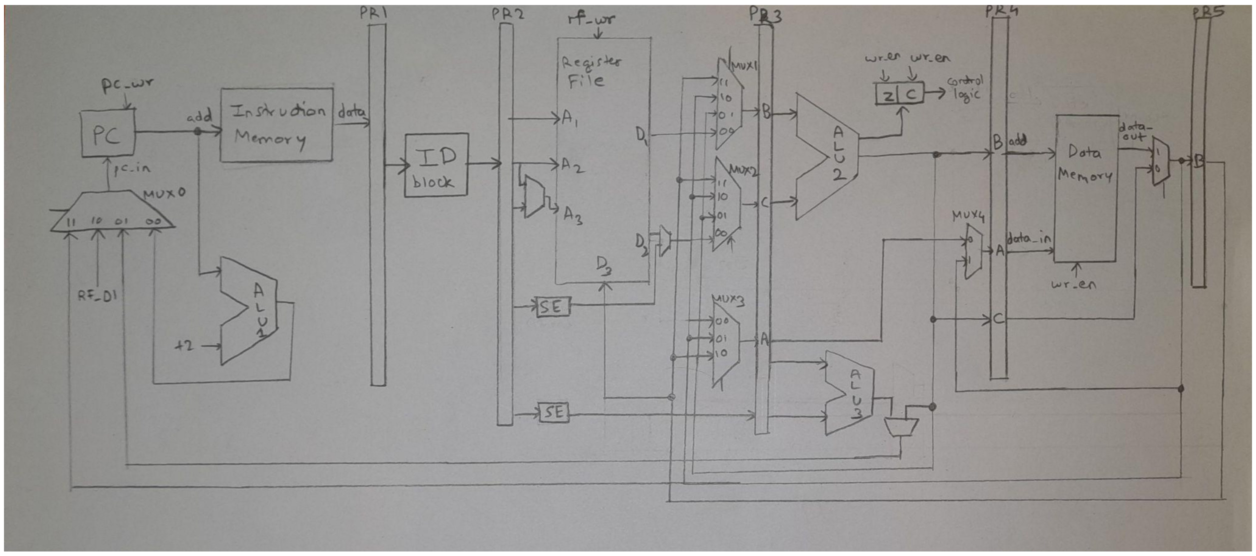
EE 309 Project - Pipelined RISC

Atharva Kulkarni (210070047)
Harshit Raj (20D070033)
Shreyas Grampurohit (21d070029)
Varad Deshpande (21d070024)

Table Of Contents

DataPath:



Inputs

Clock
Reset: Resets PC to 0, does not modify any other storage unit.

Instruction Memory

Stores instructions which are executed. This is preprogrammed by the programmer (similar to flashing the ROM of 8051) and it is read only.

Register File

Stores programmers registers R0 to R7, where PC = R0.

ALU

Operations: Add, subtract (for comparison in BEQ, etc.), nand, encoded with alu__j bits 00, 01, 10 respectively.

Data Memory

Accessed in the memory access stage. Involved in load-store operations. Can be both read and written to.

Flags

Carry and Zero Flag. These are used and updated as per instruction specification.

Pipeline Registers

(A, B and C are 16 bit data which are required in particular stages of instructions. For more details, read the data flow inside each instruction.)

Indices	84	83	82-80	79-64	63-48	47-32	31-16	15-0
PR1							Instr.	PC*+2
PR2	m_wr’	rf_wr’	ctr1**	C	B	A	Instr.	PC*+2
PR3	m_wr’	rf_wr’		C	B	A	Instr.	PC*+2
PR4	m_wr’	rf_wr’		C	B	A	Instr.	PC*+2
PR5		rf_wr’		C	B	A	Instr.	PC*+2

PC* corresponds to instruction stored in 31-16.
ctr1** is 3 bit counter output utilized in LM and SM.
rf_wr’ and m_wr’ : control signals used only in case of LM and SM.
For more details, read instruction-wise data flow.

Hazard Mitigation:

MUX 0 to 5, and data forwarding lines.

Stages:

Instruction Fetch

The instruction is fetched from instruction memory and stored into pipeline register. We also update PC using an ALU as adder and also store this in pipeline register.

Instruction Decode

It passes the data from Pipeline Register 1 to Pipeline Register 2 generally. But in case of Instructions “LM” and “SM”, it pauses the pipeline and introduces 8 separate load/store instructions for the 8 registers.

Note: Actual decoding doesn’t take place here, we pass the instruction in the pipeline register and appropriate control signals are generated in each next stage locally. This was done for simplicity.

Register Read

Operands are read from register file and sent into next stage for operations. This stage also modifies operands as needed for further stages. Eg. by using signed extenders.

Execute

The operation needed for the instruction is executed in this stage. It includes one ALU which has the hardware required for doing operations add, nand and sub (sub for comparison in BEQ), and another ALU used as adder for computing address of branch type instructions.

Memory Access

This stage is involved in LOAD-STORE instructions, to read and write to the data memory.

Write Back

Result is written to the destination register as intended by the instruction.

