9/2/2025

# Single cycle RISC-V

Abd El Rahman Khaled Fouad Abd El Rahim

## Table of contents

## Abstract

As shown In Figure 3.1, This project represents the design of RISC-V 32I microprocessor from scratch using HDL (Verilog) and tested using Questa-Sim by using exhausted testing of all possible different scenarios to maintain validation of all cases of the processor and implemented on Artix-7 (XC7A35TICPG236-1L) FPGA on Vivado by using a 12ns clock to solve any timing violations

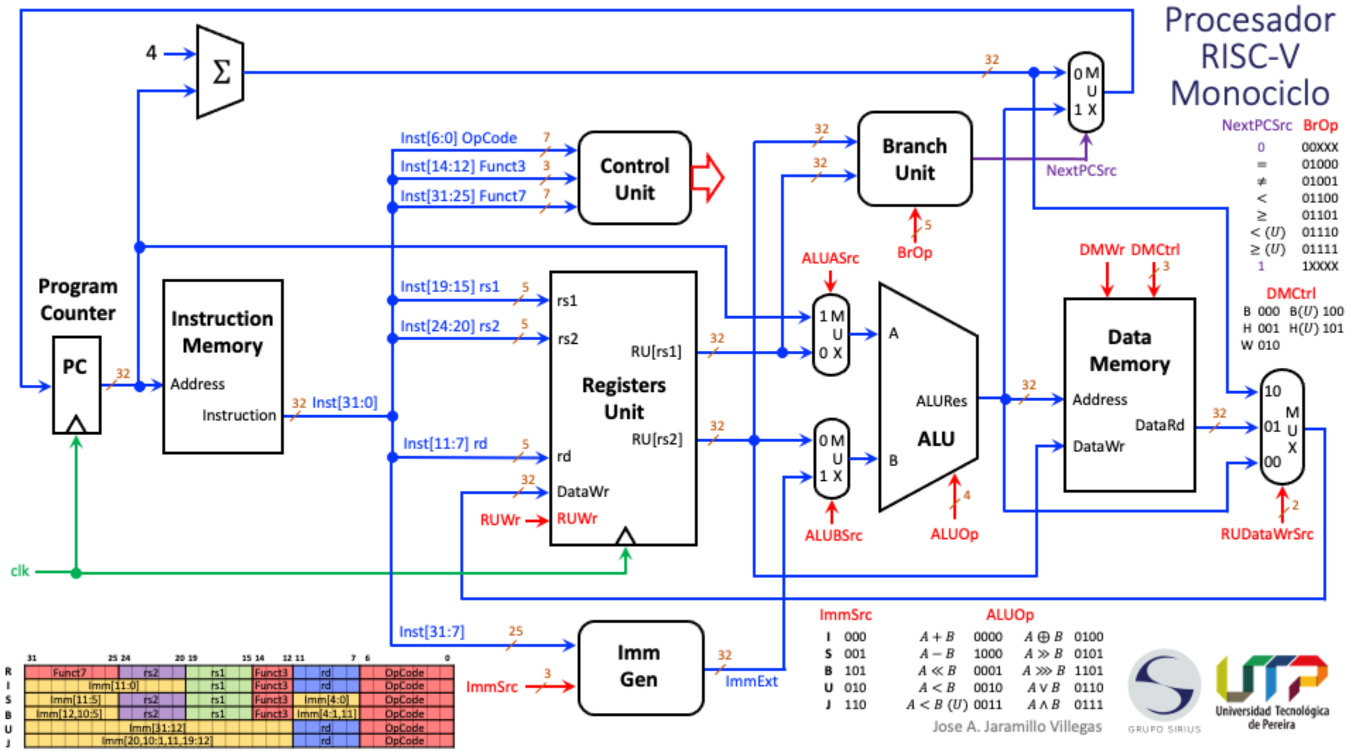


**Figure 3.1** Single cycle RISC-V processor

## Introduction

RISC-V (Reduced Instruction Set Computing - Version Five) is an open-source and highly flexible instruction set architecture (ISA) that has gained significant traction in recent years. Unlike proprietary architectures such as ARM or x86, RISC-V is free to use and allows for customization, making it a preferred choice for a wide range of applications.

The RISC-V ISA follows the RISC (Reduced Instruction Set Computing) principles, which emphasize simplicity, efficiency, and modularity. It provides a base integer instruction set (RV32I, RV64I, RV128I) and various optional extensions (such as floating-point, vector processing, and atomic operations), making it suitable for different computing needs.

That processor has different Applications in the industry: -

- Embedded Systems:
    1) Used in IoT devices, microcontrollers, and sensors.
    2) Low power consumption makes it ideal for battery-powered applications.

- Artificial Intelligence & Machine Learning:
    1) Custom RISC-V cores optimize **AI and deep learning** workloads.
    2) Used in AI accelerators and edge computing devices.

- High-Performance Computing (HPC):
    1) RISC-V is being explored for **supercomputers** and **data centers**.
    2) Efficient vector processing extensions enhance performance in **scientific computing**.

- Consumer Electronics:
  1) Found in **smartphones, wearables, and smart TVs**.
  2) Companies like **Google, Western Digital, and NVIDIA** have shown interest in RISC-V-based designs.

- Education & Research
  1) Open-source nature makes RISC-V popular in **universities and research labs**.
  2) Used for teaching computer architecture and experimenting with custom instruction sets.

## RISC-V architecture

Any microprocessor hardware contains the following: -

- **Fetch:** get the instruction from instruction memory
- **Decode:** understand what this instruction does
- **Execute:** get the data needed to perform the operation and do the operation indicated in the instruction
- **Mem op:** some instructions require data memory access such as (load & store)
- **Write back:** write the results in the registers (internal to processor)
- memory operation

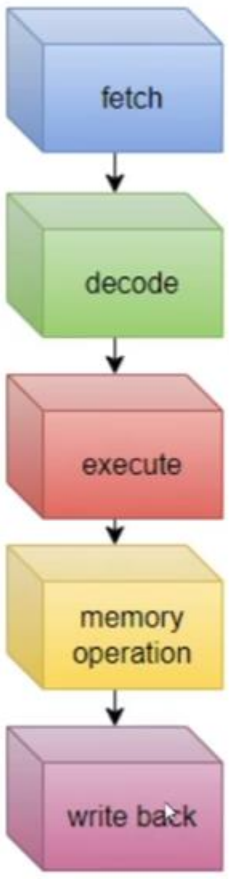


Figure 6.1

In Figure 7.1 represents the instruction memory, register file, and data memory are all read combinationally. In other words, if the address changes, then the new data appears at RD after some propagation delay; no clock is involved. The clock controls writing only. These memories are written only on the rising edge of the clock. In this fashion, the state of the system is changed only at the clock edge. The address, data, and write enable must set up before the clock edge and must remain stable until a hold time after the clock edge. Because the state elements change their state only on the rising edge of the clock, they are synchronous sequential circuits. A microprocessor is built of clocked state elements and combinational logic, so it too is a synchronous sequential circuit. Indeed, a processor can be viewed as a giant finite state machine or as a collection of simpler interacting state machines.

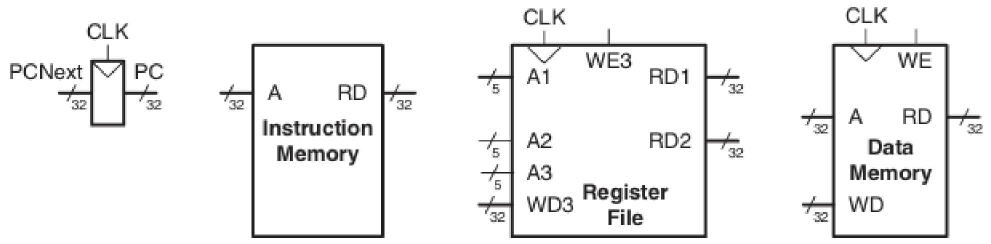


**Figure 7.1 State elements of a RISC-V processor**

In Figure 7.2 the extend unit at which the processor adds the base address to the offset to find the address to read from memory.

