CS151B Spring 2017 Homework #5 Solutions

Problem (1)

The earliest that both the zero and branch signals are available is in the EX stage. Therefore, it is possible to generate the PCSrc signal in the EX stage.
A key reason for generating the PCSrc signal in the EX stage is that, for a conditional branch instruction, this would allow the correct next value of the PC to be set one cycle earlier. This would reduce by one the number of stall cycles due to the control hazard caused by the conditional branch instruction. (Note that this would also require connecting the output of the ALU to the "1" input of the PCSrc MUX).
If we were not interested in reducing the number of stall cycles due conditional branches, a minor benefit from generating the PCSrc signal in the EX stage is that it would allow a reduction of the size of the EX/MEM register by one bit -- store the PCSrc bit instead of the Zero bit and the Branch bit.
A reason against generating the PCSrc signal in the EX stage is that this could require the clock cycle time to be increased. Specifically, the latency of the EX stage would be increased by the sum of the following: AND gate + PCSrc MUX latency + PC setup time. This could mean that the current clock cycle time would not be long enough for the EX stage to complete its operation reliably. In such a case, the clock cycle time would have to be increased.

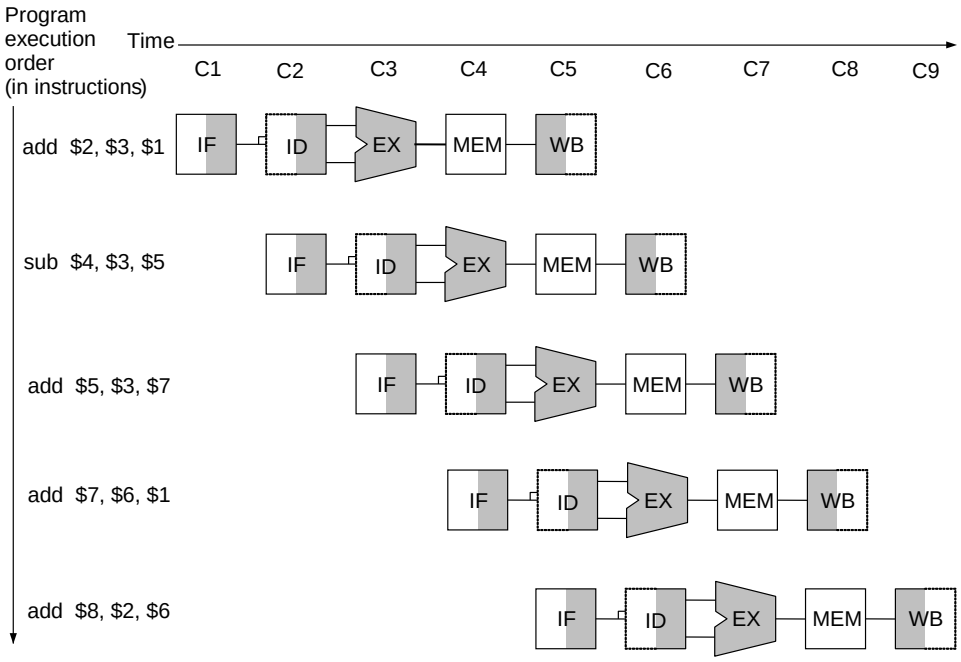
Problem (2)

The PCSrc signal is the AND of the zero output from the ALU and the Branch control signal in the MEM stage.

a) For instruction `add $5,$15,$24`, the value of the Branch control signal is 0. Hence, when the instruction is in its MEM stage, the value of the PCSrc signal is 0.

b) For instruction `beq $1,$1,200`, the value of the Branch control signal and the zero output from the ALU are both 1. Therefore, when the instruction is in its MEM stage, the value of the PCSrc signal is 1.

Problem (3)



During the fifth cycle, registers $6 and $1 of the second to last instruction are being read and register $2 of the first instruction will be written.

Problem (4)

The changes to Figure 4.51 are as follows:

A) Add a bit to the MEM/WB pipeline register. This will be a copy of the EX/MEM.Branch bit from the previous cycle.

B) Add a bit to the MEM/WB pipeline register. This will be a copy of the EX/MEM.Zero bit from the previous cycle.

C) Add a "Hazard detection unit", as in Figure 4.60. In this case, the inputs to the "Hazard detection unit" are:

ID/EX.Branch
EX/MEM.Branch
MEM/WB.Branch
EX/MEM.Zero
MEM/WB.Zero

D) As in Figure 4.60, add a MUX that allows forcing all the control signals to be 0. Call this the ControlNop MUX.

