Solutions for Chapter 6 Exercises

6.1

a. Shortening the ALU operation will not affect the speedup obtained from pipelining. It would not affect the clock cycle.

b. If the ALU operation takes 25% more time, it becomes the bottleneck in the pipeline. The clock cycle needs to be 250 ps. The speedup would be 20% less.

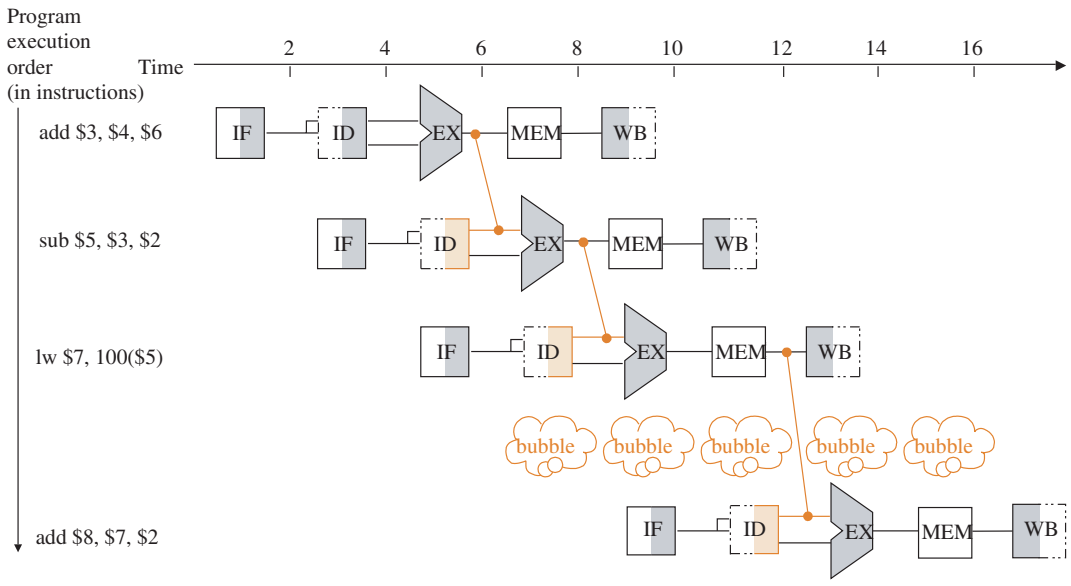
6.2

a. It takes 100 ps * 10^6 instructions = 100 microseconds to execute on a non-pipelined processor (ignoring start and end transients in the pipeline).

b. A perfect 20-stage pipeline would speed up the execution by 20 times.

c. Pipeline overhead impacts both latency and throughput.

