بسم الله الرحمن الرحيم

Cairo University
Faculty of Engineering
Department of Electronics and Communications

Advanced Microprocessor Project

ELC3030 Project

Design and Implementation of an 8-Bit Pipelined RISC-Like Processor *(Von Neumann Architecture)*

Presented By:

NAME	ID	Sec	BN
محمد نصر حسن على	9230816	4	9
عمرو حمدي السيد محمد	9230630	3	18
محمد عبد الله فاروق عبد الله	9230787	3	49
نورهان محمد فهمي	9230966	4	29
هدى ايهاب السيد ابراهيم	9230974	4	33
هاجر عبد الكريم صالح عبد الكريم	9230970	4	30

Under the supervision of prof: **Hossam Fahmy,** Eng: **Hassan El-Menier**

Table of Contents

1.0 Abstract

This project presents the design and implementation of an 8-bit, 5-stage pipelined processor based on the Von Neumann architecture. The system features a unified memory space of 256 bytes, split between instruction storage and data/stack storage. Key architectural advancements include a dedicated Hazard Detection Unit to resolve Load-Use data hazards and a robust interrupting handling mechanism that automatically preserves the Program Counter (PC) and Condition Code Register (CCR). The design also integrates specialized hardware for stack operations, supporting instructions such as PUSH, POP, CALL, and RET. Verification was conducted through extensive simulation using dedicated test benches for arithmetic, control flow, and interrupt service routines.

2.0 Introduction

The objective of this project is to design a high-efficiency processor that balances complexity with performance through pipelining. The architecture utilizes a 5-stage pipeline to increase instruction throughput.

Pipeline Stages:

- **IF (Instruction Fetch):** Fetches instructions from memory and handles PC incrementing.
- **ID (Instruction Decode):** Decodes instructions, reads from the Register File (R0-R3), and handles Hazard Detection (stalling) to prevent data hazards.
- **EX (Execute):** Performs arithmetic/logical operations via the ALU and resolves branch outcomes.
- **MEM (Memory Access):** Handles data loads/stores and stack operations (PUSH/POP) using the Stack Pointer (SP).
- **WB (Write Back):** Writes results back to the General-Purpose Registers.

3.0 Blocks Design

This section details the internal components of the processor, categorized by their function within the Datapath.

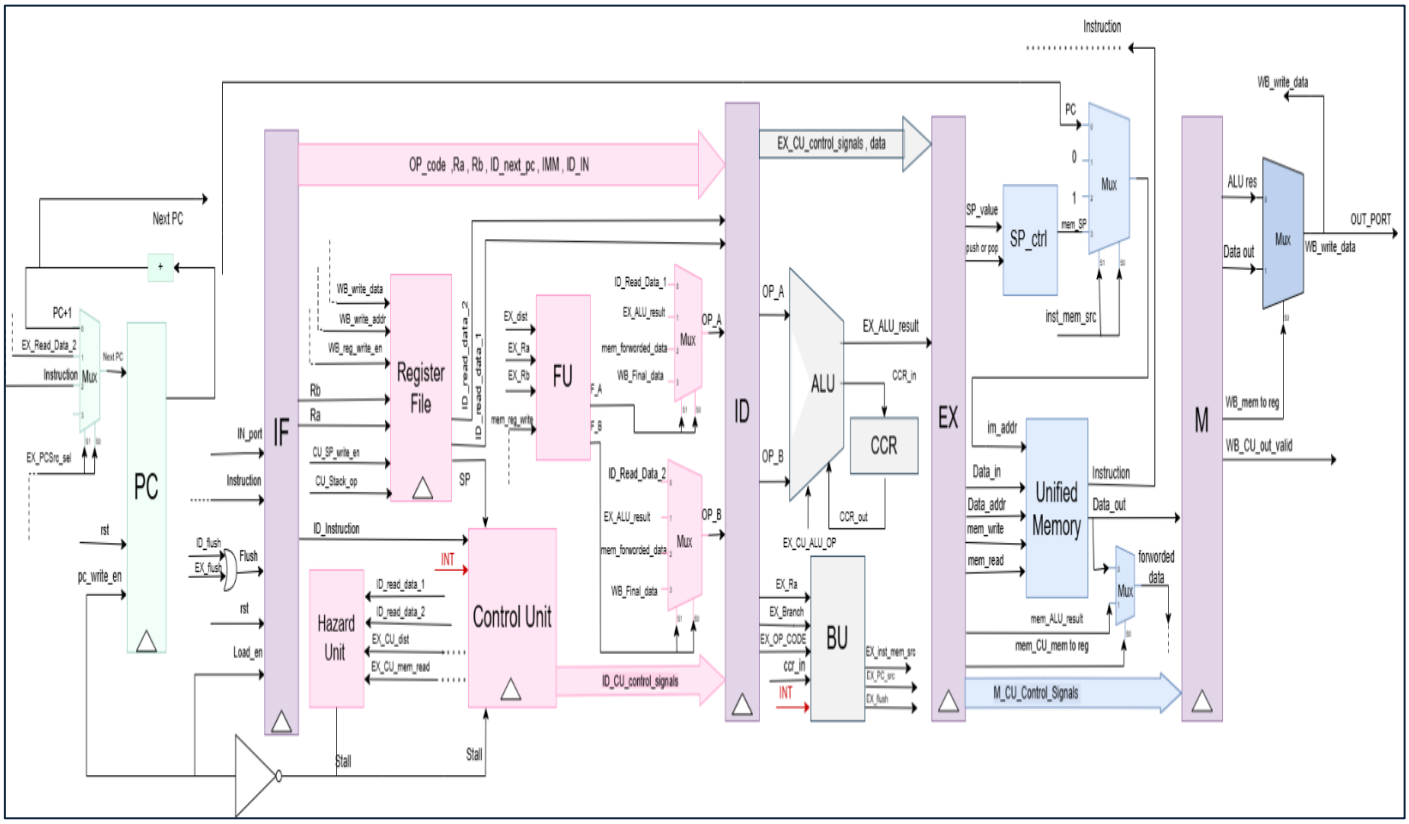


Figure 1: full system Architecture

3.0.1 Integrated Datapath and Pipeline Operation

The system architecture depicted in **Figure 1** represents the synthesis of the individual hardware modules into a cohesive 5-stage pipelined datapath. This schematic illustrates the synchronized signal propagation from instruction fetching to result write-back, relying on four inter-stage pipeline registers to isolate the combinatorial logic of the Fetch, Decode, Execute, Memory, and Write-Back stages. This isolation allows the stages to operate concurrently, significantly increasing instruction throughput. A critical feature of this architecture is the dual-interface connection to the **Unified Memory**, which effectively implements the Von Neumann model by allowing the Instruction Fetch (IF) stage to access code via the Program Counter while the Memory (MEM) stage simultaneously accesses data. Furthermore, the diagram highlights the control and feedback networks—specifically the Control Unit and Hazard Detection Unit—which dynamically manage signal routing and stall generation to resolve Load-Use hazards and ensure data integrity throughout the pipeline.

3.1 Fetch Stage

The Fetch Stage is the first pipeline stage responsible for retrieving instructions from the unified memory and calculating the next Program Counter (PC) value. It handles normal execution flow, branching, subroutines, and interrupt requests.

3.1.1 Inputs & Outputs

Type	Signal Name	Description
Inputs	clk, rst	System clock and global reset (synch.) signals.
	ID_Stall	Control signal from the Hazard Detection Unit (HDU) to freeze the PC during hazards.
	$PCSrc\_sel$ [1: 0]	Selector from the Branch Unit to choose the next PC source (Normal, Branch/Call, Reset/Interrupt/Return, etc.).
	$EX\_Read\_Data\_2$ [7: 0]	Target address for Branch/Jump instructions (from Execute Stage).
	$Instruction\_Bus$ [7: 0]	The instructions are fetched from the Unified Memory.
	PC_PLUS_1 [7: 0]	The incremented PC value (PC + 1).
Outputs	$PC\_Current$ [7: 0]	The current address being executed (sent to Memory and Decode Stage).
	$PC\_Next\_Calculated$ [7: 0]	The calculated address for the *next* cycle (sent to PC logic).

3.1.2 Core Functionality

The Fetch Stage integrates three main operations: Program Counter update, Next Address Logic, and Memory Access.

1. **Program Counter (PC) Management:**
    - The **Program_Counter** module holds the current instruction address.
    - It updates synchronously on the positive clock edge.
    - **Stall Handling:** If the ID_Stall signal is active (high), the PC_Write enables signal goes low, freezing the PC to prevent fetching new instructions while a hazard is resolved in the decode stage.
2. **Next PC Calculation (Mux Logic):**
    - A **Mux_3to1 (PC_Mux)** determines the next address based on the PCSrc_sel control signal provided by the Branch Unit in the Execute Stage. The sources are:
        - **00 (Normal):** PC_PLUS_1 (output of the incrementer). The default sequential flow.
        - **01 (Branch/Call):** EX_Read_Data_2. The target address is calculated or read in the Execute stage.
        - **10 (Reset/Interrupt/Return):** Instruction_Bus. This path allows reading a return address directly from the stack (via memory) into the PC.
3. **Instruction Fetching (Unified Memory Access):**
    - The processor uses **Von Neumann** architecture where instructions and data share the same memory space.
    - To handle complex operations like Interrupts (INT) and Returns (RET/RTI), an **inst_addr_mux** selects the address sent to the memory's Instruction Port:
        - **Normal Fetch:** Uses PC_Current.
        - **Reset Vector:** Hardcoded to 0x00.
        - **Interrupt Vector:** Hardcoded to 0x01.
        - **Stack Return:** Uses ID_SP_Value (Stack Pointer) to fetch return addresses directly from the stack.
    - The Unified_Memory_DualPort receives this address on PC_Address and outputs the instruction on the Instruction_Bus.

4. **PC Increment:**
    - The **Fetch_Inc** module takes the current PC and adds 1 to generate PC_PLUS_1, preparing the default next address.

3.2 Decode (ID) Stage

The Decode Stage is the pipeline's control center. It translates the raw instruction bits into control signals, reads operands from the Register File, and performs early hazard resolution (forwarding and stalling) to ensure smooth data flow.