E) As in Figure 4.60, make the IF/ID pipeline register a selectively modified register.

F) As in Figure 4.60, make the PC register a selectively modified register.

We first show a solution which is correct but not optimized for best performance. In this solution, there is the same number of stalls regardless of whether the branch is taken or not taken. We then show an optimized solution that reduces the number of stalls if the branch is not taken.

When the branch is taken, the branch target address is written to the PC when the branch is in the MEM stage and the branch target instruction is fetched when the branch is in the WB stage. Hence, the modifications stall the pipeline for three cycles. The stalls are of the IF and ID stages and begin when the branch is in the EX stage. This is accomplished by the hazard detection unit.

There are three aspects to this:
I) Inserting three NOP instructions following the branch
II) Preventing modifications to the PC -- the contents of the PC are needed if the branch is not taken
III) Maintaining the contents of the IF/ID register – it holds the instruction immediately following the branch which is needed if the branch is not taken.

The implementation will be based on the following:
A) When the ID/EX.Branch bit, EX/MEM.Branch bit, or MEM/WB.Branch bit, is asserted, that means that, respectively, the instruction in the EX, MEM, or WB stage is a branch.
B) When the EX/MEM.Branch **and** the EX/MEM.Zero bits are asserted or the MEM/WB.Branch **and** the MEM/WB.Zero bits are asserted, that means that, respectively, the instruction in the MEM or WB stage is a **taken** branch.

Inserting three NOP instructions following the branch.

A NOP instruction is written to the ID/EX register by asserting the ControlNop signal. Hence, this signal must be asserted for three cycles, starting from when the branch is in the EX stage. This is implemented using the following logic:

```
ControlNop = ID/EX.Branch || EX/MEM.Branch || MEM/WB.Branch
```

I) Preventing modifications to the PC.

Two cases should be considered: when the branch is not taken and when the branch is taken. When the branch is not taken, the PC value must be preserved until the value can be used following the three stalls. Hence:

```
PCWrite = ! (    ID/EX.Branch
              || (EX/MEM.Branch && !EX/MEM.Zero)
              || (MEM/WB.Branch && !MEM/WB.Zero) )
```

When the branch is taken, the branch target address is written to the PC when the branch is in the MEM stage. Furthermore, when the branch is in the WB stage, the PC is written with the address of the instruction following the branch target. Hence, PCWrite must be asserted when the branch is in the

MEM and WB stages. Since the PC is overwritten when the branch is in the MEM stage, there is no need to preserve the PC value in the preceding cycle. However, it is certainly *correct* to preserve the PC value during that cycle. Hence, the expression above for PCWrite is correct for both taken and untaken branches.

II) Maintaining the contents of the IF/ID register.

Two cases should be considered: when the branch is not taken and when the branch is taken. When the branch is not taken, the IF/ID register contains the instruction following the branch. Thus, it must be preserved until the value can be used following the three stalls. Hence:

```
IF/IDWrite = ! (    ID/EX.Branch
                 || (EX/MEM.Branch && !EX/MEM.Zero)
                 || (MEM/WB.Branch && !MEM/WB.Zero) )
```

When the branch is taken, the branch target instruction is fetched and written to the IF/ID register when the branch is in the WB stage -- IF/IDWrite must be asserted then. Hence, there is no need to preserve the value of the IF/ID register in the preceding two cycles. However, it is certainly *correct* to preserve the IF/ID value during these two cycles. Hence, an expression that is correct for both taken and untaken branches is:

```
IF/IDWrite = ! (    ID/EX.Branch
                 || EX/MEM.Branch
                 || (MEM/WB.Branch && !MEM/WB.Zero) )
```

A Performance Optimization for Untaken Branches

When the branch is not taken, it is not necessary to stall for three cycles. Specifically, when the branch is in the MEM stage, it is known that the branch is not taken and the instruction following the branch is already in the IF/ID register. Hence, the instruction following the branch can proceed with its ID stage when the branch is in the MEM stage. Thus, only *one* stall is needed when the branch is not taken. As explained earlier, three stalls are needed when the branch is taken.

I) Inserting NOP instructions following the branch.

Insert three NOP instructions following a taken branch but only one NOP instruction following an untaken branch.

```
ControlNop =   ID/EX.Branch
             || EX/MEM.Branch && EX/MEM.Zero
             || MEM/WB.Branch && MEM/WB.Zero
```

II) Preventing modifications to the PC.

