Implementation of 6-Stage Pipelined RISC Processor

EE-739 Processor Design

Gunanjan Reddy K - 23M1148
Partho Sarkar - 23M1161
Abishek R - 23M1185

Supervised by: **Virendra Singh**

Electrical Department
Indian Institute of Technology Bombay, Powai

Contents

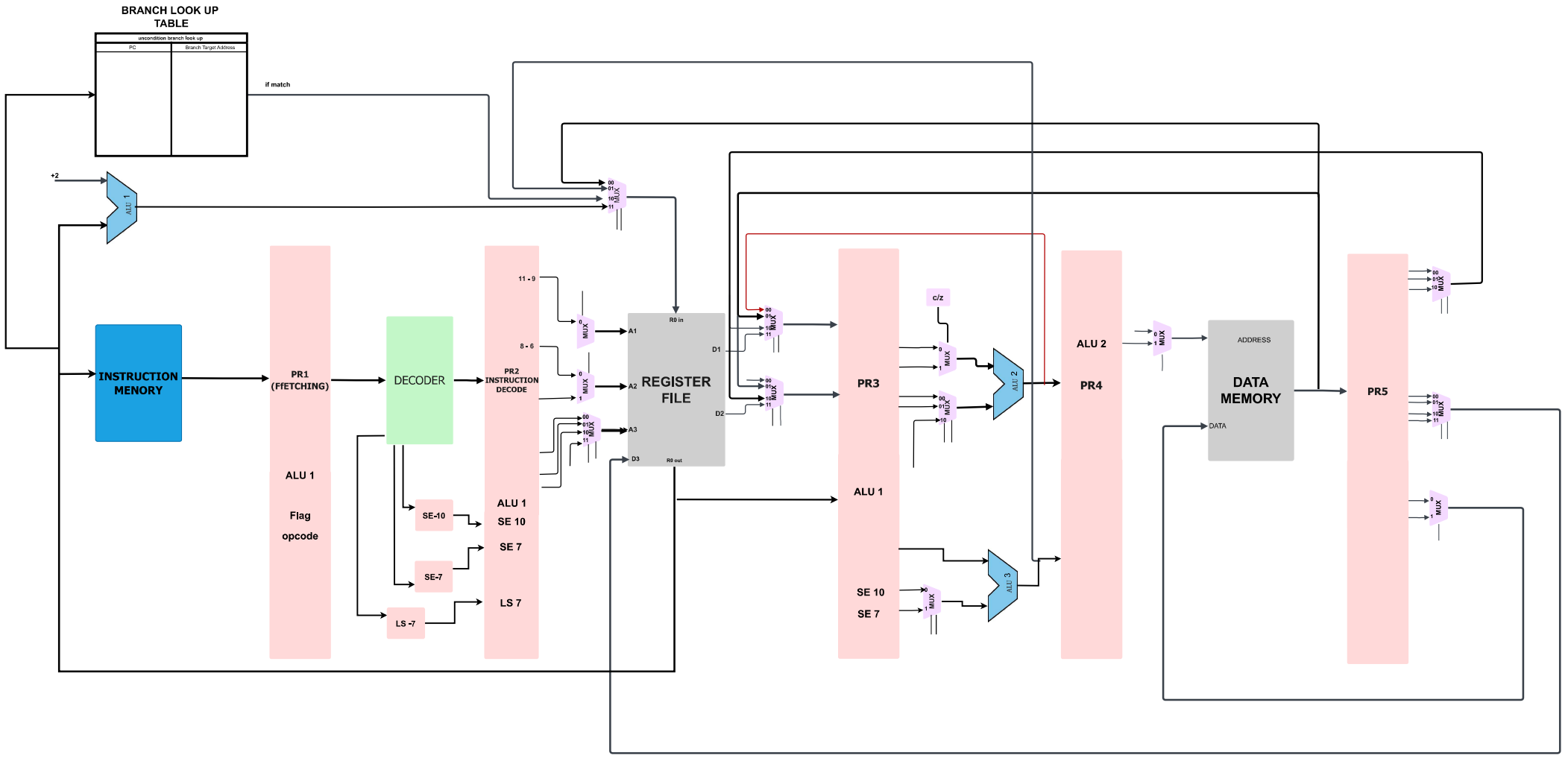
1. Simulation Approach:

Our Verilog code represents a multi-stage pipeline processor with six stages: Instruction Fetch, Instruction Decode, Operand Read, Execute, Memory Access, and Write Back. The Verilog file utilizes a pipelined design with hazard detection and data forwarding to optimize performance and maintain system reliability.

