IITB RISC 24

Group ID: #54

S. No.	Name	Roll No.
1	Aman Verma	22B3929
2	Praveen	22b3931
3	Rahul Agarwal	22B3961

Work distribution:

- **Rahul Agarwal:** ALU, 16 bit adder, Register file, temporary register, Hazard detector, Write back stage, memory access stage, pipeline register 5, report
- **Aman Verma:** Branch predictor, branch MUX, Instruction fetch stage, pipeline register 1, pipeline register 3, Register file read, Pipeline controller, report
- **Praveen:** LM-SM Controller, Instruction memory, data memory, dependency MUX, Instruction decode stage, execution stage, pipeline register 2, pipeline register 4, report

Introduction

IITB-RISC features a 6-stage pipelined processor, operating as a **16-bit computer** with **8 general-purpose registers** labeled **R0 to R7**, with **R0** dedicated to storing the **Program Counter (PC)**. The architecture supports predicated instruction execution and enables multiple load and store operations. It utilizes three machine-code instruction formats: **R, I, and J types**, accommodating a total of 14 instructions.

The **6 stages of the pipeline** include :

1. Instruction fetch
2. Instruction decode
3. Register read
4. Execute
5. Memory access
6. Write back

Hazard Mitigation

The architecture is optimized for performance, and thus includes hazard mitigation techniques. The hazards and their respective mitigation techniques used are given below:

- **Dependency hazard:**
  Dependency hazard occurs when a latter instruction uses a register as source operand, and that register happens to be modified in an earlier instruction and the modified value of the register has not been written back. For the mitigation of this hazard, we have taken use of 2 units namely **Hazard detector** and **Dependency MUX**.

  The hazard detector takes in the value of the **address of the register** that is yet to be written back, and the register, which is currently being used as source operands and on comparing them gives control signal to the dependency MUX.

  The dependency MUX takes in multiple inputs from the later stages and also the register file, and the control signals are taken in from the hazard detector.

- **Branching hazards:**
  For a branching instruction, for conditional branches, we are taking use of a **Branch Predictor unit**, which essentially stores all the addresses where branching instruction happened previously, and a history bit, which specifies if the branching occurred the previous time.

  For unconditional branches(jumps), it again adds the address to the branch predictor, but rather keeping history bit to be '1' this time.

- **General purpose register, R_o:**
  As mentioned in the problem statement, the register R_o stores the value of PC in it always. Now instructions like ADC, LM, LW etc. can modify R_o due to the reason that for the programmer, R_o is similar to any other general purpose register. But if it is actually modified, PC will be lost.

  To mitigate this, we are making a temporary register, which anytime R_o is modified, will take the value to be written in it, and the actual value in R_o is not changed by any non-branching instructions.
  Similarly whenever the value in R_o is required, we will take the data from this temporary register.

- **LM-SM Controller:**
  The LM and SM instruction asks to load or store multiple registers and memory locations. Naturally, multiple

values cannot be loaded or stored in a single cycle, and thus we used LM-SM controller to solve this problem.

The controller counts the number of set bits in the operand, and writes values of corresponding register and memory addresses in two separate arrays. It then halts the previous stages for until the counter value becomes zero again and all loading/storing is completed.

