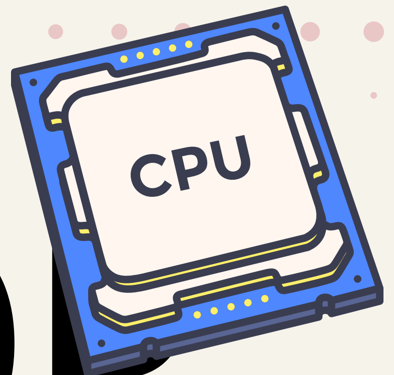
8 BIT MICROPROCESSOR



Presented By : Team 9

Team leader: Aaradhya Sharma

Member: Digvijay Pundir

Member: Aakarsh Atluri

Member: Rucha Gadgil

INTRODUCTION

Our project presents a custom-designed 8-bit microprocessor built around a RISC-inspired architecture. The processor executes a focused set of essential instructions covering arithmetic, logic, memory access, and control operations. The design emphasizes simplicity, modularity, and extensibility, making it optimized for clarity and ease of understanding.

CORE SPECIFICATIONS

Data width	8 Bits
Instruction width	16 Bits
Register File	8 general 8 bit registers
Memory Model	Separate Instruction and data memory
Hazard Support	Data forwarding ,stalling ,flushing
Pipeline Stages	5(IF,ID,EX,MEM,WB)

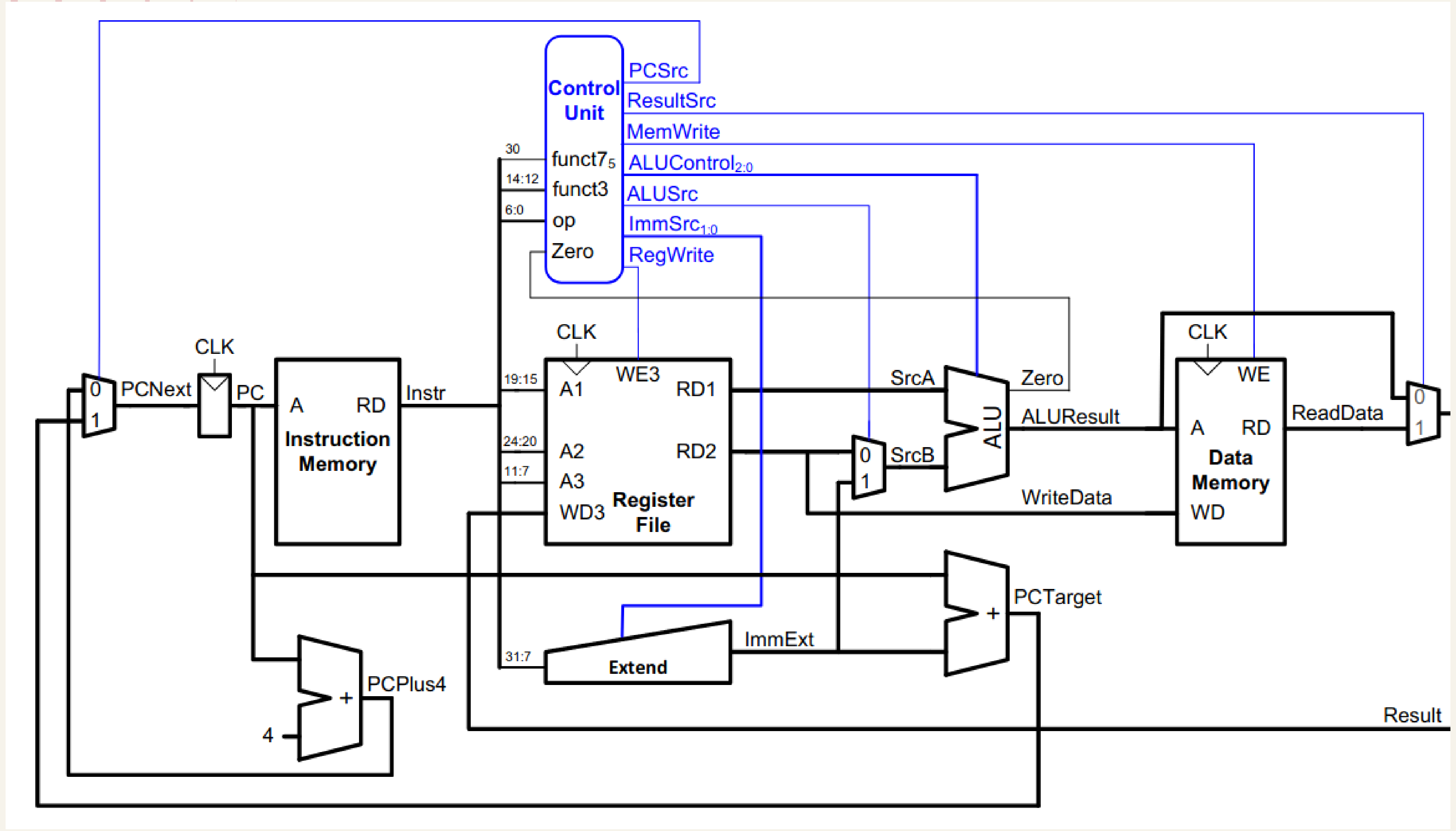
R-Type Instruction

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Op Code (4 bits)				**Funct (3 bits)**			**Rd (3 bits)**			**Rs1 (3 bits)**			**Rs2 (3 bits)**		
Operation Code				Function Code			Destination Register			Source Register 1			Source Register 2		

