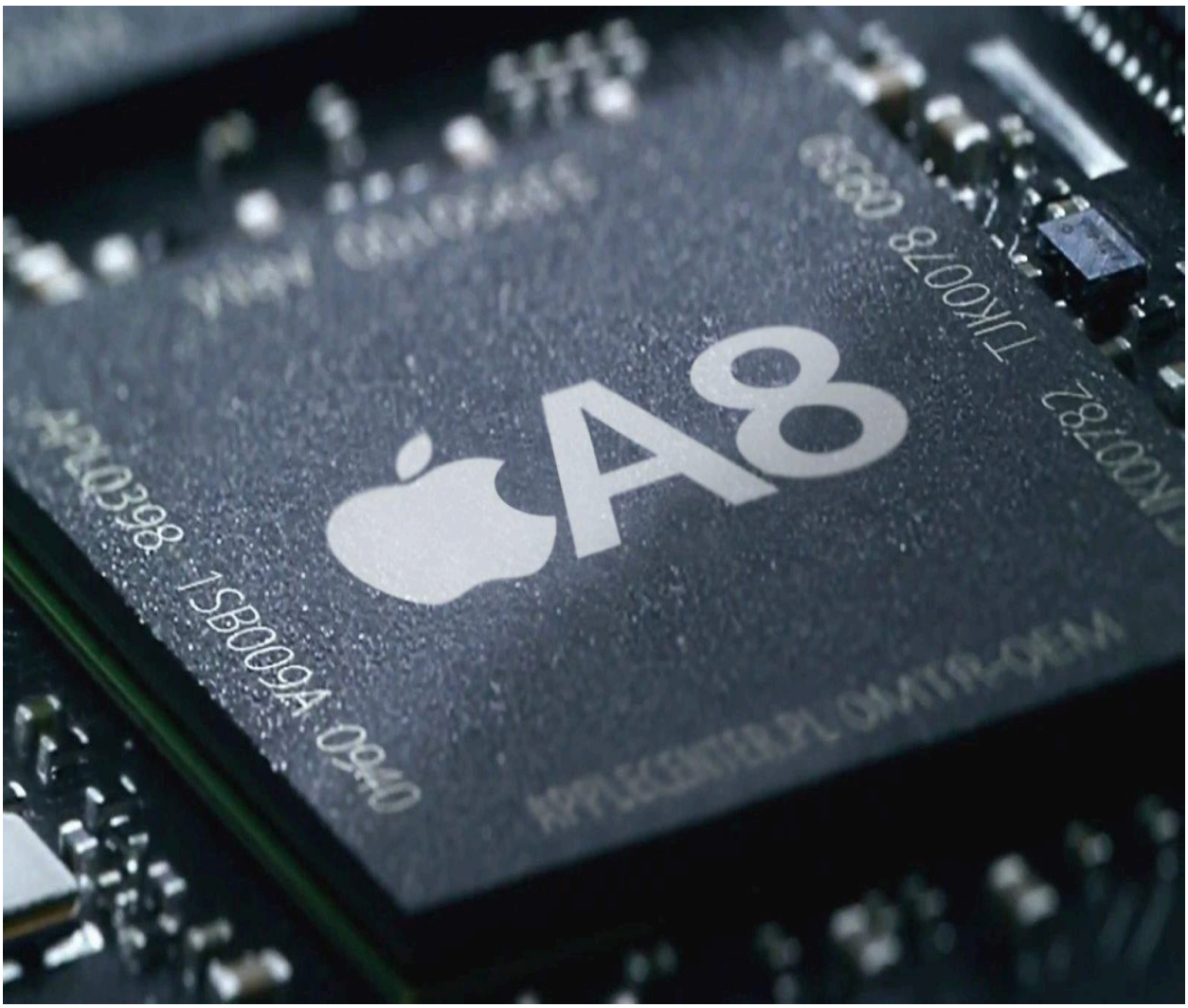
PORTFOLIO

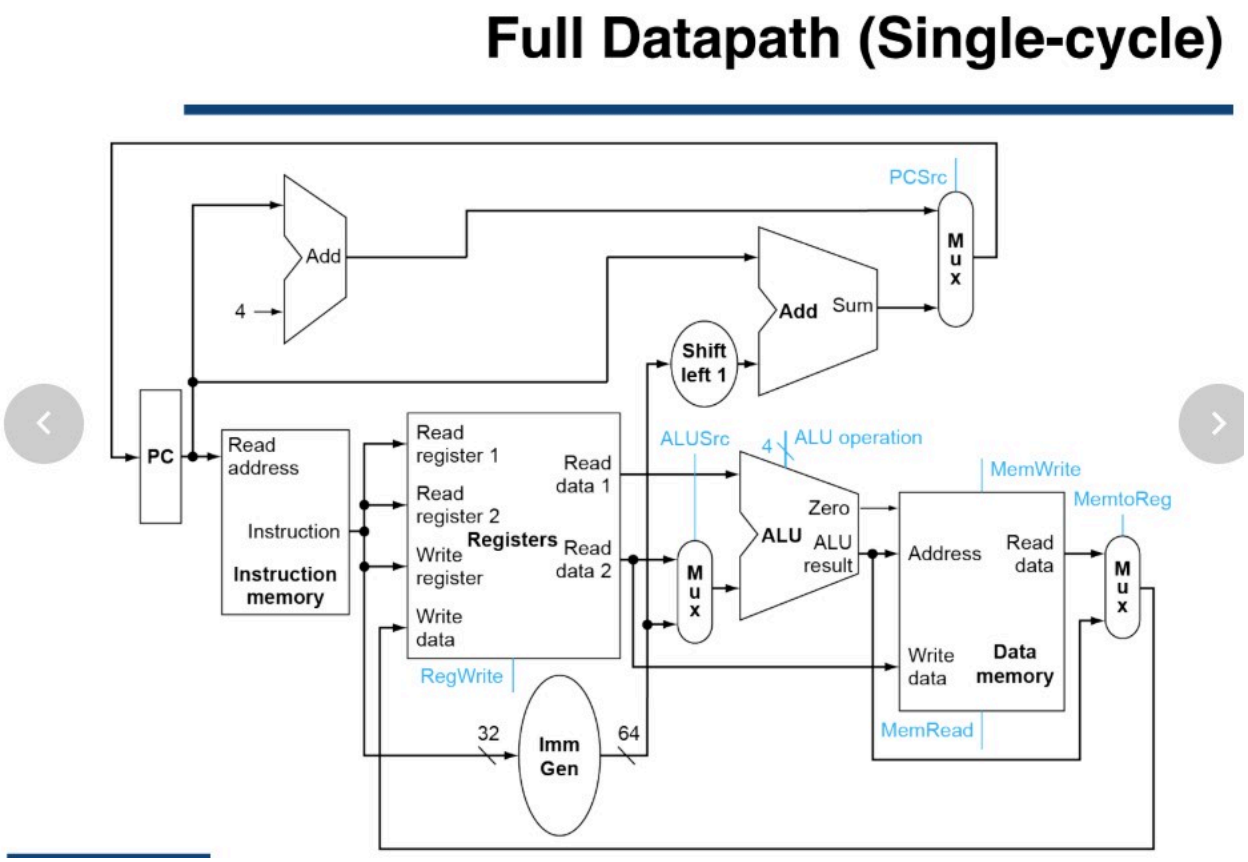
RTL DESIGN



SINGLE CYCLE 64-BIT RISC-V PROCESSOR

about

-A single-cycle 64-bit RISC-V processor is a simplified design of a RISC-V CPU that executes each instruction in a single clock cycle.



