Department of Electronics and Communication Engineering, Nirma University, Ahmedabad, Gujarat

Computer Architecture (3EC503CC24)

Assignment Report

DESIGN AND IMPLEMENTATION OF RV32IF PROCESSOR

By: Palash Singh (22BEC081)
Nikhil Dave (22BEC078)

E-mail: 22bec081@nirmauni.ac.in
22bec078@nirmauni.ac.in

Abstract

This report documents the design and implementation of a RISC-V RV32IF processor with a pipelined architecture. The processor supports integer (RV32I) and single-precision floating-point (F) operations, following the RISC-V instruction set architecture (ISA). The design focuses on instruction fetching, decoding, execution, memory access, and write-back stages, implemented using Verilog. The pipeline integrates forwarding and hazard detection mechanisms to optimize performance. The processor is tested in a simulated environment and synthesized for FPGA implementation. The project explores the fundamental concepts of CPU design, pipelining, and floating-point computation, contributing to modern embedded systems and computer architecture education.

Keywords: RISC-V, RV32IF, Pipelined Processor, Floating-Point Unit, Verilog, FPGA, Computer Architecture

State of the Art Technology Available

RISC-V is an open-source ISA gaining traction in academia and industry due to its modularity and scalability. Unlike proprietary architectures like ARM and x86, RISC-V provides flexibility in designing custom processors.

Modern processor design involves techniques such as:

- **Pipelining:** Used to improve instruction throughput by overlapping execution stages.
- **Branch Prediction:** Reducing control hazards for improved efficiency.
- **Floating-Point Computation:** Essential for applications requiring high-precision arithmetic.
- **FPGA Prototyping:** Validating hardware designs before ASIC implementation.

Existing processor implementations, such as BOOM (Berkeley Out-of-Order Machine) and Rocket Core, showcase advanced features but are often complex for academic learning. Our project focuses on a simplified yet functional RV32IF processor tailored for educational and research purposes.

Limitations or Drawbacks of Currently Available Technology

Despite advancements in processor design, several challenges persist:

- **Power Consumption:** Pipelined designs require careful power management.
- **Verification Complexity:** Debugging timing and data hazards in multi-stage pipelines.
- **Resource Constraints on FPGA:** Limited logic elements compared to ASIC implementations.
- **Integration of FPU:** Floating-point operations increase design complexity and resource usage.
- **Memory Bottlenecks:** Performance is dependent on efficient cache and memory management.

Our implementation addresses these challenges by adopting a balanced trade-off between complexity and functionality, ensuring a workable RV32IF design with optimized performance.

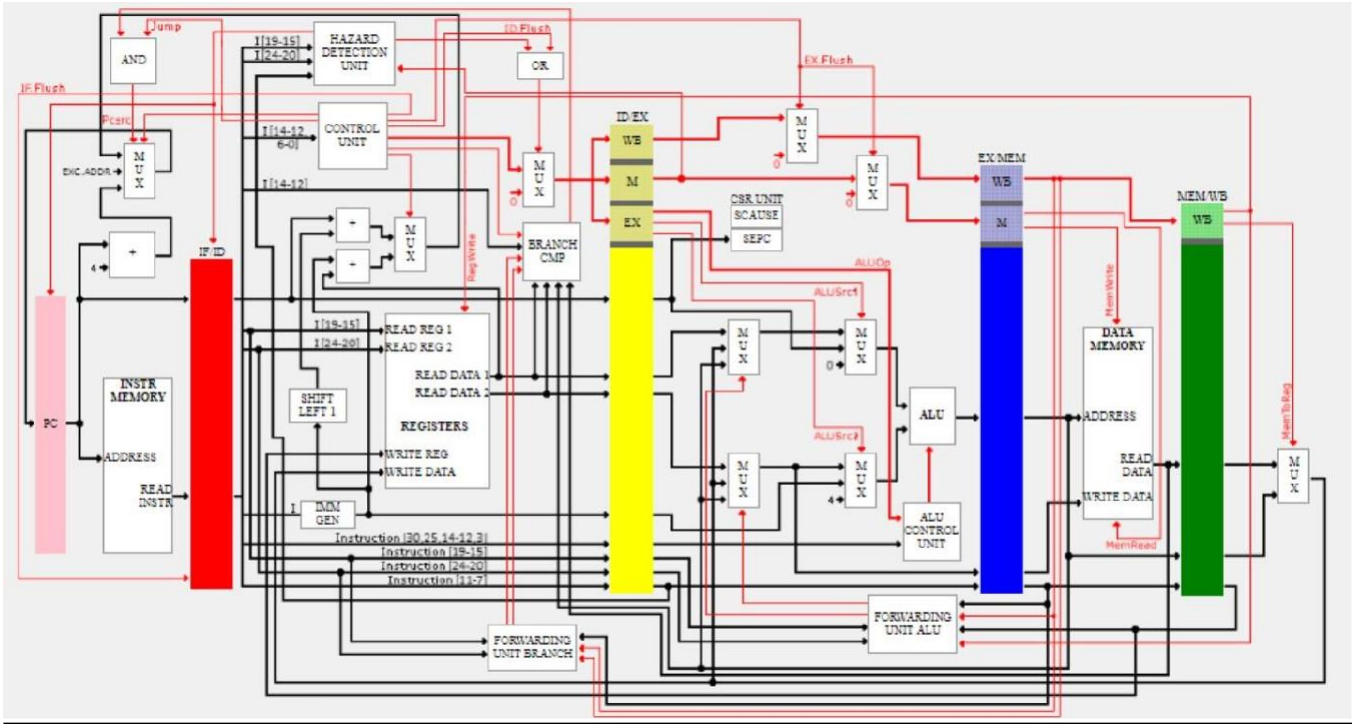
Methodology

- The RV32IF processor is designed with a 5-stage pipelined architecture for integer execution.
- The processor includes separate register files for integer and floating-point operations.
- The floating-point unit (FPU) will be implemented using Inorder Pipeline with Multicycle ALU
- The design verification is carried out using formal methods and simulation to ensure correctness.
- The processor is targeted for FPGA deployment on a Zynq 7000 device.
- The memory system includes an instruction memory for fetching instructions efficiently.
- The execution pipeline supports load/store operations, arithmetic, and branch and jump instructions.

We implemented the project by :

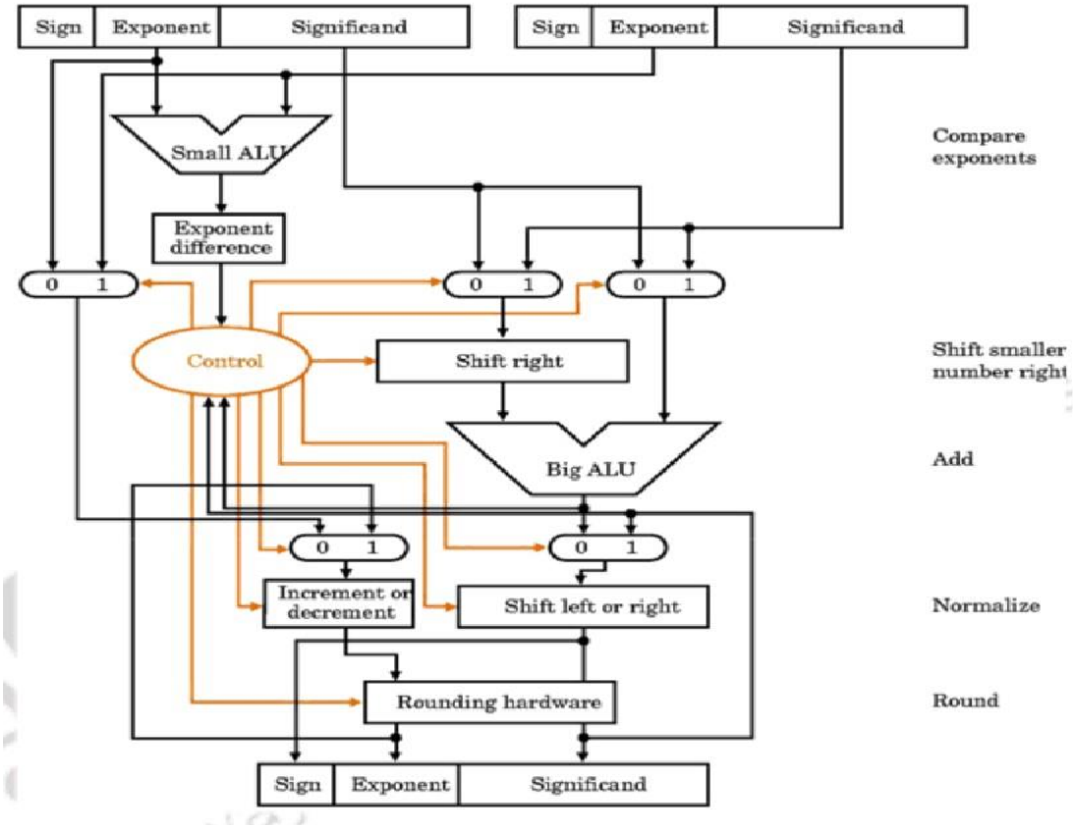
1. **Verilog Implementation**: Writing Verilog code for all hardware components to enable potential physical implementation on FPGAs.
2. **Testing Framework**: Developing comprehensive tests for each module to ensure correctness.
3. **Performance Optimization**: Identifying and implementing optimizations for both hardware designs and software components.

