MIPS Processor

Portfolio’s link: Muhammad Faraz Malik's Portfolio

BY: Muhammad Faraz Malik

Date: 14th May,2025

MIPS Processor:

A **MIPS processor** is a type of microprocessor based on the **MIPS (Microprocessor without Interlocked Pipeline Stages)** architecture, which is a **RISC (Reduced Instruction Set Computer)** architecture developed in the 1980s by MIPS Computer Systems.

Key Features of MIPS Processor:

- **RISC-based design**: Uses a small set of simple instructions for fast execution.
- **Fixed-length instructions**: Each instruction is 32 bits, simplifying decoding.
- **Load/store architecture**: Memory is accessed only through specific load and store instructions.
- **Registers**: Uses a set of 32 general-purpose registers.
- **Pipeline architecture**: Often uses 5-stage pipeline (IF, ID, EX, MEM, WB) for improved performance.

Introduction:

The Complex Engineering Problem (CEP) outlined in the document focuses on the design and implementation of a single-cycle MIPS processor on the Nexys A7 FPGA board. This project integrates fundamental concepts of digital systems design, computer architecture, and hardware description languages to create a functional processor capable of executing a subset of the MIPS Instruction Set Architecture (ISA). The primary objective is to develop a synthesizable Verilog implementation of a single-cycle MIPS processor that can execute R-type (add, sub, and, or, slt), I-type (addi, lw, sw, beq), and J-type (j) instructions, utilizing the Nexys A7s on-board memory, LEDs, and 7-segment displays for instruction storage, status indication, and result visualization. The processor must operate within a single clock cycle for all instructions, necessitating careful consideration of the critical path to ensure timing constraints are met. The design process emphasizes modular Verilog coding practices to enhance readability and maintainability, with distinct modules for the control unit, ALU, register file, and memory. The project also involves integrating the processor with the Nexys A7 FPGA, including mechanisms to load programs into memory and display outputs via on-board peripherals.

Beyond implementation, the project requires comprehensive testing through a Verilog testbench to verify functionality across various test cases, including boundary conditions. Deliverables include well-documented Verilog code, a Vivado project with a bitstream for the Nexys A7, a detailed design document, and a live demonstration of a simple MIPS program running on the FPGA. The assignment encourages a systematic approach to digital design, highlighting the importance of timing analysis, modular design, and rigorous verification to achieve a robust and efficient processor implementation. Optional extra credit opportunities, such as adding more instructions or a debugging interface, allow for further exploration of advanced design techniques.

Tools Required:

- Xilinx Vivado

Xilinx Vivado Design Suite is a comprehensive software platform used for the design, synthesis, implementation, and analysis of digital systems on Xilinx FPGA devices.

Design Modules Codes:

MIPS_Processor_Module

```
// Top-level module module

mips_processor (

   input wire clk,              // Clock input     input wire

reset,            // Reset input     output wire [15:0] led,       //

LEDs for status     output wire [6:0] seg,        // 7-segment

display segments     output wire [3:0] an          // 7-segment

display anodes

);

   // Internal signals

   wire [31:0] pc, instr, alu_result, write_data, read_data1, read_data2, imm_ext, branch_target,

jump_target;     wire [31:0] mem_data, next_pc;     wire [4:0] write_reg;     wire [3:0] alu_control;

wire reg_dst, alu_src, mem_to_reg, reg_write, mem_write, branch, jump, zero;       wire [15:0]

display_data;     reg [31:0] t3_data;          // Register to hold $t3 value, updated synchronously

   // Program Counter     reg [31:0] pc_reg;

always @(posedge clk or posedge reset) begin

      if (reset)

         pc_reg <= 32h00000000;
      else            pc_reg <=

next_pc;     end     assign pc =

pc_reg;     // Instruction Memory

instruction_memory instr_mem (

      .addr(pc[9:2]),
```

```
        .instr(instr)
    );
    // Control Unit
control_unit ctrl (
        .opcode(instr[31:26]),
        .funct(instr[5:0]),
        .reg_dst(reg_dst),
        .alu_src(alu_src),
        .mem_to_reg(mem_to_reg),
        .reg_write(reg_write),
        .mem_write(mem_write),
        .branch(branch),
        .jump(jump),
        .alu_control(alu_control)
    );
    // Register File (main instance)
          register_file reg_file (
        .clk(clk),
        .we(reg_write),
        .ra1(instr[25:21]),
        .ra2(instr[20:16]),
        .wa(write_reg),
        .wd(write_data),
        .rd1(read_data1),
```

```
        .rd2(read_data2)

    );

    // Synchronous read of $t3 for display

always @(posedge clk or posedge reset) begin

        if (reset)

            t3_data <= 32h00000000;         else if (reg_write && write_reg == 5d11) //

Update t3_data when $t3 is written            t3_data <= write_data;     end

    // Immediate Extension     assign imm_ext = {{16{instr[15]}}, instr[15:0]}; //

Sign-extend 16-bit immediate

    // ALU Source Mux     wire [31:0] alu_src_b = alu_src ?

imm_ext : read_data2;

    // ALU

    alu alu_inst (
        .a(read_data1),

        .b(alu_src_b),

        .alu_control(alu_control),

        .result(alu_result),

        .zero(zero)

    );

    // Data Memory

data_memory data_mem (

        .clk(clk),

        .we(mem_write),

        .addr(alu_result[9:2]),
```

```
        .wd(read_data2),
        .rd(mem_data)
    );
    // Write Register Mux     assign write_reg = reg_dst ?
instr[15:11] : instr[20:16];
    // Write Data Mux     assign write_data = mem_to_reg ?
mem_data : alu_result;
    // Branch and Jump Logic     assign branch_target = pc + 4 + (imm_ext << 2);
assign jump_target = {pc[31:28], instr[25:0], 2b00}; assign next_pc = jump ?
jump_target : (branch && zero ? branch_target : pc + 4);
    // LED Status
    assign led = {14b0, reset, ~reset}; // LED[1] = reset, LED[0] = running
    // 7-Segment Display: Always show $t3s value
assign display_data = t3_data[15:0];
seven_segment_display ssd (
        .clk(clk),
        .reset(reset),
        .data(display_data),
        .seg(seg),
        .an(an)
    );
    // Debug signal to verify display data
wire [15:0] debug_data = display_data;
endmodule
```

Instruction_Memory_Module

```
// Instruction Memory module

instruction_memory (     input

wire [7:0] addr,     output wire

[31:0] instr

);

    reg [31:0] mem [0:255];



    initial begin

        // Sample program:
        mem[0] = 32h20080005; // addi $t0, $zero, 5

mem[1] = 32h2009000a; // addi $t1, $zero, 10 mem[2]

= 32h01095020; // add $t2, $t0, $t1         mem[3] =

32hac0a0000; // sw $t2, 0($zero)         mem[4] =

32h8c0b0000; // lw $t3, 0($zero)         mem[5] =

32h08000000; // j 0 (loop back to start)     end

assign instr = mem[addr]; endmodule
```

Data_Memory_Module

```
// Data Memory module

data_memory (

    input wire clk,

input wire we,     input

wire [7:0] addr,     input
```

```
wire [31:0] wd,

output wire [31:0] rd

);

   reg [31:0] mem [0:255];


   always @(posedge clk) begin

       if (we)

mem[addr] <= wd; end

assign rd = mem[addr];

endmodule
```

Control_Unit_Module

```
// Control Unit module

control_unit (     input wire

[5:0] opcode,     input wire

[5:0] funct,     output reg

reg_dst,     output reg alu_src,

output reg mem_to_reg,

output reg reg_write,

output reg mem_write,

output reg branch,     output

reg jump,     output reg [3:0]

alu_control

);
```

```
always @(*) begin
    // Default control signals
reg_dst = 0;        alu_src = 0;
mem_to_reg = 0;
reg_write = 0; mem_write =
0; branch = 0; jump = 0;
alu_control = 4b0000;

    case (opcode)
6b000000: begin // R-type
reg_dst = 1;                reg_write
= 1;                case (funct)
                        6b100000: alu_control = 4b0010; // add
            6b100010: alu_control = 4b0110; // sub
                6b100100: alu_control = 4b0000; // and
                6b100101: alu_control = 4b0001; // or
6b101010: alu_control = 4b0111; // slt
default: alu_control = 4b0000;                endcase
end
        6b001000: begin // addi
alu_src = 1;                reg_write = 1;
alu_control = 4b0010; // add
end
```

```
6b100011: begin // lw
alu_src = 1;                mem_to_reg =
1;                reg_write = 1;
alu_control = 4b0010; // add
end
6b101011: begin // sw
alu_src = 1;                mem_write = 1;
alu_control = 4b0010; // add
end
6b000100: begin // beq
branch = 1;                alu_control =
4b0110; // sub            end
6b000010: begin // j                jump =
1;            end            default: begin
// Default case            end
endcase     end
endmodule
```

