IITB-RISC

Project Guide : Prof. Virendra Singh

Project Report for EE309: Microprocessors

Priyansh Jain - 210070063
Sajal Sachin Deolikar - 210070071
Ishant Landge - 210070034
Sannidhya Kaushal - 210070076

Contents

1 Problem Statement

Project

Design a 6-stage pipelined processor, *IITB-RISC-23*, whose instruction set architecture is provided. *IITB-RISC* is a 16-bit very simple computer developed for the teaching that is based on the Little Computer Architecture. The *IITB-RISC-23* is a 16-bit computer system with 8 registers. It should follow the standard 6 stage pipelines (Instruction fetch, instruction decode, register read, execute, memory access, and write back). The architecture should be optimized for performance, i.e., should include hazard mitigation techniques. Hence, it should have implemented forwarding mechanism. Implementation of branch predictor is optional.

IITB-RISC Instruction Set Architecture

IITB-RISC is a 16-bit very simple computer developed for the teaching that is based on the Little Computer Architecture. The *IITB-RISC-23* is an 8-register, 16-bit computer system. It has 8 general-purpose registers (R0 to R7). Register R0 is always stores Program Counter. All addresses are byte addresses and instructions. Always it fetches two bytes for instruction and data. This architecture uses condition code register which has two flags Carry flag (*C*) and Zero flag (*Z*). The *IITB-RISC-23* is very simple, but it is general enough to solve complex problems. The architecture allows predicated instruction execution and multiple load and store execution. There are three machine-code instruction formats (R, I, and J type) and a total of 14 instructions. They are illustrated in the figure below.

R Type Instruction format

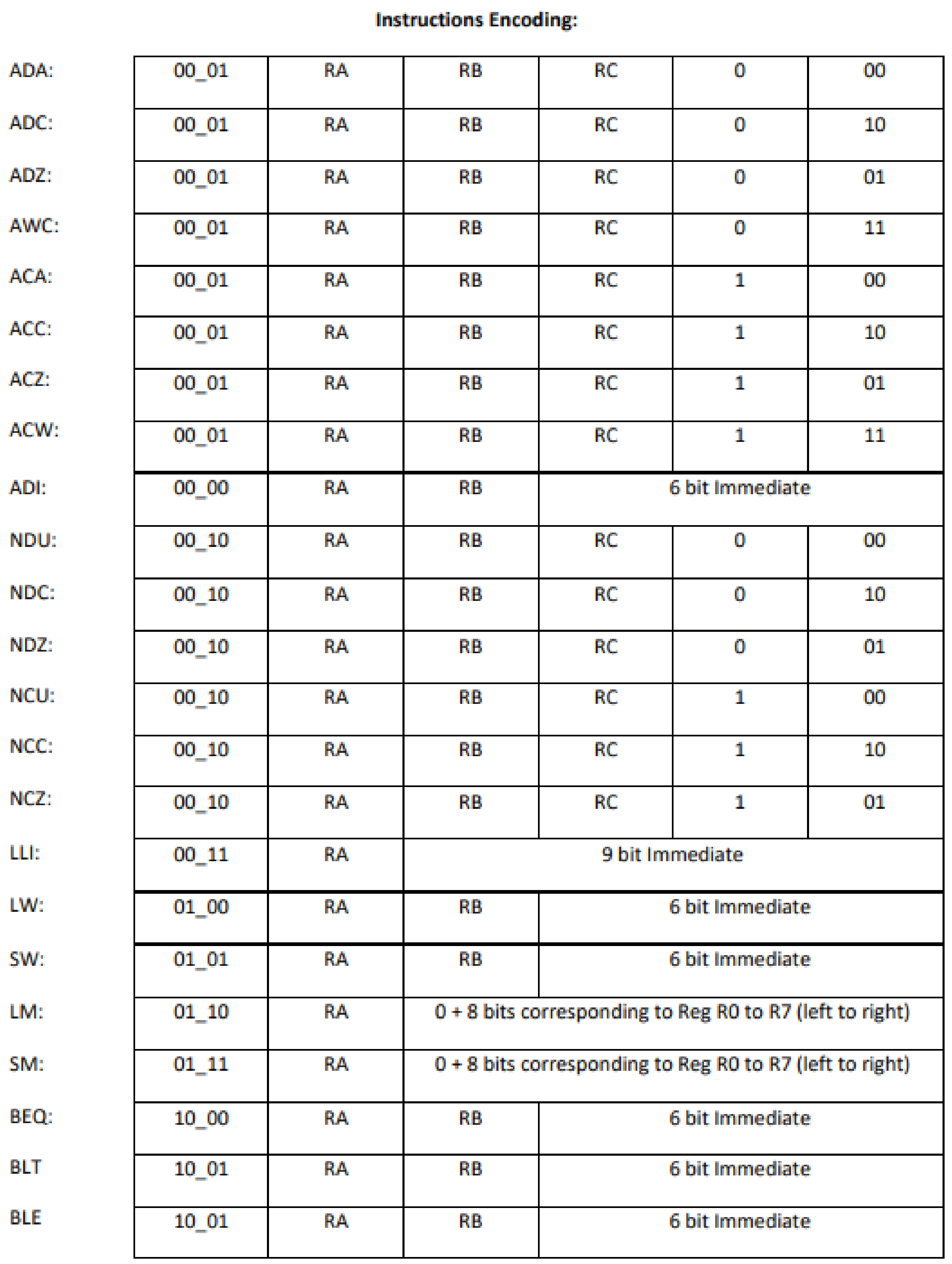
Opcode (4 bit)	Register A (RA) (3 bit)	Register B (RB) (3-bit)	Register C (RC) (3-bit)	Comple -ment (1 bit)	Condition (CZ) (2 bit)

I Type Instruction format

Opcode (4 bit)	Register A (RA) (3 bit)	Register C (RC) (3-bit)	Immediate (6 bits signed)

J Type Instruction format

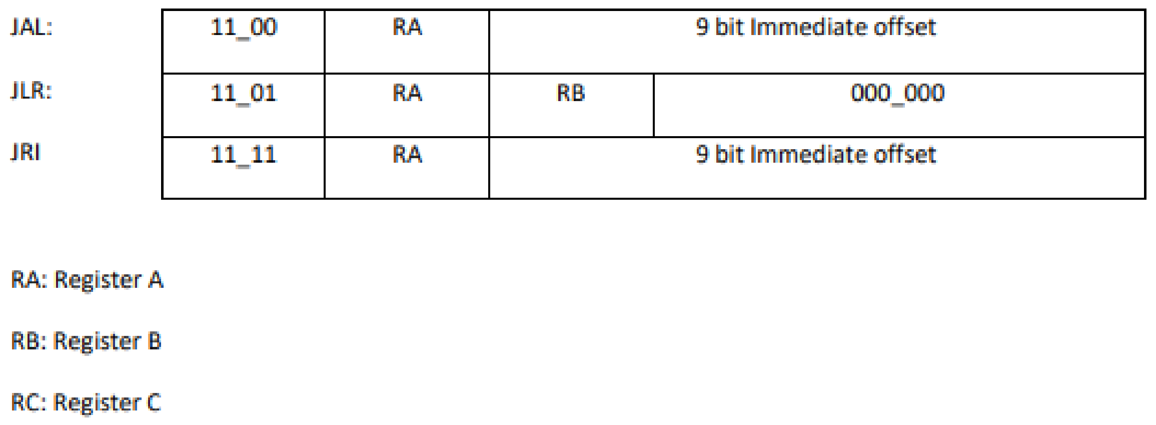
Opcode (4 bit)	Register A (RA) (3 bit)	Immediate (9 bits signed)



Instructions Encoding:

ADA:	00_01	RA	RB	RC	0	00
ADC:	00_01	RA	RB	RC	0	10
ADZ:	00_01	RA	RB	RC	0	01
AWC:	00_01	RA	RB	RC	0	11
ACA:	00_01	RA	RB	RC	1	00
ACC:	00_01	RA	RB	RC	1	10
ACZ:	00_01	RA	RB	RC	1	01
ACW:	00_01	RA	RB	RC	1	11
ADI:	00_00	RA	RB	6 bit Immediate		
NDU:	00_10	RA	RB	RC	0	00
NDC:	00_10	RA	RB	RC	0	10
NDZ:	00_10	RA	RB	RC	0	01
NCU:	00_10	RA	RB	RC	1	00
NCC:	00_10	RA	RB	RC	1	10
NCZ:	00_10	RA	RB	RC	1	01
LLI:	00_11	RA	9 bit Immediate			
LW:	01_00	RA	RB	6 bit Immediate		
SW:	01_01	RA	RB	6 bit Immediate		
LM:	01_10	RA	0 + 8 bits corresponding to Reg R0 to R7 (left to right)			
SM:	01_11	RA	0 + 8 bits corresponding to Reg R0 to R7 (left to right)			
BEQ:	10_00	RA	RB	6 bit Immediate		
BLT	10_01	RA	RB	6 bit Immediate		
BLE	10_01	RA	RB	6 bit Immediate		

NOTE: *opcode for BLE is taken as 1010.*



JAL:	11_00	RA	9 bit Immediate offset	
JLR:	11_01	RA	RB	000_000
JRI	11_11	RA	9 bit Immediate offset	