Hazards and their Mitigation:

Load Multiple and Store Multiple:

For this, in the Instruction Decode stage, we pause the pipeline by stopping the Pipeline Register 1 and PC from updating, for 7 cycles. We then send back-to-back load/store instructions respectively to the further stages which behave exactly like a single LW/SW instruction. We change the register and memory address pointers using two counters inside the Instruction Decode stage.

In the Register Read stage, we store the initial value of the address in a temporary register RefAdd to prevent it from getting changed while writing the RegFile in LM and later access it from there.

Branching (Conditional/Unconditional):

Except JLR, in all such instructions, we do not know the destination location until the Execute stage. If the branch is taken, we already have 3 instructions in the pipeline. Thus, we clear the write-enables of these instructions and let them pass through the pipeline. So they do not modify anything.

In case of JLR, we get to know the target location in the Register Read stage itself. Hence, we branch in that stage and clear the write-enables of the previous two unintended instructions in the pipeline so that they do not modify anything.

We use this target value to write to PC using MUX 0.

Arithmetic, Logical instructions and LLI:

In the instructions of the above type where R is used as destination first (A&L or LLI) and in a later instruction (A&L), as operand, there is a hazard violating the 'independence of states' assumption. Data forwarding as shown in the datapath (MUX 1 and 2) mitigates this hazard.

This hazard can occur only for 1,2,3 dependency, higher ones don't lead to hazard.

Load instructions followed by Arithmetic and Logical:

In the above case, if the dependency is of distance 2 or 3, we can use the above data forwarding to solve the hazard (using Mux 1 and 2).

But, if the dependency is immediate (excluding LLI, because we know the value to be stored in the destination register through the immediate data), we pause the pipeline for a cycle such that there is a gap of one instruction between the two hazard instructions. This makes this a 2-dependency, the solution for which has just been discussed.

This hazard can occur only for 1,2,3 dependency, higher ones don't lead to hazard.

Load instructions followed by Store:

If the dependency distance is 2 or 3, then we simply use the previous data forwarding approach but we need to forward to a separate location. Thus we use MUX 3.

If we have an immediate dependency, we can't use any of the previous approaches because the load instruction only gives the data at the Memory Access stage and by that time, the store instruction is already in Execute stage. So we need to forward from Memory Access stage to Execute stage rather than Register Read (using MUX 4).

This hazard can occur only for 1,2,3 dependency, higher ones don't lead to hazard.

Arithmetic and Logical instructions followed by Store:

If this happens, we can simply use the previous data forwarding approach (using MUX 3) for dependencies of distance 1,2,3.

This hazard can occur only for 1,2,3 dependency, higher ones don't lead to hazard.

Arithmetic and Logical instructions writing at R0 (including LLI):

This hazard is very similar to branch or jump instructions, the only difference is that we know the value of PC at the end of Execute stage. But this value is written only in the Write Back stage. Thus, we handle this hazard just like the instructions where control flow changes (i.e. which involve branching at Execute stage) by clearing the write-enables of the incoming 3 instructions. We thus use Mux 0.

Load at R0 (excluding LLI):

This is similar to the previous hazard, but here, we get to know the correct value of R0 only at the Memory Access stage. Hence we need to make one more connection to the Mux 0 connecting to the PC and use an approach similar to the previous one, that is, clearing the write-enables of the incoming 4 instructions.

JLR followed by Arithmetic and Logical or Store:

This only happens when there is a 3 dependency. For lesser dependency, JLR already disables those instructions from modifying anything. We use data forwarding for this hazard(using MUX 1 and 2).

Instructions:

ADA (00_01 RA RB RC 0 00)

Instruction Fetch	PC → Mem1_add, ALU1_A +2 → ALU1_B ALU1_C → PC Mem1_D → Instr_R1	PC_WR ALU1 → ADD
Instruction Decode	PR1 → PR2	
Register Read	Instr_R2(8-6),(5-3) → RF_A1,A2 Instr_R2(11-9) → SE16 → A_R3 RF_D1 → B_R3 RF_D2 → C_R3	
Execute	A_R3 → A_R4 B_R3 → ALU2_A C_R3 → ALU2_B 0 → ALU2_Cin ALU2_C → C_R4	ALU2 → ADD C_WR Z_WR
Memory Access	C_R4 → C_R5 A_R4 → A_R5	
Write Back	C_R5 → RF_D3 A_R5(2-0) → RF_A3	RF_WR

ADC (00_01 RA RB RC 0 10)

Instruction Fetch	PC → Mem1_add, ALU1_A +2 → ALU1_B ALU1_C → PC Mem1_D → Instr_R1	PC_WR ALU1 → ADD
Instruction Decode	PR1 → PR2	

Register Read	Instr__R2(8-6),(5-3) → RF__A1,A2 Instr__R2(11-9) → SE16 → A__R3 RF__D1 → B__R3 RF__D2 → C__R3	

Execute	A__R3 → A__R4
	B__R3 → ALU2__A
	C__R3 → ALU2__B
	0 → ALU2__Cin
	ALU2__C → C__R4
		ALU2 → ADD
		if(C==1){
		C__WR
		Z__WR
		}

