

# Intro to Processor Architecture

## Project Report

### Team – PHOENIX

Team Members – Prasoon Garg (2020102049)

Prashant Gupta (2020102030)

## Overview

The main goal of this project is to use Verilog to construct a processor architecture based on the Y86-64 ISA. All instructions in the Y86-64 ISA should be able to be executed by the processor. Modularity is required in the design process. The objective is to create a pipelined Y86-64 implementation with five stages.

## Module Description

### a) Fetch

- The fetch stage includes the instruction memory hardware unit, this unit reads instruction from the instruction memory.
- During the implementation of the fetch stage, it gets the updated PC value on the positive edge of clock signal and checks for the instruction in the instruction memory and if PC value gets out of bound then mem\_error gets generated.
- The icode and ifun values can be obtained by splitting the first byte of the instruction\_memory from the instruction register which is represented by the split block in the architecture diagram.
- The values of regA, regB and valC can be obtained from instruction\_memory depending on the value of icode which is represented as align block in the architecture diagram.

- The value of valP varies according to the value of icode and when the value of the icode is invalid then the instruct\_valid is set to 0.

## **b) Decode and Write back**

- The register file has 4 different ports (dstM, dstE, srcA, srcB) which have the values as valM, valE, valA, valB out of which only 2 can be used at a time i.e., either 2 reads or 2 writes. Read occurs by srcA and srcB while write occurs by dstM, dstE and same can be observed from the diagram of Decode and write back below.
- Now since only either read or write can happen simultaneously hence we can use only 2 variables regA and regB to store the register address.
- The values of valA and valB are present in the registers given by address regA and regB . We use an regA and regB and index and reference the register\_mem register array for valA and valB .
- Here register\_mem[4] is the rsp register which holds the stack pointer and is initially declared as 1001<sup>th</sup> position in the memory.
- Depending on the icode the corresponding register are addressed and the value is either read or written.

## **c) Execute**

- The arithmetic/logic unit is included in the execute step (ALU). Based on the setting of the ifun signal, this unit executes the operations add, subtract, and, or exclusive-or on the inputs valA and valB. Three control blocks create the data and control signals.
- The condition code register is likewise included in the execute stage. Every time our execute runs, it creates the three signals that the condition codes are based on: zero, sign, and overflow.
- However, the condition codes should only be set when an OPq instruction is performed. As a result, we emit the signal set cc, which determines if the condition code register should be changed.

- To decide whether a conditional branch or data transfer should take place, the hardware unit labelled "cond" employs a mix of condition codes and function codes. It creates the Cond signal, which is utilised in the following PC logic for conditional branching as well as for setting dstE with conditional movements.
- The Cond signal for other instructions may be set to 1 or 0, depending on the function code of the instruction and the condition code settings.