Datapath

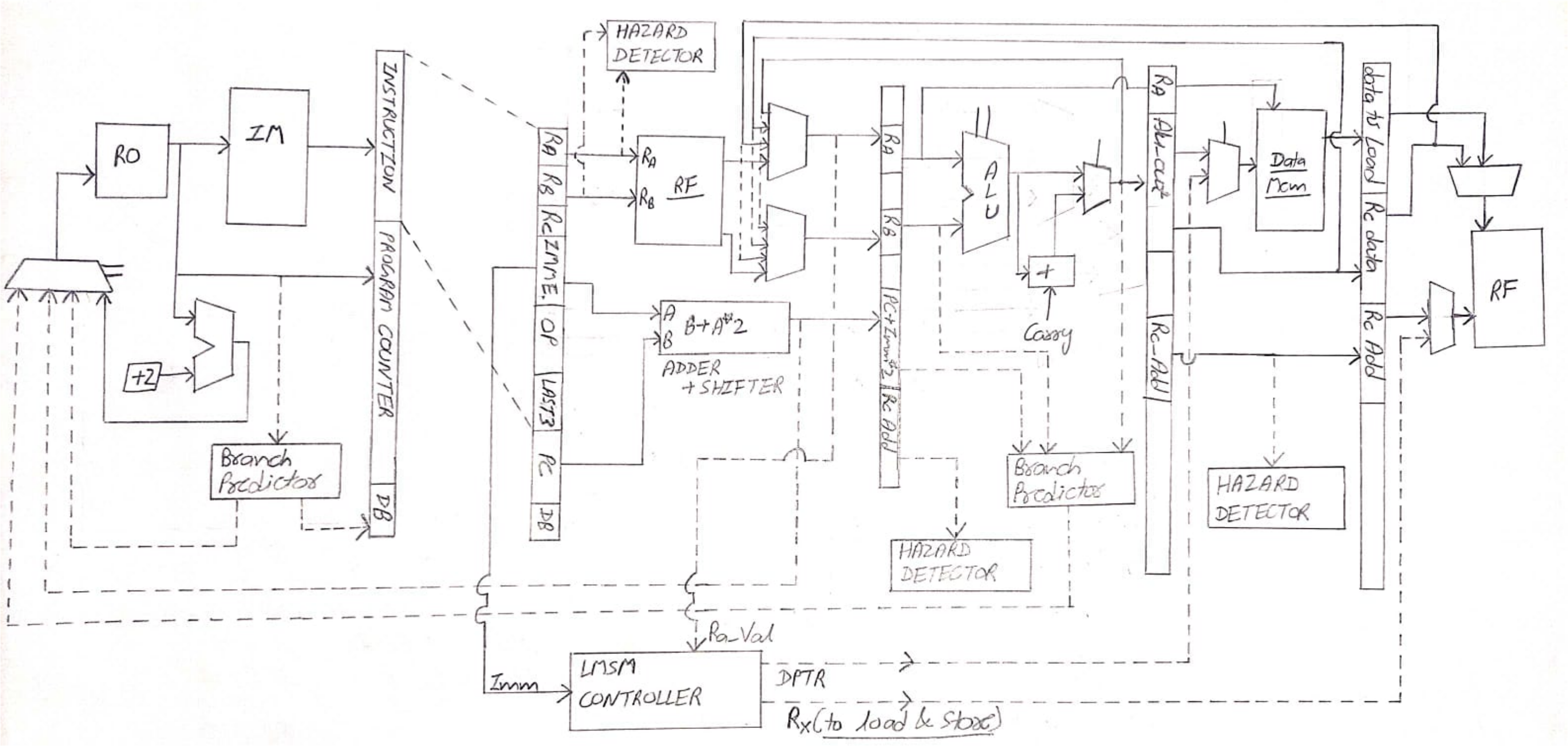


Figure 1: Datapath

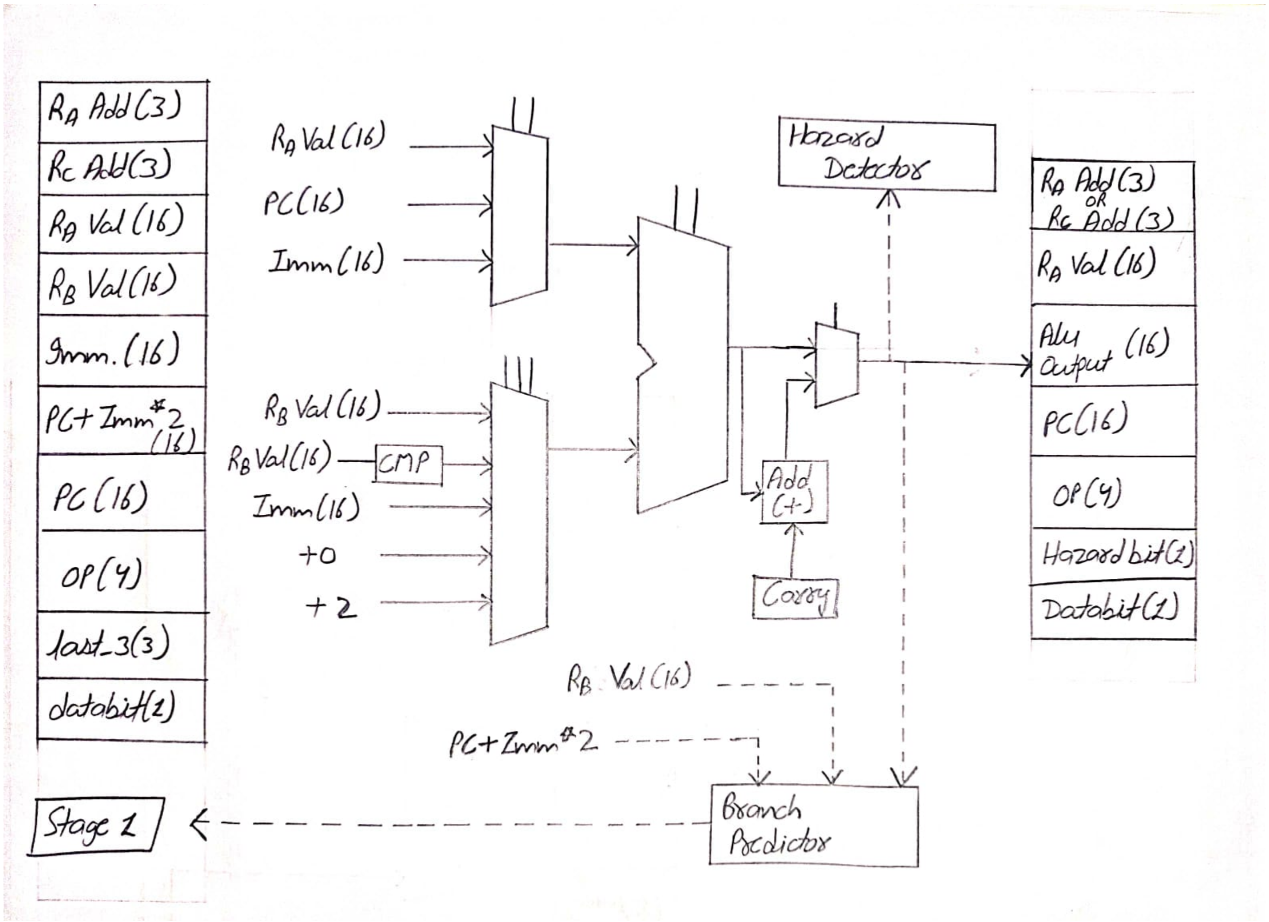
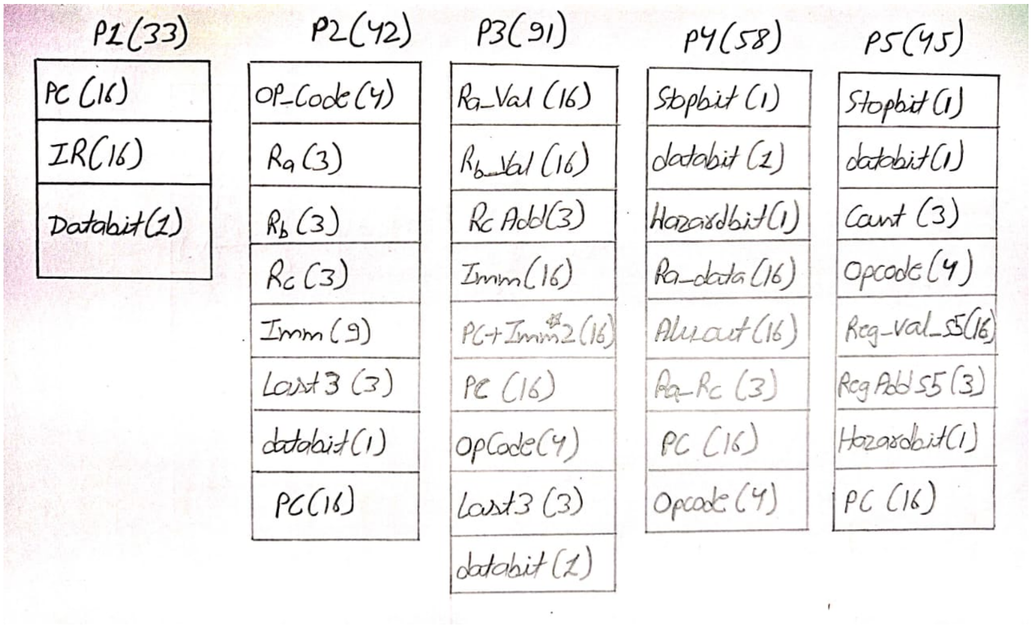