Memory Access	C__R4 → C__R5 A__R4 → A__R5	

Write Back	if(C=1){
	C__R5 → RF__D3
	A__R5(2-0) → RF__A3
	}
		if(C==1){
		RF__WR
		C__WR
		Z__WR
		}

ADZ (00__01 RA RB RC 0 01)

Instruction Fetch	PC → Mem1__add, ALU1__A +2 → ALU1__B ALU1__C → PC Mem1__D → Instr__R1	PC__WR ALU1 → ADD
Instruction Decode	PR1 → PR2	
Register Read	Instr__R2(8-6),(5-3) → RF__A1,A2 Instr__R2(11-9) → SE16 → A__R3 RF__D1 → B__R3 RF__D2 → C__R3	

Execute	A__R3 → A__R4
	B__R3 → ALU2__A
	C__R3 → ALU2__B
	0 → ALU2__Cin
	ALU2__C → C__R4
		ALU2 → ADD
		if(Z==1){
		C__WR
		Z__WR
		}

Memory Access	C__R4 → C__R5 A__R4 → A__R5	
Write Back	if(Z=1){ C__R5 → RF__D3 A__R5(2-0) → RF__A3	if(Z==1){ RF__WR C__WR

	}	Z__WR }

AWC (00__01 RA RB RC 0 11)

Instruction Fetch	PC → Mem1__add, ALU1__A +2 → ALU1__B ALU1__C → PC Mem1__D → Instr__R1	PC__WR ALU1 → ADD
Instruction Decode	PR1 → PR2	
Register Read	Instr__R2(8-6),(5-3) → RF__A1,A2 Instr__R2(11-9) → SE16 → A__R3 RF__D1 → B__R3 RF__D2 → C__R3	
Execute	A__R3 → A__R4 B__R3 → ALU2__A C__R3 → ALU2__B C → ALU2__Cin ALU2__C → C__R4	ALU2 → ADD C__WR Z__WR
Memory Access	C__R4 → C__R5 A__R4 → A__R5	
Write Back	C__R5 → RF__D3 A__R5(2-0) → RF__A3	RF__WR

ACA (00__01 RA RB RC 1 00)

Instruction Fetch	PC → Mem1__add, ALU1__A +2 → ALU1__B ALU1__C → PC Mem1__D → Instr__R1	PC__WR ALU1 → ADD
Instruction Decode	PR1 → PR2	
Register Read	Instr__R2(8-6),(5-3) → RF__A1,A2 Instr__R2(11-9) → SE16 → A__R3 RF__D1 → B__R3 RF__D2 → COMP→ C__R3	

Execute	A_R3 → A_R4 B_R3 → ALU2_A C_R3 → ALU2_B 0 → ALU2_Cin ALU2_C → C_R4	ALU2 → ADD C_WR Z_WR
Memory Access	C_R4 → C_R5 A_R4 → A_R5	
Write Back	C_R5 → RF_D3 A_R5(2-0) → RF_A3	RF_WR

ACC (00_01 RA RB RC 1 10)

Instruction Fetch	PC → Mem1_add, ALU1_A +2 → ALU1_B ALU1_C → PC Mem1_D → Instr_R1	PC_WR ALU1 → ADD
Instruction Decode	PR1 → PR2	
Register Read	Instr_R2(8-6),(5-3) → RF_A1,A2 Instr_R2(11-9) → SE16 → A_R3 RF_D1 → B_R3 RF_D2 → COMP → C_R3	

Execute	A_R3 → A_R4
	B_R3 → ALU2_A
	C_R3 → ALU2_B
	0 → ALU2_Cin
	ALU2_C → C_R4
		ALU2 → ADD
		if(C==1){
		C_WR
		Z_WR
		}

Memory Access	C_R4 → C_R5 A_R4 → A_R5	

Write Back	if(C=1){
	C_R5 → RF_D3
	A_R5(2-0) → RF_A3
	}
		if(C==1){
		RF_WR
		C_WR
		Z_WR
		}

ACZ (00_01 RA RB RC 1 01)

Instruction Fetch	PC → Mem1_add, ALU1_A	PC_WR

	+2 → ALU1_B ALU1_C → PC Mem1_D → Instr_R1	ALU1 → ADDs
Instruction Decode	PR1 → PR2	
Register Read	Instr_R2(8-6),(5-3) → RF_A1,A2 Instr_R2(11-9) → SE16 → A_R3 RF_D1 → B_R3 RF_D2 → COMP→ C_R3	

Execute	A_R3 → A_R4
	B_R3 → ALU2_A
	C_R3 → ALU2_B
	0 → ALU2_Cin
	ALU2_C → C_R4
		ALU2 → ADD
		if(Z==1){
		C_WR
		Z_WR
		}

Memory Access	C_R4 → C_R5 A_R4 → A_R5	

Write Back	if(Z=1){
	C_R5 → RF_D3
	A_R5(2-0) → RF_A3
	}
		if(Z==1){
		RF_WR
		C_WR
		Z_WR
		}

ACW (00_01 RA RB RC 1 11)