#### **d) Memory**

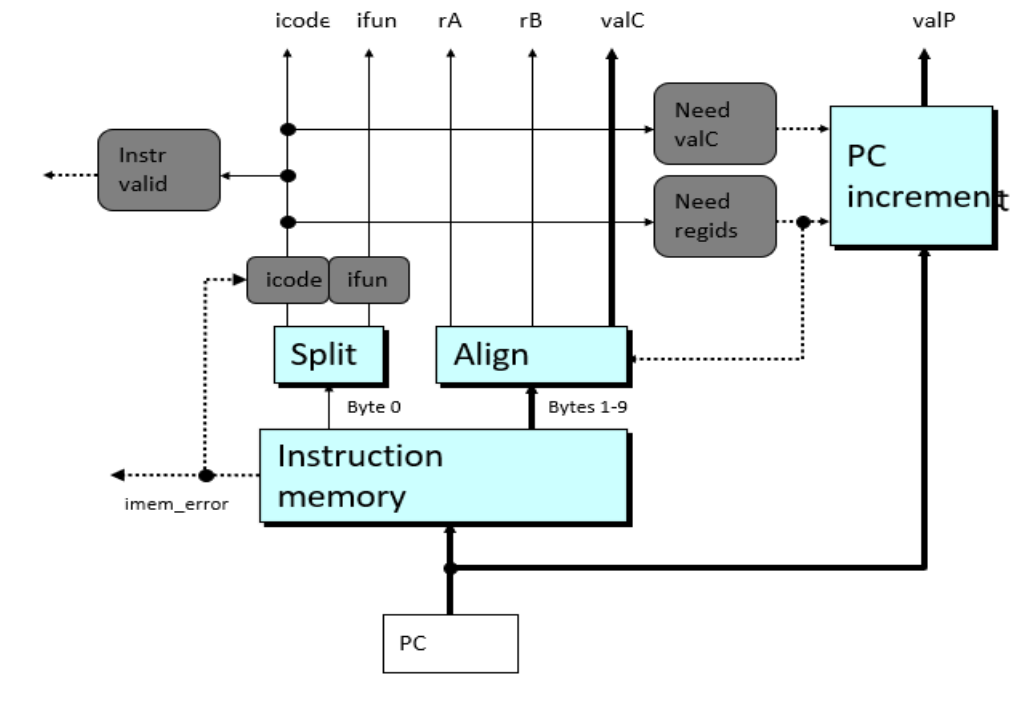
- The memory stage is the stage where the reading and writing to the memory block takes place.
- The memory block defined is a 64 bit 1024 deep array.
- The value of icode decides whether the memory stage performs the read operation or the write operation.
- The memory also stores the stack from the 1001th position which are used for the return push and pop instruction.
- When read is performed then the value is stored in the valM output register.

#### **e) PC update**

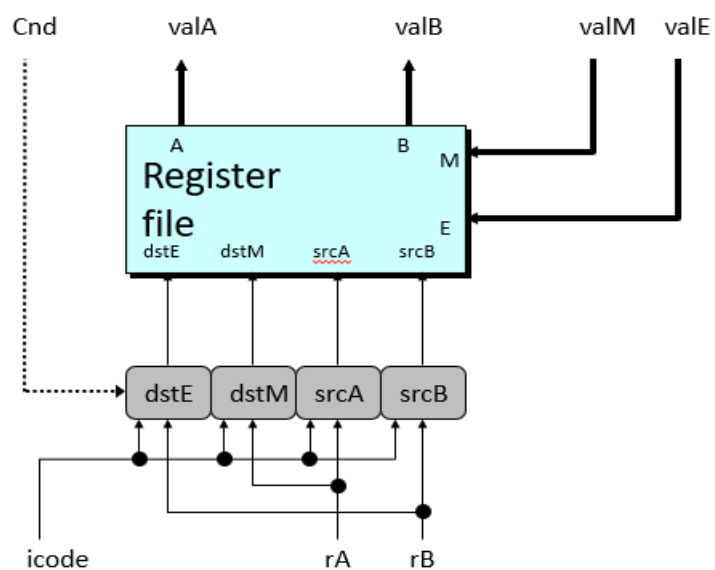
- PC update stage is where the value of the program counter is updated to accept the next instruction.
- The value of PC gets updated differently for different values of icode and does not increment when the instruction is a jump or return instruction since there the code waits for the write back stage to get the value of valM.

