Problem 1
Solution presented in discussion session

Problems 2 and 3
Involves the use of synthesis scripts

Problem 4

```
beq     $2, $3, foo
add     $3, $4, $5
sub     $5, $6, $7
or      $7, $8, $9
foo:    and   $5, $6, $7
```

For this problem, since "no branch prediction" is done, the pipeline is stalled until the result of the branch is known at the end of MEM stage. Note the true dependence (*) between **or** and **and** instruction.

On Branch not taken:

Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14
beq	IF	ID	EX	MEM	WB									
add		**stall**	**stall**	**Stall**	IF	ID	EX	MEM	WB					
sub						IF	ID	EX	MEM	WB				
or							IF	ID	EX	MEM	WB			
and								IF	ID*	ID*	ID*	EX	MEM	WB

On Branch taken:

Cycles	1	2	3	4	5	6	7	8	9
beq	IF	ID	EX	MEM	WB				
and		**stall**	**stall**	**stall**	IF	ID	EX	MEM	WB

Problem 5

(a)
```
add     $4, $4, $2
sub     $5, $3, $1
lw      $6, 200($3)
add     $7, $3, $6
```

Dependences

L1	**add**	$4, $4, $2	RAW on $6 from L3 to L4
L2	**sub**	$5, $3, $1	
L3	**lw**	$6, 200($3)	
L4	**add**	$7, $3, $6	

No Forwarding

L1	add	$4, $4, $2	
L2	sub	$5, $3, $1	
L3	lw	$6, 200($3)	
	NOP		
	NOP		
L4	add	$7, $3, $6	

Full Forwarding

L1	add	$4, $4, $2	
L2	sub	$5, $3, $1	
L3	lw	$6, 200($3)	
	NOP		
L4	add	$7, $3, $6	

```
(b)  lw     $1, 40($6)
     add    $6, $2, $2
     sw     $6, 50($1)
```

Dependences

L1	lw	$1, 40($6)	RAW on $1 from L1 to L3
L2	add	$6, $2, $2	RAW on $6 from L2 to L3
L3	sw	$6, 50($1)	WAR on $6 from L1 to L2 and L3

No Forwarding

L1	lw	$1, 40($6)	Delay L3 to avoid RAW hazard on $1 from L1
L2	add	$6, $2, $2	
	NOP		
L3	sw	$6, 50($1)	

Full Forwarding

L1	lw	$1, 40($6)	No RAW hazard on $1 from L1 (forwarded)
L2	add	$6, $2, $2	
L3	sw	$6, 50($1)	

```
(c)  lw    $5, -16($5)
     sw     $5, -16($5)
    add     $5, $5, $5
```

Dependences

L1	lw $5, -16($5)	RAW on $5 from L1 to L2 and L3
L2	sw $5, -16($5)	WAR on $5 from L1 and L2 to L3
L3	add $5, $5, $5	WAW on $5 from L1 to L3

No Forwarding

L1	lw $5, -16($5)	Delay L2 to avoid RAW hazard on $5 from L1
	NOP	
	NOP	
L2	sw $5, -16($5)	
L3	add $5, $5, $5	

Full Forwarding

L1	lw $5, -16($5)	Delay L2 to avoid RAW hazard on $5 from L1 Value for $5 is forwarded from L2 now
	NOP	
L2	sw $5, -16($5)	
L3	add $5, $5, $5	

Problem 6

4.24.1

	Always taken	Always not-taken
A	3 / 4 = 75%	1 / 4 = 25%
B	3 / 5 = 60%	2 / 5 = 40%

4.24.2

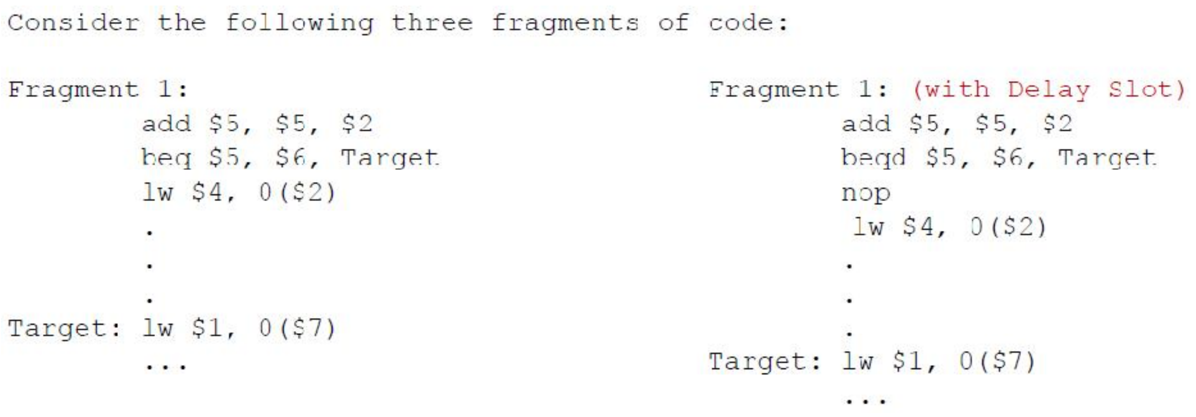
	Outcomes	Predictor value at time of prediction	Correct (C) or Incorrect (I)	Accuracy
A	T, T, NT, T	0, 1, 2, 1	I, I, I, I	0%
B	T, T, T, NT	0, 1, 2, 3	I, I, C, I	25%