Instruction Fetch	PC → Mem1_add, ALU1_A +2 → ALU1_B ALU1_C → PC Mem1_D → Instr_R1	PC_WR ALU1 → ADD
Instruction Decode	PR1 → PR2	
Register Read	Instr_R2(8-6),(5-3) → RF_A1,A2 Instr_R2(11-9) → SE16 → A_R3 RF_D1 → B_R3 RF_D2 → COMP→ C_R3	
Execute	A_R3 → A_R4 B_R3 → ALU2_A C_R3 → ALU2_B C → ALU2_Cin ALU2_C → C_R4	ALU2 → ADD C_WR Z_WR

Memory Access	C__R4 → C__R5 A__R4 → A__R5	
Write Back	C__R5 → RF__D3 A__R5(2-0) → RF__A3	RF__WR

ADI (00__00__RA__RB__6 bit Immediate)

Instruction Fetch	PC → Mem1__add, ALU1__A +2 → ALU1__B ALU1__C → PC Mem1__D → Instr__R1	PC__WR
Instruction Decode	PR1 → PR2	
Register Read	Instr__R2(11-9) → RF__A1 RF__D1 → B__R3 Instr__R2(8-6) → SE3 → A__R3 Instr__R2(5-0) → SE6 → C__R3	
Execute	A__R3 → A__R4 B__R3 → ALU2__A C__R3 → ALU2__B 0 → ALU2__Cin ALU2__C → C__R4	ALU2 → ADD C__WR Z__WR
Memory Access	C__R4 → C__R5 A__R4 → A__R5	
Write Back	C__R5 → RF__D3 A__R5(2-0) → RF__A3	RF__WR

NDU (00__10__RA__RB__RC__0__00)

Instruction Fetch	PC → Mem1__add, ALU1__A +2 → ALU1__B ALU1__C → PC Mem1__D → Instr__R1	PC__WR
Instruction Decode	PR1 → PR2	
Register Read	Instr__R2(8-6),(5-3) → RF__A1,A2 Instr__R2(11-9) → SE16 → A__R3 RF__D1 → B__R3	

	RF__D2 → C__R3	
Execute	A__R3 → A__R4 B__R3 → ALU2__A C__R3 → ALU2__B 0 → ALU2__Cin ALU2__C → C__R4	ALU2 → NAND Z__WR
Memory Access	C__R4 → C__R5 A__R4 → A__R5	
Write Back	C__R5 → RF__D3 A__R5(2-0) → RF__A3	RF__WR

NDC (00__10__RA__RB__RC__0__10)

Instruction Fetch	PC → Mem1__add, ALU1__A +2 → ALU1__B ALU1__C → PC Mem1__D → Instr__R1	PC__WR
Instruction Decode	PR1 → PR2	
Register Read	Instr__R2(8-6),(5-3) → RF__A1,A2 Instr__R2(11-9) → SE16 → A__R3 RF__D1 → B__R3 RF__D2 → C__R3	
Execute	A__R3 → A__R4 B__R3 → ALU2__A C__R3 → ALU2__B 0 → ALU2__Cin ALU2__C → C__R4	ALU2 → NAND if(C==1) Z__WR
Memory Access	C__R4 → C__R5 A__R4 → A__R5	
Write Back	C__R5 → RF__D3 A__R5(2-0) → RF__A3	if(C==1) RF__WR

NDZ (00__10__RA__RB__RC__0__01)

Instruction Fetch	PC → Mem1__add, ALU1__A	PC__WR

	+2 → ALU1_B ALU1_C → PC Mem1_D → Instr_R1	
Instruction Decode	PR1 → PR2	
Register Read	Instr_R2(8-6),(5-3) → RF_A1,A2 Instr_R2(11-9) → SE16 → A_R3 RF_D1 → B_R3 RF_D2 → C_R3	
Execute	A_R3 → A_R4 B_R3 → ALU2_A C_R3 → ALU2_B 0 → ALU2_Cin ALU2_C → C_R4	ALU2 → NAND if(Z==1) Z_WR
Memory Access	C_R4 → C_R5 A_R4 → A_R5	
Write Back	C_R5 → RF_D3 A_R5(2-0) → RF_A3	if(Z==1) RF_WR

NCU (00_10_RA_RB_RC_1_00)

Instruction Fetch	PC → Mem1_add, ALU1_A +2 → ALU1_B ALU1_C → PC Mem1_D → Instr_R1	PC_WR ALU1 → ADD
Instruction Decode	PR1 → PR2	
Register Read	Instr_R2(8-6),(5-3) → RF_A1,A2 Instr_R2(11-9) → SE16 → A_R3 RF_D1 → B_R3 RF_D2 → COMP → C_R3	
Execute	A_R3 → A_R4 B_R3 → ALU2_A C_R3 → ALU2_B 0 → ALU2_Cin ALU2_C → C_R4	ALU2 → NAND Z_WR
Memory Access	C_R4 → C_R5 A_R4 → A_R5	

Write Back	C_R5 → RF_D3 A_R5(2-0) → RF_A3	RF_WR

NCC (00_10_RA_RB_RC_1_10)