I-Type Instruction

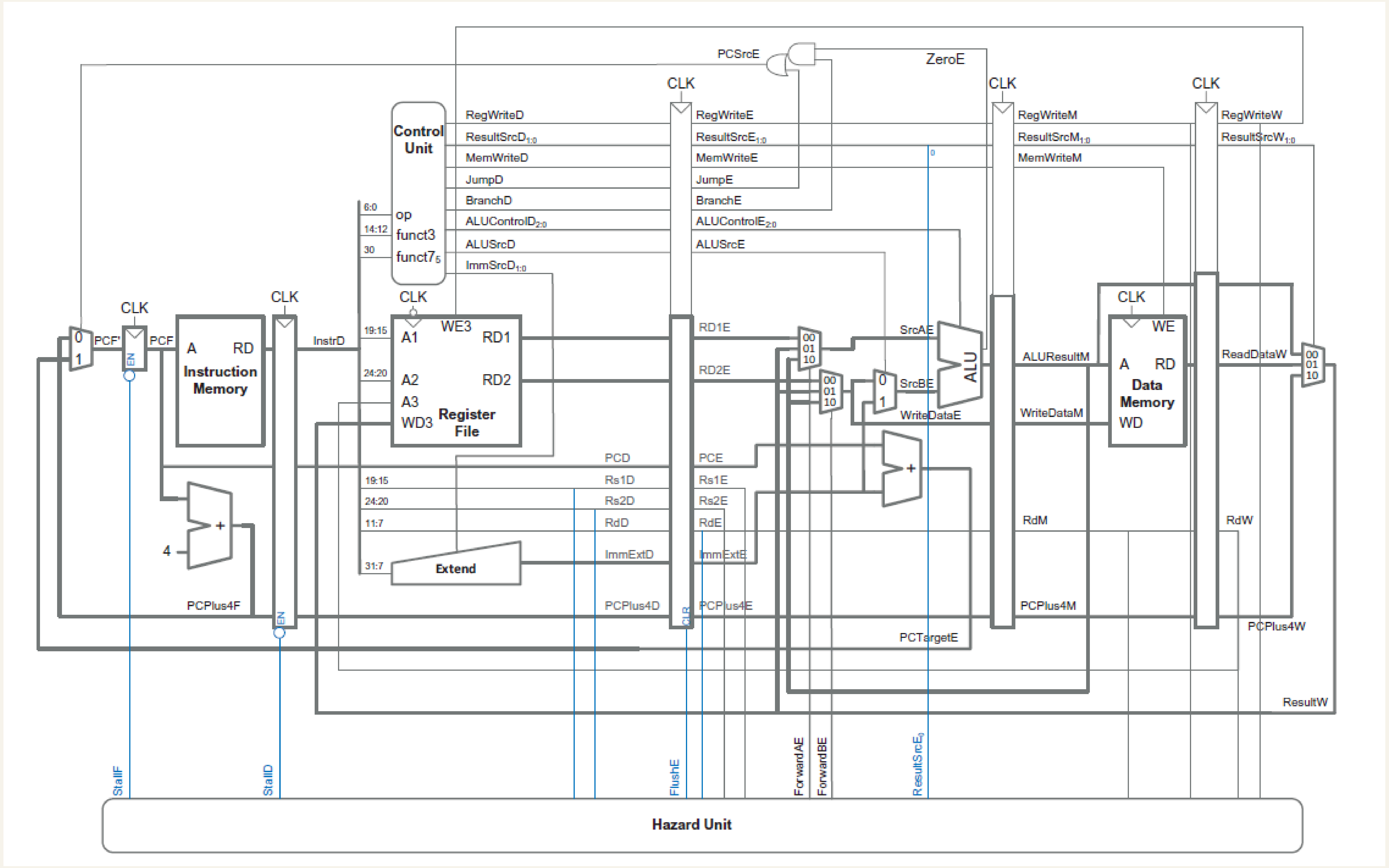
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Op Code (4 bits)				Funct (3 bits)			Dest A1 (3 bits)			Source A2 (3 bits)			Imm/Offset A3 (3 bits)		
Operation Code				Function Code			Destination Register A1			Source Register A2			Immediate/Offset A3		