The processor is designed as follows:

- **Simulation Approach:** The Verilog file employs a single module to implement the pipeline design. Output from each stage is stored in a register and used as input for the next stage on subsequent clock cycles. Hazard detection techniques resolve potential data, control, and structural hazards within the pipeline, ensuring smooth data flow and timing between stages.
- **Components:**
  - **Initial Setup:** Initializes the instruction and data memory from external files.
  - **Instruction and Data Memory:** 'instr_mem' stores instructions, while 'data_mem' stores data. These memories are loaded from external files ('instr_mem.hex' and 'data_mem.hex').
  - **Pipeline Registers:** Several pipeline registers ('PR1' through 'PR5') hold intermediate values and control signals as they flow through the pipeline.
  - **Register File:** A collection of registers ('reg_array') used for storing data and control values during computation.

    The register file is read from ('RF_read_data_1' and 'RF_read_data_2') and written to ('RF_write_dest', 'RF_write_data') based on control signals.
  - **Control Signals:** Control signals such as 'reg_write_en', 'RF_write_en', and others guide the flow of data and operations in the processor.
  - **Flags and Control Values:** Flags ('PR3_CARRY' and 'PR3_ZERO') and other control values ('PR2_alu_ctrl', 'PR2_nand_ctrl', etc.) play a crucial role in determining how operations are performed.
- **Pipeline Stages:**
  - **Instruction Fetch:** Fetches instructions from the instruction memory using the program counter (PC) and stores them in pipeline register PR1.
  - **Instruction Decode:** Decodes instructions from PR1 and generates control signals for the arithmetic logic unit (ALU) and NAND unit, and determines which registers are used. It stores the decoded information in pipeline register PR2.
  - **Operand Read:** In this stage, operands and control signals are fetched from the previous stage. The data and control signals are stored in PR3.
  - **Execute:** Executes ALU and NAND unit operations based on control signals received from PR2 and operands from the register file. Results and status flags (e.g., carry and zero) are stored in pipeline register PR3.
  - **Memory Access:** Handles memory operations such as loading and storing data based on the results from the previous stage. Updates the data in PR4, PR5, and data memory as necessary.
  - **Write Back:** Writes computed results back to the register file or updates the program counter based on results and operations from the previous stages.


BRANCH LOOK UP TABLE
unconditional branch look up
PC
Branch Target Address
if match
+2
ALU 1
INSTRUCTION MENORY
PR1 (FFETCHING)
ALU 1
Flag
opcode
DECODER
SE-10
SE-7
LS -7
PR2 INSTRUCTION DECODE
11 - 9
8 - 6
ALU 1
SE 10
SE 7
LS 7
MUX
R0 in
A1
A2
A3
D3
R0 out
REGISTER FILE
D1
D2
PR3
ALU 1
SE 10
SE 7
c/z
ALU 2
ALU 3
ALU 2
PR4
ADDRESS
DATA MEMORY
DATA
PR5

3. Test cases

3.1. Results for test conditions.

3.1.1. case 1 .

Instruction :

```
ADA R1, R2, R3
ADA R2, R1, R3
```

Here in the above instructions we are using 'R1' as an operand before it is updated with the computed value in the previous value so, it is a condition for hazard.

In order to prevent this we used forwarding from the output of the execution to the operand fetch state. so we are getting this 'R1' before updating after write-back stage.

This way we are overcoming the hazard by using the forwarding technique.
This can be observed in the graph where PR4_result is forwarded to the PR_3 operand.