RA: Register A

RB: Register B

RC: Register C

Instruction Description

Mnemonic	Name & Format	Assembly	Action
ADA	ADD (R)	*ada rc, ra, rb*	Add content of regB to regA and store result in regC. *It modifies C and Z flags*
ADC	Add if carry set (R)	*adc rc, ra, rb*	Add content of regB to regA and store result in regC, if carry flaf is set. *It modifies C & Z flags*
ADZ	Add if zero set (R)	*adz rc, ra, rb*	Add content of regB to regA and store result in regC, if zero flag is set. *It modifies C & Z flags*
AWC	Add with carry (R)	*awc rc,ra,rb*	Add content of regA to regB and Carry and store result in regC regC = regA + regB + Carry *It modifies C & Z flags*
ACA	ADD (R)	*aca rc, ra, rb*	Add content of regA to complement of regA and store result in regC. *It modifies C and Z flags*
ACC	Add if carry set (R)	*acc rc, ra, rb*	Add content of regA to Complement of regB and store result in regC, if carry flag is set. *It modifies C & Z flags*
ACZ	Add if zero set (R)	*acz rc, ra, rb*	Add content of regA to Complement of regB and store result in regC, if zero flag is set. *It modifies C & Z flags*
ACW	Add with carry (R)	*acw rc,ra,rb*	Add content of regA to Complement of regB and Carry and store result in regC regC = regA + compement of regB + Carry *It modifies C & Z flags*

ADI	Add immediate (I)	*adi rb, ra, imm6*	Add content of regA with Imm (sign extended) and store result in regB. *It modifies C and Z flags*
NDU	Nand (R)	*ndu rc, ra, rb*	NAND the content of regA to regB and store result in regC. *It modifies Z flag*
NDC	Nand if carry set (R)	*ndc rc, ra, rb*	NAND the content of regA to regB and store result in regC if carry flag is set. *It modifies Z flag*
NDZ	Nand if zero set (R)	*ndz rc, ra, rb*	NAND the content of regB to regA and store result in regC if zero flag is set. *It modifies Z flag*
NCU	Nand (R)	*ncu rc, ra, rb*	NAND the content of regA to Complement of regB and store result in regC. *It modifies Z flag*
NCC	Nand if carry set (R)	*ncc rc, ra, rb*	NAND the content of regA to complement of regB and store result in regC if carry flag is set. *It modifies Z flag*
NCZ	Nand if zero set (R)	*ncz rc, ra, rb*	NAND the content of regA to complement of regB and store result in regC, if zero flag is set. *It modifies Z flag*
LLI	Load lower immediate (J)	*lli ra, Imm*	Place 9 bits immediate into leat significant 9 bits of register A (RA) and higher 7 bits are assigned to zero.
LW	Load (I)	*lw ra, rb, Imm*	Load value from memory into reg A. Memory address is formed by adding immediate 6 bits (signed) with content of red B.

			It modifies zero flag.
SW	Store (I)	*sw ra, rb, Imm*	Store value from reg A into memory. Memory address is formed by adding immediate 6 bits (signed) with content of red B.
LM	Load multiple (J)	*lw ra, Imm*	Load multiple registers whose address is given in the immediate field (one bit per register, R0 to R7 from left to right) in reverse order from right to left, i.e, registers from R7 to R0 if corresponding bit is set. Memory address is given in reg A. Registers which are expected to be loaded from consecutive memory addresses.
SM	Store multiple (J)	sm, ra, Imm	Store multiple registers whose address is given in the immediate field (one bit per register, R0 to R7 from left to right) in reverse order from right to left, i.e, registers from R7 to R0 if corresponding bit is set. Memory address is given in reg A. Registers which are expected to store must be stored to consecutive addresses.
BEQ	Branch on Equality (I)	beq ra, rb, Imm	If content of reg A and regB are the same, branch to PC+Imm*2, where PC is the address of beq instruction
BLT	Branch on Less Than (I)	blt ra, rb, Imm	If content of reg A is less than content of regB, then it branches to PC+Imm*2, where PC is the address of beq instruction
BLE	Branch on Less or Equal (I)	ble ra, rb, Imm	If content of reg A is less than or equal to the content of regB, then it branches to PC+Imm*2, where PC is the address of beq instruction
JAL	Jump and Link (J)	jalr ra, Imm	Branch to the address PC+ Imm*2. Store PC+2 into regA, where PC is the address of the jalr instruction
JLR	Jump and Link to Register (I)	jlr ra, rb	Branch to the address in regB. Store PC+2 into regA, where PC is the address of the jlr instruction
JRI	Jump to register (J)	jri ra, Imm	Branch to memory location given by the RA + Imm*2

2 Components

2.1 General Components

- **Multiplexers:** A multiplexer (or MUX) selects between several input signals and forwards the selected input to a single output line.
- **Extenders:** Extenders are used to extend the bit width of a signal to the desired length. We have used extenders to extend the immediate values of execution or write-back. This is done by padding the immediate value with the zeroes in the MSB to make it 16-bit data.
- **Sign Extenders:** Sign Extender operates in which the number of bits representing a specific value increases to 16 while preserving the original sign and value. A Sign Extender or a Sign Extension Unit is a black box representation of such an operation. In practice, this unit is seldom a unit of its own but rather part of a more complex unit.
- **Clock:** Produces a square wave clock signal with a constant frequency. This clock is in reset mode at the start for a finite time.
- **Register:** Registers are data storage devices with binary information.
- **Temporary Registers:** Used to hold temporary data results during an arithmetic and logic operation. This is also used for storing the PC values. These registers are not available to the programmer.
- **Pipeline register:** Pipeline registers are temporary storage units that hold data as it moves through the various processing stages in a CPU's pipeline architecture. In a 6-stage pipeline, there would typically be five pipeline registers, each holding data specific to its stage. The pipeline registers enable the CPU to perform multiple operations simultaneously without waiting for each instruction to complete before moving on to the next. Pipeline registers are a common technique used to improve the performance of modern CPUs.
- **Register File:** A register file is a means of memory storage within a computer's central processing unit (CPU). The computer's register files contain bits of data and mapping locations. These locations specify certain addresses that are input components of a register file. Other inputs include data, the read-write function and the execute function.
- **Instruction Memory:** The instruction memory is responsible for storing the program instructions and making them available to the fetch stage of the pipeline. It is accessed during the fetch stage of the pipeline, where it retrieves the instruction pointed to by the program counter (PC). The instruction is then stored in the instruction fetch register (IF) and passed down the pipeline for decoding and execution. The instruction Memory is an array of 2^{16} memory registers, each of size 1 byte. In each fetch, the Instruction Memory gives 2 bytes of data for the instruction.
- **Data Memory:** Data memory is responsible for storing data used by the program instructions. It is accessed during the pipeline's memory access or write-back stage, where it reads or writes data needed by the current instruction. The Data Memory is an array of 2^{16} memory registers, each of size 1 byte. In each fetch or write, the Data Memory gives or takes 2 bytes of data specified at the location by the programmer.
- **Program Counter:** The program counter (PC) keeps track of the memory address of the instruction to be executed next. The address specified by the PC will be +2 for the next 1-word instruction and +2m for a branch instruction (where m is positive instruction) each time one instruction is executed. In our case, R0 is used as Program Counter.
- **Testbench & DUT:** A testbench is an HDL module that tests another module called the device under test (DUT). The testbench contains statements to apply inputs to the DUT and, ideally, to check that the correct outputs are produced. The input and desired output patterns are called test vectors.

2.2 Special Components

- **Datapath:** A datapath is a collection of functional units that perform data processing operations. It composes the central processing unit (CPU) and the control unit. In the Pipelined CPU, the datapath consists of various Stage units, Pipeline registers stacked in an alternate pattern, and a universal clock.

- **Stages:** There are six stages for the overall function of our RISC, namely Instruction Fetch (IF), Instruction Decode (ID), Register Read (RR), Execute (EX), Memory Access (MA) & Write Back (WB). Each stage consists of various functional units to achieve these tasks in a single cycle for every instruction. All stages have the clock and reset as input by default. All stages take input from a pipeline register (except Stage 1) and give their outputs to a pipeline register (except Stage 6).

- **Stage 1 (IF):** It has three 16-bit input addresses for the register file (2 for read and 1 for write) and two 16-bit output data from the register file. It consists of instruction memory, register file, muxes, sign-extender and adder as main sub-components. This stage is responsible for fetching the next instruction from the memory, preparing the data to be loaded into the pipeline register one and updating the program counter(PC).

- **Stage 2 (ID):** This has a controller as a main component which decodes the instruction and prepares data to be loaded into pipeline register two.

- **Stage 3 (RR):** This stage is also called the register read stage, with two 16-bit output addresses and two 16-bit input data for obtaining data from the register file. It's main components are muxes and a small controller. This stage reads the register numbers for the operands obtained from the instruction by stage 2. It also helps extend if the instruction uses an immediate value as one of its operands to minimize pipeline stalls. It prepares data to be loaded into pipeline register three.

- **Stage 4 (EX):** Its main sub-component is ALU, responsible for the instruction's actual computation or data manipulation. This stage has multiple operations like arithmetic, logical operations, memory operations, branch operations etc. Data for manipulation is taken from the previous pipeline register, and the computed data is sent to pipeline register 4.

- **Stage 5 (MA):** It consists of data memory as the main component, along with some muxes. This is mainly used for load and store operations as it accesses data memory.

- **Stage 6 (WB):** This stage is used to write back the resultant data to the register file and the data memory. It has 16-bit output data and two output single-bit Register File and Data Memory write signals. This is the last for an instruction as it returns the results to the register file. This involves receiving the result from the previous stage and writing it to the appropriate register or data Memory.