DATAPATH DESIGN

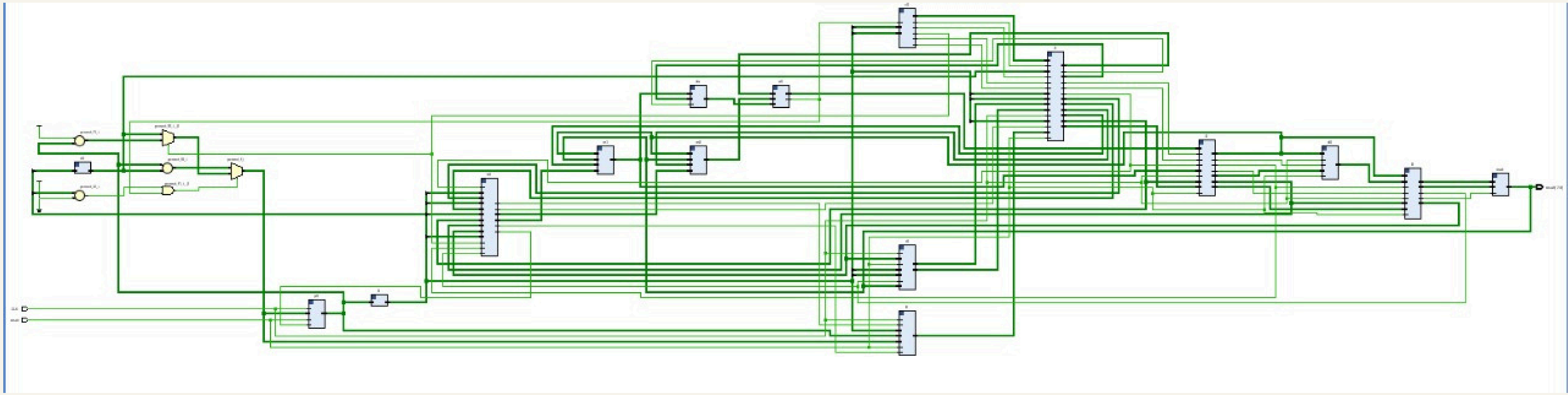


Single cycle processor datapath

The datapath is based on a load-store design adapted into a five-stage pipeline, following Harris and Harris's textbook. A single ALU performs all arithmetic, logic, and address calculations during the execute stage, reducing complexity in other stages. The Datapath uses multiplexers to select operands and passes control signals like the ALU result through pipeline registers for smooth operation.



Pipelined Microprocessor datapath



Result of design run on vivado for the pipelined processor with hazard unit

PROBLEMS FACED (SINGLE CYCLE IMPLEMENTATION)

- Private memory arrays caused errors; test signals were added for simulation access.
- Clock signal issues and faulty instructions caused the program counter to freeze; fixed by initializing registers differently.
- Control unit and datapath integration required multiple fixes.
- ALU errors were due to incorrect instruction loading, not ALU logic.
- Vivado crashes caused lost work and delays.

First Problem

Used $monitor() to track instructions and registers, improving debugging significantly.

Second Problem

Loading instructions from hex files was unreliable, so instructions were hardcoded. Conversion tools often caused errors.

PIPELINING OVERVIEW

To improve instruction throughout, the processor is designed using a five-stage pipeline architecture, a common structure in modern RISC processors. Pipelining allows multiple instructions to be processed simultaneously at different stages of execution, thereby increasing the overall efficiency and performance of the processor.

Each instruction flows through the following five sequential stages:

- Instruction Fetch (IF)
- Instruction Decode (ID)
- Execute (EX)
- Memory Access (MEM)
- Write Back (WB)

Dedicated pipeline registers between these stages store intermediate data and control signals, ensuring smooth and isolated progression across stages.

PROBLEMS FACED

- **Execution Tracking: Initially hard to trace instruction flow and register changes during simulation. Adding $monitor() to the testbench provided real-time logs of PC, instructions, and register values, making debugging much easier.**
- **Memory Access Errors: An elaboration error occurred because data_memory was private to its module. We fixed this by exposing specific memory locations through test signals for better traceability during simulation.**
- **Clock & Instruction Issues: A misconfigured datapath caused clock signal problems, and "equal to" instructions failed, freezing the PC. This was temporarily resolved by initializing registers with different values to ensure proper branching.**

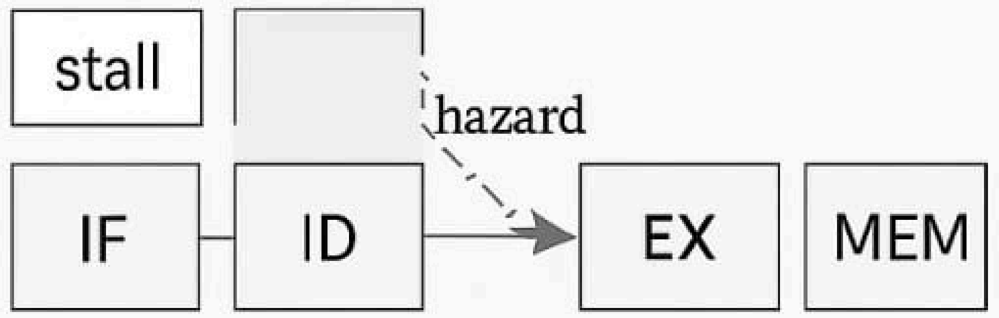
Hazard Unit

This pipelined microprocessor tackles hazards using hazard detection, selective stalling, and data forwarding techniques:

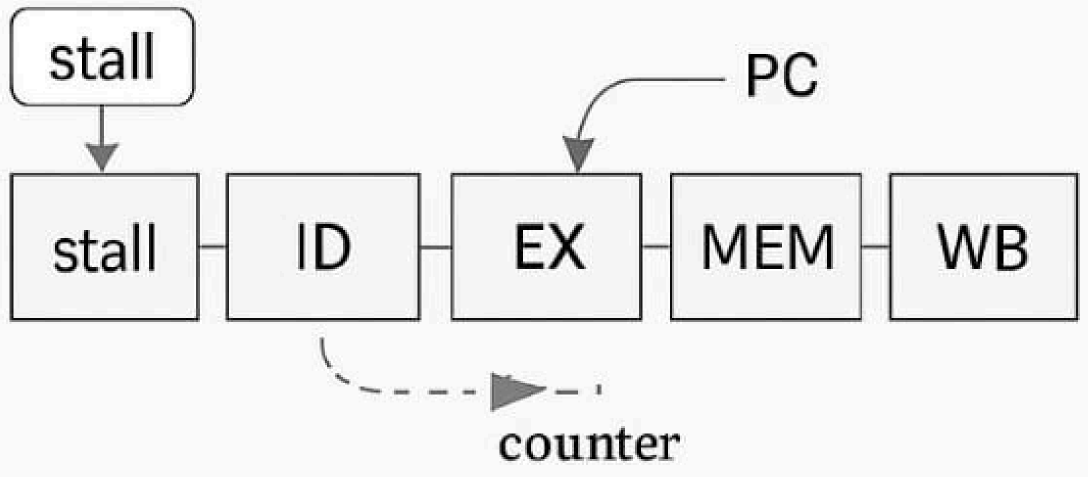
- Data Hazards: The hazard detection unit, located between ID/EX and IF/ID, identifies load-use (RAW) hazards. If an instruction in the EX stage writes to a register needed by one in the ID stage, it stalls the pipeline for one cycle.
- Stalling: On hazard detection, the processor stalls by freezing either the program counter or the IF/ID register, preventing new instruction fetch until safe.
- Data Forwarding: To minimize stalls, ALU or memory results from later pipeline stages (EX/MEM or MEM/WB) are directly forwarded to dependent instructions, bypassing the register file when possible.
- Control Hazards: For branch and jump instructions, the processor flushes the next fetched instruction if a control transfer is confirmed in the EX stage.

Structural hazards are eliminated by the microarchitecture through dedicated memories and register file ports.