Instruction Fetch	PC → Mem1_add, ALU1_A +2 → ALU1_B ALU1_C → PC Mem1_D → Instr_R1	PC_WR ALU1 → ADD
Instruction Decode	PR1 → PR2	
Register Read	Instr_R2(8-6),(5-3) → RF_A1,A2 Instr_R2(11-9) → SE16 → A_R3 RF_D1 → B_R3 RF_D2 → COMP→ C_R3	
Execute	A_R3 → A_R4 B_R3 → ALU2_A C_R3 → ALU2_B 0 → ALU2_Cin ALU2_C → C_R4	ALU2 → NAND if(C==1) Z_WR
Memory Access	C_R4 → C_R5 A_R4 → A_R5	
Write Back	C_R5 → RF_D3 A_R5(2-0) → RF_A3	if(C==1) RF_WR

NCZ (00_10_RA_RB_RC_1_01)

Instruction Fetch	PC → Mem1_add, ALU1_A +2 → ALU1_B ALU1_C → PC Mem1_D → Instr_R1	PC_WR ALU1 → ADD
Instruction Decode	PR1 → PR2	
Register Read	Instr_R2(8-6),(5-3) → RF_A1,A2 Instr_R2(11-9) → SE16 → A_R3 RF_D1 → B_R3 RF_D2 → COMP→ C_R3	
Execute	A_R3 → A_R4	ALU2 → NAND

	B__R3 → ALU2__A C__R3 → ALU2__B 0 → ALU2__Cin ALU2__C → C__R4	if(C==1) Z__WR
Memory Access	C__R4 → C__R5 A__R4 → A__R5	
Write Back	C__R5 → RF__D3 A__R5(2-0) → RF__A3	if(C==1) RF__WR

LLI (00__11__RA__9bitImm)

Instruction Fetch	PC → Mem1__Add, ALU1__A, PC__R1 +2 → ALU1__B ALU1__C → PC Mem1__D → Instr__R1	ALU1 → add PC__WR → 1
Instruction Decode	PR1 → PR2	
Register Read	Instr__R2(11-9) → SE3 → A__R3 Instr__R2(8-0) → SE9 → C__R3	
Execute	A__R3 → A__R4 C__R3 → C__R4	ALU2 → none Z__WR → 0 C__WR → 0
Memory Access	A__R4 → A__R5 C__R4 → C__R5	Mem2__RD → 0 Mem2__WR → 0
Write Back	A__R5(2-0) → RF__A3 C__R5 → RF__D3	RF__WR → 1

LW (01__00__RA__RB__6bitImm)

Instruction Fetch	PC → Mem1__Add, ALU1__A, PC__R1 +2 → ALU1__B ALU1__C → PC Mem1__D → Instr__R1	ALU1 → add PC__WR → 1
Instruction Decode	PR1 → PR2	
Register Read	Instr__R2(11-9) → SE3 → A__R3	

	Instr__R2(8-6) → RF__A2 RF__D2 → B__R3 Instr__R2(5-0) → SE6 → C__R3	
Execute	A__R3 → A__R4 B__R3 → ALU2__A C__R3 → ALU2__B ALU2__C → B__R4	ALU2 → add Z__WR → 1
Memory Access	A__R4 → A__R5 B__R4 → Mem2__Add Mem2__Data → B__R5	Mem2__RD → 1 Mem2__WR → 0
Write Back	A__R5(2-0) → RF__A3 B__R5 → RF__D3	RF__WR → 1

SW (01__01__RA__RB__6bitImm)

Instruction Fetch	PC → Mem1__Add, ALU1__A, PC__R1 +2 → ALU1__B ALU1__C → PC Mem1__D → Instr__R1	ALU1 → add PC__WR → 1
Instruction Decode	PR1 → PR2	
Register Read	Instr__R2(11-9) → RF__A1 Instr__R2(8-6) → RF__A2 RF__D1 → A__R3 RF__D2 → B__R3 Instr__R2(5-0) → SE6 → C__R3	
Execute	A__R3 → A__R4 B__R3 → ALU2__A C__R3 → ALU2__B ALU2__C → B__R4	ALU2 → add
Memory Access	B__R4 → Mem2__Add A__R4 → Mem2__Data	Mem2__RD → 0 Mem2__WR → 1
Write Back	NIL	RF__WR → 0

LM (01_10_RA_0+8bitR7-R0)

Instruction Fetch	PC → Mem1_Add, ALU1_A, PC_R1 +2 → ALU1_B ALU1_C → PC Mem1_D → Instr_R1	ALU1 → add PC_WR → 1
Instruction Decode	If 3 bit counter1 == “111”: Pipeline_Register_1 enable → 1 Else: Pipeline_Register_1 enable → 0 If 3 bit counter1 == “000”: 3 bit counter2 = “000” Convert: Instr_R1(15-12) → Instr_R2(15-12) 3 bit counter1 → Instr_R2(11-9) 3 bit counter2 → SE3to6 → Instr_R2(5-0) Instr_R1(11-9) → Instr_R2(8-6) If Instr_R1(3 bit counter1) == 1: //Enable in Imm is 1 ControlSig_R2(RF_WR) → 1 3 bit counter2 ++ 3 bit counter1 → Counter1_R2 3 bit counter1 ++	
Register Read	Instr_R2(11-9) → SE3 → A_R3 //Reg to write If Counter1_R2 == “000”: Instr_R2(8-6) → RF_A2 RF_D2 → RefAdd RefAdd → B_R3 //Mem loc to write Instr_R2(5-0) → SE6 → C_R3 //Offset	
Execute	A_R3 → A_R4 B_R3 → ALU2_A C_R3 → ALU2_B ALU2_C → B_R4 //ALU2_Z → Z-flag	ALU2 → add Z_WR → 0 C_WR → 0

