# EE309 - Microprocessors Course Project
# IITB-RISC Pipelining

Group ID : 42
Kshitij Vaidya (22B1829)
Jainam Ravani (22B1242)
Jay Mehta (22B1281)
Adit Srivastava (22B1269)

May 10, 2024

## Contents

## 1 Introduction

The IITB-RISC is a 16-bit computer system, designed with simplicity in mind for educational purposes. It is based on the Little Computer Architecture. The IITB-RISC-23 variant of this system features 8 general-purpose registers (R0 to R7), with the R0 register always storing the Program Counter. All addresses and instructions in this system are byte addresses, and it always fetches two bytes for instruction and data. This architecture incorporates a condition code register with two flags: the Carry flag (C) and the Zero flag (Z). Despite its simplicity, the IITB-RISC-23 is capable of solving complex problems. It supports predicated instruction execution and multiple load and store execution. The machine-code instruction formats include R, I, and J types, with a total of 14 instructions.

The Instruction Set is a pretty simple one with just 14 instructions of which 2 instructions are R type, 7 instructions are I type and 5 instructions are J type.

## 2 Components Used

1. 16 Bit ALU(for EX)
2. +2 Adder(for IF, EX)
3. Instruction Memory
4. Data Memory
5. Counter(for LM)
6. Temporary Register file(for SM)
7. Comparator
8. Register File
9. Pipeline Register(IF-ID, ID-RR, RR-EX, EX-Mem, Mem-WB)
10. Forwarding Unit
11. Hazard Mitigation Unit

# 3 Datapath

## 6-Stage RISC Pipeline

The 6-stage RISC pipeline consists of the following stages:

### 3.1 Instruction Fetch(IF)

In this stage, the execution of the instruction starts. The PC is fetched from R0 and fed to the Address inside the Instruction Memory/Cache and the instruction is read and stored in the IF/ID pipeline register. The PC is also incremented by 2 and stored inside R0 in this stage itself thus preparing the pipeline to start executing the next instruction in the next cycle. The values for PC and PC+2 are also stored in IF/ID pipeline register as they'll be needed in the future for some instructions.

### 3.2 Instruction Decode(ID)

In this stage, the instruction is broken down into various operands and destinations and immediate bits to prepare it for the Register Read stage where the actual operands are read. The opcodes and condition bits are also received in a similar fashion. The Opcode and condition bits will then be used in the execute stage to control the ALU and determine which operation to be do and under what condition.

### 3.3 Register Read(RR)

In this stage, the actual operands are fetched based on the register addresses fetched in the IF stage. This is the stage where the forwarding of operands is happening in case of dependency hazards.

### 3.4 Execute(EX)

In this stage, once the operands and destinations are read and gathered, we perform the main computation part of the instruction(i.e. the operation itself is performed here). The results of this operation(value of the destination register/ memory address to be accessed/ address where we need to jump) is then stored in a temporary pipeline register.

### 3.5 Memory(MEM)

In this stage, we get our one chance to access the data memory in the pipeline. It uses the outputs stored in the previous temporary register to access an address from the data memory if needed. For load instructions the MEM stage will read data from the data memory and store it in a temporary pipeline register and for store instructions the MEM stage will write data from a register to the appropriate memory location.

### 3.6 Write Back(WB)

The results obtained from the Execute and Memory stages are stored in the temporary pipeline register(Mem-WB). In this stage, we write the results obtained from the Execute and Memory stages back into the appropriate registers.

The datapath of a 6-stage RISC pipeline is a complex circuit that allows the processor to fetch, decode, execute, and write back instructions in a pipelined fashion. This means that multiple instructions can be in different stages of the pipeline at the same time, which can improve the performance of the processor.