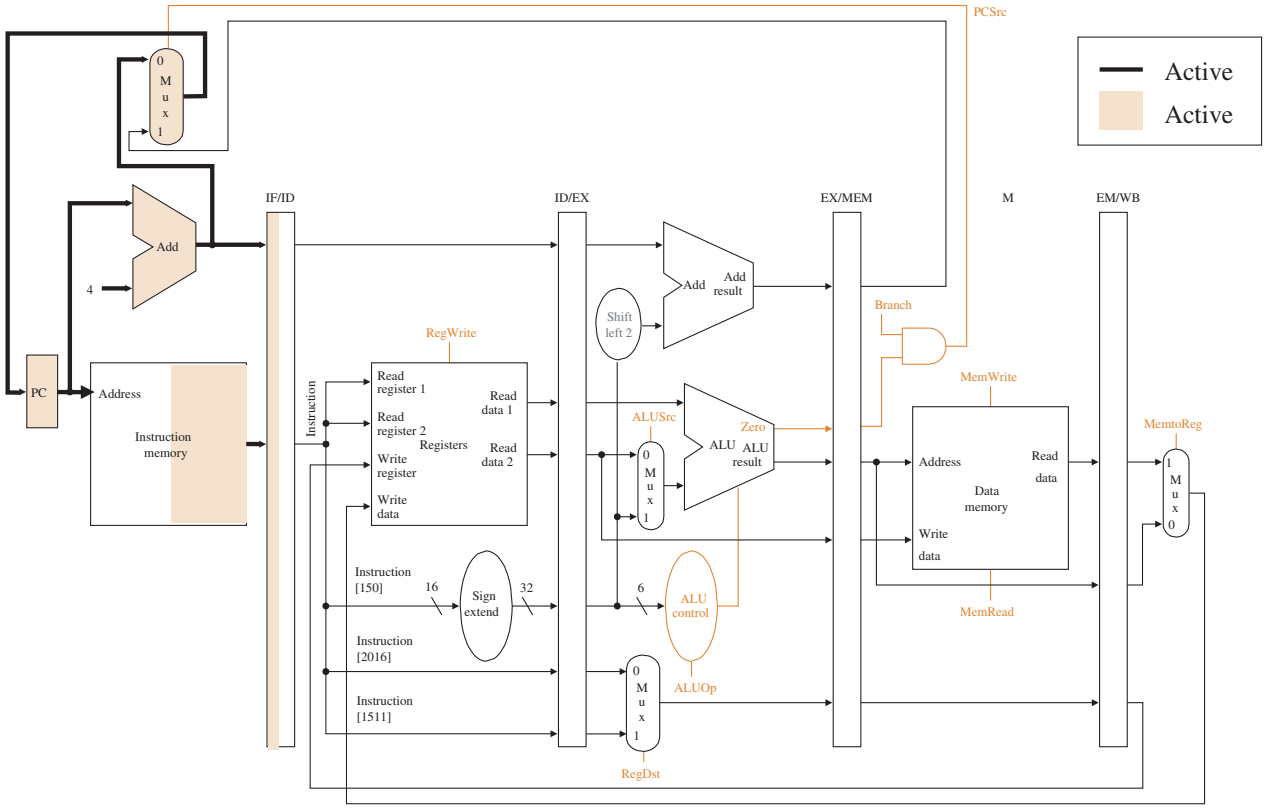
6.3 See the following figure:



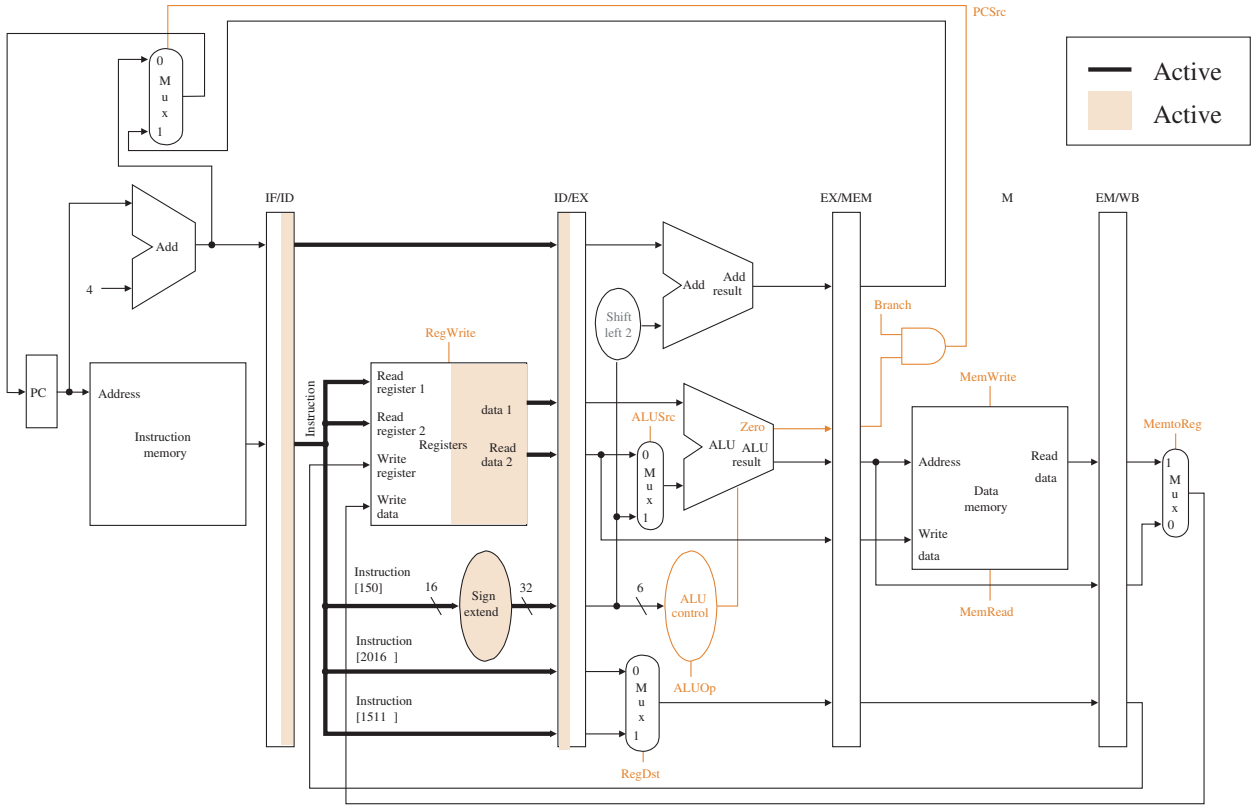
6.4 There is a data dependency through `$3` between the first instruction and each subsequent instruction. There is a data dependency through `$6` between the `lw` instruction and the last instruction. For a five-stage pipeline as shown in Figure 6.7, the data dependencies between the first instruction and each subsequent instruction can be resolved by using forwarding.