Figure 2: Execute stage



P1(33)	P2(42)	P3(91)	P4(58)	P5(45)
PC (16)	OP_Code(4)	Ra_Val (16)	Stopbit (1)	Stopbit (1)
IR(16)	Ra (3)	Rb_Val (16)	databit (1)	databit(1)
Databit(1)	Rb (3)	Rc Add(3)	Hazardbit(1)	Count (3)
	Rc (3)	Imm(16)	Ra_data (16)	Opcode (4)
	Imm (9)	PC+Imm*2 (16)	Alu_out (16)	Reg_val_S5(16)
	Last3 (3)	PC (16)	Ra_Rc (3)	Reg Add S5 (3)
	databit(1)	OpCode(4)	PC (16)	Hazardbit(1)
	PC(16)	Last3 (3)	Opcode (4)	PC (16)
		databit (1)		

Figure 3: Pipeline registers

The six stages

As mentioned above, there are a total of six stages. The inputs and outputs of each stage is described below:

1. **Instruction fetch:**

   - An adder is present to do PC+2
   - Branch predictor and branch MUX
   - Instruction memory

   By default the writing control signals are off. Program counter points to the instruction memory and takes out the instruction register and sends it off to Pipeline register 1. Through the branch MUX, the PC is updated. This PC also points to the branch predictor and checks if the PC is present in the database or not.

2. **Instruction decode:**
   Immediate value goes into the LMSM controller based on the opcode of LM and SM.

3. **Register Read:**

   - Dependency MUX
   - Register file
   - 16 bit adder

   The register address is passed into the hazard detector. Value of R_a is passed into the LMSM controller from the 16 bit adder.
   The 16 bit adder has sign extender as well as 2x multiplier which we use to calculate PC + Imm*2.

4. **Execute:**

   - ALU
   - R_a and PC+Imm*2 are given to branch predictor
   - Value of register address is given to hazard detector

   On the basis of databit and stop signal, input is given into the branch predictor. Stop signal is decided on the basis of the condition which is checked by the branch predictor itself.

5. **Memory Access:**

   - Data memory
   - Value of register address is given to hazard detector.

   Takes memory address from data pointer in case of LM and SM instruction.

6. **Write Back:**

   - Value of register address is given to hazard detector

   Gives the value to be written in registers to the register file.

Instructions

There are a total of 14 instructions and the data path and cycle flow of each instruction is described in detail below:

1. **ADA, ADC, ADZ, AWC, ACA, ACC, ACZ, ACW**
   (OP CODE = 0001)

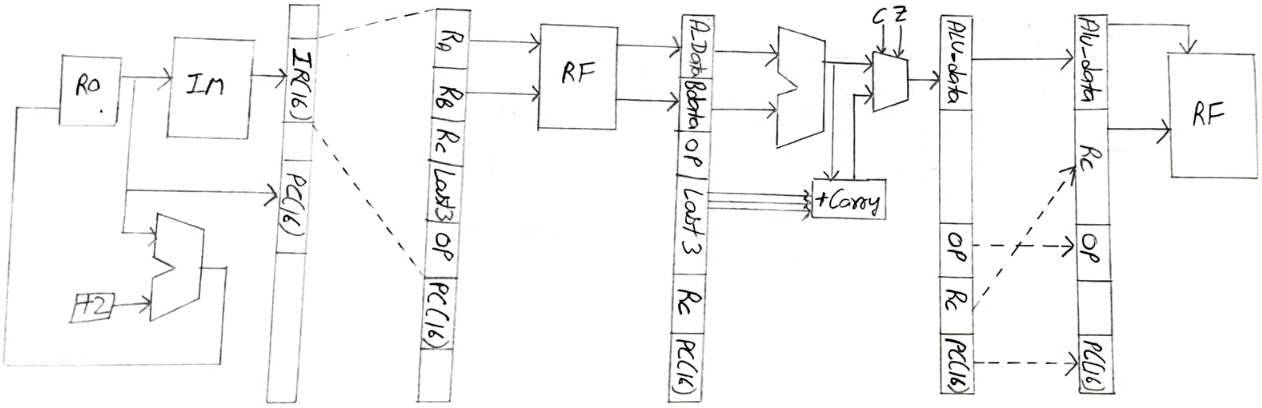


Figure 4: ADD and NAND instructions

2. **NDU, NDC, NDZ, NCU, NCC, NCZ**
   (OP CODE = 0000)

3. **ADI**
   (OP CODE = 0010)