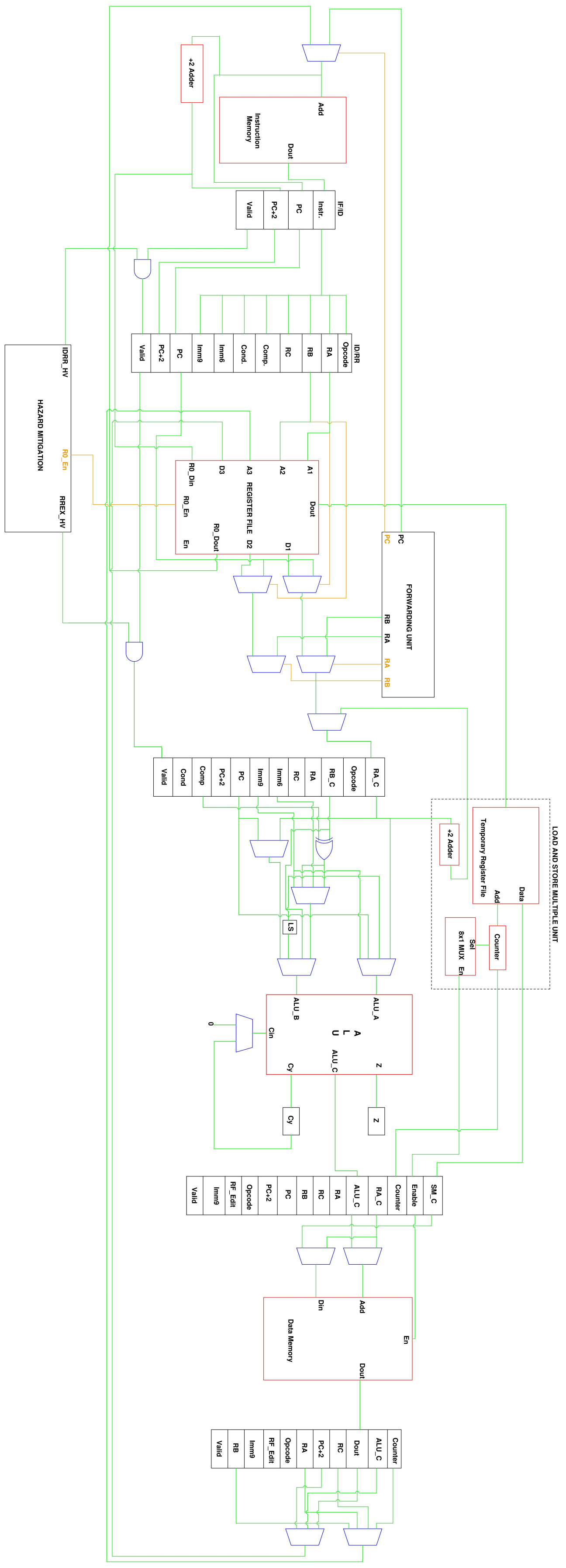

Figure 1: Final Datapath Of IITB-RISC

## 4 Load Multiple/Store Multiple

The previous explanation of stages works fine for the simple instructions like ADA, NDU, LW, BEQ etc. But it would be very difficult to execute Load Multiple or Store Multiple instructions on such a simple architecture. So for this, we decided to add some new components to ease the execution of LM/SM and integrate it into the pipeline.

The additional hardware includes:

- A +2 Adder for updating memory location to read from/write to
- A 8x1 MUX to select the immediate bit corresponding to a register
- A counter to keep track of the register to be written into/read from
- A temporary register file that stores values of all registers at the start of the execution of SM so that execution of other instructions doesn't affect it.

### 4.1 Load Multiple

When the LM instruction is in the RR stage, a hazard is raised due to which the pipeline is stalled for stages IF, ID, RR. This hazard stays for 3 cycles or when all the instructions before LM in the pipeline are invalid (which ever occurs earlier). This is done to make sure that none of the dependencies directly affect the LM instruction. Direct forwarding could have been used but since LM is a very complex instruction, we found this to be more reliable and easier to implement.

After this, the instruction moves to the EX stage where the actual execution of the instruction starts. The pipeline is then again stalled. The counter is set to 0111 and then down-counted on each cycle. The 8x1 MUX then selects the corresponding bit from immediate-9 stored in the RR/EX pipeline using the counter bits (0-2) for selecting. The content of RA_C is then incremented by 2 using the +2 adder and the content of RA_C is changed based on the enable bit received from the MUX output. The same enable bit is added to EX/MEM pipeline register.

When the counter reaches 0000, the pipeline moves forward one step and then if R0 is being changed, it's forwarded to IF stage and if R1 is an operand for the instruction in the RR stage then its forwarded to RR stage. And now the pipeline moves forward like normal.

### 4.2 Store Multiple

For the Store Multiple instruction, when the Store Multiple instruction reaches the ID stage, a hazard is raised to stall the pipeline in the IF and ID stages. This hazard stays for 3 cycles to allow all the instructions before SM in the pipeline to complete their execution. This is done to address any direct dependencies on the Store Multiple instruction. For the SM instruction, we have created a new Temporary Register File that reads the values of the Register File when the instruction is in the Register Read instruction. This step is done to make sure that the correct values of the register are read.

When the SM instruction reaches the Execute Stage, we raise another hazard to stall the system. This stall continues till be down counter reaches 0000. This means that the contents of all required registers have been stored in memory. To perform this stall, the enable bits of IF_ID, ID_RR and RR_EX pipeline registers are made 0. This holds the instructions in the pipeline registers itself while the SM loop is being executed. Finally when all the required registers have been stored to memory, the pipeline can resume normally. This means that all enable signals of pipeline registers will be made 1. Here, we don't need to worry about any immediate memory dependencies because the data memory can be accessed only once by any instruction. This means that is the subsequent instruction wants to access the same memory location as the SM instruction, it can do so normally in the pipeline without requiring any forwarding.