- **Register Hazard:** This deals with Register Fetch Data Hazards. It uses fast forwarding to provide the necessary data to an instruction in the Register Read Operation from the Execution, Memory Access or the Write Back Stage.

- **Memory Hazard:** This deals with Memory Fetch Data Hazards. It uses fast forwarding to provide the necessary data to an instruction in the Memory Access from the one instruction in the Write Back Stage.

- **Branch Detector:** It detects branch instructions. It operates in the Fetch Stage of the pipeline, where it examines the instruction before giving it to the pipeline register. If the instruction is a branch, the branch detector must calculate the target address of the branch instruction and update the program counter (PC) accordingly. Branching is done in the Fetch Stage to increase efficiency by not wasting cycles. It is a pseudo-Branch Predictor also, but it is directed to always branch when it sees a branch instruction, irrespective of the branch conditions. This is done as most of the branch instruction leads to Branch.

- **Branch Corrector:** It corrects the PC value if our earlier branch prediction was wrong. It changes some Pipeline Registers to eliminate the wrong instructions loaded between Branch Detector and Branch Corrector. This is done using Enabler and Disabler.

- **LM\SM Unit:** It uses penc, an adder and registers to give the register file address and memory address required by LM/SM as outputs, periodically and sequentially.

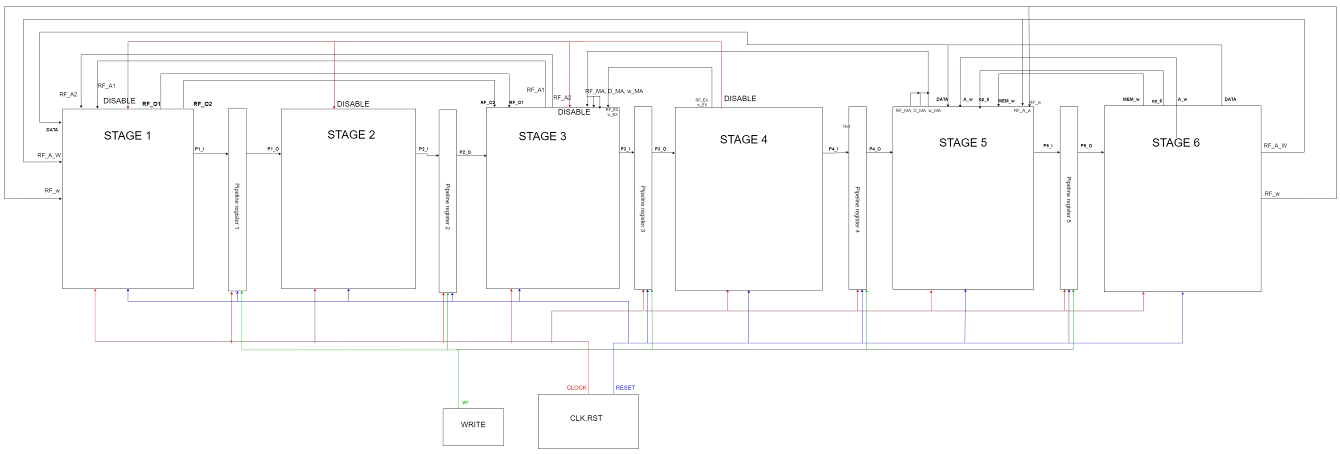
- **penc:** It is used for LM/SM. It takes an 8-bit string input and finds the first high-bit address. This address is given as an output along with an 8-bit string same as the input but with the first high bit flipped.

- **Decider:** This is used to change the Register Flag write value for some instructions if they do not satisfy the write conditionality.

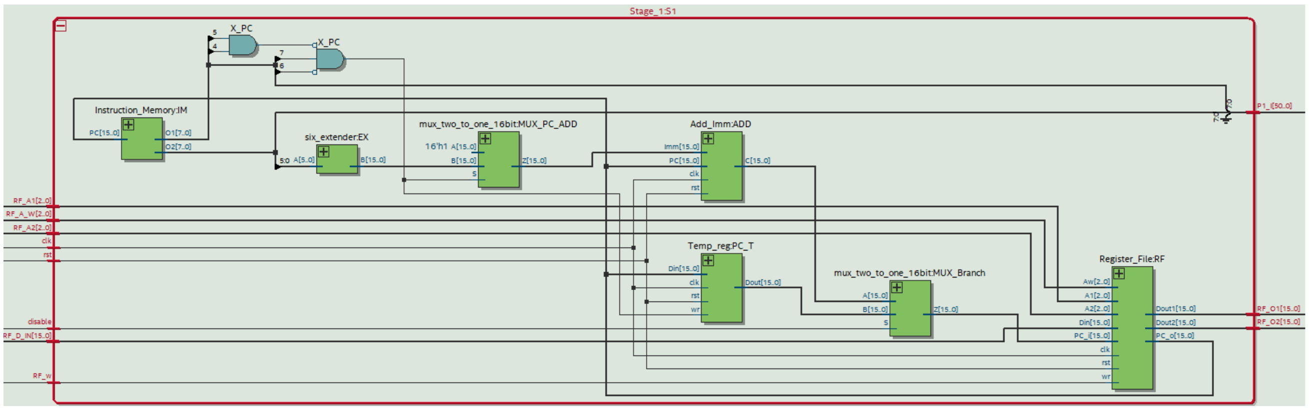
- **Controller:** It is used for Instruction Decoding. Based upon the instruction, the controller sends all control signals for the particular instruction along with the decoded information from the instruction into the pipeline register. These control signals are extracted and used at different stages for that instruction.

- **ALU:** ALU is a main component of the central processing unit, which stands for arithmetic logic unit and performs arithmetic and logic operations. It has the ability to perform all processes related to arithmetic and logic operations. We performed the following ALU operations in the ALU: ADD A with B (with and without carry), ADD A with the complement of B (with and without carry), NAND A with B, NAND A with the complement of B & subtraction. The ALU's operands and code tells which operations must perform according to input data.