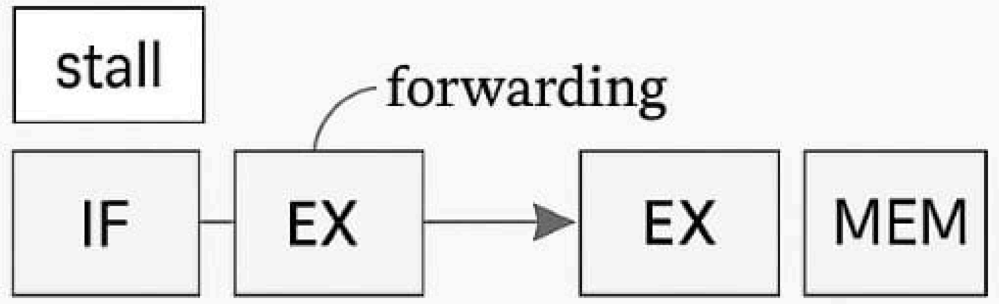
8.1 Hazard Detection Unit



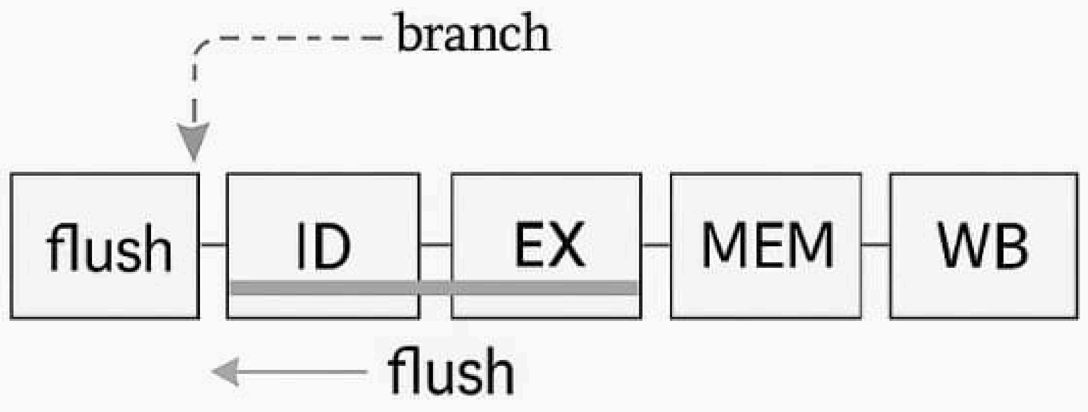
8.2 Stalling

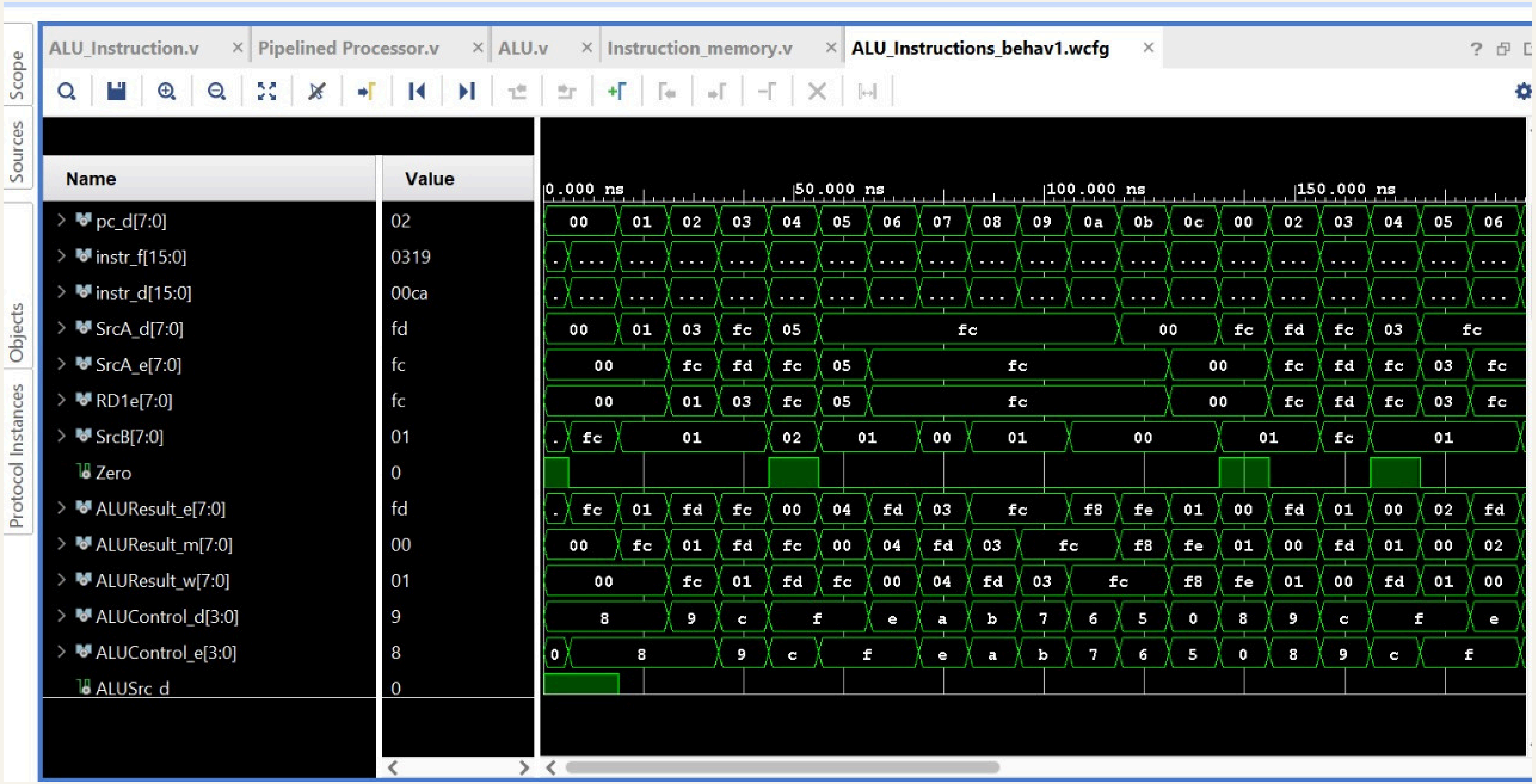


8.3 Forwarding (Bypassing)



8.4 Control Hazard Handling

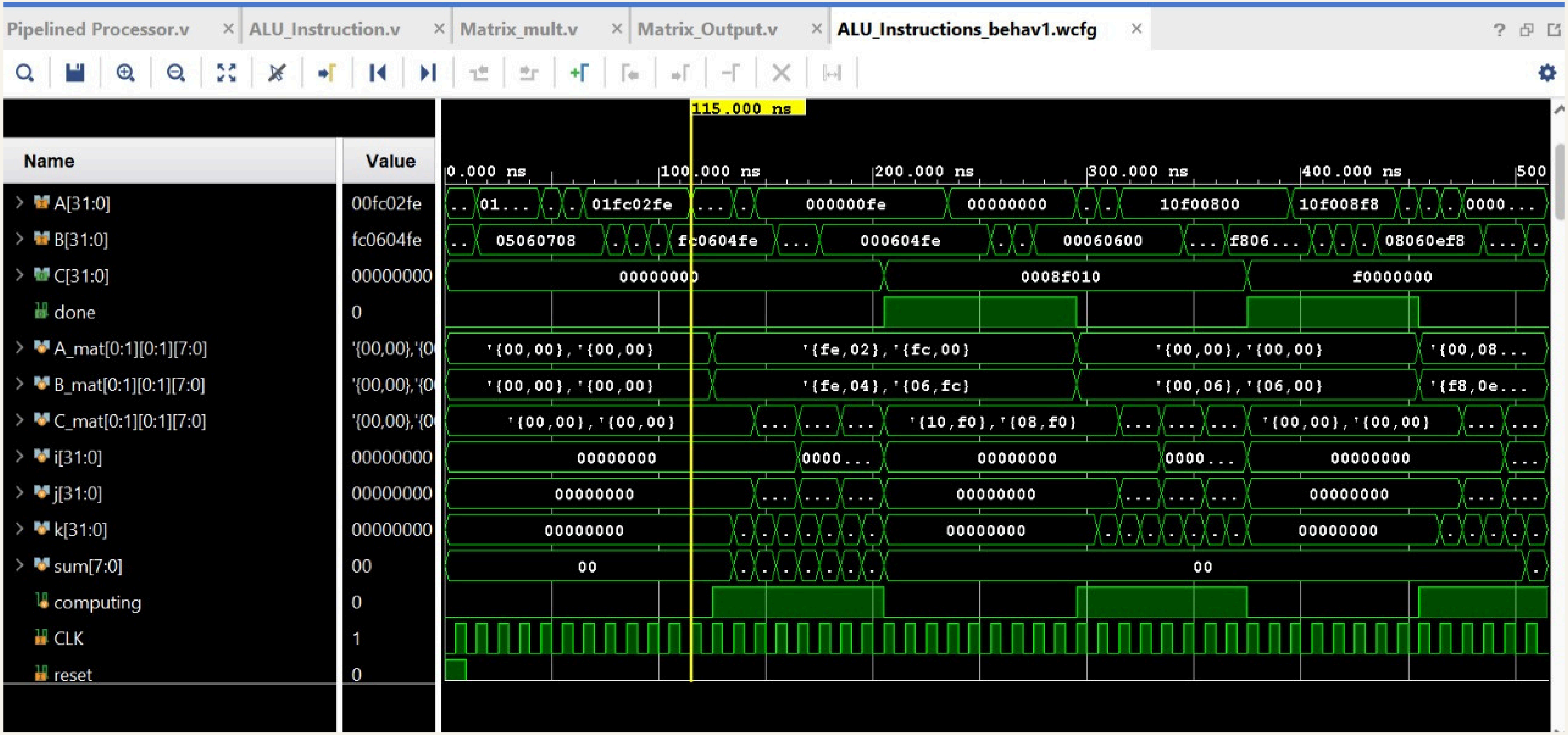




```
Name                Value
pc_d[7:0]           02      00 01 02 03 04 05 06 07 08 09 0a 0b 0c 00 02 03 04 05 06
instr_f[15:0]       0319    . ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ...
instr_d[15:0]       00ca    . ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ...
SrcA_d[7:0]         fd      00 01 03 fc 05 fc 00 fc fd fc 03 fc
SrcA_e[7:0]         fc      00 fc fd fc 05 fc 00 fc fd fc 03 fc
RD1e[7:0]           fc      00 01 03 fc 05 fc 00 fc fd fc 03 fc
SrcB[7:0]           01      . fc 01 02 01 00 01 00 01 fc 01
Zero                0
ALUResult_e[7:0]    fd      . fc 01 fd fc 00 04 fd 03 fc f8 fe 01 00 fd 01 00 02 fd
ALUResult_m[7:0]    00      00 fc 01 fd fc 00 04 fd 03 fc f8 fe 01 00 fd 01 00 02
ALUResult_w[7:0]    01      00 fc 01 fd fc 00 04 fd 03 fc f8 fe 01 00 fd 01 00
ALUControl_d[3:0]   9       8 9 c f e a b 7 6 5 0 8 9 c f e
ALUControl_e[3:0]   8       0 8 9 c f e a b 7 6 5 0 8 9 c f
ALUSrc_d            0
```