Block Diagram/Flow Chart

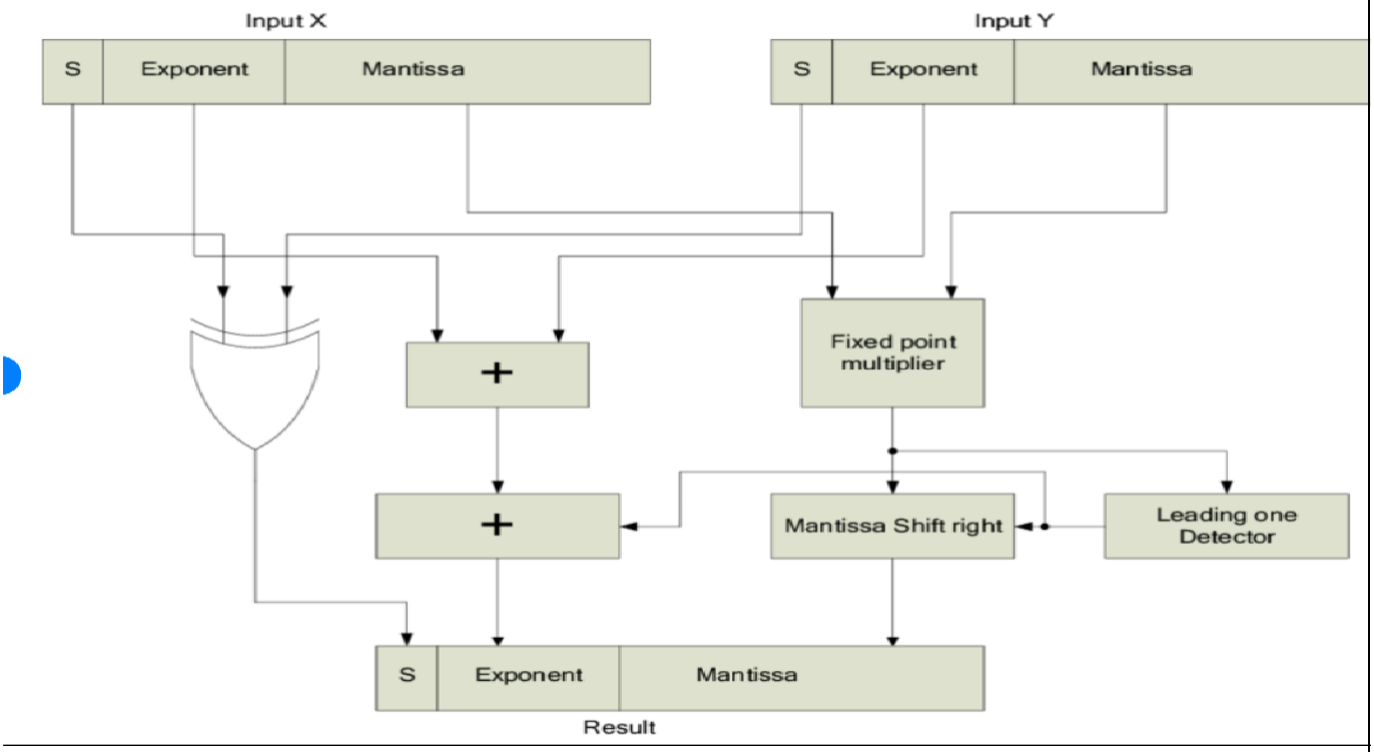


THE Structure of FP Pipeline is Similar to the one above

Floating Point Arithmetic Logic Unit (ALU)



FLOATING POINT MULITPLIER



INSTRUCTIONS SUPPORTED IN DESIGN

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR

0000000	rs2	rs1	111	rd	0110011	AND
fm	pred succ	rs1	000	rd	0001111	FENCE
1000	0011 0011	00000	000	00000	0001111	FENCE.TSO
0000	0001 0000	00000	000	00000	0001111	PAUSE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

Additionally supported instructions in floating point unit are

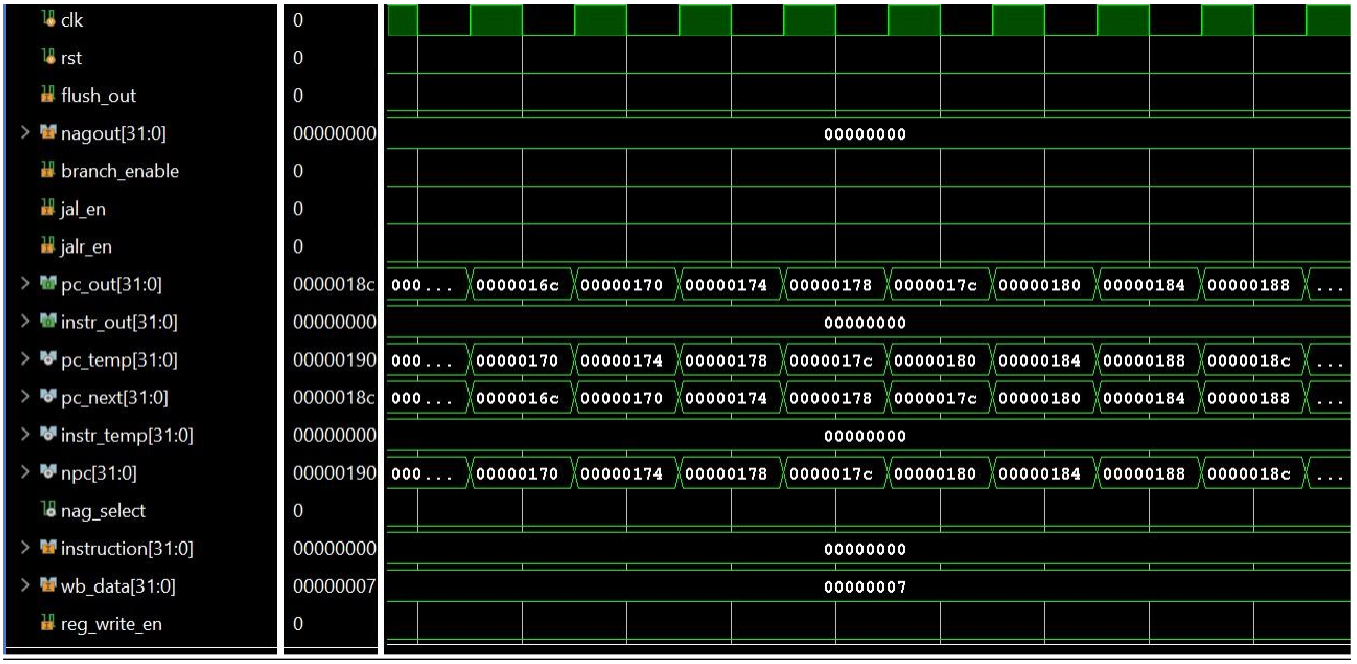
- Addition and Subtraction of single precision floating point numbers
- Multiplication of floating point numbers
- Division of floating point numbers
- Load / Store intructions