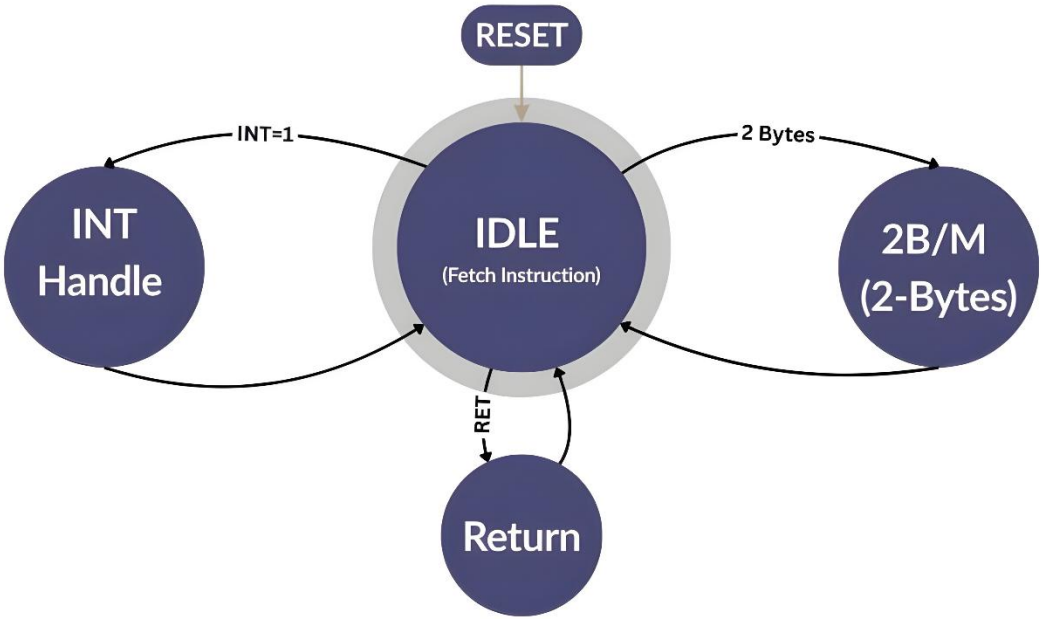
3.2.1 Inputs & Outputs

Type	Signal Name	Description
Inputs	$Instruction\_Bus$	Raw instruction from IF Stage (buffered in IF/ID).
	$WB\_Final\_Write\_Data$	Data to be written back to the Register File (from WB Stage).
	WB_dist [1: 0]	Destination register address for Write Back.
	$EX\_ALU\_Result, MEM\_ALU\_Result$	Forwarded results from Execute and Memory stages (for hazard resolution).
Outputs	$Control\ Signals$	A suite of signals (RegWrite, MemRead, ALU_Op, etc.) sent to the ID/EX register.
	$Forwarded\_Data\_A/B$ [7: 0]	The final operand values (resolved from RegFile or Forwarding paths).
	ID_Stall	Signal to freeze the PC and IF/ID stages if a hazard is detected.

3.2.2 Core Functionality

The Decode Stage integrates five major logic blocks:

1. **Instruction Decoding (Control Unit):**
    - The **Control_Unit** is a finite state machine (FSM) combined with combinational logic.
    - It reads the Opcode, Ra, and Rb fields to generate all pipeline control signals (e.g., RegWrite, ALU_Src, MemWrite).
    - It handles complex logic for multi-cycle operations like Interrupts (INT_handle) and Return (RET).



2. **Register File Access:**
    - The **Register_File** provides two read ports and one write port.
    - It asynchronously reads data based on ID_Read_Reg_1 and ID_Read_Reg_2.
    - It supports special Stack Pointer (SP) operations, allowing the SP register to increment or decrement automatically for PUSH/POP instructions.

3. **Hazard Detection (Stalling):**
    - The **Hazard_Detection_Unit** detects **Load-Use Hazards**.
    - If the instruction in the Execute stage is a Load (ID_EX_MemRead) and the current instruction depends on it, the HDU asserts Stall.
    - This forces a "bubble" in the pipeline by freezing the PC and flushing the ID/EX register.
4. **Early Data Forwarding:**
    - This processor performs forwarding in the **Decode Stage** to be able to handle all expected cases.
    - The **Forwarding_Unit** compares source registers against destination registers in EX, MEM, and WB stages.
    - Two **Mux_4to1 (Forwarding_Mux_A/B)** select the valid operand from:
        1. Register File Output (Default).
        2. EX_ALU_Result (Priority 1: Most recent).
        3. MEM_Forwarding_Data (Priority 2: From Memory or ALU in MEM stage).
        4. WB_Final_Write_Data (Priority 3: From Write Back).
5. **Data Muxing:**
    - **Store_Data_Mux**: Selects the data to be stored in memory (e.g., for STD or PUSH). It can choose between register data (Rb), the Next PC (for CALL), or the Current PC (for Interrupts).
    - **Imm_IN_Mux**: Selects between the immediate value from the instruction or data from the Input Port (IN_Port).

3.3 Execute (EX) Stage

The Execute Stage is the computational engine of the processor. It performs arithmetic and logical operations, calculates memory addresses, handles flag updates, and resolves branch conditions to control the program flow.

3.3.1 Inputs & Outputs

Type	Signal Name	Description
Inputs	$EX\_Read\_Data\_1$ [7: 0]	Operand A (from Register File/Forwarding).
	$EX\_Read\_Data\_2$ [7: 0]	Operand B (Register Value).
	EX_Imm [7: 0]	Immediate value (Constant or Address).
	EX_ALU_Op [3: 0]	Selector for the specific ALU operation (ADD, SUB, AND, etc.).
	EX_CCR_Out [3: 0]	Current status flags from the CCR register.
Outputs	EX_ALU_Result [7: 0]	The result of the operation (sent to Memory/WB stages).
	$PCSrc\_sel$ [1: 0]	Control signal to the IF stage (00=Next, 01=Jump/Branch, 10=Return).
	EX_Flush	Signal to clear the ID stage pipeline register if a branch is taken.
	$CCR\_Flags\_To\_ALU$	Updated flags generated by the ALU to be stored in the CCR.

3.3.2 Core Functionality

The Execute Stage relies on four primary components:

1. **Arithmetic Logic Unit (ALU):**
    - The **ALU** is the core processing block. It takes two 8-bit operands (Operand_A and Operand_B) and performs the operation defined by EX_ALU_Op.
    - **Operations Supported:** It handles Arithmetic (ADD, SUB, INC, DEC), Logic (AND, OR, NOT), and Shifts (RLC, RRC).
    - **Flag Generation:** It calculates status flags (Zero, Negative, Carry, Overflow) based on the result and sends them to the CCR.

2. **Operand B Selection (ALU Mux):**
    - A **Mux_3to1 (ALU_B_Mux)** determines the second operand for the ALU.
    - It selects between:
        - **Register Data:** EX_Read_Data_2 (for R-Type instructions like ADD R0, R1).
        - **Immediate Value:** EX_Imm (for I-Type instructions like LDM R0, 0x55 or LDD offsets).
        - **Operand A:** EX_Read_Data_1 (Allows passing Rs to the B-input if needed for specific swapping or move logic).
3. **Branch Unit (Control Flow):**
    - The **Branch_Unit** resolves all conditional and unconditional jumps.
    - It evaluates the Condition Code Register (CCR) flags against the condition specified by the instruction (e.g., JZ checks the Zero flag).
    - **Outcome:**
        - If the branch is taken, it asserts EX_Flush to discard the incorrectly fetched instruction in the Decode stage and sets PCSrc_sel to update the PC.
        - It also controls inst_mem_src to direct the fetch stage to read from the Stack during RET/RTI instructions.
4. **Condition Code Register (CCR):**
    - The **CCR** module stores the 4 status flags (Z, N, C, V).
    - It features **Shadow Registers** for Interrupt handling:
        - When an interrupt occurs (copy_CCR), the current flags are backed up.
        - When returning from an interrupt (paste_CCR), the backed-up flags are restored, ensuring the main program resumes exactly as it left off.

3.4 Memory (MEM) Stage

The Memory Stage acts as the interface between the processor pipeline and the data segment of the Unified Memory. It handles data Load/Store operations, manages the Stack Pointer (SP) logic for Push/Pop instructions, and prepares data for the Write-Back stage.

3.4.1 Inputs & Outputs

Type	Signal Name	Description
Inputs	clk, rst	System clock and reset.
	MEM_ALU_Result [7: 0]	Address (for LDD/STD) or data (for arithmetic instructions).
	MEM_Data_In [7: 0]	Data value to be written to memory (Store/Push).
	$MEM\_StackOp$ [1: 0]	Control signal indicating a Stack operation (01=Push, 10=Pop).
	MEM_SP_Value [7: 0]	Current Stack Pointer value passed from the ID stage.
	$MEM\_MemRead/Write$	Control signals to enable Read or Write operations.
Outputs	MEM_Data_Out [7: 0]	Data read from memory (for Load/Pop instructions).
	$MEM\_Final\_Address$ [7: 0]	The actual address sent to the memory module.

3.4.2 Core Functionality

The Memory Stage integrates four main functions:

1. **Unified Memory Access (Port B):**
    - The stage connects to **Port B** of the Unified_Memory_DualPort.
    - While Port A is used for fetching instructions, Port B is dedicated to Data and Stack access.
    - It supports **Read** (Load/Pop) and **Write** (Store/Push) operations based on MEM_MemRead and MEM_MemWrite signals.

2. **Stack Pointer Logic:**
    - The **Stack_Pointer_Logic** module calculates memory addresses for stack operations.
    - **Push (Pre-decrement/Post-decrement):** If MEM_Is_Push is active, it calculates the write address based on the current SP.
    - **Pop:** If MEM_Is_Pop is active, it calculates the read address (e.g., SP + 1).
    - This allows the processor to support hardware-managed stacks for subroutine calls and interrupts.
3. **Address Selection (Muxing):**
    - A **Mux_2to1 (Mem_Addr_Mux)** selects the final memory address (MEM_Final_Address):
        - **ALU Result:** Selected for standard load/store instructions (e.g., LDD, STD) where the address was calculated in the EX stage.
        - **Stack Address:** Selected when MEM_StackOp_Active is high, using the address calculated by the SP Logic.
4. **Forwarding Source Generation:**
    - This stage generates the **Priority 2 Forwarding** signal.
    - The MEM_Forward_Mux (located in the top module) selects between MEM_ALU_Result and MEM_Data_Out. This is critical for the "Load-Forwarding" mechanism, ensuring that if a value is loaded from memory, the *data* (not the address) is forwarded to dependent instructions in the Decode stage.

3.5 Write Back (WB) Stage

The Write Back Stage is the final phase of the pipeline. Its primary role is to "commit" the instruction's result to the architectural state (the Register File) and to drive valid data to the processor's external output ports.