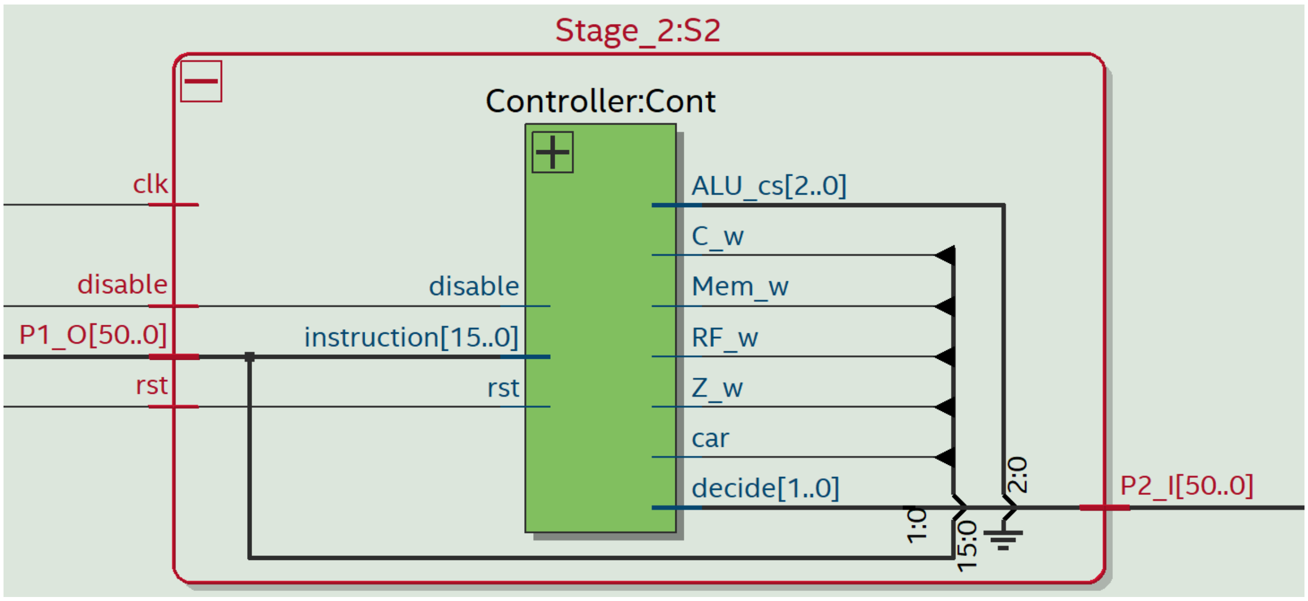
3 CPU Datapath



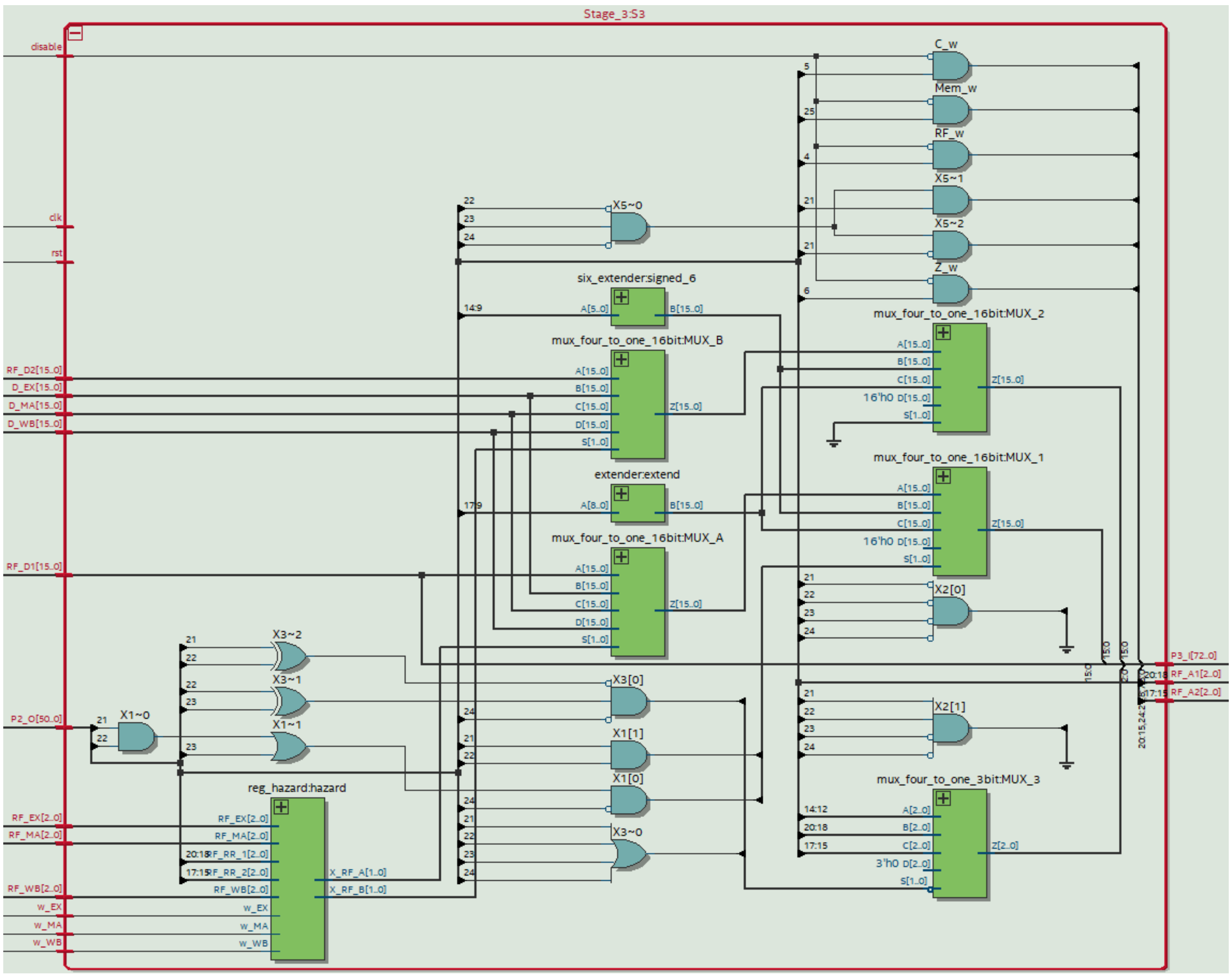
3.1 Stage 1



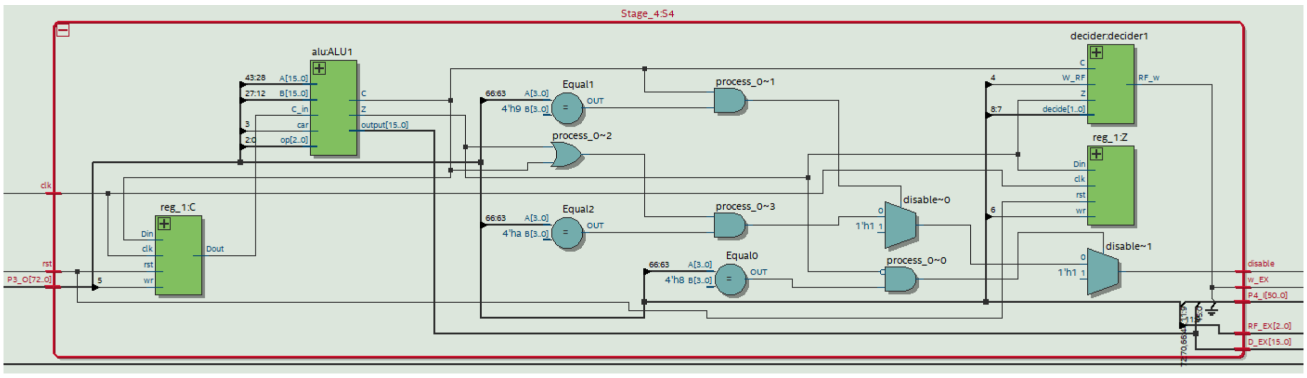
3.2 Stage 2



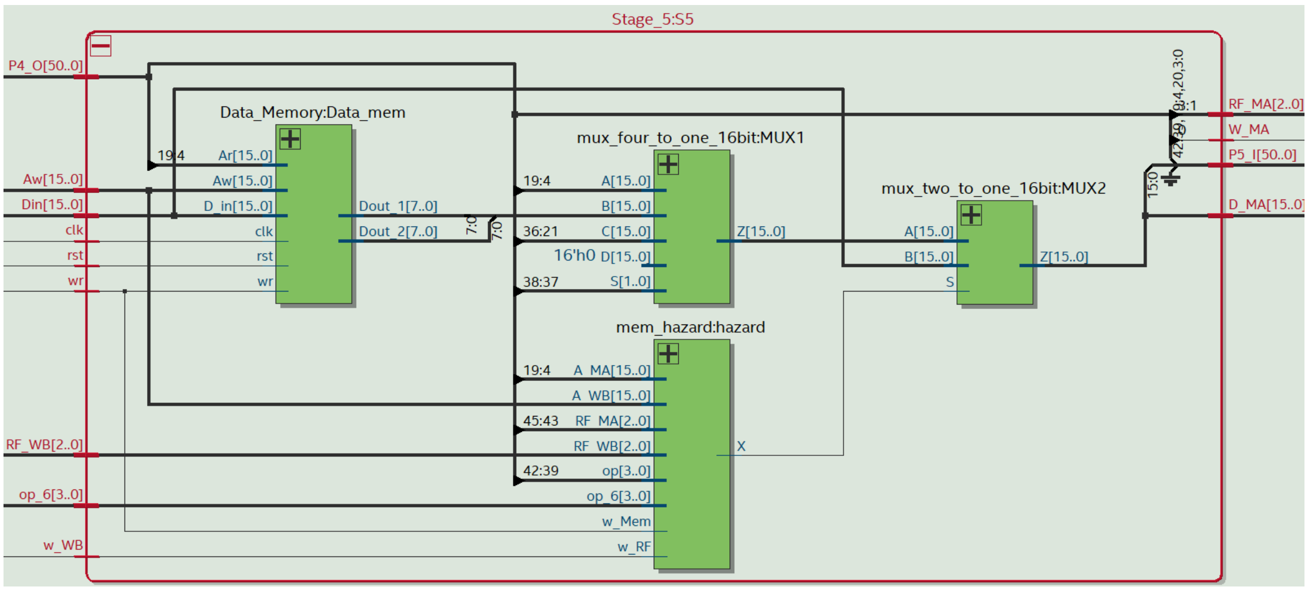
3.3 Stage 3



3.4 Stage 4

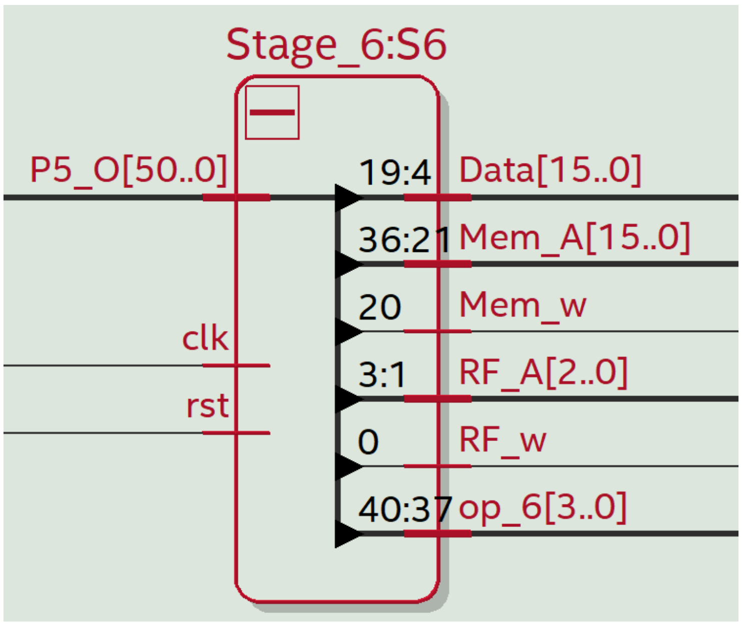


3.5 Stage 5



Instruction Formats

3.6 Stage 6



Instruction Formats

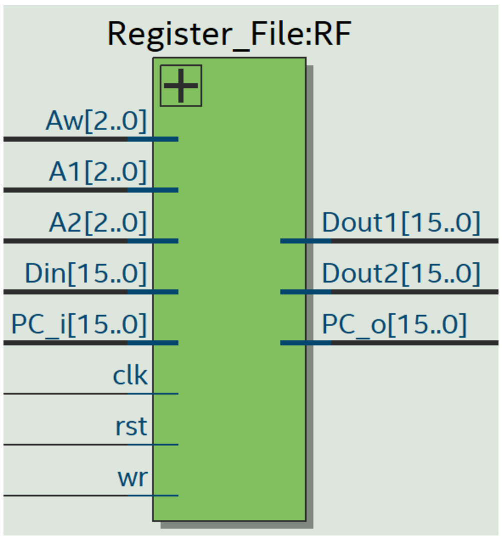
4 Hazards

In a pipeline CPU, a hazard is a situation where the pipeline cannot proceed with its normal operation, which may result in incorrect or unexpected behaviour. Hazards in a pipeline CPU can cause delays, decrease performance, and result in incorrect operation. Hazard detection and resolution techniques ensure the pipeline CPU operates efficiently and effectively. Three types of hazards can occur in a pipeline CPU: structural hazards, data hazards, and control hazards.