## Module Diagrams

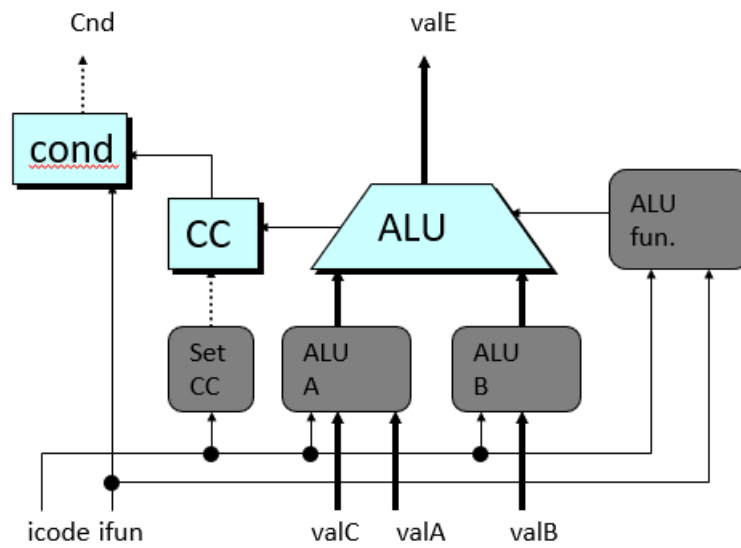
### a) Fetch



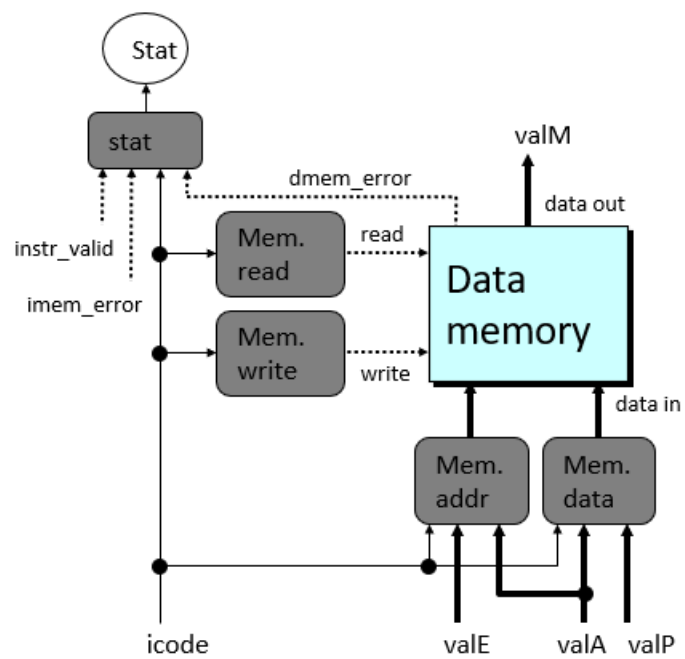
### b) Decode and Write back



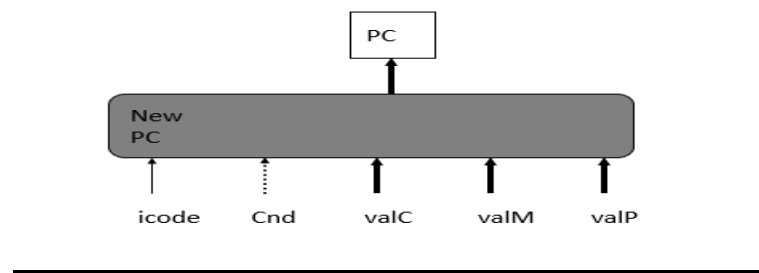
**c) Execute**



#### **d) Memory**



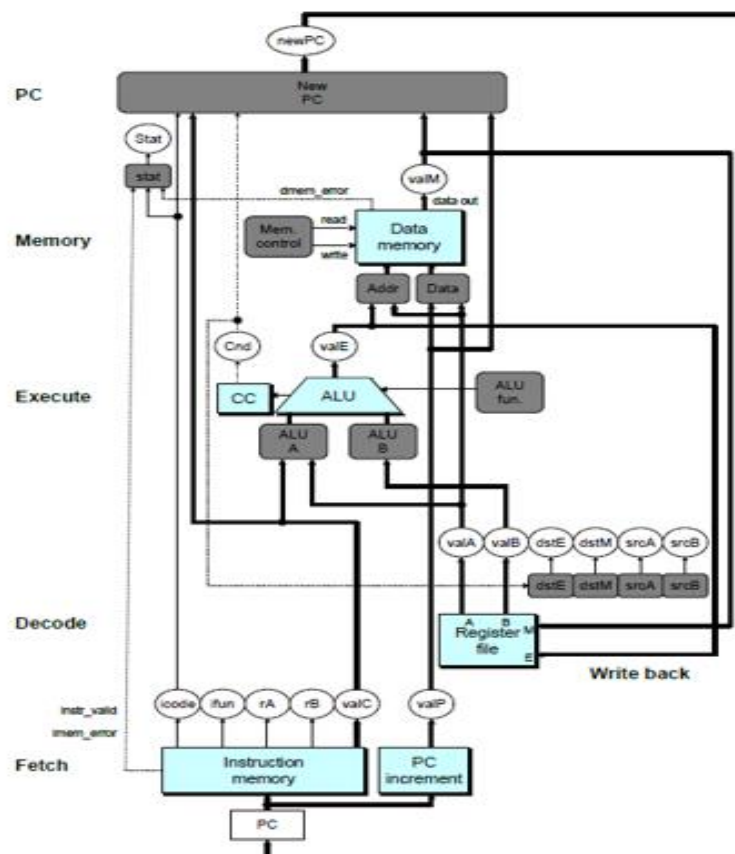
### e) PC update



## Sequential Processor Description

Now we have the components required to implement a Y86-64 processor. As a first step, we describe a processor called SEQ (for “sequential” processor). On each clock cycle, SEQ performs all the steps required to process a complete instruction. This would require a very long cycle time, however, and so the clock rate would be unacceptably low.