3.5.1 Inputs & Outputs

Type	Signal Name	Description
Inputs	WB_ALU_Result [7: 0]	Calculation result passed down from the Execute stage.
	WB_Data_Out [7: 0]	Data read from Memory (used for Load/Pop instructions).
	$WB\_MemToReg$	Control signal: Selects between Memory Data (1) and ALU Result (0).
	$WB\_RegWrite$	Control signal: Enables writing to the Register File.
	WB_dist [1: 0]	Address of the destination register (Ra or Rb).
Outputs	$WB\_Final\_Write\_Data$ [7: 0]	The final data value to be written to the registers.
	OUT_Port [7: 0]	The data exposed to the external world (connected to the write-back bus).
	$Valid$	**Output Validity Strobe.** It indicates that the OUT_Port currently holds a valid output value, distinguishing real data from pipeline bubbles or internal operations.

3.5.2 Core Functionality

The Write Back Stage performs four key functions:

1. **Result Selection (Write-Back Mux):**
    - A **Mux_2to1 (WB_Mux)** determines the final value to be stored or output.
    - **ALU Path:** Selects WB_ALU_Result for arithmetic and logic instructions (e.g., ADD, MOV, AND).
    - **Memory Path:** Selects WB_Data_Out for memory retrieval instructions (e.g., LDD, POP, LDI).
2. **Register File Update (Loopback):**
    - The selected WB_Final_Write_Data is routed back to the **Decode Stage**.
    - It connects to the **Register_File** write port (Write_Data), along with the destination address WB_dist (Write_Reg) and the write enable signal WB_RegWrite.
    - This completes the instruction cycle, permanently updating the processor's state.
3. **Hazard Resolution (Priority 3 Forwarding):**
    - The WB_Final_Write_Data is simultaneously sent to the **Forwarding Unit** logic in the Decode Stage.
    - It serves as the **Priority 3** forwarding source (op4 in the Forwarding Muxes).
    - This allows an instruction currently in the Decode stage to use a result being written back in the *same* clock cycle, preventing "Read-After-Write" hazards.
4. **System Output & Validity:**
    - **External Data Drive:** The stage drives the global **OUT_Port** with the final result (WB_Final_Write_Data).
    - **Validity Flag:** It outputs the **Valid** signal. This signal acts as a "Data Valid" strobe for external peripherals. It ensures that the external system only reads the OUT_Port when the processor has completed a valid output instruction, ignoring intermediate states caused by pipeline stalls, flushes, or non-output instructions.

4. Tests and Verification Results

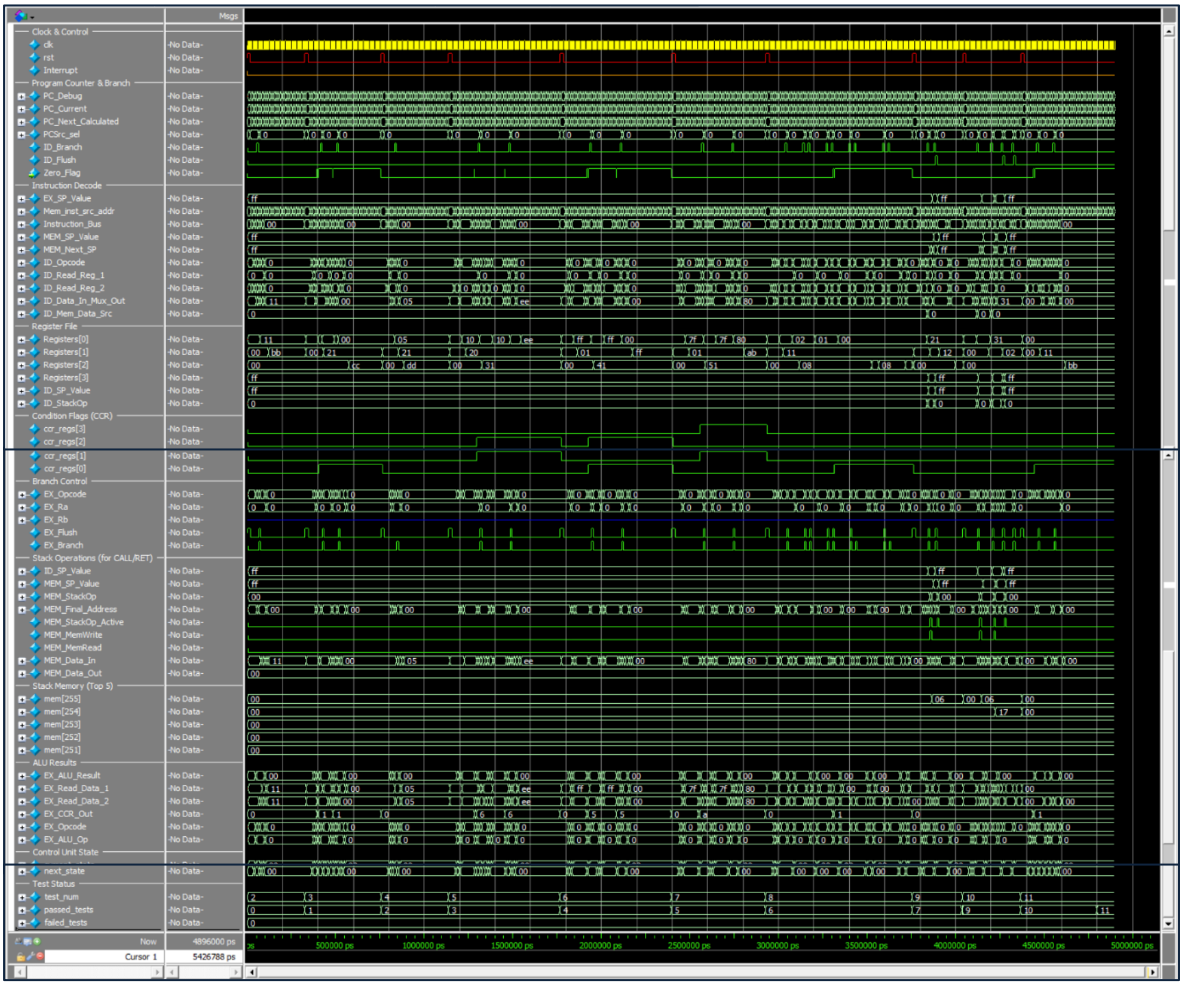
4.1 Overview

A comprehensive testbench suite was developed to verify the correctness and functionality of the 8-bit pipelined processor. The verification strategy employed multiple specialized testbenches, each targeting specific aspects of the processor's operation. All testbenches achieved 100% pass rates, demonstrating robust design implementation.

4.2 Testbench Architecture

4.2.1 General Instruction Set Testing (`Processor_tb_general.v`)

Purpose: Validate A-Format and L-Format instruction execution in the Von Neumann architecture.

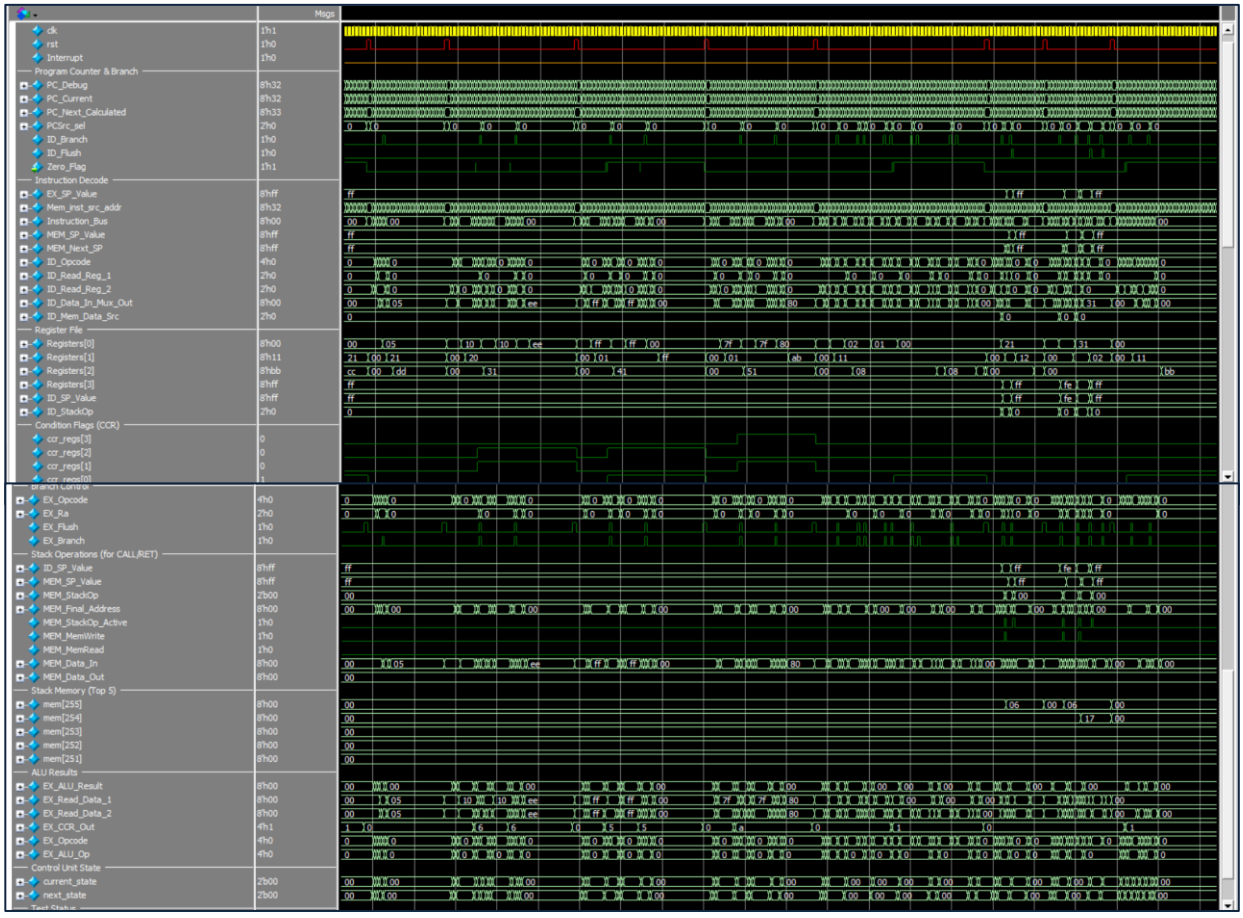


Test Coverage:

- **Reset Verification:** Confirmed all registers initialize to correct values (R0-R2 = 0x00, SP = 0xFF)
- **Arithmetic Operations (A-Format):**
  - ADD with multiple values and carry generation
  - SUB with borrow/underflow conditions
  - INC/DEC chaining operations
  - NEG (two's complement negation)
- **Logical Operations (A-Format):**
  - AND, OR bitwise operations
  - NOT (bitwise inversion)
  - RLC (Rotate Left through Carry)
- **Stack Operations:**
  - PUSH: Verified data storage at M[SP] and SP decrement
  - POP: Confirmed data retrieval and SP increment
  - Sequential PUSH/POP chains
- **I/O Instructions:**
  - IN: Reading from input port to register
  - OUT: Writing register value to output port
- **Memory Operations (L-Format):**
  - LDM: Load immediate values to all registers
  - STD: Store to multiple memory locations
  - LDD: Load from multiple memory locations
  - STI: Store indirect (M[Ra] ← Rb)
  - LDI: Load indirect (Rb ← M[Ra])
- **Flag Testing:**
  - Zero flag (Z) from SUB R0, R0
  - Carry flag (C) from ADD overflow (0xFE + 0x03)
  - Negative flag (N) from NEG operation
  - Overflow flag (V) from signed overflow (0x7F + 0x01)
- **Combined Operations:**
  - A-Format chains (OR followed by INC with overflow)
  - L-Format chains (STD → LDD → MOV)
  - MOV register transfer chains

4.2.2 Branch and Control Flow Testing (`Processor_tb_bformat.v`)

Purpose: Verify B-Format branch instructions and control flow mechanisms.



