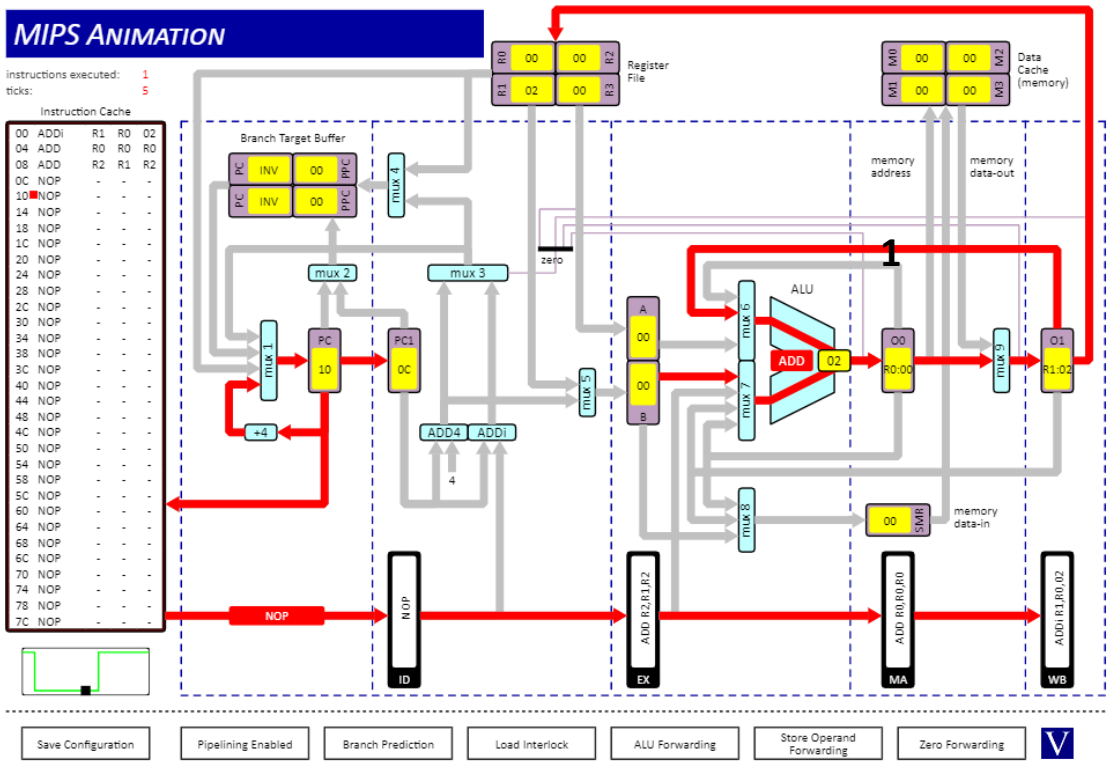
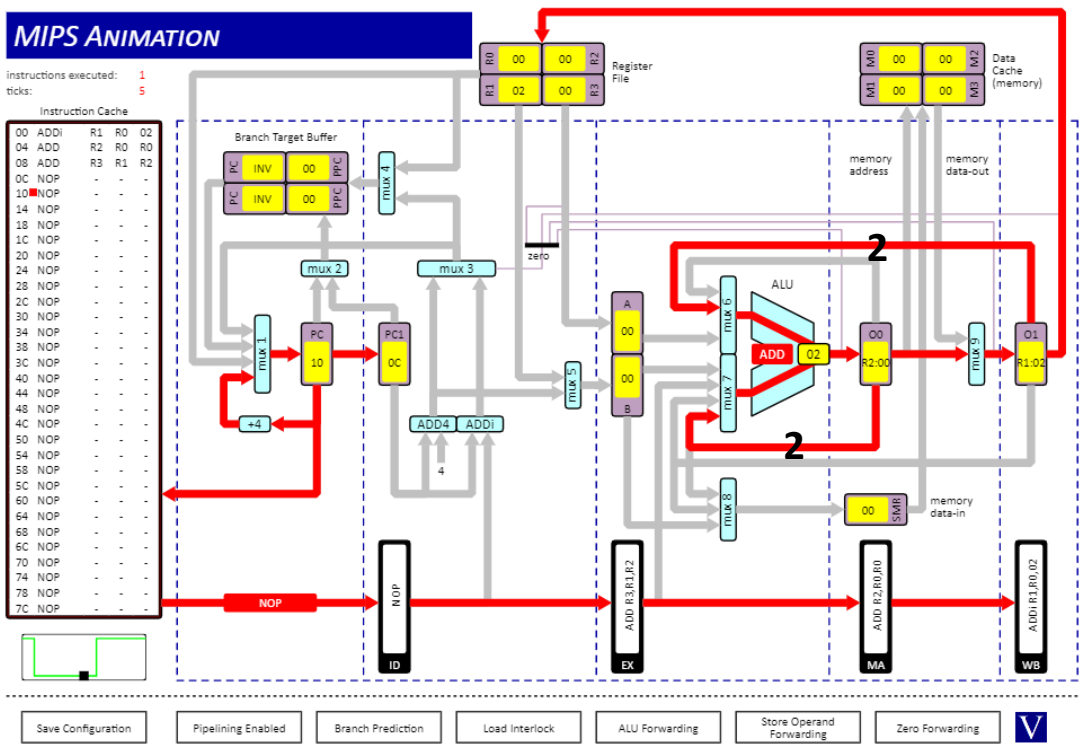
Tutorial 4 Sample Answer

