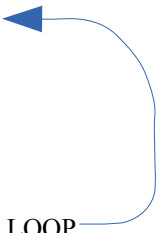


QUESTION 1**Vanilla 5-Cycle MIPS Execution Pipeline (1st Iteration)**

INSTR NO.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
1	IF	ID/R	EX	MEM	WB																	
2		IF	--	--	ID/R	EX	MEM	WB														
3					IF	--	--	ID/R	EX	MEM	WB											
4								IF	ID/R	EX	MEM	WB										
5									IF	--	--	ID/R	EX	MEM	WB							
6												IF	--	--	ID/R	EX	MEM	WB				
7															IF	ID/R	EX	MEM	WB			
8																IF	--	--	ID/R	EX	MEM	WB

For standard MIPS architecture we assume that forwarding logic is absent. Also, code and data memory is assumed to be separate (i.e. dual memory architecture). This helps avoid structural hazard when both MEM and IF stage are active in the same cycle of execution pipeline. Also, we will optimize branching using delayed branching. For delayed branching we insert instruction 1 in the delay slot after the BNE instruction and set the branch target of the BNE after instruction 1. Also the execution sequence is altered to optimize the execution. The resultant instruction sequence is as below:

Instr ⁿ #	Instr ⁿ
1	LW R2, 0(R10)
4
2
5
3
7
6
8	BNE R10, R30, LOOP
Delay Slot	LW R2, 0(R10)



Below is the execution pipeline during the first iteration (in black) and few instruction of the second iteration (in gray) for the above execution schedule. This schedule saves 9 cycles per iteration except for the last iteration, for which it saves 8 cycles due to the instruction in the delay slot.

For R10=0 and R30=40, number of iterations are 5.

$$\begin{aligned}
 \#Cycles_{\text{unscheduledCode}} &= (22 * \#iterations) - (3 * (\#iterations - 1)) + 1 = 99 \text{ cycles} \\
 \#Cycles_{\text{scheduledCode}} &= (15 * \#iterations) - (4 * (\#iterations - 1)) + 1 = 60 \text{ cycles} \\
 \text{cyclesSpeedup} &= \frac{99}{60} = 1.65
 \end{aligned}$$

After Rescheduling Instructions (Delayed Branching without Forwarding)

INSTR NO.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	IF	ID/R	EX	MEM	WB													
4		IF	ID/R	EX	MEM	WB												
2			IF	–	ID/R	EX	MEM	WB										
5					IF	ID/R	EX	MEM	WB									
3						IF	–	ID/R	EX	MEM	WB							
7								IF	ID/R	EX	MEM	WB						
6									IF	ID/R	EX	MEM	WB					
8										IF	–	ID/R	EX	MEM	WB			
1'												IF	ID/R	EX	MEM	WB		
4'													IF	ID/R	EX	MEM	WB	
...																		

QUESTION 2

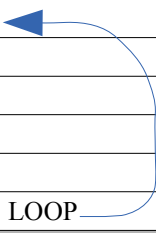
We assume code and data memory is separate (i.e. dual memory architecture). This helps avoid structural hazard when both MEM and IF stage are active in the same cycle of execution pipeline. We can re-factor the instruction set by eliminating one add instruction which is redundant. We are left with 6 instructions after refactoring as below.

Re-factored Instructions

INSTR NO.	
LOOP: 1	LW R2, 100(R6)
2	LW R3, 200(R7)
3	ADD R6, R3, R5
4	SUB R8, R2, R5
5	LW R7, 300(R8)
6	BEQ R7, R8, LOOP

We can reschedule the instructions as below with delayed branching to optimize execution.

Instr ⁿ #	Instr ⁿ
2	LW R3, 200(R7)
1
4
5
3
6	BEQ R7, R8, LOOP
Delay Slot	LW R3, 200(R7)



The pipeline for the rescheduled re-factored instruction sequence is seen in the second table below. It **enables execution of each iteration in 11 cycles**. The net number of cycles for execution can be computed as :

$$\#Cycles_{scheduledCode} = (11 * \#iterations) - (4 * (\#iterations - 1)) + 1$$

Execution Pipeline After Rescheduling Re-factored Instructions (Delayed Branching)

INSTR NO.	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	IF	ID/R	EX	MEM	WB									
1		IF	ID/R	EX	MEM	WB								
4			IF	ID/R	EX	MEM	WB							
5				IF	ID/R	EX	MEM	WB						
3					IF	ID/R	EX	MEM	WB					
6						IF	--	ID/R	EX	MEM	WB			
2'								IF	ID/R	EX	MEM	WB		
1'									IF	ID/R	EX	MEM	WB	
4'										IF	ID/R	EX	MEM	WB
...														