Memory Access	A_R4 → A_R5 B_R4 → Mem2_Add Mem2_Data → B_R5	Mem2_RD → 1 Mem2_WR → 0
Write Back	A_R5(2-0) → RF_A3 B_R5 → RF_D3	ControlSig_R 5(RF_WR) → RF_WR

SM (01_11_RA_0+8bitR7-R0)

Instruction Fetch	PC → Mem1_Add, ALU1_A, PC_R1 +2 → ALU1_B ALU1_C → PC Mem1_D → Instr_R1	ALU1 → add PC_WR → 1
Instruction Decode	If 3 bit counter1 == "111": Pipeline_Register_1 enable → 1 Else: Pipeline_Register_1 enable → 0 If 3 bit counter1 == "000": 3 bit counter2 = "000" Convert: Instr_R1(15-12) → Instr_R2(15-12) 3 bit counter1 → Instr_R2(11-9) 3 bit counter2 → SE3to6 → Instr_R2(5-0) Instr_R1(11-9) → Instr_R2(8-6) If Instr_R1(3 bit counter1) == 1: //Enable in Imm is 1 ControlSig_R2(Mem2_WR) → 1 3 bit counter2 ++ 3 bit counter1 → Counter1_R2 3 bit counter1 ++	
Register Read	Instr_R2(11-9) → RF_A1 RF_D1 → A_R3 //Value to store If Counter1_R2 == "000": Instr_R2(8-6) → RF_A2 RF_D2 → RefAdd	

	RefAdd → B_R3 //Mem loc to store Instr_R2(5-0) → SE6 → C_R3 //Offset	
Execute	A_R3 → A_R4 B_R3 → ALU2_A C_R3 → ALU2_B ALU2_C → B_R4	ALU2 → add Z_WR → 0 C_WR → 0
Memory Access	B_R4 → Mem2_Add A_R4 → Mem2_Data	Mem2_RD → 0 ControlSig_R4(Mem2_WR) → Mem2_WR
Write Back	NIL	RF_WR → 0

BEQ(10_00_RA_RB_6bitImm)

Instruction Fetch	PC → Mem1_add, PC_R1, ALU1_A +2 → ALU1_B ALU1_C → PC Mem1_D → Instr_R1	PC_WR ALU1 → ADD
Instruction Decode	PR1 → PR2	
Register Read	Instr_R2(11-9) → RF_A1 Instr_R2(8-6) → RF_A2 RF_D1 → A_R3 RF_D2 → B_R3 PC_R2 → PC_R3 Instr_R2 → Instr_R3	

Execute	A_R3 → ALU2_A
	B_R3 → ALU2_B
	if(ALU_Z == '1'){
	   // HAZARD
	   PC_R3 → ALU3_A
	   Instr_R3(5-0) → bit append → SE16 → ALU3_B
	   ALU3_C → PC
	}
		ALU_2 → SUB
		PC_WR ← ALU_Z

Memory Access	NIL	
Write Back	NIL	

BLT(10__01__RA__RB__6bitImm)

Instruction Fetch	PC → Mem1__add, PC__R1, ALU1__A +2 → ALU1__B ALU1__C → PC Mem1__D → Instr__R1	PC__WR
Instruction Decode	PR1 → PR2	
Register Read	Instr__R2(11-9) → RF__A1 Instr__R2(8-6) → RF__A2 RF__D1 → A__R3 RF__D2 → B__R3 PC__R2 → PC__R3 Instr__R2 → Instr__R3	

Execute	A__R3 → ALU2__A
	B__R3 → ALU2__B
	if(ALU__C){
	  // HAZARD
	  PC__R3 → ALU3__A
	  Instr__R3(5-0) → bit append → SE16 → ALU3__B
	  ALU3__C → PC
	}
		ALU__2 → SUB
		PC__WR ←
		ALU__C

Memory Access	NIL	
Write Back	NIL	

BLE(10__10__RA__RB__6bitImm)

Instruction Fetch	PC → Mem1__add, PC__R1, ALU1__A +2 → ALU1__B ALU1__C → PC Mem1__D → Instr__R1	PC__WR
Instruction Decode	PR1 → PR2	
Register Read	Instr__R2(11-9) → RF__A1 Instr__R2(8-6) → RF__A2 RF__D1 → A__R3 RF__D2 → B__R3 PC__R2 → PC__R3 Instr__R2 → Instr__R3	

Execute	A__R3 → ALU2__A
	B__R3 → ALU2__B
	if(ALU__Z == ‘1’ or ALU__C == ‘1’){
	// HAZARD
	PC__R3 → ALU3__A
	Instr__R3(5-0) → bit append → SE16 → ALU3__B
	ALU3__C → PC
	}
		ALU__2 → SUB
		