Q1

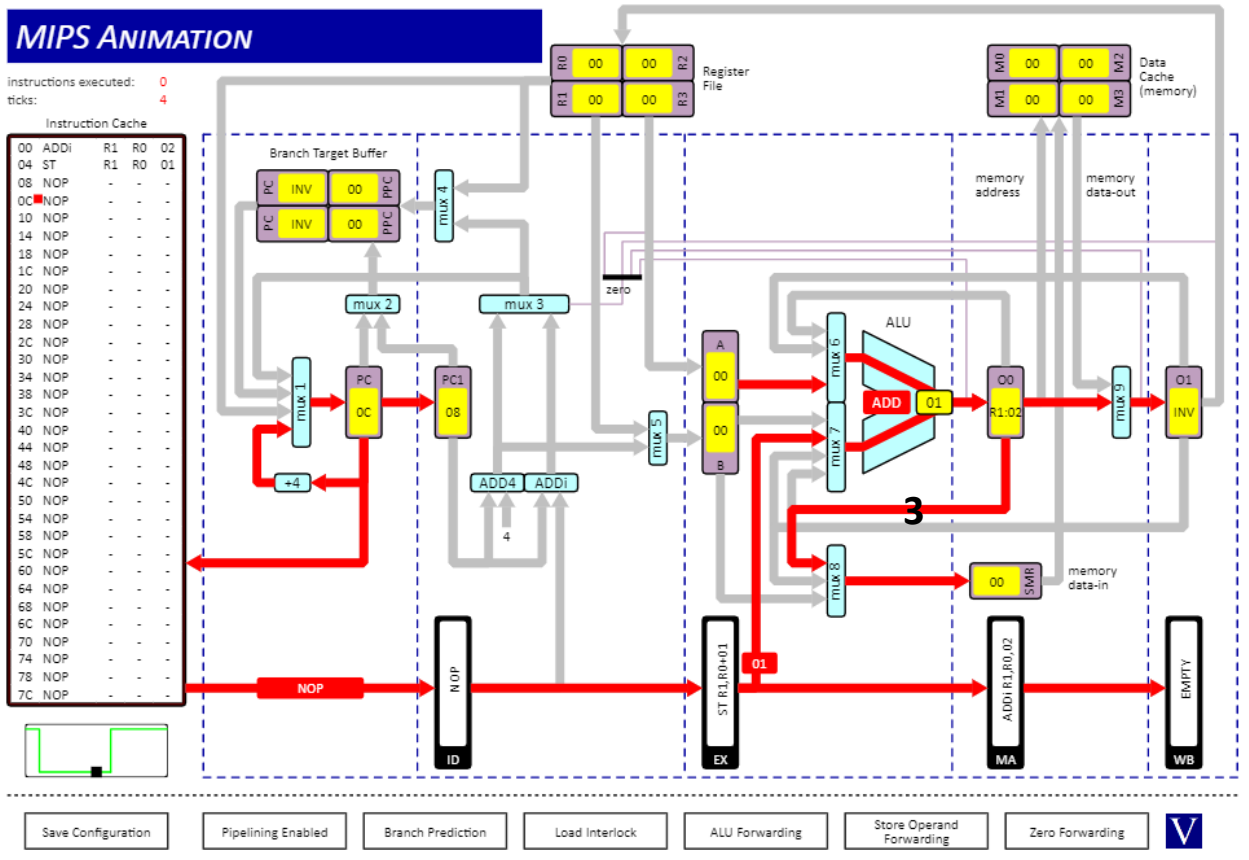
(i) O1 to MUX6