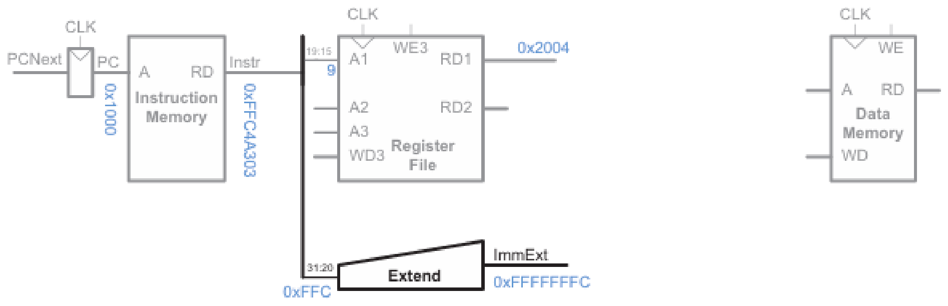


**Figure 7.2 Extend unit**

**In Figure 8.1 shows the complete single cycle processor after putting the control unit that controls all the signals in the processor**

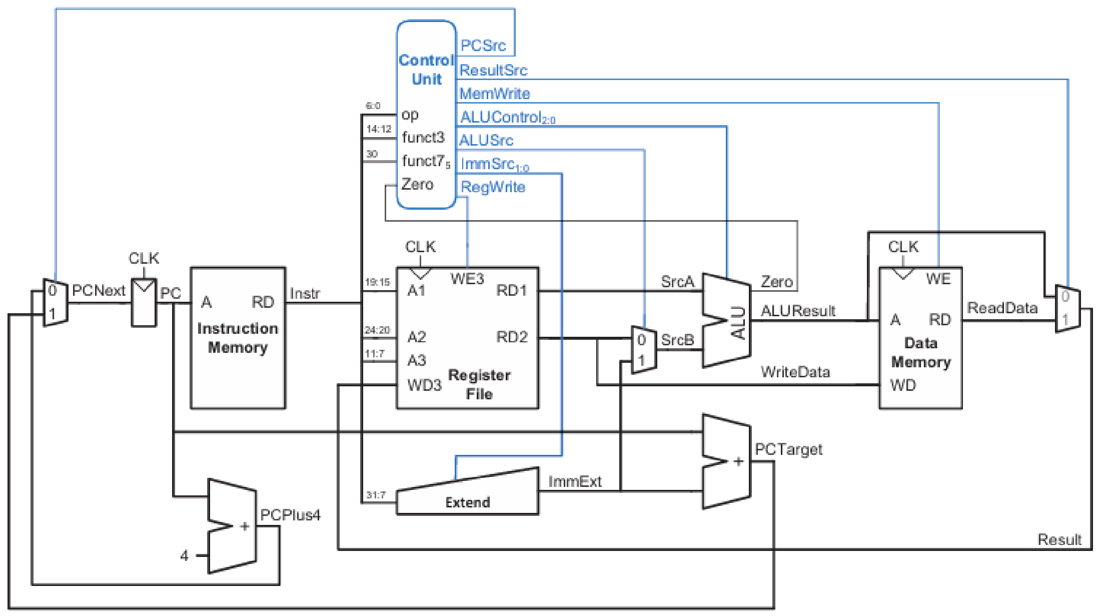


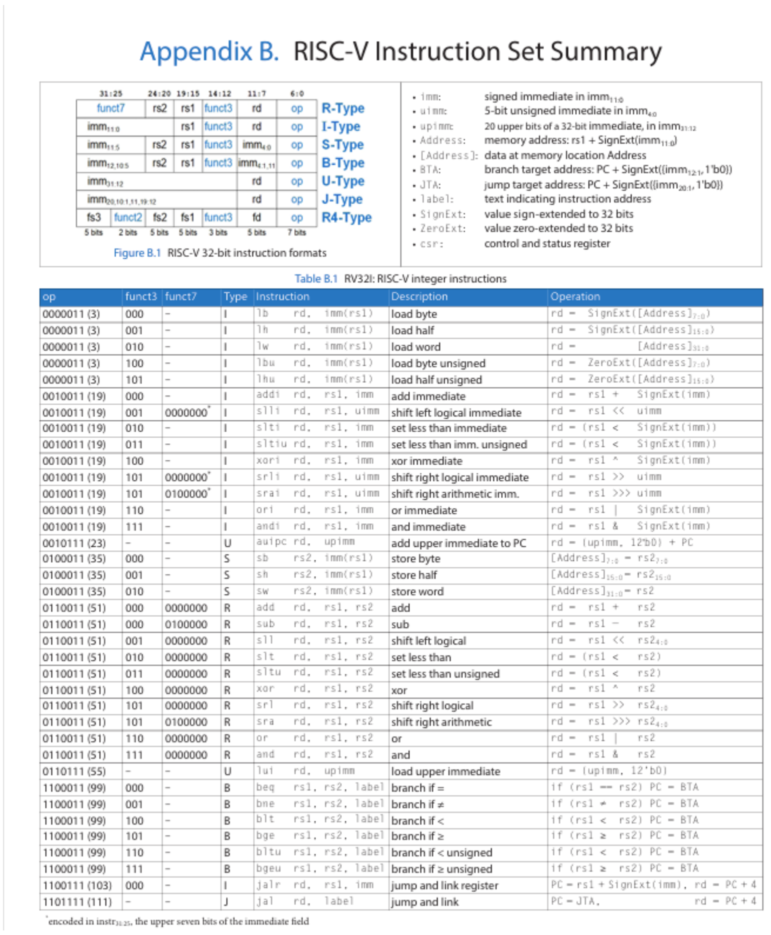
Figure 8.1 Complete single cycle processor

## RISC-V instructions

- **Base Integer Instruction Set (I)**
  - Load and Store Instructions (LW, SW…)
  - Arithmetic Instructions (ADD, ADDI…)
  - Logical Instructions (AND, OR XOR…)
  -Comparison Instructions (SLT, SLTI…)
  - Branch Instructions (Conditional Jumps) (BEQ, BGE…)
  - Jump Instructions (Unconditional Jumps) (JAL, JALR…)

- **Multiplication and Division Extension (M)**
- **Atomic Extension (A)**
- **Floating-Point Extensions (F and D)**
- **Compressed Instructions (C)**
- **Vector Extension (V)**
- **Bit Manipulation Extension (B)**

But in my design, I will be focusing on the Base Integer Instruction Set (I) as shown in Figure 10.1



## Appendix B. RISC-V Instruction Set Summary

| 31:25 | | 24:20 | 19:15 | 14:12 | 11:7 | 6:0 | |
|---|---|---|---|---|---|---|---|
| funct7 | | rs2 | rs1 | funct3 | rd | op | R-Type |
| $imm_{11:0}$ | | | rs1 | funct3 | rd | op | I-Type |
| $imm_{11:5}$ | | rs2 | rs1 | funct3 | $imm_{4:0}$ | op | S-Type |
| $imm_{12,10:5}$ | | rs2 | rs1 | funct3 | $imm_{4:1,11}$ | op | B-Type |
| $imm_{31:12}$ | | | | | rd | op | U-Type |
| $imm_{20,10:1,11,19:12}$ | | | | | rd | op | J-Type |
| fs3 | funct2 | fs2 | fs1 | funct3 | fd | op | R4-Type |
| 5 bits | 2 bits | 5 bits | 5 bits | 3 bits | 5 bits | 7 bits | |

