12/27/2025

Design and Implement of a simple 8-bit pipelined processor

Mohamed Mustafa Shiple

Table of Contents

Table of Figures

1. Introduction

In this project, We designed and built an 8-bit pipelined microprocessor using Verilog. The architecture follows a von Neumann model, which means it uses a single shared memory for both code and data. To keep the system efficient despite this shared bottleneck, We implemented a centralized control unit to manage structural hazards. The design is based on RISC principles and uses a standard 5-stage pipeline (Fetch, Decode, Execute, Memory, and Write-Back) to process multiple instructions at once and improve performance. On the hardware side, the processor includes a 4-register file, a stack pointer for handling subroutines, and a Condition Code Register (CCR) for logic-based branching. Finally, We verified the entire design through RTL simulations and waveform analysis to make sure every instruction executes correctly and on time.

2. System Architecture

2.1. Control Unit

2.1.1. Architecture Overview

The control unit is implemented as a centralized pipelined control system that manages all five pipeline stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM), and Write Back (WB). The design generates stage-specific control signals by tracking which instruction occupies each pipeline stage through dedicated pipeline registers.

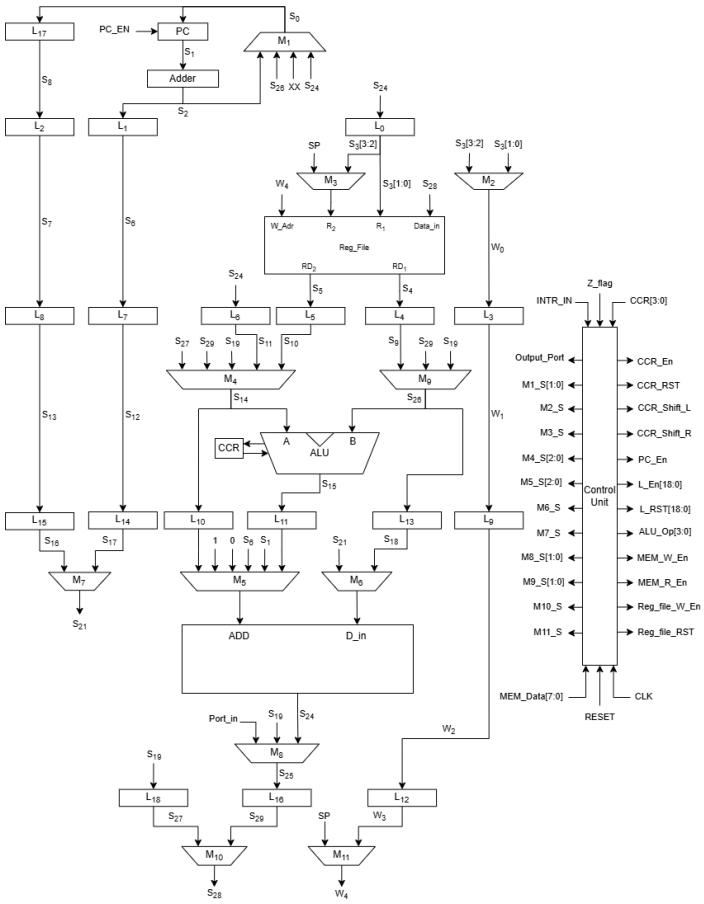

Figure 1: Architecture Design

Key Features:

- **Pipeline Stage Tracking:** Four 8-bit registers (`Prev_ID`, `Prev_EX`, `Prev_M`, `Prev_WB`) store the instruction word for each stage, enabling opcode extraction and control signal generation per stage.
- **Opcode Decoding:** Each pipeline stage extracts its 4-bit opcode and 2-bit operand fields (`ra`, `rb`) from the stored instruction word.
- **Hazard Detection:** Comprehensive logic detects Read-After-Write (RAW), Read-After-Load hazards across pipeline stages.
- **Data Forwarding:** Implements forwarding paths to resolve hazards without stalling when possible, using multiplexer select signals (`M4_S`, `M9_S`).
- **Interrupt Handling:** Detects rising edge of interrupt signal, saves PC to stack, and loads interrupt vector from memory address 1.

2.1.2. Control Flow and State Management

The control unit operates as a synchronous sequential circuit with combinational control logic. While not implemented as a traditional FSM with explicit states, it manages pipeline flow through conditional control signal generation based on current instruction opcodes and hazard conditions.