4.24.3

	Outcomes	Predictor value at time of prediction	Correct (C) or Incorrect (I)	Accuracy
A	T, T, NT, T	1st occurrence: 0, 1, 2, 1 2nd occurrence: 2, 3, 3, 2 3rd occurrence: 3, 3, 3, 2 4th occurrence: 3, 3, 3, 2	C, C, I, C	75%
B	T, T, T, NT, NT	1st occurrence: 0, 1, 2, 3, 2 2nd occurrence: 1, 2, 3, 3, 2 3rd occurrence: 1, 2, 3, 3, 2	I, C, C, I, I	40%

Problem 7

➔ Fragment 1



```
Consider the following three fragments of code:

Fragment 1:
        add $5, $5, $2
        beq $5, $6, Target
        lw $4, 0($2)
        .
        .
        .
Target: lw $1, 0($7)
        ...
```

```
Fragment 1: (with Delay Slot)
        add $5, $5, $2
        beqd $5, $6, Target
        nop
         lw $4, 0($2)
        .
        .
        .
Target: lw $1, 0($7)
        ...
```

Here, we cannot place the first **lw** into the delay slot because **$4** is not overwritten on the taken path. Likewise, the second **lw** cannot be placed in the delay slot because it is not known if **$1** is overwritten on the not-taken path. Because the branch condition depends on $5, the **$5**, the **add** cannot be placed in the slot either. The correct answer is to insert a **NOP** in the delay slot

For the taken case, there is no performance difference from the original code (there is a 3-cycle delay in both). When the branch is not-taken, the performance is worse because not-taken prediction would have started with the first **lw** a cycle earlier. Thus, the average cycles lost is: 60% * (0 cycles lost) + 40% * (1 cycle lost) = .4 cycles lost on average

➔ Fragment 2

For this problem, one common answer was to place the **lw** instruction into the delay slot. This does not work if the branch is taken. On a branch taken event, the value of **$4** will be overwritten immediately by the **sub** instruction, allowing normal execution. However, this does not address exceptions. By putting the **lw** into the delay slot, we could get an unexpected exception in this case

Consider the following code:
If (ptr != NULL)
    a = *ptr;

This would generate the code that looks something like

lw $4, PTR
beq $4, $0, NULL
lw $1, PTR
...
NULL: ...

If you used delayed branches and moved the second **lw** into the delay slot, you will dereference *ptr even when it is NULL. The correct answer involves duplicating the **SUB** instruction. Since we know that the taken branch happens more often, we can optimize for this case. First, we put a copy of the **SUB** instruction in the delay slot, and then jump to a new branch target called **NewTarg**, which sits one instruction after Target. This way we would cover all the cases. In the taken case, we have executed the **SUB** correctly, and in the not-taken case, the extra **SUB** is overwritten immediately by the **lw**. Here the average cycles gained is: 60% * (1 cycle gained) + 40% * (1 cycle lost) = .2 cycles gained on average

```
Fragment 2:

        add $5, $5, $2
        beq $5, $6, Target
        lw $4, 0($7)
        .
        .
        .
Target: sub $4, $8, $3
        ...
```

```
Fragment 2:  (w/ Delay Slot)

        add $5, $5, $2
        beqd $5, $6, NewTarg
        sub $4, $8, $3
         lw $4, 0($7)
        .
        .
        .
Target:  sub $4, $8, $3
NewTarg: ...
```

➔ Fragment 3

```
Fragment 3:

        movei $2, 21
        .
        .
        .
        addi $4, $4, 1
        beq $4, $2, Target
        .
        .
        .
Target: ...
```

```
Fragment 3: (with Delay Slot)

        movei $2, 21
        .
        .
        .
        addi $4, $4, 1
        beqd $4, $2, Target
         nop
        .
        .
        .
Target: ...
```

In the above fragment, the only instruction that can be placed in the delay slot is a **NOP**. One common answer was to decrement the immediate value in the **movei** instruction to 20 and then place the **addi** in the delay slot. However, because we do not know what happens to **$2** between the **movei** and **addi**, this is not correct (For example, imagine that the instruction right after **movei** was another **movei** that loaded **$2** again).

Because we insert a **NOP** into the delay slot, the effective penalty of the taken branch is 3 cycles. Thus, the average cycles lost is: 60% * (0 cycle lost) + 40% * (1 cycle lost) = .4 cycles lost on average

Problem 8

Answer varies depending on the instruction that was assigned to a student