The implementation and description of all the modules used in sequential processor are given above. The basic diagram for the sequential processor is as shown and all the codes have been written almost in accordance to this diagram.



## Pipelined Processor Description

The first step to pipelining is the rearrangement of computation stages as PC Update stage is last in the SEQ implementation whereas in the pipelined processor it is used in the first step.

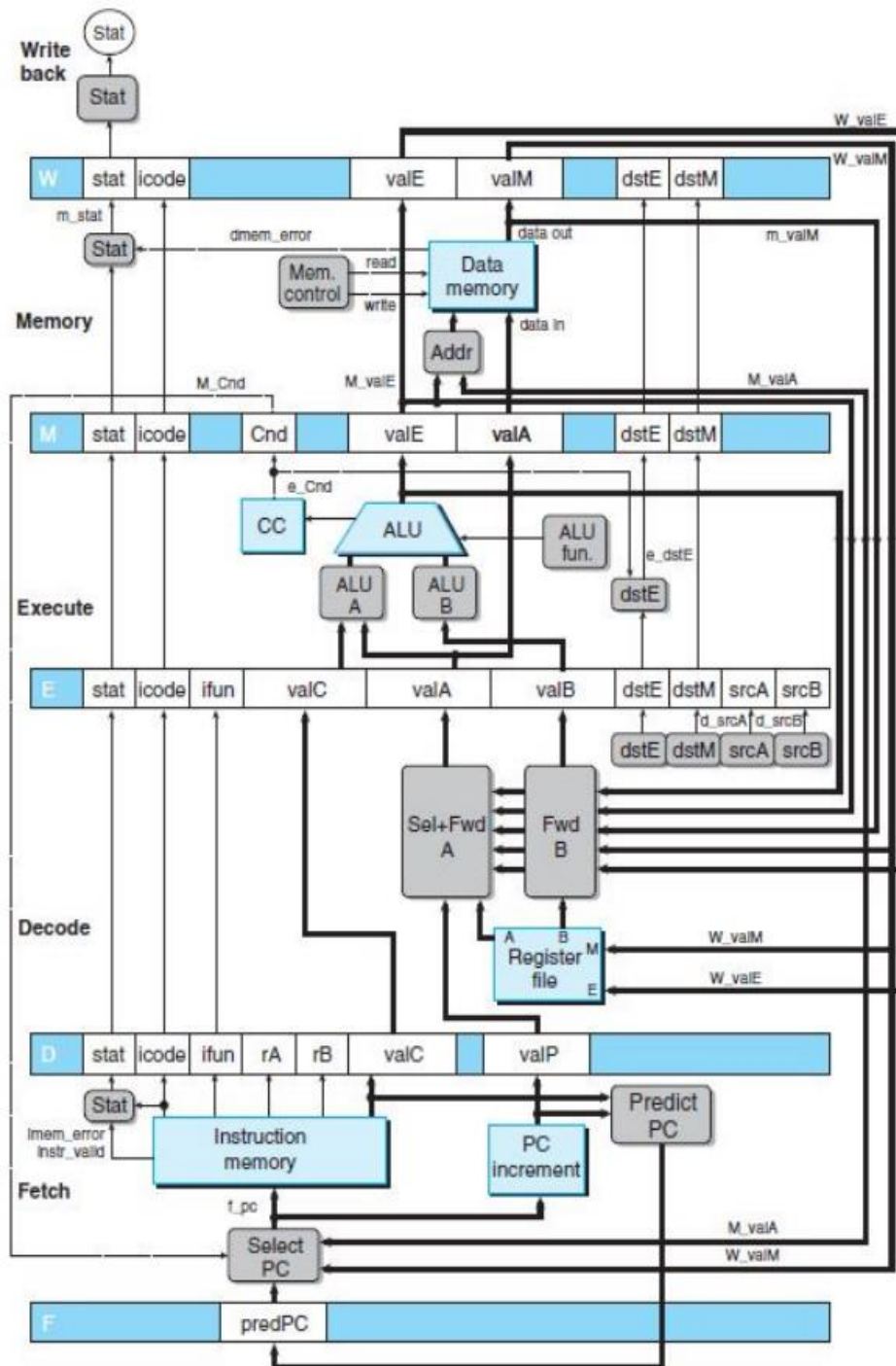
We should move the PC update stage to the beginning of the cycle for the pipelined implementation because we want to be able to acquire the next instruction without having to wait for the PC update stage of the previous instruction to finish, which would be the case if it were at the end. Circuit retiming is the term for it.

