CPU ARCHITECTURE (19 - Bit)

1) Choose an 19-bit architecture (Instruction size will be 19-bit):

1. Instruction Set Architecture (ISA)

Instruction Size: 19 bits

Instruction Format:

- **Opcode:** 5 bits
- **Register 1 (r1):** 4 bits
- **Register 2 (r2):** 4 bits
- **Register 3 (r3):** 4 bits
- **Address (addr):** 6 bits

2) Define a specialized instruction set tailored to a specific application or set of applications (e.g., signal processing, cryptography) :

Instruction Set :

> Arithmetic Instructions :

1. ADD r1, r2, r3 (Opcode: 00001)

   Operation: r1 = r2 + r3

2. SUB r1, r2, r3 (Opcode: 00010)

Operation: r1 = r2 - r3

3. MUL r1, r2, r3 (Opcode: 00011)

   Operation: r1 = r2 * r3

4. DIV r1, r2, r3 (Opcode: 00100)

   Operation: r1 = r2 / r3

5. INC r1 (Opcode: 00101)

   Operation: r1 = r1 + 1

6. DEC r1 (Opcode: 00110)

   Operation: r1 = r1 - 1

> Logical Instructions :

1. AND r1, r2, r3 (Opcode: 01001)

   Operation: r1 = r2 & r3

2. OR r1, r2, r3 (Opcode: 01010)

   Operation: r1 = r2 | r3

3. XOR r1, r2, r3 (Opcode: 01011)

   Operation: r1 = r2 ^ r3

4. NOT r1, r2 (Opcode: 01100)

   Operation: r1 = ~r2

> Control Flow Instructions :

1. JMP addr (Opcode: 10000)

   Operation: PC = addr

2. BEQ r1, r2, addr (Opcode: 10001)

Operation: if (r1 == r2) PC = addr

3. BNE r1, r2, addr (Opcode: 10010)

   Operation: if (r1 != r2) PC = addr

4. CALL addr (Opcode: 10011)

   Operation: stack[SP] = PC + 1; SP = SP - 1; PC = addr

5. RET (Opcode: 10100)

   Operation: SP = SP + 1; PC = stack[SP]

> Memory Access Instructions :

1. LD r1, addr (Opcode: 11000)

   Operation: r1 = memory[addr]

2. ST addr, r1 (Opcode: 11001)

   Operation: memory[addr] = r1

> Custom Instructions :

1. FFT r1, r2 (Opcode: 11100)

   Operation: FFT(memory[r2], result); memory[r1] = result

2. ENC r1, r2 (Opcode: 11101)

   Operation: encrypted_data = Encrypt(memory[r2]);
   memory[r1] = encrypted_data

3. DEC r1, r2 (Opcode: 11110)

   Operation: decrypted_data = Decrypt(memory[r2]);
   memory[r1] = decrypted_data

3) Create a detailed architectural specification, including pipeline stages, register file, ALU, and memory interfaces :

a. Pipeline Stages :

The CPU will use a 5-stage pipeline to balance performance and complexity:

1. **Fetch (IF)**: Fetch the 19-bit instruction from memory.
2. **Decode (ID)**: Decode the instruction and read registers.
3. **Execute (EX)**: Perform ALU operations or special function computations.
4. **Memory Access (MEM)**: Access data memory if required (for load/store instructions).
5. **Write Back (WB)**: Write the result back to the destination register.

b. Register File :

Register Count: 16 general-purpose registers, each 19 bits wide (R0 to R15).

Special Registers:

Program Counter (PC): 19-bit register to hold the address of the next instruction.

Status Register (SR): 19-bit status register to hold flags (e.g, zero flag, carry flag)

Stack Pointer (SP): 19-bit stack pointer for subroutine calls.

c. Arithmetic Logic Unit (ALU) :

Bit-width: 19-bit ALU to perform arithmetic and logical operations.

Operations Supported: Addition, subtraction, multiplication, bitwise AND/OR/XOR, shifts, specialized operations (e.g, XOR-ENC, FFT)

Multiplier: Optional hardware multiplier for efficient signal processing.

Cryptographic Unit: Dedicated logic for AES operations, XOR-based encryption.

d. Memory Interface :

Data Memory: Supports byte-addressable memory, with 19-bit addressing (524,288 unique addresses).