Pipeline Flow Control:

Normal Operation:

`IF → ID → EX → MEM → WB`

- Instructions advance through pipeline stages on each clock cycle
- `Next_ID`, `Next_EX`, `Next_M`, `Next_WB` signals determine next stage contents
- Pipeline registers update on rising clock edge when not stalled

Reset Behavior:

- PC loads from memory address 0
- All pipeline registers cleared to 0x00 (NOP)
- Condition Code Register (CCR) reset
- Register file reset to initial state

Interrupt Occurs Behavior:

- PC loads from memory address 1
- Disenable PC until instruction 0xFF (virtual instruction that handle saving PC in stack and handle saving Flags) be in Decode stage
- When 0xFF be in Decode stage enable PC and continue
- Current PC written to stack with instruction 0xFF

Stall Conditions:

- **Load-Use Hazard:** When an instruction depends on data from a load instruction in EX stage
- **Multi-Cycle Fetch:** Two-byte instructions (LDM, LDD, STD) require an extra cycle to fetch immediate/address
- **Multi-Cycle Writeback:** POP instruction requires two cycles to complete
- **Memory Busy:** When M stage is performing memory operations that conflict with fetch
- **Interrupt:** When interrupt edge is detected

Flush Conditions:

- **Branch Taken:** Conditional branches (`JZ`, `JN`, `JC`, `JV`) or LOOP when condition met (`C_branch`)
- **Unconditional Jumps:** JMP, CALL, RET, RTI instructions (`U_branch`)

2.1.3. Control Signal Generation

The control unit generates 19 enable signals (`L_En`), 19 reset signals (`L_RST`), and various multiplexer selects to control the Datapath.

Instruction Fetch (IF) Stage

- **PC_En:** Enables PC increment/load (disabled during stalls)
- **M1_S:** Selects PC source:
  - `2'b11`: PC + 1 (normal increment)
  - `2'b10`: ALU result (for branches in EX stage)
  - `2'b00`: Memory data (for RET/RTI in MEM stage)
- **MEM_R_En:** Enables memory read for instruction fetch
- **M5_S:** Selects memory address:
  - `3'b001`: PC (normal fetch)
  - `3'b011`: Address 0 (reset)
  - `3'b100`: Address 1 (interrupt vector)

Instruction Decode (ID) Stage

- **M2_S:** Selects register Ra read address (instruction `ra` field or stack pointer)
- **M3_S:** Selects register Rb read address (instruction `rb` field or zero)
- **L_En[3:8]:** Enable latches for ID pipeline registers
- Generates signals disabled during stalls or when NOP is in ID stage

Execute (EX) Stage

- **ALU_Op[3:0]:** Specifies ALU operation based on instruction opcode:
- **M4_S[2:0]:** Selects ALU input A (forwarding from M/WB stages):
  - `3'b000`: ID stage output (normal)
  - `3'b010`: MEM stage result (forward from M)
  - `3'b011`: WB stage result (forward from WB)
  - `3'b001`: Immediate value (for LDM)
  - `3'b100`: Incremented stack pointer
- **M9_S[1:0]:** Selects ALU input B (forwarding):
  - `2'b00`: ID stage output (normal)
  - `2'b10`: MEM stage result
  - `2'b01`: WB stage result
- **CCR_En:** Enables condition code register update
- **CCR_Shift_L/R:** Shifts CCR left/right for CALL/RTI flag preservation
- **Output_Port:** Enables output port for OUT instruction
- **L_En[9:15]:** Enable EX pipeline registers

Memory (MEM) Stage

- **MEM_R_En:** Enables memory read for load instructions
- **MEM_W_En:** Enables memory write for store/push instructions
- **M5_S:** Selects memory address source:
  - `3'b000`: ALU result (for LDI/STI)
  - `3'b101`: Immediate address from instruction (for LDD/STD)
- **M6_S:** Selects memory write data (register value or PC for PUSH/CALL)
- **M7_S:** Selects between regular data and flags for interrupt save
- **M8_S[1:0]:** Selects next PC increment value (1 or 2 for multi-byte instructions)
- **L_En[12,16,18]:** Enable MEM pipeline registers

Write Back (WB) Stage

- **Reg_file_W_En:** Enables register file write
- **M10_S:** Selects WB data source (memory or ALU result)
- **M11_S:** Selects destination register address (instruction field or SP for stack operations)

2.1.4. Hazard Detection and Forwarding

Data Hazards (RAW - Read After Load)