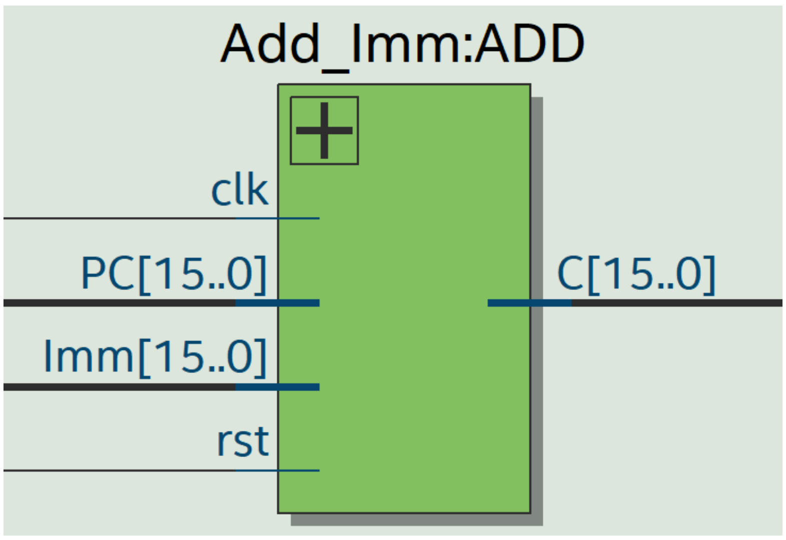
4.1 Structural Hazards

Structural hazards occur when the hardware resources required by two or more instructions overlap. This can happen when an instruction requires a hardware component already used by another instruction. For example, a structural hazard occurs if two instructions simultaneously require the ALU (Arithmetic Logic Unit). Structural hazards can be resolved by adding hardware resources or rearranging instructions. Some structural hazards with hardware components we solved:

- **Simultaneous Read Write in Register File:** To enable write operations, we need to set the register file to write mode by setting a specific input address. Once in write mode, any register can be written to. But the register file can be read at any time. This approach ensures that the register file is always read enabled, and write operations are only allowed when necessary. Furthermore, we created two input address lines to enable multiple reads, and two output data read address lines. This allows multiple reads from the register file simultaneously without causing conflicts or errors in the single register read cycle.
- **Simultaneous ALU Demand:** To solve the issue of simultaneous ALU demand in a pipelined CPU, we created specialized ALUs dedicated to specific opcodes in their respective stages. Since most instructions require an ALU, not necessarily in the same stage, a single ALU cannot meet the demand. By creating opcode-specific ALUs, we can effectively increase the number of ALUs available to the CPU and distribute the workload across multiple stages, reducing the overall demand on any AL. This allows us to work with a single master ALU in the execution stage.



Register File



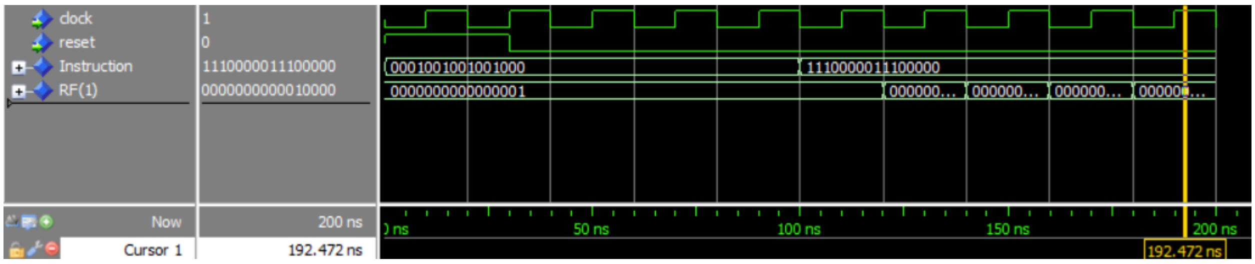
Specialised ALU

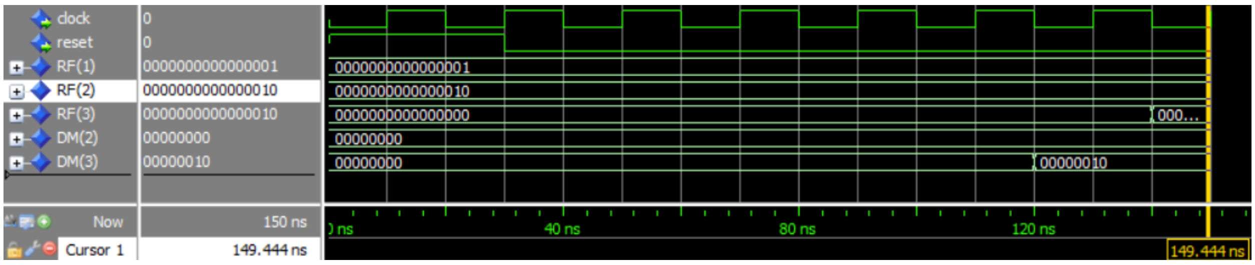
4.2 Data Hazards

Data hazards occur when an instruction requires data that is not yet available because an earlier instruction in the pipeline is still processing it. This can happen when two or more instructions that depend on each other are scheduled too closely in the pipeline, causing a stall or delay. To prevent data hazards, pipelined CPUs use various techniques such as forwarding, stalling, and reordering instructions. Data hazards methods we used with:

- **Fast Forwarding:** In a pipelined CPU, some other instruction might require the data still being processed for an instruction. Due to this, there will be a requirement for the new value even before the new value has been stored. To resolve this hazard, we use the technique of fast forwarding. For this, we directly provide the data from the Execution, Memory Access and Write-back stage to the Register Read Stage, Execution or Memory Access Stage and select between these data in decreasing order of priority. Three Components were used for fast-forwarding:

  1. The component reg_hazard lies in Stage 3 or the Register Read Stage. It decides the control value for a mux, which decides which data (Data from Register Read, Execution, Memory Access and write-back stages) must be sent to Stage 4 or the Execution Stage.
  2. The component LM_hazard lies in Stage 4. It decides the control value for a mux, which decides which data (Data passed through Register Read and Memory Access stages) must be sent to the ALU for execution.
  3. The component mem_hazard decides the control value for a mux, which decides which data (passed from Execution or the write-back stage) must be sent to Stage 6 or Write-back.

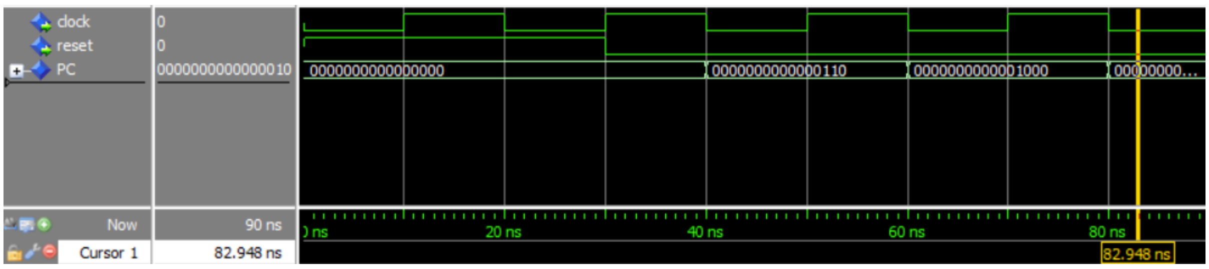




4.3 Control Hazards

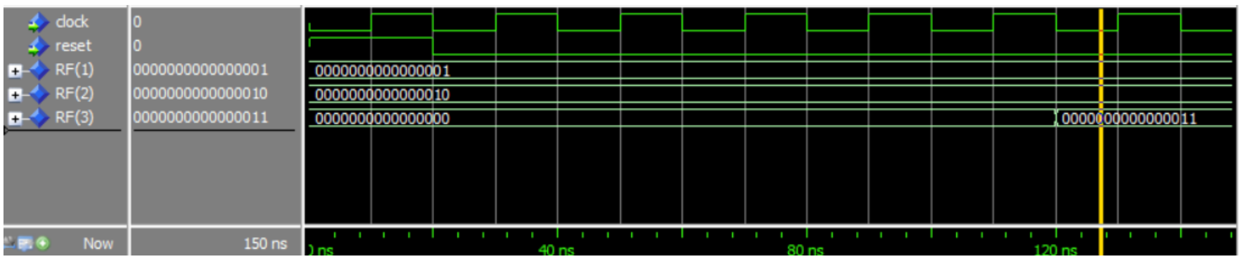
Control hazards occur when the pipeline must decide based on the outcome of a previous instruction, such as a conditional branch. If the outcome of the instruction is not yet known, the pipeline must stall until the outcome is determined. Control hazards can be resolved by predicting the outcome of the instruction and executing the next instruction accordingly or by delaying the execution of the instruction until the outcome is known. Control hazards we dealt with:

- **Branch Correction:** In Branch instructions, we branch to the value PC+2*Imm, without checking branch conditions. If we discover that our earlier branch was wrong, we update the PC value to the correct one; otherwise, the PC functions as normal. This is achieved with the help of a temporary PC register which stores PC+2 only in case of branch conditions. When we discover a faulty branch after branching, we push this stored value of PC. We also turn off all other stored wrong instructions by disabling their writes in different components.