Instruction Memory: Separate instruction memory, also 19-bit addressing.

Bus Width: 19-bit data bus for communication between CPU and memory.

Memory Controller: Handles read/write operations, ensuring correct alignment for 19-bit word size.

4. Specialized Components :

a. Cryptography Engine:

AES Unit: A hardware block for performing AES encryption rounds, optimized for speed and security.

XOR Encryption Unit: Lightweight hardware block for XOR-based encryption, suitable for low-power applications.

b. Signal Processing Unit:

FFT Processor: Dedicated hardware for FFT operations, designed for efficient real-time signal analysis.

Convolution Processor: Custom hardware for performing convolutions, useful in filtering and other DSP tasks.

5. Example Instruction Execution :

Example Instruction: ADD R1, R2, R3

IF Stage: Fetch the instruction from memory.

ID Stage: Decode the instruction, read the values from registers R2 and R3.

EX Stage: Perform the addition of R2 and R3 in the ALU.

MEM Stage: No memory access required.

WB Stage: Write the result to register R1.

VERILOG CODE

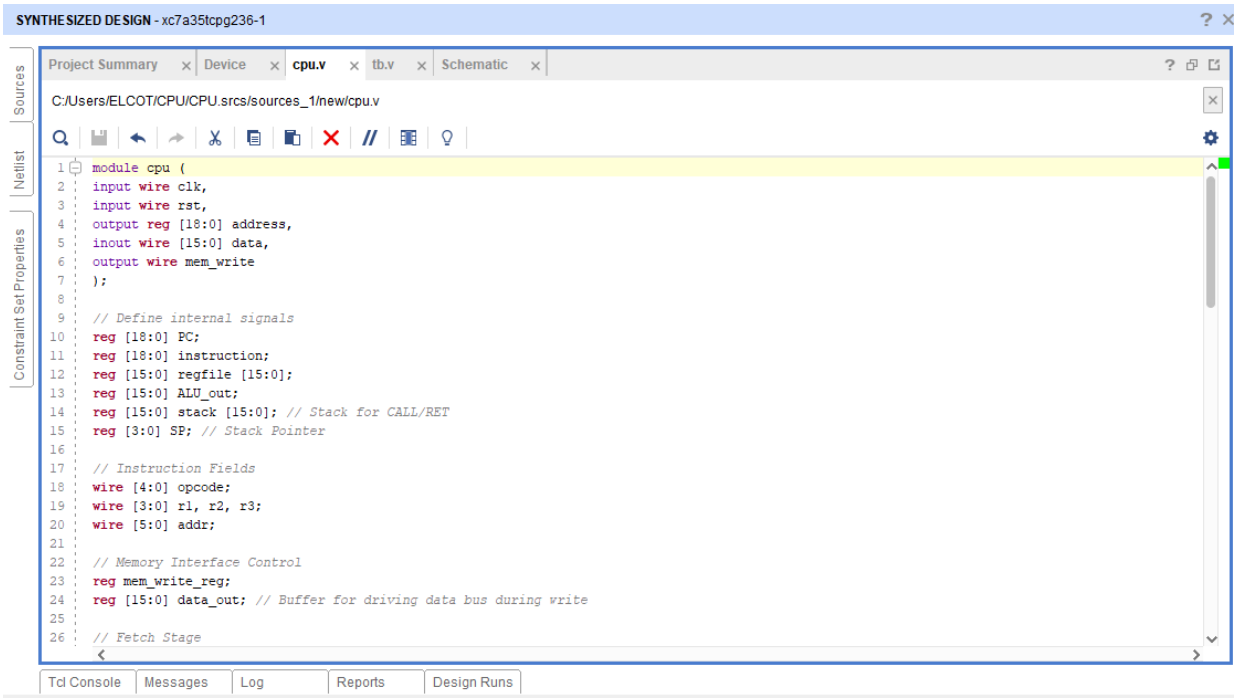
```
module cpu (
input wire clk,
input wire rst,
output reg [18:0] address,
inout wire [15:0] data,
output wire mem_write
);

// Define internal signals
reg [18:0] PC;
reg [18:0] instruction;
reg [15:0] regfile [15:0];
reg [15:0] ALU_out;
reg [15:0] stack [15:0]; // Stack for CALL/RET
reg [3:0] SP; // Stack Pointer

// Instruction Fields
wire [4:0] opcode;
wire [3:0] r1, r2, r3;
wire [5:0] addr;
```

```
// Memory Interface Control
reg mem_write_reg;
reg [15:0] data_out;

// Fetch Stage
always @(posedge clk or posedge rst) begin
  if (rst) begin
    PC <= 0;
    SP <= 0; // Initialize stack pointer
    mem_write_reg <= 1'b0;
    address <= 0;
  end else begin
  // Fetch instruction from memory
    address <= PC;
    instruction <= data;
    PC <= PC + 1;
  end
end


// Decode Stage
```

```
assign opcode = instruction[18:14];
assign r1 = instruction[13:10];
assign r2 = instruction[9:6];
assign r3 = instruction[5:2];
assign addr = instruction[5:0];

// Execute Stage
always @(*) begin
case (opcode)
5'b00001: ALU_out = regfile[r2] + regfile[r3]; // ADD
5'b00010: ALU_out = regfile[r2] - regfile[r3]; // SUB
5'b00011: ALU_out = regfile[r2] * regfile[r3]; // MUL
5'b00100: ALU_out = regfile[r2] / regfile[r3]; // DIV

// Add cases for other instructions as needed
default: ALU_out = 16'b0;
endcase
end

// Memory Access and Write-Back
always @(posedge clk) begin
case (opcode)
```

```
5'b11000: begin // LD
address <= addr;
mem_write_reg <= 1'b0;
regfile[r1] <= data;
case(opcode)
    5'b11001: begin // ST
        address <= addr;
        data_out <= regfile[r1];
        mem_write_reg <= 1'b1;
    end
    5'b10011: begin // CALL
        stack[SP] <= PC;
        SP <= SP + 1;
        PC <= addr;
    end
    5'b10100: begin // RET
        SP <= SP - 1;
        PC <= stack[SP];
    end
    default: begin
        regfile[r1] <= ALU_out;
        mem_write_reg <= 1'b0;
```

```
        end
    endcase
end
assign data = (mem_write_reg) ? data_out : 16'bz;
assign mem_write = mem_write_reg;
endmodule
```