The data dependency between the load and the last add instruction cannot be resolved by using forwarding.

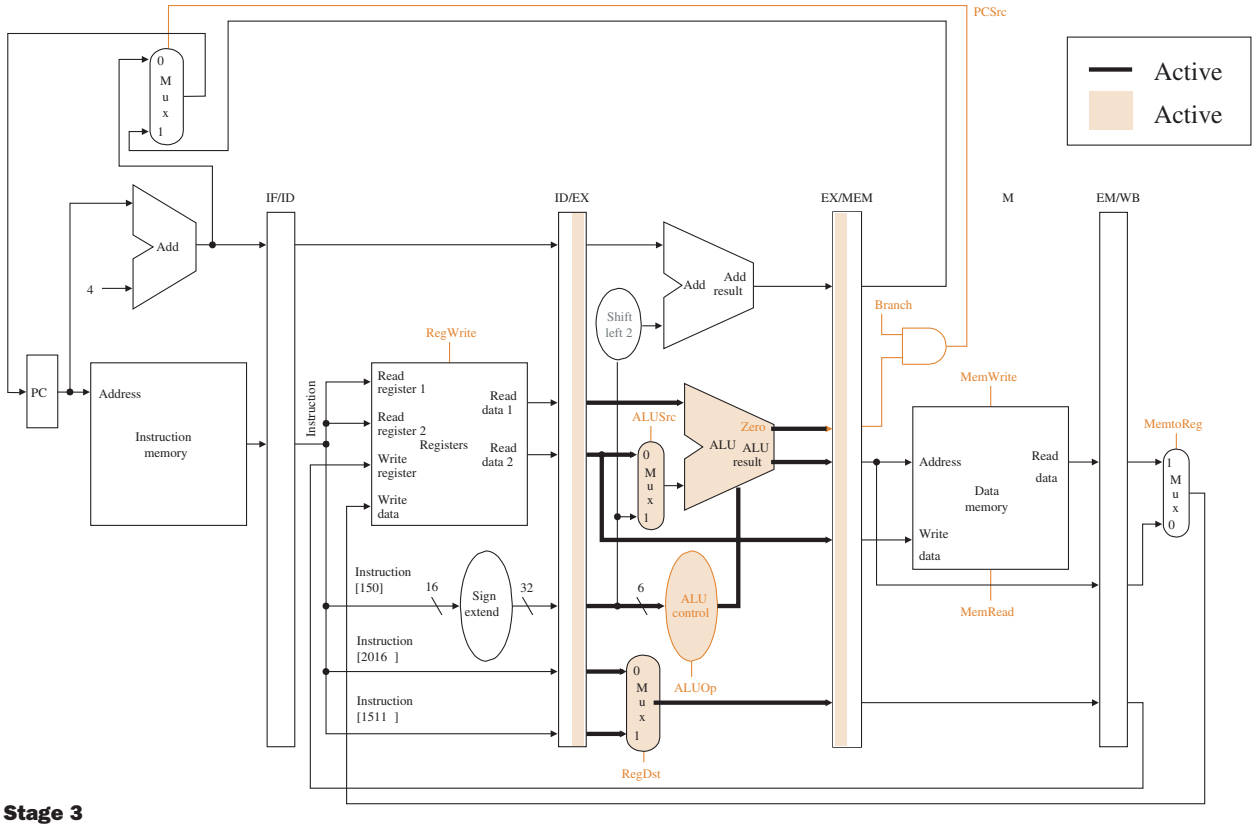
6.6 Any part of the following figure not marked as active is inactive.