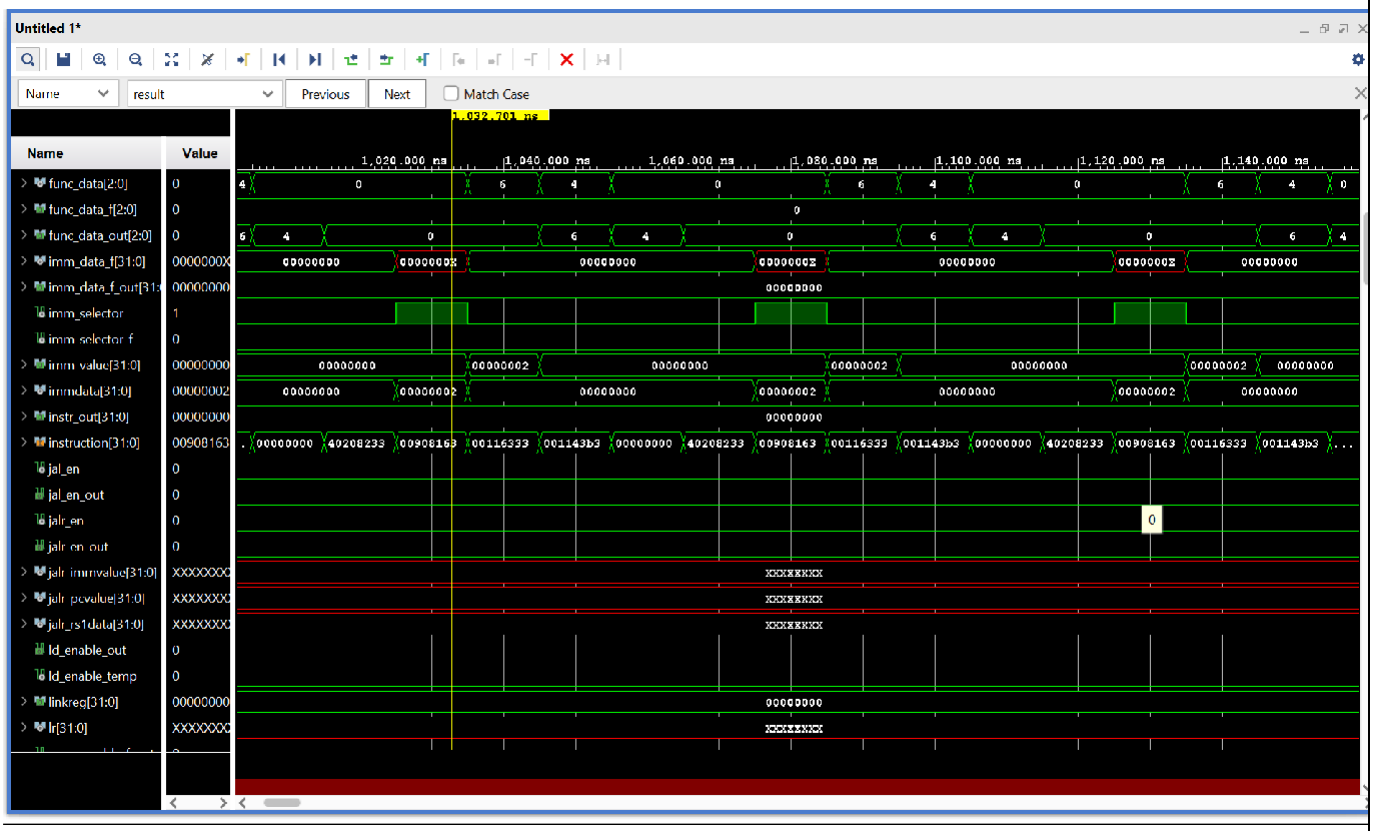
Simulation/Implementation Results

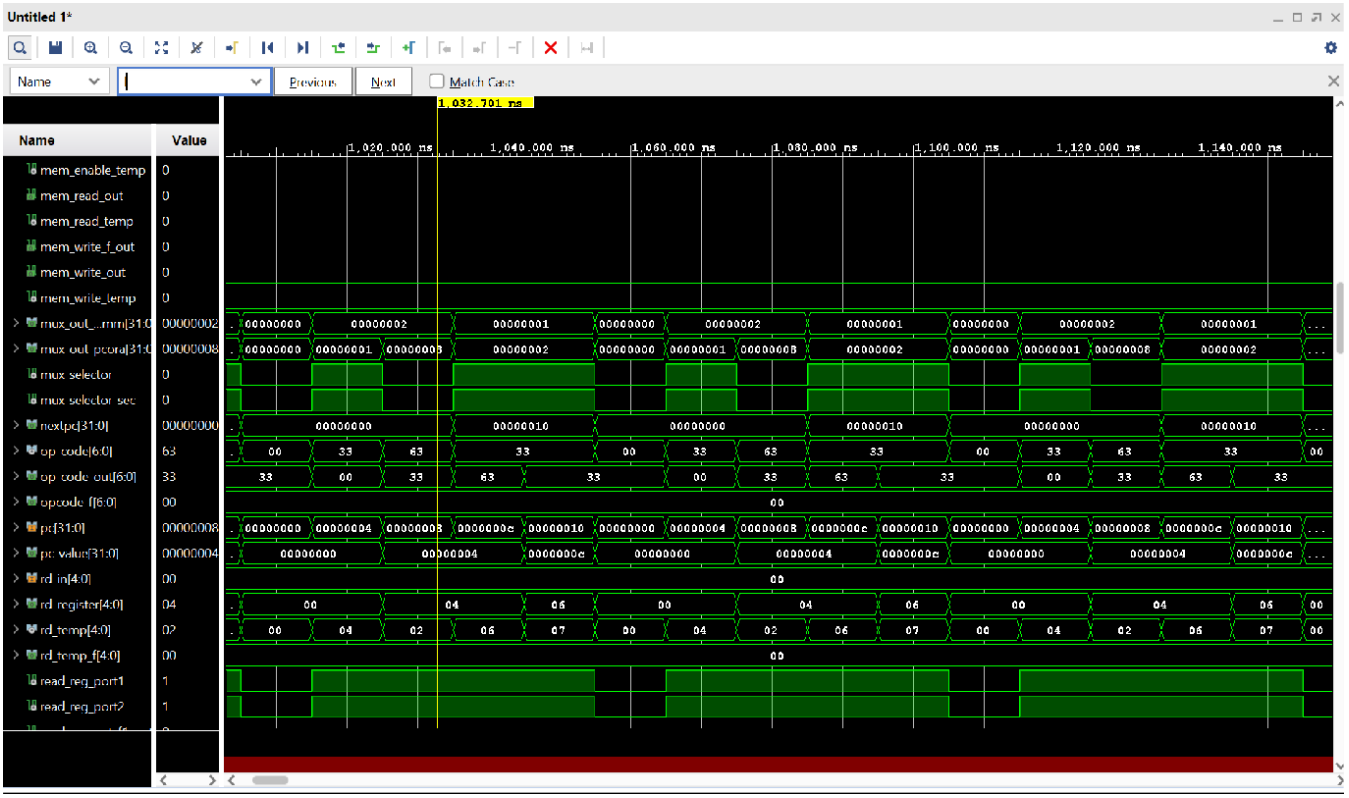
1. Software Simulation

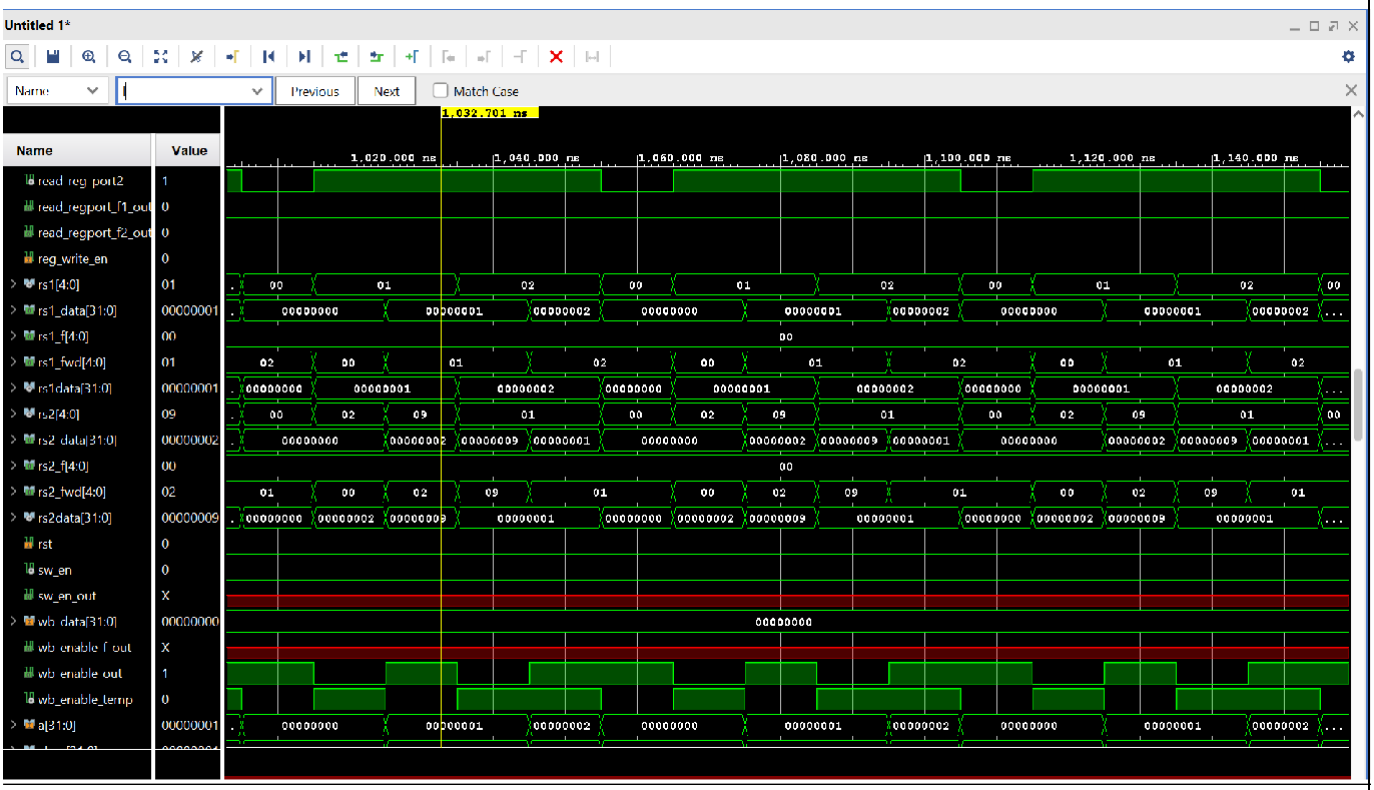
- **Tools Used**:
  - Xilinx Vivado: We implemented this whole processor design using Verilog in vivado.
  - Xsim Simulator: The design was simulated in Xsim Simulator.
- **Description**:
  - The processor design was simulated using Xsim Simulator in Vivado, allowing us to verify the correct execution of instructions and pipeline operations.
  - The simulation helped in debugging pipeline hazards, verifying data forwarding, and confirming proper interaction between different stages of the processor.
  - Screenshots of the simulation waveforms and FPGA synthesis results are included below, demonstrating successful execution of test programs.