## 5 Forwarding Unit

Forwarding is the movement of data from a later stage in the pipeline to an earlier stage because the two instructions depend on the same operand and thus the value for the later instruction must be taken according to the updated result of the previous instruction. This however is not possible if we allow the pipeline to run without any interference. Forwarding was required in the following cases:

### 5.1 Immediate Arithmetic Dependencies

Cases where the destination address of an arithmetic instruction is the source address of the next instruction, the result of the ALU can be forwarded from the execute stage to the register read stage such that the correct value of the operand is stored in the pipeline register.

### 5.2 Arithmetic Instructions with R0 as the Destination

These instructions are a kind of Jump instruction so we will need to stall for 2 cycles. The result of the Execute stage which has the correct value of R0 for the next instruction is forwarded to the Instruction Fetch stage from where the correct next instruction is executed. The incorrectly loaded instructions are also flushed by making the Valid Bits of the required Pipeline Registers 0.

### 5.3 Loading into R0

In the case where we are loading a value into the register R0, the next instruction to be executed will be the new value of R0 and not R0+2(the instruction in the stage just behind "load into R0"). We will need to stall for 3 cycles. The result of the Memory stage is then forwarded into the Instruction Fetch stage from where the correct next instruction is executed. The incorrectly loaded instructions are also flushed by making the Valid Bits of the required Pipeline Registers 0.

### 5.4 Load with Immediate Dependency

In the case where we have a load instruction which loads a value onto a register which is required to be read in the very next instruction, we will need to execute the next instruction with this newly loaded value and not the original one. We will have to stall for 1 cycle. The result of the Memory stage is forwarded to the Register Read stage from where the correct value of the dependent register is read. We will need to ensure that the very next instruction waits in the Register Read stage while our load instruction reaches the Memory stage.

# 6 Hazard Mitigation Unit

## 6.1 List of Hazards

| Hazard | Solution |
|---|---|
| Load with immediate dependency | Stall for one cycle |
| Load into R0 (same as branching) | Flush for 3 cycles |
| Branching/R0 destination of Arithmetic Logical Instruction/LLI with destination R0 | Stall for 2 cycles |
| Arithmetic dependency | Only forwarding required |
| JLR/JAL: if RA=R0 | Branching irrespective of PC+2 |
| Load Multiple Implementation | Explained in 5.2.4 |
| Store Multiple Implementation | Explained in 5.2.5 |

Table 1: List of Hazards along with their solutions

## 6.2 Implementation of Solutions

### 6.2.1 Load with immediate dependency

This hazard will be detected when the Load instruction is in execute stage. (outputs of the ID-RR and RR-EX pipeline registers are considered)

After this, to stall for one cycle we set the following bits accordingly for one cycle.

Valid-RR-EX = 0 (So that instruction which was in Register read stage doesn't execute as it has incorrect value of the dependent register)
Enable-R0 = 0 (So that a new instruction isn't loaded while stalling)
Enable-IF-ID = 0 (Stalling the instruction at Instruction Decode stage)
Enable-ID-RR = 0 (Stalling the instruction at Register Read Stage)

This ensures a one cycle stall, after which normalcy is restored.

### 6.2.2 Load into R0

This hazard is detected when the Load into R0 instruction reaches the memory stage. After forwarding the correct value of R0 for the next instruction we will need to flush the 3 instructions after Load into R0 as they are invalid instructions.

To do this, we take the following actions:

Enable-R0 = 1 (Needed to load the new value of R0 from Memory Stage)
Valid-EX-MEM = 0 (Flushing the invalid instruction at Execute Stage)
Valid-RR-EX = 0 (Flushing the invalid instruction at Register Read Stage)
Valid-ID-RR = 0 (Flushing the invalid instruction at Instruction Decode Stage)
Enables for Z and $C_y$ = 0 (So that the invalid instruction at Execute Stage doesn't incorrectly change our flags)

After the 3 invalid instructions have been flushed, we can proceed normally.

### 6.2.3 Branching/R0 destination of Arithmetic logical Instruction/LLI with destination R0

This hazard is detected in the execute stage. After forwarding the correct value of R0 for the next instruction we will need to flush the next 2 instructions as they are invalid.