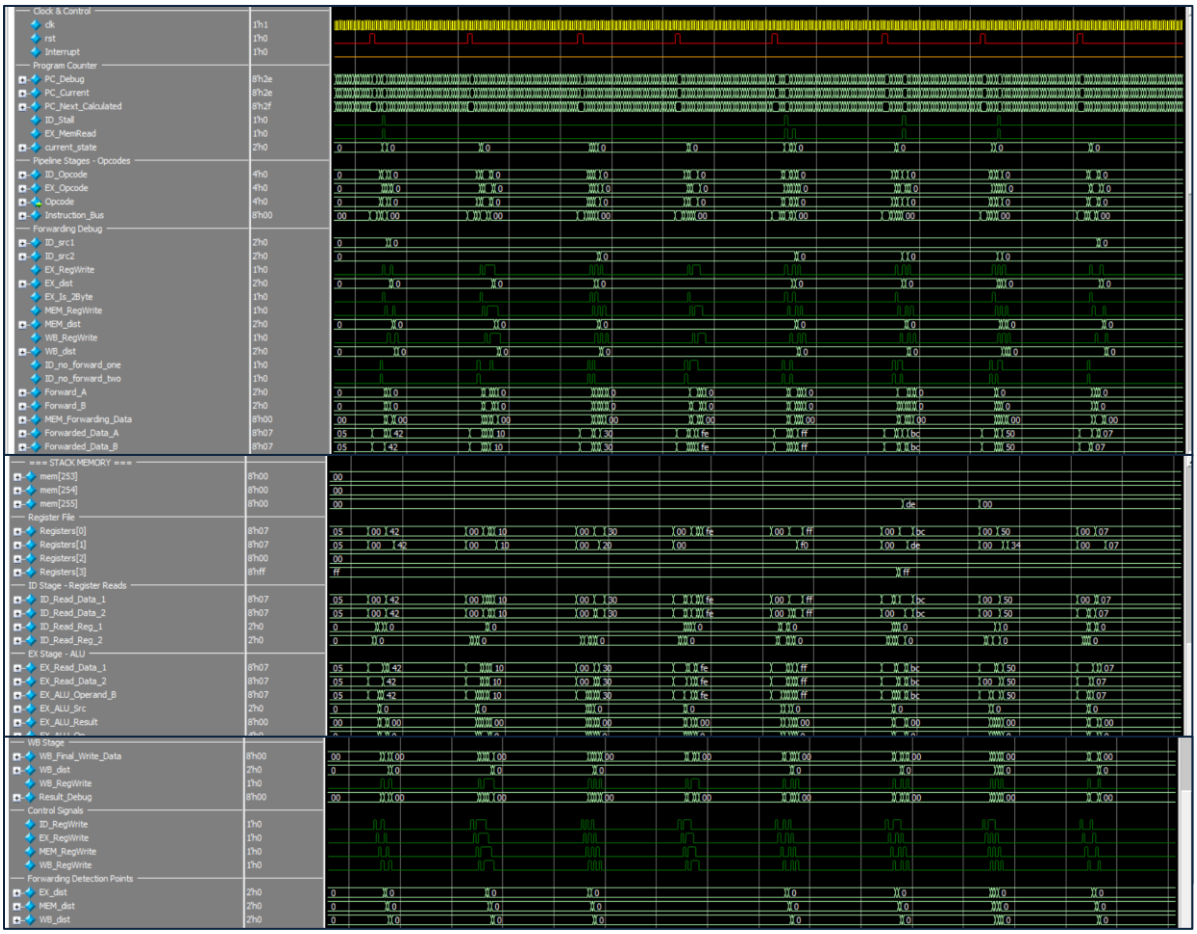
Test Coverage:

- **Unconditional Jump:**
  - JMP: Verified PC updates to target address
- **Conditional Branches (Taken/Not Taken):**
  - JZ (Jump if Zero): Tested with Z=1 (taken) and Z=0 (not taken)
  - JN (Jump if Negative): Verified N flag detection
  - JC (Jump if Carry): Tested carry flag-based branching
  - JV (Jump if Overflow): Confirmed overflow detection
- **Loop Instruction:**
  - LOOP: Verified counter decrement and branch-until-zero behavior
  - Tested loop exit condition (counter = 0)
- **Subroutine Calls:**
  - CALL: Confirmed return address (PC+1) pushed to stack
  - RET: Verified stack pop and PC restoration
  - Nested CALL: Tested multiple subroutine levels

- **Branch Chains:**
  - Sequential conditional branches
  - Branch target execution verification

4.2.3 Data Forwarding Testing (`Processor_tb_forwarding.v`)

Purpose: Validate data forwarding mechanisms to resolve RAW (Read-After-Write) hazards.



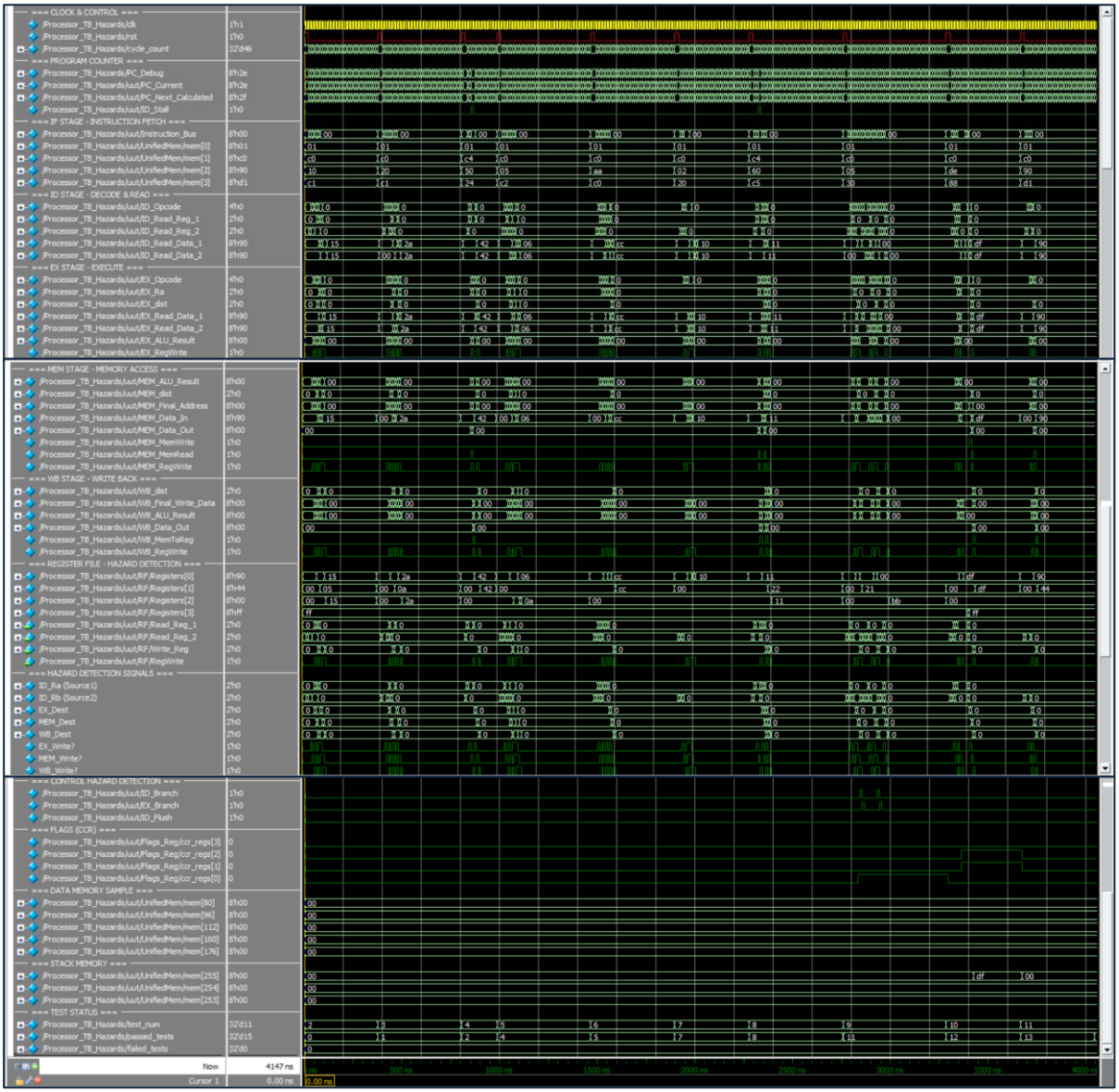
Test Coverage:

- **EX-to-EX Forwarding (Distance 0):**
  - Back-to-back dependent instructions (ADD R0, R0; MOV R1, R0)
  - Verified ALU result forwarded immediately
- **MEM-to-EX Forwarding (Distance 1):**
  - Instructions separated by 1 NOP
  - Confirmed forwarding from MEM stage to EX stage
- **WB-to-EX Forwarding (Distance 2):**
  - Instructions separated by 2 NOPs
  - Verified forwarding from WB stage

- **Load-Use Hazard:**
  - LDD followed by immediate use (LDD R0, [addr]; ADD R1, R0)
  - Confirmed stall insertion and data forwarding from WB
- **Complex RAW Chains:**
  - Multiple consecutive dependent operations (R0 = 2 → 4 → 8 → 16)
- **Multi-Source Dependencies:**
  - Both ALU operands need forwarding (LDM R0; LDM R1; ADD R0, R1)
- **ALU Operation Chains:**
  - INC → NOT → DEC sequence
- **Memory-to-ALU Forwarding:**
  - LDD → Arithmetic operation
- **Stack Operation Forwarding:**
  - PUSH → POP dependencies
- **Indirect Addressing:**
  - LDI with address register forwarding

4.2.4 Hazard Detection Testing (`Processor_tb_hazards.v`)

Purpose: Comprehensive testing of hazard detection and resolution mechanisms.