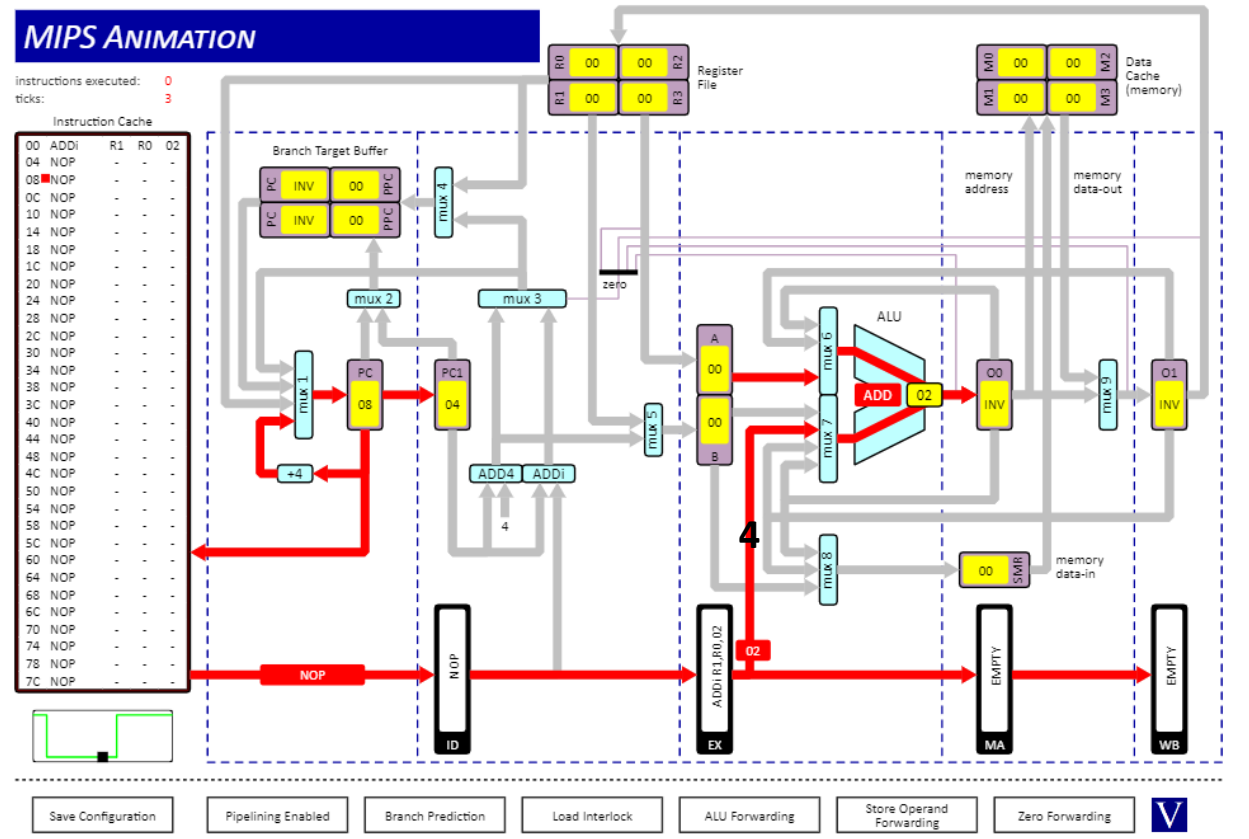
(ii) O0 to MUX7 and O1 to MUX6 (simultaneously)