Key Components of a Single-Cycle 64-bit RISC-V Processor

1. **Instruction Fetch (IF) Unit**:
    - Fetches instructions from memory using the program counter (PC).
    - Updates the PC after each instruction fetch
    - **Instruction Decode (ID) Unit**:
    - Decodes the instruction and identifies the control signals, such as ALU operations, memory operations, and branching.
    - Reads values from the register file based on source register addresses.
2. **Register File**:
    - The register file provides data to the ALU or other components and can be written back with new values.
3. **ALU (Arithmetic Logic Unit)**:
    - Performs arithmetic and logical operations based on control signals.
    - Examples include addition, subtraction, bitwise AND/OR, shifts, comparisons, etc.
    - Supports operations like `ADD`, `SUB`, `AND`, `OR`, `SLT`, etc.
4. **Data Memory**:
    - Used for `load` and `store` instructions (`lw`, `sw`, etc.).
    - Memory accesses are completed in a single cycle, consistent with the single-cycle nature.
5. **Control Unit**:
    - Generates control signals for the entire processor based on the decoded instruction.
    - Manages the ALU operation, register file access, memory read/write, and branch control.
    - Generates the appropriate signals for conditional branching and jumps.

Tools used:

Modelsim

Synopsys Vcs

Design compiler

Rtl design

Overview of the RISC-V Processor Design

1. **Program Counter (PC)**:
    - The `Program_Counter` module keeps track of the address of the next instruction to be executed.
    - It is updated every clock cycle or when a branch occurs.
    - The PC output (`PC_Out`) is used to fetch instructions from the instruction memory, and the `PC_In` determines the address of the next instruction.
2. **Adder for PC Increment**:
    - The `Adder` module (`a1`) calculates the next sequential PC address by adding 4 to the current `PC_Out`, moving to the next instruction.
3. **Instruction Memory**:
    - `Instruction_Memory` (`im1`) fetches the instruction based on the address (`PC_Out`) provided by the program counter.
    - It outputs a 32-bit `Instruction`.
4. **Instruction Parsing**:
    - `inst_parser` (`ip1`) decodes the 32-bit instruction into its components: `opcode`, `rd`, `rs1`, `rs2`, `funct3`, and `funct7`.
    - These components are used for identifying the type of instruction (e.g., arithmetic, load/store, branch).
5. **Immediate Data Extraction**:
    - `imm_data_extractor` (`id1`) extracts immediate values from the instruction. Immediate values are required for certain types of instructions (e.g., loads, branches).
6. **Register File**:
    - `registerFile` (`rf1`) reads data from two source registers (`rs1` and `rs2`) and writes the result back into a destination register (`rd`) if the `RegWrite` signal is high.
    - `ReadData1` and `ReadData2` hold the values read from the registers.
7. **Control Unit**:
    - `Control_Unit` (`cu1`) generates control signals (`Branch`, `MemRead`, `MemtoReg`, `ALUOp`, `MemWrite`, `ALUSrc`, `RegWrite`) based on the `opcode` of the instruction.
    - These control signals guide the data flow and operation of other components, such as the ALU and data memory.
8. **ALU Control**:
    - `ALU_Control` (`ac1`) decodes the `ALUOp` signal from the control unit and the `funct` field of the instruction to generate a specific `Operation` signal for the ALU.
    - This signal determines the ALU's behavior (e.g., addition, subtraction, logical operations).

9. **ALU Input Mux**:
    - `mux_64` (`ALUIn_mux`) selects between `ReadData2` (register data) or `imm_data` (immediate value) as the second input to the ALU, based on the `ALUSrc` control signal.
10. **Adder for Branching**:
    - The second `Adder` (`a2`) calculates the branch target address by adding the immediate value (shifted left by one) to the `PC_Out`.
11. **Branch Decision**:
    - `Branch_Control` (`b1`) uses the result of the ALU (`zero` and `Less` flags) and the `funct` field to determine if a branch should be taken.
    - The `branch_op` signal indicates if the condition for branching is met.
12. **Conditional Mux for Next PC**:
    - `mux_64` (`conditional_mux`) selects either the next sequential PC (`out1`) or the branch target address (`out2`) as the next `PC_In`, based on the `Branch` and `branch_op` signals.
13. **ALU**:
    - `ALU_64_bit` (`alu`) performs arithmetic and logical operations based on the control signals and operands provided.
    - It outputs the `Result` of the operation, along with `zero` and `Less` signals used for branch decisions.
14. **Data Memory**:
    - `Data_Memory` (`dm1`) performs load and store operations based on the `MemRead` and `MemWrite` control signals.
    - The `Result` from the ALU is used as the memory address, `ReadData2` provides data to be written to memory, and `Read_Data` holds the data read from memory.
15. **Write-Back Mux**:
    - `mux_64` (`DataMem_mux`) selects either the data read from memory (`Read_Data`) or the ALU result (`Result`) to be written back to the register file, based on the `MemtoReg` signal.