		PC__WR ← (ALU__Z || ALU__C)

Memory Access	NIL	
Write Back	NIL	

JAL(11__00__RA__9bitImm)

(HAZARD)

Instruction Fetch	PC → Mem1__add, PC__R1, ALU1__A +2 → ALU1__B ALU1__C → PC, A__R1 Mem1__D → Instr__R1	PC__WR
Instruction Decode	Instr__R1 → Instr__R2 PC__R1 → PC__R2 A__R1 → A__R2	
Register Read	PC__R2 → PC__R3 Instr__R2 → Instr__R3 A__R2 → A__R3	
Execute	PC__R3 → ALU3__A Instr__R3(8-0) → bit append → SE16 → ALU3__B ALU3__C → PC A__R3 → A__R4 Instr__R3 → Instr__R4	ALU3 → ADD PC__WR ← 1
Memory Access	A__R4 → A__R5 Instr__R4 → Instr__R5	
Write Back	Instr__R5(11-9) → RF__A3 A__R5 → RF__D3	

JLR(11_01_RA_RB_000_000)

(HAZARD)

Instruction Fetch	PC → Mem1_add, ALU1_A +2 → ALU1_B ALU1_C → PC, A_R1 Mem1_D → Instr_R1	PC_WR
Instruction Decode	Instr_R1 → Instr_R2 A_R1 → A_R2	
Register Read	Instr_R2(8-6) → RF_A1 RF_D1 → PC Instr_R2 → Instr_R3 A_R2 → A_R3	PC_WR
Execute	Instr_R3 → Instr_R4 A_R3 → A_R4	
Memory Access	Instr_R4 → Instr_R5 A_R4 → A_R5	
Write Back	Instr_R5(11-9) → RF_A3 A_R5 → RF_D3	

JRI(11_11_RA_9bitImm)

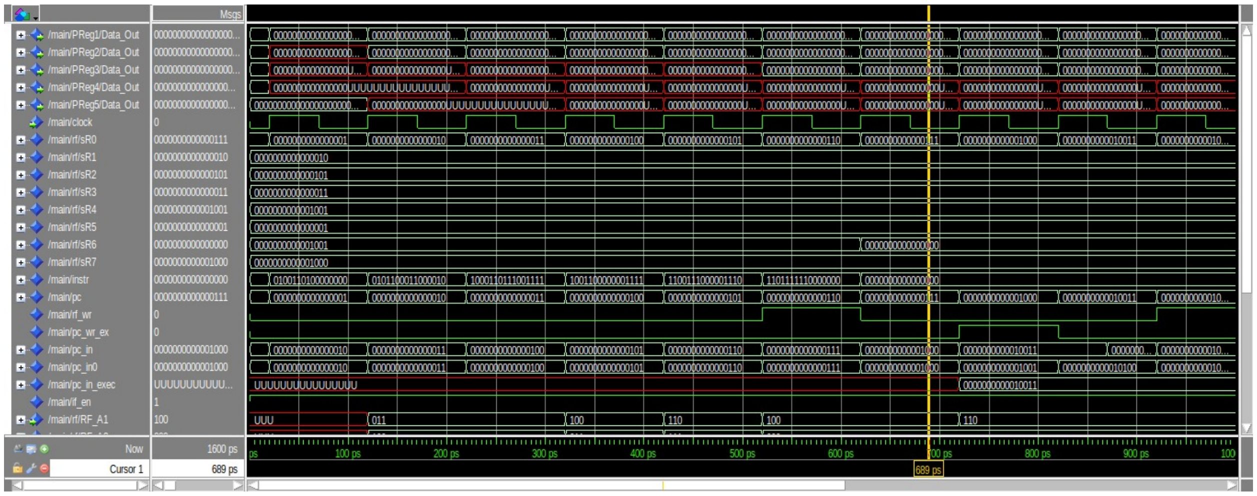
(HAZARD)

Instruction Fetch	PC → Mem1_add, ALU1_A +2 → ALU1_B ALU1_C → PC Mem1_D → Instr_R1	PC_WR ALU1 → ADD
Instruction Decode	Instr_R1 → Instr_R2 PC_R1 → PC_R2	
Register Read	Instr_R2(11-9) → RF_A1 RF_D1 → A_R3 Instr_R2 → Instr_R3	
Execute	A_R3 → ALU3_A Instr_R3(8-0) → bit append → SE16 → ALU3_B ALU3_C → PC	PC_WR

