# EE309 Project

**Sahil Dharod**
210070026

**Pratham Kheskwani**
21D070051

**Azeem Motiwala**
210070018

**Aditya Kumar**
210070003

May 6, 2023

## Contents

## 1 Abstract

The goal of this project was to design a 6-stage pipeline processor being able to execute a set of 26 instructions that were provided to us. We also had to identify and resolve the pipeline hazards using some standard ways like Data Forwarding, stalling, and flushing. The implementation of various instructions and the design of the datapath have been described in the report below.

## 2 Stages

This pipeline had 6 different stages namely **Instruction Fetch**, **Instruction Decode**, **Register Read**, **Execute**, **Memory Access** and **Write Back**. We introduced 5 pipeline registers in between the stages to pass on data, instruction, opcode, and control bits which were required in the next stage.

### 2.1 Instruction Fetch

In this stage, we have the instruction memory and the required PC which is decided by a 4-input MUX. PC fetches the required instruction from the instruction memory and gets updated to PC+1 with an Adder. The instruction is taken out of the instruction memory at this level and sent to the Instruction Decode stage via pipeline register 1.

### 2.2 Instruction Decode

In this stage, we have a register decoder, a control decoder, and an immediate decoder. All together constitute an instruction decoder. This stage is responsible for giving the operand registers address, destination registers address, immediate values, and all the control signals for an instruction.

### 2.3 Operand Read

This stage contains the register file. Data is read from the registers using this stage and passed on to the execution stage for execution. We are using 3 read ports for reading the data from the register file and one write port for writing data into the register file. The reason for using an extra reading port is ADC instruction which adds the 2 operands when the system has a carry else it does not add the 2 operands and keeps the data of the destination unmodified. Because of this, our third reading port would contain unmodified data which will be then fed into a mux so as to select the correct output.

### 2.4 Execute

This stage contains the ALU, a counter block for executing LM-SM instructions, and a reset counter block for resetting the counter when LM-SM instructions are executed completely. This counter helps to stall the system for 8 cycles so that LM-SM instructions can be executed properly. This stage also contains sign extenders for extending 6bit values to 16bits and 9bit values to 16bits so that they can be given as inputs to Alu whenever required.

### 2.5 Memory Access

This stage contains Data memory which can be accessed by load-store instructions.

### 2.6 Register Write Back

This is the last stage of the pipeline architecture. The register file is updated in this stage.

## 3 Hazards and their Solutions

### 3.1 Hazards

- **Data Dependency hazards:** Data hazards occur when instructions that exhibit data dependence modify data in different stages of a pipeline there are three different levels of data dependency hazards. Level 1,Level 2 and Level 3 dependencies in order to solve these we use data forwarding from execute, memory and write back stages respectively.

- **Load dependency :** This hazard is a specific form of hazard in which data is being loaded from the instruction and has not yet been available, in order to solve this we use stalling i.e. we allow the load instruction to pass through and stall the other instructions by prohibiting the PC from updating itself.

- **Jump and Branch:** In case of jump and branch instructions we have to waste single or more cycles depending upon the type of the jump instruction. In order to reduce the number of cycles lost we use an additional alu so that we can do simultaneous calculation of the updated PC along with the execute stage.

- **R0 related data hazards:** There are numerous hazards such as arithmetic and load hazardd related to R0 as it is mapped to PC.

## 3.2 Solutions

### 3.2.1 Data Forwading

Data forwarding is necessary so as to solve the data hazards. We incorporated data forwarding in the Operand read stage. We will use 2, 4*1 MUXs for this. These muxes get data from EX, MEM, WB, and OR stage and select the values according to the MUX control signals. These control signals are formed by comparing the destination address from EX, MEM, and WB stages with the operand address in the OR stage. If any of these gets matched, then we forward the data of that stage to the alu for execution. This way we ensure that both the operands of ALU receive updated data for execution.

### 3.2.2 Load Immediate Dependency

A specific type of data hazard known as a load-use data hazard, which occurs when data is being loaded by a load instruction but has not yet been available when it is required by another operation. A pipeline stall, is a stall that is started in order to address a risk. We introduce stalling bits in order to resolve this hazard.

### 3.2.3 Hazard Stall unit

This unit is responsible for enabling and disabling the stall and flush signals in each pipeline register. This unit is implemented globally. It takes input as opcode and registers operand destination from all the stages and gives us stall and flush signals values accordingly.

