CS-431 DIGITAL SYSTEM DESIGN

OEL (Open-Ended Lab) Project Report

Project Title

RV32I Single Cycle Processor Implementation on FPGA

Submitted To

Miss Fauzia Yasir
Lecturer, Department of Computer and Information Systems Engineering
NED University of Engineering & Technology

Submitted By

Neha Nauman Khan (CS-22024)
Iqra Jawad Ahmad (CS-22034)

Submission Date: 23rd November 2025

TABLE OF CONTENTS

1. INTRODUCTION

This report presents the design, implementation, and verification of a **single-cycle RISC-V processor** based on the **RV32I Base Integer Instruction Set Architecture (ISA)**. The processor is implemented in **Verilog HDL** and deployed on a **Xilinx Artix-7 FPGA (Nexys A7-100T)**. The objective of this project is to understand the fundamental principles of computer architecture by constructing a fully functional 32-bit processor from covering datapath design, control logic, instruction decoding, ALU operation, memory interaction, and FPGA realization. The RV32I architecture was chosen due to its simplicity, openness, and widespread adoption in both academia and industry.

2. SCDP (RV32I)

A single-cycle datapath processor executes every instruction in a single clock cycle, meaning the entire process of fetching, decoding, and executing an instruction happens at once.

3. DESIGN METHODOLOGY

The design process followed four phases:

- Architectural Specification
- RTL Design & Synthesis
- Integration
- Verification & Simulation Results

3.1 ARCHITECTURAL SPECIFICATIONS

We first selected the supported instruction subset, identified datapath components, and created a block-level architecture.

3.1.1 INSTRUCTIONS

The **Base Integer ISA** consists of 47 instructions, out of which our design implements **37 core instructions**, including arithmetic, logical, load/store, branch, and jump operations. System-level instructions such as **ECALL**, **EBREAK**, **FENCE**, and CSR-related instructions are **treated as NOPs**, as they require privileged mode, exception handling, and trap logic, which are beyond the scope of this educational implementation.

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

3.1.2 BASE INSTRUCTION FORMATS

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	SB-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	UJ-type

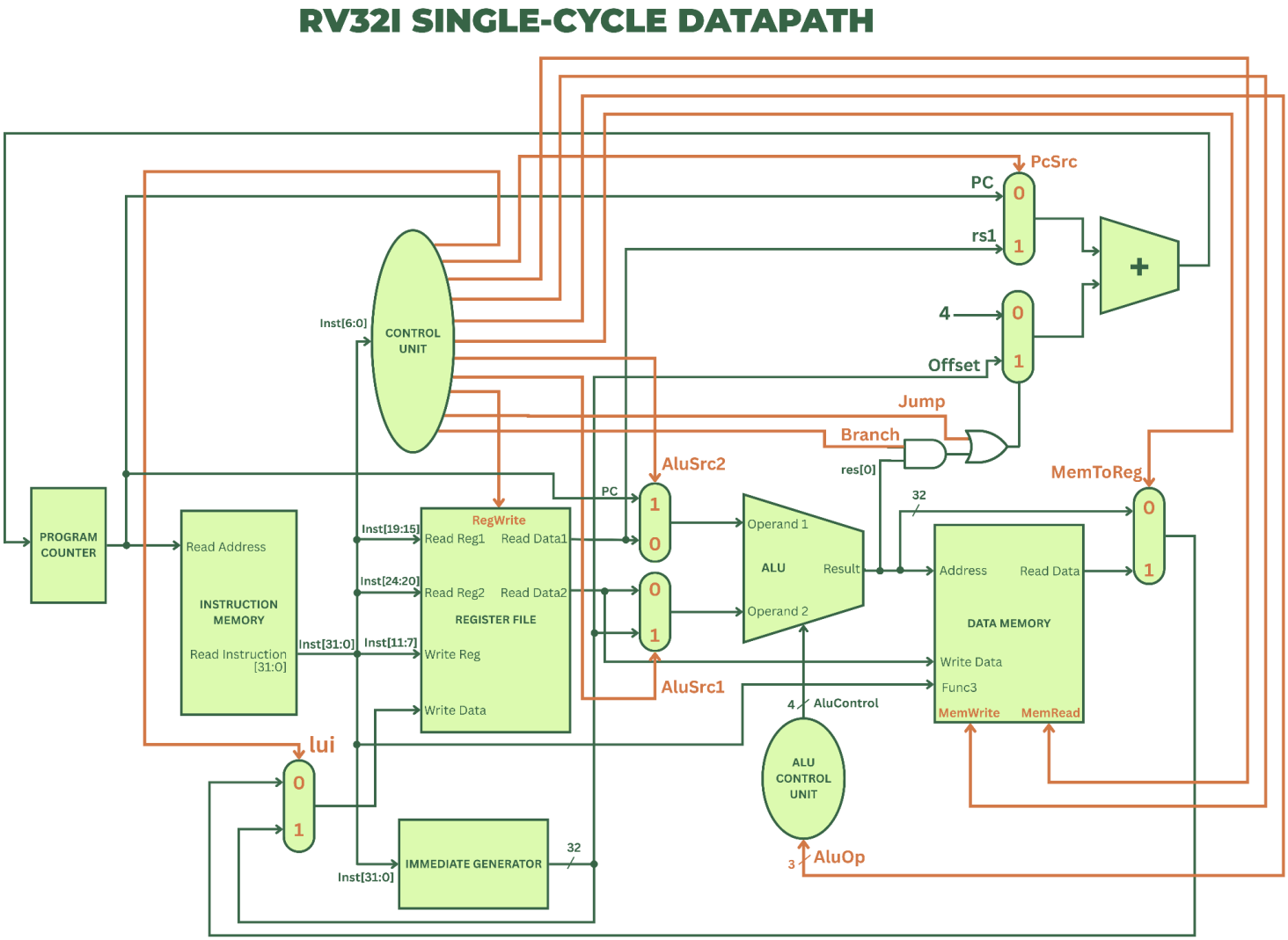
The table below lists all the instructions we included, along with their corresponding instruction formats

Instruction Format	Instructions
R	ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND, NOP
I	ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRAI, LB, LH, LW, LBU, LHU, JALR
S	SB, SH, SW
SB	BEQ, BNE, BLT, BGE, BLTU, BGEU
U	LUI, AUIPC
UJ	JAL

.

3.1.3 RV32I PROCESSOR DATAPATH

After analyzing the instruction requirements, we made a complete datapath diagram of our single cycle RV32I processor.



3.1.3 DESIGN MODULES OF SCDP RV32I

Our design follows a modular architecture and includes:

- **Program Counter**: Keeps track of the next instruction to fetch.
- **Instruction Memory**: Stores all the program instructions.
- **Register File**: 32 registers to read from and write results to.
- **Control Unit**: Generates control signals to manage the flow of data and operations.
- **ALU Control Unit**: Tells the ALU what operation to perform.
- **ALU**: Does all arithmetic and logic operations like add, sub, AND, OR, shifts, comparisons, etc.
- **Data Memory**: Stores and loads data for load/store instructions.
- **Immediate Generator**: Extracts and extends immediate values from instructions.
- **Miscellaneous**: wires, multiplexers, adders, and other small components needed to connect everything.
- **SCDP(Top Module)**: Integrated Design including all modules.
- **Hex to Seven Segment Converter**: For Displaying ALU result on FPGA's seven segment display.
- **Wrapper Module**: (Clock Divider) This module divides the FPGA's clock to allow us visualize the results on FPGA easily.

3.1.4 DESIGN MODULES OF SCDP RV32I

To ensure that the design remains **modular, scalable, and highly readable**, a dedicated package file named **RISCV_PKG.vh** was created. This file contains all global constants, parameterized definitions, ALU operation codes, and instruction-type encodings used throughout the processor. Centralizing these definitions not only improves code clarity but also allows the processor to be easily extended or modified without editing multiple files. Updating instruction types, ALU operations, memory size, word length, or the number of registers becomes as simple as changing a single line in the package file.

This parameterized approach is one of the unique strengths of our implementation because it eliminates hard-coded magic numbers, prevents inconsistencies across modules, and ensures maintainability in both simulation and FPGA implementation. All major modules, including the ALU, Control Unit, Register File, Instruction Decoder, and Datapath, import definitions directly from this package to maintain consistent behavior and symbolic naming.

RISCV_PKG.vh File:

```
// ======================================================== //
// RISCV_PKG.vh : Package file for RV32I SCDP Processor    //
// ======================================================== //
`define INSTRUCTION_SIZE 32
`define WORD_LENGTH      32
`define REG_COUNT        32
`define MEM_SIZE         128
// ALU Operation Codes
`define ADD                    4'b0000
`define SUB                    4'b0001
`define less_than              4'b0010
`define less_than_unsigned     4'b0011
`define greater_than           4'b0100
`define greater_than_unsigned  4'b0101
`define XOR                    4'b0110
`define OR                     4'b0111
`define AND                    4'b1000
`define SLL                    4'b1001
`define SRL                    4'b1010
`define SRA                    4'b1011
`define equal                  4'b1100
`define not_equal              4'b1101
`define pc_plus_4              4'b1110
```

```
// ALUOp [3:0]
`define R_TYPE      3'b000 // ADD, SUB, AND, OR, XOR, SLT, SLTU, SLL, SRL, SRA
`define I_TYPE      3'b001 //ADDI, ANDI, ORI, XORI, SLTI, SLTIU, SLLI, SRLI, SRAI, JALR
`define STORE       3'b010 // SB, SH, SW
`define BRANCH      3'b011 // BEQ, BNE, BLT, BGE, BLTU, BGEU
`define U_TYPE      3'b100 // LUI, AUIPC
`define JUMP        3'b101 // JAL, JALR
`define LOAD        3'b110 // LW, LH, LB, LHU, LBU
`define NOP         3'b111 // No operation
```

3.2 RTL DESIGNS AND SYNTHESIS

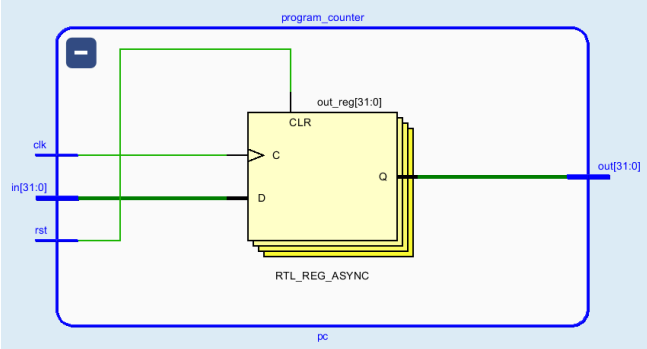
Earlier, we mentioned all the design modules of RV32I. Here is the Verilog code and the synthesised RTL Schematic of each of those modules. Each module is implemented in Verilog HDL and synthesised using the Vivado EDA tool.

3.2.1 Program Counter

Verilog Code:

```
`include "RISCV_PKG.vh"
module pc(
    input clk, rst,
    input [`INSTRUCTION_SIZE - 1 : 0] in,
    output reg [`INSTRUCTION_SIZE - 1 : 0] out);
always @(posedge clk) begin //Processors have synchronous reset
    if (rst) begin
        out <= 32'b0;
    end else begin
        out <= in;
    end
end
endmodule
```

RTL Diagram:



3.2.2 INSTRUCTION MEMORY

The instruction memory is implemented as a synchronous ROM initialized using the Verilog `$readmemh()` system task. The contents of the memory are stored in an external hexadecimal file (`MachineCode.mem`), which contains the machine code of the test assembly program in **little-endian** order, following the RISC-V specification. During simulation and FPGA synthesis, the file is automatically loaded into the instruction memory, enabling the processor to fetch and execute instructions exactly as it would in real hardware.

MachineCode.mem File:

B7 00 00 00
17 01 00 00
93 01 50 00
13 02 A0 00
93 B2 91 00

```
13 43 32 00
B3 83 51 00
33 74 62 00
B3 E4 51 00
33 C5 41 00
23 20 70 00
23 02 80 00
23 14 90 00
83 25 00 00
03 46 40 00
83 56 80 00
63 84 35 00
63 92 35 00
63 C2 41 00
63 52 32 00
63 E2 41 00
63 72 32 00
6F 07 80 00
93 07 90 00
13 08 17 00
67 00 08 00
6F 00 00 00
```

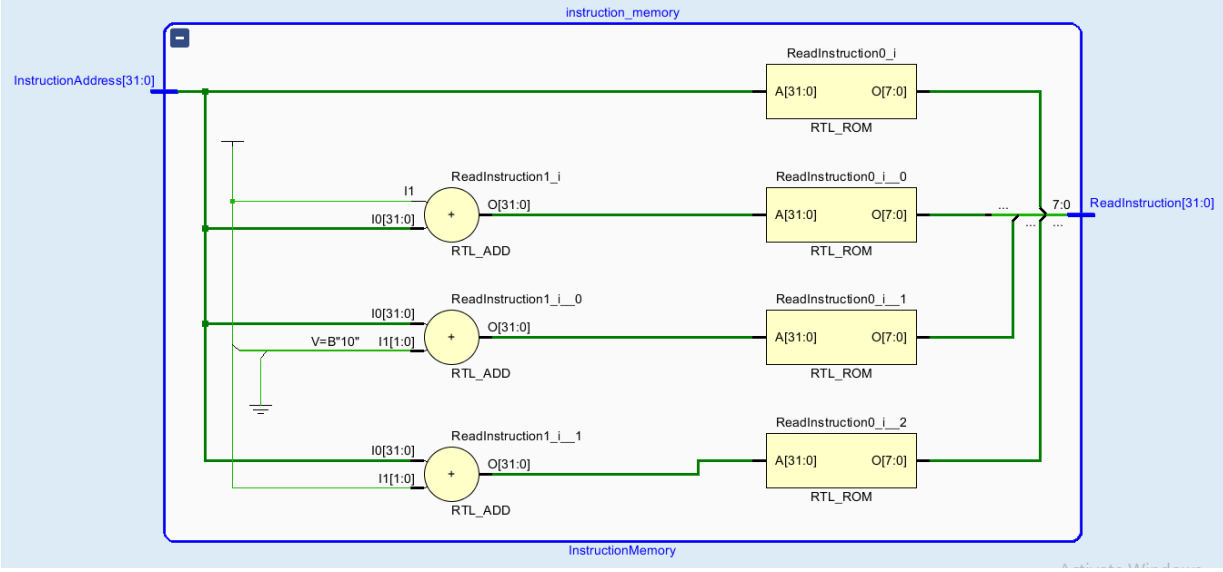
Code in RISCV Assembly:

```
# U-TYPE TESTS
        lui   x1, 0x0
        auipc x2, 0
# I-TYPE TESTS
        addi  x3, x0, 5
        addi  x4, x0, 10
        sltiu x5, x3, 9
        xori  x6, x4, 3
# R-TYPE TESTS
        add   x7, x3, x5
        and   x8, x4, x6
        or    x9, x3, x5
        xor   x10, x3, x4
# STORE / LOAD TESTS
        sw    x7, 0(x0)
        sb    x8, 4(x0)
        sh    x9, 8(x0)
        lw    x11, 0(x0)
        lbu   x12, 4(x0)
        lhu   x13, 8(x0)
# BRANCH TESTS
        beq   x11, x3, skip1 # NOT taken
        bne   x11, x3, skip1 # TAKEN
        addi x0, x0, 0
skip1: blt   x3, x4, skip2  # TAKEN
        addi x0, x0, 0
skip2: bge   x4, x3, skip3  # TAKEN
       addi x0, x0, 0
skip3: bltu  x3, x4, skip4  # TAKEN
       addi x0, x0, 0
skip4: bgeu  x4, x3, skip5  # TAKEN
       addi x0, x0, 0
# JUMP TESTS — OFFSETS FIXED
skip5: jal   x14, jump_target
       addi  x15, x0, 9     # should be SKIPPED
jump_target: addi  x16, x14, 1 # return PC + 1
             jalr  x0, x16, 0  # return
# FINAL HALT LOOP
halt: jal x0, halt   # infinite loop
```

Verilog Code:

```
`include "RISCV_PKG.vh"
module InstructionMemory(
    input [`INSTRUCTION_SIZE-1:0] InstructionAddress,
    output reg [`INSTRUCTION_SIZE-1:0] ReadInstruction);
  reg [7:0] Memory[0:`MEM_SIZE - 1]; // Byte-addressable memory with memory size in bytes
  initial begin
     $readmemh("MachineCode.mem", Memory);
  end
  always @(*) begin
      ReadInstruction[7:0]=Memory[InstructionAddress+0];//little-endian  format:LSB  at  lowest
address
    ReadInstruction[15:8]  = Memory[InstructionAddress + 1];
    ReadInstruction[23:16] = Memory[InstructionAddress + 2];
    ReadInstruction[31:24] = Memory[InstructionAddress + 3];
    end
endmodule
```

RTL Diagram:



3.2.3 DATA MEMORY

This Data Memory module implements the byte-addressable memory used by our RISC-V processor for all load and store operations. The memory follows a **little-endian** layout, meaning the least significant byte is stored at the lowest address. The module supports all basic RISC-V load/store variants using the 3-bit **funct3** field, which determines the access size and whether sign-extension or zero-extension is applied.