QUESTION 3

A) For the given architecture we require:

1. Incrementer in the IF pipeline stage to increment PC as we go through the execution.
2. Adder in ALU₁ pipeline stage for arithmetic operations.
3. Adder in ALU₂ pipeline stage for arithmetic operations.

Thus, **we require 3 adders** for this architecture. This requirement may be justified using the following instruction sequence. The pipeline stages highlighted in green show the three stages, requiring the use of adders, being utilized simultaneously. We assume that data forwarding logic is implemented in the architecture for this example.

Example Instruction Sequence

INSTR NO.				
LOOP: 1	ADD	R1,	R1,	16
2	SUB	R6,	R6,	1
3	LW	R3,	0(R1)	
4	ADD	R3,	R3,	R3
5	SW	R3,	100(R1)	SW
6	BNE	R6,	0	LOOP
...				

Execution Pipeline

INSTR NO.	1	2	3	4	5	6	7	8	9	10	11
1	IF	RF	ALU1	MEM	ALU2	WB					
2		IF	RF	ALU1	MEM	ALU2	WB				
3			IF	RF	ALU1	MEM	ALU2	WB			
4				IF	RF	ALU1	MEM	ALU2	WB		
5					IF	RF	ALU1	MEM	ALU2	WB	
6						IF	RF	ALU1	MEM	ALU2	WB
...											

B) A branch instruction, such as BNE R1, R2, R3, requires three registers to be read simultaneously in one cycle during the RF pipeline stage. The offset in R3 will be added to PC to compute the effective branch taken target address. This requires **three register read ports**. As for **register write ports, only one is required** since registers can be written only during the WB stage which is not concurrent i.e. at most one instruction can be in this stage.

For evaluating memory read and write ports we assume that both data and code are stored in a single memory module. Thus, considering the example instruction sequence in part A above, we see that instruction 3 reads data from memory simultaneously with instruction 6 being read from memory in clock cycle 6. This is highlighted in blue. We know that at most one memory operand is allowed by the architecture i.e. register-register and register-memory support only.

This suggests that we need **2 memory read ports**. In addition we need only **1 memory write port** as in any given cycle at most one instruction can be in the MEM stage during which it may write to memory.

C) Forwarding logic is needed in this architecture among various pipeline stages:

1. $ALU_2 \rightarrow MEM$: Write upstream result to memory downstream or use upstream result as downstream memory address to read from.

e.g. ...
 ADD R3, R1, R2
 ...
 SW R3, 0(R9)

2. $ALU_2 \rightarrow ALU_2$: Use upstream result in downstream calculation.

e.g. ...
 ADD R3, R1, R2
 SUB R6, R3, R5

Also handled by this logic is use of upstream data read from memory in downstream arithmetic operation and/or branch decision. In this case the buffer value is passed along the pipeline from the MEM to ALU_2 stage.

e.g. ...
 LW R1, 0(R2)
 ADD R3, R1, R2

3. $ALU_2 \rightarrow ALU_1$: Use upstream result for downstream effective address computation.

e.g. ...
 ADD R3, R1, R2
 ...
 ...
 SW R3, 100(R3)

4. $MEM \rightarrow ALU_1$: Use upstream data read from memory in downstream effective address/branch calculation.

e.g. ...
 LW R1, 0(R2)
 ...
 SW R3, 100(R1)

5. $WB \rightarrow ALU_2$: Use upstream data read from memory in downstream arithmetic operation and/or branch decision, but in this case there are two instructions between the LW and ADD instructions. So the buffered value in MEM needs to be passed along the pipeline from MEM to ALU_2 and then successively from ALU_2 to WB.

e.g. ...
 LW R1, 0(R2)
 ...
 ADD R3, R1, R2
 ...