ALU_Module

```
// ALU module alu (     input
wire [31:0] a,     input wire
[31:0] b,     input wire [3:0]
alu_control,     output reg
```

```
[31:0] result,     output wire

zero

);

   always @(*) begin

case (alu_control)

            4b0000: result = a & b; // AND

            4b0001: result = a | b; // OR

            4b0010: result = a + b; // ADD

            4b0110: result = a - b; // SUB

            4b0111: result = (a < b) ? 32d1 : 32d0; // SLT

default: result = 32d0;         endcase     end     assign

zero = (result == 32d0); endmodule
```

Register_File_Module

```
// Register File module

register_file (     input wire clk,

input wire we,     input wire

[4:0] ra1, ra2, wa,     input wire

[31:0] wd,     output wire

[31:0] rd1, rd2

);     reg [31:0] registers [31:0];

integer i;     initial begin

for (i = 0; i < 32; i = i + 1)

registers[i] = 32d0;     end
```

```
always @(posedge clk) begin

if (we && wa != 0)

registers[wa] <= wd;     end


   assign rd1 = (ra1 != 0) ? registers[ra1] : 32d0;

assign rd2 = (ra2 != 0) ? registers[ra2] : 32d0;

endmodule
```

Seven_Segment_Display_Module

```
// 7-Segment Display Controller module

seven_segment_display (

   input wire clk,

input wire reset,     input

wire [15:0] data,

output reg [6:0] seg,

output reg [3:0] an

);     reg [3:0] digit;

reg [16:0] clk_div;


   always @(posedge clk or posedge reset) begin

      if (reset)            clk_div

<= 0;        else             clk_div

<= clk_div + 1;     end
```

```
always @(*) begin         case
(clk_div[16:15])
        2b00: begin an = 4b1110; digit = data[3:0]; end   // Digit 0 (rightmost)
        2b01: begin an = 4b1101; digit = data[7:4]; end   // Digit 1
        2b10: begin an = 4b1011; digit = data[11:8]; end  // Digit 2
        2b11: begin an = 4b0111; digit = data[15:12]; end // Digit 3 (leftmost)         endcase     end
always @(*) begin
      case (digit)
        4h0: seg = 7b1000000; // 0
        4h1: seg = 7b1111001; // 1
        4h2: seg = 7b0100100; // 2
        4h3: seg = 7b0110000; // 3
        4h4: seg = 7b0011001; // 4
        4h5: seg = 7b0010010; // 5
        4h6: seg = 7b0000010; // 6
        4h7: seg = 7b1111000; // 7
        4h8: seg = 7b0000000; // 8
        4h9: seg = 7b0010000; // 9
        4hA: seg = 7b0001000; // A
        4hB: seg = 7b0000011; // B
        4hC: seg = 7b1000110; // C
        4hD: seg = 7b0100001; // D
```

4hE: seg = 7b0000110; // E 4hF:

seg = 7b0001110; // F default: seg =

7b1000000; // Default to 0 endcase end

endmodule

Design Modules Codes Explanation:

MIPS_Processor_Module_Explanation

The provided Verilog code implements a single-cycle MIPS processor for the Nexys A7 FPGA, adhering to the Complex Engineering Problem (CEP) requirements. Below is a brief, pointed explanation of the top-level module:

Module Declaration: **mips_processor:** Top-level module interfacing with Nexys A7.
Inputs: clk (clock), reset (reset signal).
Outputs: led (16-bit status LEDs), seg (7-segment display segments), an (7-segment display anodes).

Program Counter (PC): **pc_reg:** 32-bit register storing current PC, initialized to 0 on reset.
Updates on posedge clk to next_pc (next instruction address).
pc: Output of pc_reg.

Instruction Memory:

instruction_memory module: Fetches 32-bit instruction (instr) from memory using pc[9:2] as the address.

Control Unit: **control_unit module:** Decodes opcode (instr[31:26]) and funct (instr[5:0]).
Generates control signals: reg_dst, alu_src, mem_to_reg, reg_write, mem_write, branch, jump, alu_control.

Register File: **register_file module:** 32 registers, 32bit wide.
Reads: ra1 (instr[25:21]), ra2 (instr[20:16]) produce read_data1, read_data2.
Writes: Enabled by reg_write, writes write_data to write_reg. **t3**

Register Tracking: **t3_data:** 32bit register to hold $t3 (register 11) value for display.
Updates on posedge clk when reg_write is active and write_reg is 11; resets to 0.