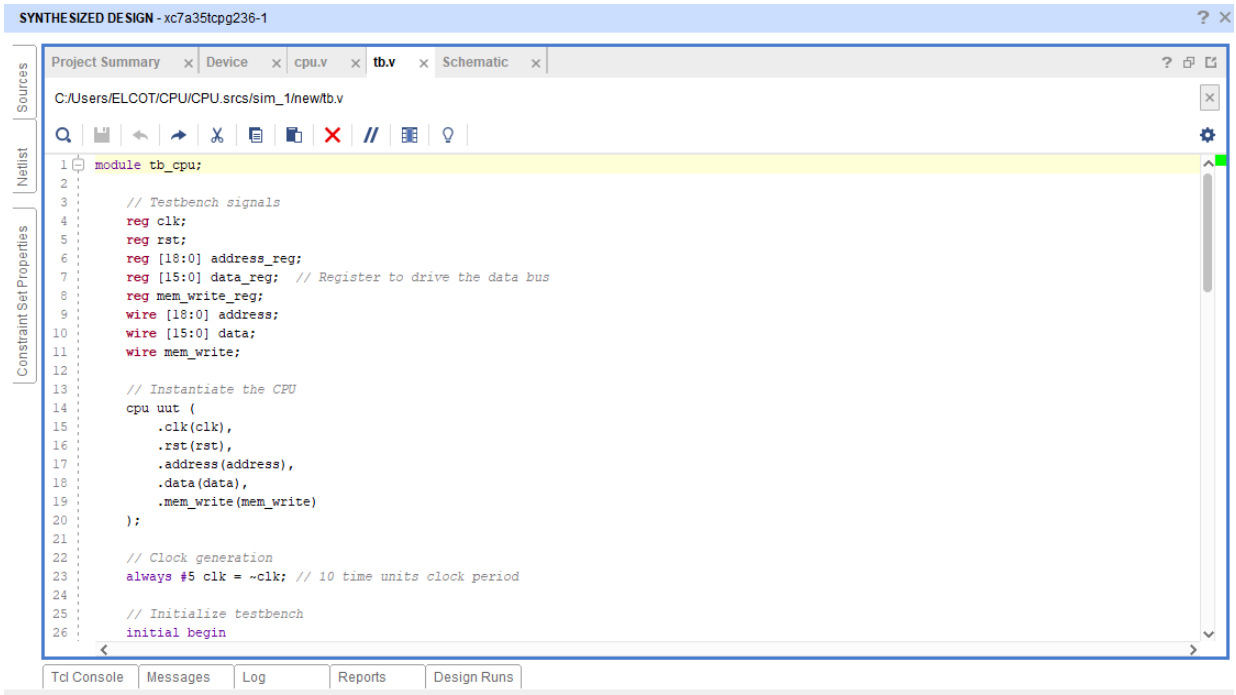
TESTBENCH CODE

```
module tb_cpu;
reg clk;
reg rst;
reg [18:0] address_reg;
reg [15:0] data_reg;
reg mem_write_reg;

wire [18:0] address;
wire [15:0] data;
wire mem_write;


// Instantiate the CPU module
cpu uut (
    .clk(clk),
    .rst(rst),
    .address(address),
    .data(data),
    .mem_write(mem_write)
```

