4.10.1 In the pipelined execution show below, *** represents a stall when an instruction cannot be fetched because a load or store instruction is using the memory in that cycle. Cycles are represented from left to right, & for each instruction we show the pipeline stage it is in during that cycle:-

Instruction	Pipeline stage											Cycles
Sw r16, 12(r6)	IF	ID	EX	MEM	WB							11
lw r16, 8(r6)		IF	ED	EX	MEM	WB						
beq rs, r4, label #Assume rs! = r4			IF	ID	EX	MEM	WB					
add rs, r1, r4				***	***	IF	ID	EX	MEM	WB		
slt rs, r15, r4							IF	ID	EX	MEM	WB	

We can not add NOPs to code to eliminate this hazard as NOPs need to be fetched just like any other instructions, so this hazard must be addressed with a hardware hazard detection unit in the processor.

4.10.2 This change only saves one cycle in an entire execution without data hazards (such as the one given). This cycle is saved because the last instruction finishes one cycle earlier (one less stage to go through). If there were data hazards from loads to other instruction the change would help eliminate some stall cycles.

Instructions Executed	Cycles with 5 stages	Cycles with 4 Stages	Speed up
5	4+5 = 9	3+5 = 8	9/8 = 1.13

4.11.1

lw r1, 0(r1)	WB							
lw r1, 0(r1)	EX	MEM	WB					
beq r1, r0, loop	ID	***	EX	MEM	WB			
lw r1, 0(r1)	IF	***	ID	EX	MEM	WB		
and r1, r1, r2			IF	ID	***	EX	MEM	WB
lw r1, 0(r1)				IF	***	ID	EX	MEM
lw r1, 0(r1)						IF	ID	***
beq r1, r0, loop							IF	***

4.11.2 In a particular ~~cycle~~, clock cycle, a pipeline stage is not doing useful work if it is stalled or if the instruction going through that stage is not doing any useful work there. In the pipeline execution diagram from above, a stage is stalled if its name is not shown for a particular cycles, and stages in which the particular instruction is not doing useful work are marked in blue. Note that a beq instruction is doing useful work in the MEM stage, because it is determining the correct value of the next instruction's PC in that stage.

Cycles per loop iteration	Cycles in which all stages do useful work	% of cycles in which all stage do useful work
8	0	0 %

4.12.1 Fractions of cycles stalled when no forwarding is used.

For the first instruction there will be two stall cycles due to the dependencies of first and second next instruction. Therefore dependency will be one stall cycle with second next instruction.

$\Rightarrow$ 1 + (Ex to 1st only + Ex to 2nd only + memory to 1st) * 2 + (Ex to 2nd only) * 1 (Memory to 2nd) * 1 + other RAW dependecies * 1

$\Rightarrow$ 1 + (0.35 * 2) + (0.05 * 1) + (0.1 * 1) = (0.1 * 1) = 1.95

Therefore, fraction of cycle stalled is 0.95/1.95 = 48.7 %

4.12.2 Fractions of cycles stalled when full forwarding is used

With full forwarding only Read & write dependencies cause cycles to be stalled. There will be only one cycle stalled with memory stage of one instruction to next instruction.

Memory to 1st + Memory to 2nd + other RAW dependencies

$\Rightarrow$ 1 + 0.2 + 0.1 + 0.1

= 1.4

Therefore, fraction of cycle stalled is 0.4/1.4 = 28.5 %

4.12.4 To Calculate speed up:-

Taking the computed results with no forwarding & full forwarding :-

Speed up without forwarding $= 1.95 * 130ps = 253.5 ps$

Speed up with forwarding $= 1.20 * 150 ps = 180 ps$

Total speed up $= 1 + [(253.5 - 180)/180) = 1.4 ps$

4.13.1

```
add r5, r2, r1
 nop
 nop
 lw r3, 4(r5)
 lw r2, 0(r2)
 nop
 or r3, r5, r3
 nop
 nop
 sw, r3, 0(r5)
```

4.13.2 Instruction can be moved up by swapping its place with another instruction which has no dependencies such that some NOP slots can be filled with such instructions. Int he givene example, we ~~[illegible]~~ may make use of R7 to eliminate dependencies of WAW/WAR so that we can have more instructions to be moved up.

```
i1: add r5, r2, r1
i3: lw r2, 0(r2)        moved up to fill nop slot
nop
i2: lw r3, 4(r5)
nop                     had to add another nop here, so there
nop                     is no performance gain.
i4: or r3, r5, r3
nop
nop
i5: sw, r3, 0(r5)
```

4.14.1

Instructions	1	2	3	4	5	6	7	8	9	10	11	12	13	14
lw r2, 0(r1)	IF	ID	Ex	Mem	WB									
label1: beq r2, r0, label2		IF	ID	***	Ex	Mem	WB							
lw r3, 0(r2)						IF	ID	Ex	Mem	WB				
beq r3, r0, label1							IF	ID	**	Ex	Mem	WB		
label1: beq r2, r0, label2 (branch taken								IF	**	ID	Ex	Mem	WB	
sw r1, 0(r2)										IF	ID	Ex	Mem	WB

4.14.2

Instructions	1	2	3	4	5	6	7	8	9	10	11	12	13	14
lw r2, 0(r1)	IF	ID	Ex	Mem	WB									
label1: beq r2, r0, label 2		IF	ID	*	Ex	Mem	WB							
lw r3, 0(r2)			IF	*	ID	Ex	Mem	WB						
beq r3, r0, label1						IF	ID	Ex	Mem	WB				
add r1, r3, r1							IF	ID	Ex	Mem	WB			
label1: beq r2, r0, label2 (branch taken)								IF	ID	Ex	Mem	WB		
lw r3, 0(r2)									IF	ID	Ex	Mem	WB	
sw r1, 0(r2)										IF	ID	Ex	Mem	WB

4.15.1 The first three execution cycles are IF, ID, and EX.
The branch outcomes are determined in the EX stage.
Number of stall cycles due to mis-predicted branches = 3

Breakdown of dynamic instructions into BEQ = 25% = 0.25

Accuracy of the predicted branches with Always-Taken predictor = 45% = 0.45
Accuracy of the mis-predicted branches with Always-taken predictor =
$1.00 - 0.45 = 0.55$
Therefore, Extra CPI due to mis-predicted branches with Always-Taken predictor = $3 * 0.55 * 0.25$
$= 0.4125$
$\Rightarrow 0.41$