Immediate Extension: **imm_ext:** Signextends 16bit immediate (instr[15:0]) to 32 bits.

ALU: **alu module:** Performs operations on read_data1 and alu_src_b (selected by alu_src).
alu_src_b: Mux selects read_data2 or imm_ext.
Outputs: alu_result (operation result), zero (zero flag for branching).

Data Memory: **data_memory module:** Handles load/store operations.
Writes read_data2 to addr (alu_result[9:2]) if mem_write is active.
Reads data into mem_data.

Write Register Mux: **write_reg:** Selects destination register (instr[15:11] if reg_dst, else instr[20:16]).

Write Data Mux: **write_data:** Selects mem_data (for loads) or alu_result (for Rtype/Itype) based on mem_to_reg.

Branch and Jump Logic: **branch_target:** Computes branch address (pc + 4 + (imm_ext << 2)). **jump_target:** Forms jump address ({pc[31:28], instr[25:0], 2b00}).
next_pc: Selects jump_target (if jump), branch_target (if branch and zero), or pc + 4.

LED Status: **led:** Displays processor status (led[1] = reset, led[0] = ~reset).

7Segment Display: **seven_segment_display module:** Displays lower 16 bits of t3_data (display_data).
Outputs: seg (segment patterns), an (anode signals for digit selection).

Debug Signal: **debug_data:** Mirrors display_data for verification.

Instruction_Memory_Module_Explanation

The provided Verilog code defines the instruction_memory module for the single-cycle MIPS processor, as specified in the Complex Engineering Problem (CEP). Below is a brief, pointed explanation of the module:

Module Declaration: **instruction_memory:** Stores MIPS instructions for the processor.
Inputs: addr (8-bit address, corresponding to pc[9:2]).
Outputs: instr (32-bit instruction fetched from memory).

Memory Array: **mem:** 256-entry array of 32-bit registers (mem[0:255]), representing instruction memory.
Each entry holds a single 32-bit MIPS instruction.

Initial Program: **initial block:** Preloads memory with a sample MIPS program at simulation start.
Instructions:

- mem[0]: addi $t0, $zero, 5 (set $t0 = 5).
- mem[1]: addi $t1, $zero, 10 (set $t1 = 10).
- mem[2]: add $t2, $t0, $t1 (set $t2 = $t0 + $t1 = 15).
- mem[3]: sw $t2, 0($zero) (store $t2 value at memory address 0).
- mem[4]: lw $t3, 0($zero) (load value from memory address 0 into $t3).
- mem[5]: j 0 (jump to address 0, creating an infinite loop).

Instruction Fetch:

assign instr = mem[addr]: Continuously outputs the 32-bit instruction stored at mem[addr] to instr.

Data_Memory_Module_Explanation

The provided Verilog code defines the data_memory module for the single-cycle MIPS processor, as specified in the Complex Engineering Problem (CEP). Below is a brief, pointed explanation of the module:

Module Declaration:

data_memory: Implements the data memory for load (lw) and store (sw) operations.

Inputs:

- clk: Clock signal for synchronous writes.
- we: Write enable signal (active high for store operations).
- addr: 8-bit address (alu_result[9:2]) for memory access.
- wd: 32-bit write data (from read_data2 for stores).

Output:

- rd: 32-bit read data (output for loads).

Memory Array:

mem: 256-entry array of 32-bit registers (mem[0:255]), representing data memory.
Each entry stores a 32-bit word.

Write Operation:

always @(posedge clk): Synchronous write block triggered on positive clock edge.
If we is high, writes wd to mem[addr].
Supports sw instruction by storing data at the specified address.

Read Operation:

assign rd = mem[addr]: Continuously outputs the 32-bit data stored at mem[addr] to rd.
Supports lw instruction by providing data from the specified address.

Control_Unit_Module_Explanation

The provided Verilog code defines the control_unit module for the single-cycle MIPS processor, as specified in the Complex Engineering Problem (CEP). Below is a brief, pointed explanation of the module:

Module Declaration:

control_unit: Generates control signals based on instruction type.

Inputs:

- opcode: 6-bit instruction opcode (instr[31:26]).
- funct: 6-bit function code (instr[5:0]) for R-type instructions.

Outputs (all reg for combinational logic):

- reg_dst: Selects destination register (1 for R-type, 0 for I-type).
- alu_src: Selects ALU second operand (1 for immediate, 0 for register).
- mem_to_reg: Selects write-back data (1 for memory, 0 for ALU result).
- reg_write: Enables register write (1 for write, 0 otherwise).
- mem_write: Enables memory write (1 for sw, 0 otherwise).
- branch: Enables branch (beq).
- jump: Enables jump (j).
- alu_control: 4-bit signal to control ALU operation.