Two cases should be considered: when the branch is not taken and when the branch is taken. When the branch is not taken, the PC value must be preserved until the value can be used following the single stall. Hence:

PCWrite = !ID/EX.Branch

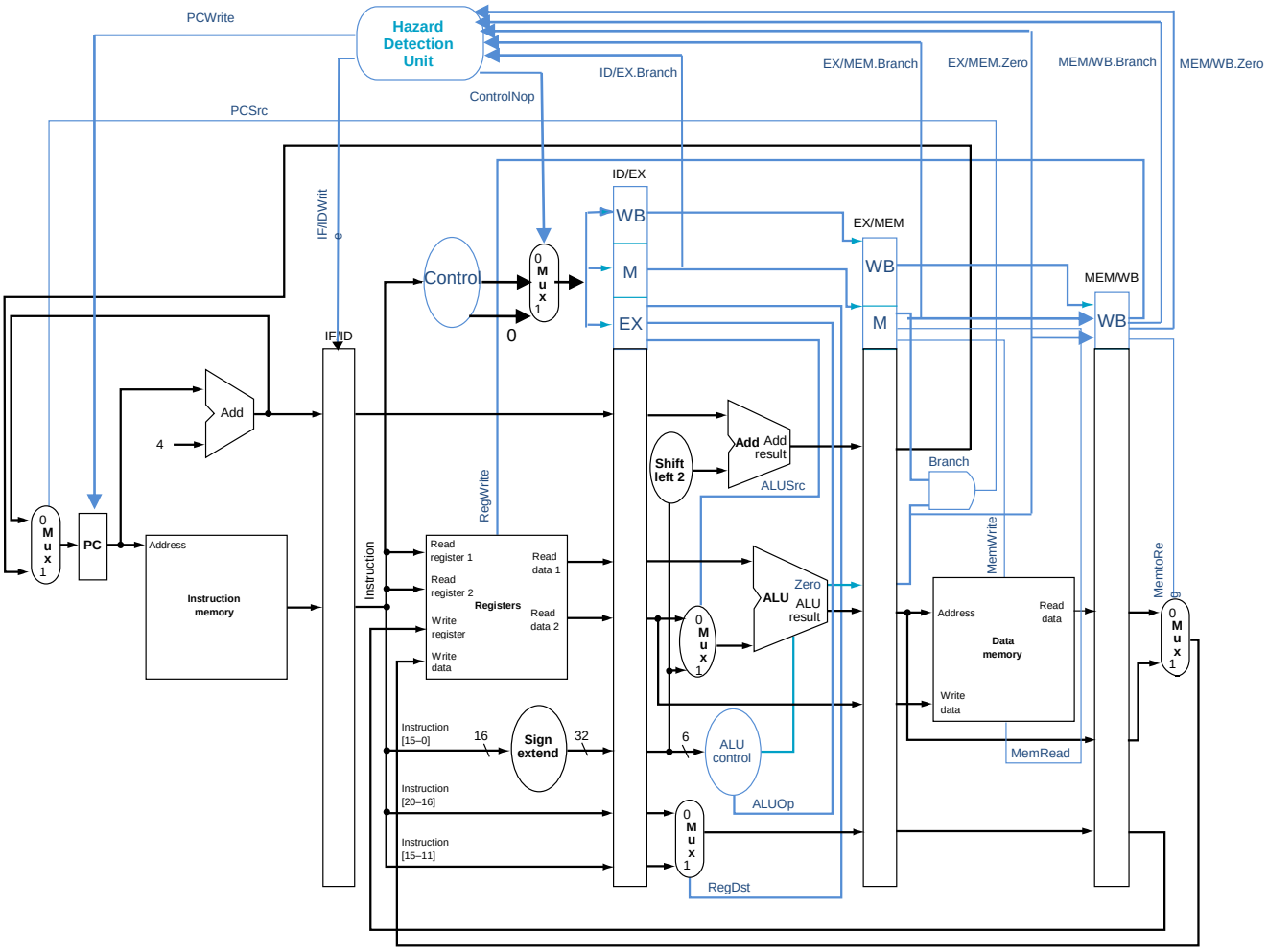
When the branch is taken, the branch target address is written to the PC when the branch is in the MEM stage. Hence, there is no need to preserve the PC value in the preceding cycle. However, it is certainly *correct* to preserve the PC value during that cycle. Thus, the expression above for PCWrite is correct for both taken and untaken branches.

III) Maintaining the contents of the IF/ID register.