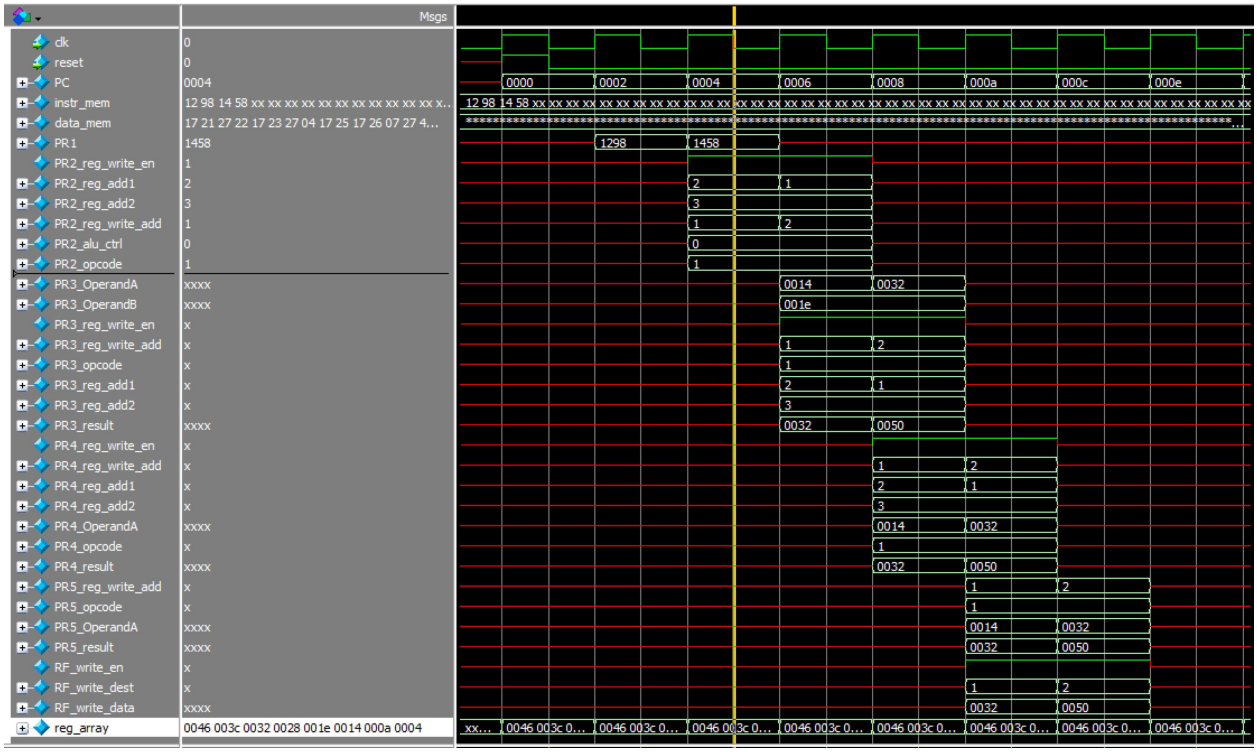


Figure 2: output wave form for case 1

3.1.2. case 2 .

Instruction :

```
ADA R1, R2, R3
ADA R4, R5, R6
ADA R2, R3, R1
```

There is dependency in 'n' th instruction and 'n-2' instruction r1 needs to be updated in third instruction after calculating it's value from the 1st instruction.

so we used data forwarding from the memory access PR4 to the PR2 register.