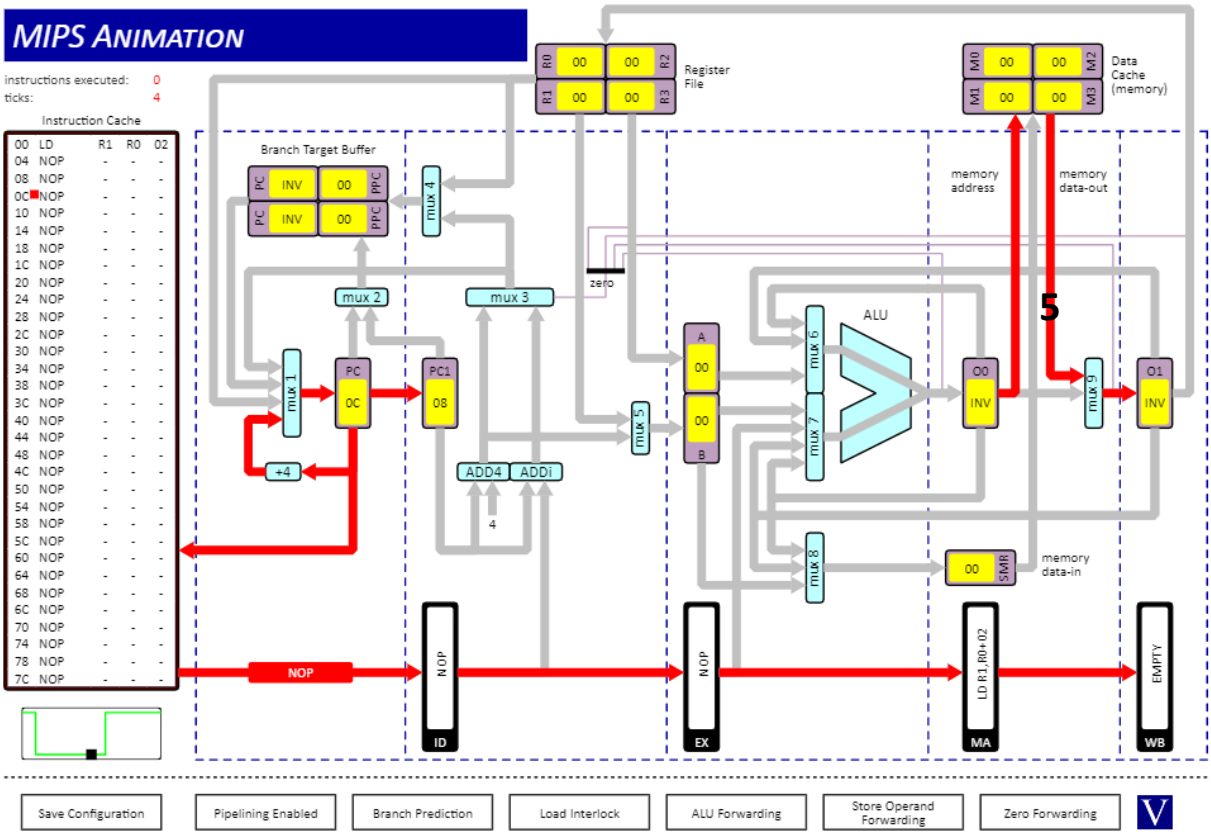
(iii) O0 to MUX8



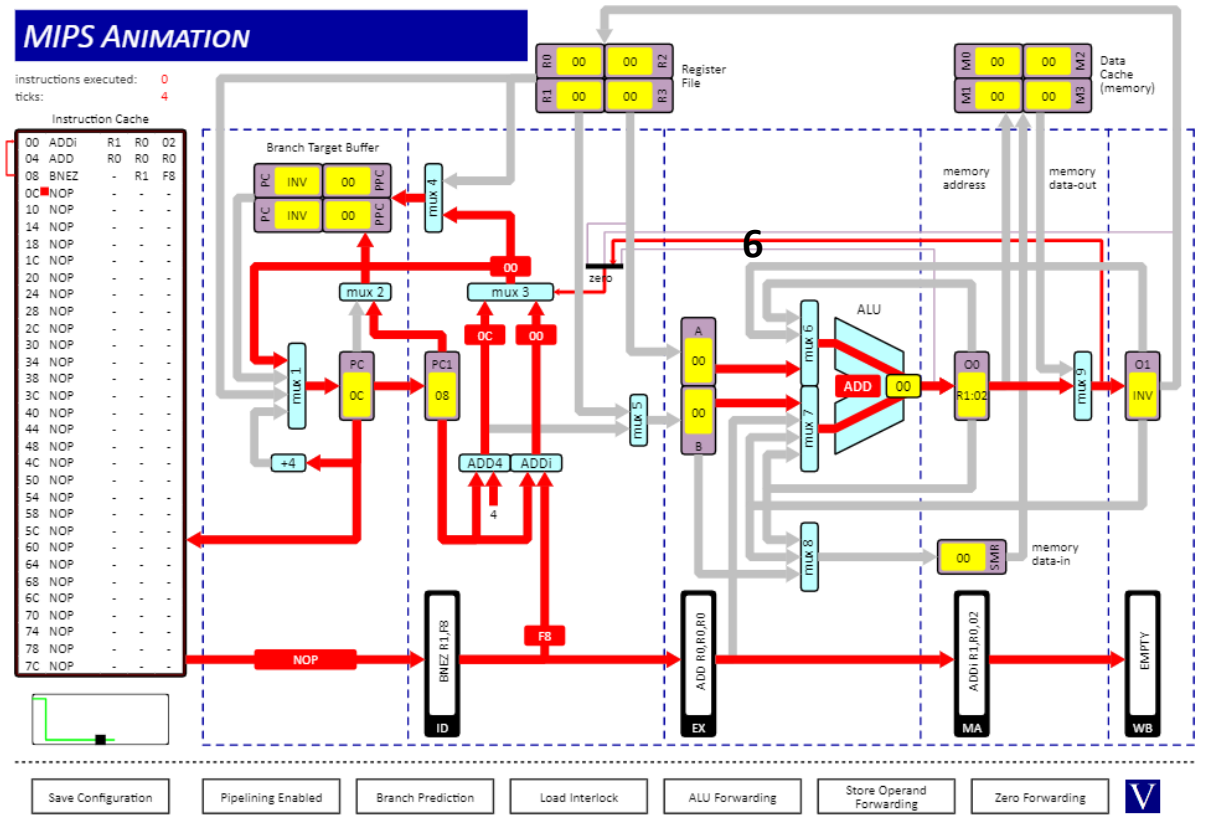