Figure B.1 RISC-V 32-bit instruction formats

| | |
|---|---|
| • `imm`: | signed immediate in $imm_{11:0}$ |
| • `uimm`: | 5-bit unsigned immediate in $imm_{4:0}$ |
| • `upimm`: | 20 upper bits of a 32-bit immediate, in $imm_{31:12}$ |
| • `Address`: | memory address: rs1 + SignExt($imm_{11:0}$) |
| • `[Address]`: | data at memory location Address |
| • `BTA`: | branch target address: PC + SignExt({$imm_{12:1}$, 1'b0}) |
| • `JTA`: | jump target address: PC + SignExt({$imm_{20:1}$, 1'b0}) |
| • `label`: | text indicating instruction address |
| • `SignExt`: | value sign-extended to 32 bits |
| • `ZeroExt`: | value zero-extended to 32 bits |
| • `csr`: | control and status register |

Table B.1 RV32I: RISC-V integer instructions

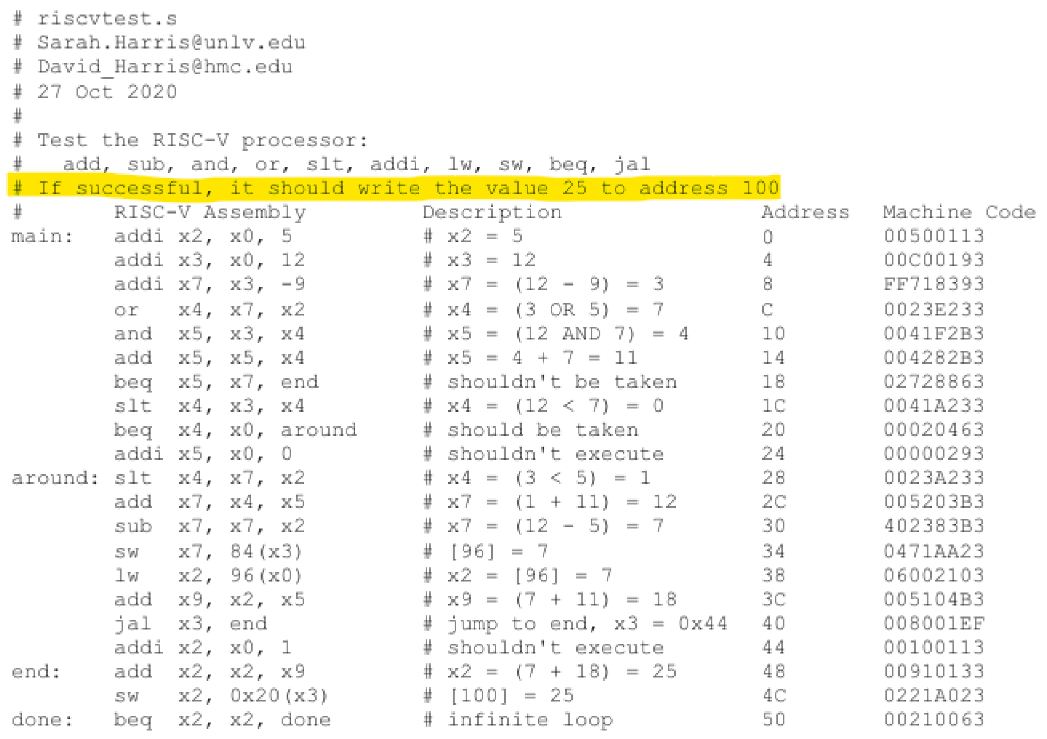
| op | funct3 | funct7 | Type | Instruction | Description | Operation |
|---|---|---|---|---|---|---|
| 0000011 (3) | 000 | – | I | `lb rd, imm(rs1)` | load byte | rd = SignExt([Address]$_{7:0}$) |
| 0000011 (3) | 001 | – | I | `lh rd, imm(rs1)` | load half | rd = SignExt([Address]$_{15:0}$) |
| 0000011 (3) | 010 | – | I | `lw rd, imm(rs1)` | load word | rd = [Address]$_{31:0}$ |
| 0000011 (3) | 100 | – | I | `lbu rd, imm(rs1)` | load byte unsigned | rd = ZeroExt([Address]$_{7:0}$) |
| 0000011 (3) | 101 | – | I | `lhu rd, imm(rs1)` | load half unsigned | rd = ZeroExt([Address]$_{15:0}$) |
| 0010011 (19) | 000 | – | I | `addi rd, rs1, imm` | add immediate | rd = rs1 + SignExt(imm) |
| 0010011 (19) | 001 | 0000000* | I | `slli rd, rs1, uimm` | shift left logical immediate | rd = rs1 << uimm |
| 0010011 (19) | 010 | – | I | `slti rd, rs1, imm` | set less than immediate | rd = (rs1 < SignExt(imm)) |
| 0010011 (19) | 011 | – | I | `sltiu rd, rs1, imm` | set less than imm. unsigned | rd = (rs1 < SignExt(imm)) |
| 0010011 (19) | 100 | – | I | `xori rd, rs1, imm` | xor immediate | rd = rs1 ^ SignExt(imm) |
| 0010011 (19) | 101 | 0000000* | I | `srli rd, rs1, uimm` | shift right logical immediate | rd = rs1 >> uimm |
| 0010011 (19) | 101 | 0100000* | I | `srai rd, rs1, uimm` | shift right arithmetic imm. | rd = rs1 >>> uimm |
| 0010011 (19) | 110 | – | I | `ori rd, rs1, imm` | or immediate | rd = rs1 \| SignExt(imm) |
| 0010011 (19) | 111 | – | I | `andi rd, rs1, imm` | and immediate | rd = rs1 & SignExt(imm) |
| 0010111 (23) | – | – | U | `auipc rd, upimm` | add upper immediate to PC | rd = {upimm, 12'b0} + PC |
| 0100011 (35) | 000 | – | S | `sb rs2, imm(rs1)` | store byte | [Address]$_{7:0}$ = rs2$_{7:0}$ |
| 0100011 (35) | 001 | – | S | `sh rs2, imm(rs1)` | store half | [Address]$_{15:0}$ = rs2$_{15:0}$ |
| 0100011 (35) | 010 | – | S | `sw rs2, imm(rs1)` | store word | [Address]$_{31:0}$ = rs2 |
| 0110011 (51) | 000 | 0000000 | R | `add rd, rs1, rs2` | add | rd = rs1 + rs2 |
| 0110011 (51) | 000 | 0100000 | R | `sub rd, rs1, rs2` | sub | rd = rs1 − rs2 |
| 0110011 (51) | 001 | 0000000 | R | `sll rd, rs1, rs2` | shift left logical | rd = rs1 << rs2$_{4:0}$ |
| 0110011 (51) | 010 | 0000000 | R | `slt rd, rs1, rs2` | set less than | rd = (rs1 < rs2) |
| 0110011 (51) | 011 | 0000000 | R | `sltu rd, rs1, rs2` | set less than unsigned | rd = (rs1 < rs2) |
| 0110011 (51) | 100 | 0000000 | R | `xor rd, rs1, rs2` | xor | rd = rs1 ^ rs2 |
| 0110011 (51) | 101 | 0000000 | R | `srl rd, rs1, rs2` | shift right logical | rd = rs1 >> rs2$_{4:0}$ |
| 0110011 (51) | 101 | 0100000 | R | `sra rd, rs1, rs2` | shift right arithmetic | rd = rs1 >>> rs2$_{4:0}$ |
| 0110011 (51) | 110 | 0000000 | R | `or rd, rs1, rs2` | or | rd = rs1 \| rs2 |
| 0110011 (51) | 111 | 0000000 | R | `and rd, rs1, rs2` | and | rd = rs1 & rs2 |
| 0110111 (55) | – | – | U | `lui rd, upimm` | load upper immediate | rd = {upimm, 12'b0} |
| 1100011 (99) | 000 | – | B | `beq rs1, rs2, label` | branch if = | if (rs1 == rs2) PC = BTA |
| 1100011 (99) | 001 | – | B | `bne rs1, rs2, label` | branch if ≠ | if (rs1 ≠ rs2) PC = BTA |
| 1100011 (99) | 100 | – | B | `blt rs1, rs2, label` | branch if < | if (rs1 < rs2) PC = BTA |
| 1100011 (99) | 101 | – | B | `bge rs1, rs2, label` | branch if ≥ | if (rs1 ≥ rs2) PC = BTA |
| 1100011 (99) | 110 | – | B | `bltu rs1, rs2, label` | branch if < unsigned | if (rs1 < rs2) PC = BTA |
| 1100011 (99) | 111 | – | B | `bgeu rs1, rs2, label` | branch if ≥ unsigned | if (rs1 ≥ rs2) PC = BTA |
| 1100111 (103) | 000 | – | I | `jalr rd, rs1, imm` | jump and link register | PC = rs1 + SignExt(imm), rd = PC + 4 |
| 1101111 (111) | – | – | J | `jal rd, label` | jump and link | PC = JTA, rd = PC + 4 |