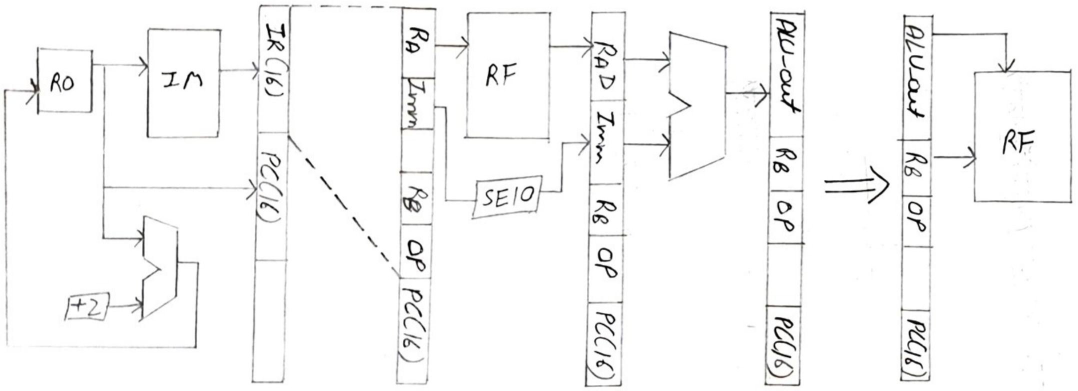


Figure 5: ADI instruction

4. **LLI**
   (OP CODE = 0011)

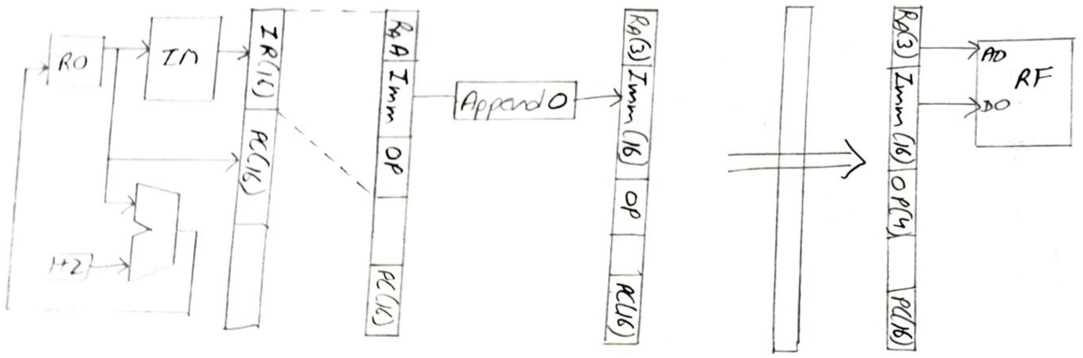


Figure 6: LLI instruction

5. **LW**
   (OP CODE = 0100)

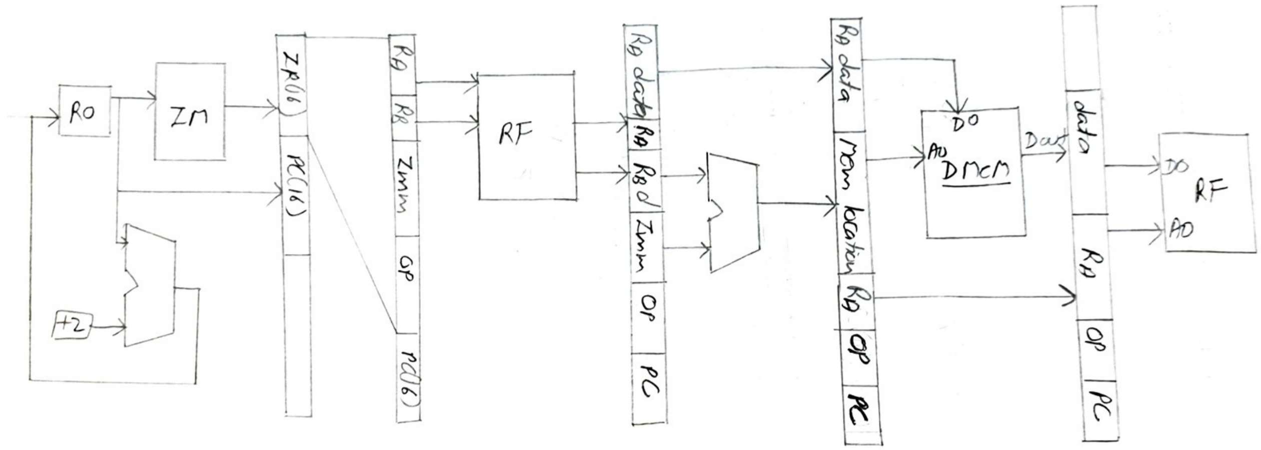


Figure 7: LW and SW instructions

6. **SW**
   (OP CODE = 0101)

7. **LM**
(OP CODE = 0110)