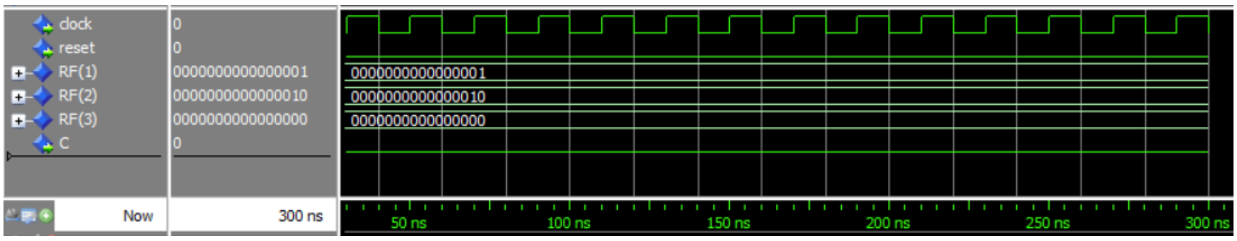


5 Simulation Results

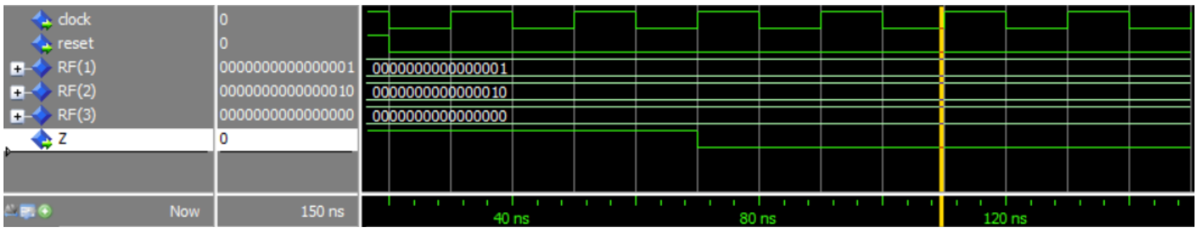
5.1 ADA



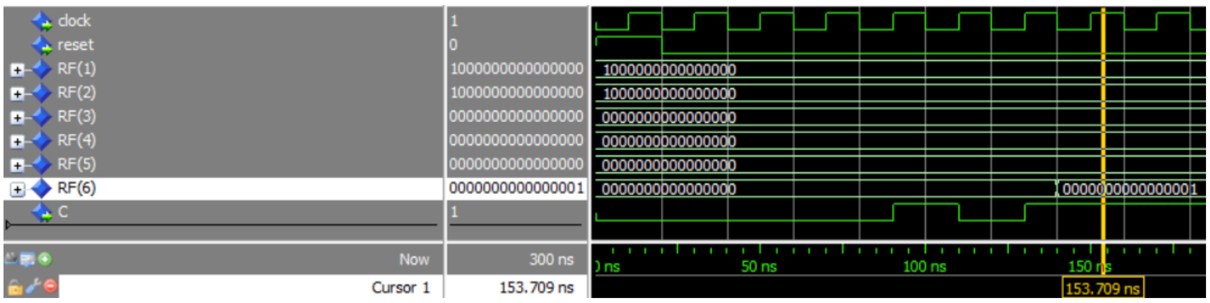
5.2 ADC



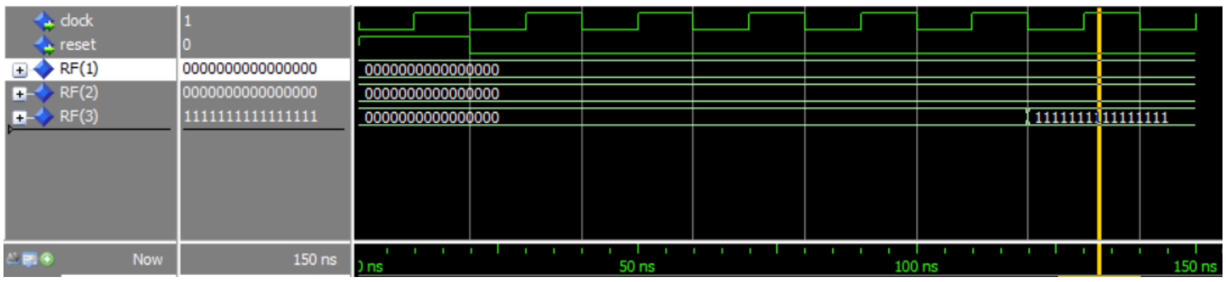
5.3 ADZ



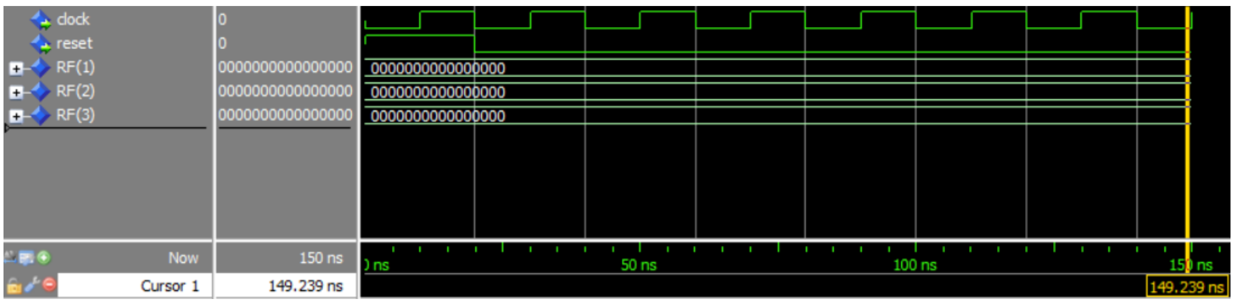
5.4 AWC



5.5 ACA

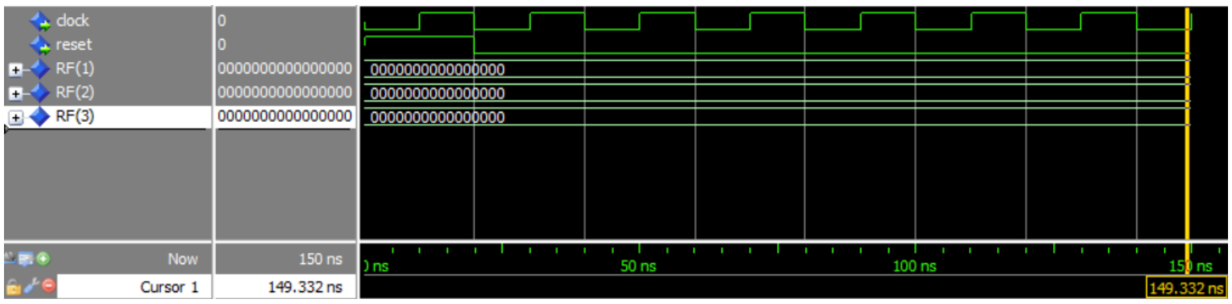


5.6 ACC



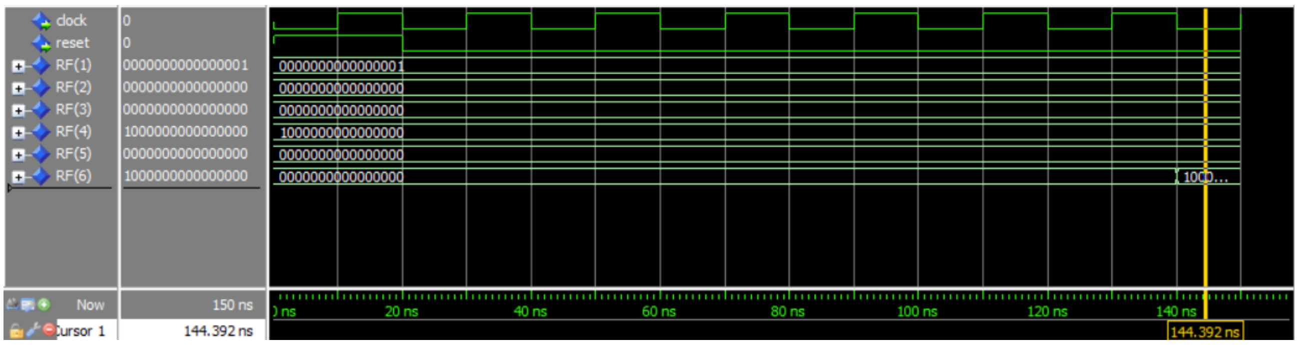
```
clock    0
reset    0
RF(1)    0000000000000000
RF(2)    0000000000000000
RF(3)    0000000000000000
Now      150 ns
Cursor 1 149.239 ns
```