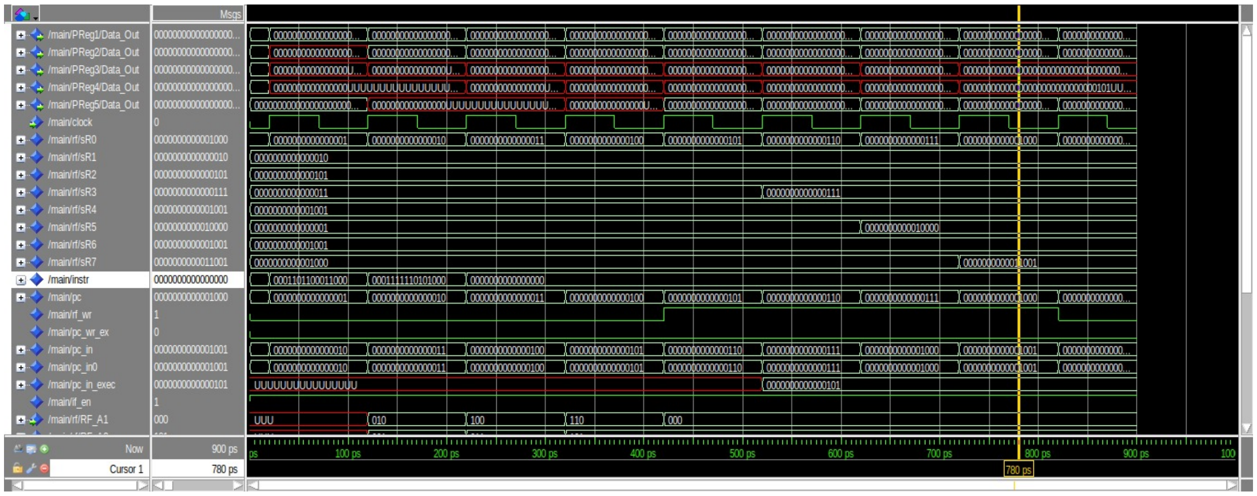
Memory Access	NIL	
Write Back	NIL	

RTL Simulation

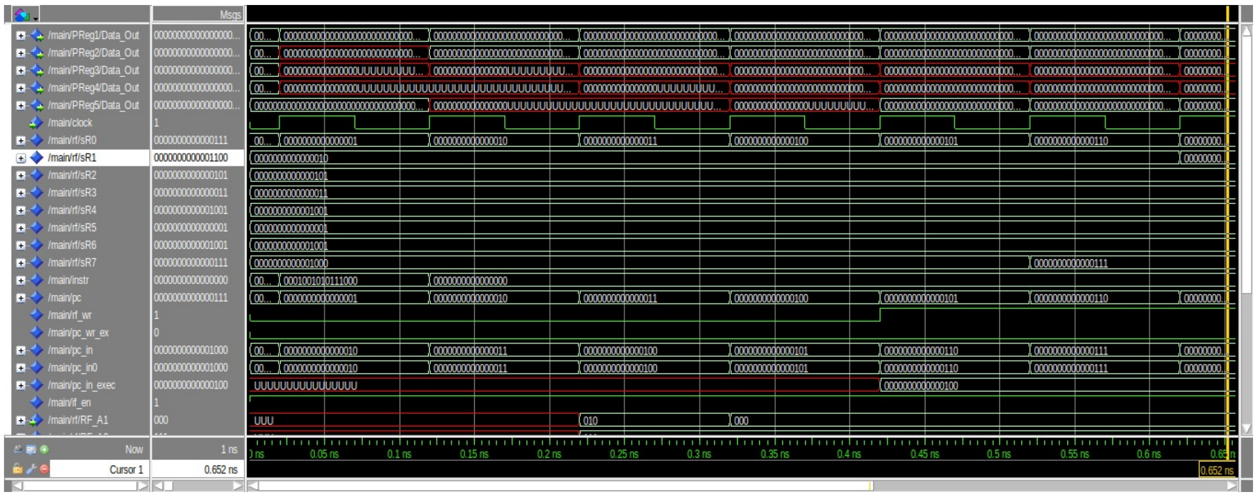
- Instructions without hazards:



- Arithmetic and logical instructions with hazards:



- Load instruction with hazards:



```
                              Msgs
/main/PReg1/Data_Out    0000000000000000...
/main/PReg2/Data_Out    0000000000000000...
/main/PReg3/Data_Out    0000000000000000...
/main/PReg4/Data_Out    0000000000000000...
/main/PReg5/Data_Out    0000000000000000...
/main/clock             1
/main/rf/sR0            0000000000000111
/main/rf/sR1            0000000000001100
/main/rf/sR2            0000000000000101
/main/rf/sR3            0000000000000011
/main/rf/sR4            0000000000001001
/main/rf/sR5            0000000000000001
/main/rf/sR6            0000000000001001
/main/rf/sR7            0000000000000111
/main/instr             0000000000000000
/main/pc                0000000000000111
/main/rf_wr             1
/main/pc_wr_ex          0
/main/pc_in             0000000000001000
/main/pc_in0            0000000000001000
/main/pc_in_exec        0000000000000100
/main/if_en             1
/main/rf/RF_A1          000
                  Now   1 ns
             Cursor 1   0.652 ns
```

References

- 6 stage pipeline: https://www.javatpoint.com/instruction-pipeline
- EE 739 Course Material : https://www.ee.iitb.ac.in/~viren/Courses/2015/EE739.htm
- EE 309 Course Material
- D.A. Patterson and J.L. Hennessy, Computer Organization and Design, Morgan Kaufman Pub., N. Delhi, 2005