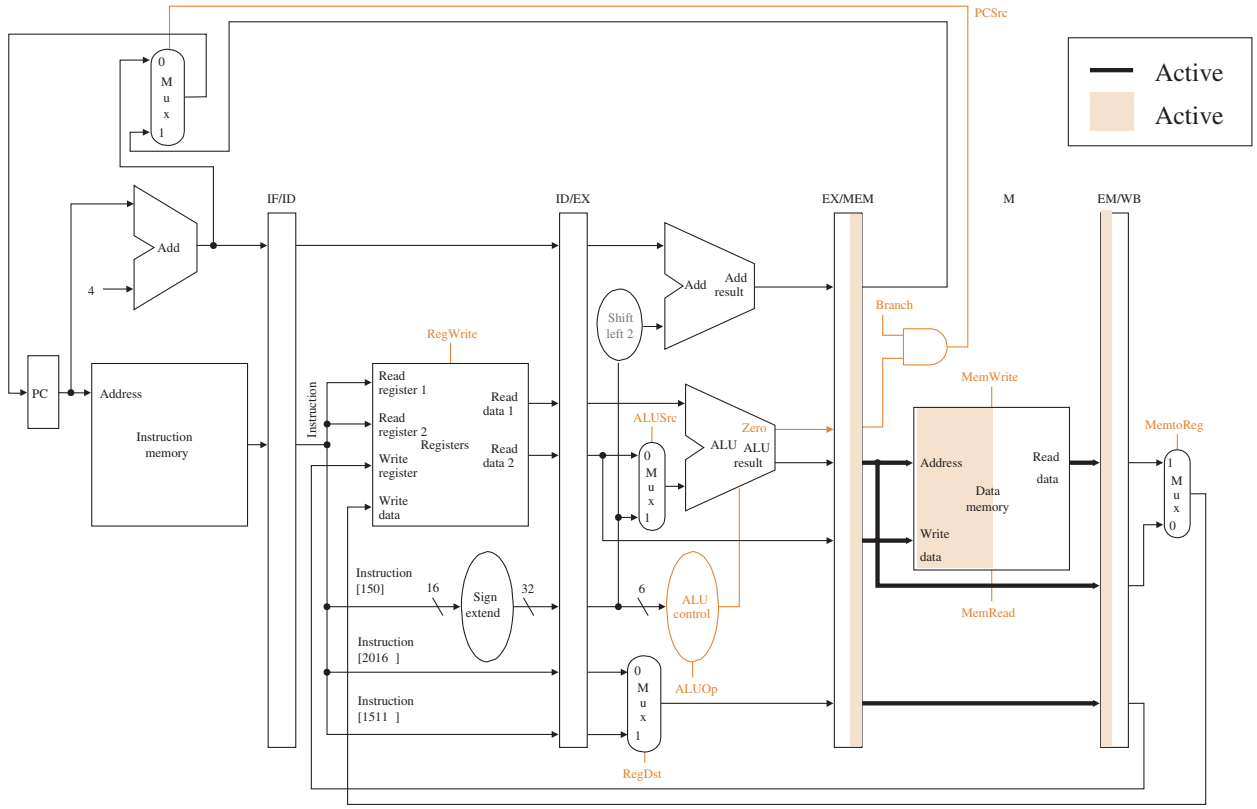
Stage 1


PCSrc
Active
Active
IF/ID
ID/EX
EX/MEM
M
EM/WB
0
M
u
x
1
Add
4
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
data 1
Read data 2
Shift left 2
Add
Add result
ALUSrc
Zero
ALU
ALU result
Branch
MemWrite
Address
Data memory
Read data
Write data
MemRead
MemtoReg
Instruction [150]
16
Sign extend
32
6
ALU control
ALUOp
Instruction [2016]
Instruction [1511]
RegDst

Stage 2


PCSrc
Active
Active
IF/ID
ID/EX
EX/MEM
M
EM/WB
0
M
u
x
1
Add
4
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
Branch
MemWrite
ALUSrc
Zero
ALU
ALU result
Address
Data memory
Read data
Write data
MemtoReg
MemRead
Instruction [150]
16
Sign extend
32
6
ALU control
ALUOp
Instruction [2016]
Instruction [1511]
RegDst

Stage 3



Stage 4

Since this is an sw instruction, there is no work done in the WB stage.