### 3.2.4 PC Forwading

We use PC forwarding in branch and jump instructions, whenever we encounter a branch or a jump instruction we waste a minimum of a single cycle(till the instruction decode stage) in case of unconditional jumps i.e. instructions like BEQ, BLT we have to waste at least 3 cycles, till execute stage in order to determine whether we have to jump or not. We use PC forwarding from execute state directly from the 2nd alu in case of the jump instruction. In the case of conditional jump instruction, the alu is busy so we use a 3 rd alu in order to calculate the jump address. We also use forwarding of data memory into the PC mux in case of load instructions.

## 4 Datapath

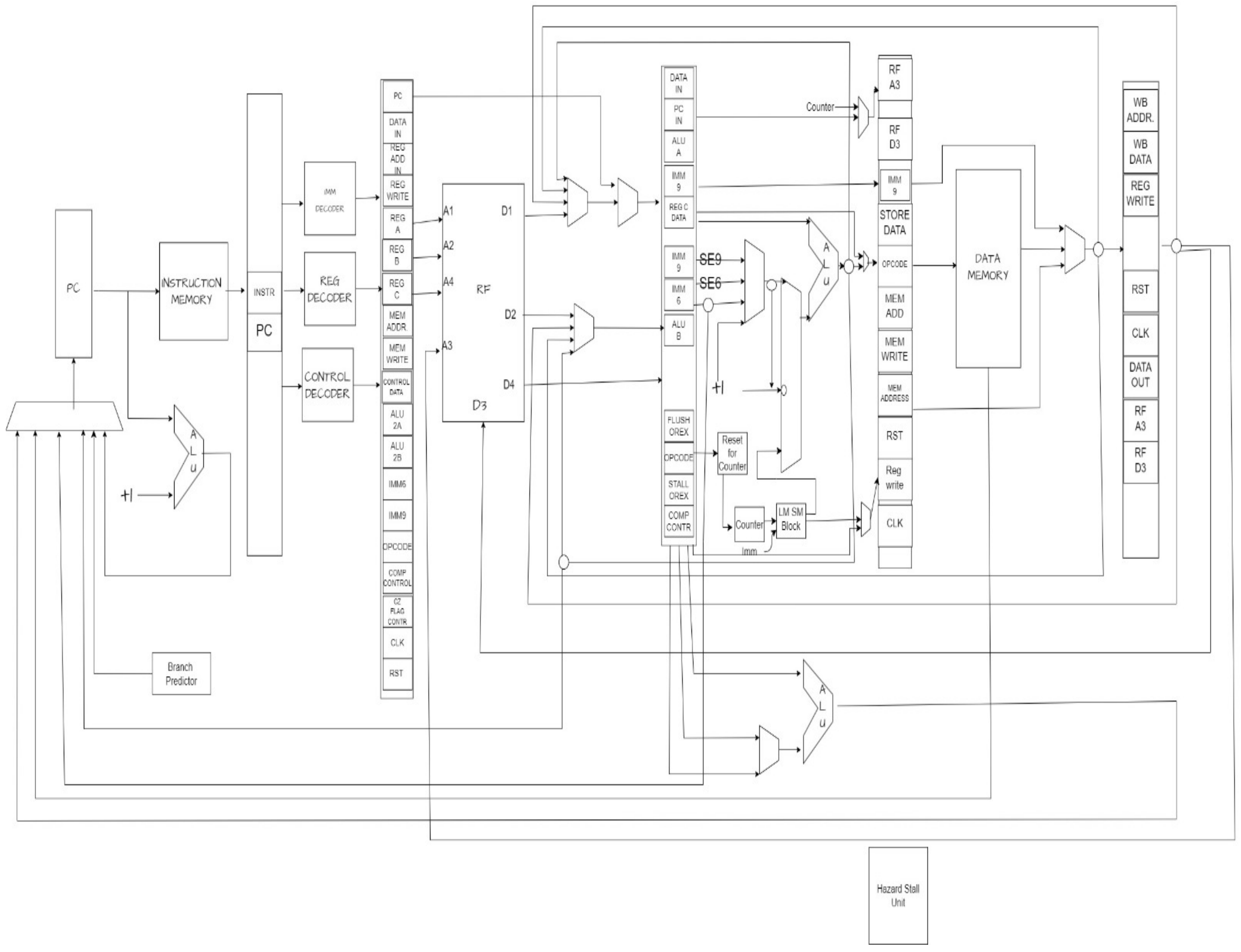

Figure 1: datapath

## 5 Branch Predictor

We are using a static branch predictor which is not-taken. That is we always assume that we are not going to jump and take the next consecutive instruction. Then we check whether our prediction is correct or not in the execution stage and branch accordingly.

### Modification Required for making it a one-bit dynamic branch predictor

- PC regular is now (PC+1 muxed with branch predicted)control being opcode.

- branch predictor gets input by matching predicted and outputex for branch instructions.