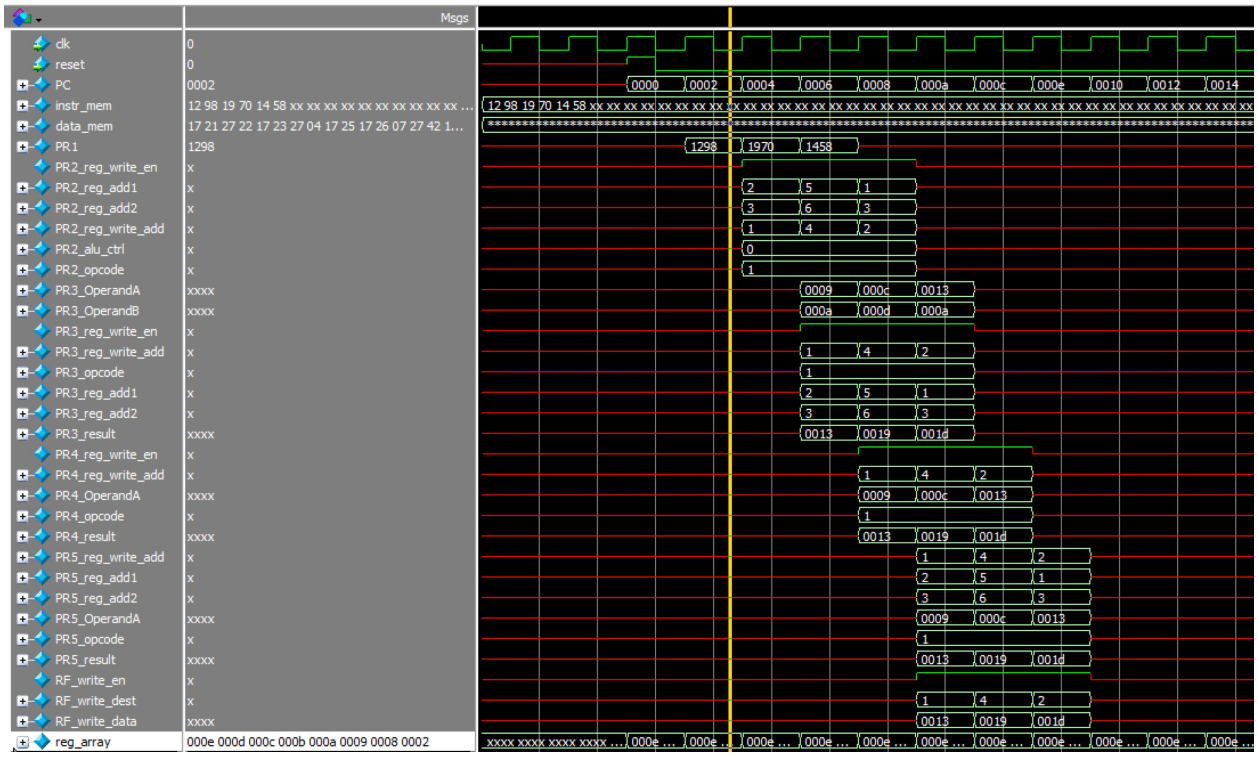


Figure 3: output wave form for case 2

3.1.3. case 3 .

Instruction for 32 bit addition :

```
LLI  R1,00H
LW  R4,R1,04H
LW  R3,R1,02H
LW  R5,R1,06H
LW  R2,R1,00H

ADA  R6,R2,R4
AWC  R7,R3,R5

LLI  R2,00H
AWC  R4,R1,R2
SW  R6,R1,0CH
SW  R7,R1,0AH
SW  R4,R1,08H
```

Here in this test cases we are doing 32 bit addition.
Initially, we are loading 'R4', 'R5', 'R2', 'R3', 'R6', 'R7'. Where 'R2' and 'R4' are the 16 bit lSB's of operands for addition and sum result is stored in R6 and carry flag is modified.

remaining 16 bit 'R3' and 'R5' are MSb's of operands for addition, both are added with previous carry and result is stored in 'R7'.

Later results are stored in data memory.