Two cases should be considered: when the branch is not taken and when the branch is taken. When the branch is not taken, the IF/ID register contains the instruction following the branch. Thus, it must be preserved until the value can be used following the single stall. Hence:

IF/IDWrite = !ID/EX.Branch

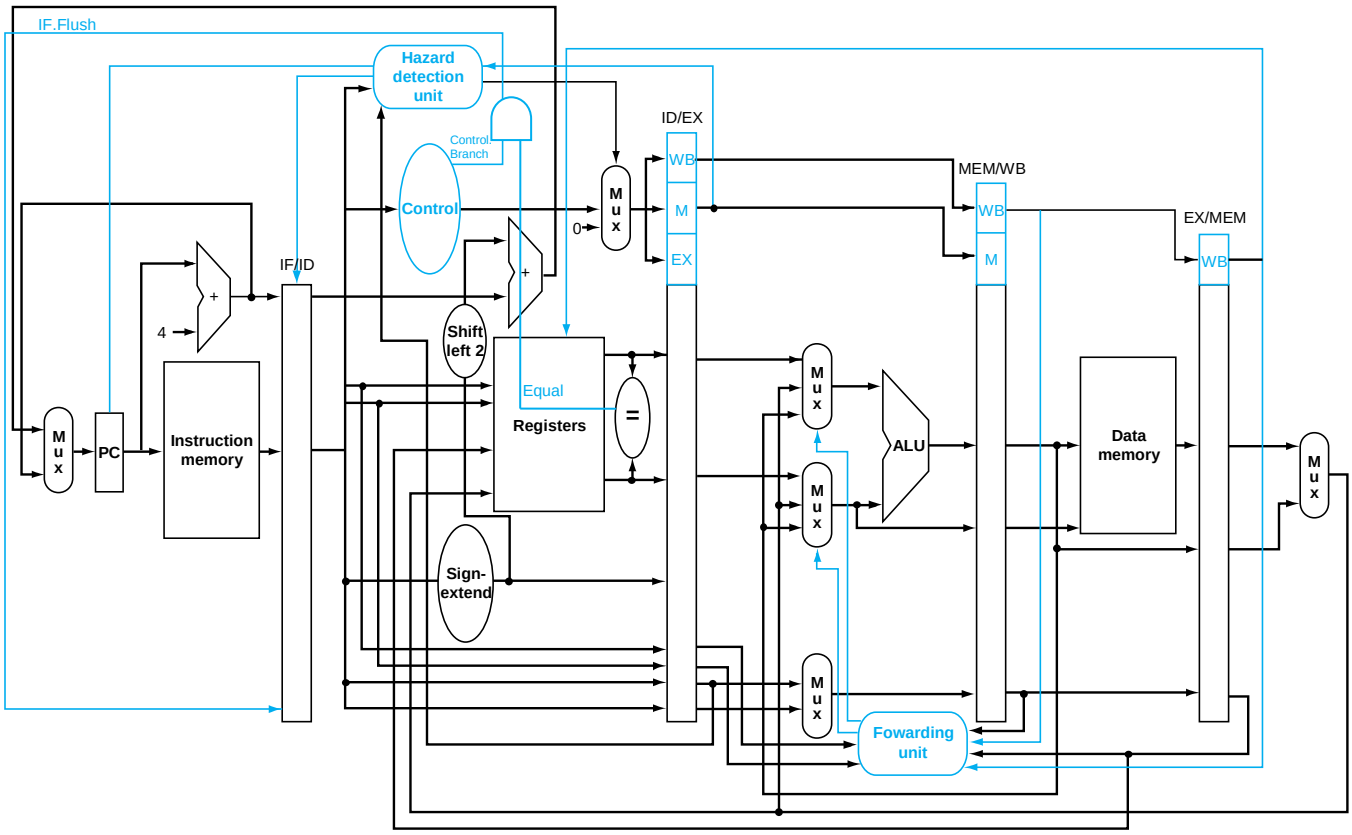
When the branch is taken, the branch target instruction is fetched and written to the IF/ID register when the branch is in the WB stage -- IF/IDWrite must be asserted then. Hence, there is no need to preserve the value of the IF/ID register in the preceding two cycles. However, it is certainly *correct* to preserve the IF/ID value during the first of these two cycles. Hence, the expression above for IF/IDWrite is correct for both taken and untaken branches.