To do this, we take the following actions:

Enable-R0 = 1 (Needed to load the new value of R0 from Execute Stage)
Valid-RR-EX = 0 (Flushing the invalid instruction at Register Read Stage)
Valid-ID-RR = 0 (Flushing the invalid instruction at Instruction Decode Stage)

After the 2 invalid instructions have been taken care of, we proceed normally.

### 6.2.4 Load Multiple

This hazard is detected when the Load Multiple Instruction reaches the Register Read Stage.(using the opcode in the ID-RR pipeline register)

We first let all the instructions in the pipeline that are ahead of LM complete. (we stall for 3 cycles)

To do this we take the following actions:

Enable-R0 = 0 (So that a new instruction isn't loaded while stalling)
Enable-IF-ID = 0 (Stalling the Instruction at Instruction Decode Stage)
Enable-ID-RR = 0 (Stalling the LM instruction at Register Read Stage)
Enable-RR-EX = 1 (So that the Instruction just before LM moves forward)
Valid-RR-EX = 0 (Not executing LM until the previous 3 instructions are complete)
Counter= 0111 (A counter is set to perform an 8 cycle stall while LM is getting executed)

When Valid-RR-EX = Valid-EX-MEM = Valid-MEM-WB = 0(the 3 cycle stall is complete) :
LM moves forward (one step)

Enable-R0 = 1
Enable-IF-ID = 1
Enable-ID-RR = 1
Enable-RR-EX=1

Now a hazard is raised in the execute stage. We now stall until the counter reaches 0(8 cycles to execute LM for R7-R0)

Enable-R0 = 0
Enable-IF-ID = 0
Enable-ID-RR = 0
Enable-RR-EX = 0
Enable-RA-C= 1 if corresponding bit of the 8 bit immediate is 1 as RA will be incremented only if a register is updated (using a MUX)

When counter=0000, R0 is in Execute stage, R1 is in Memory Stage ...

Here we wait for one cycle until R0 reaches the memory stage after which we let the pipeline execute normally.(This is equivalent to the measures taken when there was a "Load into R0" instruction)

Enable-RR-EX = 1
Valid-RR-EX = 0
Enable-R0 = 0

Enable-IF-ID = 0
Enable-ID-RR = 0
for one cycle

After this we proceed normally.

Enable-RR-EX = 1
Enable-R0 = 1
Enable-IF-ID = 1
Enable-ID-RR = 1

### 6.2.5 Store Multiple

This hazard is detected when the Store Multiple Instruction reaches the Register Read Stage.(using the opcode in the ID-RR pipeline register)

We first let all the instructions in the pipeline that are ahead of SM complete. (we stall for 3 cycles)

To do this we take the following actions:

Enable-R0 = 0 (So that a new instruction isn't loaded while stalling)
Enable-IF-ID = 0 (Stalling the Instruction at Instruction Decode Stage)
Enable-ID-RR = 0 (Stalling the SM instruction at Register Read Stage)
Valid-RR-EX = 0 (Not executing SM until the previous 3 instructions are complete)
Counter= 0111 (A counter is set to perform an 8 cycle stall while SM is getting executed)

When Valid-RR-EX = Valid-EX-MEM = Valid-MEM-WB = 0(the 3 cycle stall is complete) :
SM moves forward (one step)

Enable-R0 = 1
Enable-IF-ID = 1
Enable-ID-RR = 1
Enable-TempRF=1
For one cycle

Now a hazard is raised in the execute stage. We now stall until the counter reaches 0(8 cycles to execute SM for R7-R0)

Enable-R0 = 0
Enable-IF-ID = 0
Enable-ID-RR = 0
Enable-RR-EX = 0
Enable-RA-C= 1 if corresponding bit of the 8 bit immediate is 1 as RA will be incremented only if a register is updated (using a MUX)

When counter=0000, R0 is in Execute stage, R1 is in Memory Stage ...

Unlike LM, here we don't have to worry about the "Load into R0" hazard and thus we can continue our pipeline normally.

Enable-RR-EX = 1
Enable-R0 = 1
Enable-IF-ID = 1
Enable-ID-RR = 1

## 7 Work Distribution

Contributions of each team member towards the completion of the project :

| **Team Member** | **Contributions** |
|---|---|
| Adit Srivastava | Report,VHDL Design of Miscellaneous Components, Debugging, Hazard and Forwarding Ideation |
| Jainam Ravani | VHDL Implementation of Pipelined Datapath, Debugging the Final Code, Hazard Unit Ideation |
| Jay Mehta | VHDL Implementation for Forwarding Unit, Hazard Mitigation Unit Implementation, ALU |
| Kshitij Vaidya | Primary Datapath Design, Forwarding, Hazard Mitigation Unit Design, Debugging |

Table 2: Contributions from Team Members