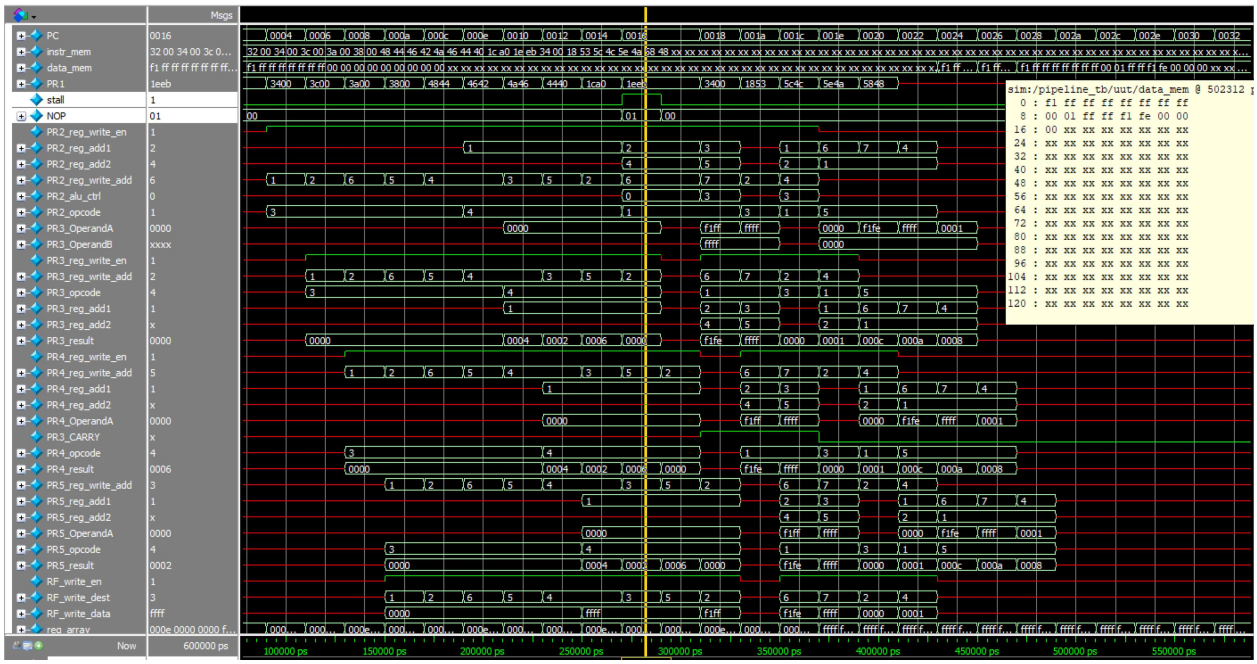


Figure 4: output wave form for case 3

3.1.4. case 4 .

Instruction: STORE IMMEDIATE

```
20  LLI  R7, 00H
21  LLI  R1, 01H
22  LLI  R5, 05H
23  LLI  R4, 00H
24  LLI  R2, 00H
25  LLI  R6, 00H
26  LLI  R3, 00H
27
28  ADA  R4, R1, R5
29  ADA  R7, R1, R5
30
31  SW   R4, R2, 00H
```

In this instruction, we are loading Immediate data into registers and performing addition operations and results are stored in registers, and these results are then stored in the data memory.

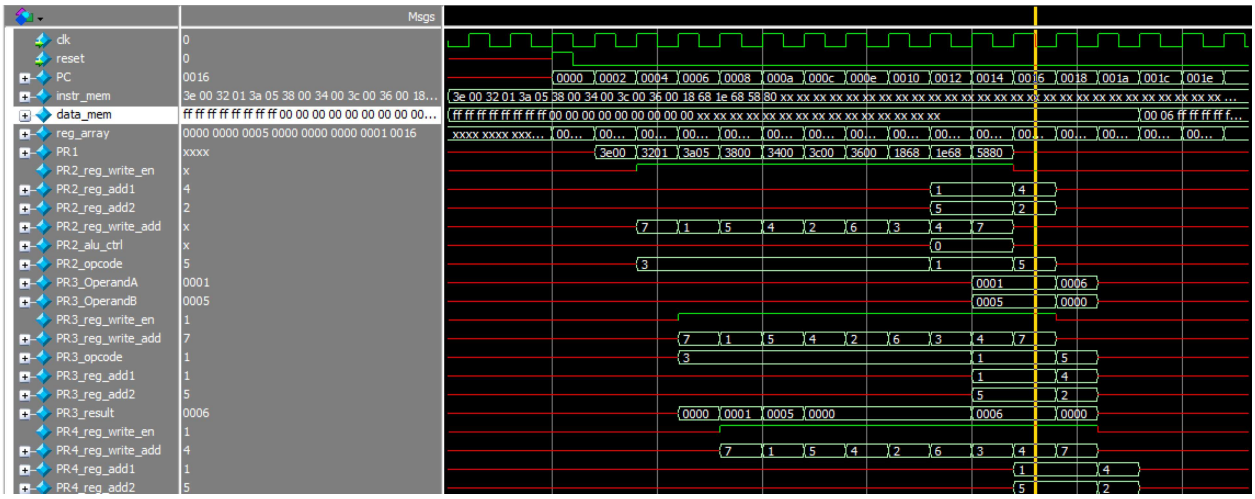


Figure 5: output waveform for case 4

3.1.5. case 5 .

Instruction: MEMORY ACCESS

```
LLI  R2,  00H
LLI  R4,  00H
LLI  R5,  05H
LLI  R6,  00H
LLI  R7,  04H

LW  R1,R2,00H
LW  R3,R2,02H

ADA  R4,  R1,  R3
ADA  R5,  R7,  R3

SW   R4,  R2,  08H
SW   R5,  R2,  0AH
```

Load word and store word operations are performed in this instruction.