6.12 No solution provided.

6.13 No solution provided.

6.14 No solution provided.

6.17 At the end of the first cycle, instruction 1 is fetched.

At the end of the second cycle, instruction 1 reads registers.

At the end of the third cycle, instruction 2 reads registers.

At the end of the fourth cycle, instruction 3 reads registers.

At the end of the fifth cycle, instruction 4 reads registers, and instruction 1 writes registers.

Therefore, at the end of the fifth cycle of execution, registers `$6` and `$1` are being read and register `$2` will be written.

6.18 The forwarding unit is seeing if it needs to forward. It is looking at the instructions in the fourth and fifth stages and checking to see whether they intend to write to the register file and whether the register written is being used as an ALU input. Thus, it is comparing 3 = 4? 3 = 2? 7 = 4? 7 = 2?

6.19 The hazard detection unit is checking to see whether the instruction in the ALU stage is an `lw` instruction and whether the instruction in the ID stage is reading the register that the `lw` will be writing. If it is, it needs to stall. If there is an `lw` instruction, it checks to see whether the destination is register 6 or 1 (the registers being read).

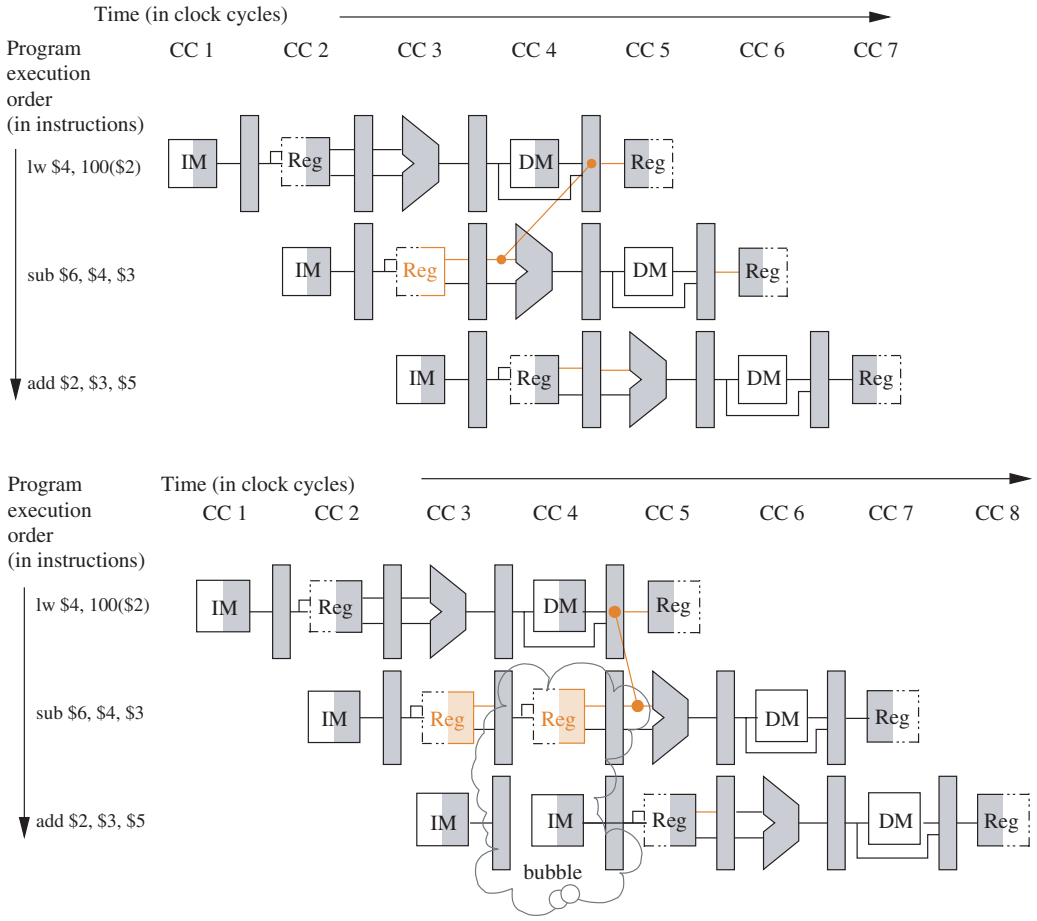
6.21

a. There will be a bubble of 1 cycle between a `lw` and the dependent `add` since the load value is available after the MEM stage.