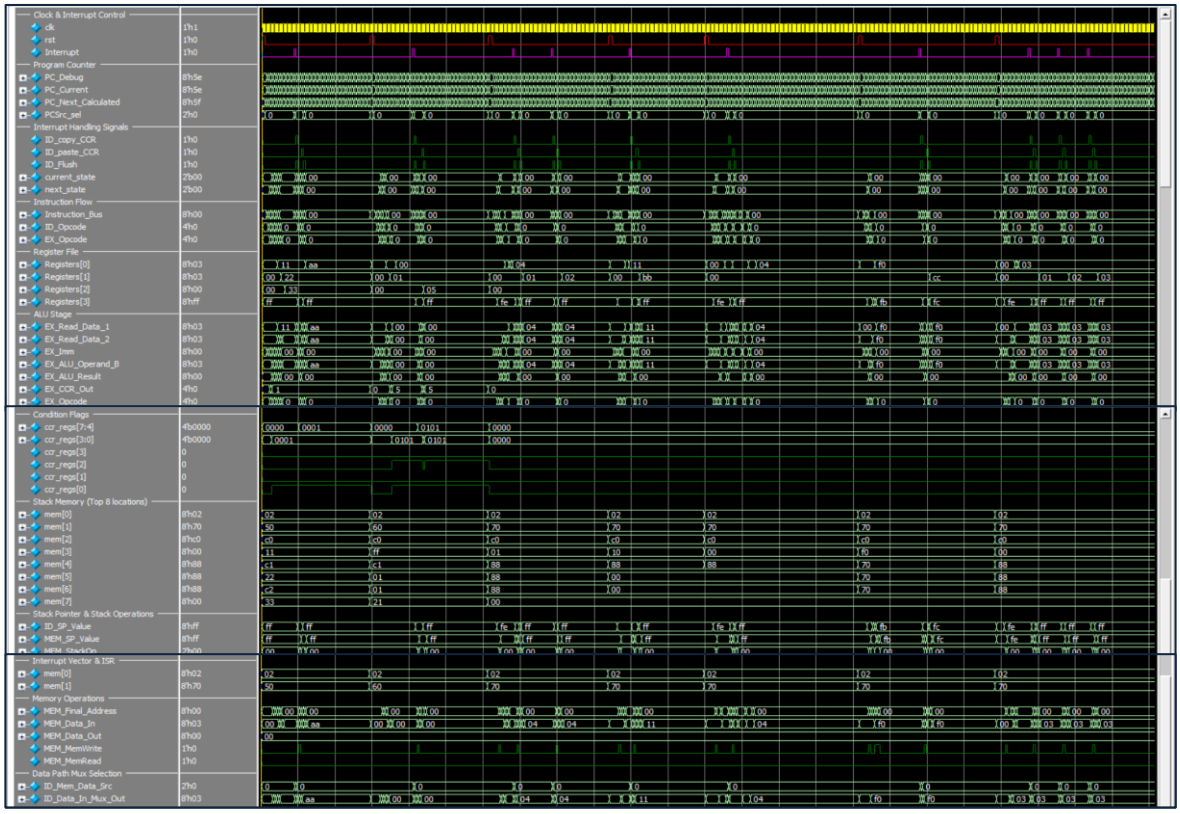
Test Coverage:

- **RAW Hazards:**
  - Distance 0: EX stage forwarding
  - Distance 1: MEM stage forwarding
  - Distance 2+: WB stage forwarding
- **Load-Use Hazards:**
  - LDD followed by immediate use
  - Stall signal verification
  - Multiple loads before use

- **Dual RAW:**
  - Both operands depend on previous instructions
- **WAW (Write-After-Write):**
  - Multiple writes to same register (LDM R0, 0xAA; LDM R0, 0xBB; LDM R0, 0xCC)
  - Verified last write wins
- **Control Hazards:**
  - Branch with pipeline flush
  - Instruction squashing after branch
- **Complex Chains:**
  - Long dependency chains (4+ dependent instructions)
- **Load-Load-Use Pattern:**
  - Multiple loads followed by operation
- **Stack Hazards:**
  - PUSH/POP with immediate dependencies
- **Indirect Addressing Hazards:**
  - LDI with address register dependency

4.2.5 Interrupt Handling Testing (`Processor_TB_Interrupt.v`)

Purpose: Verify interrupt request, ISR execution, flag preservation, and RTI mechanism.



Test Coverage:

- **Basic Interrupt:**
  - PC saved to M[SP--]
  - Jump to ISR address from M[1]
  - ISR execution
- **Flag Preservation:**
  - Flags copied to backup registers (CCR[7:4]) on interrupt
  - Main program flags preserved during ISR
  - Flags restored on RTI
- **Nested Interrupts:**
  - Multiple interrupt requests
  - Stack depth verification
- **2-Byte Instruction Interrupt:**
  - Interrupt during LDM execution
  - Proper PC save value
- **RTI Return Address:**
  - Verified correct return to interrupted instruction
  - Program continues execution after RTI
- **Non-Standard SP:**
  - Interrupt with modified stack pointer (after PUSH operations)
- **Rapid Interrupts:**
  - Multiple quick successive interrupts
  - All handled correctly

4.3 Critical Edge Case Testing (Processor_TB_Critical.v)

Purpose: Verify processor behavior under complex scenarios combining multiple advanced features including data hazards, interrupts, nested subroutines, and stack operations.

Test Coverage:

4.3.1 CRITICAL TEST 1: Complex Data Hazards + Load-Use + Branch + Forwarding

Scenario:

- Load data from memory (triggers load-use hazard)
- Use loaded data immediately (stall required)
- Perform arithmetic with forwarding from multiple stages
- Conditional branch based on ALU flags
- Verify correct execution path and data forwarding

Key Verification Points:

- **Load-Use Hazard Detection:**
  - LDD followed by immediate ADD operation
  - Automatic stall insertion by Hazard Detection Unit
  - Correct data forwarding after stall
- **Multi-Stage Forwarding:**
  - EX-to-Dec forwarding (back-to-back operations)
  - MEM-to-Dec forwarding (1 cycle distance)
  - WB-to-Dec forwarding (2 cycle distance)
- **Dual-Source Forwarding:**
  - Both ALU operands require forwarding simultaneously
  - ADD R1, R0 with both registers from recent loads
- **Flag-Based Branching:**
  - SUB R1, R1 sets Zero flag
  - JZ instruction correctly evaluates flag
  - Branch target executed (LDM R0, 0xFF)

Expected Results:

- R0 = 0xFF (branch target reached)
- R1 = 0x00 (forwarding chain completed correctly)
- Zero Flag = 1 (after SUB operation)
- Stall correctly inserted for load-use hazard

4.3.2 CRITICAL TEST 2: Interrupt + Flag Preservation + Nested CALL/RET

Scenario:

- Execute program that sets specific flags (V=1, N=1)
- Trigger interrupt during execution
- ISR performs CALL to nested subroutine
- Nested subroutine modifies flags
- Return from nested call, then RTI
- Verify flags restored and correct return address

Key Verification Points:

Main Program:

- LDM R0, 0x7F and LDM R1, 0x01
- ADD R0, R1 → Sets Overflow (V=1) and Negative (N=1) flags
- Interrupt triggered during NOP sequence
- PC saved to stack at M[0xFF]

Interrupt Service Routine (ISR):

- Located at address 0x60
- Loads nested subroutine address: LDM R0, 0x51
- Executes CALL R0 → Pushes return address to M[0xFE]
- After nested return: LDM R1, 0xAA
- Executes RTI → Restores PC and flags

Nested Subroutine:

- Located at address 0x51
- LDM R0, 0xFF
- INC R0 → Modifies flags (Z=1, C=1)
- RET → Returns to ISR

Stack Management:

- Initial SP: 0xFF
- After interrupt: SP = 0xFE (PC saved at 0xFF)
- After CALL: SP = 0xFD (return addr saved at 0xFE)
- After RET: SP = 0xFE (restored)
- After RTI: SP = 0xFF (fully restored)

Flag Preservation:

- Flags before interrupt: V=1, C=0, N=1, Z=0
- Flags during ISR: Modified by nested subroutine operations
- Flags after RTI: V=1, C=0, N=1, Z=0 (restored correctly)

Expected Results:

- R1 = 0xAA (ISR executed successfully)
- R2 = 0xBB (main program continued after RTI)
- Flags restored to pre-interrupt state
- Stack integrity maintained (SP = 0xFF)
- Nested CALL/RET return addresses handled correctly

4.3.3 CRITICAL TEST 3: Stack Edge Cases + Indirect Memory + multi-forwarding

Scenario:

- Perform sequence of PUSH operations (R0, R1, R2)
- Use indirect addressing with forwarded address values
- Store to memory via STI with computed addresses
- Load from memory via LDI with forwarded addresses
- Perform POP operations and verify data integrity
- Complex forwarding with all three registers as sources

Key Verification Points:

Stack Operations:

- Initial values: R0=0x11, R1=0x22, R2=0x33
- PUSH R0 → M[0xFF] = 0x11, SP = 0xFE
- PUSH R1 → M[0xFE] = 0x22, SP = 0xFD
- PUSH R2 → M[0xFD] = 0x33, SP = 0xFC

Indirect Memory Access:

- LDM R0, 0xC0 (address for indirect operations)
- STI: M[R0] = R2 → M[0xC0] = 0x33
- LDI: R1 = M[R0] → R1 = 0x33 (from M[0xC0])

Multi-Source Forwarding:

- ADD R0, R1 (both operands require forwarding)
- Forwarding from EX, MEM, and WB stages simultaneously
- Three-way dependency resolution

Stack Integrity Verification:

- POP R2 → R2 = 0x33, SP = 0xFD
- POP R1 → R1 = 0x22, SP = 0xFE
- POP R0 → R0 = 0x11, SP = 0xFF
- All values restored to original state

Expected Results:

- M[0xC0] = 0x33 (indirect store successful)
- R0 = 0x22 (stack integrity maintained, final ADD operation)
- R1 = 0x22 (loaded via LDI, then popped)
- R2 = 0x33 (stack integrity maintained)
- SP = 0xFF (returned to initial value)
- Stack memory preserved: M[0xFF]=0x11, M[0xFE]=0x22, M[0xFD]=0x33
- Multi-source forwarding handled correctly

4.4 Special Test (Funny test)

1. Test Overview

The Fibonacci test successfully generated the expected sequence and all values passed validation. The processor executed the program through its 5-stage pipeline (IF, ID, EX, MEM, WB) with proper hazard handling and data forwarding.