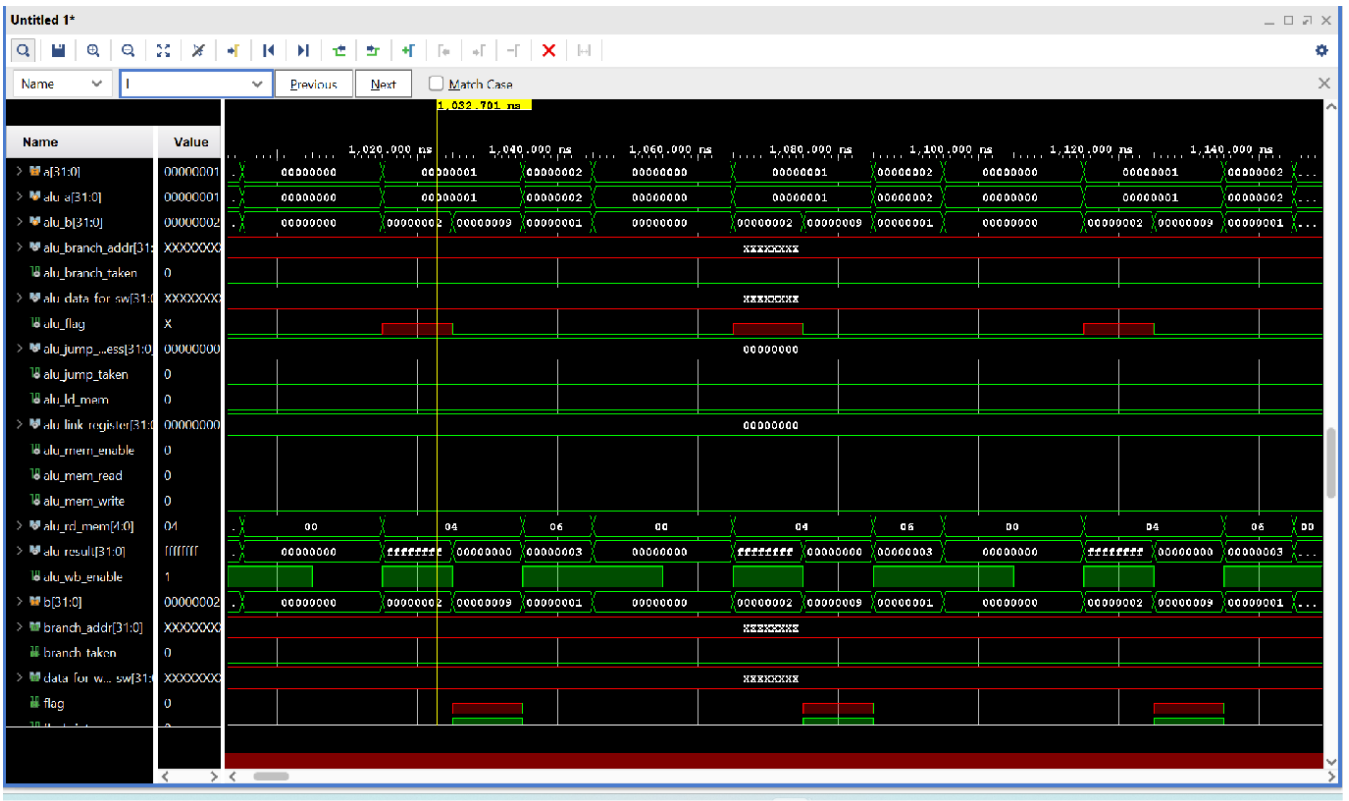
I class Branch instruction

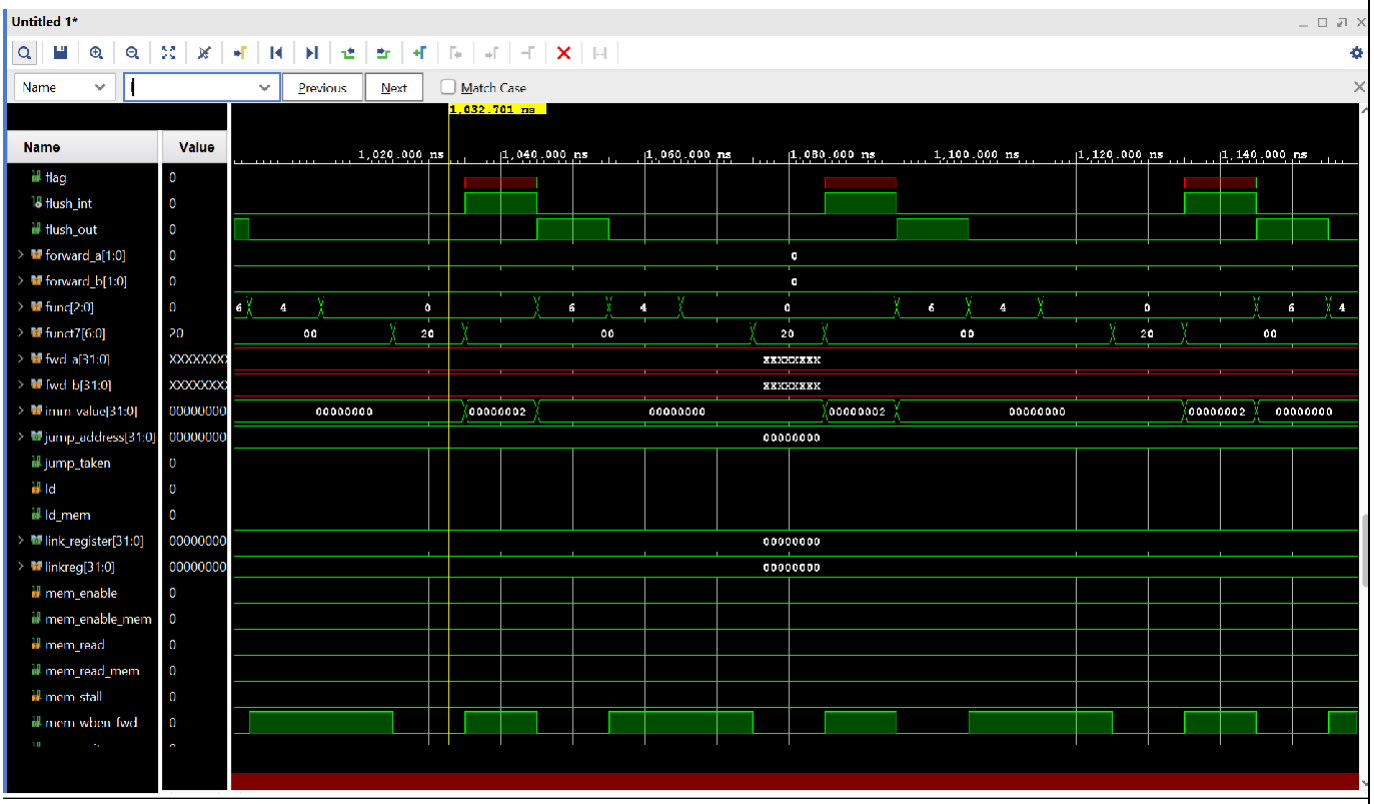


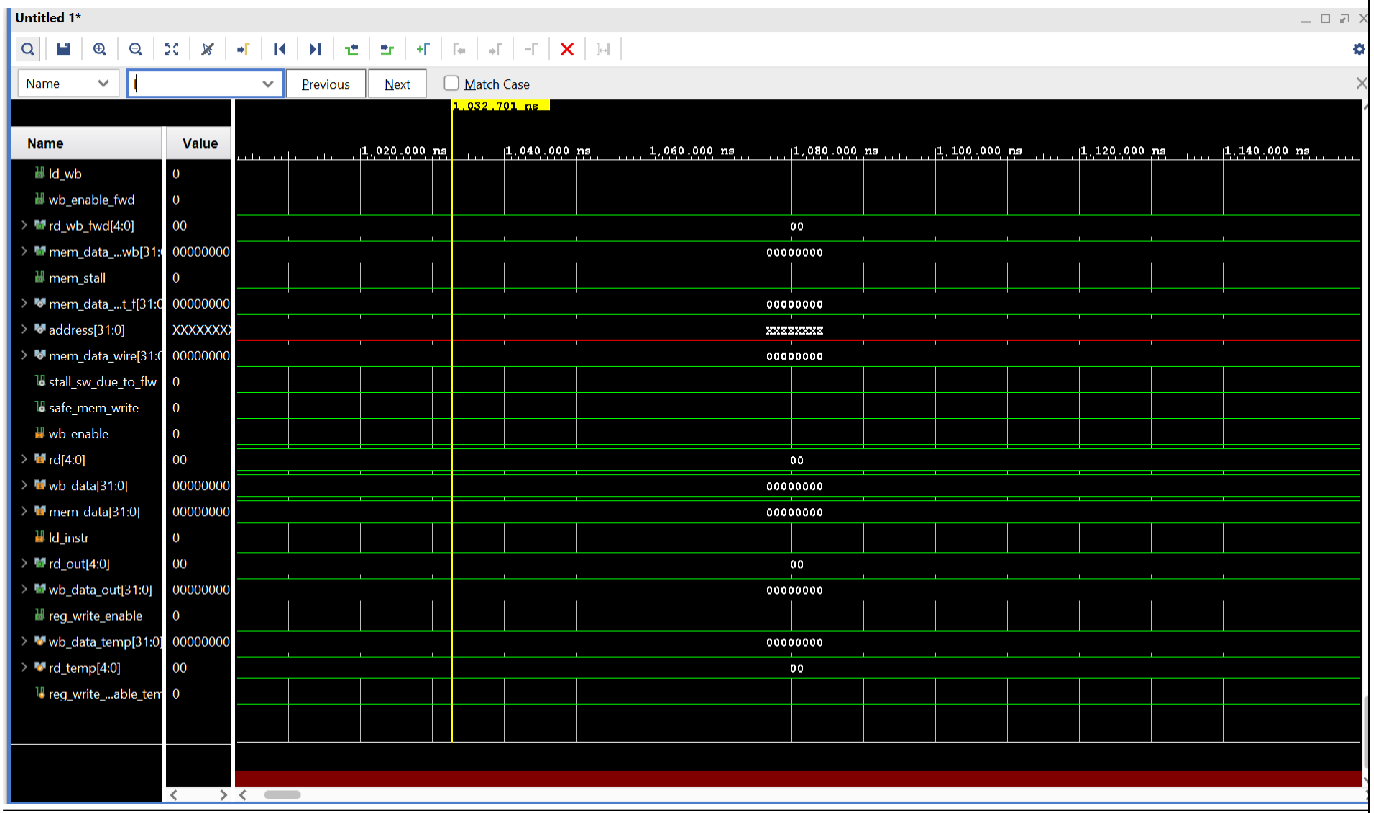

Untitled 1*
Name
result
Previous
Next
Match Case
1,032.701 ns
Name
Value
1,020.000 ns
1,040.000 ns
1,060.000 ns
1,080.000 ns
1,100.000 ns
1,120.000 ns
1,140.000 ns
func_data[2:0]
func_data_f[2:0]
func_data_out[2:0]
imm_data_f[31:0]
imm_data_f_out[31:
imm_selector
imm_selector_f
imm_value[31:0]
immdata[31:0]
instr_out[31:0]
instruction[31:0]
jal_en
jal_en_out
jalr_en
jalr_en_out
jalr_immvalue[31:0]
jalr_pcvalue[31:0]
jalr_rs1data[31:0]
ld_enable_out
ld_enable_temp
linkreg[31:0]
lr[31:0]