Using the needed values from different stages from instructions that have passed that stage, the PC update stage at the start of the cycle may now continue to provide updated PC values to the fetch stage.

The next step to pipelining is inserting the pipeline registers. We reorganise parts of the hardware and signals in the SEQ implementation and put pipeline register between each stage in a pipelined implementation. These registers prevent signals from one stage from leaking into the next, interfering with the processing taking place there.

The next step is to relabel the signals according to their stages for example d\_icode, f\_icode and so on respectively for decode and fetch stage registers.

Below is the diagram for the implementation of the pipelined processor with 5 stages.





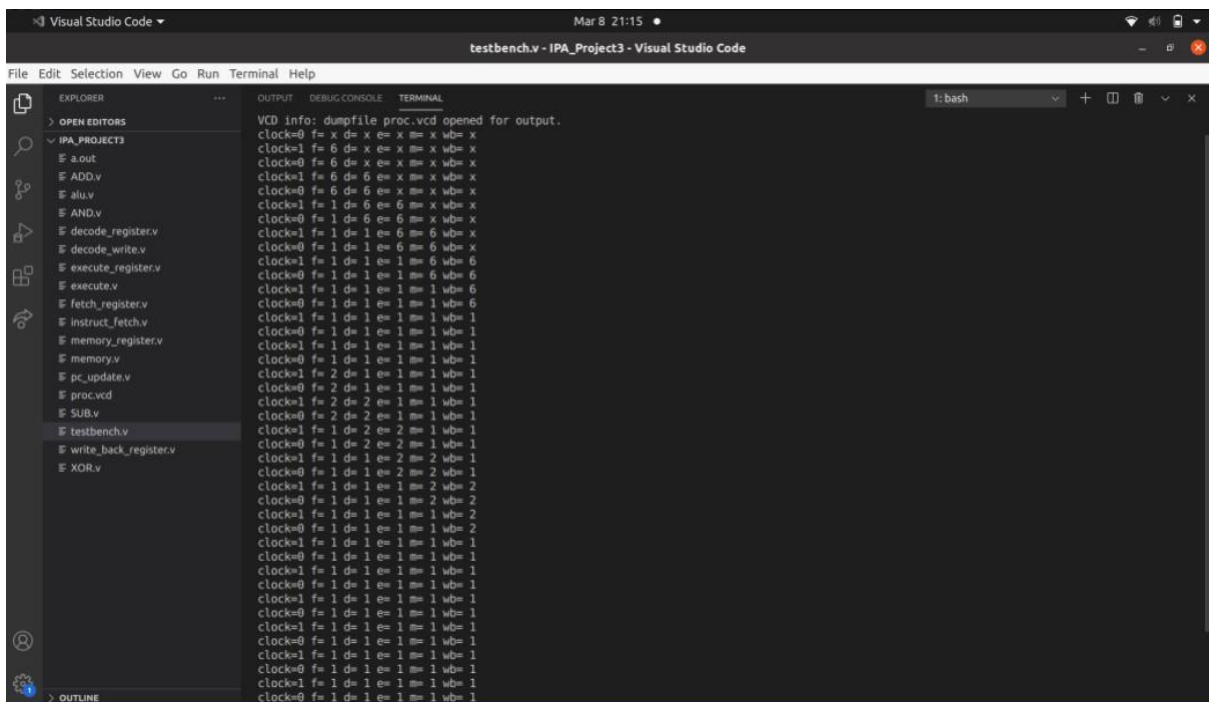