Read After Load: The control unit detects when an instruction in EX stage needs data that a load instruction in M stage will produce. This creates a one-cycle stall until memory data is available. When WB stage contains a load result that EX stage needs, forwarding occurs through `M4_S = 3'b011` or `M9_S = 2'b01`

RAW: data is forwarded from MEM or WB stages without stalling:

- **EX-MEM Forwarding:** `RAW_EX_M_ra`, `RAW_EX_M_rb` detect when ID needs EX stage results
- **EX-WB Forwarding:** `RAW_EX_WB_ra`, `RAW_EX_WB_rb` detect when ID needs MEM stage results

Control Hazards

Branch Resolution: Branches are resolved in EX stage. When taken (`C_branch` or `U_branch` high):

- IF and ID stages flushed by inserting NOPs
- PC updated with branch target from register
- Two-cycle penalty for taken branches

Branch Types:

- **Conditional:** JZ, JN, JC, JV (check CCR flags)

- **Unconditional:** JMP, CALL
- **Loop:** LOOP (decrements counter, branches if non-zero)

Structural Hazards

Memory Conflicts: Single memory cannot serve instruction fetch and data access simultaneously. Control unit prioritizes data memory operations:

When `mem_busy = 1`, instruction fetch stalls by disabling `PC_En` and holding IF stage.

Two-Byte Instruction Handling: LDM, LDD, STD require two memory accesses (opcode + immediate/address)

2.2. Register File

2.2.1. Architecture Overview

The processor register file is implemented as a dedicated hardware module responsible for storing and providing fast access to the general-purpose registers. The design follows the ISA specification, which defines **four 8-bit general purpose registers (R0–R3)**, where **R3 also functions as the Stack Pointer (SP)**. The initial value of the stack pointer is set to 255, pointing to the top of the stack.
The register file supports **simultaneous reading of two registers and writing to one register**, which is essential for supporting arithmetic and logical instructions in a pipelined processor.

2.2.2. Register File Interface

The register file module includes the following key signals:

- **Read address ports** for selecting source registers (ra and rb)
- **Write address port** for selecting the destination register
- **Write enable signal** to control register updates
- **Clock signal** for synchronous write operation
- **Reset signal** to initialize register values
- **Two read data outputs** to supply operands to the execution stage

This interface allows the register file to be accessed efficiently during the instruction decode stage of the pipeline.

2.2.3. Read and Write Operation

Register **write operations are synchronous**, occurring on the rising edge of the clock when the write enable signal is asserted. This ensures proper synchronization with the pipeline and avoids race conditions.

Register **read operations are asynchronous**, meaning that the contents of the selected registers are continuously available at the output without waiting for a clock edge. This design choice reduces latency and supports single-cycle operand fetch during instruction decoding.

On reset, all registers are initialized to zero except the stack pointer register (R3), which is initialized to 255 in accordance with the ISA requirements

2.2.4. Pipeline Integration

The register file is accessed during the **Instruction Decode (ID)** stage. The source operands are read using the register indices encoded in the instruction, while the destination register is updated during the **Write Back (WB)** stage. This separation of read and write timing supports correct pipeline operation and avoids read-after-write hazards.

2.3. Memory

2.3.1. Architecture Overview

The processor memory is implemented as a unified 256-byte byte-addressable memory, consistent with the **Von Neumann architecture** used in the design. This single memory module stores both instructions and data, emphasizing resource sharing as required by the project specifications.
Each memory location stores an 8-bit value, and the full address space is accessible using an 8-bit address.

2.3.2. Memory Interface

The memory module provides the following functionality:

- **Address input** to specify the memory location
- **Write data input** for store operations
- **Read data output** for load and instruction fetch operations
- **Write enable signal** to control memory updates
- **Clock signal** for synchronous writes

This interface supports all load, store, instruction fetch, and stack operations defined in the ISA.

2.3.3. Read and Write Operation

Memory **write operations are synchronous**, performed on the rising edge of the clock when the write enable signal is asserted. This applies to store instructions (STD, STI), stack operations (PUSH), and interrupt handling.

Memory **read operations are asynchronous**, allowing instruction fetch and data access to occur without additional clock latency. This is particularly important for maintaining pipeline efficiency during instruction fetch and load operations.