Untitled 1*
Name
Previous
Next
Match Case
1,032.701 ns
Name
Value
1,020.000 ns
1,040.000 ns
1,060.000 ns
1,080.000 ns
1,100.000 ns
1,120.000 ns
1,140.000 ns
mem_enable_temp
mem_read_out
mem_read_temp
mem_write_f_out
mem_write_out
mem_write_temp
mux_out_...mm[31:0]
mux_out_pcoral[31:0]
mux_selector
mux_selector_sec
nextpc[31:0]
op_code[6:0]
op_code_out[6:0]
opcode_f[6:0]
pc[31:0]
pc_value[31:0]
rd_in[4:0]
rd_register[4:0]
rd_temp[4:0]
rd_temp_f[4:0]
read_reg_port1
read_reg_port2

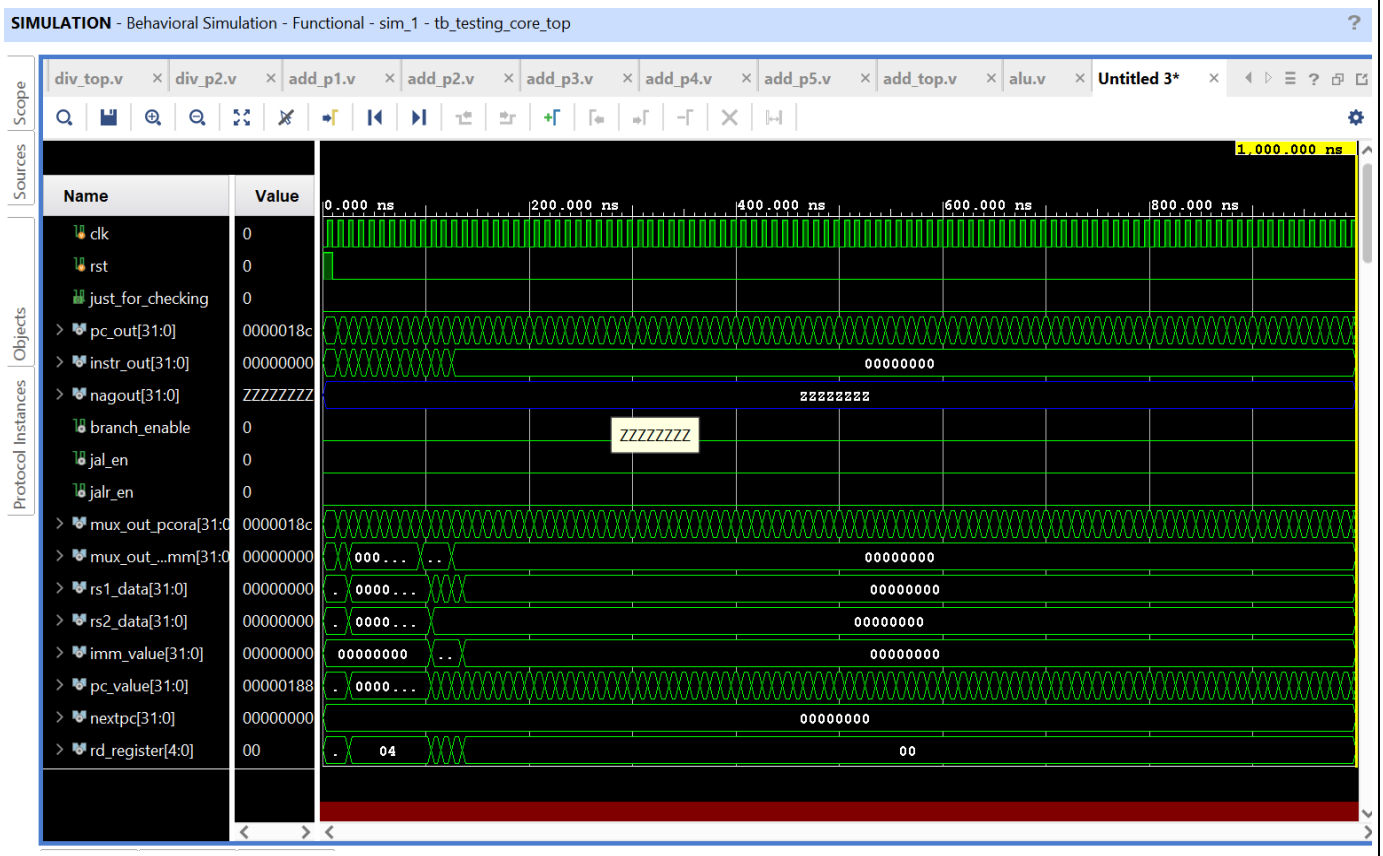

Untitled 1*
Name
Previous
Next
Match Case
1,032.701 ns
Name
Value
read_reg_port2
read_regport_f1_out
read_regport_f2_out
reg_write_en
rs1[4:0]
rs1_data[31:0]
rs1_f[4:0]
rs1_fwd[4:0]
rs1data[31:0]
rs2[4:0]
rs2_data[31:0]
rs2_f[4:0]
rs2_fwd[4:0]
rs2data[31:0]
rst
sw_en
sw_en_out
wb_data[31:0]
wb_enable_f_out
wb_enable_out
wb_enable_temp
a[31:0]

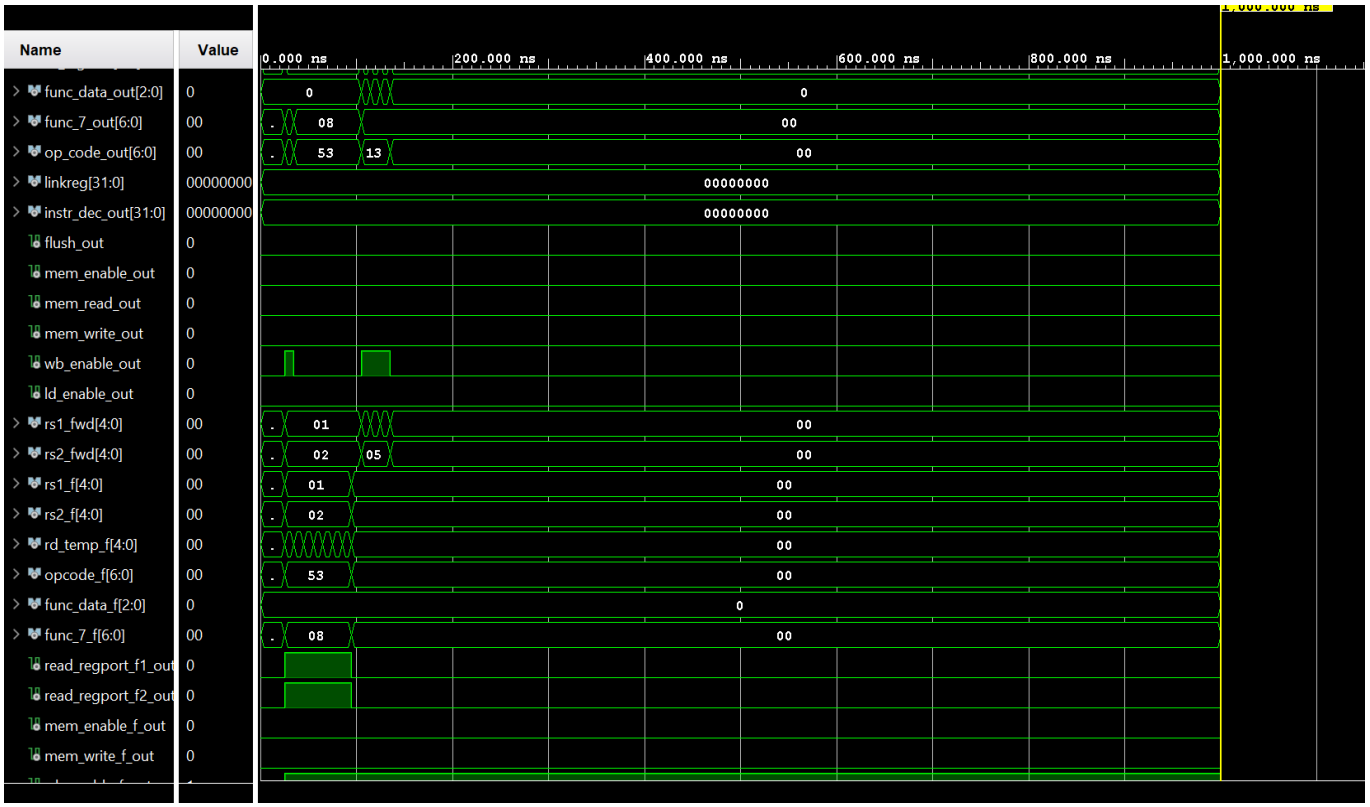

Untitled 1*
Name
Previous
Next
Match Case
1,032.701 ns
Name
Value
a[31:0]
alu_a[31:0]
alu_b[31:0]
alu_branch_addr[31:
alu_branch_taken
alu_data_for_sw[31:
alu_flag
alu_jump_...ess[31:0
alu_jump_taken
alu_ld_mem
alu_link_register[31:
alu_mem_enable
alu_mem_read
alu_mem_write
alu_rd_mem[4:0]
alu_result[31:0]
alu_wb_enable
b[31:0]
branch_addr[31:0]
branch_taken
data_for_w..._sw[31:
flag