*encoded in instr$_{31:25}$, the upper seven bits of the immediate field

**Figure 10.1**

## Testing script

I validate the design functionality by exhausted testing script that tests all the instructions as shown in Figure 11.1 and if in address 100 (25 in hexadecimal) value =25 then the design functionality is successfully completed

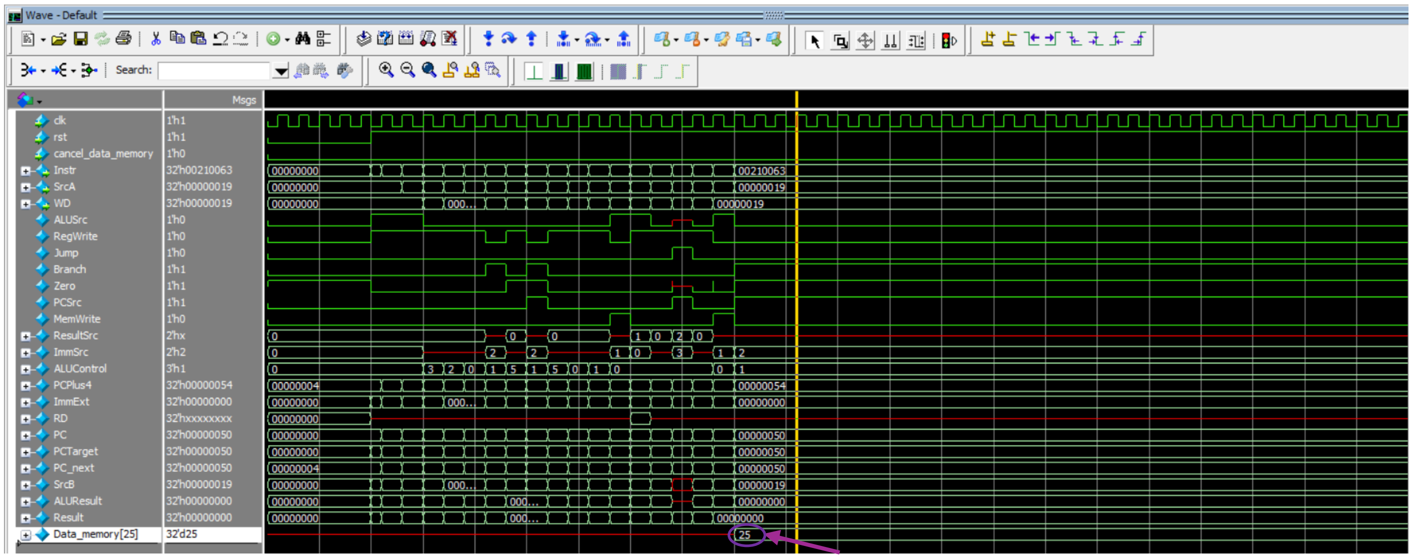


```
# riscvtest.s
# Sarah.Harris@unlv.edu
# David_Harris@hmc.edu
# 27 Oct 2020
#
# Test the RISC-V processor:
#   add, sub, and, or, slt, addi, lw, sw, beq, jal
# If successful, it should write the value 25 to address 100
#       RISC-V Assembly         Description                  Address   Machine Code
main:   addi x2, x0, 5          # x2 = 5                     0         00500113
        addi x3, x0, 12         # x3 = 12                    4         00C00193
        addi x7, x3, -9         # x7 = (12 - 9) = 3          8         FF718393
        or   x4, x7, x2         # x4 = (3 OR 5) = 7          C         0023E233
        and  x5, x3, x4         # x5 = (12 AND 7) = 4        10        0041F2B3
        add  x5, x5, x4         # x5 = 4 + 7 = 11            14        004282B3
        beq  x5, x7, end        # shouldn't be taken         18        02728863
        slt  x4, x3, x4         # x4 = (12 < 7) = 0          1C        0041A233
        beq  x4, x0, around     # should be taken            20        00020463
        addi x5, x0, 0          # shouldn't execute          24        00000293
around: slt  x4, x7, x2         # x4 = (3 < 5) = 1           28        0023A233
        add  x7, x4, x5         # x7 = (1 + 11) = 12         2C        005203B3
        sub  x7, x7, x2         # x7 = (12 - 5) = 7          30        402383B3
        sw   x7, 84(x3)         # [96] = 7                   34        0471AA23
        lw   x2, 96(x0)         # x2 = [96] = 7              38        06002103
        add  x9, x2, x5         # x9 = (7 + 11) = 18         3C        005104B3
        jal  x3, end            # jump to end, x3 = 0x44     40        008001EF
        addi x2, x0, 1          # shouldn't execute          44        00100113
end:    add  x2, x2, x9         # x2 = (7 + 18) = 25         48        00910133
        sw   x2, 0x20(x3)       # [100] = 25                 4C        0221A023
done:   beq  x2, x2, done       # infinite loop              50        00210063
```

**Figure 11.1** Testing script

## Quest-Sim Snapshots

In Figure 12.1 the result of my testing script will be shown in Address number 100 (25 in hexadecimal) and the design functionality is successfully completed



| | |
|---|---|
| clk | 1'h1 |
| rst | 1'h1 |
| cancel_data_memory | 1'h0 |
| Instr | 32'h00210063 |
| SrcA | 32'h00000019 |
| WD | 32'h00000019 |
| ALUSrc | 1'h0 |
| RegWrite | 1'h0 |
| Jump | 1'h0 |
| Branch | 1'h1 |
| Zero | 1'h1 |
| PCSrc | 1'h1 |
| MemWrite | 1'h0 |
| ResultSrc | 2'hx |
| ImmSrc | 2'h2 |
| ALUControl | 3'h1 |
| PCPlus4 | 32'h00000054 |
| ImmExt | 32'h00000000 |
| RD | 32'hxxxxxxxx |
| PC | 32'h00000050 |
| PCTarget | 32'h00000050 |
| PC_next | 32'h00000050 |
| SrcB | 32'h00000019 |
| ALUResult | 32'h00000000 |
| Result | 32'h00000000 |
| Data_memory[25] | 32'd25 |

Figure 12.1 Questa-Sim snapshots

## Elaborated Design Vivado

I used Artix-7 (XC7A35TICPG236-1L) FPGA on Vivado by using a 12 ns clock for my design and the elaborated design in Figure 13.1