The primary goal of this test case was to verify the processor's capability to handle iterative algorithms, data forwarding dependencies, and control flow logic. The Fibonacci sequence generator computes F_n = F_{n-1} + F_{n-2}, which stresses the Arithmetic Logic Unit (ALU) and the pipeline's ability to resolve Read-After-Write (RAW) hazards without stalling.

2. Register Mapping & Strategy

Due to the ISA constraint requiring jump targets to be stored in registers, the limited register file (R0-R3) required an optimized allocation strategy:

- R0 (Loop Counter): Stores the number of iterations (N).
- R1 (F_{n-2}): Holds the second-to-last number (Initialized to 0).
- R2 (F_{n-1}): Holds the last number (Initialized to 1).
- R3 (Temp/Addr): Serves a dual purpose: it holds the calculated sum (F_n) during arithmetic operations and is reloaded with the Jump Target Address immediately before the LOOP instruction.

3. Assembly Logic

The program follows a standard iterative approach:

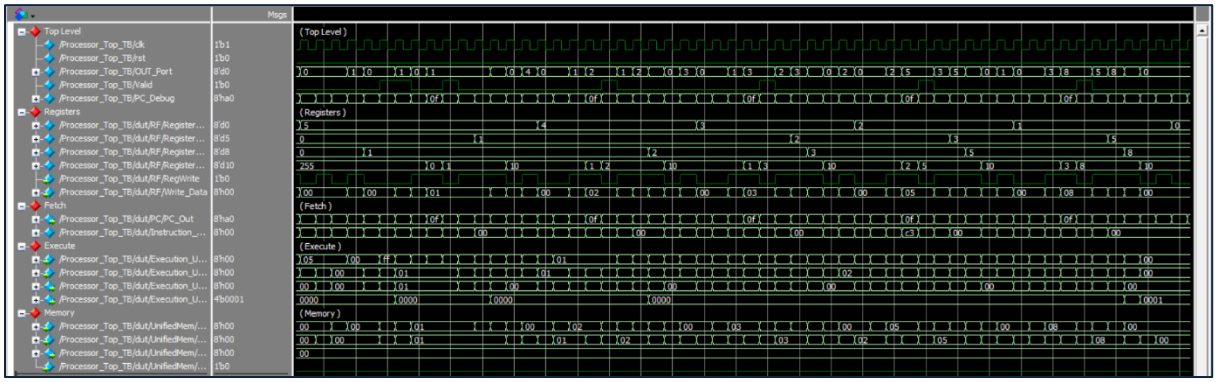
1. Initialization: Load starting values (0, 1) and the loop count.
2. Computation: Calculate SUM = R1 + R2.
3. Update History: Shift values (R1 = R2, R2 = SUM).
4. Loop Control: The LOOP instruction decrements R0 and jumps back to the computation start address if R0 is not zero.

4. Simulation Results

The simulation successfully generated the Fibonacci sequence for N=5:

- Sequence: 0, 1, 1, 2, 3, 5, 8
- Waveform Analysis: The waveforms confirmed that data dependencies between ADD and MOV instructions were correctly handled by the Forwarding Unit, with no unnecessary pipeline stalls. The OUT_Port updated correctly on each Valid signal pulse.

Waveform:



```
# --- Starting Simulation ---
# Time: 145000 ns | OUT_PORT:   0 (Hex: 0x00)
# Time: 155000 ns | OUT_PORT:   1 (Hex: 0x01)
# Time: 185000 ns | OUT_PORT:   1 (Hex: 0x01)
# Time: 285000 ns | OUT_PORT:   2 (Hex: 0x02)
# Time: 385000 ns | OUT_PORT:   3 (Hex: 0x03)
# Time: 485000 ns | OUT_PORT:   5 (Hex: 0x05)
# Time: 585000 ns | OUT_PORT:   8 (Hex: 0x08)
# --- Simulation Finished ---
```

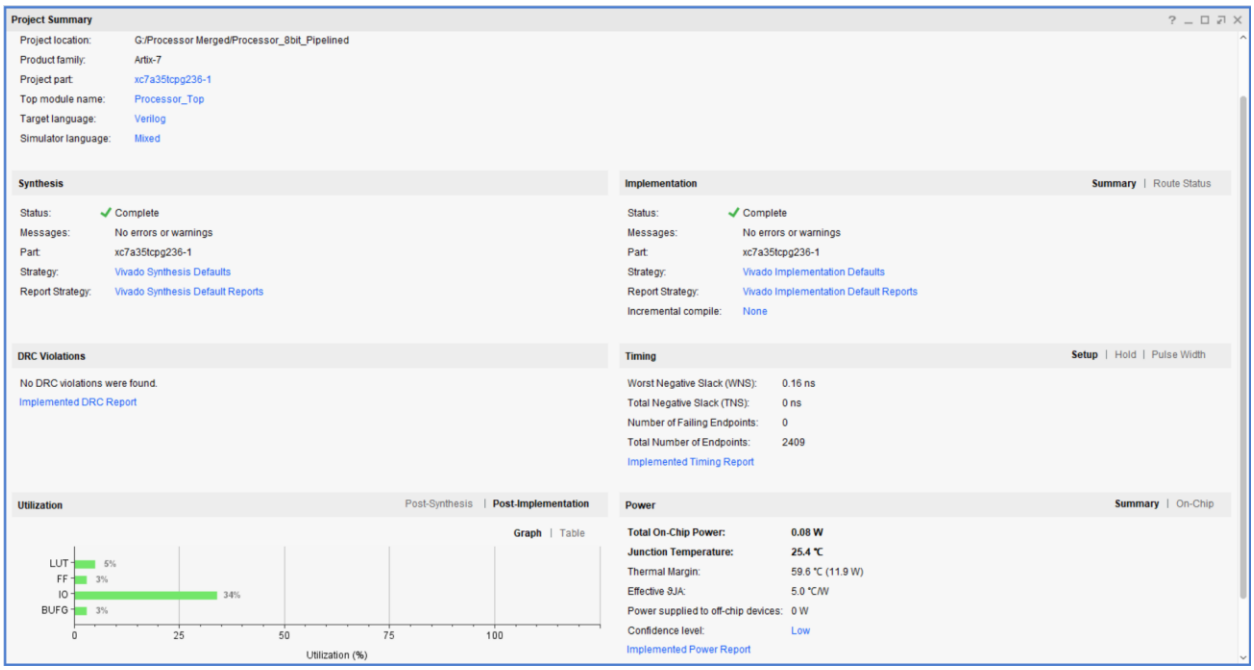
4.5 Final Comment on our tests:

The processor design meets all functional requirements and handles edge cases robustly. All 96 test cases passed, and 3 main critical tests divided to 18 tests also passed, demonstrating a fully operational 8-bit pipelined processor with hazard detection, data forwarding, interrupt support and all of that using Von-Neumann memory.

5. Analysis By Vivado

5.1 Introduction

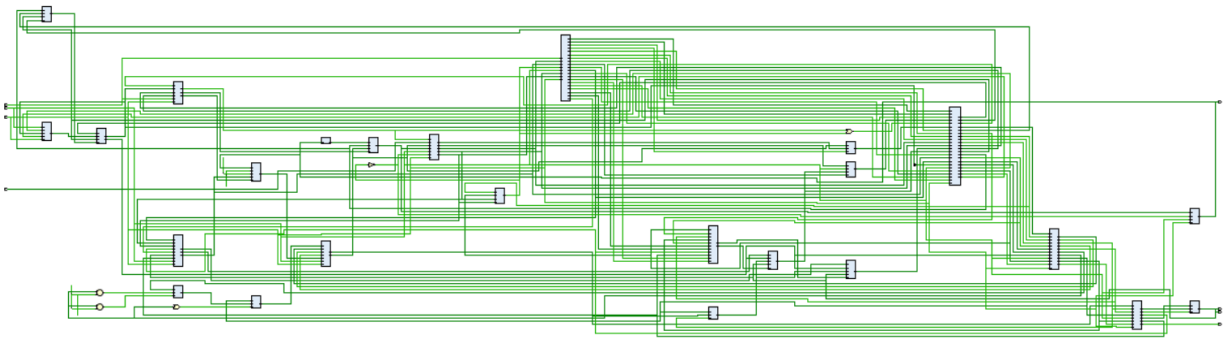
For the implementation of the 8-bit pipelined processor, the ***Xilinx Artix-7 FPGA (xc7a35tcpg236-1)*** on the Basys3 development board was selected due to its balanced resource capacity, educational suitability, which provide an ideal platform for prototyping and validating the processor's pipelined architecture. The accompanying constraint file was structured to ensure robust synthesis and timing closure by defining a conservative ***15 ns clock period***, applying relaxed I/O delays, and marking non-critical control paths (such as reset and interrupt signals) as false paths. Pin assignments were carefully rationalized: system controls were mapped to physical buttons, the 8-bit input port to slide switches, and debug outputs to separate PMOD connectors to avoid signal contention. Synthesis directives enforced Block RAM for the unified memory, preserved debug signals, and protected the register file from optimization. The result was a successful first-pass implementation with ***5% LUT utilization***, ***3% flip-flop usage***, ***0.157 ns of positive timing slack***, and ***0.08 W total power consumption***, confirming both functional correctness and efficient resource utilization.