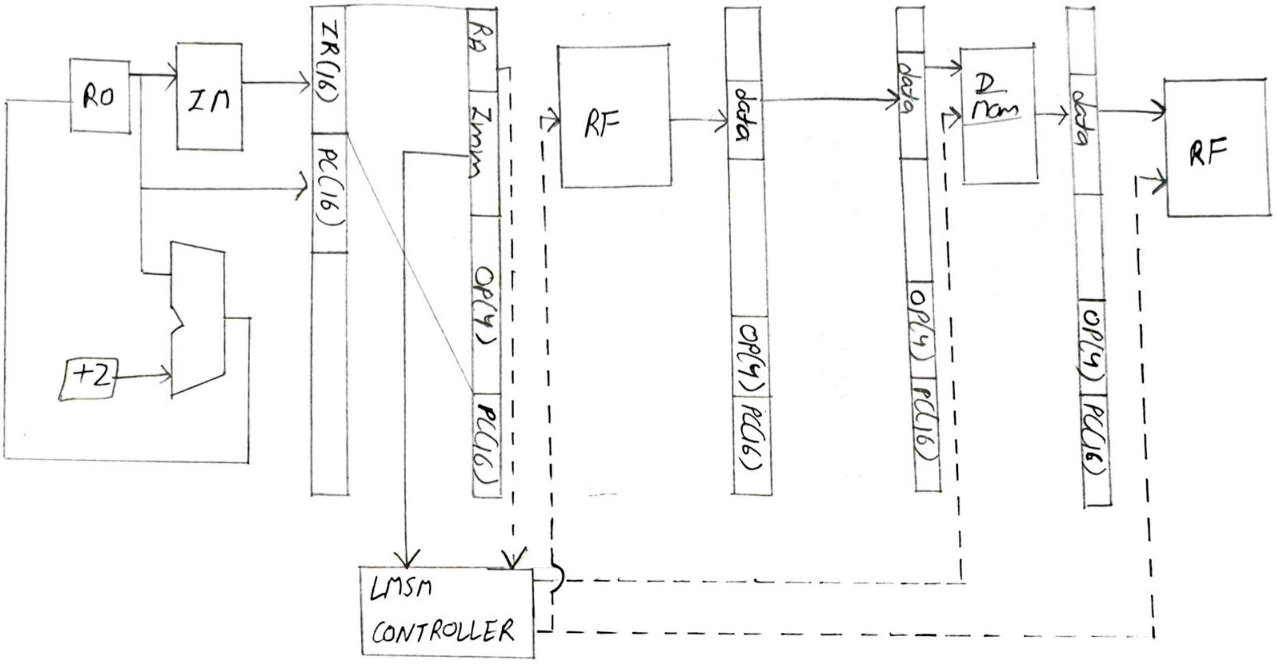


Figure 8: LM and SM instructions

8. **SM**
(OP CODE = 0111)

9. **BEQ**
   (OP CODE = 1000)

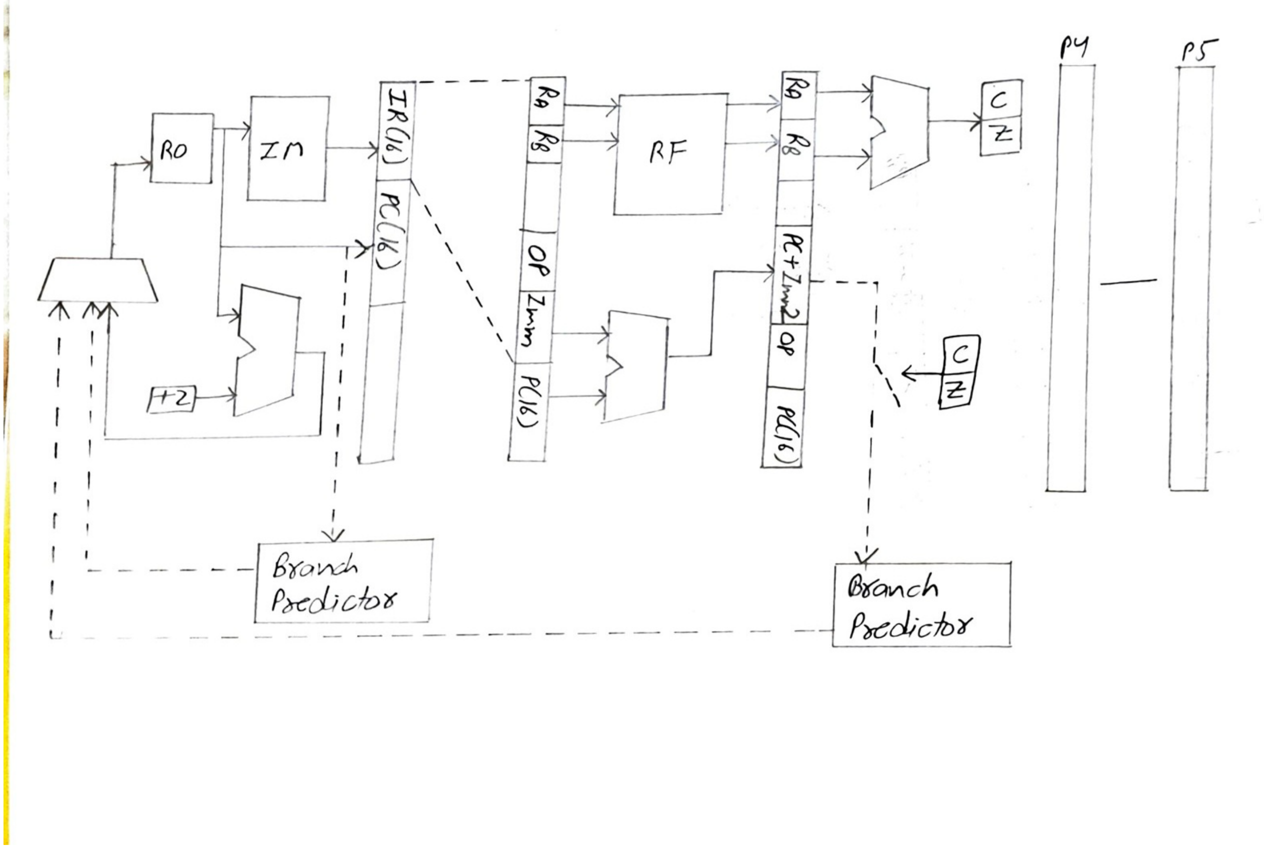


Figure 9: BEQ, BLT and BLE instructions

10. **BLT**
    (OP CODE = 1001)

11. **BLE**
    (OP CODE = 1010)

12. **JAL**

(OP CODE = 1100)