## Instruction supported

The processor supports all instructions in the Y86-64 instruction set.

Instruction	What does it do?
IHALT	Code for halt instruction
INOP	Code for nop instruction
IRRMOVQ	Code for rrmovq (register to memory) instruction
IIRMOVQ	Code for irmovq (immediate to register) instruction
IRMMOVQ	Code for rmmovq (register to memory) instruction
IMRMOVQ	Code for mrmovq (memory to register) instruction
IOPL	Code for integer operation instructions
IJXX	Code for jump instructions
ICALL	Code for call instruction
IRET	Code for ret instruction
IPUSHQ	Code for pushq instruction
IPOPQ	Code for popq instruction



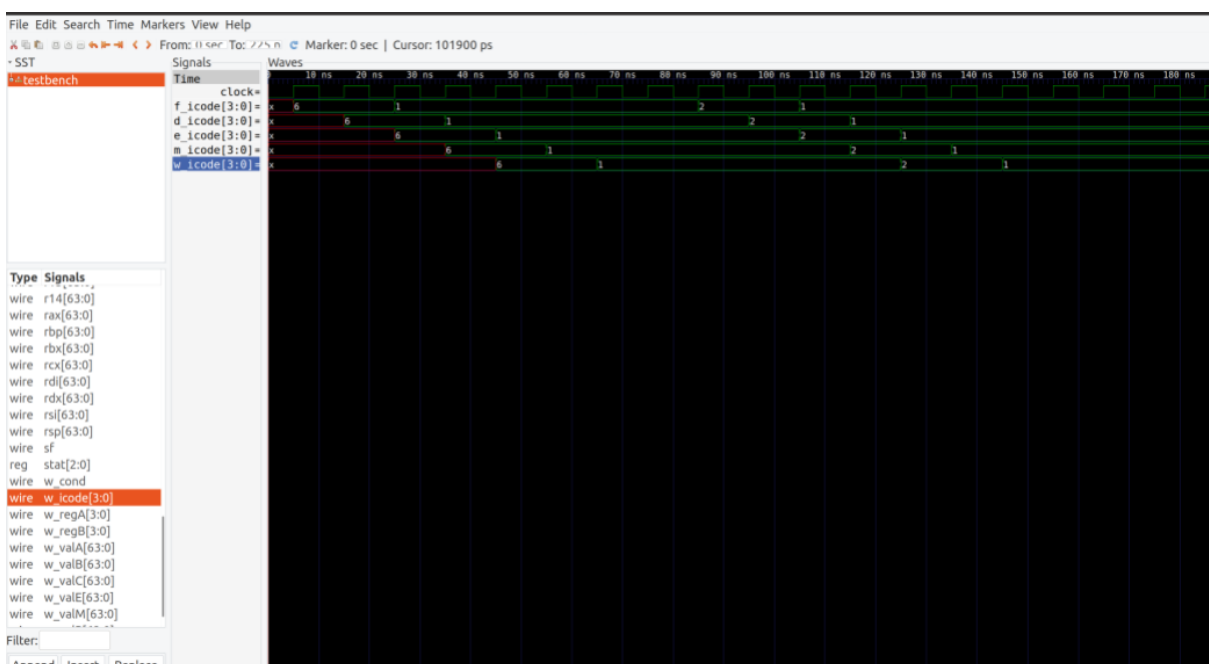
3. The terminal output for the fetch stage implementation of the pipeline processor with inputs same as in the case of sequential processor.



