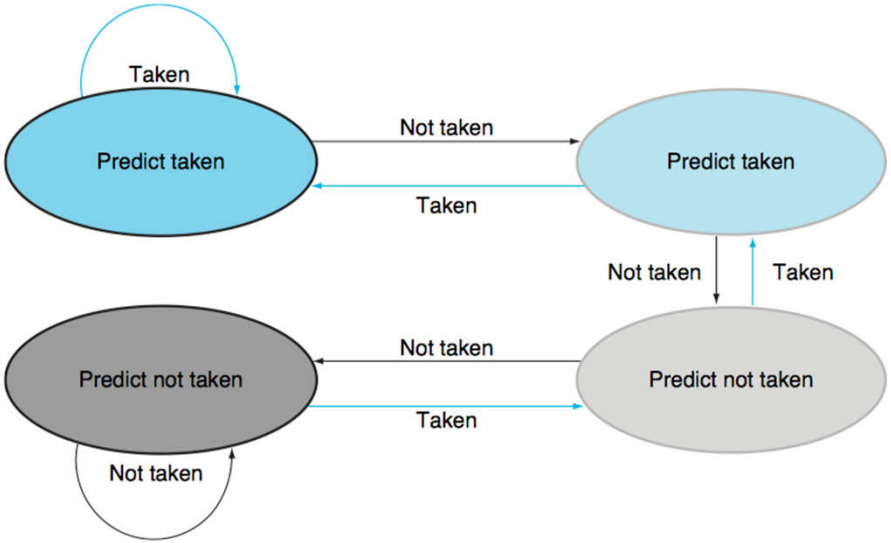
4.23.1 Each branch that is not correctly predicted by the always-taken predictor will cause 3 stall cycles, so we have:

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

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)?

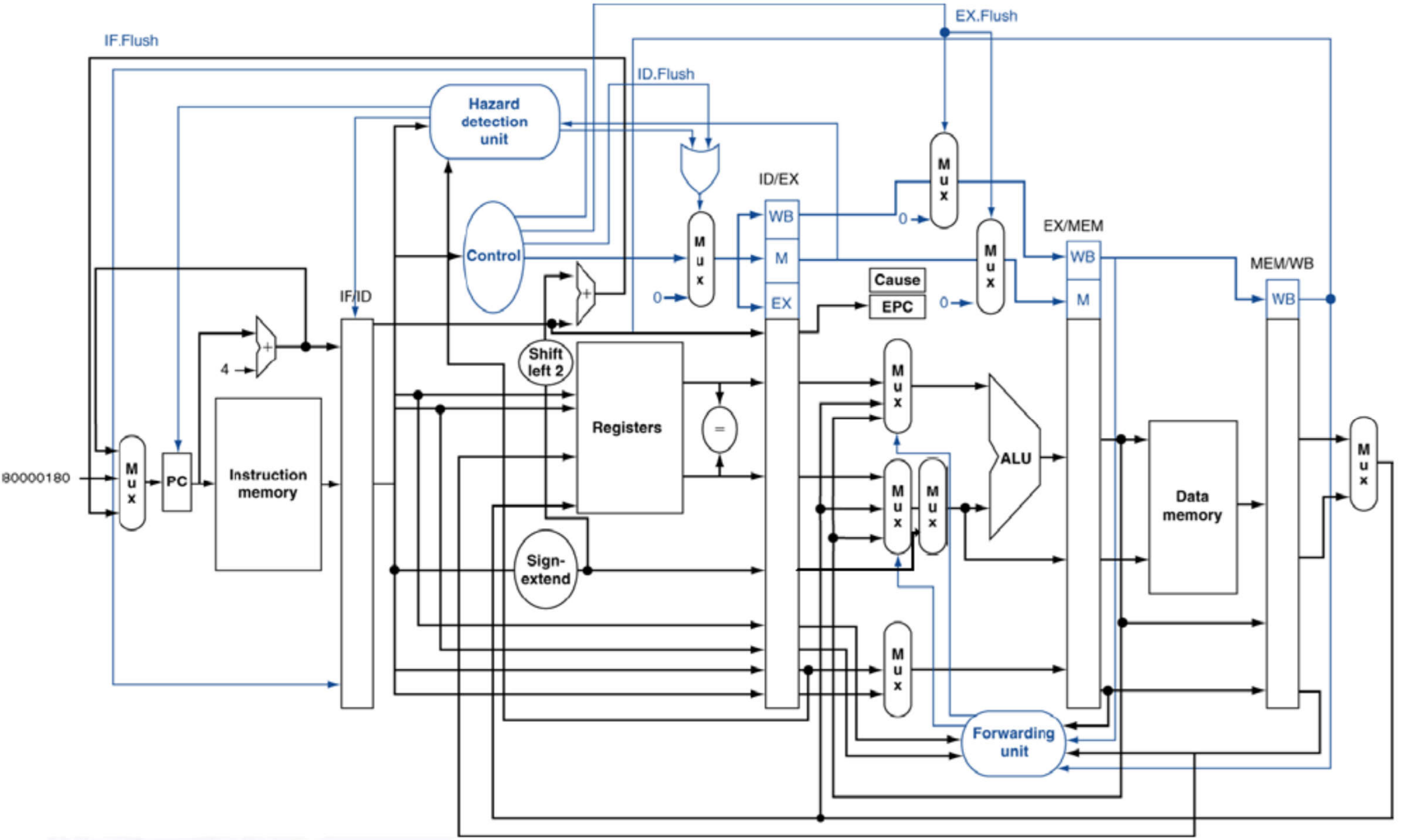
Solution 4.24

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%


IF.Flush
EX.Flush
ID.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
=

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.