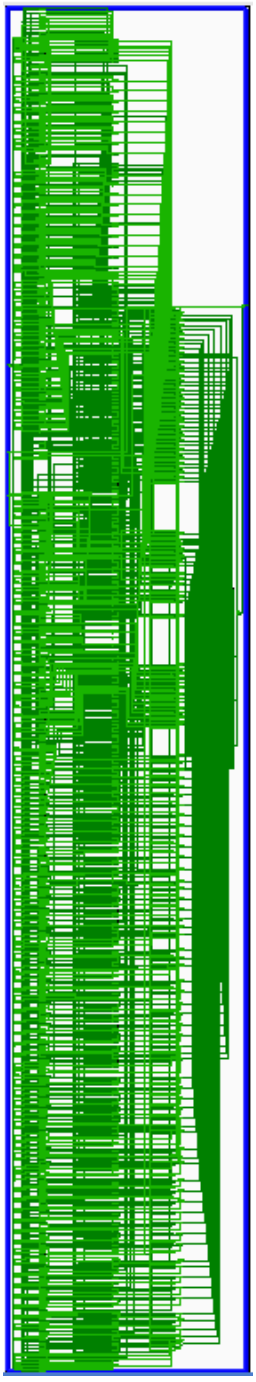
For **store** instructions, SB writes a single byte, SH writes two bytes (half-word), and SW writes a full 32-bit word. During **load** operations, the module returns either a signed or unsigned value depending on `funct3`: LB, LH, and LW return sign-extended data, while LBU and LHU use zero-extension. Memory is cleared to zero on reset. This design ensures correct handling of variable-sized accesses while maintaining compatibility with the RISC-V RV32I memory model.

Verilog Code:

```
`include "RISCV_PKG.vh"
module datamemory(
    input clk, reset, mem_read, mem_write,
    input[$clog2(`MEM_SIZE)-1:0]address,
    input [`INSTRUCTION_SIZE-1:0] write_data,
    input [2:0] funct3,
    output reg [`INSTRUCTION_SIZE-1:0] read_data);
    integer k;
    reg [7:0] mem [0:`MEM_SIZE-1];  // byte-addressable memory
    // WRITE Operation
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            for (k = 0; k < `MEM_SIZE; k = k + 1)
                mem[k] <= 8'h00;
        end
        else if (mem_write) begin
            case (funct3)
                3'b000: begin  // SB
                    mem[address] <= write_data[7:0];
                end
                3'b001: begin  // SH
                    mem[address]     <= write_data[7:0];
                    mem[address + 1] <= write_data[15:8];
                end
                3'b010: begin  // SW
```

```
                    mem[address]     <= write_data[7:0];
                    mem[address + 1] <= write_data[15:8];
                    mem[address + 2] <= write_data[23:16];
                    mem[address + 3] <= write_data[31:24];
                end
                default: ; // do nothing
            endcase
        end
    end
    // READ Operation (Combinational)
    always @(*) begin
      if (mem_read) begin
        case (funct3)
         3'b000:read_data = {{24{mem[address][7]}}, mem[address]};//LB
         3'b100:read_data = {{24{1'b0}}, mem[address]};//LBU
         3'b001: read_data ={{16{mem[address+1][7]}},mem[address+1],mem[address]};// LH
         3'b101:  read_data = {{16{1'b0}}, mem[address+1], mem[address]};// LHU
         3'b010:read_data ={mem[address+3],mem[address+2],mem[address+1],mem[address]};//LW
         default: read_data = 32'b0;
        endcase
      end else begin
         read_data = 32'b0;
      end
  end
endmodule
```

RTL Diagram:



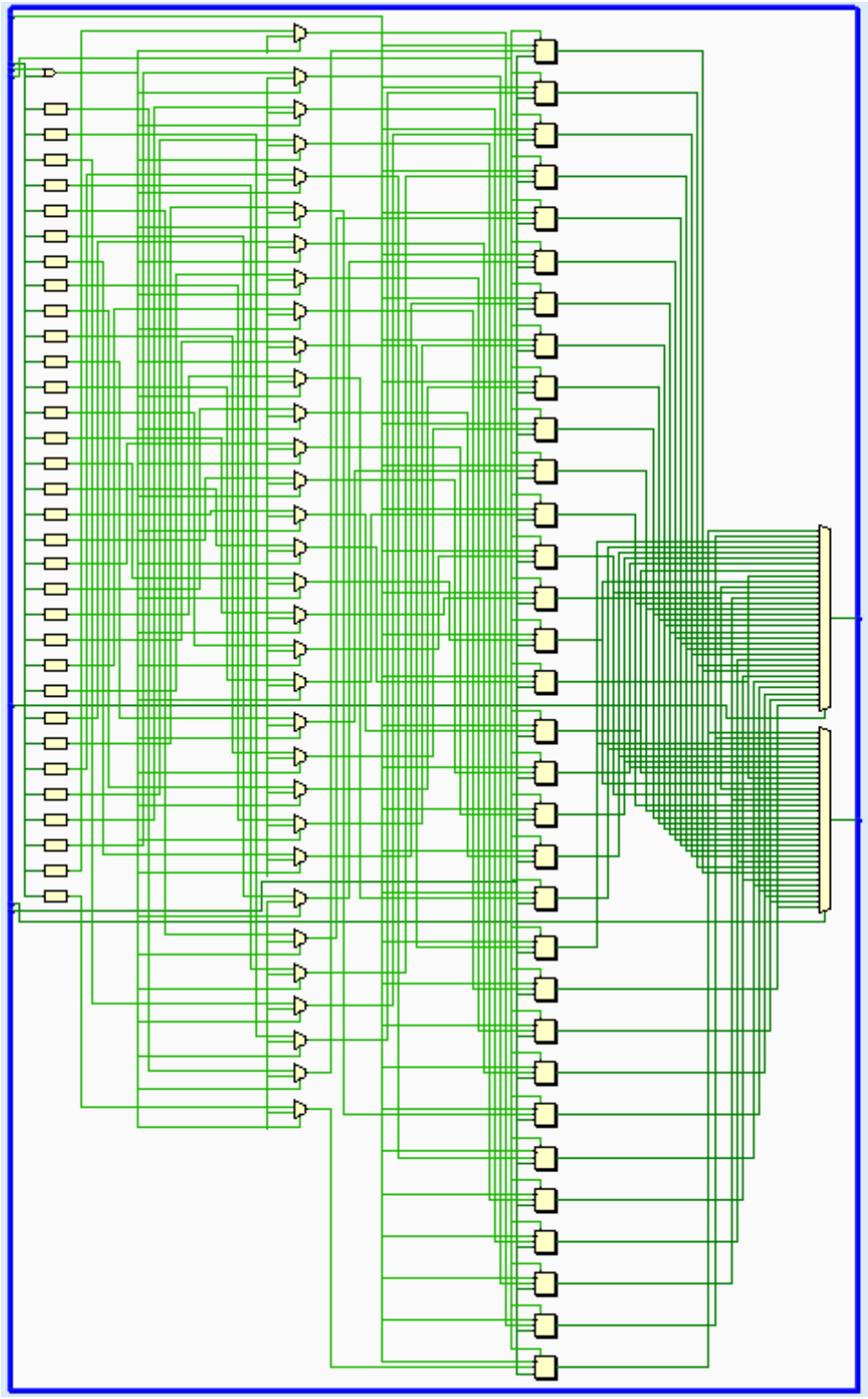
3.2.3 REGISTER FILE

Verilog Code:

```
`include "RISCV_PKG.vh"
module regfile(
 input clk, rst, regwrite,
 input [$clog2(`REG_COUNT) - 1 : 0] rs1, rs2, rd,
 input [`INSTRUCTION_SIZE - 1 : 0] write_data,
 output [`INSTRUCTION_SIZE - 1 : 0] read_data1, read_data2
 );

 integer k;
 reg [`INSTRUCTION_SIZE - 1 : 0] Registers [`REG_COUNT - 1 : 0];  //32 reg each of 32 bit
 assign read_data1=Registers[rs1];
 assign read_data2=Registers[rs2];
```

```
 always @ (posedge clk)
    begin
        if (rst) begin
            for (k=0;k<`REG_COUNT;k=k+1)
                Registers[k]<=32'b0;
        end
        else begin
            if(regwrite && rd) Registers[rd]<= write_data;
        end
    end
endmodule
```

RTL Diagram:



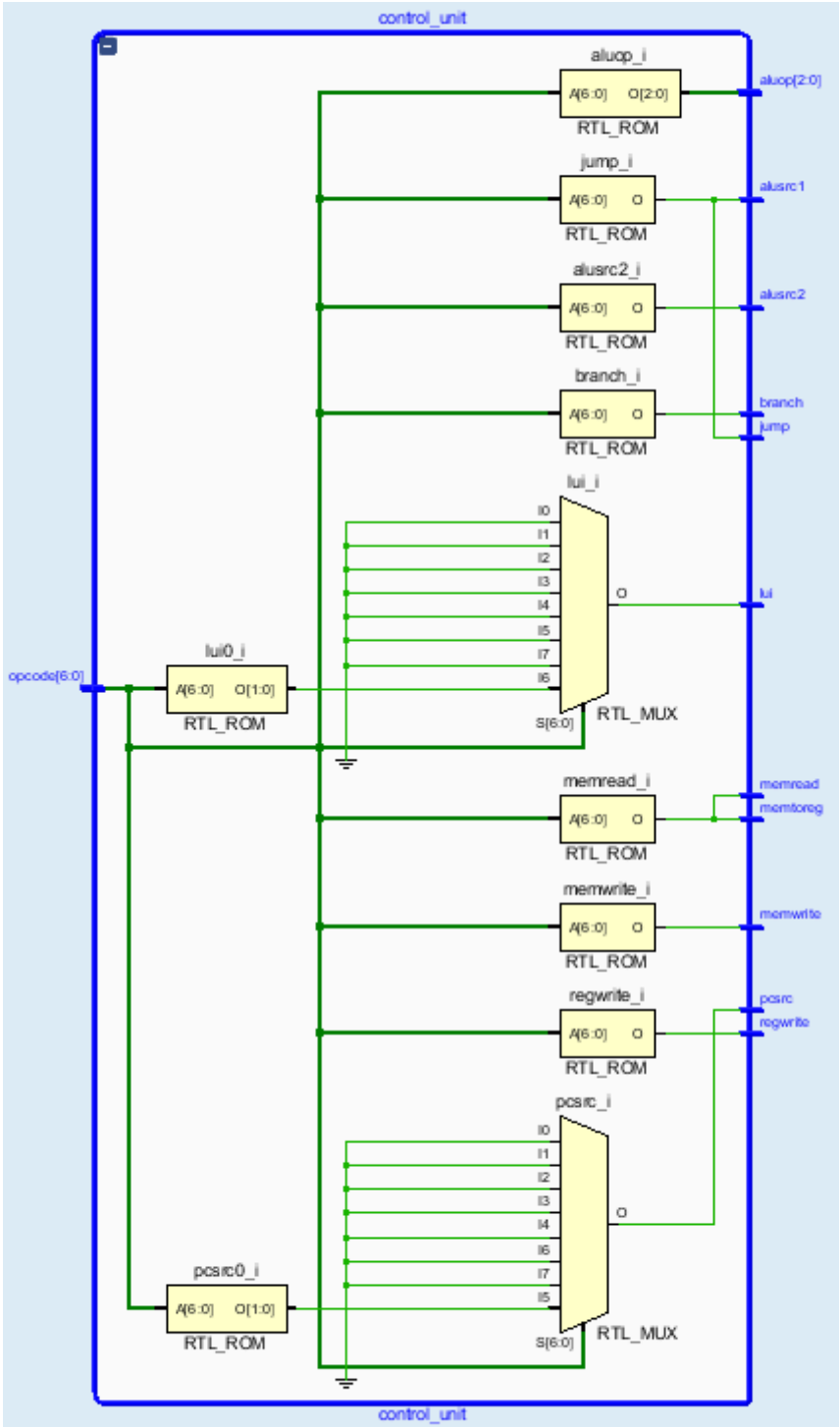
3.2.4 CONTROL UNIT

The Control Unit is responsible for decoding the 7-bit opcode of a RISC-V instruction and generating all the control signals required by the datapath. Based on the instruction type (R-type, I-type, Load, Store, Branch, Jump, or U-type), this module enables or disables specific components such as the register file, ALU, memory unit, and PC update logic.

Verilog Code:

```
`include "RISCV_PKG.vh"
module control_unit(
    input [6:0]opcode,
     output reg regwrite, memread, memwrite, branch, jump, memtoreg, alusrc1, alusrc2, lui,
pcsrc,
    output reg [2:0] aluop);
always @(*) begin
    // Default values
     regwrite = 0; memread = 0; memwrite = 0; branch = 0; jump = 0; memtoreg = 0; alusrc1 = 0;
alusrc2 = 0; lui = 0; pcsrc = 0; aluop = `NOP;

    case (opcode)
        7'b0110011: begin // R-type
            regwrite = 1;
            aluop    = `R_TYPE;
        end
        7'b0010011: begin // I-type
            regwrite = 1;
            alusrc2  = 1;
            aluop    = `I_TYPE;
        end
        7'b0000011: begin // Load
            regwrite = 1;
            memread  = 1;
            alusrc2  = 1;
            memtoreg = 1;
            aluop    = `LOAD;
        end
        7'b0100011: begin // Store
            memwrite = 1;
            alusrc2  = 1;
            aluop    = `STORE;
        end
        7'b1100011: begin // Branch
            branch   = 1;
            aluop    = `BRANCH;
        end
        7'b1101111, 7'b1100111: begin // JAL and JALR
            regwrite = 1;
            jump     = 1;
            alusrc1  = 1; // For JALR immediate offset
            pcsrc    = (opcode == 7'b1100111) ? 1 : 0; // JALR only
            aluop    = `JUMP;
        end
        7'b0110111, 7'b0010111: begin // LUI and AUIPC
            regwrite = 1;
            alusrc2  = 1;
            lui      = (opcode == 7'b0110111) ? 1 : 0; // LUI only
            aluop    = `U_TYPE;
        end
        default: ; // Do nothing; defaults already set

    endcase
end
endmodule
```

RTL Diagram:



3.2.5 ALU CONTROL UNIT

The ALU Control Unit receives three key fields from the instruction, `aluop` (generated by the main Control Unit), `funct3`, and `funct7`, and uses them to determine the exact arithmetic or logical operation the ALU must perform. While the Control Unit only identifies the broad instruction type (R-type, I-type, Load, Store, Branch, etc.), the ALU Control Unit performs the fine-grained decoding needed to distinguish between operations like ADD vs SUB, SRL vs SRA, or signed vs unsigned comparisons.