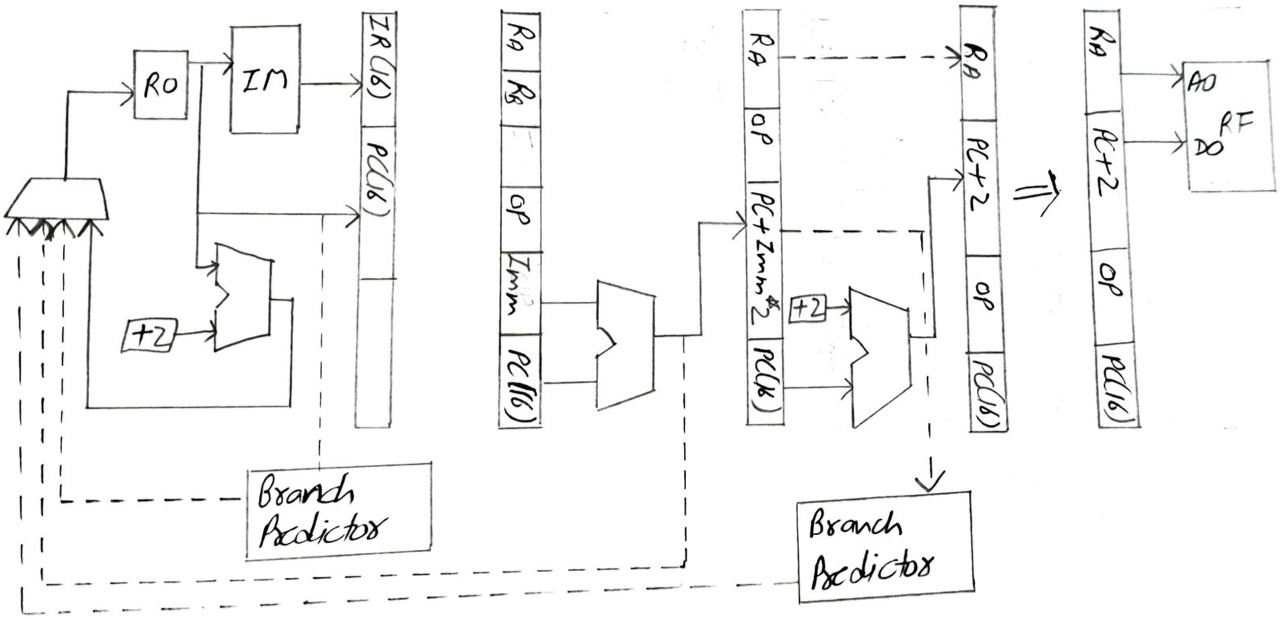


Figure 10: JAL instruction

13. **JLR**

(OP CODE = 1101)

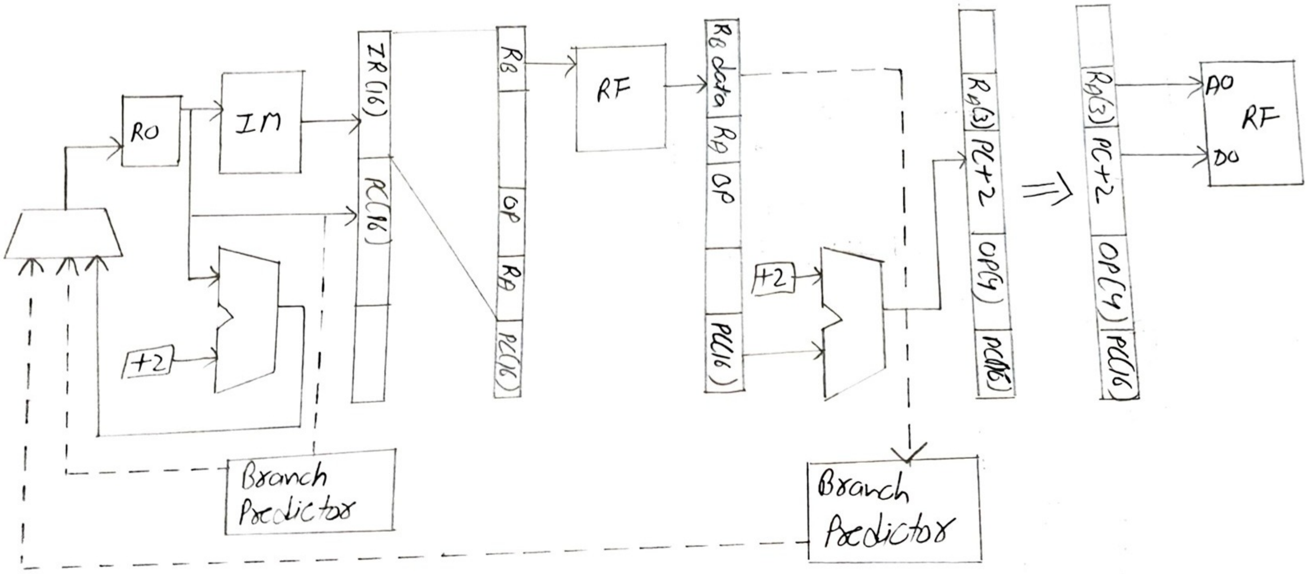


Figure 11: JLR instruction

14. **JRI**
(OP CODE = 1111)