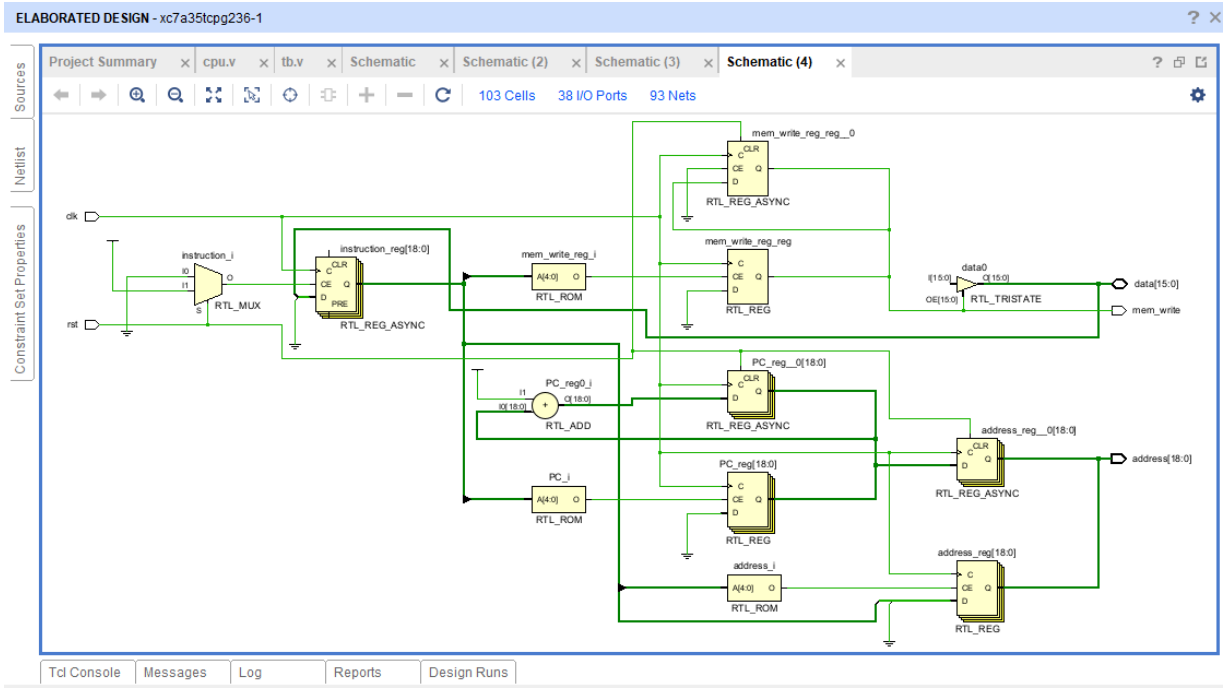
```
);

// Generate clock signal
always #5 clk = ~clk; // 10 units

initial begin
    clk = 0;
    rst = 0;
    address_reg = 19'b0;
    data_reg = 16'b0;
    mem_write_reg = 1'b0;

    rst = 1;
    #10;
    rst = 0;

    #20;
// Write test data to memory
write_memory(19'h00001, 16'h1234); // Example data
#10;

// Read data from memory to check
```

```
        read_memory(19'h00001, data_reg);
        #10;
        $display("Read Data = %h", data_reg);

        // Add more test cases as needed
       // Finish simulation
        #100;
        $finish;
        end

        // Monitor signals
        initial begin
        $monitor("At time %t, address = %h, data = %h, mem_write =
%b", $time, address, data, mem_write);
        end

        // Data bus driver
        assign data = (mem_write_reg) ? data_reg : 16'bz;

        // Assign outputs to internal regs for tasks
        assign address = address_reg;
        assign mem_write = mem_write_reg;
```