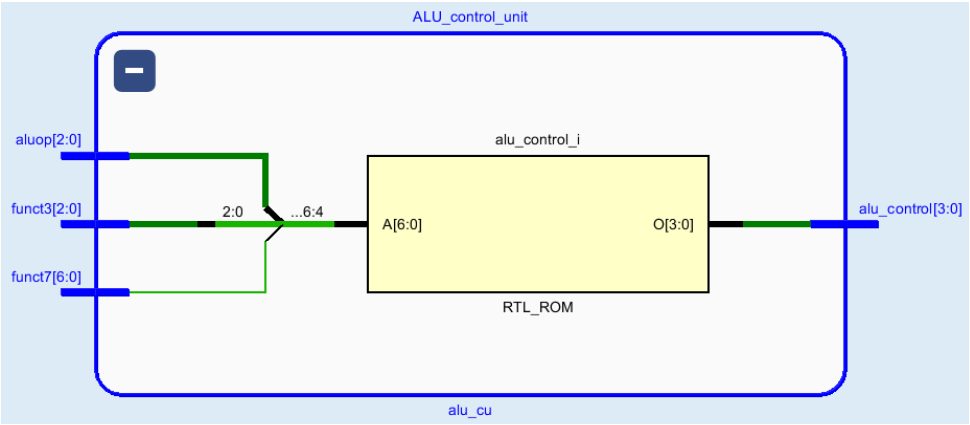
Using a `casez` pattern match, the module checks combinations of `aluop`, the MSB of `funct7` (for ADD/SUB and SRL/SRA differentiation), and `funct3` to generate a 4-bit `alu_control` signal. This signal selects one of the implemented ALU operations such as ADD, SUB, AND, OR, XOR, shifts (SLL, SRL, SRA), signed/unsigned comparisons (SLT, SLTU), and branch condition checks (BEQ, BNE, BLT, BGE, BLTU, BGEU).
Jump instructions map to a special `pc_plus_4` ALU operation. Unknown or unsupported patterns default safely to zero.

This module acts as the “precise operation decoder” for the processor, ensuring that each instruction invokes the correct ALU function.

Verilog Code:

```
`include "RISCV_PKG.vh"
module alu_cu(
    input [2:0] aluop,
    input [2:0] funct3,
    input [6:0] funct7,
    output reg [3:0] alu_control);
always @(*) begin
    casez ({aluop, funct7[5], funct3})

           {`STORE, 1'b?, 3'b???}, {`U_TYPE, 1'b?, 3'b???}, {`NOP, 1'b?, 3'b???}, {`R_TYPE,
1'b0, 3'b000}, {`I_TYPE, 1'b?, 3'b000}, {`LOAD, 1'b?, 3'b???}: alu_control = `ADD;  // ADD,
LW, SW, AUIPC, ADDI, NOP
        {`R_TYPE, 1'b1, 3'b000}: alu_control = `SUB;  // SUB
             {`BRANCH, 1'b?, 3'b100},{`I_TYPE,   1'b?, 3'b010},{`R_TYPE,   1'b0, 3'b010}:
alu_control = `less_than;  // SLT, SLTI, BLT
        {`BRANCH, 1'b?, 3'b101}: alu_control = `greater_than;  // BGE
            {`BRANCH, 1'b?, 3'b110}, {`I_TYPE,   1'b?, 3'b011}, {`R_TYPE,   1'b0, 3'b011}:
alu_control = `less_than_unsigned;  // SLTU, SLTIU, BLTU
        {`BRANCH, 1'b?, 3'b111}: alu_control = `greater_than_unsigned;  // BGEU
        {`R_TYPE, 1'b0, 3'b100}, {`I_TYPE, 1'b?, 3'b100}: alu_control = `XOR;  // XOR, XORI
        {`R_TYPE, 1'b0, 3'b110}, {`I_TYPE, 1'b?, 3'b110}: alu_control = `OR;   // OR, ORI
        {`R_TYPE, 1'b0, 3'b111}, {`I_TYPE, 1'b?, 3'b111}: alu_control = `AND;  // AND, ANDI
        {`I_TYPE, 1'b0, 3'b001}, {`R_TYPE, 1'b0, 3'b001}: alu_control = `SLL;  // SLL, SLLI
        {`I_TYPE, 1'b0, 3'b101}, {`R_TYPE, 1'b0, 3'b101}: alu_control = `SRL;  // SRL, SRLI
        {`R_TYPE, 1'b1, 3'b101}, {`I_TYPE, 1'b1, 3'b101}: alu_control = `SRA;  // SRA, SRAI
        {`BRANCH, 1'b?, 3'b000}: alu_control = `equal;      // BEQ
        {`BRANCH, 1'b?, 3'b001}: alu_control = `not_equal;  // BNE
        {`JUMP, 1'b?, 3'b???}: alu_control = `pc_plus_4;  // JAL and JALR
        default: alu_control = 4'b0000;
    endcase
end
endmodule
```

RTL Diagram:



3.2.6 ALU (ARITHMETIC & LOGIC UNIT)

The ALU (Arithmetic Logic Unit) executes all arithmetic, logical, and comparison operations required by the RV32I SCDP processor. It receives two 32-bit operands (`rs1` and `rs2`) along with a 4-bit `alu_control` signal generated by the ALU Control Unit. Based on this control code, the ALU performs operations such as addition, subtraction, logical AND/OR/XOR, shift operations (SLL, SRL, SRA), signed and unsigned comparisons, and jump PC updates.

For branch instructions (BEQ, BNE, BLT, BGE, BLTU, BGEU), the ALU is responsible only for **evaluating the branch condition**, not performing the branch itself.
To keep the design simple and clean, the ALU outputs **comparison results as a single-bit flag stored in the LSB of the 32-bit result**:

- `result = 1` → condition is **true** (branch should be taken)
- `result = 0` → condition is **false** (branch should not be taken)

The remaining 31 bits are padded with zeros.

Examples:

- `equal` (BEQ): sets result = `1` if `rs1 == rs2`, else `0`
- `not_equal` (BNE): sets result = `1` if `rs1 != rs2`, else `0`
- `less_than` (BLT): sets result = `1` if signed rs1 < signed rs2
- `greater_than_unsigned` (BGEU): result = `1` if unsigned rs1 ≥ rs2

The **branch decision logic** in the main processor simply checks `result[0]`, making the design modular and reducing control complexity.
Additionally:

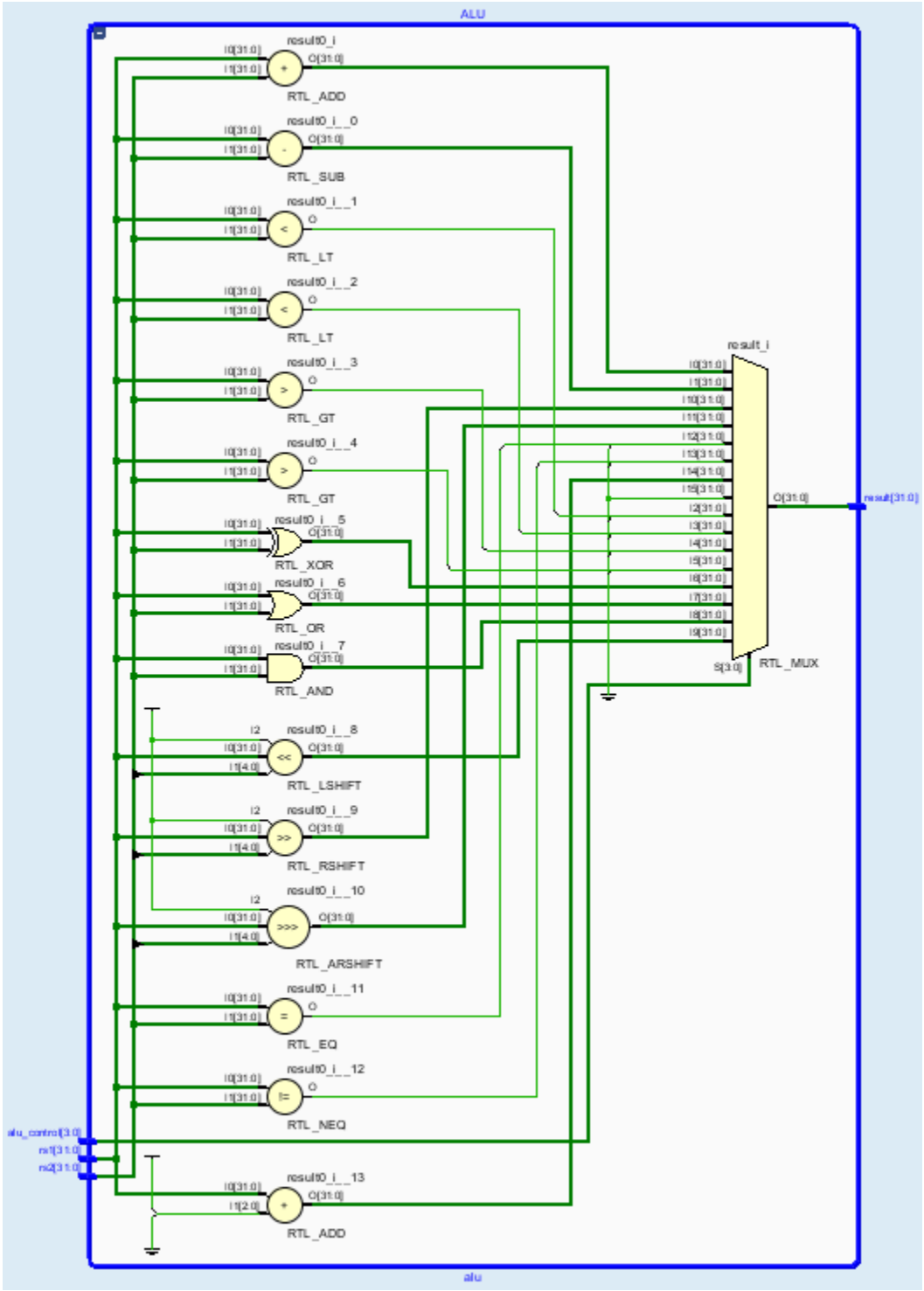
- Shift amounts use only `rs2[4:0]`, respecting RISC-V's 5-bit shift limit.
- For `pc_plus_4`, the ALU adds 4 to the current PC value to support JAL/JALR link register updates.
- Default output is zero for unsupported operations, ensuring safe operation.

This ALU design keeps computations clean, supports all RV32I requirements, and integrates neatly with the branch and jump hardware.

Verilog Code:

```
`include "RISCV_PKG.vh"
module alu(
    input [`INSTRUCTION_SIZE-1:0] rs1,
    input [`INSTRUCTION_SIZE-1:0] rs2,
    input [3:0] alu_control,
    output reg [`INSTRUCTION_SIZE-1:0] result
);
always @(*) begin
    case (alu_control)
        `ADD:                   result = rs1 + rs2;
        `SUB:                   result = rs1 - rs2;
        `less_than:             result = {31'b0, ($signed(rs1) < $signed(rs2))};
        `less_than_unsigned:    result = {31'b0, (rs1 < rs2)};
        `greater_than:          result = {31'b0, ($signed(rs1) > $signed(rs2))};
        `greater_than_unsigned: result = {31'b0, (rs1 > rs2)};
        `XOR:                   result = rs1 ^ rs2;
        `OR:                    result = rs1 | rs2;
        `AND:                   result = rs1 & rs2;
        `SLL:                   result = rs1 << rs2[4:0];
        `SRL:                   result = rs1 >> rs2[4:0];
        `SRA:                   result = $signed(rs1) >>> rs2[4:0];
        `equal:                 result = {31'b0, (rs1 == rs2)};
        `not_equal:             result = {31'b0, (rs1 != rs2)};
        `pc_plus_4:             result = rs1 + 4;   // rs1 holds the PC value
       default:                 result = 32'b0;
    endcase
end
endmodule
```

RTL Diagram:



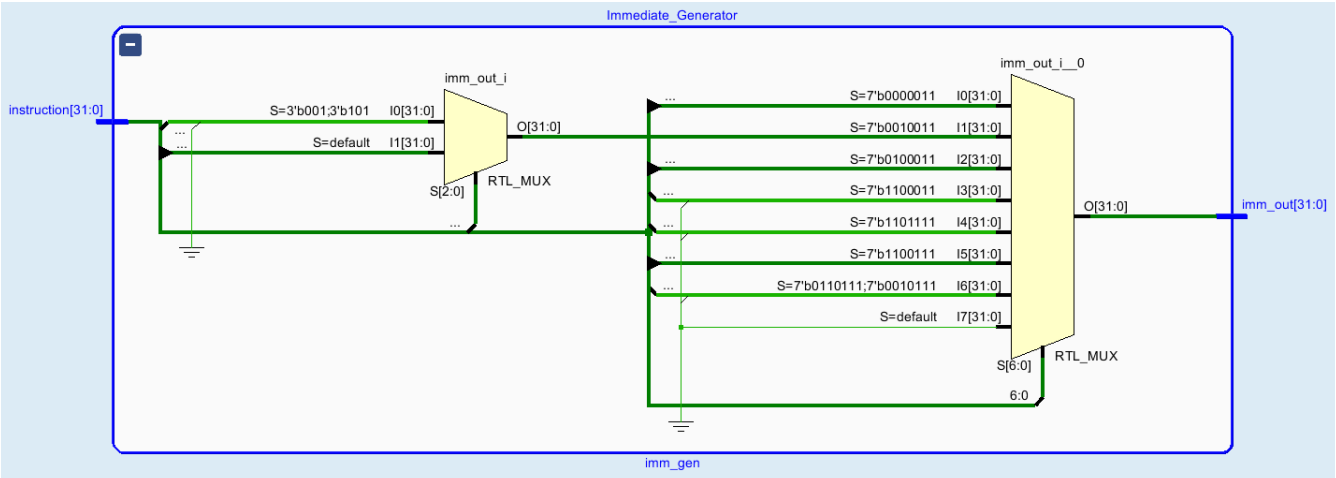
3.2.7 IMMEDIATE GENERATOR

Verilog Code:

```
`include "RISCV_PKG.vh"
module imm_gen(
    input  [`INSTRUCTION_SIZE-1:0] instruction,
    output reg [`INSTRUCTION_SIZE-1:0] imm_out);
wire [6:0] opcode = instruction[6:0];
wire [2:0] funct3 = instruction[14:12];
always @(*) begin
    case (opcode)
```

```
        // I-type (Load): LB, LH, LW, LBU, LHU
        7'b0000011:
            imm_out = {{20{instruction[31]}}, instruction[31:20]};
        // I-type (ALU immediate): ADDI,SLTI,SLTIU,XORI,ORI,ANDI,SLLI,SRLI,SRAI
        7'b0010011: begin
        // Shift instructions use shamt (bits [24:20])
            case (funct3)
              3'b001, 3'b101:imm_out = {{27{1'b0}}, instruction[24:20]}; // SLLI,SRLI,SRAI
                default:imm_out = {{20{instruction[31]}}, instruction[31:20]};
            endcase
        end
        // S-type (Store): SB, SH, SW
        7'b0100011:
            imm_out = {{20{instruction[31]}}, instruction[31:25], instruction[11:7]};
        // B-type (Branch): BEQ, BNE, BLT, BGE, BLTU, BGEU
        7'b1100011:
            imm_out = {{19{instruction[31]}}, instruction[31], instruction[7],
                       instruction[30:25], instruction[11:8], 1'b0};
        // J-type (JAL)
        7'b1101111:
            imm_out = {{11{instruction[31]}}, instruction[31], instruction[19:12],
                       instruction[20], instruction[30:21], 1'b0};
        // I-type (JALR)
        7'b1100111:
            imm_out = {{20{instruction[31]}}, instruction[31:20]};
        // U-type (LUI, AUIPC)
        7'b0110111, // LUI
        7'b0010111: // AUIPC
            imm_out = {instruction[31:12], 12'b0};
        // Default
        default:
            imm_out = 32'b0;
    endcase
end
endmodule
```

RTL Diagram:



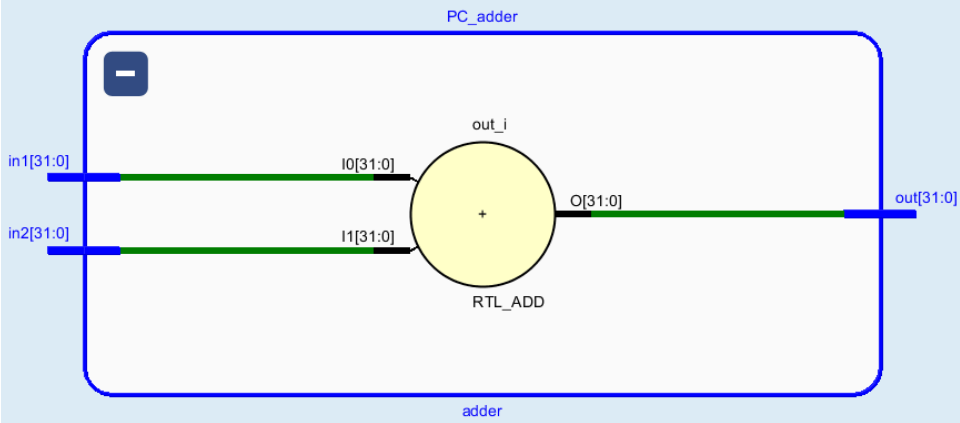
3.2.8 ADDER

Verilog Code:

```
`include "RISCV_PKG.vh"
module adder(
    input [`INSTRUCTION_SIZE-1:0] in1,
    input [`INSTRUCTION_SIZE-1:0] in2,
    output [`INSTRUCTION_SIZE-1:0] out
```

```
);
    assign out = in1 + in2;
endmodule
```