   There is no bubble between an `add` and the dependent `lw` since the `add` result is available after the EX stage and it can be forwarded to the EX stage for the dependent `lw`. Therefore, CPI = cycle/instruction ≈ 1.5.

b. Without forwarding, the value being written into a register can only be read in the same cycle. As a result, there will be a bubble of 2 cycles between an `lw` and the dependent `add` since the load value is written to the register after the MEM stage. Similarly, there will be a bubble of 2 cycles between an `add` and the dependent `lw`. Therefore, CPI ≈ 3.

6.22 It will take 8 cycles to execute this code, including a bubble of 1 cycle due to the dependency between the `lw` and `sub` instructions.



6.23

Input	Number of bits	Usage
ID/EX.RegisterRs	5	operand reg number, compare to see if match
ID/EX.RegisterRt	5	operand reg number, compare to see if match
EX/MEM.RegisterRd	5	destination reg number, compare to see if match
EX/MEM.RegWrite	1	TRUE if writes to the destination reg
MEM/WB.RegisterRd	5	destination reg number, compare to see if match
MEM/WB.RegWrite	1	TRUE if writes to the destination reg
Output	**Number of bits**	**Usage**
ForwardA	2	forwarding signal
ForwardB	2	forwarding signal

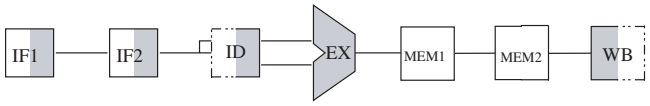
6.29 No solution provided.

6.30 The performance for the single-cycle design will not change since the clock cycle remains the same.

For the multicycle design, the number of cycles for each instruction class becomes the following: loads: 7, stores: 6, ALU instructions: 5, branches: 4, jumps: 4.

CPI = 0.25 * 7 + 0.10 * 6 + 0.52 * 5 + 0.11 * 4 + 0.02 * 4 = 5.47. The cycle time for the multicycle design is now 100 ps. The average instruction becomes 5.47 * 100 = 547 ps. Now the multicycle design performs better than the single-cycle design.

6.33 See the following figure.



when defined by lw	when defined by R-type
used in i1 => 2-cycle stall	used in i1 => forward
used in i2 => 1-cycle stall	used in i2 => forward
used in i3 => forward	used in i3 => forward

6.34 Branches take 1 cycle when predicted correctly, 3 cycles when not (including one more memory access cycle). So the average clock cycle per branch is 0.75 * 1 + 0.25 * 3 = 1.5.

For loads, if the instruction immediately following it is dependent on the load, the load takes 3 cycles. If the next instruction is not dependent on the load but the second following instruction is dependent on the load, the load takes two cycles. If neither two following instructions are dependent on the load, the load takes one cycle.

The probability that the next instruction is dependent on the load is 0.5. The probability that the next instruction is not dependent on the load, but the second following instruction is dependent, is 0.5 * 0.25 = 0.125. The probability that neither of the two following instructions is dependent on the load is 0.375.

Thus the effective CPI for loads is 0.5 * 3 + 0.125 * 2 + 0.375 * 1 = 2.125.

Using the date from the example on page 425, the average CPI is 0.25 * 2.125 + 0.10 * 1 + 0.52 * 1 + 0.11 * 1.5 + 0.02 * 3 = 1.47.

Average instruction time is 1.47 * 100ps = 147 ps. The relative performance of the restructured pipeline to the single-cycle design is 600/147 = 4.08.

6.35 The opportunity for both forwarding and hazards that cannot be resolved by forwarding exists when a branch is dependent on one or more results that are still in the pipeline. Following is an example:

```
lw  $1, $2(100)
add $1, $1, 1
beq $1, $2, 7
```