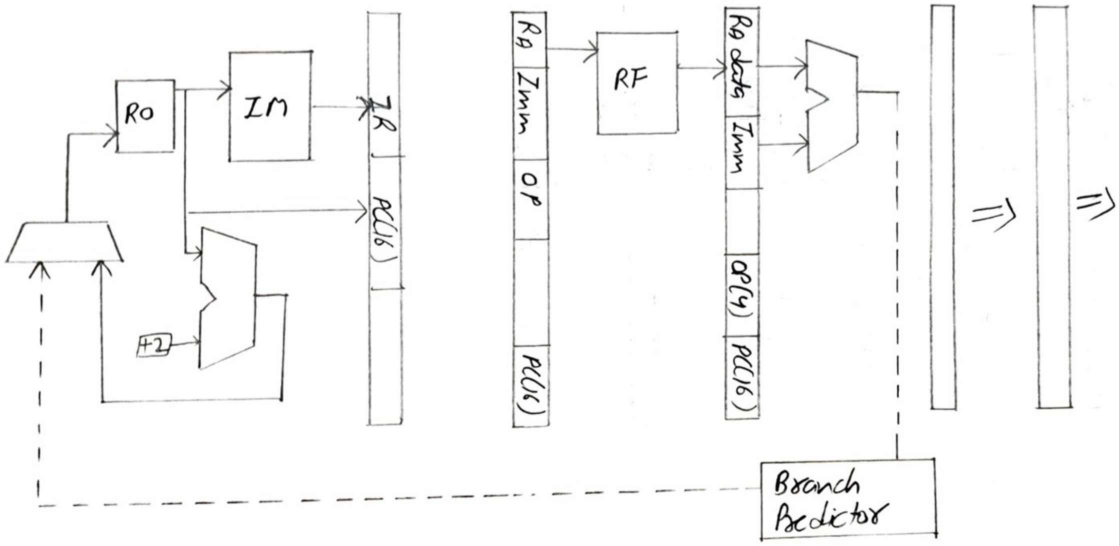


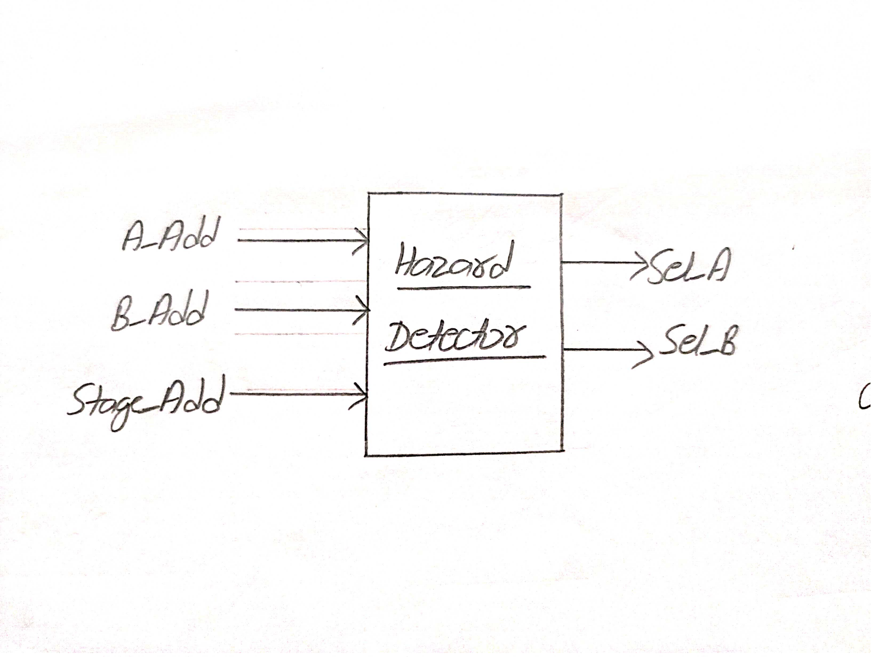
Figure 12: JRI instruction

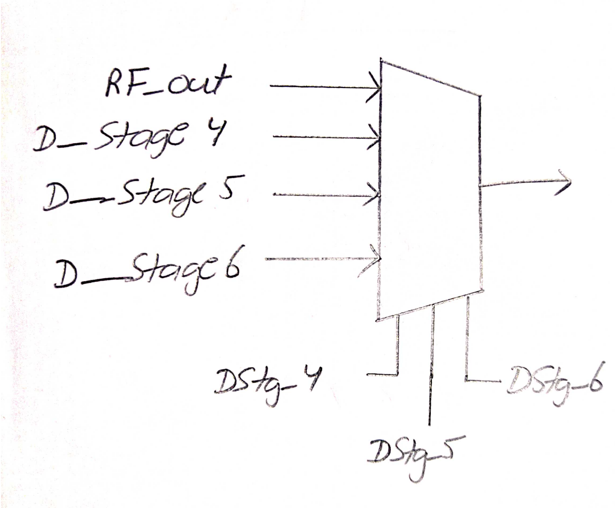
Innovations and Control blocks

This section includes detailed description of all the additional blocks and controllers we have added to the architecture:

1. **Hazard Unit and dependency MUX:**

   The hazard unit is a block that deals with the dependency hazards. The hazard unit takes in addresses of register A and B constantly, a hazard bit and address of another register associated with register dependency.





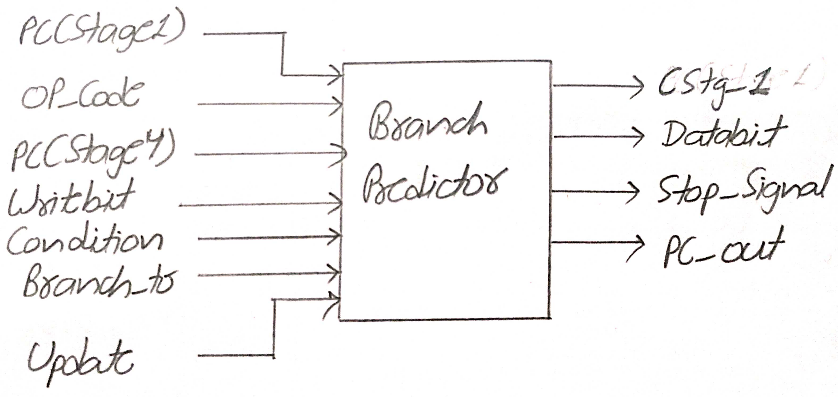
- The hazard unit then compares the address of register that is yet to be modified, and is present in one of the execute, memory read or write back stage, and register A and B that are to be passed on in the pipeline register between register read and execute stage.
- If the register addresses don't match, then program runs in normal way, and the value stored in register A and B are passed on as it is.
- But if the addresses match, the hazard unit sends modified control signals to the dependency MUX.
- The dependency mux has 4 inputs, one from the register file directly, and 3 from the upcoming 3 stages. The hazard unit, upon seeing from which stage the address came for comparison modifies the control

signal, and lets that value pass on instead of the register file value and thus clearing the dependency hazard.

- If address comparisons come from multiple stages, then the hazard unit will take only that one which is nearest to register read stage, as that will correspond to the latest instruction that modifies the register.