Initially we perform loading of immediate data into registers(we loaded our registers with 00H) for easy calculation of the address needed to access memory in load word.

after loading we perform an add operation on the data and store the results in the data memory.

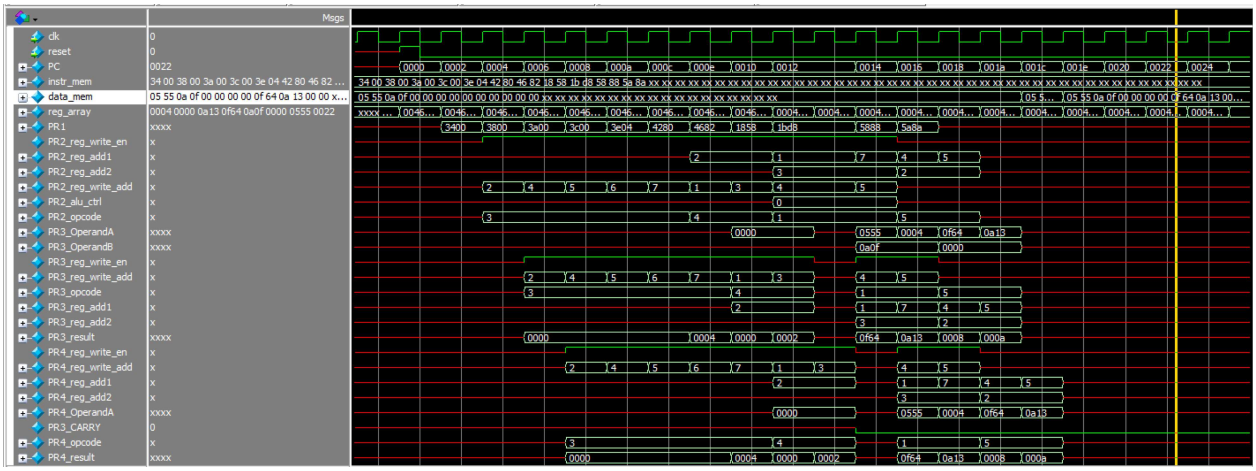


Figure 6: output waveform for case 5

3.1.6. case 6 .

Instruction:

```
46 LLI  R3,00H
47 ADA  R6,R3,R4
```

Immediate data is loaded into the 'R3' register.

Then an addition operation.