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.

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.

Exercise 5.2

In this exercise we look at memory locality properties of matrix computation. The following code is written in C, where elements within the same row are stored contiguously.

a.	`for (I=0; I<8; I++)`
	`  for (J=0; J<8000; J++)`
	`  A[I][J]=B[I][0]+A[J][I];`

b.	`for (J=0; J<8000; J++)`
	`  for (I=0; I<8; I++)`
	`  A[I][J]=B[I][0]+A[J][I];`

5.2.1 [5] <5.1> How many 32-bit integers can be stored in a 16-byte cache line?

5.2.2 [5] <5.1> References to which variables exhibit temporal locality?

5.2.3 [5] <5.1> References to which variables exhibit spatial locality?

Solution:

5.2.1 4

5.2.2

a.	I, J
b.	B[I][0]

5.2.3

a.	A[I][J]
b.	A[J][I]

Exercise 5.3

Caches are important to providing a high-performance memory hierarchy to processors. Below is a list of 32-bit memory address references, given as word addresses.

a.	3, 180, 43, 2, 191, 88, 190, 14, 181, 44, 186, 253
b.	21, 166, 201, 143, 61, 166, 62, 133, 111, 143, 144, 61

5.3.1 [10] <5.2> For each of these references, identify the binary address, the tag, and the index given a direct-mapped cache with 16 one-word blocks. Also list if each reference is a hit or a miss, assuming the cache is initially empty.

5.3.2 [10] <5.2> For each of these references, identify the binary address, the tag, and the index given a direct-mapped cache with two-word blocks and a total size of 8 blocks. Also list if each reference is a hit or a miss, assuming the cache is initially empty.

For the a) part, give the number in binary

Tag Index

```
      11
10110100
  101011
      10
10111111
 1011000
10111110
    1110
10110101
  101100
10111010
11111101
```

A sample of 5.3.1

	Binary address	tag	index	h/m
3	11	0	0011	miss
180	10110100	1011	0100	miss
43	101011	10	1011	miss
2	10	0	0010	miss
191	10111111	1011	1111	miss
88	1011000	101	1000	miss
190	10111110	1011	1110	miss
14	1110	0	1110	miss
181	10110101	1011	0101	miss
44	101100	10	1100	miss
186	10111010	1011	1010	miss
253	11111101	1111	1101	miss

Index	
0000	
0001	
0010	2
0011	3
0100	180
0101	181
0110	
0111	
1000	88
1001	
1010	186
1011	43
1100	44
1101	253
1110	190→14
1111	191

A sample of 5.3.2

	Binary address	tag	index	h/m
3	11	0	001	miss
180	10110100	1011	010	miss
43	101011	10	101	miss
2	10	0	001	hit
191	10111111	1011	111	miss
88	1011000	101	100	miss
190	10111110	1011	111	hit
14	1110	0	111	miss
181	10110101	1011	010	hit
44	101100	10	110	miss
186	10111010	1011	101	miss
253	11111101	1111	110	miss

Index		
000		
001	2	3
010	180	181
011		
100	88	89
101	42→186	43→187
110	44→252	45→253
111	190→14	191→15

5.3.3 [20] <5.2, 5.3> You are asked to optimize a cache design for the given references. There are three direct-mapped cache designs possible, all with a total of 8 words of data: C1 has 1-word blocks, C2 has 2-word blocks, and C3 has 4-word blocks. In terms of miss rate, which cache design is the best? If the miss stall time is 25 cycles, and C1 has an access time of 2 cycles, C2 takes 3 cycles, and C3 takes 5 cycles, which is the best cache design?

There are many different design parameters that are important to a cache's overall performance. The table below lists parameters for different direct-mapped cache designs.

	Cache Data Size	Cache Block Size	Cache Access Time
a.	32 KB	2 words	1 cycle
b.	32 KB	4 words	2 cycle

5.3.4 [15] <5.2> Calculate the total number of bits required for the cache listed in the table, assuming a 32-bit address. Given that total size, find the total size of the closest direct-mapped cache with 16-word blocks of equal size or greater. Explain why the second cache, despite its larger data size, might provide slower performance than the first cache.

5.3.3:

C1: 0 hit stall time=27*12
C2: 2 hit stall time=25*10+3*12
C3: 1 hit stall time=25*11+5*12

5.3.4:

32 KB = 32 * 2^10 bytes

Index = $\frac{32*2^{10}\ \text{bytes}}{2\ \text{word}*4\ \frac{\text{bytes}}{\text{word}}}$ = 2^12

Tag field = 32 – (12+1+2) = 17

Total Size = 2^12 * (2*4*8 bits/bytes + 17 + 1) = 328 kb

Larger cache -> longer access time