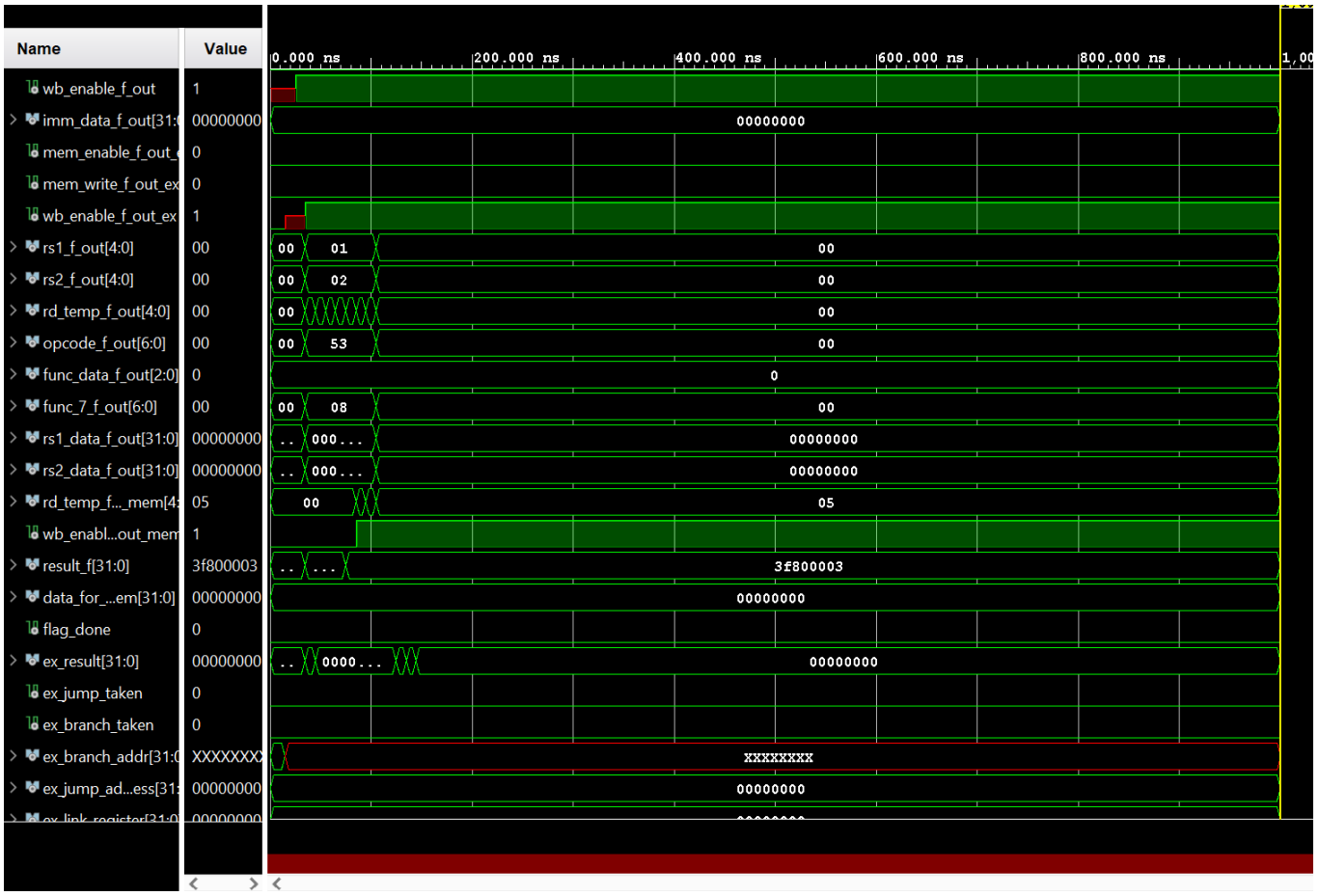


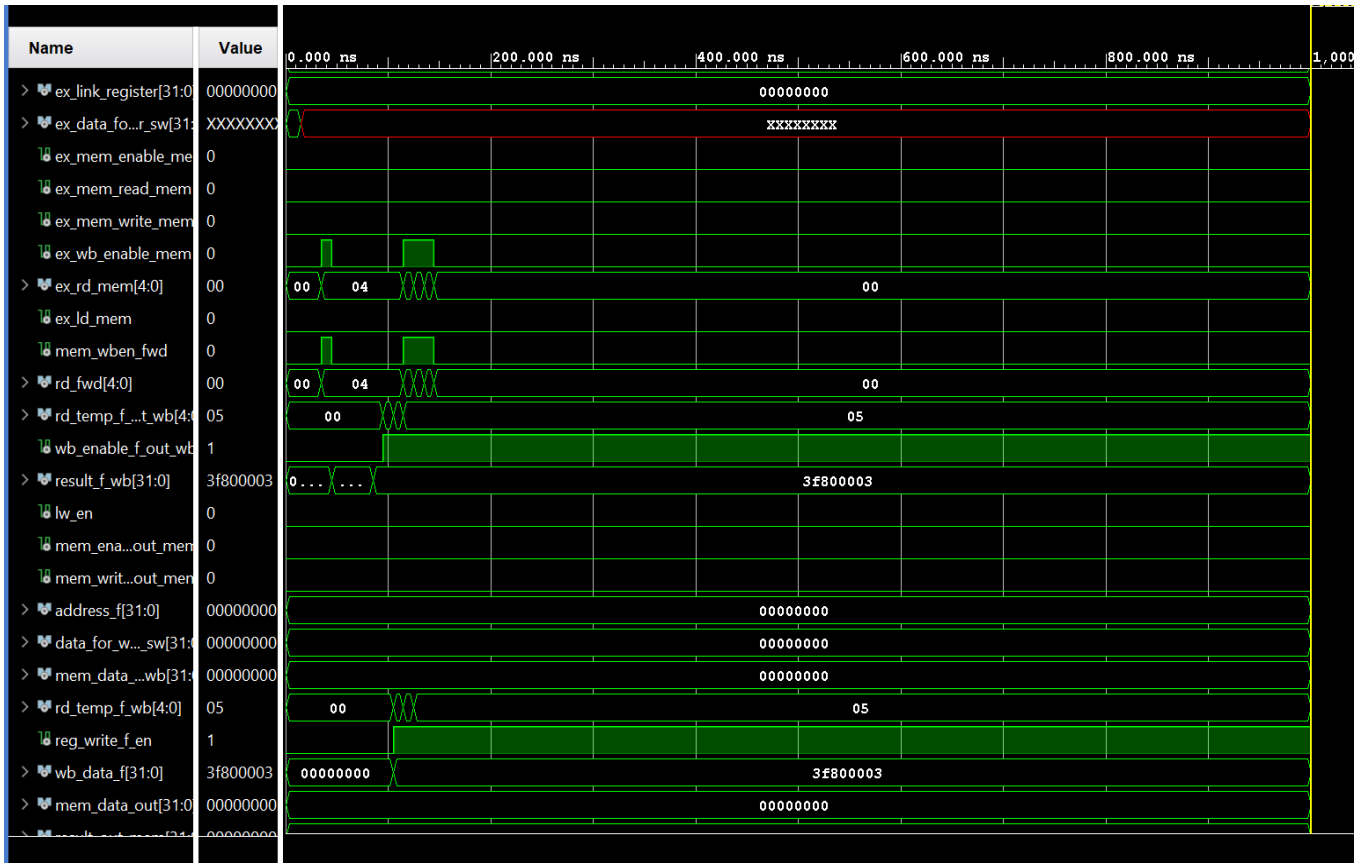


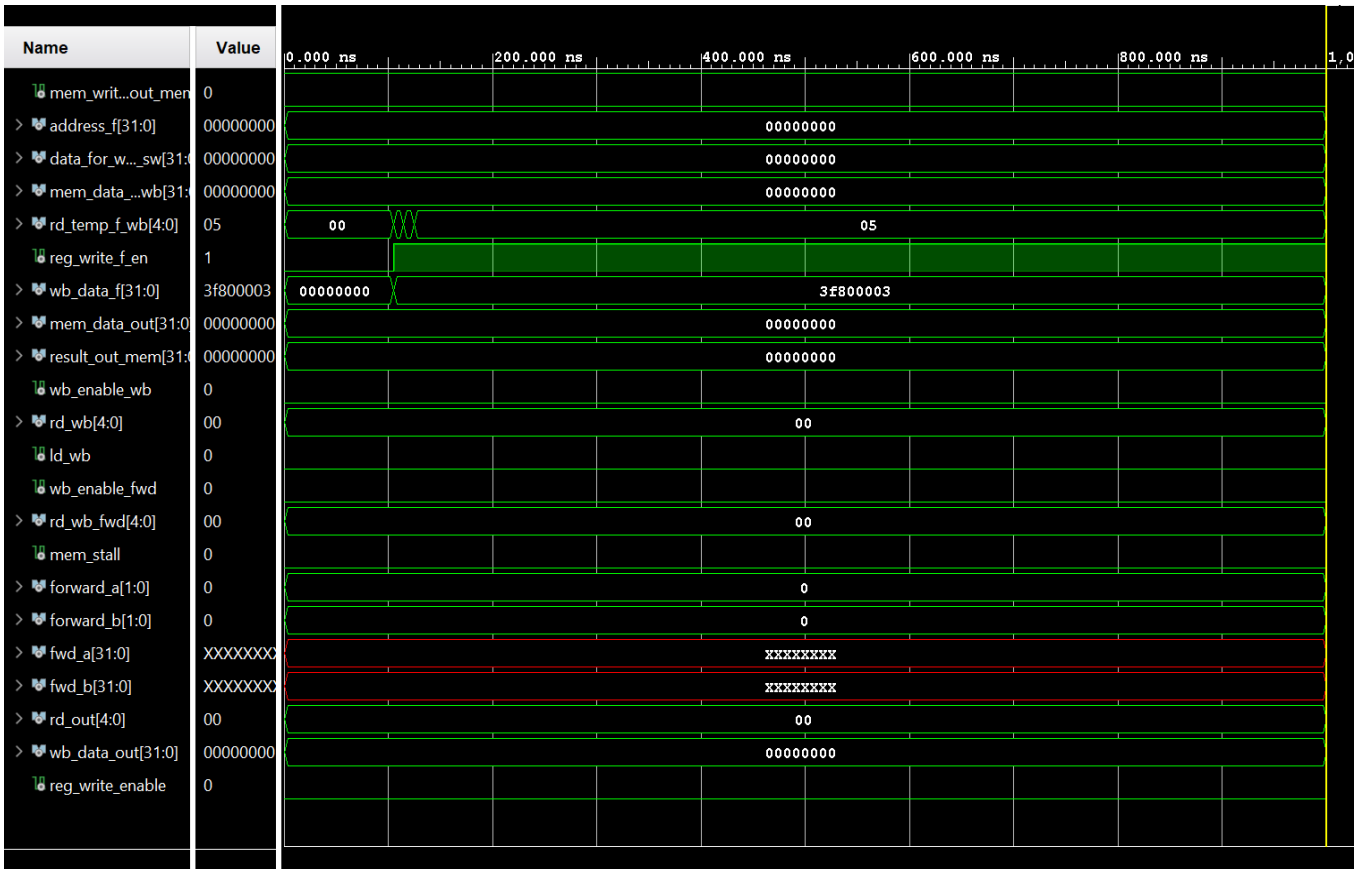
Floating point multiply instruction


SIMULATION - Behavioral Simulation - Functional - sim_1 - tb_testing_core_top
div_top.v
div_p2.v
add_p1.v
add_p2.v
add_p3.v
add_p4.v
add_p5.v
add_top.v
alu.v
Untitled 3*
Scope
Sources
Objects
Protocol Instances
Name
Value
clk
rst
just_for_checking
pc_out[31:0]
instr_out[31:0]
nagout[31:0]
branch_enable
jal_en
jalr_en
mux_out_pcora[31:0]
mux_out_...mm[31:0]
rs1_data[31:0]
rs2_data[31:0]
imm_value[31:0]
pc_value[31:0]
nextpc[31:0]
rd_register[4:0]
0.000 ns
200.000 ns
400.000 ns
600.000 ns
800.000 ns
1,000.000 ns
ZZZZZZZZ


Name
Value
func_data_out[2:0]
func_7_out[6:0]
op_code_out[6:0]
linkreg[31:0]
instr_dec_out[31:0]
flush_out
mem_enable_out
mem_read_out
mem_write_out
wb_enable_out
ld_enable_out
rs1_fwd[4:0]
rs2_fwd[4:0]
rs1_f[4:0]
rs2_f[4:0]
rd_temp_f[4:0]
opcode_f[6:0]
func_data_f[2:0]
func_7_f[6:0]
read_regport_f1_out
read_regport_f2_out
mem_enable_f_out
mem_write_f_out
0.000 ns
200.000 ns
400.000 ns
600.000 ns
800.000 ns
1,000.000 ns


Name
Value
0.000 ns
200.000 ns
400.000 ns