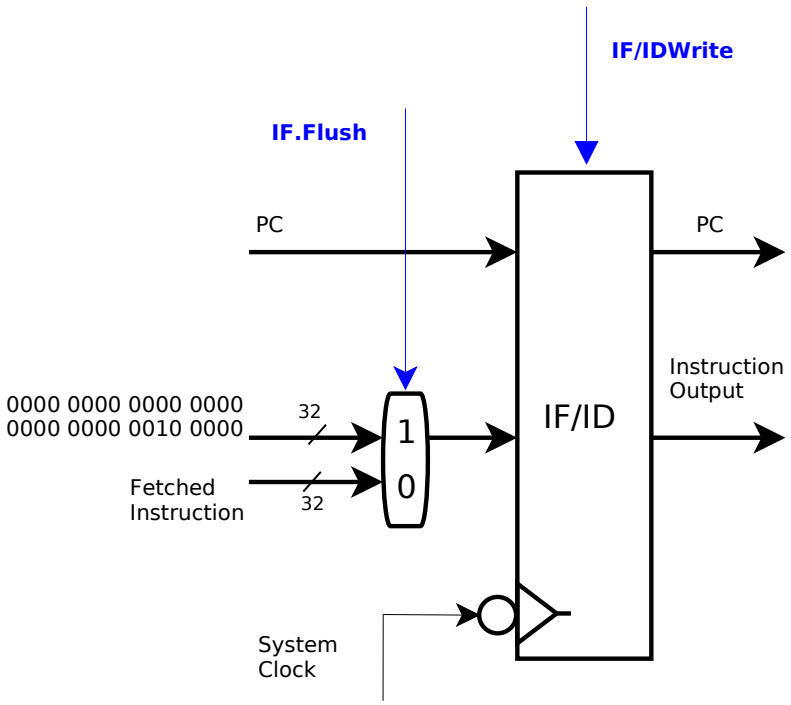
Problem (5)

A) The condition for asserting the IF.Flush signal is "Control.BRANCH && Equal"

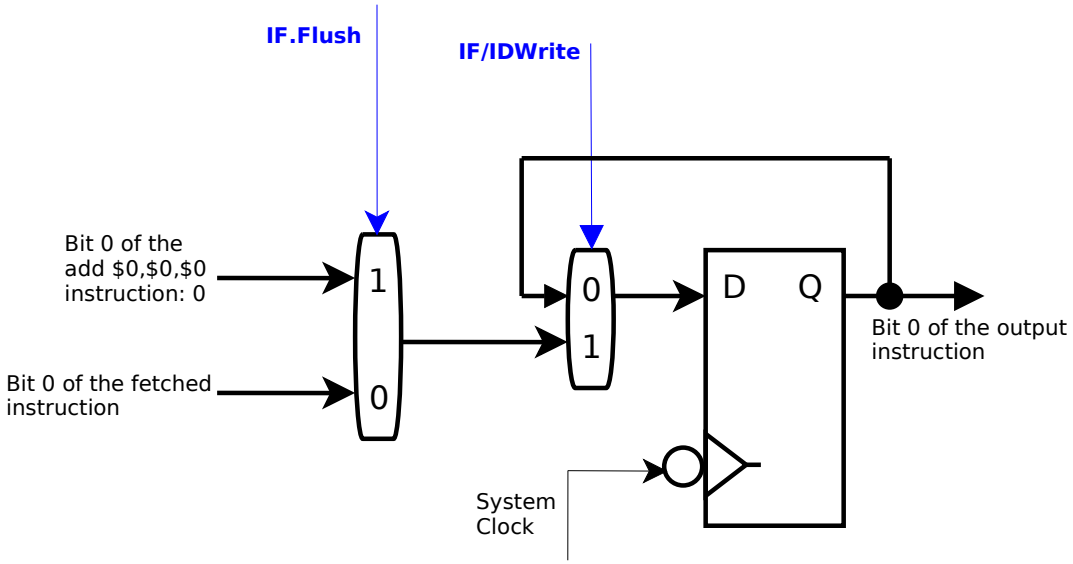
We need to add an AND gate that generates the IF.Flush signal. Its inputs are the Branch signal from the Control unit (which is Control.BRANCH) and the output of the comparator in the ID stage of the pipeline (which is Equal).



B) When IF.Flush is asserted, the instruction currently being fetched must be replaced with a `nop`. The explanation on page 319 of the book indicates that zeroing the instruction field achieves that. With a full implementation of the MIPS ISA, this is true since the machine instruction consisting of 32 0's is `sll $0,$0,0` (see page A-55). However, since the implementation we have doesn't implement the `sll` instruction, we use `add $0,$0,$0` instead. This has the binary representation
0000 0000 0000 0000 0000 0000 0010 0000.



Problem 5, Figure I: The modified IF/ID register.



Problem 5, Figure II: A one-bit slice of the IF/ID register -- bit 0.

Problem (6)

With the original ISA, the code executes in 9 cycles.

		CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
i1	sw	IF	ID	EX	MEM	WB				
i2	lw		IF	ID	EX	MEM	WB			
i3	beq			IF	ID	EX	MEM	WB		
i4	add				IF	ID	EX	MEM	WB	
i5	slt					IF	ID	EX	MEM	WB

With the new ISA, the address being accessed must be computed and stored in a register before it is used in a `lw` or `sw`. Thus, with the new ISA another register is needed for the address. We'll assume that register `r3` is available.

```
addi r3,  r6,  12
sw   r16, (r3)
addi r3,  r6,  8
lw   r16, (r3)
beq  r5,  r4,  Label
add  r5,  r1,  r4
slt  r5,  r1,  r4
```

Assuming that forwarding from ALU outputs is implemented, this code executes in 10 cycles. In the following table, E/M denotes the combined EX/MEM stage.

		CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10
i1	addi	IF	ID	E/M	WB						
i2	addi		IF	ID	E/M	WB					
i3	sw			IF	ID	E/M	WB				
i4	lw				IF	ID	E/M	WB			
i5	beq					IF	ID	E/M	WB		
i6	add						IF	ID	E/M	WB	
i7	slt							IF	ID	E/M	WB

$$Speedup = \frac{performance_{new}}{performance_{old}} = \frac{T_{exec-old}}{T_{exec-new}} = \frac{\#cycles_{old} \times T_{cycle}}{\#cycles_{new} \times T_{cycle}} = \frac{\#cycles_{old}}{\#cycles_{new}} = \frac{9}{10} = 0.9$$

Thus, performance is actually **decreased**.

Problem (7)

We assume that the change specified in the problem is from Figure 4.51 to 4.65.

The execution time, in cycles is
(#cycles without stalls) + (#stall cycles)

#cycles without stalls = #instructions + #stages - 1 = 5 + 5 - 1 = 9

#cycles sub old = 9 + 3 = 12
#cycles sub new = 9 + 1 = 10

$$Speedup = \frac{performance_{new}}{performance_{old}} = \frac{T_{exec-old}}{T_{exec-new}} = \frac{\#cycles_{old} \times T_{cycle}}{\#cycles_{new} \times T_{cycle}} = \frac{\#cycles_{old}}{\#cycles_{new}} = \frac{12}{10} = 1.2$$

Problem (8)

95 opcodes require 7 bits to encode. 64 registers require 6 bits. Instructions are 32 bits wide, and since there are two registers for the instruction class in question, this leaves 13 bits for the immediate field (32 – (7+6+6) = 13). Therefore, the range of a 13 bit 2's compliment number is -2^{12} to $+2^{12}-1$ or in decimal -4096 to 4095.

Problem (9)

Assume that `a[]` and `b[]` are arrays of 4-byte int's.

MIPS:

```
      add  $t0, $zero, $zero     # i <- 0
      addi $t1, $zero, 10   # set max iterations of loop
loop: sll  $t2, $t0, 2      # $t2 <- i * 4
      add  $t2, $t2, $a1    # $t2 <- address of b[i]
      lw   $t4, 0($t2)      # $t4 <- b[i]
      add  $t4, $t4, $t0    # $t4 <- b[i] + i
      sll  $t2, $t0, 3      # $t2 <- i * 4 * 2
      add  $t2, $t2, $a0    # $t2 <- address of a[2i]
      sw   $t4, 0($t2)      # a[2i] <- b[i] + i
      addi $t0, $t0, 1      # i++
      bne  $t0, $t1, loop   # loop if i != 10
```

PowerPC:

```
      add  $t0, $zero, $zero     # i <- 0
      addi $t1, $zero, 10   # set max iterations of loop
      addi $a1, $a1, -4     # adjust the starting base address for lwu
loop: lwu  $t4, 4($a1)      # $t4 <- b[i]; $a1 <- $a1 + 4
      add  $t4, $t4, $t0    # $t4 <- b[i] + i
      sll  $t3, $t0, 3      # $t3 <- i * 4 * 2
      sw   $t4, $a0+$t3     # a[2i] <- b[i] + i
      addi $t0, $t0, 1      # i++
      bne  $t0, $t1, loop   # loop if i != 10
```

Solutions to Homework #5 Practice Problems

Problem (10)

A)

Yes, the specified hazard detection and stall logic is sufficient to ensure correct operation of the specified subset of the MIPS instruction set. Except for *lw*, no stalls are necessary between any other instructions since the forwarding logic can be used to forward operands to dependent instructions. The only stalls necessary are for the load-use data hazards which the hazard detection hardware already provides.