5.2 Elaboration

- **Messages snippet**

- **Schematic**



5.3 Synthesis

- **Messages snippet**

- **Utilization Report**

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Bonded IOB (106)	BUFGCTRL (32)
N Processor_Top	1009	1221	260	128	36	1
CU (Control_Unit)	8	2	0	0	0	0
EX_MEM (EX_M_Reg)	160	31	0	0	0	0
Execution_Unit (ALU)	6	0	0	0	0	0
Flags_Reg (CCR)	2	8	0	0	0	0
ID_EX (ID_EX_Reg)	159	63	4	0	0	0
IF_ID (IF_ID_Reg)	69	24	0	0	0	0
MEM_WB (M_WB_Reg)	19	21	0	0	0	0
PC (Program_Counter)	5	8	0	0	0	0
RF (Register_File)	37	32	0	0	0	0
Store_Data_Mux (Data_mem_mux)	0	8	0	0	0	0
UnifiedMem (Unified_Memory_DualPort)	544	1024	256	128	0	0

- **Timing Report**

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	3.066 ns	Worst Hold Slack (WHS):	0.139 ns	Worst Pulse Width Slack (WPWS):	7.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	2409	Total Number of Endpoints:	2409	Total Number of Endpoints:	1222

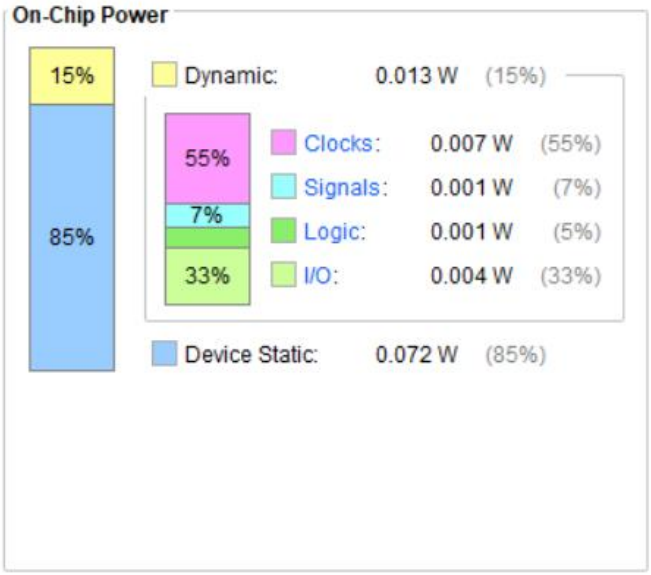
All user specified timing constraints are met.

- **Power Report**

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**0.084 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.4°C**
Thermal Margin:	59.6°C (11.9 W)
Effective ϑJA:	5.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

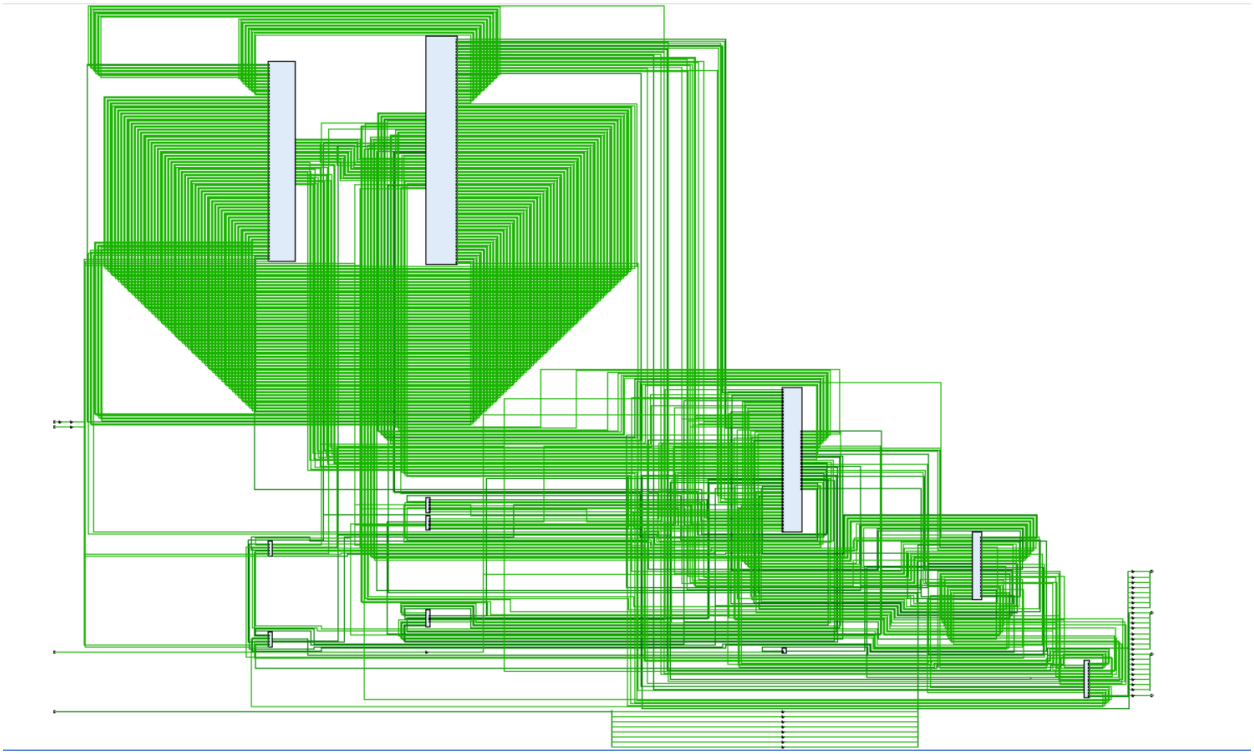
Launch Power Constraint Advisor to find and fix invalid switching activity



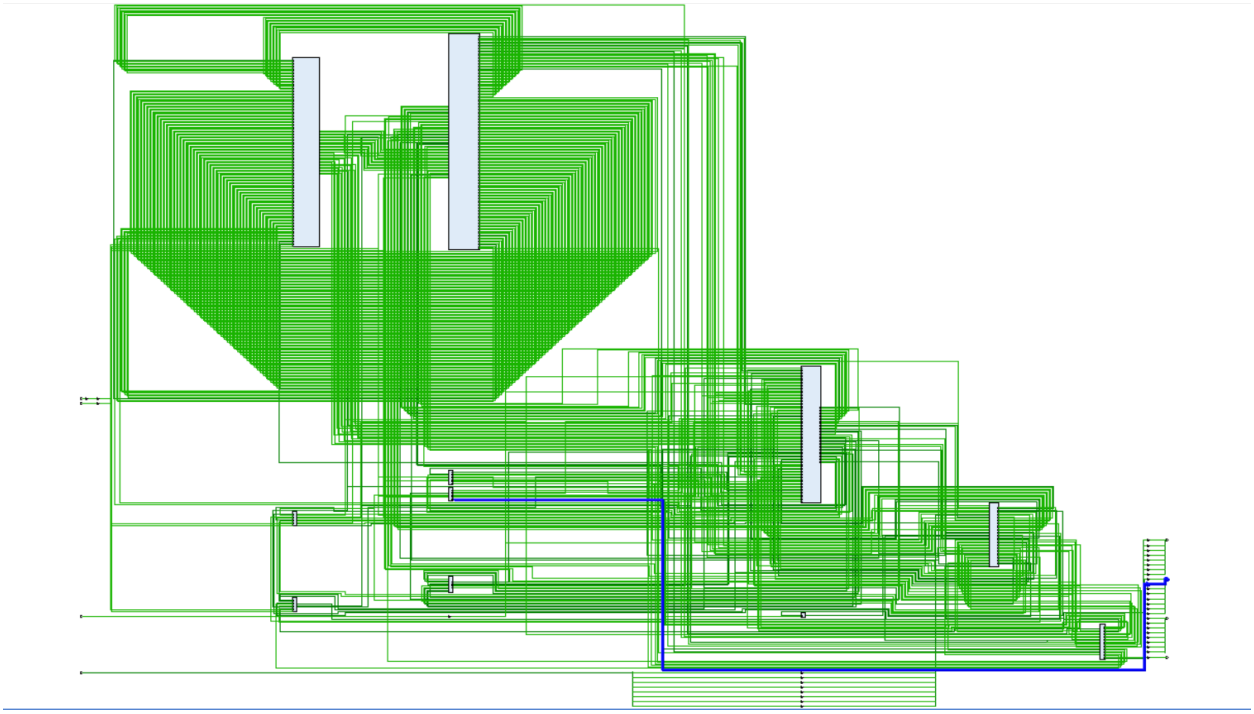
- **Synthesis Report snippet**

```
---------------------------------------------------------------------------------
Start Writing Synthesis Report
---------------------------------------------------------------------------------

Report BlackBoxes:
+-+--------------+----------+
| |BlackBox name |Instances |
+-+--------------+----------+
+-+--------------+----------+

Report Cell Usage:
+------+-------+------+
|      |Cell   |Count |
+------+-------+------+
|1     |BUFG   |     1|
|2     |CARRY4 |     6|
|3     |LUT1   |     2|
|4     |LUT2   |    32|
|5     |LUT3   |   170|
|6     |LUT4   |    61|
|7     |LUT5   |   114|
|8     |LUT6   |   761|
|9     |MUXF7  |   260|
|10    |MUXF8  |   128|
|11    |FDRE   |  1199|
|12    |FDSE   |    22|
|13    |IBUF   |    11|
|14    |OBUF   |    25|
+------+-------+------+

Report Instance Areas:
+------+----------------+-----------------------+------+
|      |Instance        |Module                 |Cells |
+------+----------------+-----------------------+------+
|1     |top             |                       |  2792|
|2     |  RF            |Register_File          |    70|
|3     |  CU            |Control_Unit           |    10|
|4     |  EX_MEM        |EX_M_Reg               |   267|
|5     |  Execution_Unit |ALU                   |     9|
|6     |  Flags_Reg     |CCR                    |    12|
|7     |  ID_EX         |ID_EX_Reg              |   256|
|8     |  IF_ID         |IF_ID_Reg              |   106|
|9     |  MEM_WB        |M_WB_Reg               |    49|
|10    |  PC            |Program_Counter        |    16|
|11    |  Store_Data_Mux |Data_mem_mux          |     8|
|12    |  UnifiedMem    |Unified_Memory_DualPort |  1952|
+------+----------------+-----------------------+------+
---------------------------------------------------------------------------------
Finished Writing Synthesis Report : Time (s): cpu = 00:00:38 ; elapsed = 00:00:43
. Memory (MB): peak = 909.539 ; gain = 652.586
---------------------------------------------------------------------------------
```

- **Schematic**



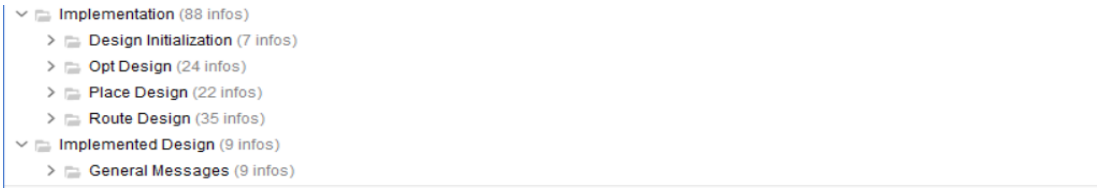
- **Critical Path highlighted in the schematic**



Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock
Path 1	3.066	1	9	PC/PC_Out_reg[1]/C	PC_Debug[1]	4.960	4.161	0.800	15.000	sys_clk	sys_clk

5.4 Implementation

- **Messages**



- **Utilization Report**

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	LUT Flip Flop Pairs (20800)	Bonded IOB (106)	BUFGCTRL (32)
N Processor_Top	1008	1221	260	128	573	1008	61	36	1
CU (Control_Unit)	8	2	0	0	7	8	0	0	0
EX_MEM (EX_M_Reg)	160	31	0	0	90	160	0	0	0
Execution_Unit (ALU)	6	0	0	0	6	6	0	0	0
Flags_Reg (CCR)	2	8	0	0	4	2	0	0	0
ID_EX (ID_EX_Reg)	158	63	4	0	74	158	10	0	0
IF_ID (IF_ID_Reg)	69	24	0	0	42	69	0	0	0
MEM_WB (M_WB_Reg)	19	21	0	0	17	19	1	0	0
PC (Program_Counter)	5	8	0	0	10	5	0	0	0
RF (Register_File)	37	32	0	0	12	37	8	0	0
Store_Data_Mux (Data_mem_mux)	0	8	0	0	2	0	0	0	0
UnifiedMem (Unified_Memory_DualPort)	544	1024	256	128	464	544	0	0	0

- **Timing Report**

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.160 ns	Worst Hold Slack (WHS):	0.131 ns	Worst Pulse Width Slack (WPWS):	7.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	2409	Total Number of Endpoints:	2409	Total Number of Endpoints:	1222

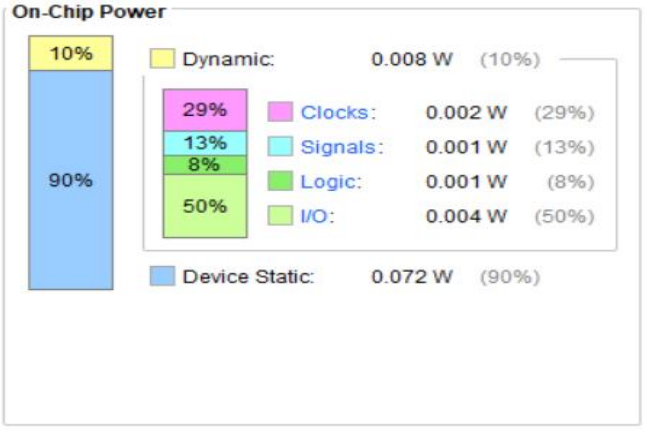
All user specified timing constraints are met.

- **Power Report**

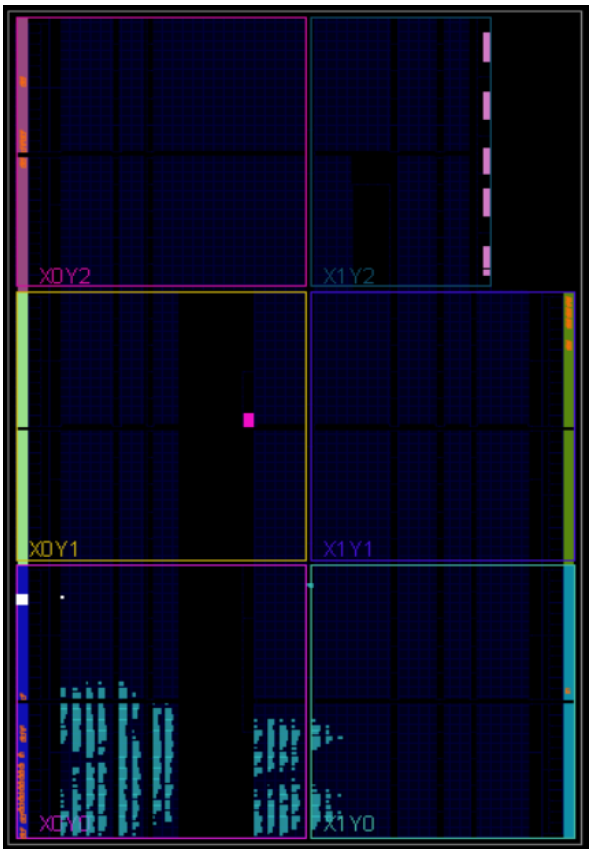
Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.08 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.4°C**
Thermal Margin:	59.6°C (11.9 W)
Effective ϑJA:	5.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

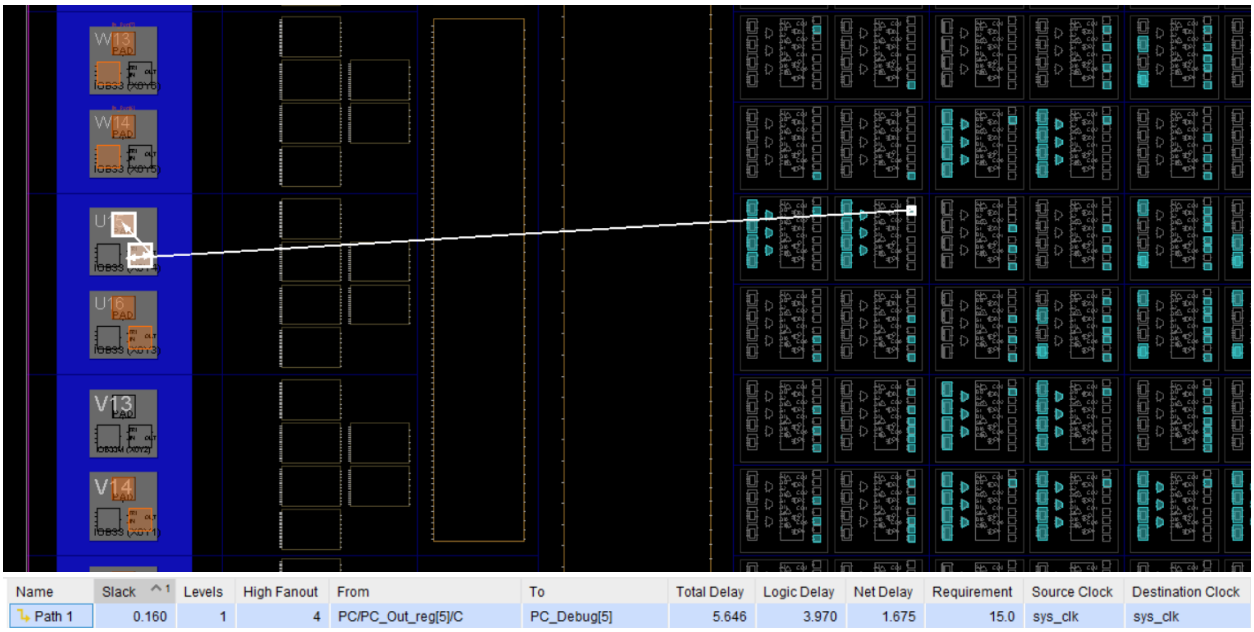
Launch Power Constraint Advisor to find and fix invalid switching activity



- **Device snippet**



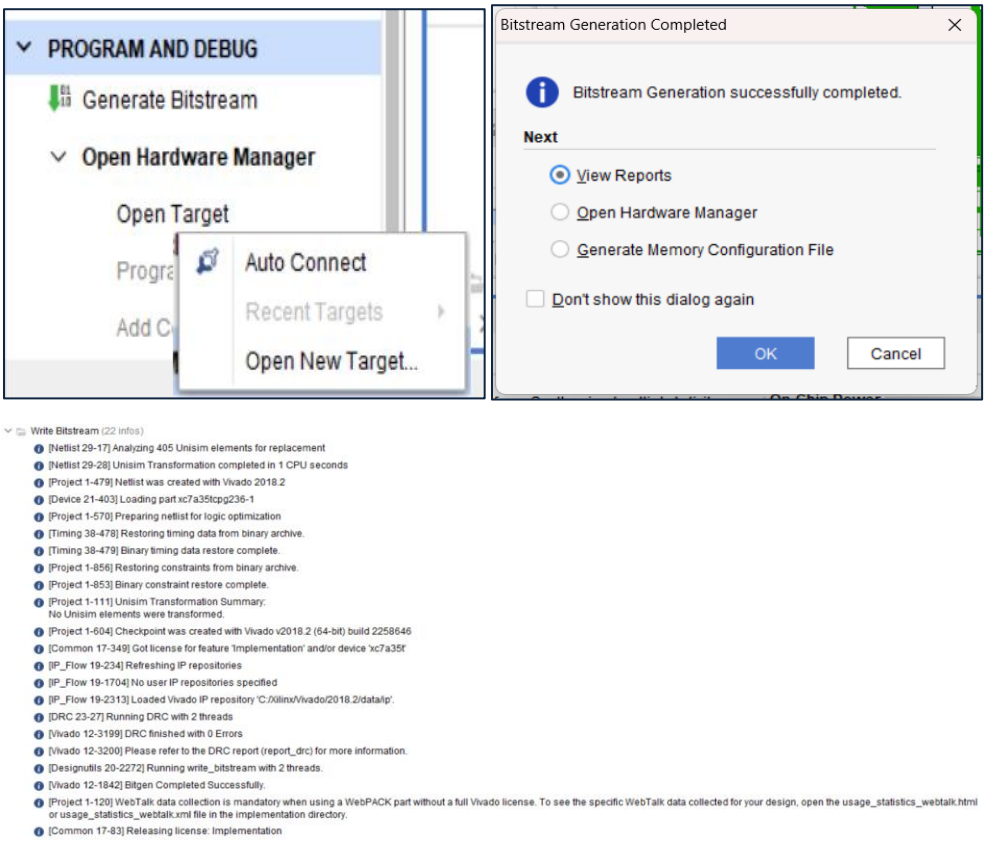
- **Critical Path highlighted in Device snippet**



Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock
Path 1	0.160	1	4	PC/PC_Out_reg[5]/C	PC_Debug[5]	5.646	3.970	1.675	15.0	sys_clk	sys_clk

5.5 Program & Debug

- **Generating Bitstream**



- **Bitstream.bit snippet**

Finally, now we can connect to the FPGA we choose and select our Bitstream.bit file and try our design on the FPGA.

6. Conclusion

This project successfully implements a fully functional 8-bit Von Neumann processor with a 5-stage pipeline. The design integrates advanced features such as hardware-based hazard detection and automated interrupt handling, which are typically found in more complex architectures. The modular block design—separating control logic, datapath, and memory management—allowed for targeted verification of each subsystem. The successful execution of all testbenches confirms that the processor meets the design objectives, efficiently handling data processing, stack manipulation, and flow control.

الحمدلله رب العالمين