MATRIX MULTIPLICATION IMPLEMENTATION

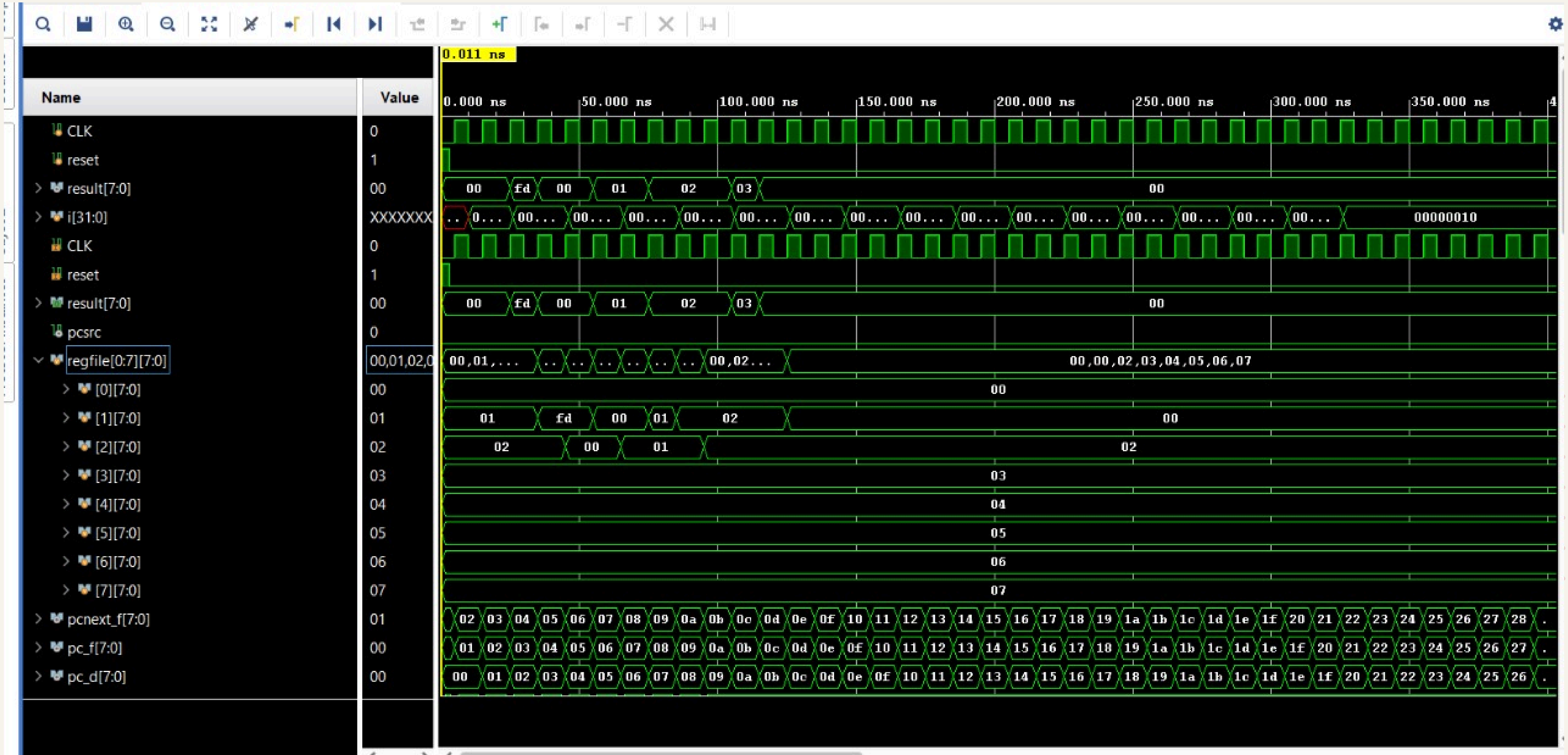
Initially, we used multi-dimensional arrays for the 2×2 matrix multiplication module to read and write matrix data. However, Vivado does not support multi-dimensional arrays as module ports, which caused synthesis errors. To fix this, we flattened the matrix inputs and outputs into 1D vectors and manually indexed them inside the module. The multiplication is triggered by a control signal decoded from a special instruction, and data is written across sequential registers controlled by an internal state machine to avoid write conflicts.