(iv) EX to MUX7



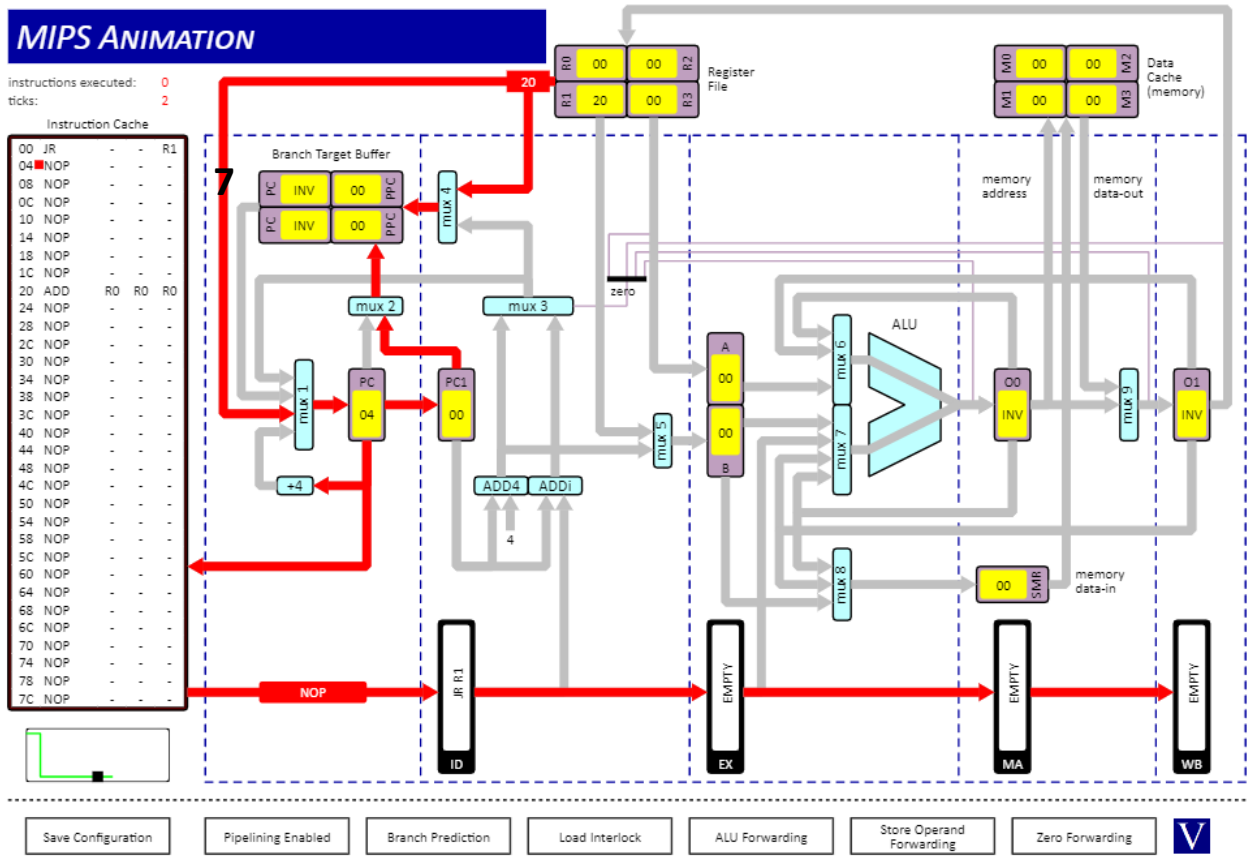
(v) Data cache to MUX9 (memory data-out)