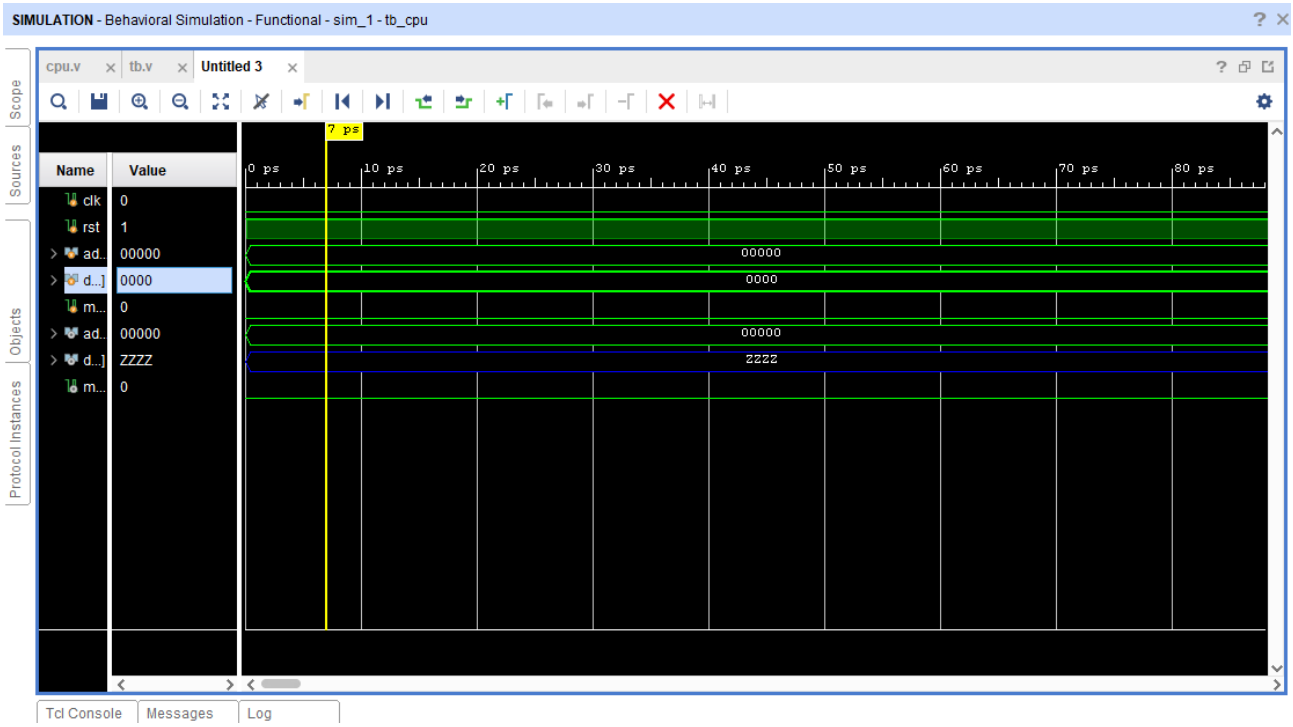
```
    // Procedure to store information in memory
    procedure store_memory_data(input [18:0] address, input [15:0] data);
    begin
       address_register <= address;
       data_register <= data;
       memory_write_enable <= 1;
       @(posedge clock);
       memory_write_enable <= 0;
    end
    endmodule

    // Procedure to retrieve data from memory
    procedure retrieve_memory_data(input [18:0] address, output [15:0] data);
    begin
       address_register <= address;
       @(posedge clock);
       data <= data_register;
    end
    endmodule.
```