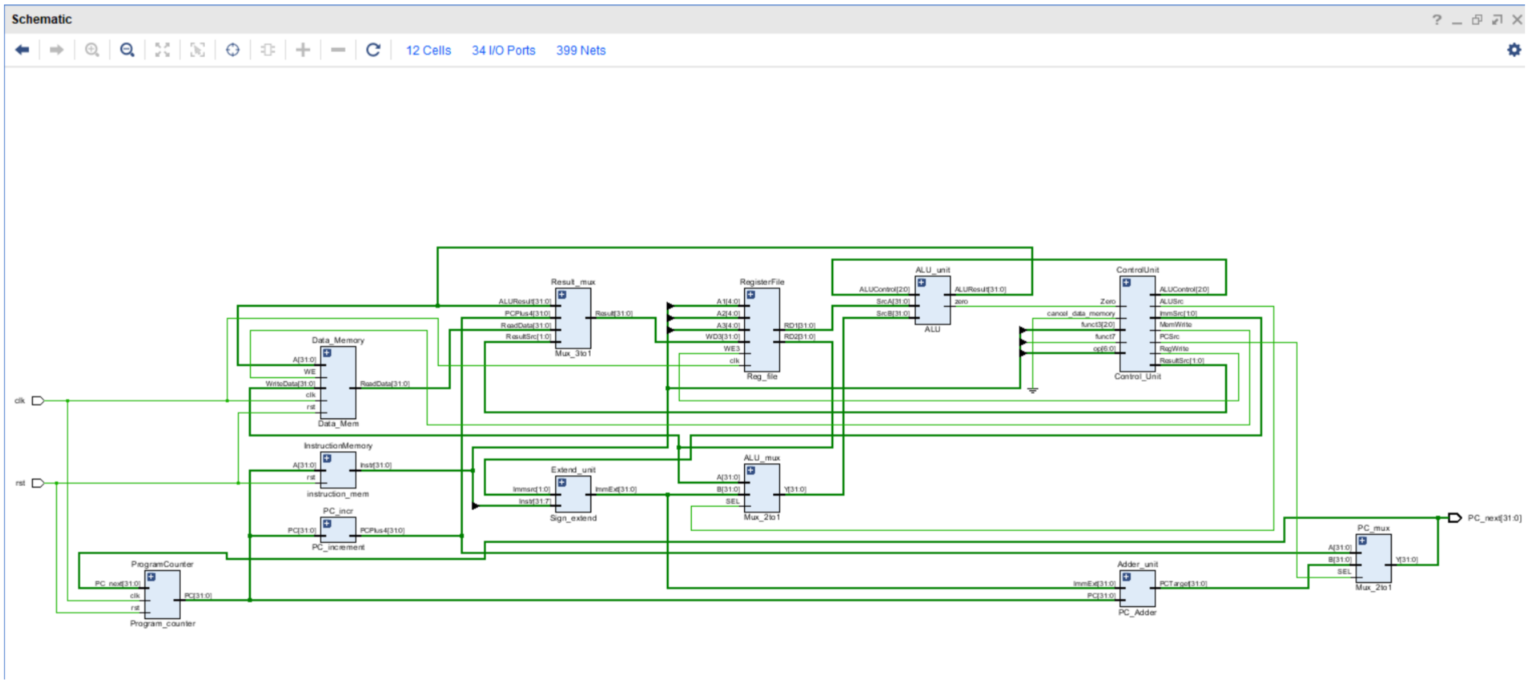


**Figure 13.1 Elaborated Design**

In Figure 13.2, the messages of the report of the elaborated design showing there are no errors

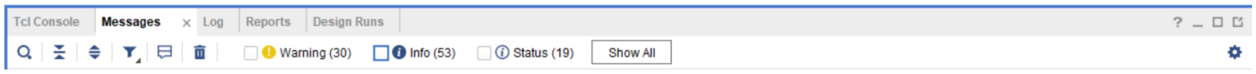


**Figure 13.2**

## Schematic Design Vivado

Now let's move forward to the next step which is synthesizing the design and Figures 14.1 and 15.1 shows the schematic of the synthesized design

