# IIT-B Pipelined RISC Report

Ayush Joshi-210100036 Harsh Shah-210100063
Soham Inamdar-210100149 Mayank Gupta-210101002

06 May 2023

## Contents

**1 Introduction** **1**

**2 Explanation of Stages** **2**
2.1 Instruction Fetch . . . . . 2
2.2 Instruction Decode . . . . . 2
2.3 Register Read . . . . . 2
2.4 Execute . . . . . 3
2.5 Memory access . . . . . 3
2.6 Write Back . . . . . 3

**3 Hazard Mitigation** **3**
3.1 Forwarding . . . . . 3
3.2 Stalling . . . . . 4

**4 Architecture** **4**

**5 Observations and Results** **4**

## 1 Introduction

In this project we have designed a 6-staged pipeline processor, IITB-RISC-23 providing an architecture of 26 instructions with 3 different instruction types. It is a very simple 16-bit computer system with 8 registers, 2 types of memory namely instruction memory and data memory, 1 ALU to perform arithmetic and logical operations, 2 adders, 2 sign extenders and a few MUXs. It should follow the standard 6-stage pipelines namely **instruction fetch, instruction decode, register read, execute, memory access and write back**. We have also implemented hazard mitigation techniques like **forwarding** and **stalling** to counter hazards caused by jumps and dependencies.

# 2 Explanation of Stages

## 2.1 Instruction Fetch

In this statge the first action performed is to read the instruction from memory using the address in the PC which is stored in R0. The PC and the corresponding instruction is stored in the pipeline register IF-ID to be used in further stages of the architecture. PC is stored so that it can be used for branching instructions in the execution stage. Then the PC address is incremented to the next so that it is ready to fetch the next instruction in the next cycle.

## 2.2 Instruction Decode

In this stage we need to break down every instruction and assign control bits. We are using a total of 11 control bits, 5 for controlling MUX related decisions while the other 6 are for ALU. Then we store all the control bits, instruction and PC in the ID-RR pipeline register. The control bits for all the instructions are as follows:

| Instruction | MUX_SE | MUX_BEQ | MUX_ALU | MEM_WR | MUX_WB | ALU_OP1 | ALU_OP2 | ALU_OP3 | ALU_OP4 | ALU_OP5 | ALU_OP6 | Control Signal |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ADA | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 00000000000 |
| ADC | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 00000010000 |
| ADZ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 00000001000 |
| AWC | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 00000000100 |
| ACA | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 00000000001 |
| ACC | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 00000010001 |
| ACZ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 00000001001 |
| ACW | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 00000000101 |
| ADI | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 00100000000 |
| NDU | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 00000100000 |
| NDC | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 00000110000 |
| NDZ | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 00000101000 |
| NCU | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 00000100001 |
| NCC | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 00000110001 |
| NCZ | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 00000101001 |
| LLI | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 10100000000 |
| LW | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 00101000000 |
| SW | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 00110000000 |
| LM | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 00001000000 |
| SM | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 00010000000 |
| BEQ | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 00011000000 |
| BLT | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 00011000000 |
| BLE | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 00011000000 |
| JAL | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 11100000000 |
| JLR | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 01000000000 |
| JRI | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 11100000000 |

## 2.3 Register Read

Here we read the value of Register $R_A$,$R_B$ and put it in the next pipeline register that is EX-MA with Instruction and PC. Note we read the value of register even if it is not needed or it doesn't even give the address of any particular register in the case of immediate, because we will be using our control bits to assign the value's hence not creating any problem. We also take of dependency of value of registers in the stage by using forwarding which is discussed later.

### 2.4 Execute

In this stage the opcode of the instruction in pipeline register is checked to determine the type of instruction. The corresponding control bits for the ALU_OP, (the alu operation) are assigned to the ALU. The control bits for the MUX are also assigned and then passed on to the next pipeline register. The 6 ALU_OP bits are encoded as (from left hand side) :

0 for ADD, 1 for NAND
0 for no carry, 1 for carry
0 for no zero, 1 for zero
0 for not with carry , 1 for with carry
0 for no immediate, 1 for immediate
0 for no complement operation, 1 for complement

### 2.5 Memory access

This stage writes data into memory if the MEM_WR control bit is set to 1. It can read or write data corresponding to the address of the memory location received from the ALU output. The data to be written (if MEM_WR is 1) is received from the register as per the register mentioned in the instruction. Memory is always read, irrespective of the fact that we need it or not since we can control its usage from the MUX in the write back stage.

### 2.6 Write Back

In this, stage we write data back to the registers. The control bit MUX_WB determines whether the data to be written comes from the ALU(MUX_WB=0) or memory(MUX_WB=1). The destination register is determined using the op_code of the instruction.

## 3 Hazard Mitigation

Hazards occur when there are instructions with dependencies on each other or in which we need to make a jump to different PC. To solve the dependency issue we use data forwarding and in instructions such as BEQ in which jumps are involved we need to use stalling as well.

### 3.1 Forwarding

Data forwarding is used when we encounter instructions like those given below:

```
ADA R1, R2, R3 --add content of R2 and R3 and write to R1
NDU R4, R5, R1 --perform nand operation on content of R5 and R1 and write to R4
```

In the above example, it is clear that the second instruction depends on the first one. Due to the nature of our pipeline, the second instruction will be in execution before data is written back in the first one. Hence, we use data forwarding which means that we forward ALU output of instr. 1(in memory access stage) to instr. 2(in execution stage). Hence we have implement data forwarding for values of both registers $R_A$,$R_B$, where we check if it has any dependency and if it has any we update its value directly from the corresponding stage but in an order sequence. Suppose we have register dependency on previous instruction and the one before the previous as well, then we'll have to consider the value written in previous instruction because it is the latest updated one.