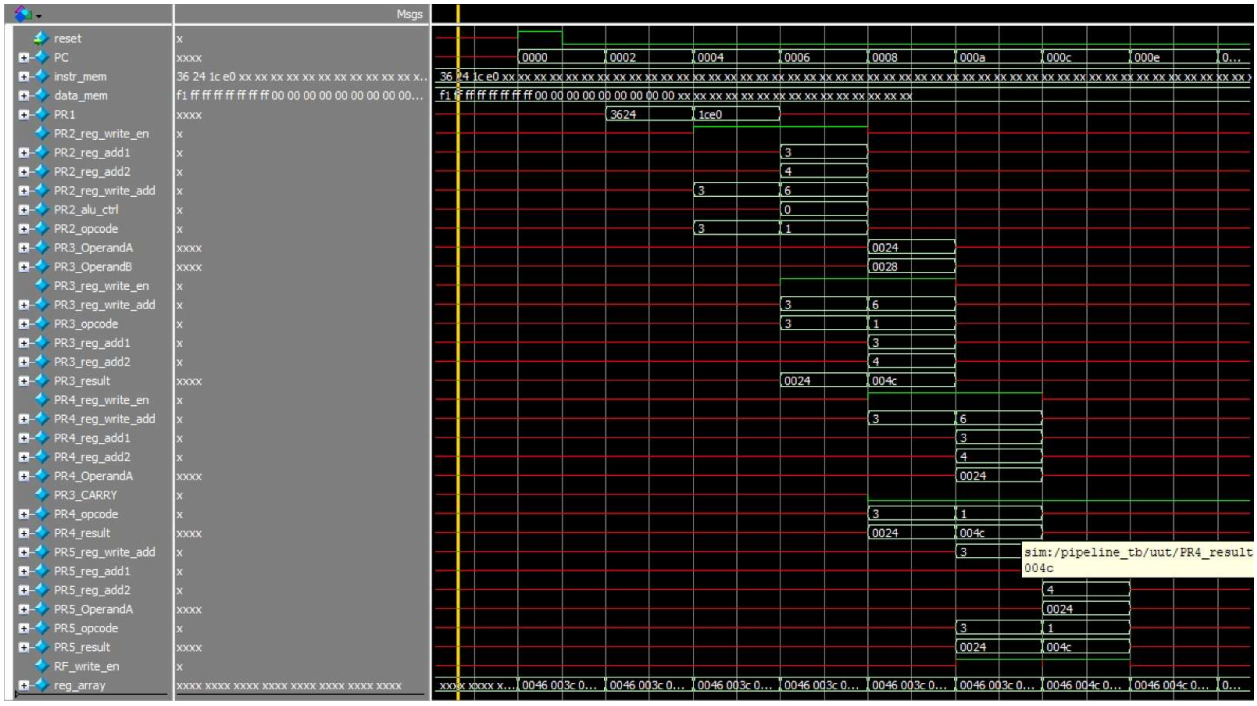


Figure 7: output waveform for case 6

3.1.7. case 7 .

Instruction:

```
48 ADA R1,R2,R3
49 LW R4,R2,00H
50 BEQ R4,R1,04H
```

In this instruction, there are multiple dependencies. we added two registers 'R2' and 'R3' and result is stored in the 'R1' register.

after that 'R4' is loaded with data from the address calculated.

then branching to the location if both the operands 'R4' and 'R1' are equal whose values are depended on previous instructions.

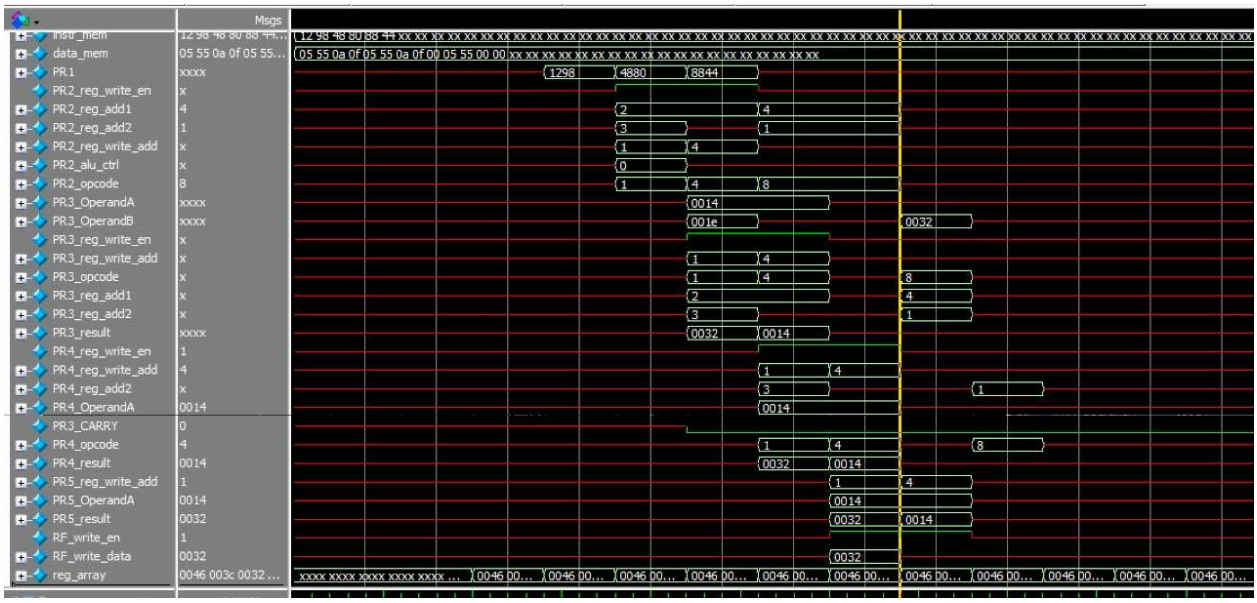


Figure 8: output waveform for case 7

3.1.8. case 8 .

Instruction:

```
ADA  R1,R2,R1
JRI  R1,02H
```

Here data dependency is present with the JUMP instruction(uncondition jump).

we are adding data of two registers whose results is required for the computation of address where we need to jump.