2.3.4. Stack and Interrupt Support

The memory module plays a critical role in stack and interrupt handling. During **CALL**, **PUSH**, and **interrupt entry**, data such as return addresses and flags are stored in memory at locations pointed to by the stack pointer. During **RET**, **POP**, and **RTI** instructions, values are retrieved from the stack accordingly.
The use of a unified memory for both instructions and stack data highlights the resource-sharing aspect of the design.

2.3.5. Pipeline Integration

The memory module is accessed during both the **Instruction Fetch (IF)** and **Memory Access (MEM)** stages. Instruction bytes are fetched using the program counter, while data accesses occur during load and store instructions. Control logic ensures correct arbitration between instruction and data access to maintain correct execution.

2.3.6. FSM and Memory Interaction

The processor utilizes a centralized **Finite State Machine (FSM) control unit** to coordinate all memory accesses due to the use of a **single shared memory resource**. The FSM ensures correct sequencing of instruction fetch, data access, and stack operations while preventing resource conflicts between pipeline stages.

During the **Instruction Fetch (IF)** state, the FSM asserts the memory read control signal and selects the program counter (PC) as the memory address source. The fetched instruction byte is then latched into the pipeline register, and the PC is either incremented or updated based on control flow instructions.

For **L-format instructions**, which may require a second byte (effective address or immediate value), the FSM introduces an additional memory access state to fetch the second byte. This guarantees correct decoding and execution while preserving pipeline correctness.

In the **Memory Access (MEM)** state, the FSM controls data memory operations for load and store instructions. For load instructions (LDD, LDI), the FSM enables memory read and forwards the retrieved data to the write-back stage. For store instructions (STD, STI), the FSM asserts the memory write enable signal and supplies the appropriate data and address.

Stack operations such as **PUSH**, **POP**, **CALL**, **RET**, and **RTI** are also managed by the FSM through controlled memory access cycles. The FSM coordinates stack pointer updates and memory read/write operations to ensure correct saving and restoring of return addresses and processor state.

During an **interrupt event**, the FSM immediately suspends normal instruction execution, saves the current program counter and condition flags onto the stack through controlled memory writes, and redirects the PC to the interrupt service routine entry point. Upon execution of the **RTI** instruction, the FSM restores the saved context by reading from memory and resuming normal program flow.

By explicitly scheduling memory operations across FSM states, the control unit guarantees correct functionality of the unified memory system while maintaining pipeline stability and proper execution ordering.

2.4. ALU

2.4.1. Architecture Overview

The **Arithmetic Logic Unit (ALU)** is responsible for performing all arithmetic, logical, shift, and flag control operations in the processor. It operates on two 8-bit operands (**A and B**) and produces an 8-bit result along with updated condition code flags.

The ALU receives: Two 8-bit operands (A, B), A 4-bit operation selector (ALU_Op), The current condition flags (Flags_in).
It outputs: An 8-bit result (Result), Updated condition flags (Flags_out).

2.4.2. ALU Flow and State Management

The ALU itself is **stateless** and does not store any internal state. All operations are evaluated **combinationally** whenever any input changes.

The execution flow inside the ALU is as follows:

I. Default values are assigned to the result and flags to prevent latches.
II. The operation is selected using a case statement based on ALU_Op.
III. Arithmetic, logical, shift, or flag control operations are executed.

IV. Flags (Z, N, C, V) are updated according to the operation type.
V. Flags that are not affected by certain operations are preserved from Flags_in.
VI. The updated flags are packed and sent to the output.

Because the ALU has no internal memory and no clock dependency, pipeline state management and sequencing are handled externally by the **FSM control unit**, not within the ALU.

2.4.3. Signal Generation for ALU

ALU Operation Control (ALU_OP)

The ALU_Op signal is generated by the control unit based on the decoded opcode and instruction format. Each value of ALU_Op selects a specific operation inside the ALU:

ALU_OP	Operation
0000	NOP
0001	MOV
0010	ADD
0011	SUB
0100	AND
0101	OR
0110	NOT
0111	NEG
1000	INC A
1001	DEC A
1010	INC B
1011	DEC B
1100	RLC
1101	RRC
1110	SETC
1111	CLRC

Flag Generation Logic

The ALU generates four condition flags:

- Zero (Z): Set when the result equals zero.
- Negative (N): Set when the MSB of the result is 1.
- Carry (C): Generated from the 9th bit of arithmetic and shift operations.
- Overflow (V): Indicates signed arithmetic overflow.

Flag behavior:

- Arithmetic and shift operations update flags.
- Logical and move operations update Z and N only.
- Some operations preserve existing flags from Flags_in.