600.000 ns
800.000 ns
wb_enable_f_out
imm_data_f_out[31:
mem_enable_f_out_
mem_write_f_out_ex
wb_enable_f_out_ex
rs1_f_out[4:0]
rs2_f_out[4:0]
rd_temp_f_out[4:0]
opcode_f_out[6:0]
func_data_f_out[2:0]
func_7_f_out[6:0]
rs1_data_f_out[31:0]
rs2_data_f_out[31:0]
rd_temp_f..._mem[4:
wb_enabl...out_mem
result_f[31:0]
data_for_...em[31:0]
flag_done
ex_result[31:0]
ex_jump_taken
ex_branch_taken
ex_branch_addr[31:
ex_jump_ad...ess[31:
00000000
3f800003
XXXXXXXX


Name
Value
0.000 ns
200.000 ns
400.000 ns
600.000 ns
800.000 ns
ex_link_register[31:0]
ex_data_fo...r_sw[31:
ex_mem_enable_me
ex_mem_read_mem
ex_mem_write_mem
ex_wb_enable_mem
ex_rd_mem[4:0]
ex_ld_mem
mem_wben_fwd
rd_fwd[4:0]
rd_temp_f_...t_wb[4:
wb_enable_f_out_wb
result_f_wb[31:0]
lw_en
mem_ena...out_mem
mem_writ...out_mem
address_f[31:0]
data_for_w..._sw[31:
mem_data_...wb[31:
rd_temp_f_wb[4:0]
reg_write_f_en
wb_data_f[31:0]
mem_data_out[31:0]
00000000
XXXXXXXX
3f800003



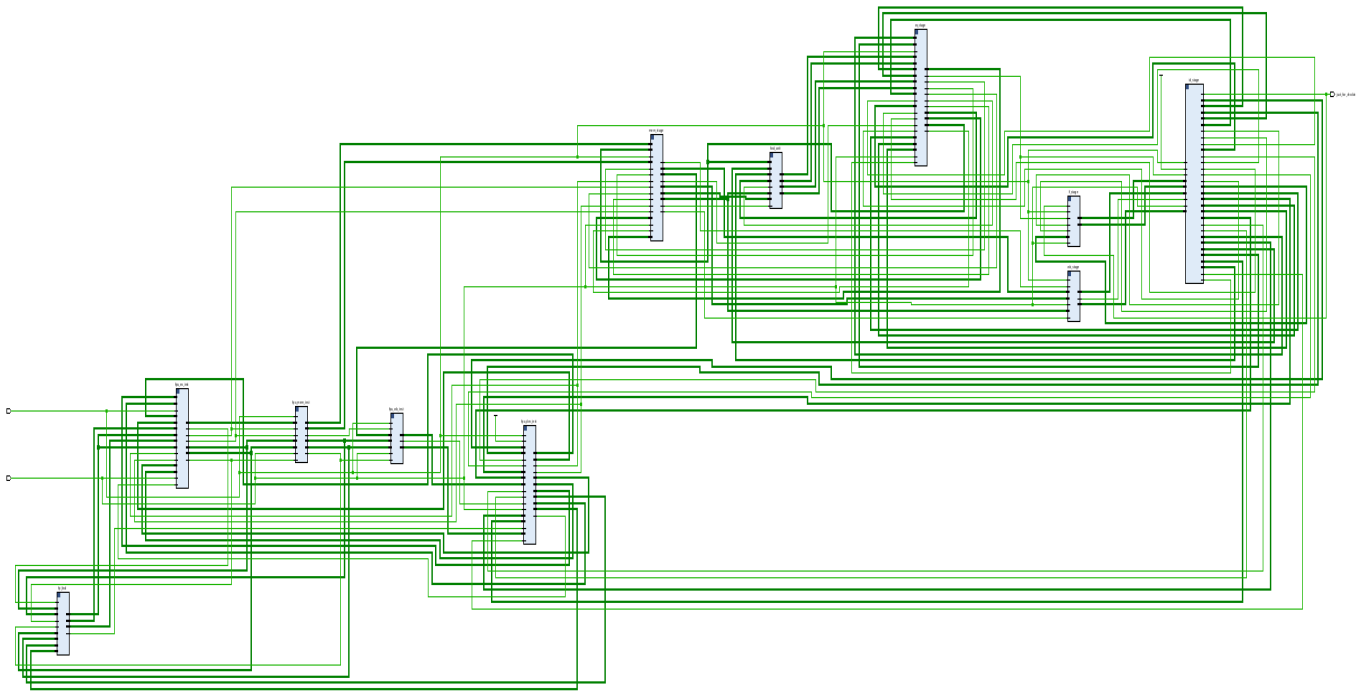
CPU Emulator executing instructions.

- **Results**:
    - The software simulation produced the expected results, confirming that the design logic is correct at a high level.