RTL Diagram:

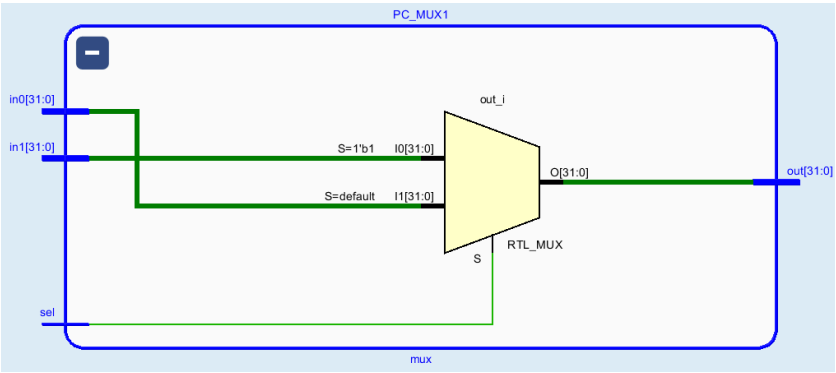


3.2.9 MULTIPLEXER

Verilog Code:

```
`include "RISCV_PKG.vh"
module mux(
    input sel,
    input [`INSTRUCTION_SIZE-1:0] in0,
    input [`INSTRUCTION_SIZE-1:0] in1,
    output [`INSTRUCTION_SIZE-1:0] out
);
    assign out = (sel) ? in1 : in0;
endmodule
```

RTL Diagram:



3.2.10 FINAL SYNTHESIS REPORT

```
Report Check Netlist:
+------+-----------------+-------+---------+-------+-----------------+
|      |Item             |Errors |Warnings |Status |Description      |
+------+-----------------+-------+---------+-------+-----------------+
|1     |multi_driven_nets|      0|        0|Passed |Multi driven nets|
+------+-----------------+-------+---------+-------+-----------------+
-------------------------------------------------------------------------------
Start Handling Custom Attributes
-------------------------------------------------------------------------------
```

```
---------------------------------------------------------------------------------
Finished Handling Custom Attributes : Time (s): cpu = 00:00:04 ; elapsed = 00:00:04 . Memory
(MB): peak = 492.730 ; gain = 180.371
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
Finished RTL Optimization Phase 1 : Time (s): cpu = 00:00:04 ; elapsed = 00:00:04 . Memory
(MB): peak = 492.730 ; gain = 180.371
---------------------------------------------------------------------------------
Finished RTL Optimization Phase 2 : Time (s): cpu = 00:00:22 ; elapsed = 00:00:24 . Memory
(MB): peak = 856.297 ; gain = 543.938
---------------------------------------------------------------------------------

Report RTL Partitions:
+-+--------------+------------+----------+
| |RTL Partition |Replication |Instances |
+-+--------------+------------+----------+
+-+--------------+------------+----------+
---------------------------------------------------------------------------------
Start RTL Component Statistics
---------------------------------------------------------------------------------
Detailed RTL Component Info :
+---Adders :
         3 Input     32 Bit       Adders := 1
         2 Input     32 Bit       Adders := 2
         2 Input     29 Bit       Adders := 1
         2 Input      7 Bit       Adders := 3
+---XORs :
         2 Input     32 Bit         XORs := 1
+---Registers :
                     32 Bit    Registers := 33
                     29 Bit    Registers := 1
                      8 Bit    Registers := 128
                      1 Bit    Registers := 1
+---Muxes :
         2 Input     32 Bit        Muxes := 8
        16 Input     32 Bit        Muxes := 1
         8 Input     32 Bit        Muxes := 1
         2 Input     29 Bit        Muxes := 1
         2 Input      8 Bit        Muxes := 316
         4 Input      8 Bit        Muxes := 127
         3 Input      8 Bit        Muxes := 50
        16 Input      4 Bit        Muxes := 1
         8 Input      3 Bit        Muxes := 1
         2 Input      2 Bit        Muxes := 2
         3 Input      2 Bit        Muxes := 3
         8 Input      1 Bit        Muxes := 8
         2 Input      1 Bit        Muxes := 604
         4 Input      1 Bit        Muxes := 128
         3 Input      1 Bit        Muxes := 62
---------------------------------------------------------------------------------
Finished RTL Component Statistics
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
Start RTL Hierarchical Component Statistics
```

```
---------------------------------------------------------------------------------
Hierarchical RTL Component report
Module wrapper
Detailed RTL Component Info :
+---Adders :
         2 Input     29 Bit       Adders := 1
+---Registers :
                     29 Bit    Registers := 1
                      1 Bit    Registers := 1
+---Muxes :
         2 Input     29 Bit        Muxes := 1
         2 Input      1 Bit        Muxes := 1
Module InstructionMemory
Detailed RTL Component Info :
+---Adders :
         2 Input      7 Bit       Adders := 3
Module control_unit
Detailed RTL Component Info :
+---Muxes :
         8 Input      3 Bit        Muxes := 1
         2 Input      2 Bit        Muxes := 2
         8 Input      1 Bit        Muxes := 8
Module regfile
Detailed RTL Component Info :
+---Registers :
                     32 Bit    Registers := 32
+---Muxes :
         2 Input      1 Bit        Muxes := 32
Module mux
Detailed RTL Component Info :
+---Muxes :
         2 Input     32 Bit        Muxes := 1
Module alu_cu
Detailed RTL Component Info :
+---Muxes :
        16 Input      4 Bit        Muxes := 1
Module alu
Detailed RTL Component Info :
+---Adders :
         3 Input     32 Bit       Adders := 1
         2 Input     32 Bit       Adders := 1
+---XORs :
         2 Input     32 Bit         XORs := 1
+---Muxes :
        16 Input     32 Bit        Muxes := 1
Module datamemory
Detailed RTL Component Info :
+---Registers :
                      8 Bit    Registers := 128
+---Muxes :
         2 Input     32 Bit        Muxes := 1
         2 Input      8 Bit        Muxes := 316
         4 Input      8 Bit        Muxes := 127
         3 Input      8 Bit        Muxes := 50
```

```
         3 Input      2 Bit        Muxes := 3
         2 Input      1 Bit        Muxes := 571
         4 Input      1 Bit        Muxes := 128
         3 Input      1 Bit        Muxes := 62
Module imm_gen
Detailed RTL Component Info :
+---Muxes :
         2 Input     32 Bit        Muxes := 1
         8 Input     32 Bit        Muxes := 1
Module adder
Detailed RTL Component Info :
+---Adders :
         2 Input     32 Bit       Adders := 1
Module pc
Detailed RTL Component Info :
+---Registers :
                     32 Bit    Registers := 1
---------------------------------------------------------------------------------
Finished RTL Hierarchical Component Statistics
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
Start Part Resource Summary
---------------------------------------------------------------------------------
Part Resources:
DSPs: 240 (col length:80)
BRAMs: 270 (col length: RAMB18 80 RAMB36 40)
---------------------------------------------------------------------------------
Finished Part Resource Summary
---------------------------------------------------------------------------------
Start ROM, RAM, DSP and Shift Register Reporting
---------------------------------------------------------------------------------

ROM:
+------------------+------------+---------------+----------------+
|Module Name       | RTL Object | Depth x Width | Implemented As |
+------------------+------------+---------------+----------------+
|InstructionMemory | p_0_out    | 128x8         | LUT            |
|InstructionMemory | p_0_out    | 128x8         | LUT            |
|InstructionMemory | p_0_out    | 128x8         | LUT            |
|InstructionMemory | p_0_out    | 128x8         | LUT            |
|InstructionMemory | p_0_out    | 128x8         | LUT            |
|InstructionMemory | p_0_out    | 128x8         | LUT            |
|InstructionMemory | p_0_out    | 128x8         | LUT            |
|InstructionMemory | p_0_out    | 128x8         | LUT            |
+------------------+------------+---------------+----------------+

---------------------------------------------------------------------------------
Finished ROM, RAM, DSP and Shift Register Reporting
---------------------------------------------------------------------------------

Report RTL Partitions:
+-+--------------+------------+----------+
| |RTL Partition |Replication |Instances |
+-+--------------+------------+----------+
```

```
+-+--------------+------------+----------+
----------------------------------------------------------------------------------
Start Applying XDC Timing Constraints
----------------------------------------------------------------------------------
----------------------------------------------------------------------------------
Finished Applying XDC Timing Constraints : Time (s): cpu = 00:02:51 ; elapsed = 00:02:55 .
Memory (MB): peak = 959.945 ; gain = 647.586
----------------------------------------------------------------------------------
----------------------------------------------------------------------------------
Start Timing Optimization
----------------------------------------------------------------------------------
----------------------------------------------------------------------------------
Finished Timing Optimization : Time (s): cpu = 00:03:10 ; elapsed = 00:03:15 . Memory (MB):
peak = 959.945 ; gain = 647.586
----------------------------------------------------------------------------------

Report RTL Partitions:
+-+--------------+------------+----------+
| |RTL Partition |Replication |Instances |
+-+--------------+------------+----------+
+-+--------------+------------+----------+
----------------------------------------------------------------------------------
Start Technology Mapping
----------------------------------------------------------------------------------
----------------------------------------------------------------------------------
Finished Technology Mapping : Time (s): cpu = 00:03:21 ; elapsed = 00:03:26 . Memory (MB):
peak = 1057.508 ; gain = 745.148
----------------------------------------------------------------------------------

Report RTL Partitions:
+-+--------------+------------+----------+
| |RTL Partition |Replication |Instances |
+-+--------------+------------+----------+
+-+--------------+------------+----------+
----------------------------------------------------------------------------------
Start IO Insertion
----------------------------------------------------------------------------------
----------------------------------------------------------------------------------
Start Flattening Before IO Insertion
----------------------------------------------------------------------------------
----------------------------------------------------------------------------------
Finished Flattening Before IO Insertion
----------------------------------------------------------------------------------
----------------------------------------------------------------------------------
Start Final Netlist Cleanup
----------------------------------------------------------------------------------
----------------------------------------------------------------------------------
Finished Final Netlist Cleanup
----------------------------------------------------------------------------------
----------------------------------------------------------------------------------
Finished IO Insertion : Time (s): cpu = 00:03:23 ; elapsed = 00:03:27 . Memory (MB): peak =
1057.508 ; gain = 745.148
----------------------------------------------------------------------------------
```

```
Report Check Netlist:
+------+------------------+-------+---------+-------+------------------+
|      |Item              |Errors |Warnings |Status |Description       |
+------+------------------+-------+---------+-------+------------------+
|1     |multi_driven_nets |      0|        0|Passed |Multi driven nets |
+------+------------------+-------+---------+-------+------------------+
-------------------------------------------------------------------------------
Start Renaming Generated Instances
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
Finished Renaming Generated Instances : Time (s): cpu = 00:03:23 ; elapsed = 00:03:28 .
Memory (MB): peak = 1057.508 ; gain = 745.148
-------------------------------------------------------------------------------

Report RTL Partitions:
+-+--------------+------------+----------+
| |RTL Partition |Replication |Instances |
+-+--------------+------------+----------+
+-+--------------+------------+----------+
-------------------------------------------------------------------------------
Start Rebuilding User Hierarchy
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
Finished Rebuilding User Hierarchy : Time (s): cpu = 00:03:23 ; elapsed = 00:03:28 . Memory
(MB): peak = 1057.508 ; gain = 745.148
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
Start Renaming Generated Ports
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
Finished Renaming Generated Ports : Time (s): cpu = 00:03:24 ; elapsed = 00:03:28 . Memory
(MB): peak = 1057.508 ; gain = 745.148
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
Start Handling Custom Attributes
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
Finished Handling Custom Attributes : Time (s): cpu = 00:03:24 ; elapsed = 00:03:29 . Memory
(MB): peak = 1057.508 ; gain = 745.148
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
Start Renaming Generated Nets
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
Finished Renaming Generated Nets : Time (s): cpu = 00:03:24 ; elapsed = 00:03:29 . Memory
(MB): peak = 1057.508 ; gain = 745.148
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
Start Writing Synthesis Report
-------------------------------------------------------------------------------

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

```
+-+--------------+----------+
+-+--------------+----------+

Report Cell Usage:
+------+-------+------+
|      |Cell   |Count |
+------+-------+------+
|1     |BUFG   |     2|
|2     |CARRY4 |    53|
|3     |LUT1   |     3|
|4     |LUT2   |   188|
|5     |LUT3   |   540|
|6     |LUT4   |   740|
|7     |LUT5   |   296|
|8     |LUT6   |  3381|
|9     |MUXF7  |  1217|
|10    |MUXF8  |   242|
|11    |FDCE   |  2110|
|12    |IBUF   |     2|
|13    |OBUF   |    32|
+------+-------+------+

Report Instance Areas:
+------+----------------------+------------------+------+
|      |Instance              |Module            |Cells |
+------+----------------------+------------------+------+
|1     |top                   |                  |  8806|
|2     |  cpu_inst            |SCDP              |  8696|
|3     |    ALU               |alu               |    24|
|4     |    PC_adder          |adder             |     8|
|5     |    data_memory       |datamemory        |  2926|
|6     |    instruction_memory |InstructionMemory |    32|
|7     |    program_counter   |pc                |  2298|
|8     |    register_file     |regfile           |  3408|
+------+----------------------+------------------+------+
--------------------------------------------------------------------------------
Finished Writing Synthesis Report : Time (s): cpu = 00:03:24 ; elapsed = 00:03:29 . Memory
(MB): peak = 1057.508 ; gain = 745.148
--------------------------------------------------------------------------------
Synthesis finished with 0 errors, 0 critical warnings and 33 warnings.
```

3.3 INTEGRATION

3.3.1 SCDP TOP MODULE

The **Single-Cycle Datapath (SCDP) Top Module** integrates all major components of our RISC-V processor, including the Instruction Memory, Control Unit, Register File, ALU, Immediate Generator, Data Memory, and Program Counter, into one cohesive hardware system. This module serves as the central point where instruction flow, data flow, and control flow converge in a single cycle, enabling the processor to fetch, decode, execute, access memory, and write back results without any pipelining. To facilitate hardware-level debugging on the FPGA, the top module also exposes selected internal signals such as the lower bits of the ALU result, Data Memory output, Register File write-back value, and Program Counter. These signals are routed to the FPGA's

LEDs and seven-segment display, allowing real-time observation of processor activity and verification of correct execution.

For details about how these signals are mapped to pins, how the constraint file is configured, and how the seven-segment display is interfaced on the FPGA board, **see the FPGA Implementation Section**.

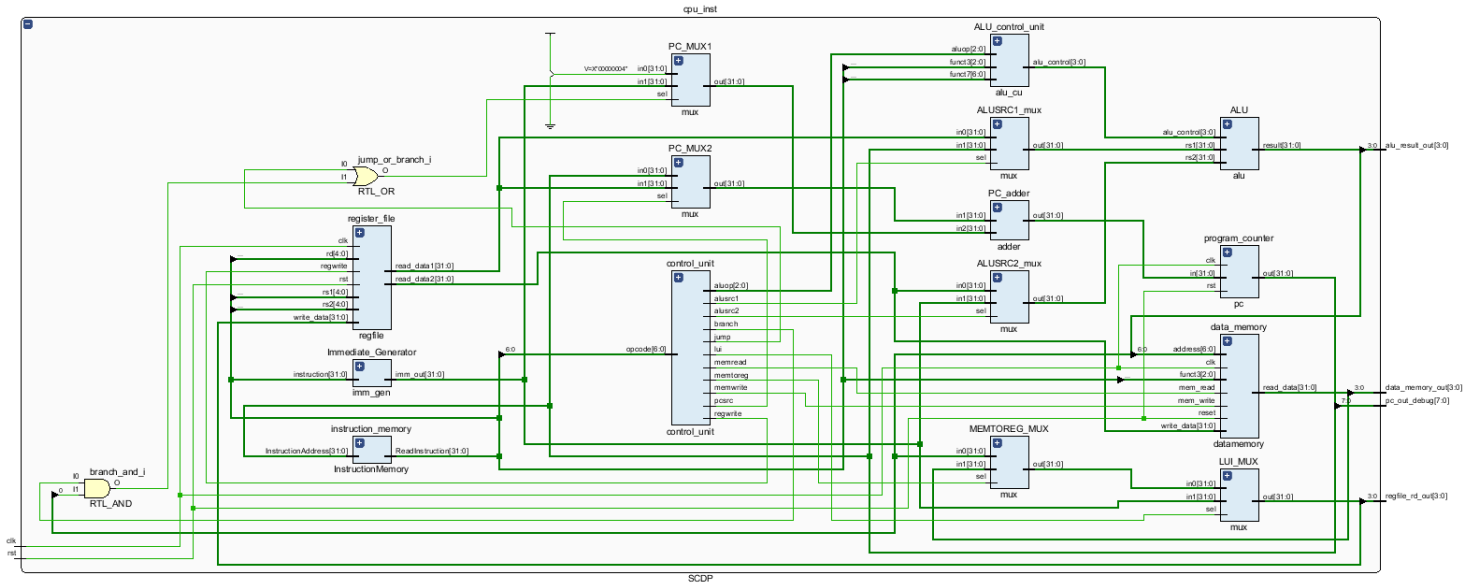
Verilog Code:

```
`include "RISCV_PKG.vh"
module SCDP(
    input clk, rst,
     // === Added Outputs for Observation ===
    output [3:0] alu_result_out,
    output [3:0] regfile_rd_out,
    output [3:0] data_memory_out,
    output [7:0] pc_out_debug
    );

     wire memread, memwrite, branch, jump, regwrite, pcsrc, alusrc1, alusrc2, lui, memtoreg,
b_or_j, branch_taken;
    wire [2:0] aluop;
    wire [3:0] alu_control;
      wire [`INSTRUCTION_SIZE-1:0] instruction, rs1_data, rs2_data, rd_data, pc_in, pc_out,
operand1,    operand2,    alu_result,    data_memory_output,    alu_or_data_out,    immediate,
adder_input_1, adder_input_2;

    InstructionMemory instruction_memory (
        .InstructionAddress(pc_out), // Connect to PC output
        .ReadInstruction(instruction) // Connect to instruction fetch logic
    );
    control_unit control_unit (
        .opcode(instruction[6:0]), // Connect to instruction opcode
        .aluop(aluop), // Connect to ALU control logic
        .memread(memread), // Connect to data memory read logic
        .memwrite(memwrite), // Connect to data memory write logic
        .branch(branch), // Connect to branch control logic
        .jump(jump), // Connect to jump control logic
        .regwrite(regwrite), // Connect to register file write logic
        .pcsrc(pcsrc), // Connect to PC source selection logic
        .alusrc1(alusrc1), // Connect to ALU source 1 selection logic
        .alusrc2(alusrc2), // Connect to ALU source 2 selection logic
        .lui(lui), // Connect to LUI control logic
        .memtoreg(memtoreg) // Connect to memory to register selection logic
    );

    regfile register_file (
        .clk(clk),
        .rst(rst),
        .rs1(instruction[19:15]),
        .rs2(instruction[24:20]),
        .rd(instruction[11:7]),
        .read_data1(rs1_data),
        .read_data2(rs2_data),
        .write_data(rd_data), // Connect to write back data logic
        .regwrite(regwrite) // Connect to control unit regwrite signal
    );

    mux ALUSRC1_mux (
        .sel(alusrc1),
        .in0(rs1_data), // Connect to register file read data 1
        .in1(pc_out), // Connect to PC value for JAL/JALR
        .out(operand1) // Connect to ALU input 1
```

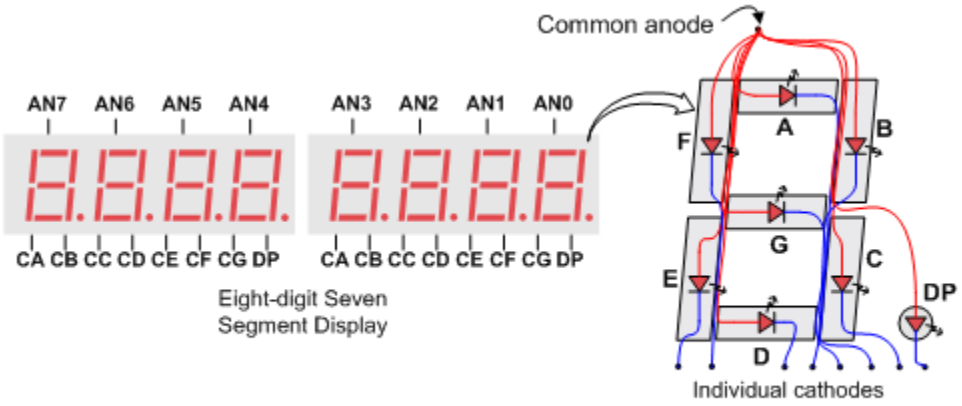
```
);

mux ALUSRC2_mux (
    .sel(alusrc2),
    .in0(rs2_data), // Connect to register file read data 2
    .in1(immediate), // Connect to immediate value from ImmGen
    .out(operand2) // Connect to ALU input 2
);

alu_cu ALU_control_unit (
    .aluop(aluop), // Connect to control unit aluop signal
    .funct3(instruction[14:12]), // Connect to instruction funct3 field
    .funct7(instruction[31:25]), // Connect to instruction funct7 field
    .alu_control(alu_control) // Connect to ALU control input
);

alu ALU (
    .rs1(operand1),
    .rs2(operand2),
    .alu_control(alu_control),
    .result(alu_result) // Connect to MemToReg mux
);

datamemory data_memory (
    .clk(clk),
    .reset(rst),
    .funct3(instruction[14:12]),
    .address(alu_result[$clog2(`MEM_SIZE)-1:0]), // Connect to ALU result
    .write_data(rs2_data), // Connect to register file read data 2
    .mem_read(memread), // Connect to control unit memread signal
    .mem_write(memwrite), // Connect to control unit memwrite signal
    .read_data(data_memory_output) // Connect to MemToReg mux
);

mux MEMTOREG_MUX (
    .sel(memtoreg),
    .in0(alu_result), // Connect to ALU result
    .in1(data_memory_output), // Connect to data memory output
    .out(alu_or_data_out) // Connect to register file write data
);

imm_gen Immediate_Generator (
    .instruction(instruction),
    .imm_out(immediate) // Connect to ALUSRC2 mux
);

mux LUI_MUX (
    .sel(lui),
    .in0(alu_or_data_out), // Connect to MemToReg mux output
    .in1(immediate), // Connect to immediate value from ImmGen
    .out(rd_data) // Connect to register file write data
);

and branch_and (
    branch_taken,
    branch,
    alu_result[0] // Assuming ALU result zero flag indicates branch condition
);

or jump_or_branch (
    b_or_j,
    jump,
```

```
        branch_taken
    );

    mux PC_MUX1 (
        .sel(b_or_j),
        .in0(32'd4), // Next sequential instruction
        .in1(immediate), //
        .out(adder_input_1) // Connect to PC input
    );

    mux PC_MUX2 (
        .sel(pcsrc),
        .in0(pc_out), // Current PC value
        .in1(rs1_data), // For JALR, use rs1_data as base
        .out(adder_input_2) // Connect to PC input
    );

    adder PC_adder (
        .in1(adder_input_2),
        .in2(adder_input_1),
        .out(pc_in) // Connect to PC input
    );
    pc program_counter (
        .clk(clk),
        .rst(rst),
        .in(pc_in), // Connect to next PC logic
        .out(pc_out) // Connect to instruction memory address
    );

    // === Assign Outputs for Observation ===
    assign alu_result_out = alu_result[3:0];
    assign regfile_rd_out = rd_data[3:0];
    assign data_memory_out = data_memory_output[3:0];
    assign pc_out_debug = pc_out[7:0];

endmodule
```

RTL Diagram:



3.3.2 HEX TO 7-SEGMENT CONVERTER



The **hex_to_7seg** module is responsible for converting a 4-bit hexadecimal value into the corresponding 7-segment display pattern. Since a seven-segment display requires control over each individual segment (a–g), the module uses a lookup table implemented through a `case` statement. For every valid 4-bit input (0–F), the module outputs an **8-bit pattern** that turns ON or OFF the appropriate segments to visually represent that hexadecimal digit. In our design, the **output is 8 bits wide**, where:

- **Bits [6:0]** correspond to the seven display segments **a, b, c, d, e, f, g**
- **Bit [7] (MSB)** is reserved for the **decimal point (DP)**

We are not using the decimal point in this project, so the MSB is always set to **1**, meaning the decimal point remains **off** for all characters. However, keeping it in the design allows future extensibility—if needed, the DP can be activated by setting the MSB to **0**.The module follows an **active-low configuration**, meaning:

- **0** → LED/segment ON
- **1** → LED/segment OFF

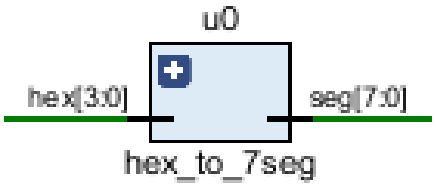
This is consistent with the Nexys A7 seven-segment display logic. Therefore, each 8-bit pattern in the case table has zeroes for segments that must turn ON to display the desired digit and ones for segments that should remain OFF.

Verilog Code:

```
// ==================== HEX TO 7-SEGMENT MODULE ====================
module hex_to_7seg(
    input [3:0] hex,
    output reg [7:0] seg // {a,b,c,d,e,f,g}
);
    always @(*) begin
        case(hex)
            4'h0: seg = 8'b10000001;
            4'h1: seg = 8'b11001111;
            4'h2: seg = 8'b10010010;
            4'h3: seg = 8'b10000110;
            4'h4: seg = 8'b11001100;
            4'h5: seg = 8'b10100100;
            4'h6: seg = 8'b10100000;
            4'h7: seg = 8'b10001111;
            4'h8: seg = 8'b10000000;
            4'h9: seg = 8'b10000100;
            4'hA: seg = 8'b10001000;
            4'hB: seg = 8'b11100000;
            4'hC: seg = 8'b10110001;
            4'hD: seg = 8'b11000010;
            4'hE: seg = 8'b10110000;
```

```
            4'hF: seg = 8'b10111000;
            default: seg = 8'b11111111; // all off
        endcase
    end
endmodule
```

RTL Diagram:



3.3.3 WRAPPER MODULE

The Wrapper Module serves as the top-level integration block that connects the internal SCDP (Single Cycle Datapath) processor to the external hardware peripherals on the FPGA board. Since the main SCDP module only implements the CPU logic, the wrapper is responsible for generating a human-observable output and providing a slower, more readable clock for debugging. This allows us to run the processor step-by-step and visualize its results on the 7-segment display.

The first section of the wrapper implements a clock divider, which takes the 100 MHz onboard clock and divides it down to a very slow clock (approximately 0.5 Hz). This slow clock is fed into the SCDP processor so that each instruction executes every few seconds, making it easier to observe changes in the register file, ALU result, memory output, and program counter during the FPGA demonstration.

We then instantiate the SCDP CPU and connect its outputs (ALU result, register file read data, memory data, and PC value) to debugging ports. These ports can later be used for additional display mechanisms or logic analyzers.

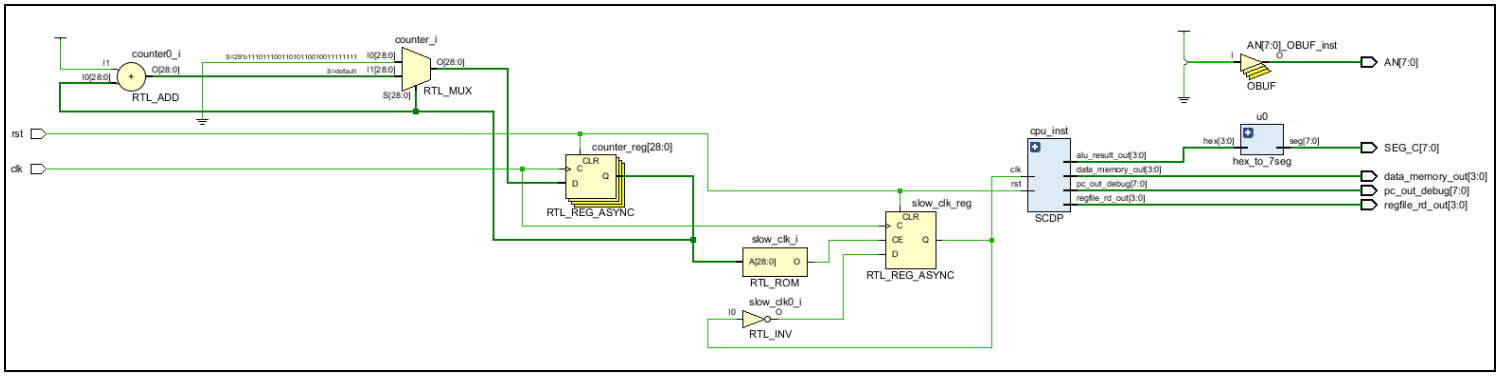
Finally, the wrapper connects the ALU output to the 7-segment display driver (hex_to_7seg). Here the AN line selects which digit of the 7-segment display is active, and SEG_C receives the corresponding segment pattern. In this basic setup, we only enable one digit (AN = 8'b11111110) and display the 4-bit ALU lower nibble in hexadecimal form.

Verilog Code:

```
`include "RISCV_PKG.vh"
// ==================== WRAPPER MODULE ====================
module wrapper (
    input clk,
    input rst,
    output [7:0] AN ,
    output [7:0] SEG_C,
    output [3:0] data_memory_out,
    output [3:0] regfile_rd_out,
    output [7:0] pc_out_debug
);
wire [3:0] alu_result_out;
    // ==================== CLOCK DIVIDER ====================
    parameter DIVIDE_BY = 500000000; // adjust to slow CPU clock (~0.5 Hz)
    reg [$clog2(DIVIDE_BY)-1:0] counter = 0;
    reg slow_clk = 0;
```

```
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            counter <= 0;
            slow_clk <= 0;
        end else begin
            if (counter == DIVIDE_BY-1) begin
                counter <= 0;
                slow_clk <= ~slow_clk;
            end else begin
                counter <= counter + 1;
            end
        end
    end
    // ==================== INSTANTIATE SCDP ====================
    SCDP cpu_inst (
        .clk(slow_clk),
        .rst(rst),
        .alu_result_out(alu_result_out),
        .regfile_rd_out(regfile_rd_out),
        .data_memory_out(data_memory_out),
        .pc_out_debug(pc_out_debug)
    );
    // ==================== CONNECT 7-SEGMENT DISPLAYS ====================
    assign AN = 8'b11111110;
    // ALU_RESULT
    hex_to_7seg u0(
    .hex(alu_result_out),
    .seg(SEG_C));
endmodule
```

RTL Diagram:



3.4 VERIFICATION & SIMULATION RESULTS

Verification is a critical phase in the design of any processor, ensuring that every module and the complete system behave exactly as intended. For this project, the verification strategy was divided into two major stages:

1. **Unit Testing (Module-Level Verification)**
   – Using *automated, task-based testbenches*
2. **Design Testing (Full Processor Verification)**
   – Using a *directed testbench*

This two-stage approach allowed me to first validate each individual block of the RV32I processor and then confirm that all modules worked correctly when integrated together.

3.4.1 UNIT TESTING VIA AUTOMATED (TASK-BASED) TESTBENCHES

WHAT IS AN AUTOMATED TESTBENCH?

An **automated testbench** is a self-checking testbench that automatically applies input stimulus, computes expected outputs, and compares them with the DUT (Design Under Test).
It **reduces manual effort**, **improves test coverage**, and **ensures consistency** during testing.

WHY TASK-BASED TESTBENCHES?

We used task-based testbenches to automate testing of individual modules (e.g., ALU, Register File, Control Unit, Immediate Generator, PC logic, etc.).
Using Verilog `task` blocks allowed me to:

- Apply multiple test vectors quickly
- Reuse the same task for different input combinations
- Automatically check expected outputs
- Generate neat and readable simulation logs

Instead of writing long manual testbenches, we created a **structured and automated test framework**, making module verification highly efficient and scalable. This significantly reduced debugging time.

3.4.1.1 INSTRUCTION MEMORY

Testbench:

```
`timescale 1ns/1ns
`include "RISCV_PKG.vh"
module InstructionMemory_tb;

    reg  [`INSTRUCTION_SIZE-1:0] InstructionAddress;
    wire [`INSTRUCTION_SIZE-1:0] ReadInstruction;
    integer pass_count;
    integer fail_count;
    integer i;

    // Byte-array to load MachineCode.mem inside TB
    reg [7:0] memfile_bytes [0:`MEM_SIZE-1];

    // Instantiate UUT
    InstructionMemory uut (
        .InstructionAddress(InstructionAddress),
        .ReadInstruction(ReadInstruction)
    );
    // Task: Compare Expected vs Actual
    task check_result;
        input [31:0] actual;
        input [31:0] expected;
        input integer addr;
        begin
            #1;
            if (actual === expected) begin
                $display("[PASS] PC=%0d | Expected=%h, Got=%h", addr, expected, actual);
                pass_count = pass_count + 1;
            end else begin
                $display("[FAIL] PC=%0d | Expected=%h, Got=%h", addr, expected, actual);
                fail_count = fail_count + 1;
            end
```

```
        end
    endtask

    // Build expected instruction word (little-endian) from memfile_bytes
    function [31:0] build_word_from_bytes;
        input integer addr;
        begin
            // little-endian: lowest-address holds LSB
                    build_word_from_bytes = { memfile_bytes[addr+3], memfile_bytes[addr+2],
memfile_bytes[addr+1], memfile_bytes[addr+0] };
        end
    endfunction

    initial begin
        $display("\n=== Instruction Memory TB ===\n");

        // Initialize counters
        pass_count = 0;
        fail_count = 0;

        // Read machine code file into testbench byte array
        $readmemh("MachineCode.mem", memfile_bytes);
        $display("Byte 0 = %h, Byte 1 = %h", memfile_bytes[0], memfile_bytes[1]);

        // Run through instructions (assume instructions are 4 bytes)
        for (i = 0; i <= `MEM_SIZE - 4; i = i + 4) begin
            InstructionAddress = i[(`INSTRUCTION_SIZE-1):0]; #2;
            // reconstruct expected instruction from bytes
            check_result(ReadInstruction, build_word_from_bytes(i), i);
        end

        // Summary
        #5;
        $display("\n=== TEST SUMMARY ===");
        $display("PASS Count = %0d", pass_count);
        $display("FAIL Count = %0d", fail_count);
        if (fail_count == 0)
            $display("ALL TESTS PASSED :)");
        else
            $display("SOME TESTS FAILED :(");

        $stop;
    end

endmodule
```

Output:

```
# === Instruction Memory TB ===
#
# Byte 0 = b7, Byte 1 = 00
# [PASS] PC=0 | Expected=000000b7, Got=000000b7
# [PASS] PC=4 | Expected=00000117, Got=00000117
# [PASS] PC=8 | Expected=00500193, Got=00500193
# [PASS] PC=12 | Expected=00a00213, Got=00a00213
# [PASS] PC=16 | Expected=0091b293, Got=0091b293
# [PASS] PC=20 | Expected=00324313, Got=00324313
# [PASS] PC=24 | Expected=005183b3, Got=005183b3
# [PASS] PC=28 | Expected=00627433, Got=00627433
# [PASS] PC=32 | Expected=0051e4b3, Got=0051e4b3
# [PASS] PC=36 | Expected=0041c533, Got=0041c533
# [PASS] PC=40 | Expected=00702023, Got=00702023
# [PASS] PC=44 | Expected=00800223, Got=00800223
# [PASS] PC=48 | Expected=00901423, Got=00901423
# [PASS] PC=52 | Expected=00002583, Got=00002583
# [PASS] PC=56 | Expected=00404603, Got=00404603
# [PASS] PC=60 | Expected=00805683, Got=00805683
# [PASS] PC=64 | Expected=00358463, Got=00358463
# [PASS] PC=68 | Expected=00359263, Got=00359263
# [PASS] PC=72 | Expected=0041c263, Got=0041c263
# [PASS] PC=76 | Expected=00325263, Got=00325263
# [PASS] PC=80 | Expected=0041e263, Got=0041e263
# [PASS] PC=84 | Expected=00327263, Got=00327263
# [PASS] PC=88 | Expected=0080076f, Got=0080076f
# [PASS] PC=92 | Expected=00900793, Got=00900793
# [PASS] PC=96 | Expected=00170813, Got=00170813
# [PASS] PC=100 | Expected=00080067, Got=00080067
# [PASS] PC=104 | Expected=0000006f, Got=0000006f
# [PASS] PC=108 | Expected=xxxxxxxx, Got=xxxxxxxx
# [PASS] PC=112 | Expected=xxxxxxxx, Got=xxxxxxxx
# [PASS] PC=116 | Expected=xxxxxxxx, Got=xxxxxxxx
# [PASS] PC=120 | Expected=xxxxxxxx, Got=xxxxxxxx
# [PASS] PC=124 | Expected=xxxxxxxx, Got=xxxxxxxx
#
# === TEST SUMMARY ===
# PASS Count = 32
# FAIL Count = 0
# ALL TESTS PASSED :)
```

3.4.1.2 REGISTER FILE

Testbench:

```
`timescale 1ns/1ns
`include "RISCV_PKG.vh"

module regfile_tb;
    // DUT Signals
    reg clk, rst, regwrite;
    reg [$clog2(`REG_COUNT)-1:0] rs1, rs2, rd;
    reg [`INSTRUCTION_SIZE-1:0] write_data;
    wire [`INSTRUCTION_SIZE-1:0] read_data1, read_data2;

    integer pass_count = 0, fail_count = 0;

    // Instantiate DUT
    regfile uut (
        .clk(clk),
        .rst(rst),
        .regwrite(regwrite),
```

```
    .rs1(rs1),
    .rs2(rs2),
    .rd(rd),
    .write_data(write_data),
    .read_data1(read_data1),
    .read_data2(read_data2)
);

// Clock Generation
initial clk = 0;
always #5 clk = ~clk; // 10ns period

// Task: Compare Expected vs Actual
task check_result;
    input [31:0] actual;
    input [31:0] expected;
    input [127:0] msg;
    begin
        #1;
        if (actual === expected) begin
            $display("[PASS] %s | Expected = %h, Got = %h", msg, expected, actual);
            pass_count = pass_count + 1;
        end else begin
            $display("[FAIL] %s | Expected = %h, Got = %h", msg, expected, actual);
            fail_count = fail_count + 1;
        end
    end
endtask

// Main Test Procedure
initial begin
    $display("\n=== Starting Register File Testbench ===");

    // 1. Reset
    rst = 1; regwrite = 0;
    rs1 = 0; rs2 = 0; rd = 0; write_data = 0;
    #10 rst = 0;
    $display("[INFO] Reset complete, all registers = 0");

    // 2. Write to x5 = 0xA5A5A5A5
    rd = 5; write_data = 32'hA5A5A5A5; regwrite = 1;
    #10 regwrite = 0;

    // Read x5
    rs1 = 5; rs2 = 5;
    #2 check_result(read_data1, 32'hA5A5A5A5, "Write/Read x5");

    // 3. Write to x10 = 0x12345678
    rd = 10; write_data = 32'h12345678; regwrite = 1;
    #10 regwrite = 0;

    rs1 = 10;
    #2 check_result(read_data1, 32'h12345678, "Write/Read x10");

    // 4. Write to x15 = 0x87654321
    rd = 15; write_data = 32'h87654321; regwrite = 1;
    #10 regwrite = 0;

    rs1 = 15;
    #2 check_result(read_data1, 32'h87654321, "Write/Read x15");

    // 5. Try writing to x0 (should not change)
```

```
        rd = 0; write_data = 32'hFFFFFFFF; regwrite = 1;
        #10 regwrite = 0;

        rs1 = 0;
        #2 check_result(read_data1, 32'h00000000, "x0 remains 0 after write attempt");

        // 6. Read x5 and x10 together
        rs1 = 5; rs2 = 10;
        #2;
        check_result(read_data1, 32'hA5A5A5A5, "Parallel read rs1=x5");
        check_result(read_data2, 32'h12345678, "Parallel read rs2=x10");

        // Summary
        #10;
        $display("\n=== TEST SUMMARY ===");
        $display("PASS Count = %0d", pass_count);
        $display("FAIL Count = %0d", fail_count);

        if (fail_count == 0)
            $display("ALL TESTS PASSED SUCCESSFULLY :)");
        else
            $display("SOME TESTS FAILED :(");
        $stop;
    end

endmodule
```

Output:

```
=== Starting Register File Testbench ===
[INFO] Reset complete, all registers = 0
[PASS]    Write/Read x5 | Expected = a5a5a5a5, Got = a5a5a5a5
[PASS]   Write/Read x10 | Expected = 12345678, Got = 12345678
[PASS]   Write/Read x15 | Expected = 87654321, Got = 87654321
[PASS] er write attempt | Expected = 00000000, Got = 00000000
[PASS] llel read rs1=x5 | Expected = a5a5a5a5, Got = a5a5a5a5
[PASS] lel read rs2=x10 | Expected = 12345678, Got = 12345678

=== TEST SUMMARY ===
PASS Count = 6
FAIL Count = 0
ALL TESTS PASSED SUCCESSFULLY :)
```

3.4.1.3 DATA MEMORY

Testbench:

```
`timescale 1ns/1ns
`include "RISCV_PKG.vh"
module datamemory_tb;
    // Testbench signals
    reg clk, reset, mem_read, mem_write;
    reg [$clog2(`MEM_SIZE)-1:0] address;
    reg [`INSTRUCTION_SIZE-1:0] write_data;
    reg [2:0] funct3;
    wire [`INSTRUCTION_SIZE-1:0] read_data;

    integer pass_count = 0, fail_count = 0;

    // Instantiate the DUT
    datamemory uut (
        .clk(clk),
        .reset(reset),
        .mem_read(mem_read),
```

```
    .mem_write(mem_write),
    .address(address),
    .write_data(write_data),
    .funct3(funct3),
    .read_data(read_data)
);

// Clock generation
initial clk = 0;
always #5 clk = ~clk;

// Task to check correctness
task check_result;
    input [31:0] expected;
    input [127:0] msg;
    begin
        #1; // allow read_data to update
        if (read_data === expected) begin
            $display("[PASS] %s | Expected = %h, Got = %h", msg, expected, read_data);
            pass_count = pass_count + 1;
        end else begin
            $display("[FAIL] %s | Expected = %h, Got = %h", msg, expected, read_data);
            fail_count = fail_count + 1;
        end
    end
endtask

// Test procedure
initial begin
    $display("\n=== Starting Data Memory Testbench ===");

    // Initialize
    reset = 1; mem_read = 0; mem_write = 0;
    address = 0; write_data = 0; funct3 = 0;
    #10 reset = 0;

    // 1. Test SW + LW
    address = 8'h00; write_data = 32'hAABBCCDD; funct3 = 3'b010;
    mem_write = 1; #10 mem_write = 0;

    // Read back LW
    mem_read = 1; funct3 = 3'b010;
    #2 check_result(32'hAABBCCDD, "LW after SW at addr 0x00");
    mem_read = 0;

    // 2. Test SH + LH
    address = 8'h10; write_data = 32'h0000BEEF; funct3 = 3'b001;
    mem_write = 1; #10 mem_write = 0;

    // Signed halfword read
    mem_read = 1; funct3 = 3'b001;
    #2 check_result(32'hFFFFBEEF, "LH after SH (sign-extended)");
    mem_read = 0;

    // 3. Test SH + LHU
    mem_read = 1; funct3 = 3'b101;
    #2 check_result(32'h0000BEEF, "LHU after SH (zero-extended)");
    mem_read = 0;

    // 4. Test SB + LB
    address = 8'h20; write_data = 32'h000000AA; funct3 = 3'b000;
    mem_write = 1; #10 mem_write = 0;
```

```
        // Signed byte load
        mem_read = 1; funct3 = 3'b000;
        #2 check_result(32'hFFFFFFAA, "LB after SB (sign-extended)");
        mem_read = 0;

        // 5. Test SB + LBU
        mem_read = 1; funct3 = 3'b100;
        #2 check_result(32'h000000AA, "LBU after SB (zero-extended)");
        mem_read = 0;

        // Summary
        #10;
        $display("\n=== TEST SUMMARY ===");
        $display("PASS Count = %0d", pass_count);
        $display("FAIL Count = %0d", fail_count);

        if (fail_count == 0)
            $display("ALL TESTS PASSED SUCCESSFULLY :) ");
        else
            $display("SOME TESTS FAILED :(");

        $stop;
    end
endmodule
```

Output:

```
# === Starting Data Memory Testbench ===
# [PASS]  SW at addr 0x00 | Expected = aabbccdd, Got = aabbccdd
# [PASS]  (sign-extended) | Expected = ffffbeef, Got = ffffbeef
# [PASS]  (zero-extended) | Expected = 0000beef, Got = 0000beef
# [PASS]  (sign-extended) | Expected = ffffffaa, Got = ffffffaa
# [PASS]  (zero-extended) | Expected = 000000aa, Got = 000000aa
#
# === TEST SUMMARY ===
# PASS Count = 5
# FAIL Count = 0
# ALL TESTS PASSED SUCCESSFULLY :)
```

3.4.1.4 CONTROL UNIT

Testbench:

```
`timescale 1ns / 1ps
`include "RISCV_PKG.vh"
module control_unit_tb;
    // Inputs
    reg [6:0] opcode;
    // Outputs
    wire regwrite, memread, memwrite, branch, jump, memtoreg, alusrc1, alusrc2, lui, pcsrc;
    wire [2:0] aluop;
    // Instantiate the DUT (Device Under Test)
    control_unit uut (
        .opcode(opcode),
        .regwrite(regwrite),
        .memread(memread),
        .memwrite(memwrite),
        .branch(branch),
        .jump(jump),
        .memtoreg(memtoreg),
        .alusrc1(alusrc1),
        .alusrc2(alusrc2),
        .lui(lui),
```

```
        .pcsrc(pcsrc),
        .aluop(aluop)
    );

    // Variables for test management
    integer total_tests = 0;
    integer failed_tests = 0;

    // Task to check and display results
    task check_output;
        input [6:0] opcode_t;
        input exp_regwrite, exp_memread, exp_memwrite, exp_branch, exp_jump,
              exp_memtoreg, exp_alusrc1, exp_alusrc2, exp_lui, exp_pcsrc;
        input [2:0] exp_aluop;
        begin
            total_tests = total_tests + 1;
            #5;
             if ({regwrite, memread, memwrite, branch, jump, memtoreg, alusrc1, alusrc2, lui,
pcsrc, aluop} !==
                 {exp_regwrite, exp_memread, exp_memwrite, exp_branch, exp_jump, exp_memtoreg,
exp_alusrc1, exp_alusrc2, exp_lui, exp_pcsrc, exp_aluop}) begin
                $display("Test %0d FAILED for opcode = %b", total_tests, opcode_t);
                failed_tests = failed_tests + 1;
            end else begin
                $display("Test %0d PASSED for opcode = %b", total_tests, opcode_t);
            end
        end
    endtask

    // Test procedure
    initial begin
        $display("========== Starting Control Unit Testbench ==========");

        // Default (NOP)
        opcode = 7'b0000000;
        check_output(opcode, 0,0,0,0,0,0,0,0,0,0, `NOP);

        // R-type
        opcode = 7'b0110011;
        check_output(opcode, 1,0,0,0,0,0,0,0,0,0, `R_TYPE);

        // I-type
        opcode = 7'b0010011;
        check_output(opcode, 1,0,0,0,0,0,0,1,0,0, `I_TYPE);

        // Load
        opcode = 7'b0000011;
        check_output(opcode, 1,1,0,0,0,1,0,1,0,0, `LOAD);

        // Store
        opcode = 7'b0100011;
        check_output(opcode, 0,0,1,0,0,0,0,1,0,0, `STORE);

        // Branch
        opcode = 7'b1100011;
        check_output(opcode, 0,0,0,1,0,0,0,0,0,0, `BRANCH);

        // JAL
        opcode = 7'b1101111;
        check_output(opcode, 1,0,0,0,1,0,1,0,0,0, `JUMP);

        // JALR
```

```
        opcode = 7'b1100111;
        check_output(opcode, 1,0,0,0,1,0,1,0,0,1, `JUMP);

        // LUI
        opcode = 7'b0110111;
        check_output(opcode, 1,0,0,0,0,0,0,1,1,0, `U_TYPE);