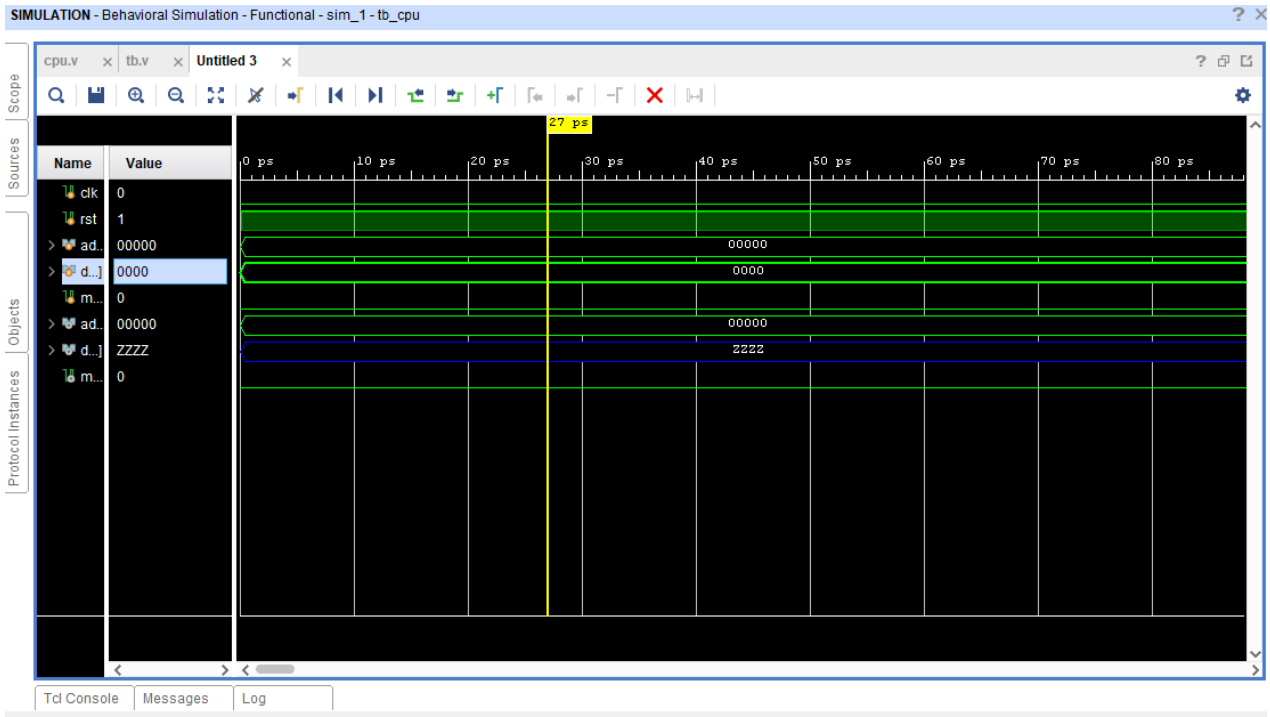


SCHEMATIC VIEW

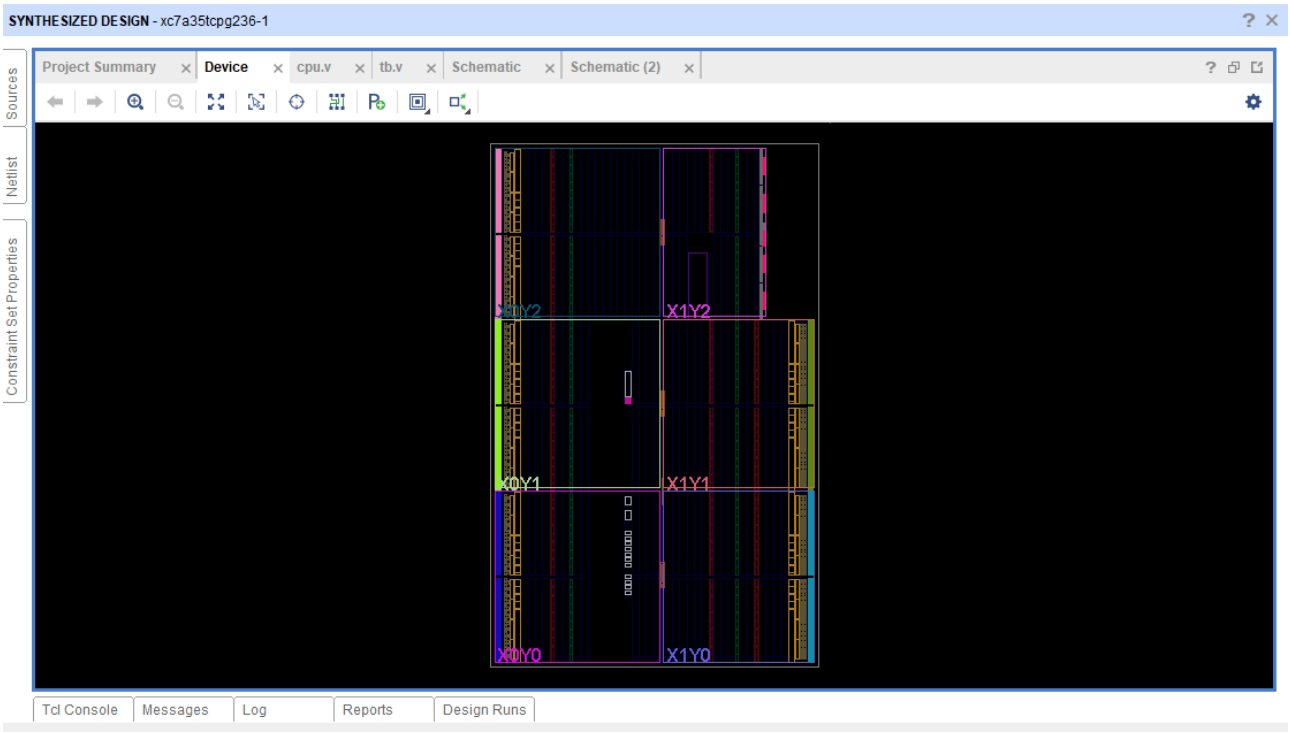