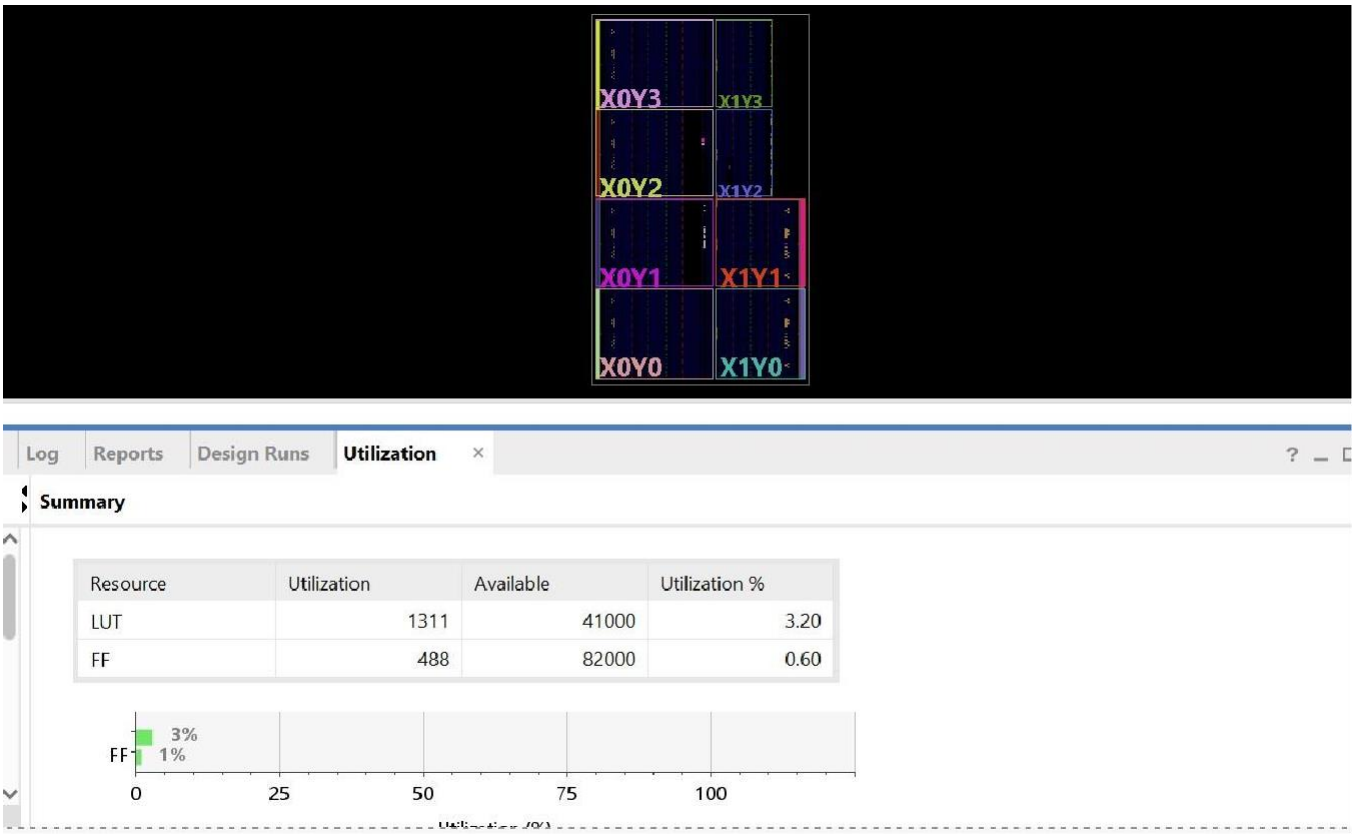
---

2. Hardware Simulation

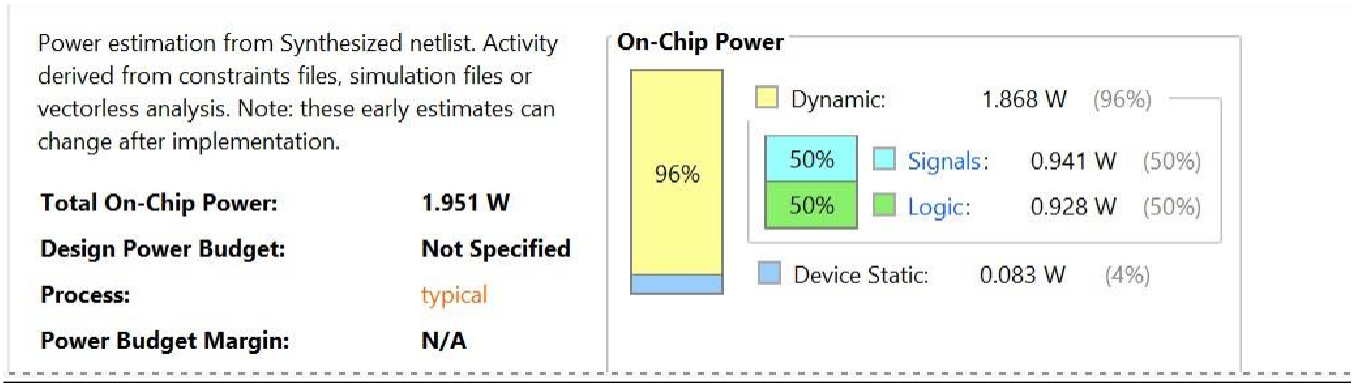
- **Tools Used**: ○ Vivado (v2024.2): For synthesizing and simulating the Verilog implementation.
- **Description**:
    - The Verilog code was synthesized and simulated using Vivado. The synthesis process involved converting the RTL design into a gate-level netlist targeting the Xilinx Zynq-7000 FPGA (xc7z020clg484-1).
    - Screenshots of the Vivado synthesis and simulation results are included below.



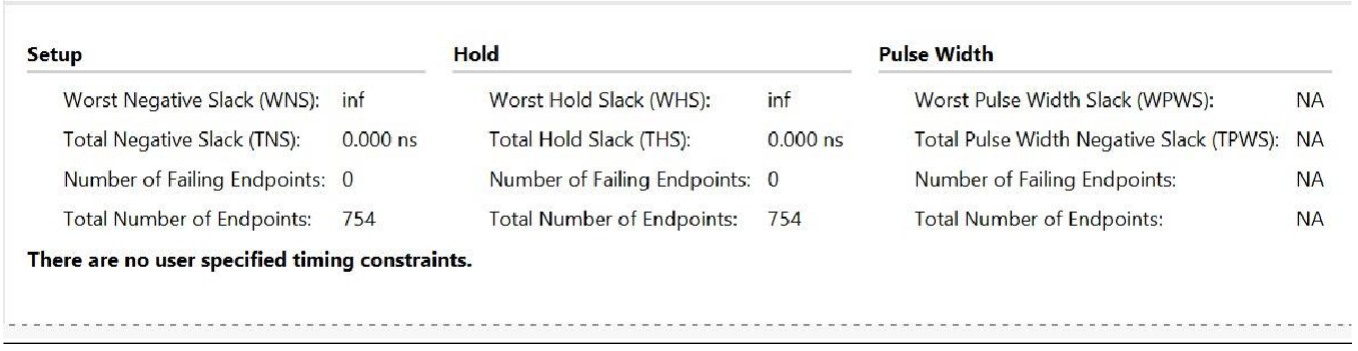
Vivado RTL showing successful completion.



Vivado synthesis log showing successful completion along with resource utilization.



Power dissipation



Timing Summary

- **Results**:

- The hardware simulation produced results that were entirely consistent with the software simulation.

Analysis of the Obtained Results

Our implementation demonstrated several key achievements:

1. Functionality: The pipelined processor successfully executed test programs that performed arithmetic operations, memory access, branch , jump and instructions.
2. Architecture Validation: The pipeline architecture we implemented proved sufficient for general computing tasks in the simulation environment.
3. Verilog Implementation: Our Verilog code successfully described the hardware components, though real hardware synthesis and testing remaining for future work(fpu) .
4. Performance Considerations: Simulation indicated that our design would generate output for each base-class instruction in exactly 5- clock cycles.
5. Resource Utilization: While not physically implemented, our design maintained reasonable resource usage in terms of logic elements that would be required in an FPGA implementation.
6. Power Consumption: Power Consumption is as minimum as possible for the design

The main limitation was the lack of hardware verification, which would have provided more concrete performance metrics.

Other Relevant Information

Throughout this project, we gained valuable insights into:

1. The importance of modular design and clear interfaces between components
2. The challenges of debugging complex digital systems
3. The trade-offs between hardware and software implementations
4. The practical limitations of theoretical computer architectures

Additionally, we noted that a physical implementation would require consideration of:

- Power consumption
- Heat dissipation
- Signal integrity
- Clock distribution
- Physical layout constraints

NOTE:

1) In the above implementation store word instruction must not be followed immediately by load word as writing in the data memory is synchronous while reading is asynchronous. An NOP must be inserted between these two consecutive instruction
2) Also if floating point instruction is given it must be followed by NOP instruction as floating point arithmetic is not single cycle

Potential Extensions

Several enhancements could be made to the current implementation:

1. **Cache Implementation**: Adding L1/L2 caches would enhance memory access times
2. **Extended Instruction Set**: Adding SIMD instructions
3. **Graphics Acceleration**: Implementing basic GPU functionality

Conclusion

The RV32IF processor successfully implements a pipelined RISC-V architecture, demonstrating efficient execution of integer and floating-point instructions. The FPGA synthesis results confirm optimal resource utilization, ensuring that the design can be deployed effectively on Zynq 7000 hardware.

The modular design approach allowed for easy debugging and verification, with simulation results validating the correct execution of all implemented instructions. While the integer pipeline is fully operational, the floating-point unit (FPU) based on Tomasulo's Algorithm remains an ongoing development due to its complexity.

This project provided valuable insights into instruction pipelining, hazard detection, and FPGA-based processor implementation. Future work will focus on integrating Tomasulo's Algorithm for out-of-order execution, optimizing pipeline stalls, and enhancing memory performance with caching techniques.

The successful implementation of the RV32IF processor reinforces the importance of custom processor design tailored for performance and flexibility, making it a stepping stone for advanced research in RISC-V-based architectures and modern computing systems.

References

[1] https://five-embeddev.com/riscv-user-isa-manual/Priv-v1.12/intro.html

[2] Palnitkar, S. (2003). *Verilog HDL: A Guide to Digital Design and Synthesis*. Prentice Hall.

[3] Vahid, F. (2010). *Digital Design with RTL Design, VHDL, and Verilog*. Wiley.

[4] Harris, D., & Harris, S. (2012). *Digital Design and Computer Architecture*. Morgan Kaufmann.

[5] NPTEL Videos- https://archive.nptel.ac.in/courses/106/105/106105163/