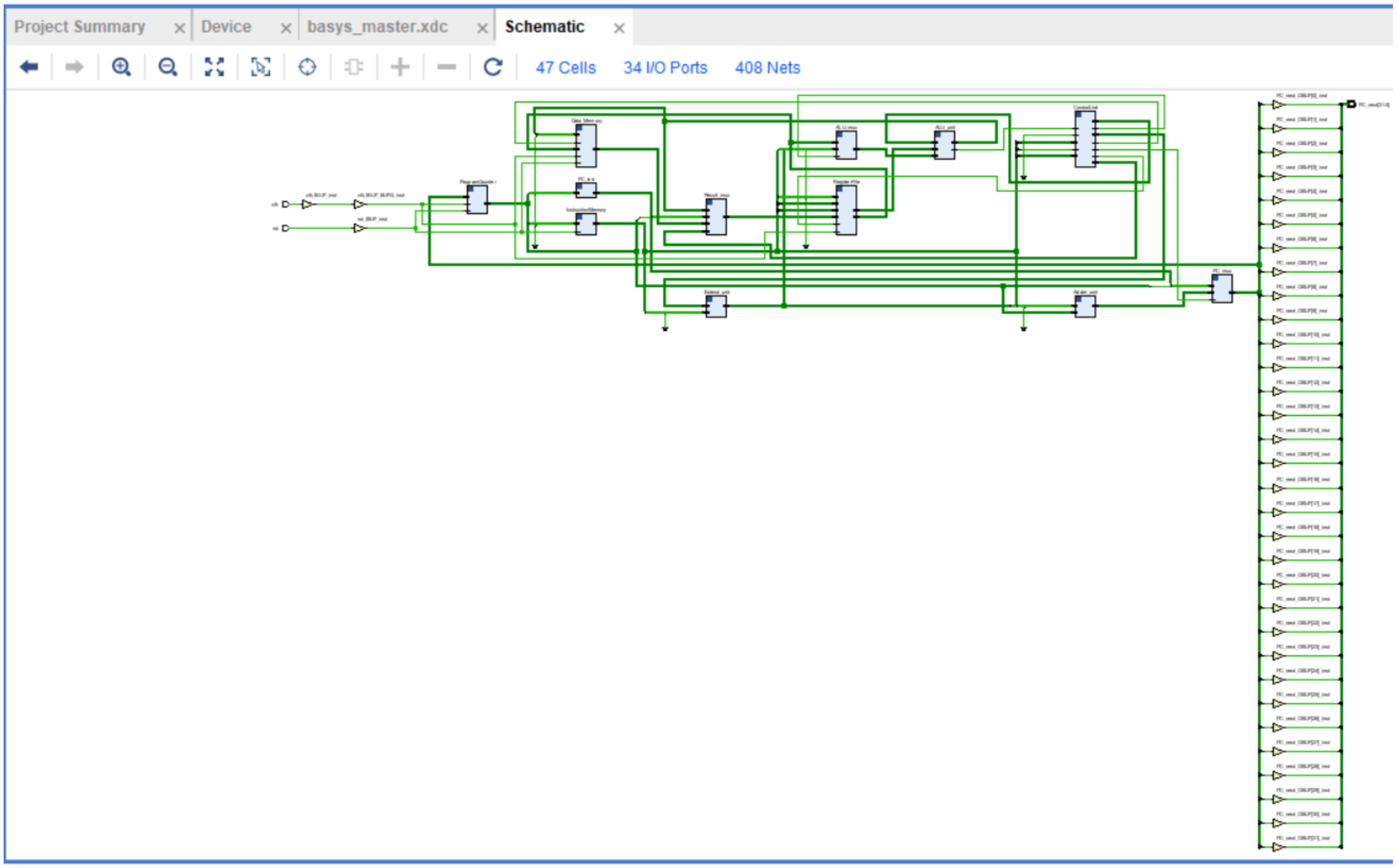


**Figure 14.1** **overall synthesized design**

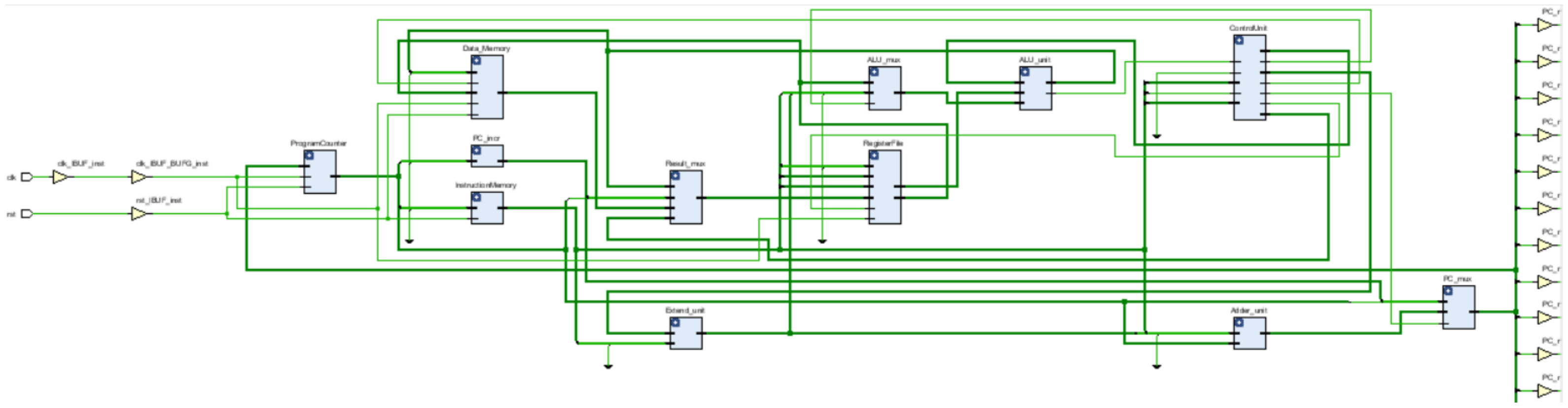
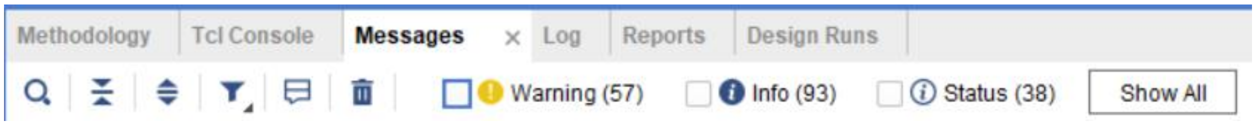


**Figure 15.1 Zoomed in Synthesized design**

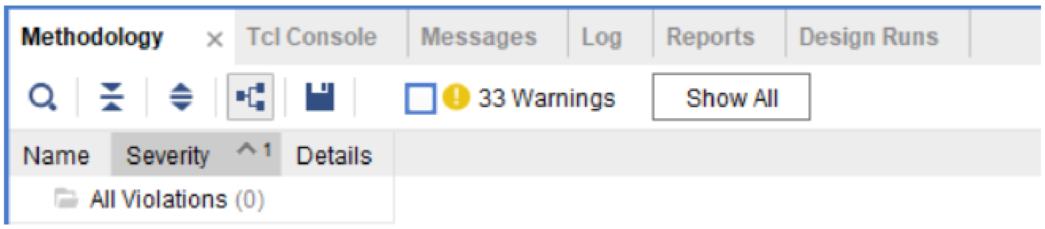
In Figures 15.2 and 15.3 shows that there are no errors in the synthesized design, and it is done successfully



Methodology | Tcl Console | **Messages** × | Log | Reports | Design Runs

Warning (57) Info (93) Status (38) Show All

**Figure 15.2 Messages of synthesized design**



**Methodology** × | Tcl Console | Messages | Log | Reports | Design Runs

33 Warnings Show All

| Name | Severity ^1 | Details |
|---|---|---|
| All Violations (0) | | |

**Figure 15.3 Methodology of synthesized design**

## Implementation Design Vivado

As we completed the synthesization of the design let's move forward to the implementation step as shown in Figure 16.1

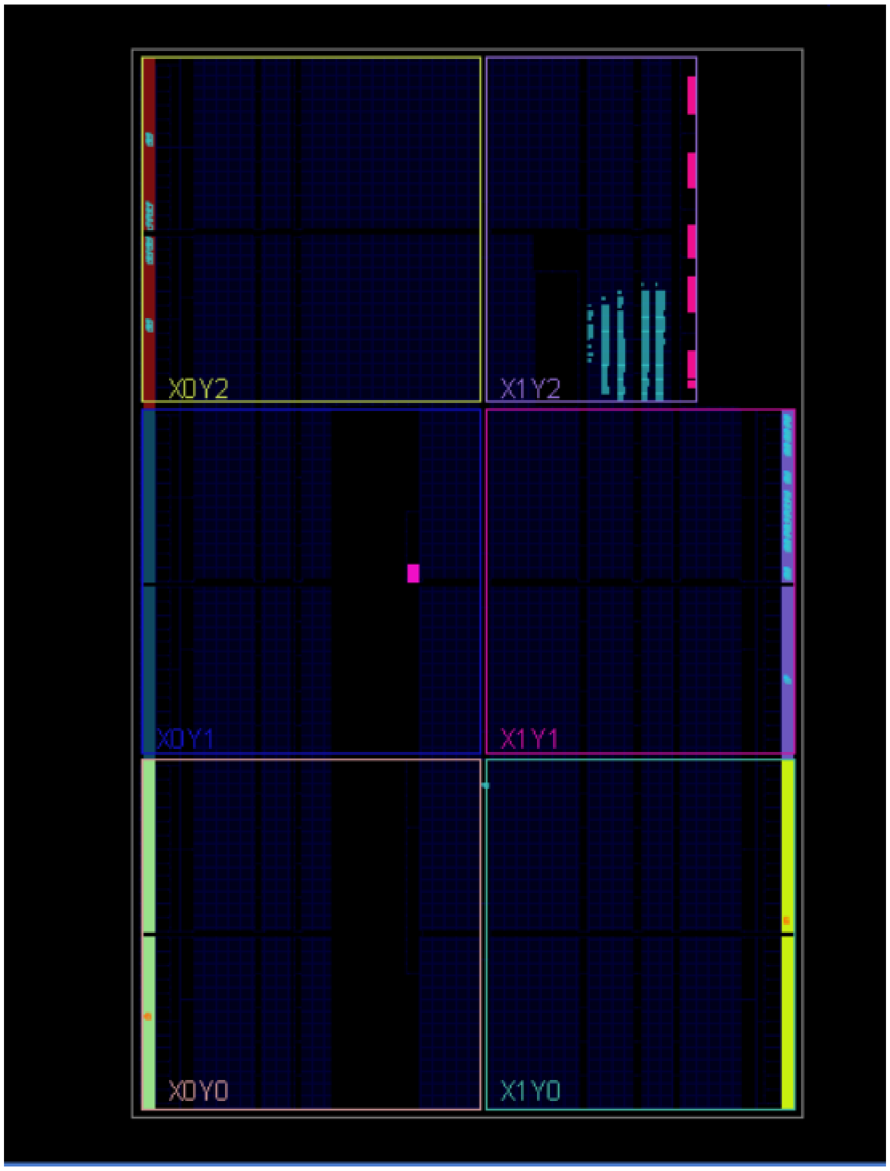


**Figure 16.1** Implementation Design

In Figures 17.1 and 17.2 show that there are no errors in the implemented design, and it is done successfully

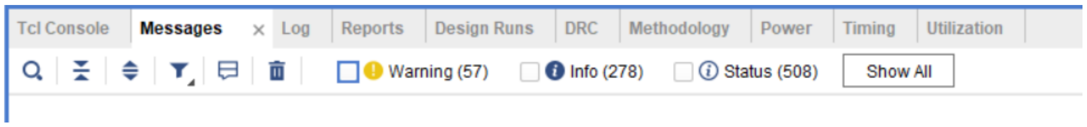
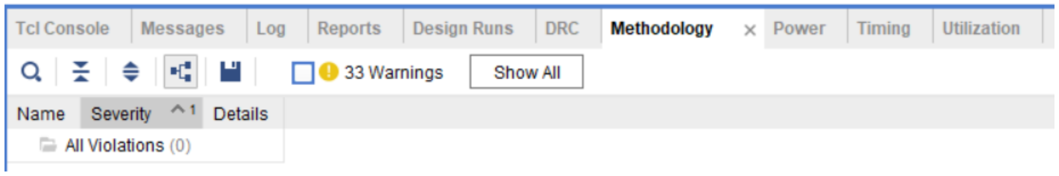