6.36 Prediction accuracy = 100% * PredictRight/TotalBranches

a. Branch 1: prediction: T-T-T, right: 3, wrong: 0

   Branch 2: prediction: T-T-T-T, right: 0, wrong: 4

   Branch 3: prediction: T-T-T-T-T-T, right: 3, wrong: 3

   Branch 4: prediction: T-T-T-T-T, right: 4, wrong: 1

   Branch 5: prediction: T-T-T-T-T-T-T, right: 5, wrong: 2

   Total: right: 15, wrong: 10

   Accuracy = 100% * 15/25 = 60%

b. Branch 1: prediction: N-N-N, right: 0, wrong: 3

Branch 2: prediction: N-N-N-N, right: 4, wrong: 0

Branch 3: prediction: N-N-N-N-N-N, right: 3, wrong: 3

Branch 4: prediction: N-N-N-N-N, right: 1, wrong: 4

Branch 5: prediction: N-N-N-N-N-N-N, right: 2, wrong: 5

Total: right: 10, wrong: 15

Accuracy = 100% * 10/25 = 40%

c. Branch 1: prediction: T-T-T, right: 3, wrong: 0

Branch 2: prediction: T-N-N-N, right: 3, wrong: 1

Branch 3: prediction: T-T-N-T-N-T, right: 1, wrong: 5

Branch 4: prediction: T-T-T-T-N, right: 3, wrong: 2

Branch 5: prediction: T-T-T-N-T-T-N, right: 3, wrong: 4

Total: right: 13, wrong: 12

Accuracy = 100% * 13/25 = 52%

d. Branch 1: prediction: T-T-T, right: 3, wrong: 0

Branch 2: prediction: T-N-N-N, right: 3, wrong: 1

Branch 3: prediction: T-T-T-T-T-T, right: 3, wrong: 3

Branch 4: prediction: T-T-T-T-T, right: 4, wrong: 1

Branch 5: prediction: T-T-T-T-T-T-T, right: 5, wrong: 2

Total: right: 18, wrong: 7

Accuracy = 100% * 18/25 = 72%

6.37 No solution provided.

6.38 No solution provided.

6.39 Rearrange the instruction sequence such that the instruction reading a value produced by a load instruction is right after the load. In this way, there will be a stall after the load since the load value is not available till after its MEM stage.

```
lw  $2, 100($6)
add $4, $2, $3
lw  $3, 200($7)
add $6, $3, $5
sub $8, $4, $6
lw  $7, 300($8)
beq $7, $8, Loop
```

6.40 Yes. When it is determined that the branch is taken (in WB), the pipeline will be flushed. At the same time, the `lw` instruction will stall the pipeline since the load value is not available for `add`. Both flush and stall will zero the control signals. The flush should take priority since the `lw` stall should not have occurred. They are on the wrong path. One solution is to add the flush pipeline signal to the Hazard Detection Unit. If the pipeline needs to be flushed, no stall will take place.

6.41 The store instruction can read the value from the register if it is produced at least 3 cycles earlier. Therefore, we only need to consider forwarding the results produced by the two instructions right before the store. When the store is in EX stage, the instruction 2 cycles ahead is in WB stage. The instruction can be either a `lw` or an ALU instruction.

```
assign EXMEMrt = EXMEMIR[20:16];

 assign bypassVfromWB = (IDEXop == SW) & (IDEXrt != 0) &
    ( ((MEMWBop == LW)   & (IDEXrt == MEMWBrt)) |
      ((MEMWBop ==ALUop) & (IDEXrt == MEMWBrd)) );
```

This signal controls the store value that goes into EX/MEM register. The value produced by the instruction 1 cycle ahead of the store can be bypassed from the MEM/WB register. Though the value from an ALU instruction is available 1 cycle earlier, we need to wait for the load instruction anyway.

```
assign bypassVfromWB2 = (EXMEMop == SW) & (EXMEMrt != 0) &
     (!bypassVfromWB) &
   ( ((MEMWBop == LW)    & (EXMEMrt == MEMWBrt)) |
     ((MEMWBop == ALUop) & (EXMEMrt == MEMWBrd)) );
```

This signal controls the store value that goes into the data memory and MEM/WB register.