Execution Flow of the Processor

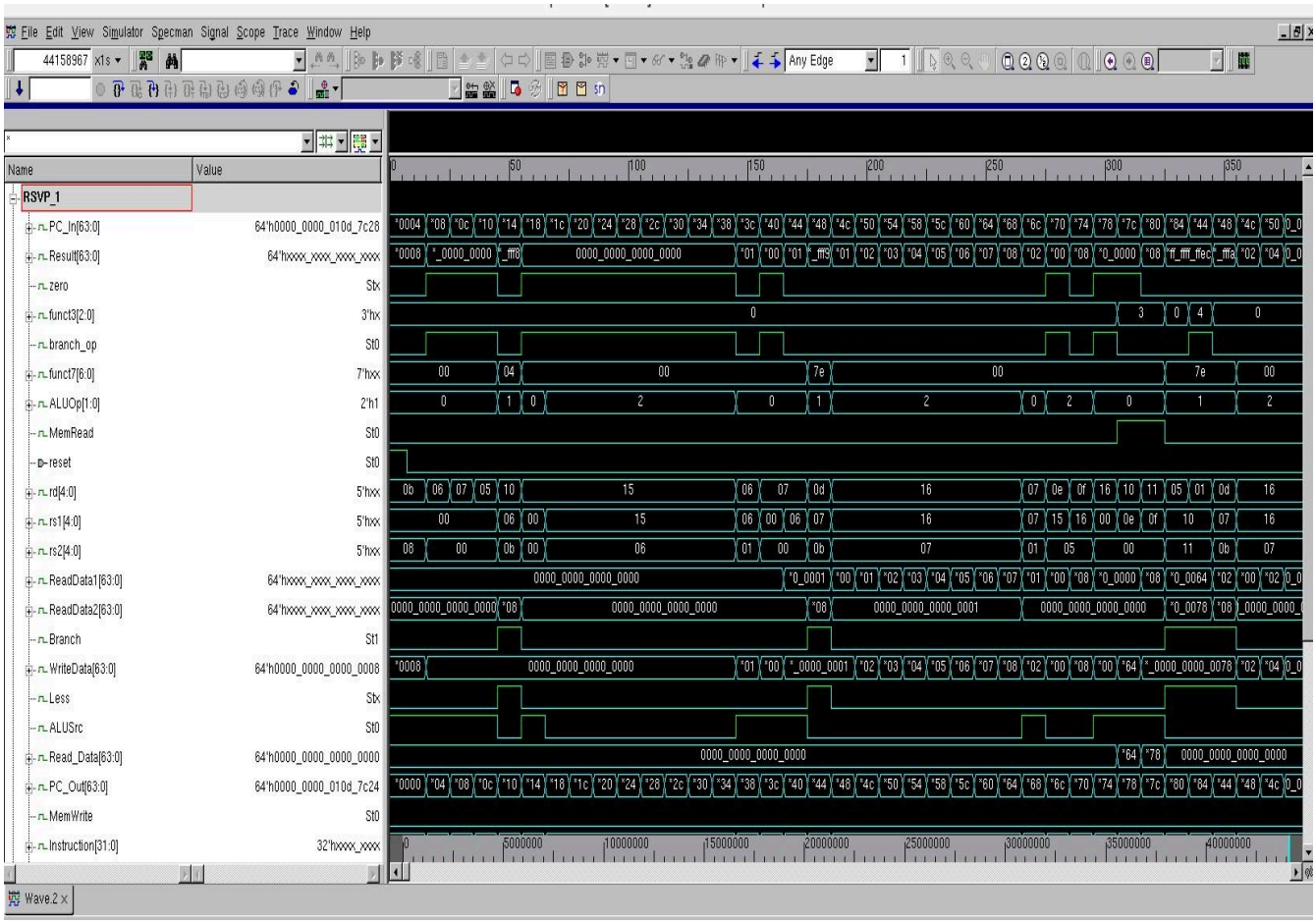
1. **Fetch**:
    - The PC points to the address of the current instruction. The instruction memory fetches the instruction.
2. **Decode**:
    - The instruction is decoded into its constituent parts, and the immediate data is extracted.
    - Control signals are generated based on the decoded `opcode`.
3. **Execute**:
    - The ALU computes a result using register values or immediate data.
    - Branch target address is computed if the instruction is a branch.
4. **Memory Access**:
    - For load instructions, data is read from the data memory.
    - For store instructions, data is written to the data memory.
    - For other instructions, this stage may be bypassed.
5. **Write-Back**:
    - The result (from either the ALU or memory) is written back into the register file at the destination register (`rd`).