2.5. Program Counter (PC)

2.5.1. Architecture Overview

The Program Counter (PC) is implemented as a synchronous register that holds the address of the next instruction to be fetched from memory. It operates as part of the unified memory system and supplies the instruction address during the Instruction Fetch (IF) stage. The PC value is updated only on the rising edge of the clock and is controlled through an explicit enable signal, allowing the control unit to precisely manage instruction flow.

2.5.2. PC Operation and Control

When the PC enable signal is asserted, the PC loads the next address value provided at its input. This value may correspond to the next sequential instruction, a branch or jump target, an interrupt vector, or a return address loaded from memory. When the enable signal is deasserted, the PC holds its current value, effectively stalling instruction fetch. This behavior is used during pipeline stalls caused by data hazards, memory conflicts, multi-cycle instructions, and interrupt handling.

2.6. Condition Code Register (CCR)

The Condition Code Register (CCR) is a specialized register that holds the processor's status flags, which are generated by arithmetic and logical operations and reflect the current execution state of the processor. These flags play a critical role in guiding conditional branch instructions and other control flow mechanisms. Architecturally, the CCR is implemented as a synchronous register that updates on the rising edge of the clock when the CCR enable signal is asserted. Under normal operation, it captures the status outputs from the ALU, reflecting the results of arithmetic or logical operations. When the enable signal is deasserted, the CCR maintains its previous state, ensuring that processor flags are preserved during pipeline stalls, memory access conflicts, or control flow changes. Beyond standard updates, the CCR also supports controlled shift operations, which are utilized in instructions such as CALL, RTI, and interrupt handling, enabling the processor to save and restore flags during transitions between normal and privileged execution modes.