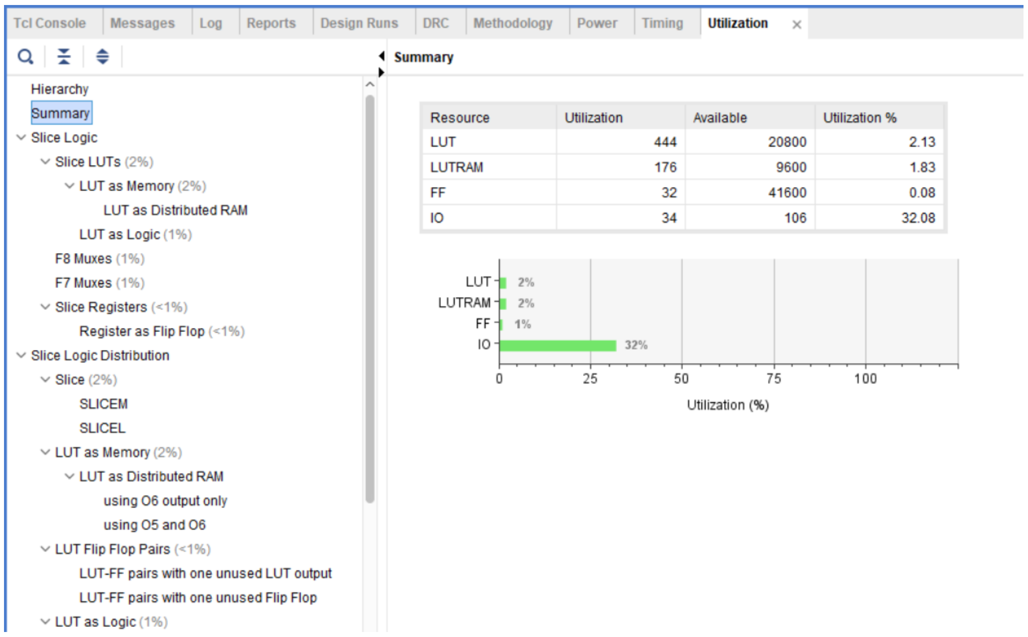
**Figure 17.1 Message of implemented design**



All Violations (0)

**Figure 17.2 Methodology of implemented design**

In Figure 17.3 shows the utilization of the implemented design on the FPGA used (Artix-7)

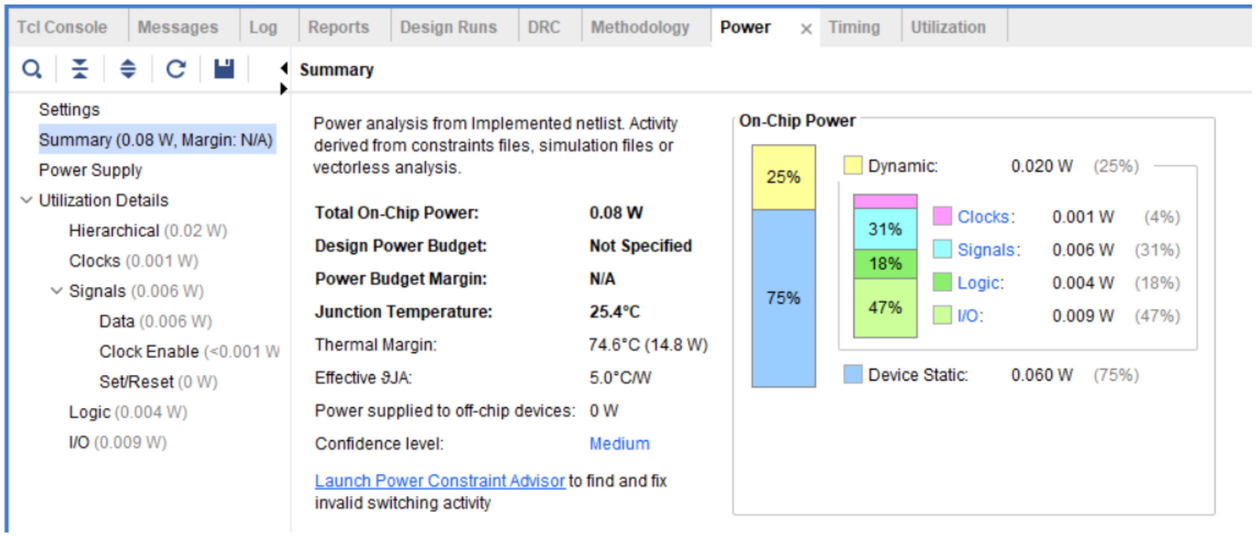


Hierarchy
Summary
Slice Logic
  Slice LUTs (2%)
    LUT as Memory (2%)
      LUT as Distributed RAM
    LUT as Logic (1%)
  F8 Muxes (1%)
  F7 Muxes (1%)
  Slice Registers (<1%)
    Register as Flip Flop (<1%)
Slice Logic Distribution
  Slice (2%)
    SLICEM
    SLICEL
  LUT as Memory (2%)
    LUT as Distributed RAM
      using O6 output only
      using O5 and O6
  LUT Flip Flop Pairs (<1%)
    LUT-FF pairs with one unused LUT output
    LUT-FF pairs with one unused Flip Flop
  LUT as Logic (1%)

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 444 | 20800 | 2.13 |
| LUTRAM | 176 | 9600 | 1.83 |
| FF | 32 | 41600 | 0.08 |
| IO | 34 | 106 | 32.08 |

**Figure 17.3 Utilization report of the implemented design**

In Figure 18.1 clarifies the Power report of the implementation and shows the static power and dynamic power of the design



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: | 74.6°C (14.8 W) |
| Effective ϑJA: | 5.0°C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Medium |

Launch Power Constraint Advisor to find and fix invalid switching activity

| | | | |
|---|---|---|---|
| 🟨 | Dynamic: | 0.020 W | (25%) |
| 🟪 | Clocks: | 0.001 W | (4%) |
| 🟦 | Signals: | 0.006 W | (31%) |
| 🟩 | Logic: | 0.004 W | (18%) |
| 🟩 | I/O: | 0.009 W | (47%) |
| 🟦 | Device Static: | 0.060 W | (75%) |

**Figure 18.1** **Power report of the implemented design**

## STA analysis and configurations

RISC-V is designed with 12 ns clock to solve any time violations such as hold time violations and setup time violations, and we will take a closer look at the lowest delay path that is related to the hold time and most delay path (critical path) that is related to the setup time and make sure that our design is clear of violations.