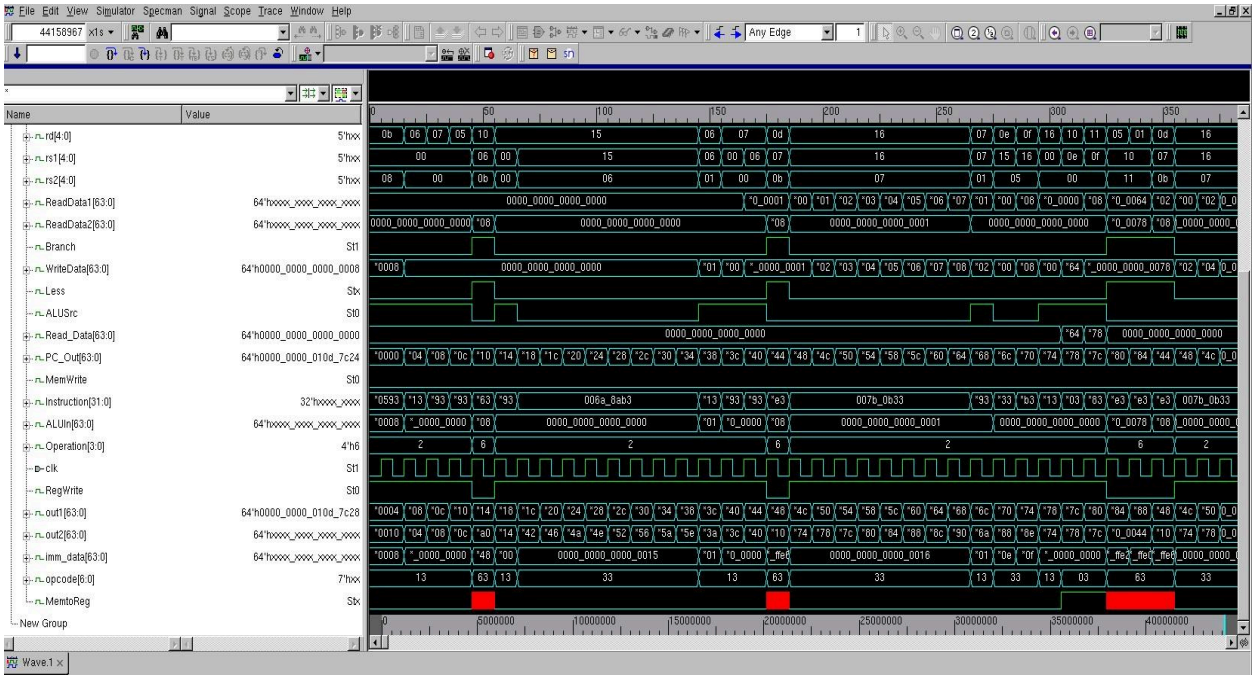
```
module RISC_V_Processor
(
  input clk, reset
);
    wire[63:0] PC_In, PC_Out, out1, out2, imm_data, ReadData1, ReadData2, ALUIn, Result, Read_Data, WriteData;
  wire[31:0] Instruction;
  wire[6:0] opcode,funct7;
  wire[4:0] rd,rs1,rs2;
  wire[2:0] funct3;
  wire Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite, zero, Less, branch_op;
  wire[1:0] ALUOp;
  wire[3:0] Operation;

  Program_Counter pc1
  (
    .clk(clk),
    .reset(reset),
    .PC_In(PC_In),
    .PC_Out(PC_Out)
  );

  Adder a1
  (
    .a(PC_Out),
    .b(64'd4),
    .out(out1)
  );

  Instruction_Memory im1
  (
    .Inst_Address(PC_Out),
    .Instruction(Instruction)
  );

  inst_parser ip1
  (
    .instruction(Instruction),
    .opcode(opcode),
    .rs1(rs1),
    .rs2(rs2),
    .rd(rd),
    .funct3(funct3),
```

```
module RISC_V_Processor

  mux_64 ALUIn_mux
  (
    .a(ReadData2),
    .b(imm_data),
    .sel(ALUSrc),
    .dataout(ALUIn)
  );

  Adder a2
  (
    .a(PC_Out),
    .b({imm_data[62:0],1'b0}),
    .out(out2)
  );

  mux_64 conditional_mux
  (
    .a(out1),
    .b(out2),
    .sel(Branch & branch_op),
    .dataout(PC_In)
  );

  ALU_64_bit alu
  (
    .a(ReadData1),
    .b(ALUIn),
    .ALUOp(Operation),
    .Result(Result),
    .zero(zero),
    .Less(Less)
  );

  Branch_Control b1
  (
    .Funct({Instruction[30], Instruction[14:12]}),
    .zero(zero),
    .Less(Less),
      .branch_op(branch_op)
  );
```