3. Implementation & Synthesize

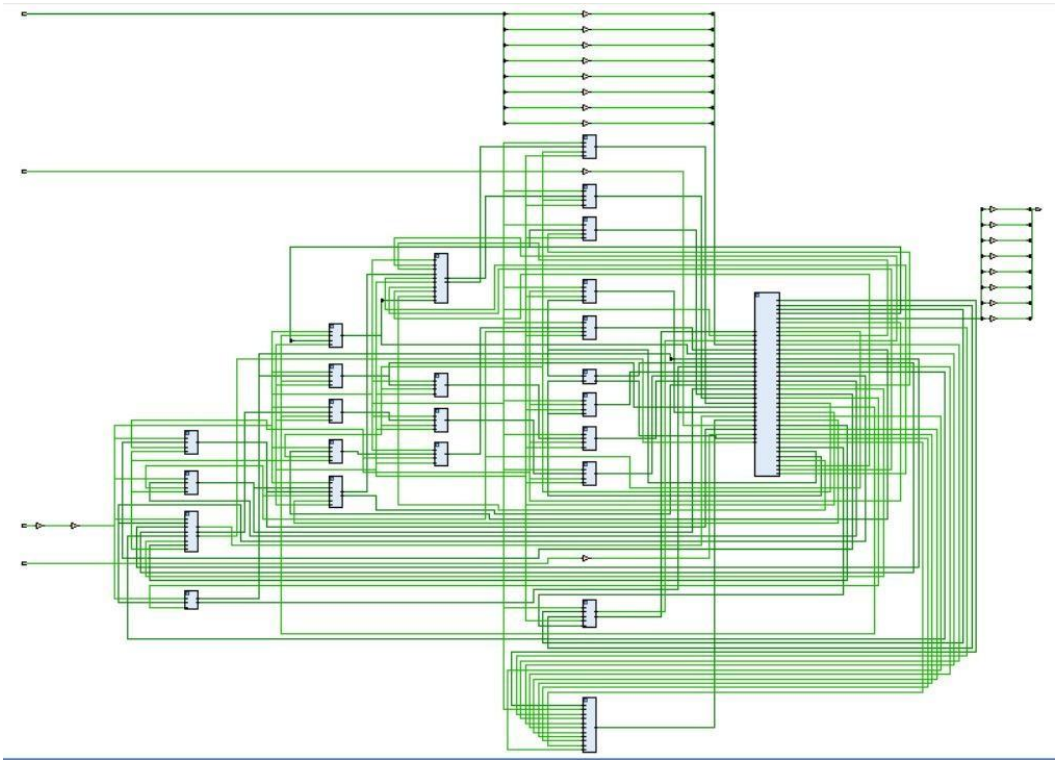

Figure 2: Schematic after Synthesize

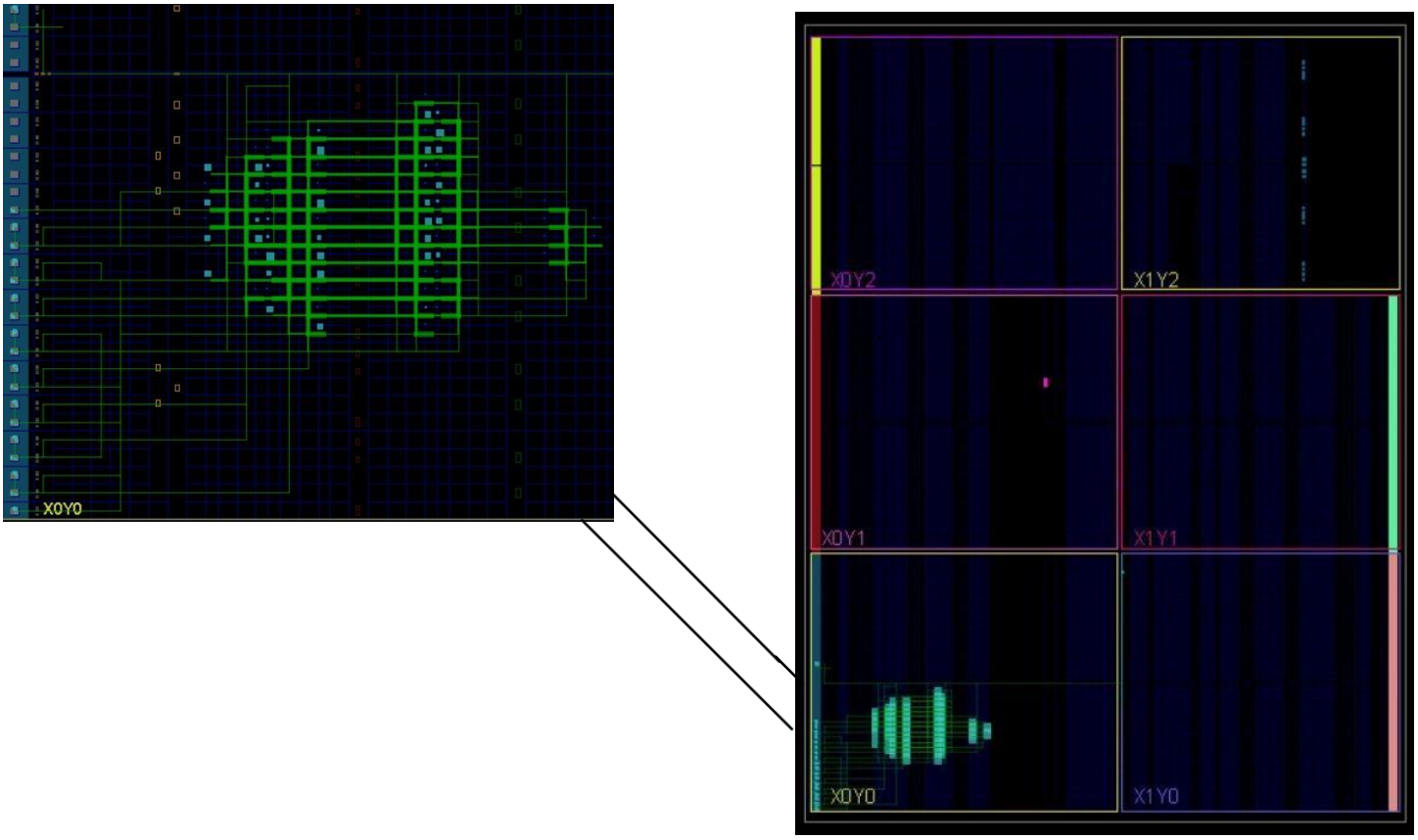

Figure 3:Implementation on FPGA (Artix-7-xc7a15tcpg236-3)

3.1. Timing Summary

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	1.324 ns	Worst Hold Slack (WHS):	0.131 ns	Worst Pulse Width Slack (WPWS):	3.950 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:	916	Total Number of Endpoints:	916	Total Number of Endpoints:	251

All user specified timing constraints are met.