5.7 ACZ



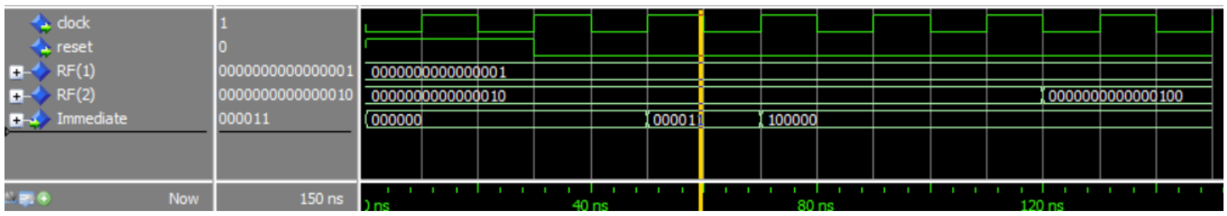
```
clock    0
reset    0
RF(1)    0000000000000000
RF(2)    0000000000000000
RF(3)    0000000000000000
Now      150 ns
Cursor 1 149.332 ns
```

5.8 ACW



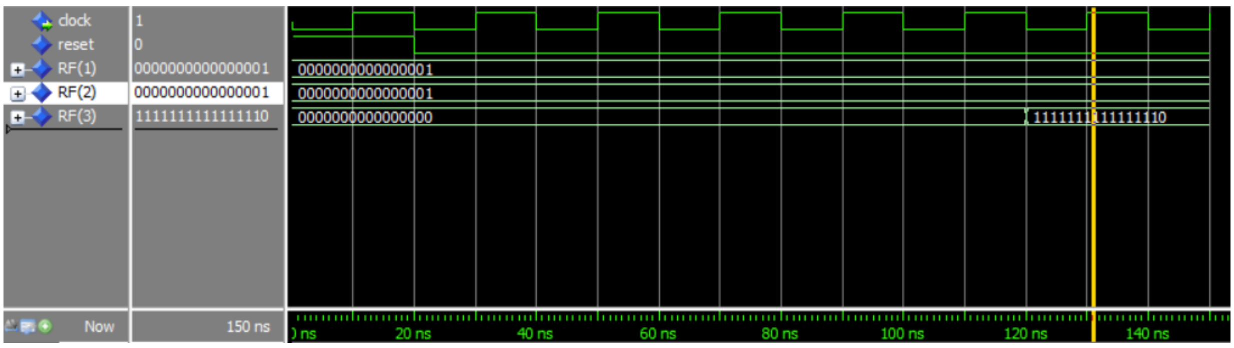
```
clock    0
reset    0
RF(1)    0000000000000001
RF(2)    0000000000000000
RF(3)    0000000000000000
RF(4)    1000000000000000
RF(5)    0000000000000000
RF(6)    1000000000000000
Now      150 ns
Cursor 1 144.392 ns
```

5.9 ADI



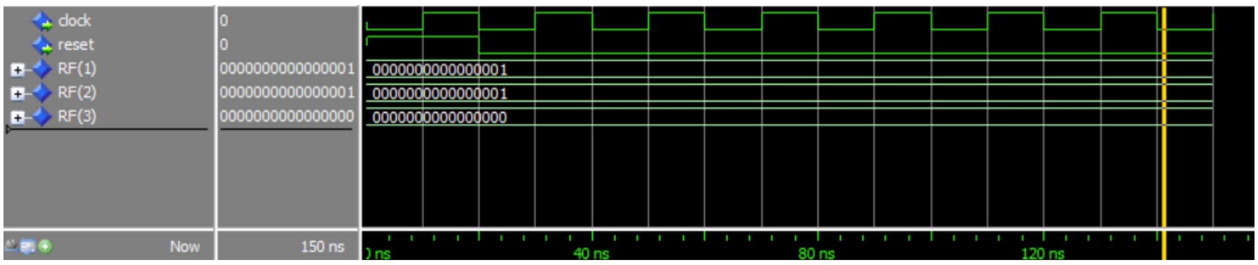
```
clock      1
reset      0
RF(1)      0000000000000001
RF(2)      0000000000000010
Immediate  000011
Now        150 ns
```

5.10 NDU

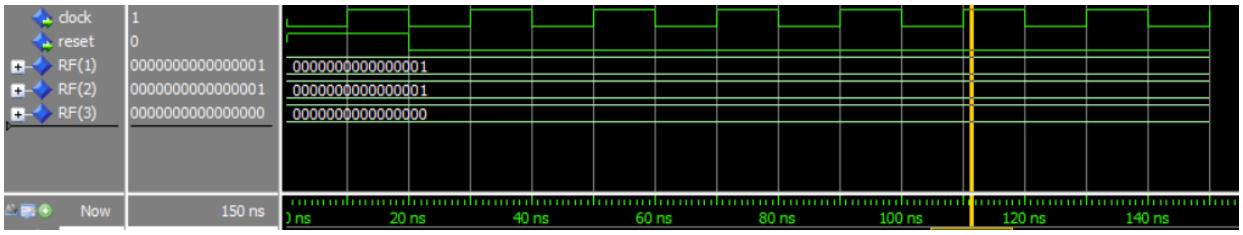


```
clock    1
reset    0
RF(1)    0000000000000001
RF(2)    0000000000000001
RF(3)    1111111111111110
Now      150 ns
```