SIMULATION OUTPUT

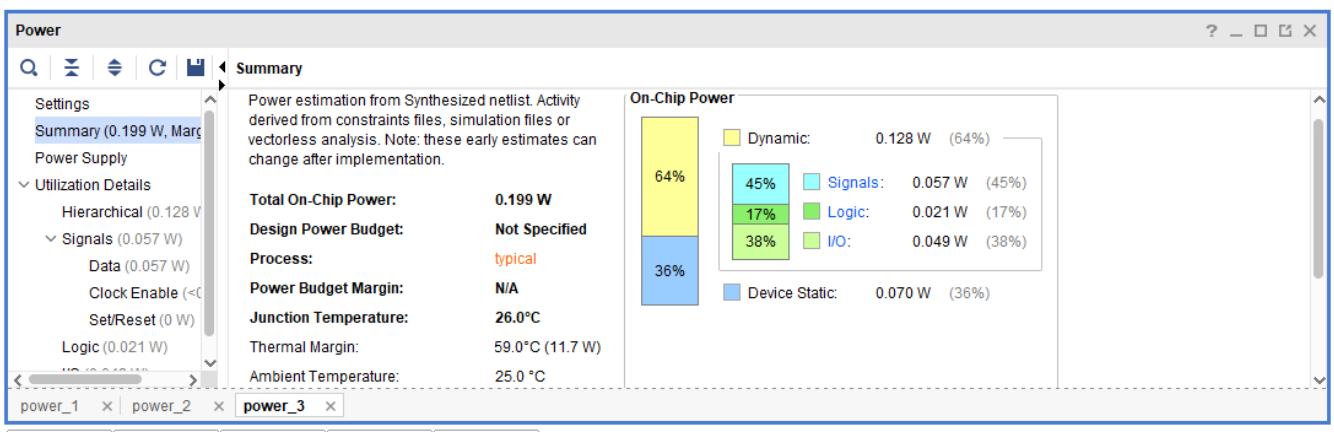




SYNTHESIS LAYOUT



TOTAL ON-CHIP POWER



THANK YOU!