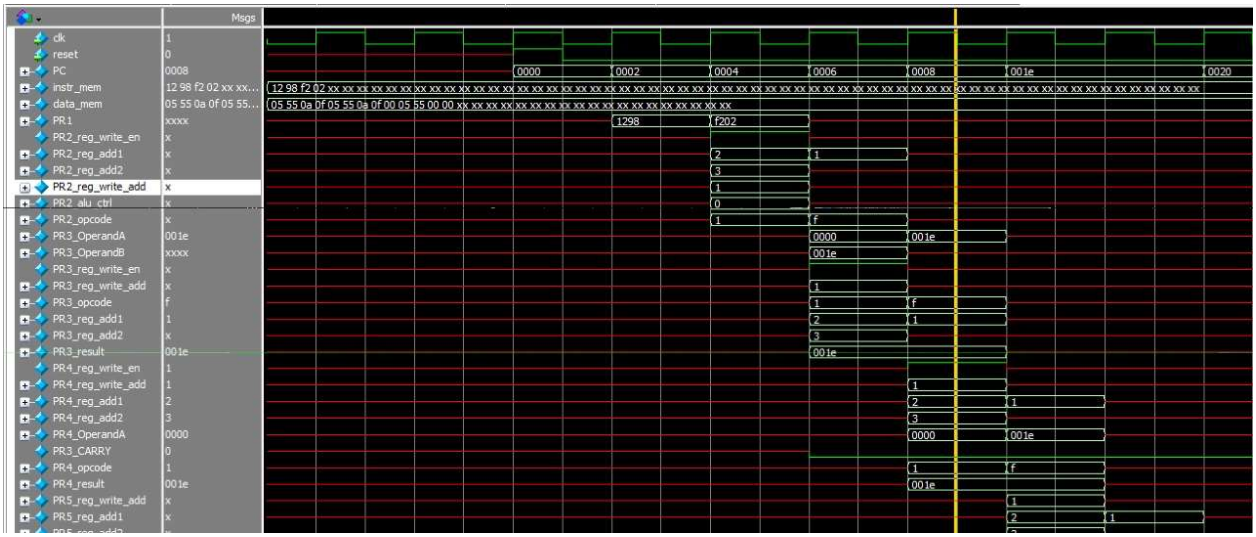


Figure 9: output waveform for case 8

4. Conclusion

In summary, our 6-stage pipelined processor has been thoroughly tested, encompassing a range of instruction sets, including unconditional jumping. Moreover, we've diligently addressed hazards by implementing forwarding conditions, ensuring smooth execution and minimizing stalls within the pipeline.

However, despite these achievements, our processor currently lacks branch prediction, a feature vital for optimizing performance in handling conditional branches. While our design is robust and capable, integrating branch prediction mechanisms would further elevate its efficiency, particularly in scenarios involving conditional branching.

In conclusion, while our processor exhibits reliability and functionality, there's still room for enhancement. Integrating branch prediction alongside our existing features will ensure that our design remains competitive and aligned with modern processor standards.

5. work contribution

Gunanjan Reddy K - 23M1148
Implementation of branch prediction table, Memory access stage, Write Backstage, testing of instructions.

Partho Sarkar - 23M1161
Implementation of Operand stage, Execution stage, Data Forwarding: Data dependencies, Testing of instructions.

Abishek R - 23M1185
Implementation of instruction fetch stage, Instruction decode stage, Data Forwarding: data dependencies.

6. Future Scope

Looking ahead, the next crucial step for our processor's evolution lies in incorporating branch prediction. By doing so, we can significantly enhance its efficiency, particularly in scenarios involving conditional branching. This improvement would reduce pipeline stalls and further optimize performance.

Furthermore, there's ample scope for future enhancements beyond branch prediction. Potential avenues include exploring more advanced pipeline architectures, optimizing memory access, and integrating parallel processing techniques.

Additionally, advancements in hardware acceleration and energy efficiency could be pursued to keep pace with evolving computing requirements.

In conclusion, while our processor design demonstrates reliability and functionality, there's an exciting roadmap ahead for its development. By integrating branch prediction and exploring other avenues for improvement, we can ensure that our processor remains competitive and aligned with the ever-changing landscape of computing technology.