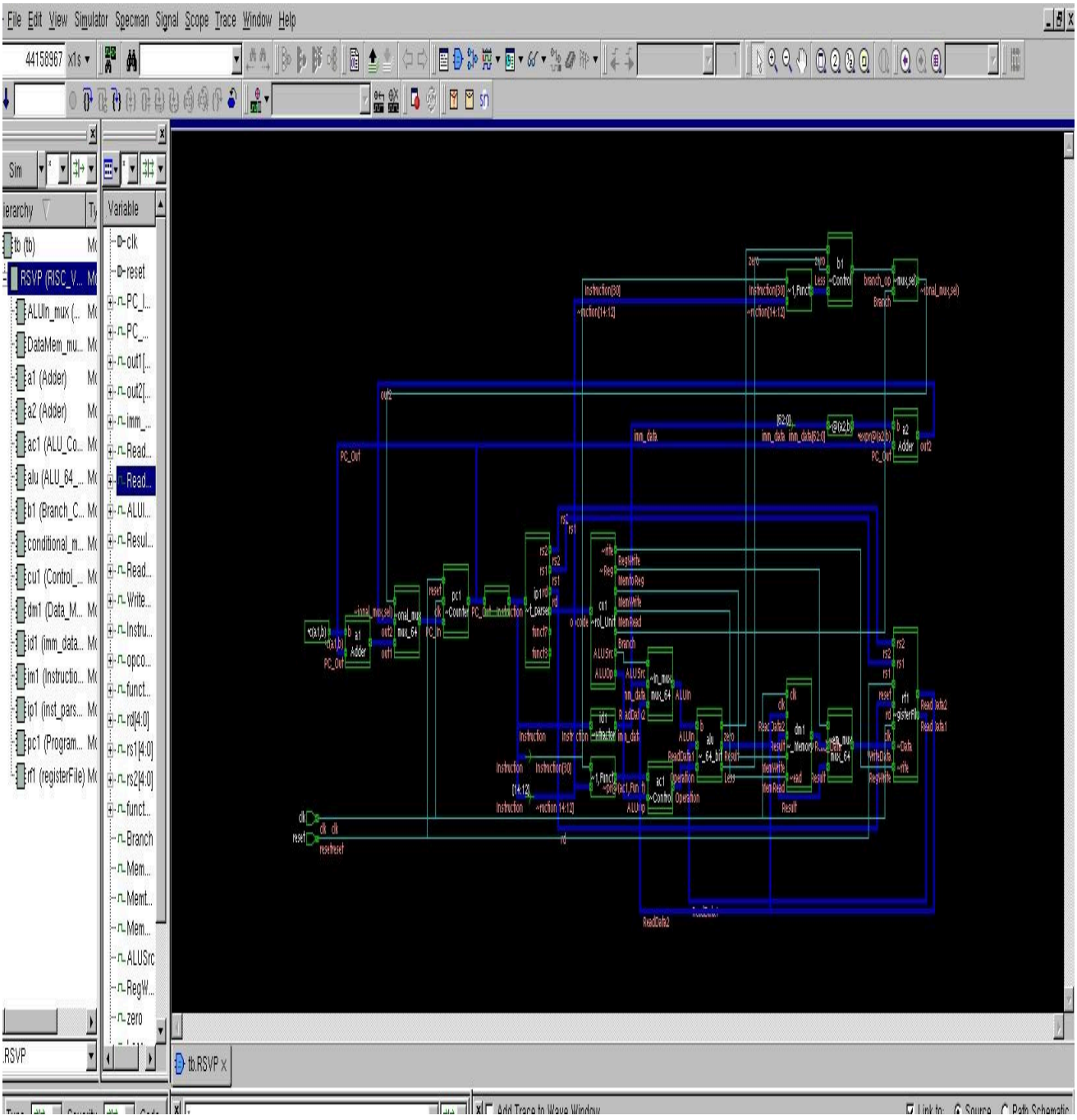
```
module ALU_64_bit(

input[63:0] a,
input[63:0] b,
input[3:0] ALUOp,
output reg[63:0] Result,
output reg zero,
output reg Less
);

  always @(a or b or ALUOp)
  begin
    case (ALUOp)
    4'b0000:
          Result = a & b;
    4'b0001:
      Result = a | b;
    4'b0010:
      Result = a + b;
    4'b0110:
      Result = a - b;
    endcase
    zero <= Result ? 64'b0 : 64'b1;
    Less = Result[63];

  end

endmodule
```