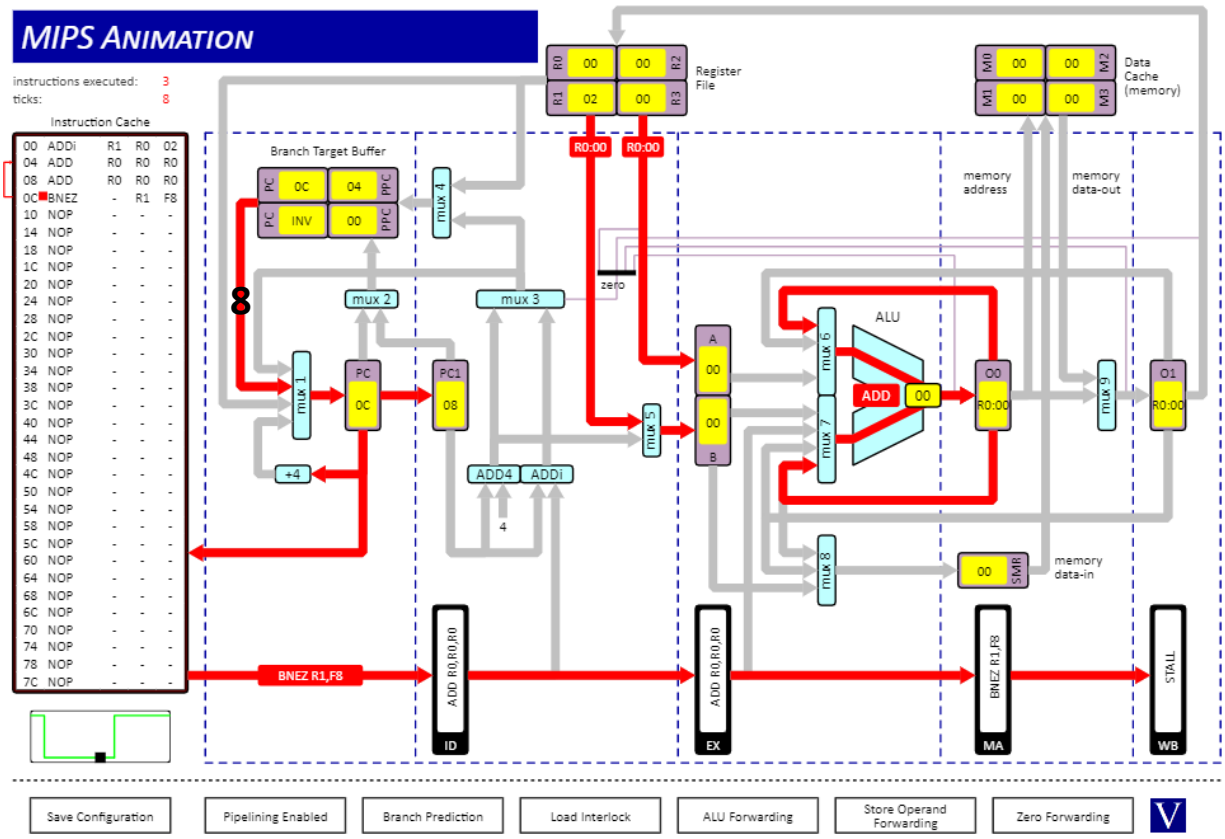
(vi) O0 to Zero detector



(vii) Register File to MUX1



(viii) Branch Target Buffer to MUX1



Q2 The table below shows the computed result and number of ticks needed to execute the code segment including the HALT instruction at the end.