### 3.2 Stalling

Stalling is used generally when dependencies present themselves in the form of BRANCH and JUMP type instructions. Here, we stall i.e. stop the execution through the pipeline for one or more cycles as and when required by the instructions. In order to achieve this, we set a default op_code which is interpreted as no instruction. Hence, the data will be forwarded through the pipeline but no action will be taken.

## 4 Architecture

Following is the overall pipelined architecture of the RISC designed by us:

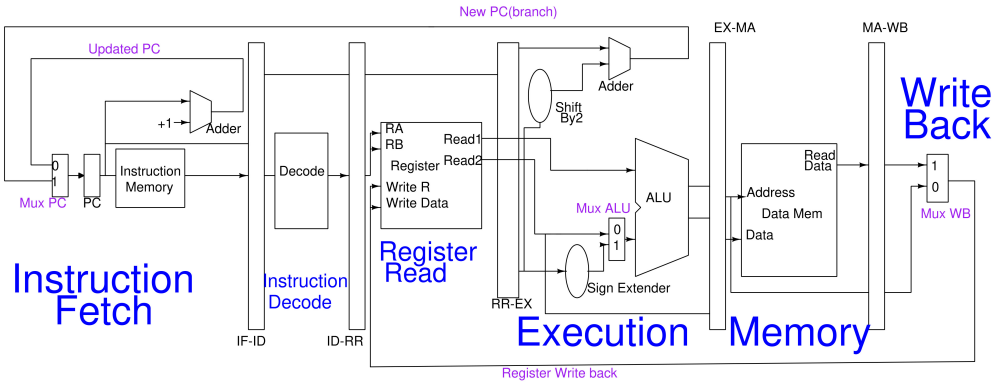


Figure 1: Pipelined Architecture of RISC

## 5 Observations and Results

The results that we obtained for a few different instructions are as follows:

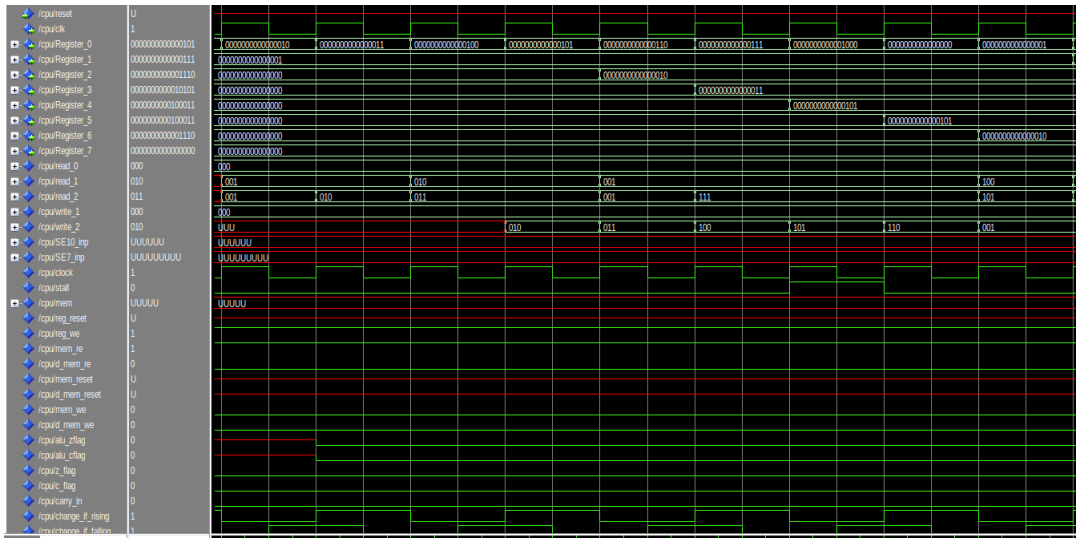


Figure 2: Waveform of series of instructions having dependencies

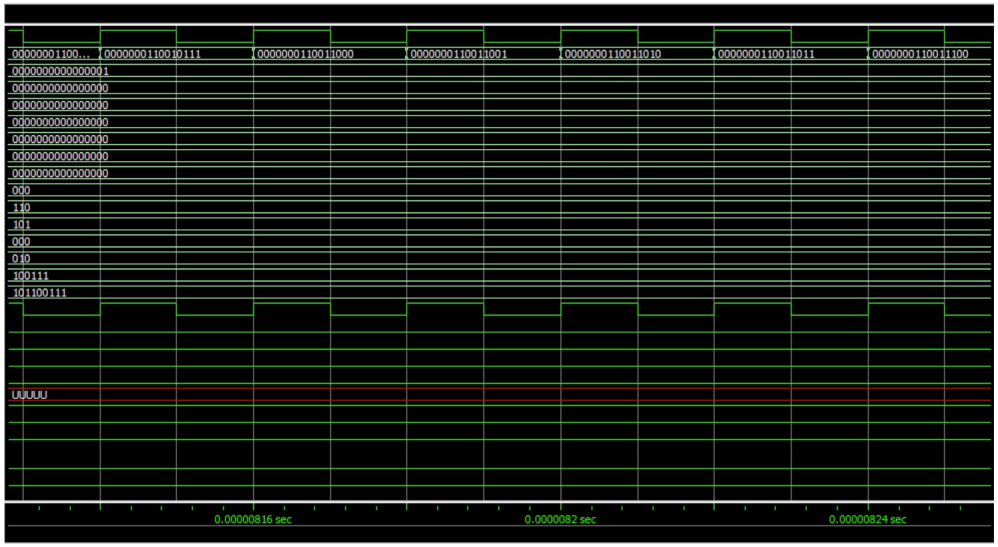


Figure 3: Waveform of series of add instruction