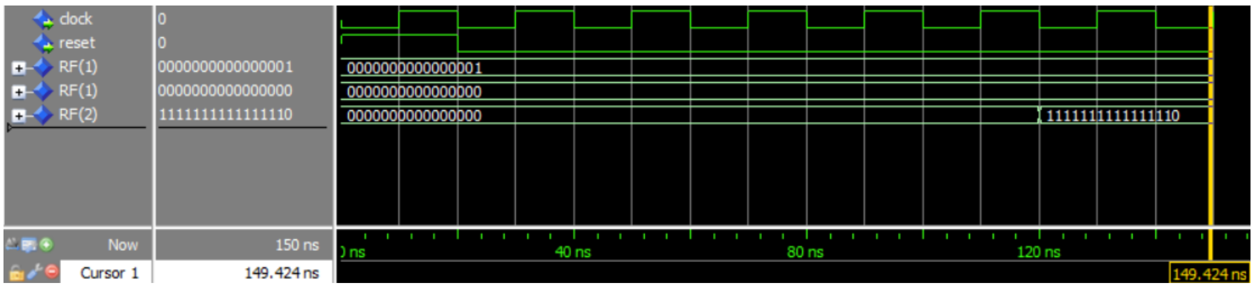
5.11 NDC



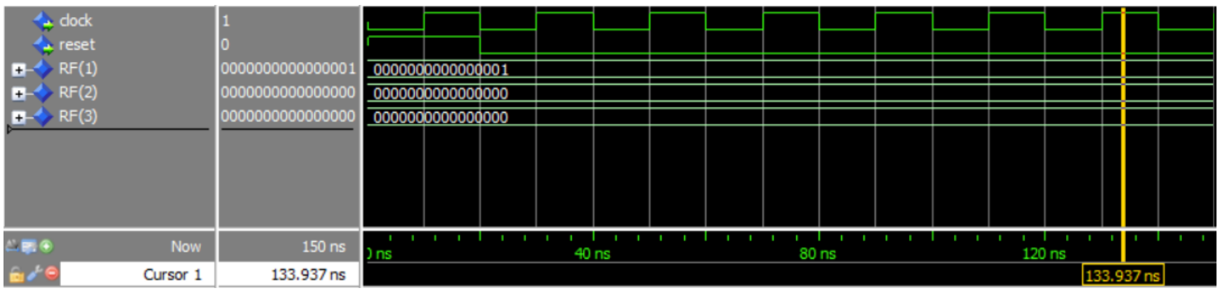
5.12 NDZ



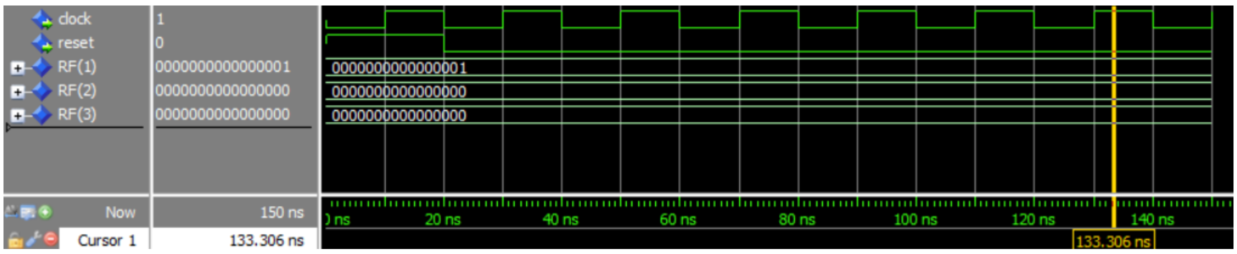
5.13 NCU



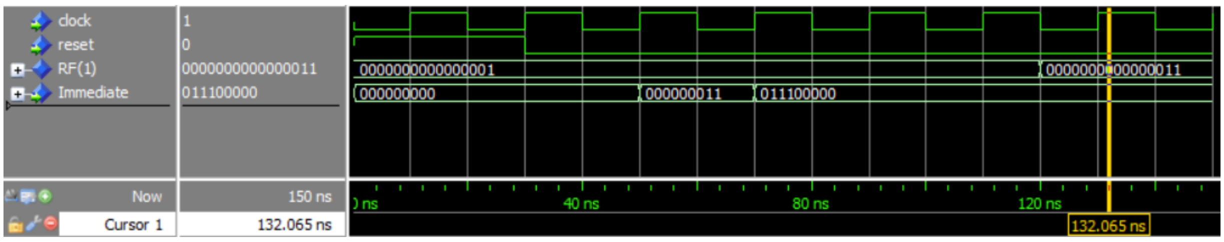
5.14 NCC



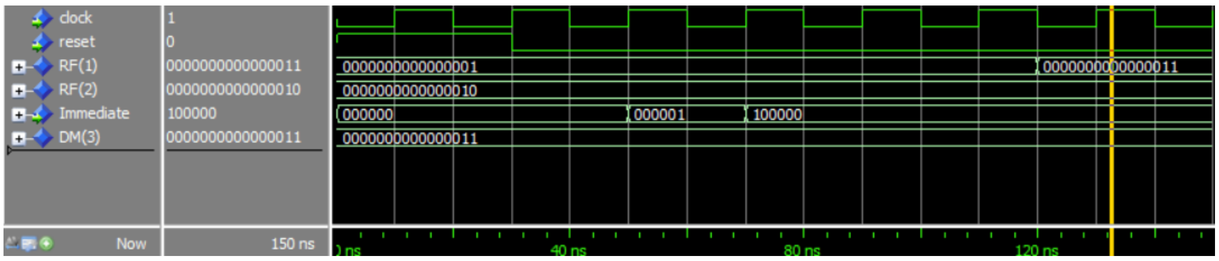
5.15 NCZ



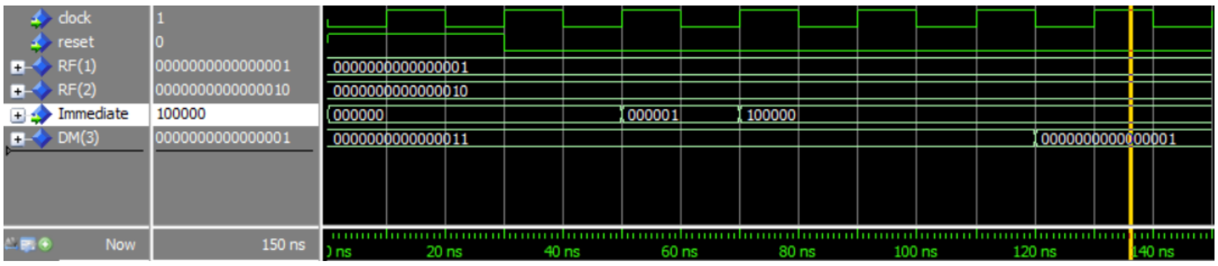
5.16 LLI



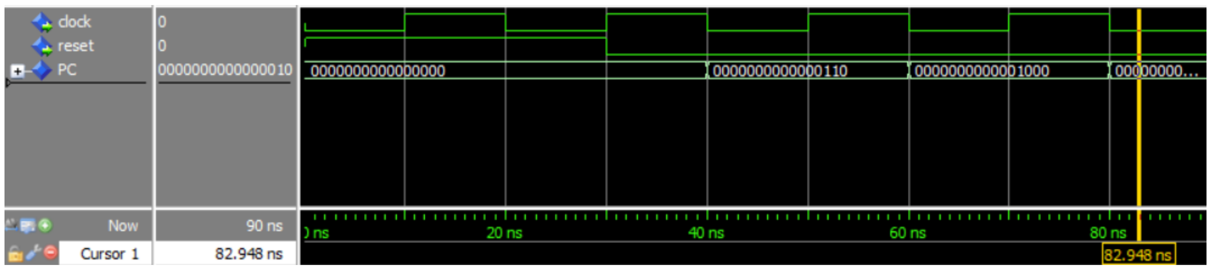
5.17 LW



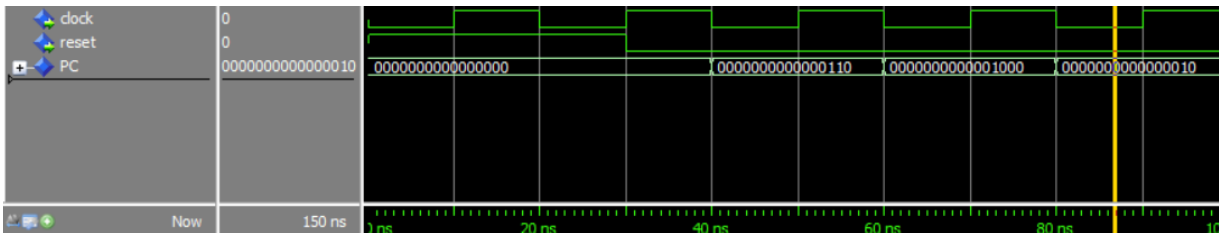
5.18 SW



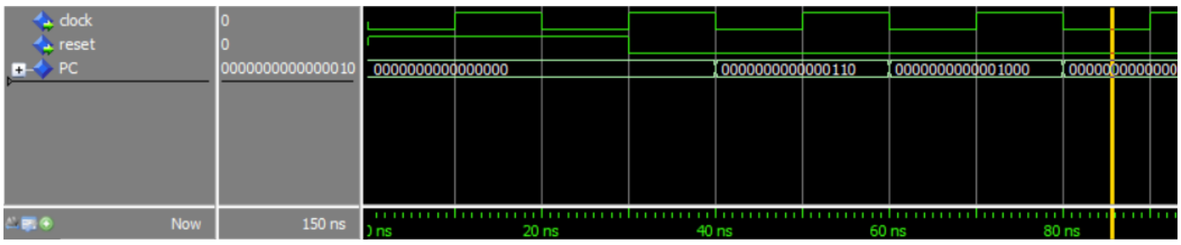
5.19 BEQ



5.20 BLT



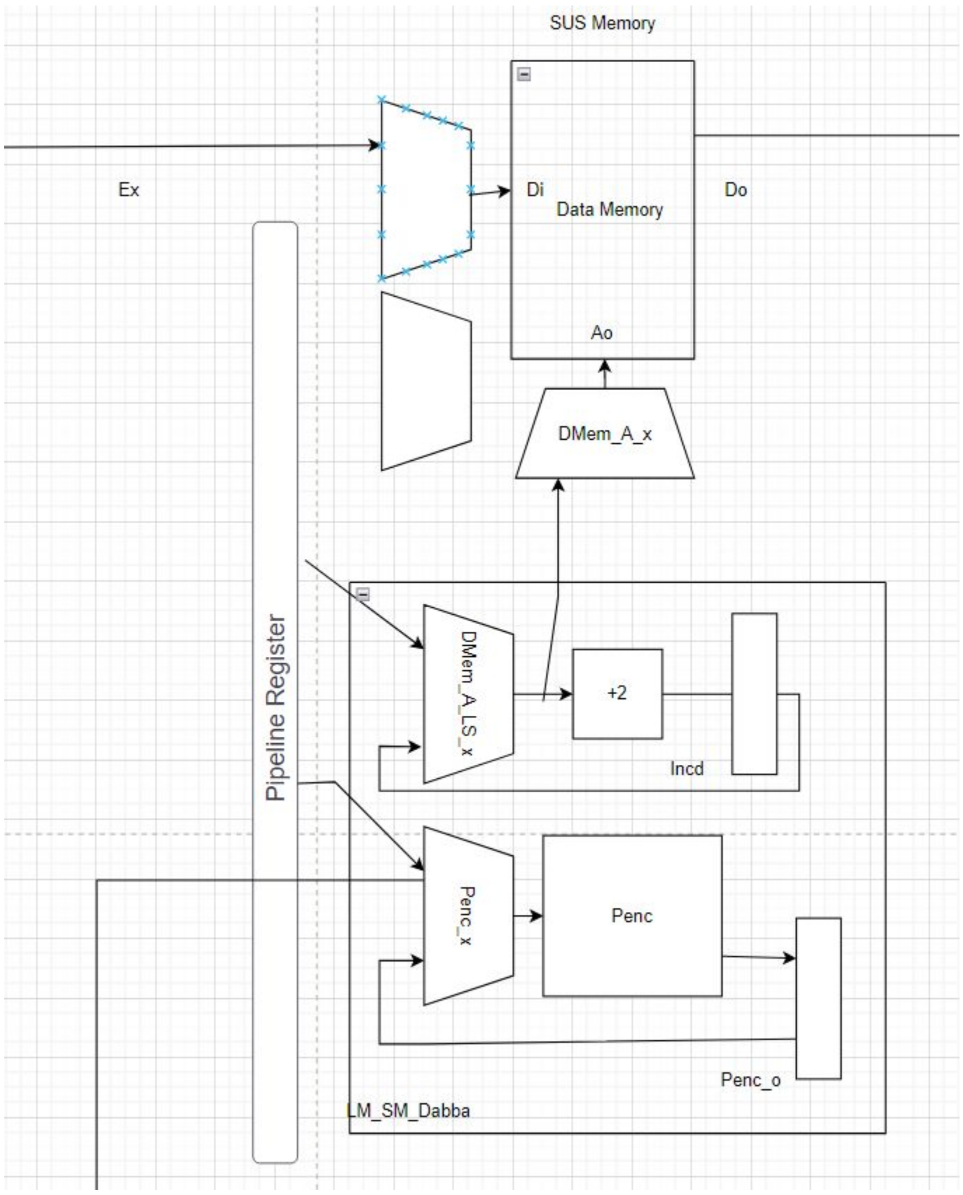
5.21 BLE



6 Remaining Instructions

6.1 LM, SM

They are implemented using the penc component by pausing the rest of the CPU. Once completed, the CPU pipeline operates as normal.



6.2 JAL, JLR, JRI

They are implemented using the penc component by pausing the rest of the CPU. Once completed, the CPU pipeline operates as normal.