6.42

```
assign bypassAfromMEM = (IDEXrs != 0) &
  ( ((EXMEMop ==LW)     & (IDEXrs == EXMEMrt)) |
    ((EXMEMop == ALUop) & (IDEXrs == EXMEMrd)) );
assign bypassAfromWB = (IDEXrs != 0) & (!bypassAfromMEM) &
   ( ((MEMWBop == LW)    & (IDEXrs == MEMBrt)) |
     ((MEMWBop == ALUop) & (IDEXrs == MEMBrd)) ):
```

6.43 The branch cannot be resolved in ID stage if one branch operand is being calculated in EX stage (assume there is no dumb branch having two identical operands; if so, it is a jump), or to be loaded (in EX and MEM).

```
assign branchStallinID = (IFIDop == BEQ) &
   ( ((IDEXop == ALUop) & ((IFIDrs == IDEXrd) |
      (IFIDrt == IDEXrd)) ) | // alu in EX
     ((IDEXop == LW) & ((IFIDrs == IDEXrt) |
      (IFIDrt == IDEXrt)) ) | // lw in EX
     ((EXMEMop == LW) & ((IFIDrs == EXMEMrt) |
      (IFIDrt == EXMEMrt)) ) ); // lw in MEM
```

Therefore, we can forward the result from an ALU instruction in MEM stage, and an ALU or `lw` in WB stage.

```
assign bypassIDA = (EXMEMop == ALUop) & (IFIDrs == EXMEMrd);
assign bypassIDB = (EXMEMop == ALUop) & (IFIDrt == EXMEMrd);
```

Thus, the operands of the branch become the following:

```
 assign IDAin = bypassIDA ? EXMEMALUout : Regs[IFIDrs];
 assign IDBin = bypassIDB ? EXMEMALUout : Regs[IFIDrt];
```

And the branch outcome becomes:

```
assign takebranch = (IFIDop == BEQ) & (IDAin == IDBin);
```

6.44 For a delayed branch, the instruction following the branch will always be executed. We only need to update the PC after fetching this instruction.

```
if(~stall) begin IFIDIR <= IMemory[PC]; PC <= PC+4; end;
if(takebranch) PC <= PC + {16{IFIDIR[15]} +4; end;
```

6.45

```
module PredictPC(currentPC, nextPC, miss, update, destination);
input currentPC, update, destination;
output nextPC, miss;
integer index, tag;
//512 entries, direct-map
reg[31:0] brTargetBuf[0:511], brTargetBufTag[0:511];
index = (currentPC>>2) & 511;
tag = currentPC>>(2+9);
if(update) begin //update the destination and tag
  brTargetBuf[index]=destination;
  brTargetBufTag[index]=tag; end;
else if(tag==brTargetBufTag[index]) begin //a hit!
  nextPC=brTargetBuf[index]; miss=FALSE; end;

  else miss=TRUE;
endmodule;
```

6.46 No solution provided.

6.47

```
    Loop:    lw   $2, 0($10)
             lw   $5, 4($10)
             sub  $4, $2, $3
             sub  $6, $5, $3
             sw   $4, 0($10)
             sw   $6, 4($10)
             addi $10, $10, 8
             bne  $10, $30, Loop
```

6.48 The code can be unrolled twice and rescheduled. The leftover part of the code can be handled at the end. We will need to test at the beginning to see if it has reached the leftover part (other solutions are possible.

```
Loop:       addi $10, $10, 12
            bgt  $10, $30, Leftover
            lw   $2, -12($10)
            lw   $5, -8($10)
            lw   $7, -4($10)
            sub  $4, $2, $3
            sub  $6, $5, $3
            sub  $8, $7, $3
            sw   $4, -12($10)
            sw   $6, -8($10)
            sw   $8, -4($10)
            bne  $10, $30, Loop
            jump Finish
Leftover:   lw   $2, -12($10)
            sub  $4, $2, $3
            sw   $4, -12($10)
            addi $10, $10, -8
            beq  $10, $30, Finish
            lw   $5, 4($10)
            sub  $6, $5, $3
            sw   $6, 4($10)
Finish: ...
```

6.49

alu or branch				lw/sw	
Loop:	addi	$20,	$10, 0	lw $2, 0($10)	
				lw $5, 4($10)	
	sub	$4,	$2, $3	lw $7, 8($10)	
	sub	$6,	$5, $3	lw $8, 12($10)	
	sub	$11,	$7, $3	sw $4, 0($10)	
	sub	$12,	$8, $3	sw $6, 4($10)	
	addi	$10,	$10, 16	sw $11, 8($20)	
	bne	$10,	$30, Loop	sw $12, 12($20)	

6.50 The pipe stages added for wire delays do not produce any useful work. With imperfect pipelining due to pipeline overhead, the overhead associated with these stages reduces throughput. These extra stages increase the control logic complexity since more pipe stages need to be covered. When considering penalties for branches mispredictions, etc., adding more pipe stages increase penalties and execution latency.

6.51 No solution provided.