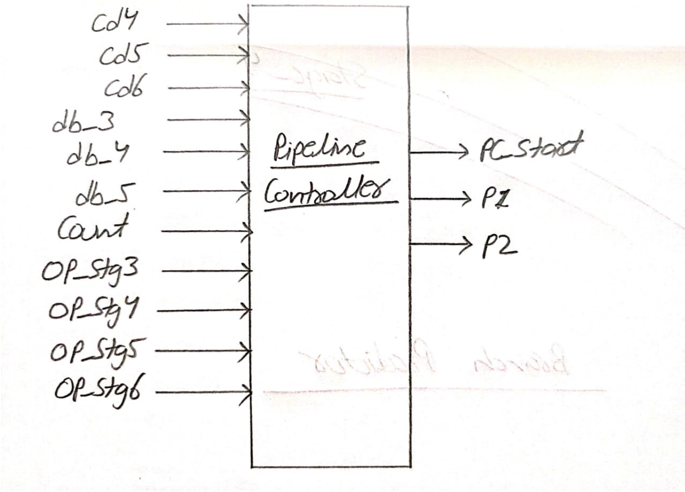
2. **Branch predictor:**

For every branching hazard, without a branch predictor, we lost 3 cycles of operation for each taken branch. For programs with looping instructions, losing such a large number of cycles would cost with performance of the system. Thus we made a branch predictor which essentially predicts if a particular branching instruction would end up taking the branch or not.



- The branch predictor is a table that stores a the PC of the branching instruction, b. the address where the branching would be directed to, and c. a history bit. The table would be initially empty.
- As a branching or jump instruction would come up, as we would know by its opcode. It would search for the PC in the table. If found in the table, and the history bit corresponding to it is 1, it would bring the next instruction present on the address corresponding to the PC.
- If not found in the table, it would forward a write-bit forward through the pipeline registers. The write-bit would be helpful in the execute stage where we would fill up the table with this PC and the corresponding address.
- For jump type instructions, the history bit always remains 1, and the process of adding the instruction to the table is same as that of branching instructions.

3. **Pipeline controller:** Pipeline controller controls stages PC-start, P1 and P2. It sets the stage execution signals, based on the opcode of $(LM, SM, JAL, LW)_{stage3}$, $(BEQ, BLT, BLE, JRI, JLR)_{Stage4}$, and by default, all the stage execution signals are 1.

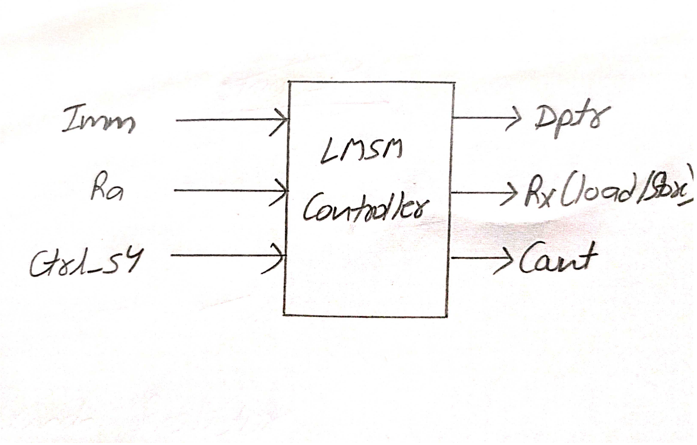


Based on the instruction's databit, condition bit and count, this sets the stage execution signal for stage 1, 2 and 3. For JAL instruction, if the PC + IMM*2 was present was present in the branch predictor, then that databit is passed on to the further stages to control the stage execution of register read and instruction decode stage and opens each stage one by one, until JAL instruction reaches the 5th stage.

For LM-SM, on the basis of value of count given by the LM-SM controller, if the count is non zero, when the instruction reaches 3rd stage, it stops the instruction fetch and instruction decode stage, until the count reaches zero again.

For BEQ, BLT, BLE, JRI, JLR, if the corresponding PC is not present in the database, or if the condition is not met, it stops the previous 3 stages, and starts execution of each stage one by one until the instruction reaches the 6th stage.

4. **LM-SM Block:** As, LM or SM instruction is detected in Instruction decode stage, the immediate is sent to the block. It counts the number of 1's and stores in temp-count. While computing the temp-count value, it also stores the register addresses in an array. When the instruction reaches the execute stage, a control signal is high, and this sends the memory address pointers to memory addresses and register addresses to register file for loading and storing.



5. **Adder 16 Block:** The adder 16 is used in stage 1 to make PC to PC +2 and in stage 3 to make PC + Imm*2.

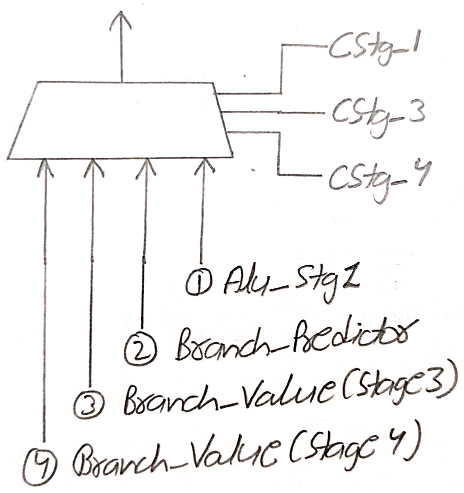
6. **Temporary register:** The temporary register stores the data and gives output the same data upon reading. When the write control signal is one, it allows to write into the register. Writing is synchronous and reading is asynchronous. It is used for storing value of R_o, when user tries to modify it using some non-branching instruction.

7. **ALU:** We are performing 3 operations with the help of ALU:

   - Adding A and B
   - Adding A,B and carry
   - NAND A and B
   - NAND of A,B and carry

8. **Branch MUX:** Branch MUX takes in input from 4 places:

   - from 16 bit adder in stage 1 (PC+2)
   - from branch predictor (If a particular PC is found in the branch predictor, and its history bit is 1) $(Controlsignal = Cstg1)$
   - from register fetch stage (JAL) $(Controlsignal = Cstg3)$
   - from execute stage (BEQ, BLT, BLE, JLR, JRI),$(Controlsignal = Cstg4)$



It has control signals from respective stages and blocks. The highest priority is given to input from stage 4, then 3, then branch predictor and then from stage 1.

9. **Register file:** There are 8 registers in the register file, out of which,R_o stores the program counter.

10. **Data memory:** Single in - single out, 8 bit address.

11. **Instruction memory:** Single in - single out, 8 bit address. Writing is not allowed in this block.