In Figure 19.1 shows the Critical path of the implemented design and in Figure 19.2 shows the lowest delay path of the design

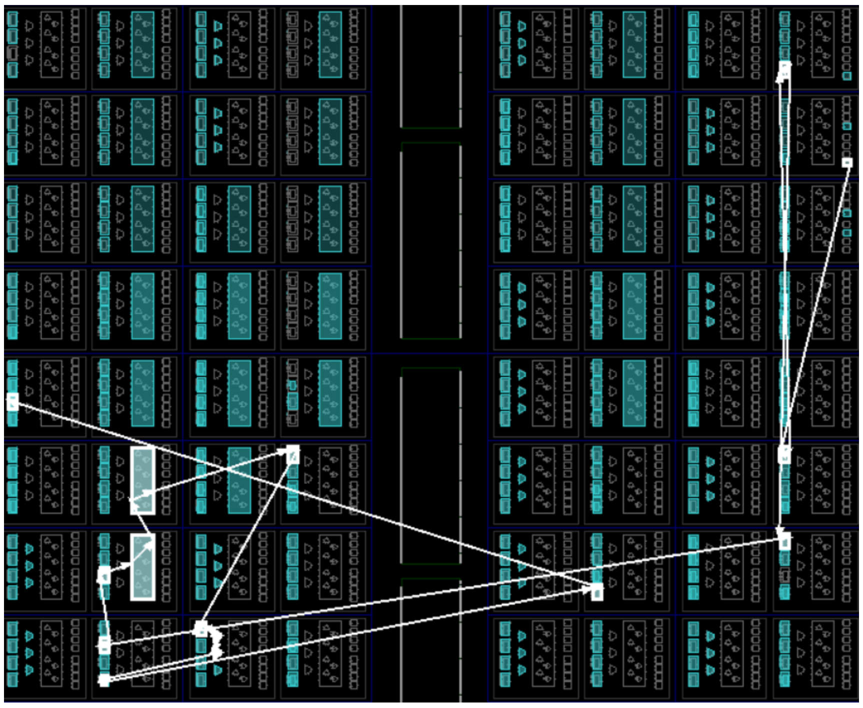


**Figure 19.1** Critical path

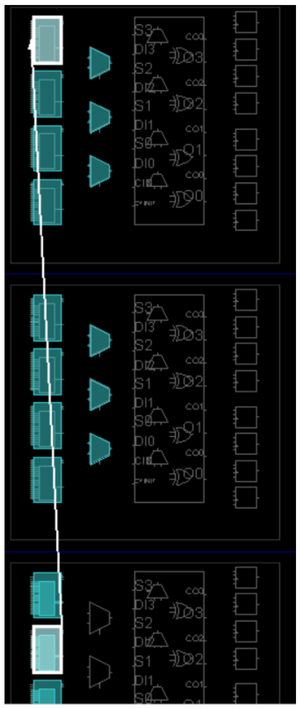
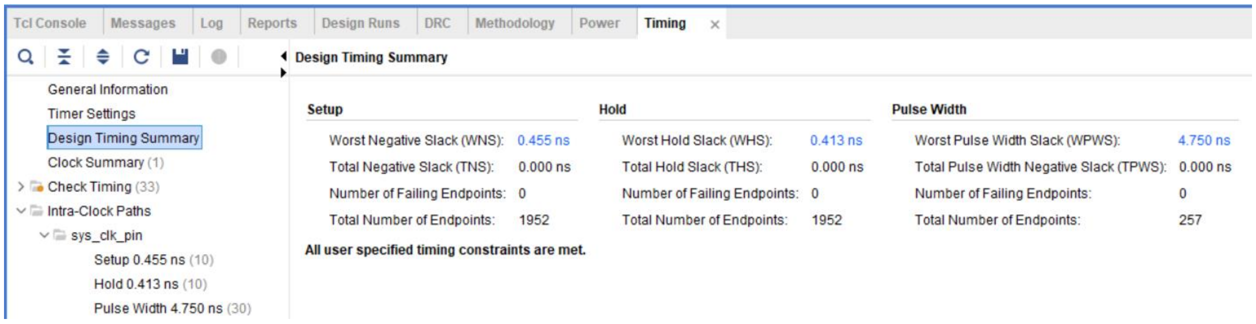


**Figure 19.2** lowest delay path

In Figure 20.1 the timing report of the implemented design is clear of any violations as the slack is a positive value

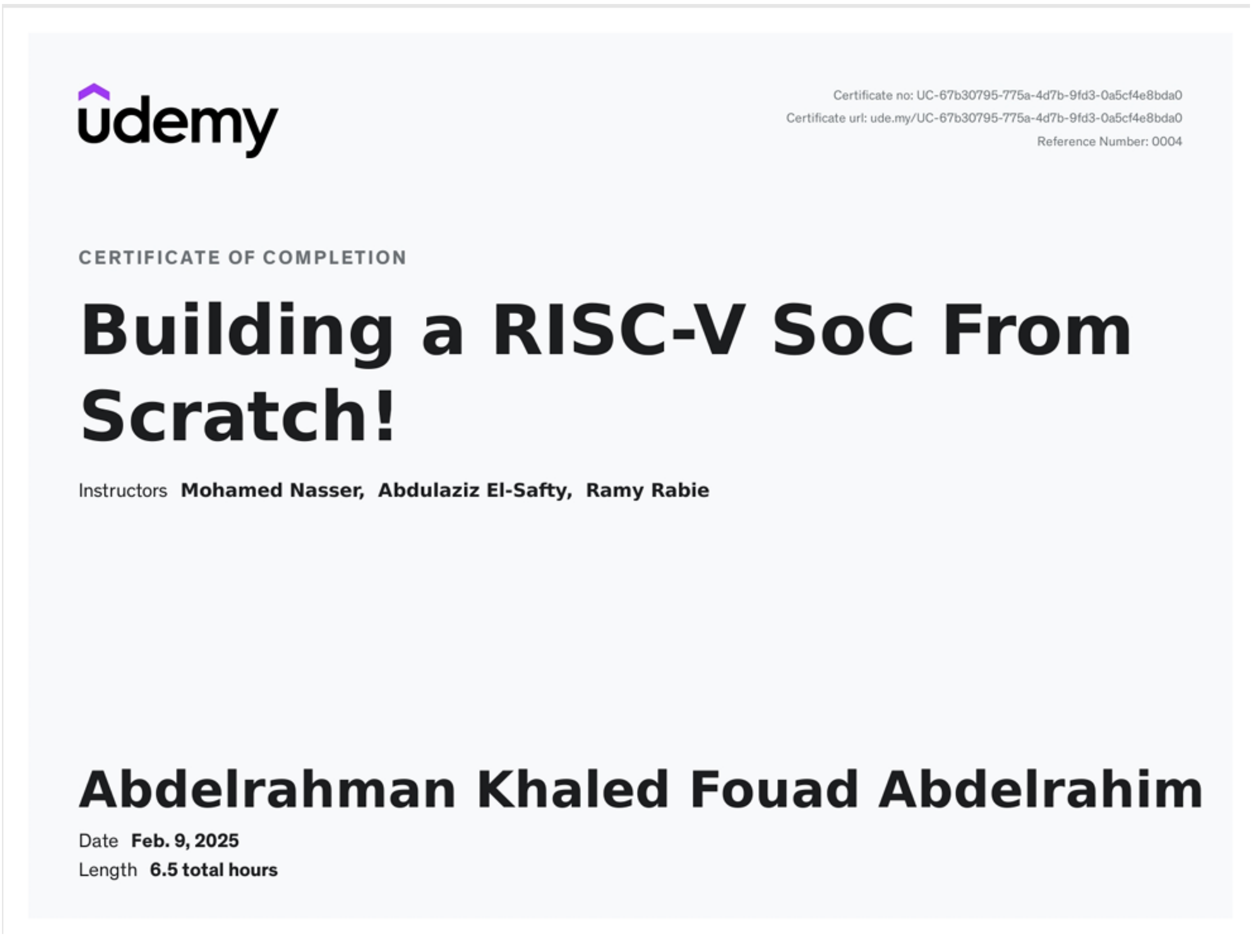


| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 0.455 ns | Worst Hold Slack (WHS): | 0.413 ns | Worst Pulse Width Slack (WPWS): | 4.750 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 1952 | Total Number of Endpoints: | 1952 | Total Number of Endpoints: | 257 |

All user specified timing constraints are met.

**Figure 20.1** **Timing report of the implemented design**

## Udemy Certificate



Certificate no: UC-67b30795-775a-4d7b-9fd3-0a5cf4e8bda0
Certificate url: ude.my/UC-67b30795-775a-4d7b-9fd3-0a5cf4e8bda0
Reference Number: 0004

CERTIFICATE OF COMPLETION

**Building a RISC-V SoC From Scratch!**

Instructors **Mohamed Nasser, Abdulaziz El-Safty, Ramy Rabie**

**Abdelrahman Khaled Fouad Abdelrahim**

Date **Feb. 9, 2025**
Length **6.5 total hours**