        // AUIPC
        opcode = 7'b0010111;
        check_output(opcode, 1,0,0,0,0,0,0,1,0,0, `U_TYPE);

        // Print Summary
        $display("=====================================================");
        $display("Total Tests Run: %0d", total_tests);
        $display("Tests Failed   : %0d", failed_tests);
        if (failed_tests == 0)
            $display("ALL TESTS PASSED SUCCESSFULLY!");
        else
            $display("Some tests FAILED. Please review outputs above.");
        $display("=====================================================");
        $finish;
    end
endmodule
```

Output:

```
# ========== Starting Control Unit Testbench ==========
# Test 1 PASSED for opcode = 0000000
# Test 2 PASSED for opcode = 0110011
# Test 3 PASSED for opcode = 0010011
# Test 4 PASSED for opcode = 0000011
# Test 5 PASSED for opcode = 0100011
# Test 6 PASSED for opcode = 1100011
# Test 7 PASSED for opcode = 1101111
# Test 8 PASSED for opcode = 1100111
# Test 9 PASSED for opcode = 0110111
# Test 10 PASSED for opcode = 0010111
# =====================================================
# Total Tests Run: 10
# Tests Failed   : 0
# ALL TESTS PASSED SUCCESSFULLY!
# =====================================================
```

3.4.1.5 ALU CONTROL UNIT

Testbench:

```
`timescale 1ns/1ns
`include "RISCV_PKG.vh"
module alu_cu_tb;
    // DUT signals
    reg  [2:0] aluop;
    reg  [2:0] funct3;
    reg  [6:0] funct7;
    wire [3:0] alu_control;
    integer errors = 0;
    integer tests  = 0;
    // instantiate DUT
    alu_cu uut (
        .aluop(aluop),
        .funct3(funct3),
        .funct7(funct7),
        .alu_control(alu_control)
    );
```

```
    // check task (PASS formatting improved)
    task check;
        input [2:0]  t_aluop;
        input [6:0]  t_funct7;
        input [2:0]  t_funct3;
        input [3:0]  expected;
        input        check_enable;
        input [80*8:1] name;
        begin
            tests = tests + 1;
            aluop  = t_aluop;
            funct7 = t_funct7;
            funct3 = t_funct3;
            #1;
            if (check_enable) begin
                if (alu_control !== expected) begin
                      $error("FAIL: %0s — aluop=%b funct7[5]=%b funct3=%b -> got %b expected
%b",
                          name, aluop, funct7[5], funct3, alu_control, expected);
                    errors = errors + 1;
                end else begin
                    // UPDATED: No extra spaces
                    $display("PASS: %0s", name);
                end
            end else begin
                $display("SKIP: %0s — ALU output don't-care (skipped check)", name);
            end
        end
    endtask
    initial begin
        $display("===== ALU_CU: Running full RV32I test vector (38 insns) =====");
        // ---------------- R-type (10)
        check(`R_TYPE, 7'b0000000, 3'b000, `ADD,  1, "ADD");
        check(`R_TYPE, 7'b0100000, 3'b000, `SUB,  1, "SUB");
        check(`R_TYPE, 7'b0000000, 3'b001, `SLL,  1, "SLL");
        check(`R_TYPE, 7'b0000000, 3'b010, `less_than, 1, "SLT");
        check(`R_TYPE, 7'b0000000, 3'b011, `less_than_unsigned, 1, "SLTU");
        check(`R_TYPE, 7'b0000000, 3'b100, `XOR,  1, "XOR");
        check(`R_TYPE, 7'b0000000, 3'b101, `SRL,  1, "SRL");
        check(`R_TYPE, 7'b0100000, 3'b101, `SRA,  1, "SRA");
        check(`R_TYPE, 7'b0000000, 3'b110, `OR,   1, "OR");
        check(`R_TYPE, 7'b0000000, 3'b111, `AND,  1, "AND");

        // ---------------- I-type arithmetic (9)
        check(`I_TYPE, 7'b0000000, 3'b000, `ADD, 1, "ADDI");
        check(`I_TYPE, 7'b0000000, 3'b001, `SLL, 1, "SLLI");
        check(`I_TYPE, 7'b0000000, 3'b010, `less_than, 1, "SLTI");
        check(`I_TYPE, 7'b0000000, 3'b011, `less_than_unsigned, 1, "SLTIU");
        check(`I_TYPE, 7'b0000000, 3'b100, `XOR, 1, "XORI");
        check(`I_TYPE, 7'b0000000, 3'b101, `SRL, 1, "SRLI");
        check(`I_TYPE, 7'b0100000, 3'b101, `SRA, 1, "SRAI");
        check(`I_TYPE, 7'b0000000, 3'b110, `OR, 1, "ORI");
        check(`I_TYPE, 7'b0000000, 3'b111, `AND,1, "ANDI");

        // ---------------- Loads (5)
        check(`LOAD, 7'b0000000, 3'b000, `ADD, 1, "LB");
        check(`LOAD, 7'b0000000, 3'b001, `ADD, 1, "LH");
        check(`LOAD, 7'b0000000, 3'b010, `ADD, 1, "LW");
        check(`LOAD, 7'b0000000, 3'b100, `ADD, 1, "LBU");
        check(`LOAD, 7'b0000000, 3'b101, `ADD, 1, "LHU");

        // ---------------- Stores (3)
```

```
        check(`STORE, 7'b0000000, 3'b000, `ADD, 1, "SB");
        check(`STORE, 7'b0000000, 3'b001, `ADD, 1, "SH");
        check(`STORE, 7'b0000000, 3'b010, `ADD, 1, "SW");

        // ---------------- Branch (6)
        check(`BRANCH, 7'b0000000, 3'b000, `equal, 1, "BEQ");
        check(`BRANCH, 7'b0000000, 3'b001, `not_equal, 1, "BNE");
        check(`BRANCH, 7'b0000000, 3'b100, `less_than, 1, "BLT");
        check(`BRANCH, 7'b0000000, 3'b101, `greater_than, 1, "BGE");
        check(`BRANCH, 7'b0000000, 3'b110, `less_than_unsigned, 1, "BLTU");
        check(`BRANCH, 7'b0000000, 3'b111, `greater_than_unsigned, 1, "BGEU");

        // ---------------- U-type
        check(`U_TYPE, 7'b0000000, 3'b000, 4'b0000, 0, "LUI (don't-care)");
        check(`U_TYPE, 7'b0000000, 3'b000, `ADD, 1, "AUIPC");

        // ---------------- Jumps
        check(`JUMP, 7'b0000000, 3'b000, `pc_plus_4, 1, "JAL");
        check(`JUMP, 7'b0000000, 3'b000, `pc_plus_4, 1, "JALR");

        // ---------------- NOP
        check(`NOP, 7'b0000000, 3'b000, `ADD, 1, "NOP");

        // Summary
        #1;
        if (errors == 0)
                $display("Total tests run : %0d, Total failures   : %0d ALL TESTS PASSED
ALHAMDULILLAH!", tests, errors);
        else
             $display("Total tests run : %0d, Total failures  : %0d SOME TESTS FAILED",tests,
errors);
        $stop;
    end
endmodule
```

Output:

```
# ===== ALU_CU: Running full RV32I test vector (38 insns) =====
# PASS: ADD
# PASS: SUB
# PASS: SLL
# PASS: SLT
# PASS: SLTU
# PASS: XOR
# PASS: SRL
# PASS: SRA
# PASS: OR
# PASS: AND
# PASS: ADDI
# PASS: SLLI
# PASS: SLTI
# PASS: SLTIU
# PASS: XORI
# PASS: SRLI
# PASS: SRAI
# PASS: ORI
# PASS: ANDI
# PASS: LB
# PASS: LH
# PASS: LW
# PASS: LBU
# PASS: LHU
# PASS: SB
# PASS: SH
# PASS: SW
# PASS: BEQ
# PASS: BNE
# PASS: BLT
# PASS: BGE
# PASS: BLTU
# PASS: BGEU
# SKIP: LUI (don't-care) â ALU output don't-care (skipped check)
# PASS: AUIPC
# PASS: JAL
# PASS: JALR
# PASS: NOP
# Total tests run : 38, Total failures  : 0 ALL TESTS PASSED ALHAMDULILLAH!
```

3.4.2 DESIGN TESTING VIA DIRECTED TESTBENCH

WHAT IS A DIRECTED TESTBENCH?

A **directed testbench** is a testbench where **specific, predefined instructions** or input sequences are fed into the processor to check whether it performs the expected operations.

It is not fully automated; instead, it focuses on verifying correct system behavior for **known and controlled scenarios**.

WHY A DIRECTED TESTBENCH FOR PROCESSOR LEVEL VERIFICATION?

For the full RV32I single-cycle processor, a **directed testbench** was used because the complete design does not require repeatedly stimulating a single module with multiple input patterns (as done in automated, task-based unit testing). Instead, the processor must be verified **as a whole**, running a **specific sequence of instructions** stored in the Instruction Memory through `$readmemh`.

A directed testbench is ideal in this case because:

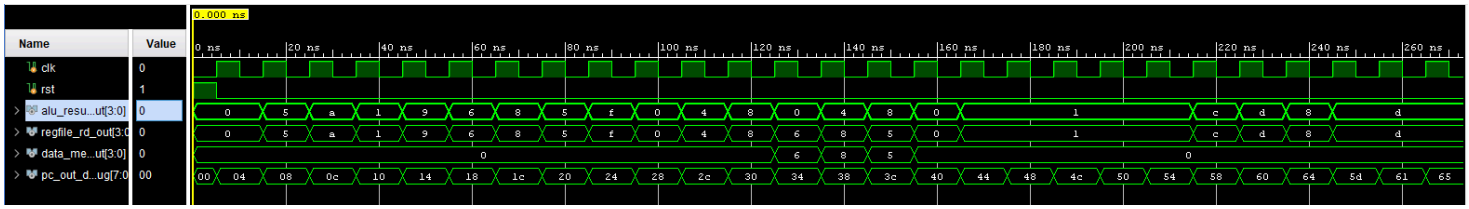
- **The processor executes a known, manually-chosen instruction sequence** (e.g., ADD, SUB, LW, SW, BEQ).
- These instructions are intentionally selected to exercise **all major components together**, the ALU, register file, data memory, PC update logic, control unit, etc.
- The goal is not random or repeated input generation, but to **observe how the complete datapath behaves cycle-by-cycle** when executing a real program.
- This allows you to verify whether internal components produce the correct values during actual instruction execution (e.g., register write-backs, PC updates, memory accesses).
- The waveform from this simulation can then be **directly compared with the FPGA output**, ensuring functional equivalence between simulation and hardware.

In short, **directed testing fits full-design verification** because the focus is on validating the processor's integrated behavior using a known program, rather than performing generalized or repetitive stimulus generation used in automated unit tests.

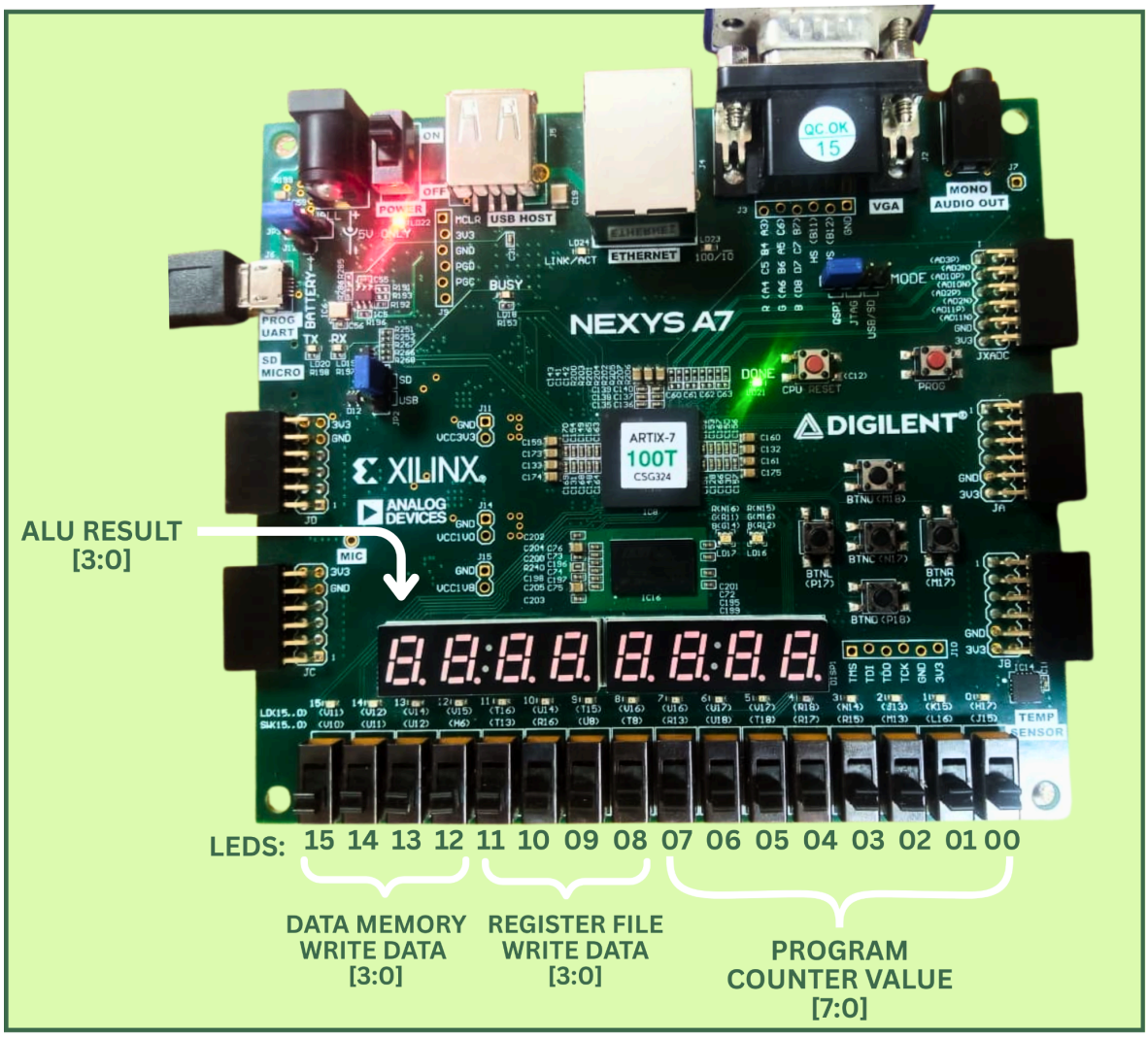
DIRECTED TESTBENCH FOR SCDP TOP MODULE

```
`timescale 1ns/1ns
`include "RISCV_PKG.vh"

module scdp_tb;

    // Clock and reset
    reg clk, rst;

    // Wires for observing outputs
    wire [3:0] alu_result_out;
    wire [3:0] regfile_rd_out;
    wire [3:0] data_memory_out;
    wire [7:0] pc_out_debug;

    // Instantiate the SCDP module
    SCDP dut (
        .clk(clk),
        .rst(rst),
        .alu_result_out(alu_result_out),
        .regfile_rd_out(regfile_rd_out),
        .data_memory_out(data_memory_out),
        .pc_out_debug(pc_out_debug)
    );

    // Clock generation (10ns period)
    always #5 clk = ~clk;

    // Reset sequence
    initial begin
        clk = 0;
        rst = 1;
        @(posedge clk);
        rst = 0;
    end

    // Run simulation for a fixed number of cycles
    initial begin
        #1000;
        $stop;
    end

endmodule
```

SIMULATION WAVEFORM



4. FPGA IMPLEMENTATION



In the FPGA implementation of my RV32I single-cycle processor, important internal signals were mapped to on-board LEDs and the seven-segment display to observe processor behavior in real time. The following mappings were done using the XDC constraint file:

LED MAPPINGS

- **PC Value (pc_out_debug[7:0]) → LEDs [7:0]**
  Displays the current Program Counter value on every clock cycle.

- **Register File Write Data (regfile_rd_out[3:0]) → LEDs [11:8]**
  Shows the data being written back into the register file.

- **Data Memory Write Data (data_memory_out[3:0]) → LEDs [15:12]**
  Indicates data written to or read from the data memory during load/store instructions.

SEVEN SEGMENT DISPLAY MAPPING

- **ALU Result [3:0] → Seven-Segment Display (SEG_C + AN signals)**
  The lower 4 bits of the ALU result are sent to the 7-segment display (through a display-driver module) to visualize arithmetic/logic outcomes directly on hardware.

OTHERS

- **Clock Input (clk) → 100 MHz on-board clock**
  Mapped through the dedicated clock pin.

- **Reset Input (rst) → Button (BTNC)**
  Used to reset the processor on hardware.

4.1 CONSTRAINT FILE