Combinational Logic: **always @(*):** Updates outputs whenever inputs (opcode, funct) change.

Default values: All control signals set to 0 to avoid unintended behavior.

Opcode Decoding:

Uses case statement to decode opcode and set control signals:

R-type (opcode = 6'b000000):

- reg_dst = 1, reg_write = 1.
- funct decoding:
- 6'b100000: alu_control = 4'b0010 (add).
- 6'b100010: alu_control = 4'b0110 (sub).
- 6'b100100: alu_control = 4'b0000 (and).
- 6'b100101: alu_control = 4'b0001 (or).
- 6'b101010: alu_control = 4'b0111 (slt).
- Default: alu_control = 4'b0000.

addi (opcode = 6'b001000):

- alu_src = 1, reg_write = 1, alu_control = 4'b0010 (add).

lw (opcode = 6'b100011):

- alu_src = 1, mem_to_reg = 1, reg_write = 1, alu_control = 4'b0010 (add).

sw (opcode = 6'b101011):

- alu_src = 1, mem_write = 1, alu_control = 4'b0010 (add).

beq (opcode = 6'b000100):

- branch = 1, alu_control = 4'b0110 (sub).

j (opcode = 6'b000010):

- jump = 1.

Default: All signals remain at default (0).

ALU_Module_Explanation

The provided Verilog code defines the alu (Arithmetic Logic Unit) module for the single-cycle MIPS processor, as specified in the Complex Engineering Problem (CEP). Below is a brief, pointed explanation of the module:

Module Declaration: **alu:** Performs arithmetic and logical operations for the MIPS processor.
Inputs:

- a: 32-bit first operand (from read_data1).
- b: 32-bit second operand (from alu_src_b, either read_data2 or imm_ext).
- alu_control: 4-bit control signal (from control_unit) to select operation. **Outputs:**
- result: 32-bit result of the ALU operation (reg for combinational logic).
- zero: 1-bit flag indicating if result is zero (for beq).

Combinational Logic: **always @(*):** Updates result whenever inputs (a, b, alu_control) change.
Uses case statement to select operation based on alu_control:

- 4'b0000: result = a & b (bitwise AND, for and instruction).
- 4'b0001: result = a | b (bitwise OR, for or instruction).
- 4'b0010: result = a + b (addition, for add, addi, lw, sw).
- 4'b0110: result = a - b (subtraction, for sub, beq).
- 4'b0111: result = (a < b) ? 32'd1 : 32'd0 (set less than, for slt; outputs 1 if a < b, else 0).
- Default: result = 32'd0 (for undefined control signals).

Zero Flag:
assign zero = (result == 32'd0): Outputs 1 if result is zero, used for branch condition in beq.

Register_File_Module_Explanation

The provided Verilog code defines the register_file module for the single-cycle MIPS processor, as specified in the Complex Engineering Problem (CEP). Below is a brief, pointed explanation of the module:

Module Declaration: **register_file:** Implements a 32-register file for storing and accessing 32-bit data.
Inputs:

- clk: Clock signal for synchronous writes.
- we: Write enable signal (from reg_write, active high for register writes).
- ra1, ra2: 5-bit read addresses (from instr[25:21] and instr[20:16]) for registers.
- wa: 5-bit write address (from write_reg, selecting destination register).
- wd: 32-bit write data (from write_data, either ALU result or memory data).

Outputs:

- rd1, rd2: 32-bit read data from registers at ra1 and ra2.

Register Array: **registers:** Array of 32 registers, each 32 bits wide (registers[31:0]).
initial block: Initializes all registers to 0 at simulation start.

Write Operation: **always @(posedge clk):** Synchronous write block triggered on positive clock edge.
If we is high and wa != 0, writes wd to registers[wa].
Protects $zero (register 0) from being overwritten (MIPS convention).

Read Operation:
assign rd1 = (ra1 != 0) ? registers[ra1] : 32'd0: Outputs data from registers[ra1] unless ra1 is 0, then outputs 0.
assign rd2 = (ra2 != 0) ? registers[ra2] : 32'd0: Outputs data from registers[ra2] unless ra2 is 0, then outputs 0.
Asynchronous reads ensure immediate data access for the single-cycle datapath.