Name	Value
A[31:0]	00fc02fe
B[31:0]	fc0604fe
C[31:0]	00000000
done	0
A_mat[0:1][0:1][7:0]	'{00,00},'{0
B_mat[0:1][0:1][7:0]	'{00,00},'{0
C_mat[0:1][0:1][7:0]	'{00,00},'{0
i[31:0]	00000000
j[31:0]	00000000
k[31:0]	00000000
sum[7:0]	00
computing	0
CLK	1
reset	0

BONUS TASK 1

Manually translated a C sorting program into assembly to validate processor functionality and demonstrate instruction set understanding



Name	Value
CLK	0
reset	1
result[7:0]	00
i[31:0]	XXXXXXX
CLK	0
reset	1
result[7:0]	00
pcsrc	0
regfile[0:7][7:0]	00,01,02,0
[0][7:0]	00
[1][7:0]	01
[2][7:0]	02
[3][7:0]	03
[4][7:0]	04
[5][7:0]	05
[6][7:0]	06
[7][7:0]	07
pcnext_f[7:0]	01
pc_f[7:0]	00
pc_d[7:0]	00

METHODOLOGY

Limited Registers

C variables like i, j, and n had to be mapped to just 8 general-purpose registers. We assigned roles upfront (e.g., R1 for i, R2 for j, R3 for n) to keep the code organized.

Manual Loop Construction

Since our processor lacks high-level loop constructs, we built for loops using SUB, BEQ, and JUMP instructions with loop counters in registers

Memory Access

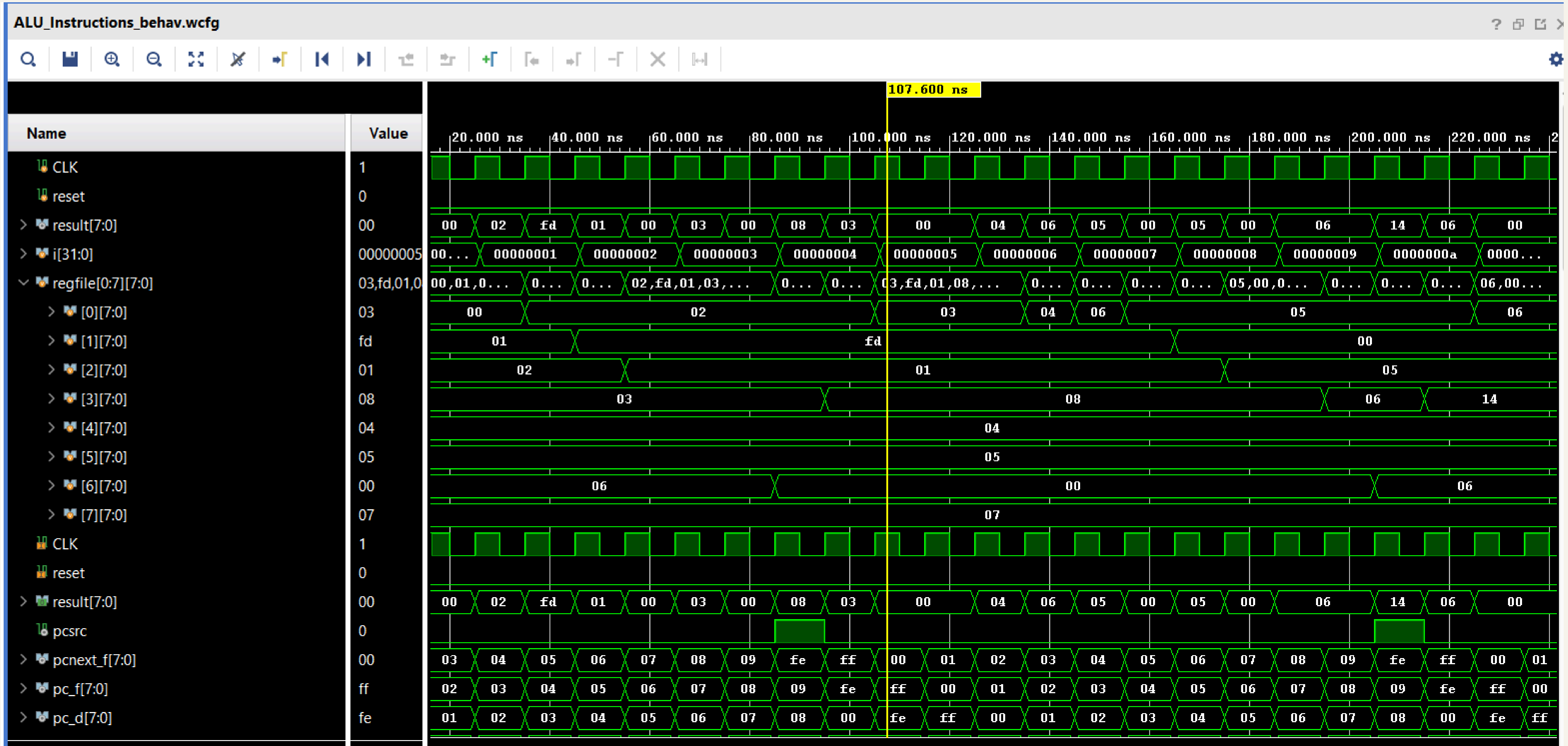
Unlike C's arr[i], we had to manually use LOAD and STORE with register-based addressing. Loop counters doubled as memory addresses for accessing array elements.