Figure 4: Timing Summary

Target Period: 10.000 ns

Worst Negative Slack: 1.324 ns

Minimum achievable period: 10.000 - 1.324 = 8.676 ns

Maximum achievable frequency: 1/8.676 ns = 115.3 MHz

Timing margin: 1.324 ns (13.24% margin above target)

3.2. Utilization

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	400	10400	3.85
LUTRAM	32	9600	0.33
FF	218	20800	1.05
IO	19	106	17.92
BUFG	1	32	3.13

Figure 5: Utilization

4. Test Benches & Results

4.1. First Code

```
1. LDM R0, 0x05
2. LDM R1, 0x03
3. LDM R2, 0xFF
4. NOP
5. MOV R0, R1
6. ADD R0, R1
7. SUB R0, R1
8. AND R0, R2
9. OR R0, R2
10. SETC
11. RLC R1
12. RRC R1
13. CLRC
14. LDM R1, 0xAA
15. PUSH R1
16. POP R2
17. OUT R0
18. IN R1
19. LDM R1, 0x55
20. NOT R1
21. LDM R1, 0x05
22. NEG R1
23. INC R1
24. DEC R1
25. LDM R0, 0x05
26. LDM R1, 0xFB
27. ADD R0, R1
28. LDM R0, 0x0A
29. LDM R1, 0x0A
30. SUB R0, R1
31. LDM R0, 0x0F
32. LDM R1, 0xF0
33. AND R0, R1
34. LDM R0, 0x7F
35. LDM R1, 0x05
36. ADD R0, R1
37. LDM R0, 0x05
38. LDM R1, 0x0A
39. SUB R0, R1
40. LDM R1, 0x0F
41. NOT R1
42. LDM R0, 0xFF
43. LDM R1, 0x02
44. ADD R0, R1
45. LDM R0, 0x00
46. LDM R1, 0x01
47. SUB R0, R1
48. SETC
49. LDM R1, 0x80
50. RLC R1
51. LDM R1, 0x01
52. RRC R1
53. CLRC
54. SETC
55. LDM R0, 0x7F
56. LDM R1, 0x01
57. ADD R0, R1
58. LDM R0, 0x80
59. LDM R1, 0xFF
60. ADD R0, R1
61. LDM R0, 0x7F
62. LDM R1, 0xFF
63. SUB R0, R1
64. LDM R1, 0x7F
65. INC R1
66. LDM R1, 0x80
67. DEC R1
68. LDM R1, 0x80
69. NEG R1
70. LDM R0, 0xFF
71. LDM R1, 0x01
72. ADD R0, R1
73. MOV R0, R1
74. NOP
75. NOP
```