Seven_Segment_Display_Module_Explanation

The provided Verilog code defines the seven_segment_display module for the single-cycle MIPS processor, as specified in the Complex Engineering Problem (CEP). This module controls the Nexys A7 FPGA's 7-segment display to show the lower 16 bits of the $t3 register value. Below is a brief, pointed explanation of the module:

Module Declaration:

seven_segment_display: Drives the 7-segment display to show a 16-bit value as four hexadecimal digits.
Inputs:

- clk: Clock signal for digit refresh.
- reset: Reset signal to initialize the display.
- data: 16-bit input data (lower 16 bits of $t3 register).

Outputs:

- seg: 7-bit signal controlling segment patterns (a-g) of the 7-segment display.
- an: 4-bit signal controlling anode activation for four digits (active low).

Clock Divider: **clk_div:** 17-bit register to divide the clock for digit multiplexing.
always @(posedge clk or posedge reset):

- On reset, clk_div is cleared to 0.
- Otherwise, increments clk_div on each clock cycle.

Upper bits (clk_div[16:15]) control digit selection for refresh rate.
Digit Selection:
always @(*): Combinational block to select active digit and data.
Based on clk_div[16:15]:

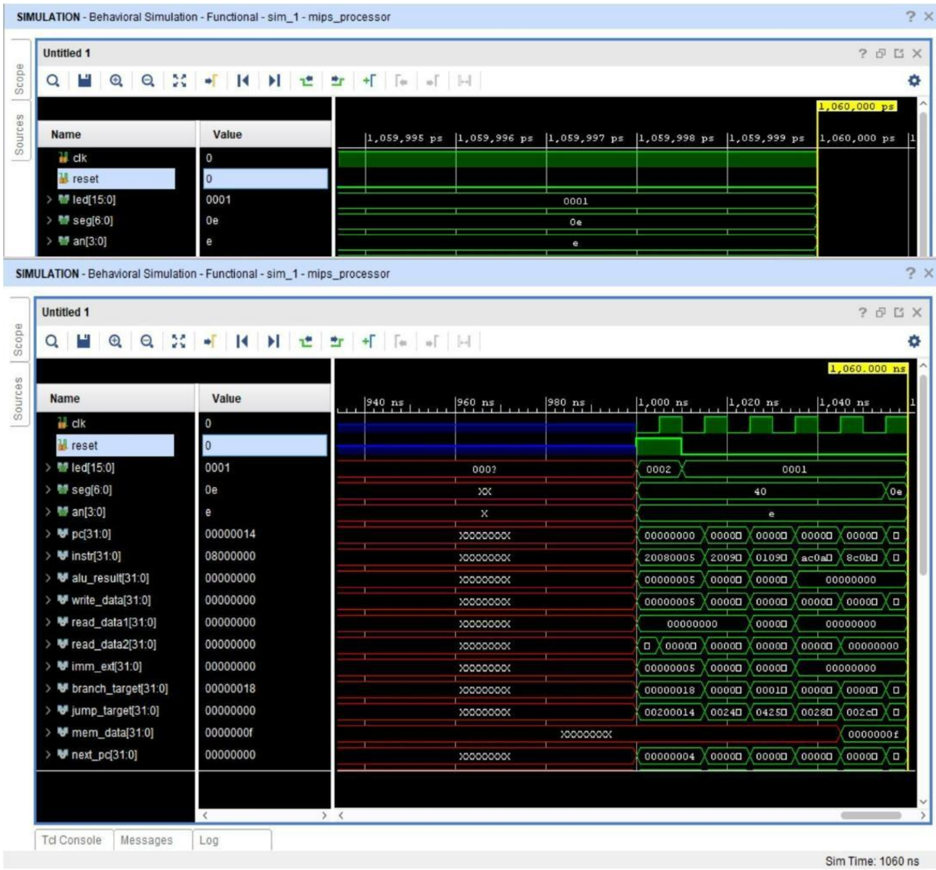
- 2'b00: an = 4'b1110, digit = data[3:0] (rightmost digit).
- 2'b01: an = 4'b1101, digit = data[7:4] (second digit).
- 2'b10: an = 4'b1011, digit = data[11:8] (third digit).
- 2'b11: an = 4'b0111, digit = data[15:12] (leftmost digit).

an activates one digit at a time (active low); digit is the 4-bit value to display.
Segment Encoding:
always @(*): Combinational block to map digit to 7-segment patterns.
case (digit): Maps 4-bit digit (0-F) to seg (7-bit pattern for segments a-g):

- 4'h0: 7'b1000000 (0)
- 4'h1: 7'b1111001 (1)
- ...
- 4'hF: 7'b0001110 (F)
- Default: 7'b1000000 (0)

Each pattern is active low (0 = segment on, 1 = segment off).