Conditional Checks

Implementing if (arr[i] > arr[j]) was difficult with just subtraction. To simplify, we added a custom SLT (Set on Less Than) instruction, enabling clean conditional logic with BEQ.

BONUS TASK 2

Factorial (Recursive Function)


ALU_Instructions_behav.wcfg

Name	Value
CLK	1
reset	0
result[7:0]	00
i[31:0]	00000005
regfile[0:7][7:0]	03,fd,01,0
[0][7:0]	03
[1][7:0]	fd
[2][7:0]	01
[3][7:0]	08
[4][7:0]	04
[5][7:0]	05
[6][7:0]	00
[7][7:0]	07
CLK	1
reset	0
result[7:0]	00
pcsrc	0
pcnext_f[7:0]	00
pc_f[7:0]	ff
pc_d[7:0]	fe

METHODOLOGY

Recursive Function Limitations

Due to the processor's lack of stack, function call, and return address support, implementing true recursion was not feasible. Instructions like CALL, RET, PUSH, and POP are absent, making it impossible to track recursive depth.

Iterative Approximation

We converted the recursive factorial logic into an iterative form. Instead of calling fact(n-1), we built a loop that multiplies values from n down to 1 using general-purpose registers.

Processor Constraints

Without a software stack, recursion had to be simulated. This trade-off allowed the factorial to be computed efficiently while remaining within the instruction limits of our custom processor.

Recursive Logic Retained

While structurally iterative, the implementation maintains the mathematical essence of recursion by repeatedly applying result = result * counter until the base case is reached.

Assembler

To facilitate efficient testing and simulation of programs for the custom 8-bit pipelined processor, a Python-based assembler was developed. This tool translates human-readable assembly instructions into corresponding 16-bit binary and hexadecimal machine codes, based on the custom Instruction Set Architecture (ISA).

Key features include:

- Automatic opcode mapping and instruction formatting
- Reliable conversion logic to eliminate manual encoding errors
- Enhanced development speed and accuracy during processor testing

This assembler plays a crucial role in streamlining program development and ensuring consistency with the processor's architecture.

PROBLEMS

Lack of Visibility into Execution

- Initially difficult to track instruction execution and register updates.
- Solution: Used monitor() in the testbench to log key signals (PC, current instruction, register addresses, write enables, data, etc.), enabling real-time internal state tracking.

Data Memory Access Error

- Encountered elaboration error: ‘data memory’ not declared under prefix ‘Direct messages’.
- Cause: Memory arrays are private within modules unless exposed.
- Solution: Added test signals in the processor module to observe specific memory locations.

PROBLEMS

Clock Signal Malfunction

- The CLK was not functioning due to a misconfigured datapath.
- Some instructions (e.g., equal to) caused the program counter to stall.
- Temporary Fix: Initialized registers with unequal values to bypass the issue during testing.

Pipeline Integration Challenges

- After transitioning from single-cycle to pipelined processor, new latches introduced between modules.
- Required: Additional signal wiring to support temporary data storage and hazard handling.

Debugging Flush and Stall Signals

- Implementing flush and stall signals was initially difficult.
- Reason: Incorrect forwarding logic in the hazard detection unit.

CONCLUSION

We successfully designed and tested an 8-bit pipelined microprocessor with a modular RISC-based architecture, handling core instructions and pipeline hazards effectively. This project deepened our understanding of digital design, assembly programming, and hardware debugging.
We thank the SNT Council for the opportunity, and we're grateful to everyone who supported us and took the time to attend our presentation.

REFERENCES

- Digital Design and Computer Architecture by David Harris
- Advanced Computer Architecture – Prof. John Jose, IIT Guwahati (NPTEL)
- Hardware Modeling using Verilog – Prof. Indranil Sengupta, IIT Kharagpur (NPTEL)
- GitHub Repositories:
- General-Purpose 8-bit Pipelined Microprocessor
- MIPS Processor in Verilog
- RISC-V Documentation
- Stack Overflow – For resolving common bugs and Vivado crashes

IIT Indore

THANK YOU

Presented By : Team El009