```
## This file is a general .xdc for the Nexys A7-100T
## To use it in a project:
## - uncomment the lines corresponding to used pins
## - rename the used ports (in each line, after get_ports) according to the top level signal
names in the project

## Clock signal
set_property -dict { PACKAGE_PIN E3 IOSTANDARD LVCMOS33 } [get_ports { clk }];
#IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clk}];

## LEDs
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { pc_out_debug[0] }];
#IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { pc_out_debug[1] }];
#IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { pc_out_debug[2] }];
#IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports { pc_out_debug[3] }];
#IO_L8P_T1_D11_14 Sch=led[3]
set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { pc_out_debug[4] }];
#IO_L7P_T1_D09_14 Sch=led[4]
set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports { pc_out_debug[5] }];
#IO_L18N_T2_A11_D27_14 Sch=led[5]
set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports { pc_out_debug[6] }];
#IO_L17P_T2_A14_D30_14 Sch=led[6]
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports { pc_out_debug[7] }];
#IO_L18P_T2_A12_D28_14 Sch=led[7]
```

```
set_property -dict { PACKAGE_PIN V16    IOSTANDARD LVCMOS33 } [get_ports { regfile_rd_out[0]
}]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
set_property -dict { PACKAGE_PIN T15    IOSTANDARD LVCMOS33 } [get_ports { regfile_rd_out[1]
}]; #IO_L14N_T2_SRCC_14 Sch=led[9]
set_property -dict { PACKAGE_PIN U14    IOSTANDARD LVCMOS33 } [get_ports { regfile_rd_out[2]
}]; #IO_L22P_T3_A05_D21_14 Sch=led[10]
set_property -dict { PACKAGE_PIN T16    IOSTANDARD LVCMOS33 } [get_ports { regfile_rd_out[3]
}]; #IO_L15N_T2_DQS_DOUT_CSO_B_14 Sch=led[11]

set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports { data_memory_out[0]
}]; #IO_L16P_T2_CSI_B_14 Sch=led[12]
set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports { data_memory_out[1]
}]; #IO_L22N_T3_A04_D20_14 Sch=led[13]
set_property -dict { PACKAGE_PIN V12   IOSTANDARD LVCMOS33 } [get_ports { data_memory_out[2]
}]; #IO_L20N_T3_A07_D23_14 Sch=led[14]
set_property -dict { PACKAGE_PIN V11   IOSTANDARD LVCMOS33 } [get_ports { data_memory_out[3]
}]; #IO_L21N_T3_DQS_A06_D22_14 Sch=led[15]

#7 segment display
set_property -dict { PACKAGE_PIN T10     IOSTANDARD LVCMOS33 } [get_ports { SEG_C[6]}];
#IO_L24N_T3_A00_D16_14 Sch=ca
set_property -dict { PACKAGE_PIN R10     IOSTANDARD LVCMOS33 } [get_ports { SEG_C[5] }];
#IO_25_14 Sch=cb
set_property -dict { PACKAGE_PIN K16     IOSTANDARD LVCMOS33 } [get_ports { SEG_C[4] }];
#IO_25_15 Sch=cc
set_property -dict { PACKAGE_PIN K13     IOSTANDARD LVCMOS33 } [get_ports { SEG_C[3] }];
#IO_L17P_T2_A26_15 Sch=cd
set_property -dict { PACKAGE_PIN P15     IOSTANDARD LVCMOS33 } [get_ports { SEG_C[2] }];
#IO_L13P_T2_MRCC_14 Sch=ce
set_property -dict { PACKAGE_PIN T11     IOSTANDARD LVCMOS33 } [get_ports { SEG_C[1] }];
#IO_L19P_T3_A10_D26_14 Sch=cf
set_property -dict { PACKAGE_PIN L18     IOSTANDARD LVCMOS33 } [get_ports { SEG_C[0] }];
#IO_L4P_T0_D04_14 Sch=cg
set_property -dict { PACKAGE_PIN H15     IOSTANDARD LVCMOS33 } [get_ports { SEG_C[7] }];
#IO_L19N_T3_A21_VREF_15 Sch=dp
set_property -dict { PACKAGE_PIN J17      IOSTANDARD LVCMOS33 } [get_ports { AN[7] }];
#IO_L23P_T3_FOE_B_15 Sch=an[0]
set_property -dict { PACKAGE_PIN J18      IOSTANDARD LVCMOS33 } [get_ports { AN[1] }];
#IO_L23N_T3_FWE_B_15 Sch=an[1]
set_property -dict { PACKAGE_PIN T9       IOSTANDARD LVCMOS33 } [get_ports { AN[2] }];
#IO_L24P_T3_A01_D17_14 Sch=an[2]
set_property -dict { PACKAGE_PIN J14      IOSTANDARD LVCMOS33 } [get_ports { AN[3] }];
#IO_L19P_T3_A22_15 Sch=an[3]
set_property -dict { PACKAGE_PIN P14      IOSTANDARD LVCMOS33 } [get_ports { AN[4] }];
#IO_L8N_T1_D12_14 Sch=an[4]
set_property -dict { PACKAGE_PIN T14      IOSTANDARD LVCMOS33 } [get_ports { AN[5] }];
#IO_L14P_T2_SRCC_14 Sch=an[5]
set_property -dict { PACKAGE_PIN K2       IOSTANDARD LVCMOS33 } [get_ports { AN[6] }];
#IO_L23P_T3_35 Sch=an[6]
set_property -dict { PACKAGE_PIN U13      IOSTANDARD LVCMOS33 } [get_ports { AN[0] }];
#IO_L23N_T3_A02_D18_14 Sch=an[7]

##CPU Reset Button
```

```
#set_property -dict { PACKAGE_PIN C12    IOSTANDARD LVCMOS33 } [get_ports { CPU_RESETN }];
#IO_L3P_T0_DQS_AD1P_15 Sch=cpu_resetn

##Buttons
set_property -dict { PACKAGE_PIN N17      IOSTANDARD LVCMOS33 } [get_ports { rst }];
#IO_L9P_T1_DQS_14 Sch=btnc
#set_property -dict { PACKAGE_PIN M18      IOSTANDARD LVCMOS33 } [get_ports { BTNU }];
#IO_L4N_T0_D05_14 Sch=btnu
#set_property -dict { PACKAGE_PIN P17      IOSTANDARD LVCMOS33 } [get_ports { BTNL }];
#IO_L12P_T1_MRCC_14 Sch=btnl
#set_property -dict { PACKAGE_PIN M17      IOSTANDARD LVCMOS33 } [get_ports { BTNR }];
#IO_L10N_T1_D15_14 Sch=btnr
#set_property -dict { PACKAGE_PIN P18      IOSTANDARD LVCMOS33 } [get_ports { BTND }];
#IO_L9N_T1_DQS_D13_14 Sch=btnd
```

4.2 DEVICE UTILIZATION REPORT

The Device Utilization Report shows how much of the FPGA's available hardware resources your design uses. It includes:

- Lookup Tables (LUTs)
- Flip-Flops (FFs)
- BRAM (Block RAM)
- DSP slices (if used)
- IO pins

Device Utilization Report

```
Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.
------------------------------------------------------------------------------------
| Tool Version : Vivado v.2018.2 (win64) Build 2258646 Thu Jun 14 20:03:12 MDT 2018
| Date         : Tue Nov 18 22:57:02 2025
| Host         : DESKTOP-QHTSA1Q running 64-bit major release  (build 9200)
|   Command                :   report_utilization   -file   wrapper_utilization_synth.rpt   -pb
wrapper_utilization_synth.pb
| Design       : wrapper
| Device       : 7a100tcsg324-1
| Design State : Synthesized
------------------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Slice Logic
1.1 Summary of Registers by Type
2. Memory
3. DSP
4. IO and GT Specific
5. Clocking
6. Specific Feature
7. Primitives
8. Black Boxes
9. Instantiated Netlists
```

1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	4646	0	63400	7.33
LUT as Logic	4646	0	63400	7.33
LUT as Memory	0	0	19000	0.00
Slice Registers	2110	0	126800	1.66
Register as Flip Flop	2110	0	126800	1.66
Register as Latch	0	0	126800	0.00
F7 Muxes	1217	0	31700	3.84
F8 Muxes	242	0	15850	1.53

* Warning! The Final LUT count, after physical optimizations and full implementation, is typically lower. Run opt_design after synthesis, if not already completed, for a more realistic count.

1.1 Summary of Registers by Type

Total	Clock Enable	Synchronous	Asynchronous
0	_	-	-
0	_	-	Set
0	_	-	Reset
0	_	Set	-
0	_	Reset	-
0	Yes	-	-
0	Yes	-	Set
2110	Yes	-	Reset
0	Yes	Set	-
0	Yes	Reset	-

2. Memory

Site Type	Used	Fixed	Available	Util%
Block RAM Tile	0	0	135	0.00
RAMB36/FIFO*	0	0	135	0.00
RAMB18	0	0	270	0.00

* Note: Each Block RAM Tile only has one FIFO logic available and therefore can accommodate only one FIFO36E1 or one FIFO18E1. However, if a FIFO18E1 occupies a Block RAM Tile, that tile can still accommodate a RAMB18E1

3. DSP

Site Type	Used	Fixed	Available	Util%
DSPs	0	0	240	0.00

4. IO and GT Specific

Site Type	Used	Fixed	Available	Util%
Bonded IOB	34	0	210	16.19
Bonded IPADs	0	0	2	0.00
PHY_CONTROL	0	0	6	0.00
PHASER_REF	0	0	6	0.00
OUT_FIFO	0	0	24	0.00
IN_FIFO	0	0	24	0.00
IDELAYCTRL	0	0	6	0.00
IBUFDS	0	0	202	0.00
PHASER_OUT/PHASER_OUT_PHY	0	0	24	0.00
PHASER_IN/PHASER_IN_PHY	0	0	24	0.00
IDELAYE2/IDELAYE2_FINEDELAY	0	0	300	0.00
ILOGIC	0	0	210	0.00
OLOGIC	0	0	210	0.00

5. Clocking

Site Type	Used	Fixed	Available	Util%
BUFGCTRL	2	0	32	6.25
BUFIO	0	0	24	0.00
MMCME2_ADV	0	0	6	0.00
PLLE2_ADV	0	0	6	0.00
BUFMRCE	0	0	12	0.00
BUFHCE	0	0	96	0.00
BUFR	0	0	24	0.00

6. Specific Feature

Site Type	Used	Fixed	Available	Util%

BSCANE2	0	0	4	0.00
CAPTUREE2	0	0	1	0.00
DNA_PORT	0	0	1	0.00
EFUSE_USR	0	0	1	0.00
FRAME_ECCE2	0	0	1	0.00
ICAPE2	0	0	2	0.00
PCIE_2_1	0	0	1	0.00
STARTUPE2	0	0	1	0.00
XADC	0	0	1	0.00

7. Primitives

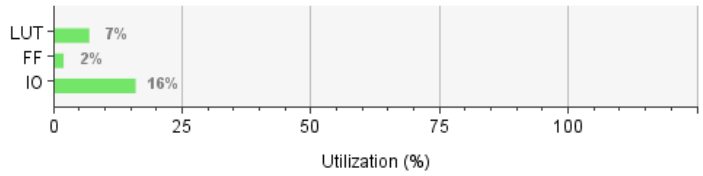
Ref Name	Used	Functional Category
LUT6	3381	LUT
FDCE	2110	Flop & Latch
MUXF7	1217	MuxFx
LUT4	740	LUT
LUT3	540	LUT
LUT5	296	LUT
MUXF8	242	MuxFx
LUT2	188	LUT
CARRY4	53	CarryLogic
OBUF	32	IO
LUT1	3	LUT
IBUF	2	IO
BUFG	2	Clock

Hierarchy

Name	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	Bonded IOB (210)	BUFGCTRL (32)
wrapper	4646	2110	1217	242	34	2
cpu_inst (SCDP)	4624	2080	1217	242	0	0
ALU (alu)	0	0	0	0	0	0
data_memory (data...	1168	1024	463	174	0	0
instruction_memo...	4	0	27	0	0	0
PC_adder (adder)	0	0	0	0	0	0
program_counter (pc)	1669	32	229	0	0	0
register_file (regfile)	1783	1024	498	68	0	0

Summary

Resource	Utilization	Available	Utilization %
LUT	4646	63400	7.33
FF	2110	126800	1.66
IO	34	210	16.19



Primitives

Ref Name	Used	Functional Category
LUT6	3381	LUT
FDCE	2110	Flop & Latch
MUXF7	1217	MuxFx
LUT4	740	LUT
LUT3	540	LUT
LUT5	296	LUT
MUXF8	242	MuxFx
LUT2	188	LUT
CARRY4	53	CarryLogic
OBUF	32	IO
LUT1	3	LUT
IBUF	2	IO
BUFG	2	Clock

5. RESULT ANALYSIS

The RV32I Single-Cycle Processor was successfully synthesized and implemented on the Nexys A7-100T FPGA. Both simulation and on-board testing confirmed that the design executed the loaded instructions correctly and produced accurate updates to the PC, register file, data memory, and ALU outputs.

5.1 SIMULATION RESULTS

- Directed testbench simulation validated end-to-end processor behavior by pre-loading instructions through the instruction memory using `$readmemh`.
- Cycle-accurate waveform inspection verified that each module, ALU, Register File, Control Unit, Data Memory, and PC update, functioned correctly and matched expected RV32I behavior.
- Intermediate values (ALU result, RegFile write data, Data memory output, PC) were cross-checked to ensure architectural correctness.

5.2 FPGA HARDWARE RESULTS

After synthesis, implementation, and bitstream generation, the design was tested on hardware.
The following mappings were used for real-time observation:

- ALU Result [3:0] → Seven-Segment Display
- PC Value [7:0] → LEDs [0–7]
- Register File Write Data [3:0] → LEDs [8–11]
- Data Memory Write Data [3:0] → LEDs [12–15]
- Reset Button → Push Button (btnC)

The observed LED and seven-segment outputs matched the simulation waveforms, confirming correct hardware execution of the RV32I single-cycle datapath.

5.3 HARDWARE DEMONSTRATION

A complete working demo of the processor running on the Nexys A7 FPGA is shown here:
🔗 Video Link: *Demo Video*

5.4 GITHUB REPOSITORY

Full Verilog source code, testbenches, constraint files, and documentation are available at:
🔗 https://github.com/nehanaumankhan/RV32I-Pipelined-Processor/tree/main/RV32I_SCDP

7. FUTURE EXTENSIONS

- Extend the design to a **complete 64-bit RISC-V Pipelined Processor** (RV64I).
- Develop it into a reusable **IP core** suitable for SoC integration.
- Implement **hazard detection**, **data forwarding**, and a **five-stage pipeline** (IF, ID, EX, MEM, WB).
- Integrate caches and memory management features required to run higher-level software.
- **Final goal:** *to run a lightweight Linux OS on the custom RISC-V core, InshaAllah.*

8. REFERENCES

1. *Computer Organization and Design: The RISC-V Edition—The Hardware/Software Interface by David A. Patterson and John L. Hennessy.*
   🔗 https://drive.google.com/drive/folders/1kF5sbXB8fc3VHs61HDQ3f9pgF6M3iD2C

2. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*
   University of California, Berkeley Tech Report:
   🔗 https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.pdf

9. BLOG LINKS

✍️ ARTICLE 1: RISC-V Single-Cycle Processor Datapath Design (SCDP) - *By Neha Nauman Khan*

✍️ ARTICLE 2: Designing the Control Unit and ALU Control Unit of RV32I - *By Neha Nauman Khan*

✍️ ARTICLE 3: Design and Synthesis of the RV32I Datapath in Verilog (Vivado) - *By Iqra Jawad Ahmed*

✍️ ARTICLE 4: Verifying the RV32I Processor: Directed & Automated Testbenches - *By Iqra Jawad Ahmed*

✍️ ARTICLE 5: RV32I Processor Integration: From Modules to Complete Top Design - *By Iqra Jawad Ahmed*

✍️ ARTICLE 6: FPGA Implementation of RV32I - *By Neha Nauman Khan*