Circuit Simulation:



Circuit Simulation Explanation:

This image shows a simulation of a MIPS processor circuit using a behavioral and functional simulation tool. Here's an explanation in bullet points:

Simulation Type: Behavioral and functional simulation of a MIPS processor.

Tool Used: Likely a hardware description language (HDL) simulator (Xilinx Vivado).

Signals Displayed: ○ **clk**: Clock signal, toggling between 0 and 1, driving the processor's timing. ○ **reset**: Reset signal, set to 0, indicating the processor is not in reset mode. ○ **led[15:0]**: 16-bit LED output, showing "0001" (likely an output indicator). ○ **seg[6:0]**: 7-segment display output, showing "0e" (hexadecimal representation).

- **an[3:0]**: 4-bit anode signal for the 7-segment display, showing "e" (possibly enabling a specific digit).
- **pc[31:0]**: Program counter, incrementing (e.g., from 00000014 to 00000018), showing instruction fetch progress.
- **instr[31:0]**: Instruction register, holding the current instruction (e.g., 08000000, a jump instruction).
- **alu_result[31:0]**: ALU result, currently 00000000 (no ALU operation result yet).
- **write_data[31:0]**, **read_data1[31:0]**, **read_data2[31:0]**: Register file data, all 00000000 (no data read/write yet).
- **imm_ext[31:0]**: Immediate value (extended), 00000000 (no immediate value used).
- **branch_target[31:0]**: Branch target address, 00000018 (matches PC increment).
- **jump_target[31:0]**: Jump target address, 00000000 (no jump executed). ○ **mem_data[31:0]**: Memory data, 0000000f (data read from memory).
- **next_pc[31:0]**: Next program counter value, 00000004 (initial PC value).

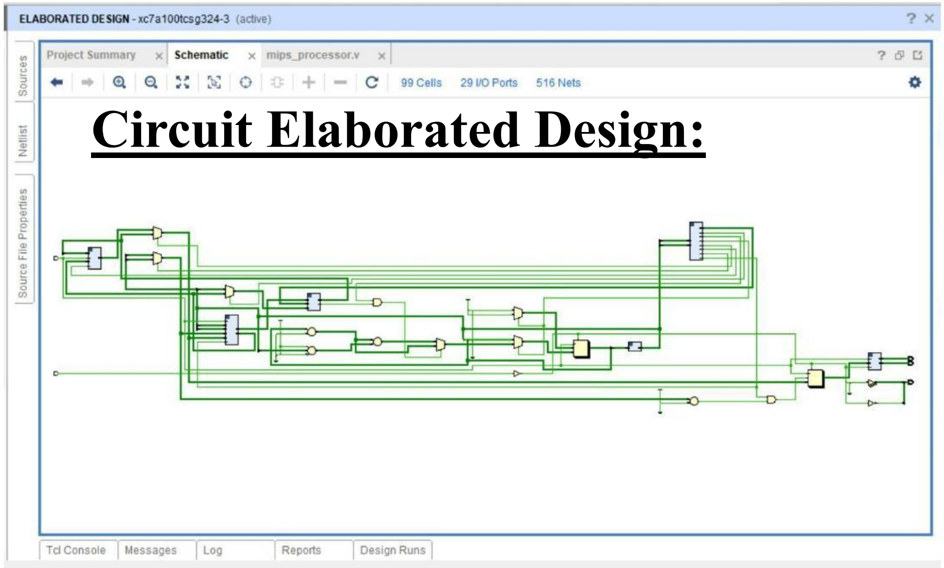
Time Scale: Simulation runs from 940 ns to 1,060 ns, with a cursor at 1,060 ns.

Waveform View: Displays signal transitions over time, with binary (0/1) and hexadecimal representations.