	result in r1	ticks
ALU Forwarding	0x15	10
ALU Interlock	0x15	18
No ALU interlock	0x06	10

ALU forwarding and *ALU interlock* both generate the same result (0x15). This is the result assuming conventional instruction operation. See table below.

	r1 (0x01)	r2 (0x02)
r1 = r1 + r2	r1 = 0x03	
r2 = r1 + r2		r2 = 0x05
r1 = r1 + r2	r1 = 0x08	
r2 = r1 + r2		r2 = 0x0d
r1 = r1 + r2	r1 = 0x15	

No ALU interlock uses the register values obtained from the register file during the ID phase. This may be before the results of previous instructions have been written back to the register file. How the result of 0x06 is obtained is illustrated in the table below. NB: two phase register file access is still operational.

	RF r1	r1 + r2	RF r2
r1 = r1 + r2	0x01	0x03	0x02
r2 = r1 + r2	0x01	0x03	0x02
r1 = r1 + r2	0x01	0x03	0x02
r2 = r1 + r2	0x03	0x05	0x02
r1 = r1 + r2	0x03	0x06	0x03

ALU forwarding and *No ALU interlock* both take 10 ticks to execute. Four ticks to fill the pipeline and six ticks to execute the six instructions including HALT.

ALU interlock adds an additional eight ticks as, after the first add instruction, the remaining four add instructions stall for 2 ticks each while waiting for the result(s) of the previous instructions to propagate to the register file. This is illustrated in the table below.

	data stalls	reason
r1 = r1 + r2	0	
r2 = r1 + r2	2	wait for r1
r1 = r1 + r2	2	wait for r2 (and r1)
r2 = r1 + r2	2	wait for r1 (and r2)
r1 = r1 + r2	2	wait for r2 (and r1)

Q3 (i) The program computes r1 = r2 * r3 using Booth's shift and add multiply algorithm. The code uses memory location 0 for temporary storage. As r2 and r3 are initially 0x09 and 0x08 respectively, the computed result is 0x48 (72_{10}).

(ii) With *branch prediction* enabled, it takes 38 instructions and 50 ticks for the code to execute. The instruction execution order is shown in the table below.

XOR	1				
BEQZ	2	11	19	27	36 **+1**
ST	3	12	20	28	
ANDi	4	13	21	29	
BEQZ	5	14 **+1**	22	30 **+1**	
ADD	6			31	
LD	7	15	23	32	
SRLi	8 *+1*	16 *+1*	24 *+1*	33 *+1*	
SLLi	9	17	25	34	
J	10 **+1**	18	26	35	
ST					37
HALT					38

The 50 ticks comprise 4 ticks to fill the pipeline, 38 ticks for the 38 instructions and 8 ticks for the stalls marked as +1 in the table. An *italic +1* indicates a data stall (load hazard in this case) and a **bold +1** indicates a control stall. For example, the second BEQZ is predicted correctly the first time it is executed as it doesn't branch (this is the default prediction as the branch is not in the branch target buffer), predicted incorrectly the second time as it now branches, correctly the third time as it branches again and incorrectly the fourth time as it doesn't branch.

(iii) With *branch interlock* enabled, it takes 38 instructions and 53 ticks for the code to execute. The difference is that, without branch prediction, every branch (jump) that branches requires a control stall. These are shown in the table below.

XOR	1				
BEQZ	2	11	19	27	36 **+1**
ST	3	12	20	28	
ANDi	4	13	21	29	
BEQZ	5	14 **+1**	22 **+1**	30	
ADD	6			31	
LD	7	15	23	32	
SRLi	8 *+1*	16 *+1*	24 *+1*	33 *+1*	
SLLi	9	17	25	34	
J	10 **+1**	18 **+1**	26 **+1**	35 **+1**	
ST					37
HALT					38

(iv) Swapping the two shift instructions removes the four data stalls (load hazards). This is because the SRLi instruction uses the result of the LD. This results in the 38 instructions being executed in 46 ticks. This is shown in the table below.

XOR	1				
BEQZ	2	11	19	27	36 **+1**
ST	3	12	20	28	
ANDi	4	13	21	29	
BEQZ	5	14 **+1**	22	30 **+1**	
ADD	6			31	
LD	7	15	23	32	
SLLi	8	16	24	33	
SRLi	9	17	25	34	
J	10 **+1**	18	26	35	
ST					37
HALT					38