B)

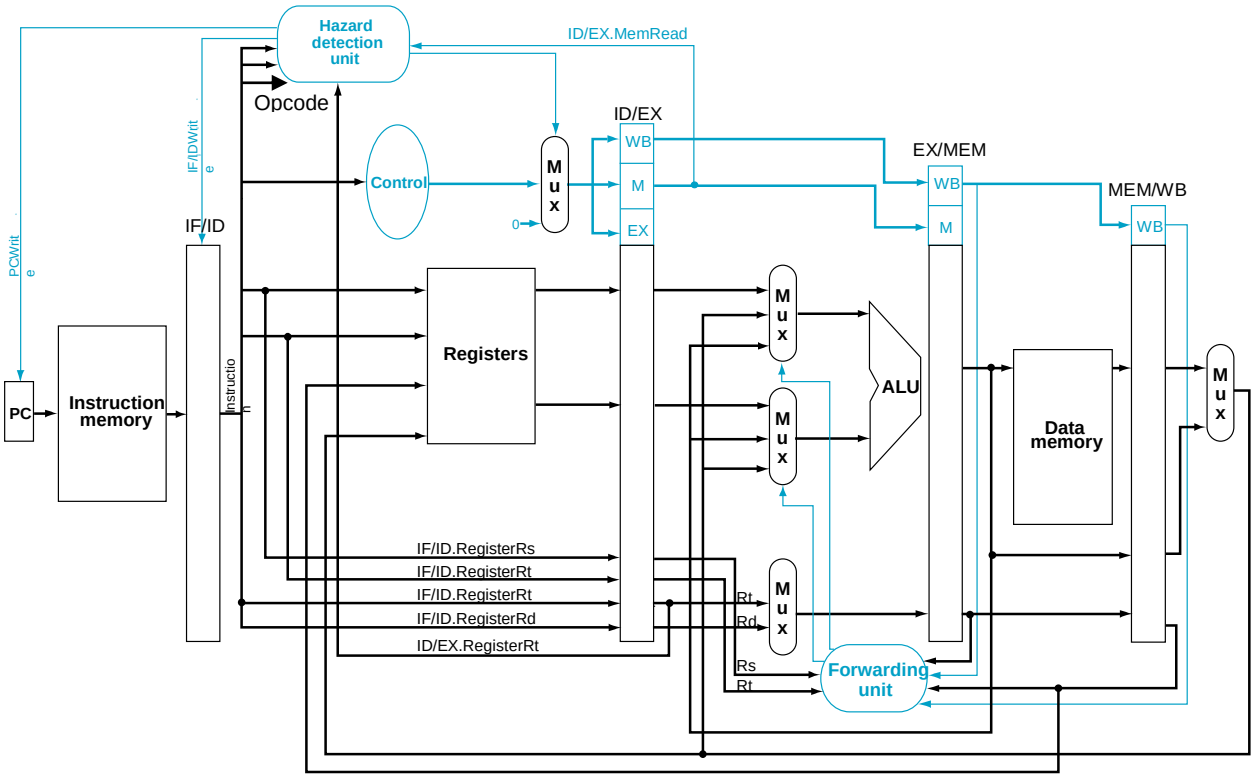
(I)
The claim is correct. The hazard detection logic specified on page 314 does not consider false dependencies that can occur between a *lw* instruction and an I-type instruction. This false dependency can occur if the destination register of the *lw* instruction is the same as the destination register of the I-type instruction. The implemented hazard detection logic will force an unnecessary pipeline stall in these cases.

(II)
The logic on page 314 needs to be modified as follows:

IsIType(OpCode) = { True: if OpCode is I-type
False: if OpCode is not I-type

```
if  ( ID/EX.MemRead and
        (  ( ID/EX.RegisterRt = IF/ID.RegisterRs ) or
           (  ( ID/EX.RegisterRt = IF/ID.RegisterRt ) and
              ! IsType( IF/ID.OpCode )
            )
        )
   )
   stall the pipeline
```

Change Figure 4.60 to have Opcode go into the hazard detection unit.



(III)
The changes in part (II) will not likely have a significant impact on performance since the added delay to the hazard detection logic will not likely be greater than an ALU or a memory operation. It should also be noted that a compiler will not likely generate code in which the destination register of a *lw* instruction will be immediately overwritten by the following I-type instruction. Therefore, the stated performance problem will not likely occur in the first place to have a significant performance impact.

Problem (11)

Since the `sw` uses the memory in cycle 4 and the `lw` uses the memory in cycle 5, no instruction can be fetched during either one of these cycles. Hence, the fetch for the `add` must be delayed by two cycles.

		CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11
i1	sw	IF	ID	EX	MEM	WB						
i2	lw		IF	ID	EX	MEM	WB					
i3	beq			IF	ID	EX	MEM	WB				
i4	add				stall	stall	IF	ID	EX	MEM	WB	
i5	slt							IF	ID	EX	MEM	WB

Hence, the total execution time is 11 cycles.
The cycle time is determined by the stage with the longest latency. In this case, it is set to 200ps due to the IF stage.
Thus, the total execution time is 200 * 11 = 2200ps.

Unlike data hazards, this structural hazard cannot be handled by `nop` instructions. The problem is that `nop` instructions have to be fetched, just like every other instruction. Since the memory is busy, the `nop` instructions cannot be fetched. The **hardware** must detect the resource conflict and stall the instruction fetches.

Problem (12)

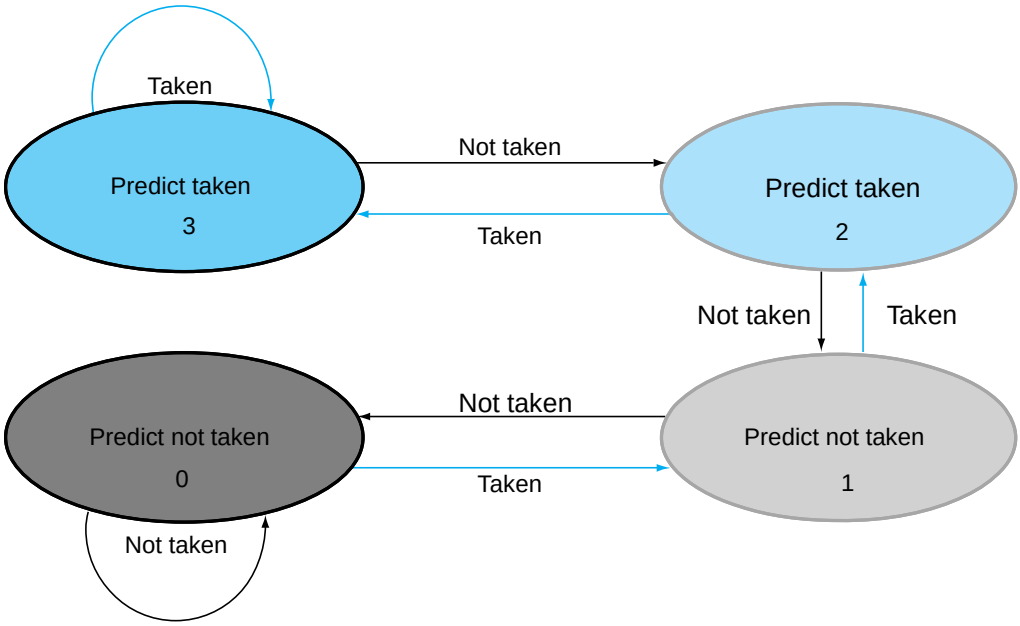
Bold letters indicate accurate predictions.

Always taken: **T**, NT, **T**, **T**, NT
Prediction accuracy = 3/5 = 60%

Always not taken: T, **NT**, T, T, **NT**
Prediction accuracy = 2/5 = 40%

Problem (13)

For clarity, we attach state labels to the states in Figure 4.63:



Bold letters indicate accurate predictions.

Branch outcome	T	NT	T	T
Stated used for prediction	0	1	0	1
Prediction	NT	**NT**	NT	NT

Prediction accuracy = 1/4 = 25%

Problem (14)

0 Address	
Instructions	**Size in bits**
PUSH B	24
PUSH C	24
MUL	8
PUSH A	24
ADD	8
PUSH D	24
PUSH E	24
PUSH F	24
MUL	8
SUB	8
DIV	8
POP X	24
Total:	**200**

1 Address	
Instructions	**Size in bits**
LOAD E	24
MUL F	24
STORE T1	24
LOAD D	24
SUB T1	24
STORE T1	24
LOAD B	24
MUL C	24
ADD A	24
DIV T1	24
STORE X	24
Total:	**264**

T1 is a temporary memory location.

2 Address	
Instructions	**Size in bits**
MOV R1,B	28
MUL R1,C	28
ADD R1,A	28
MOV R2,E	28
MUL R2,F	28
MOV R3,D	28
SUB R3,R2	16
DIV R1,R3	16
MOV X,R1	28
Total:	**228**

R1, R2 and R3 are temporary registers.

3 Address	
Instructions	**Size in bits**
MUL R1,B,C	44
ADD R1,R1,A	32
MUL R2,E,F	44
SUB R2,D,R2	32
DIV X,R1,R2	32
Total:	**184**

R1 and R2 are temporary registers.