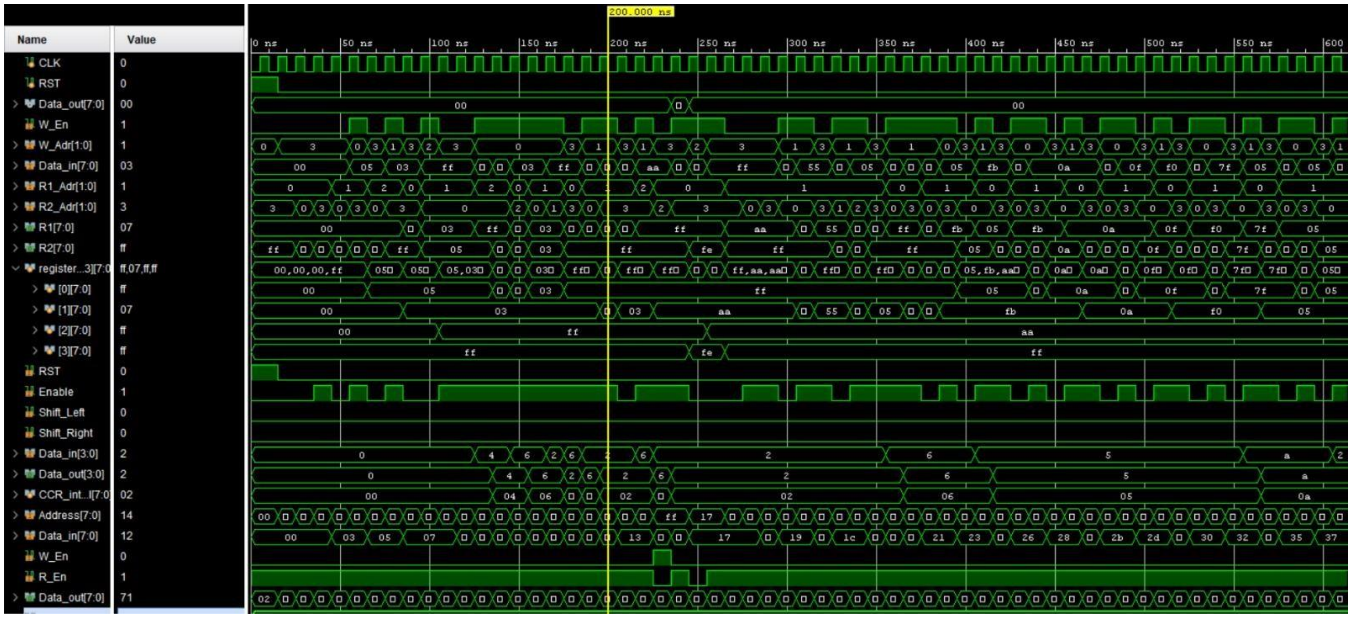

Figure 6: Results of The First Simulation

$$Clock\ Cycles = 116$$

$$Instructions = 75$$

$$CPI = \frac{116}{75} = 1.5467$$

4.2. Second Code

```
1. LDM R0, 10
2. LDM R1, 5
3. ADD R0, R1
4. LDM R2, 3
5. NOP
6. SUB R0, R2
7. STD R0, 2
8. OUT R0
9. AND R1, R2
10. OR R1, R2
11. INC R1
12. LDM R3, 31
13. JMP R3
14. NOP
```

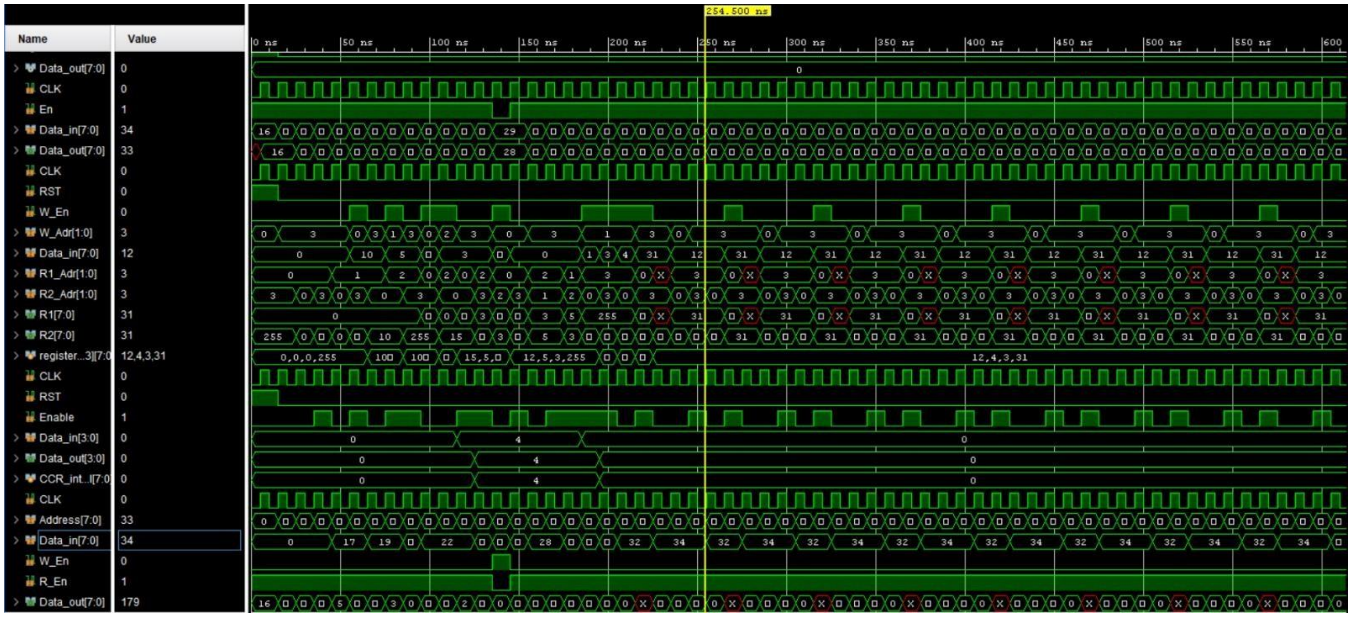

Figure 7: Results of Second Simulation

$$Clock\ Cycles = 24$$

$$Instructions = 14$$

$$CPI = \frac{24}{14} = 1.714$$