The screenshot shows the Visual Studio Code interface with the terminal window open. The terminal displays the output of a VCD file named 'proc.vcd'. The output shows a sequence of clock cycles (clock=0, clock=1) and the values of various signals (f, d, e, m, w) in hexadecimal. The signals are listed as follows:

```
VCD info: dumpfile proc.vcd opened for output.
clock=0 f= x d= x e= x m= x w= x
clock=1 f= 6 d= x e= x m= x w= x
clock=0 f= 6 d= x e= x m= x w= x
clock=1 f= 6 d= 6 e= x m= x w= x
clock=0 f= 6 d= 6 e= x m= x w= x
clock=1 f= 1 d= 6 e= 6 m= x w= x
clock=0 f= 1 d= 6 e= 6 m= x w= x
clock=1 f= 1 d= 1 e= 6 m= 6 w= x
clock=0 f= 1 d= 1 e= 6 m= 6 w= x
clock=1 f= 1 d= 1 e= 1 m= 6 w= 6
clock=0 f= 1 d= 1 e= 1 m= 6 w= 6
clock=1 f= 1 d= 1 e= 1 m= 1 w= 6
clock=0 f= 1 d= 1 e= 1 m= 1 w= 6
clock=1 f= 1 d= 1 e= 1 m= 1 w= 1
clock=0 f= 1 d= 1 e= 1 m= 1 w= 1
clock=1 f= 1 d= 1 e= 1 m= 1 w= 1
clock=0 f= 1 d= 1 e= 1 m= 1 w= 1
clock=1 f= 2 d= 1 e= 1 m= 1 w= 1
clock=0 f= 2 d= 1 e= 1 m= 1 w= 1
clock=1 f= 2 d= 2 e= 1 m= 1 w= 1
clock=0 f= 2 d= 2 e= 1 m= 1 w= 1
clock=1 f= 1 d= 2 e= 2 m= 1 w= 1
clock=0 f= 1 d= 2 e= 2 m= 1 w= 1
clock=1 f= 1 d= 1 e= 2 m= 2 w= 1
clock=0 f= 1 d= 1 e= 2 m= 2 w= 1
clock=1 f= 1 d= 1 e= 1 m= 2 w= 2
clock=0 f= 1 d= 1 e= 1 m= 2 w= 2
clock=1 f= 1 d= 1 e= 1 m= 1 w= 2
clock=0 f= 1 d= 1 e= 1 m= 1 w= 2
clock=1 f= 1 d= 1 e= 1 m= 1 w= 1
clock=0 f= 1 d= 1 e= 1 m= 1 w= 1
clock=1 f= 1 d= 1 e= 1 m= 1 w= 1
clock=0 f= 1 d= 1 e= 1 m= 1 w= 1
clock=1 f= 1 d= 1 e= 1 m= 1 w= 1
clock=0 f= 1 d= 1 e= 1 m= 1 w= 1
clock=1 f= 1 d= 1 e= 1 m= 1 w= 1
clock=0 f= 1 d= 1 e= 1 m= 1 w= 1
clock=1 f= 1 d= 1 e= 1 m= 1 w= 1
clock=0 f= 1 d= 1 e= 1 m= 1 w= 1
clock=1 f= 1 d= 1 e= 1 m= 1 w= 1
```

4. The GTKwave output for the above terminal output. We can see from the below diagram that the instruction value forwards to the next stage and hence the pipeline implementation is correct.



5. The below is the GTKwave output of pipeline processor for the same instruction as sequential processor and here as well we get the same and correct output as in case of sequential. (Here numbers are in decimal)



## Instructions to run

- Sequential and Pipeline
  - a) Run the command iverilog testbench.v
  - b) Then 2 files will be generated out.vcd and a.out
  - c) Run the ./a.out for the output on the terminal and gtkwave out.vcd to obtain the plot.

## Challenges Faced

1. The first challenge was to understand the implementation of various modules present in the sequential processor design.
2. The second challenge was to integrate all these modules into one testbench which was pretty difficult.

3. Third was the implementation of the Pipeline processor since it required a lot of extra register and register files and control logics so that instructions pass onto the next stage.