Synopsys VCS simulation results

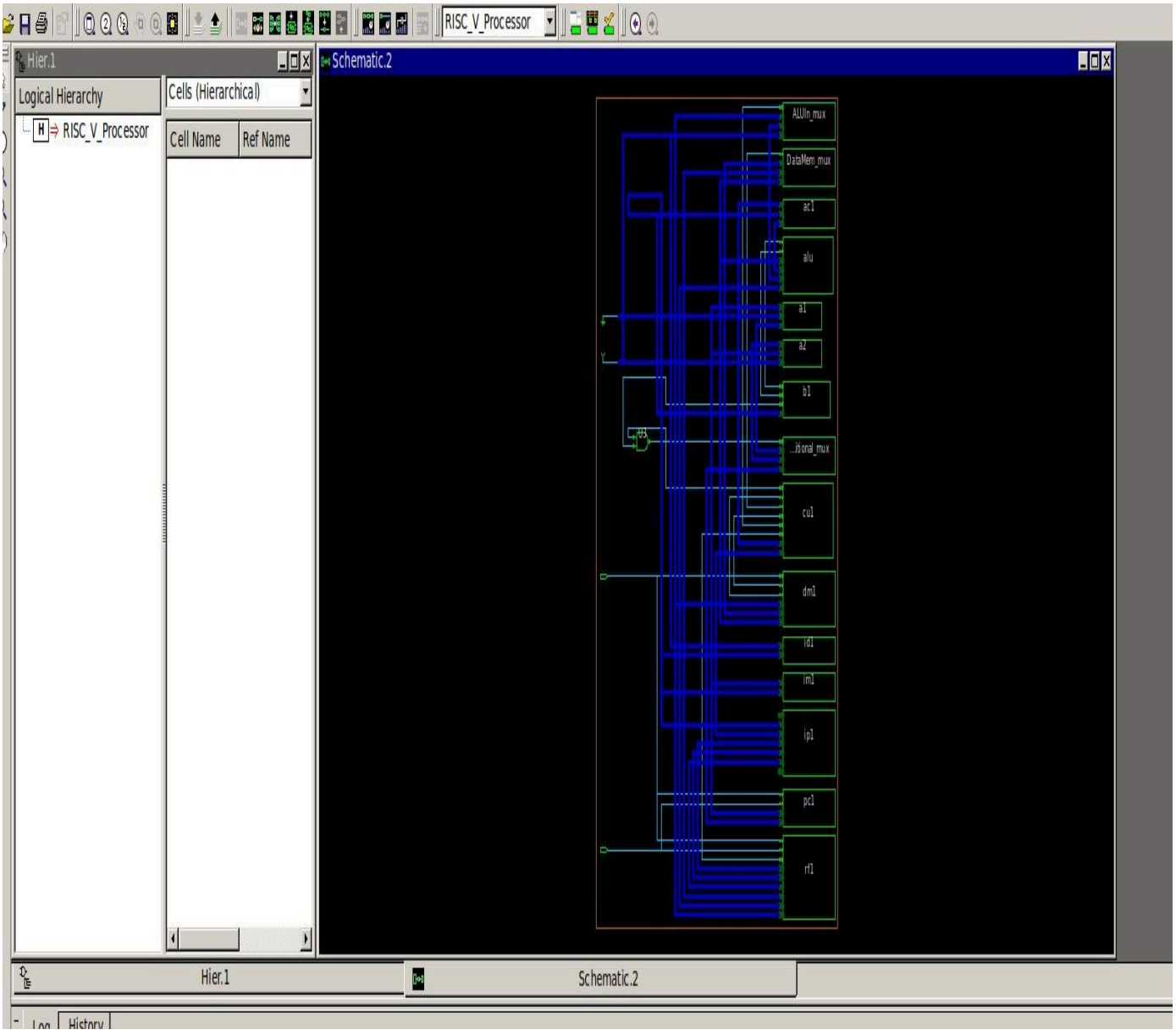




SCHEMATICS


File Edit View Simulator Specman Signal Scope Trace Window Help
Sim
Hierarchy
Variable
tb (tb)
RSVP (RISC_V...
ALUIn_mux (...
DataMem_mu...
a1 (Adder)
a2 (Adder)
ac1 (ALU_Co...
alu (ALU_64_...
b1 (Branch_C...
conditional_m...
cu1 (Control_...
dm1 (Data_M...
id1 (imm_data...
im1 (Instructio...
ip1 (inst_pars...
pc1 (Program...
rf1 (registerFile)
clk
reset
PC_Out
imm_data
Instruction
RegWrite
MemtoReg
MemWrite
MemRead
Branch
ALUSrc
ALUOp
ALUIn
ReadData1
ReadData2
Operation
Less
zero
Result
tb.RSVP

Synthesis by Synopsys design compiler



Power results after synthesis

```
Global Operating Voltage = 1.8
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1pW


  Cell Internal Power  = 137.8948 nW   (64%)
  Net Switching Power  =  77.2416 nW   (36%)
                         ---------
Total Dynamic Power    = 215.1364 nW  (100%)

Cell Leakage Power     =  44.3038 pW
```