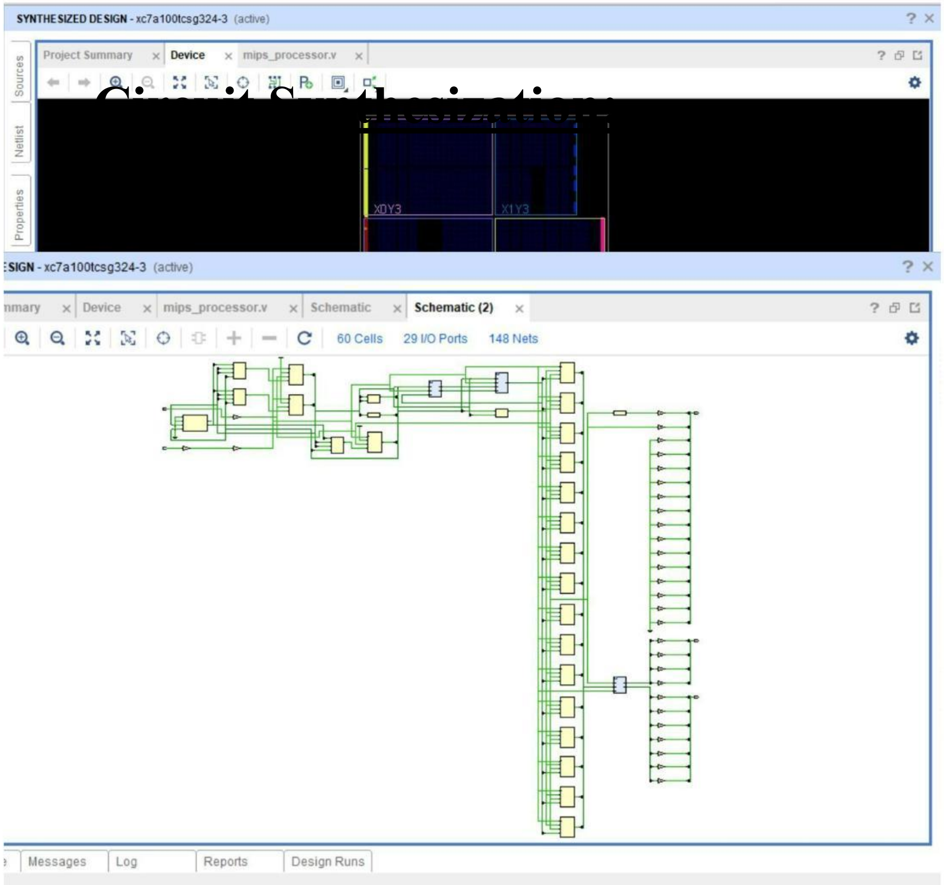
Simulation Progress: The processor is executing instructions, with the PC incrementing and instructions being fetched (e.g., 08000000).

Key Observation: The simulation shows the processor in an early stage, with minimal activity in ALU, memory, and registers, likely during initialization or a simple instruction sequence.

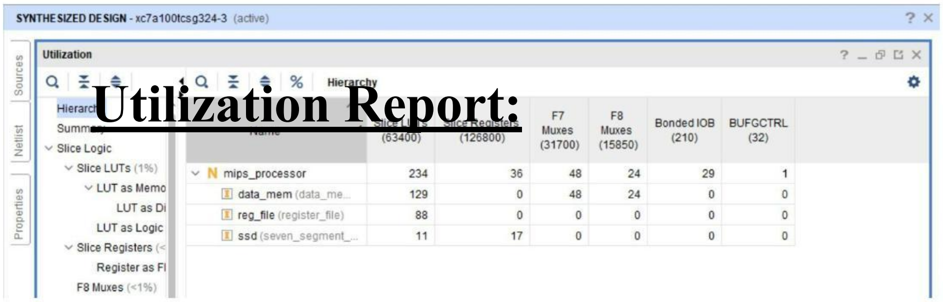
Circuit Elaborated Design:



Circuit Synthesization:



Utilization Report:



Utilization Report Explanation:

Design:

Synthesis report for a design targeting an xc7a100tcsg324-3 FPGA.

Hierarchy:

Breakdown of resource utilization: **mips_processor**:

Main module utilizing resources.

- **Slice LUTs**: 234 out of 63,400 (0.37%), used for logic operations. ○ **Slice Registers**: 36 out of 126,800 (0.03%), used for data storage. ○ **F7 Muxes**: 48 out of 3,170 (1.51%), used for 7-input LUT configurations. ○ **F8 Muxes**: 24 out of 1,585 (1.51%), used for 8-input LUT configurations. ○ **Bonded IOB**: 29 out of 210 (13.81%), used for input/output connections.
- **BUFGCTRL**: 1 out of 32 (3.13%), used for global clock buffering.

data_mem (data_memory):

Submodule for data memory. ○ **Slice LUTs**: 129 out of 63,400 (0.20%), used for memory implementation. ○ **F7 Muxes**: 48 out of 3,170 (1.51%). ○ **F8 Muxes**: 24 out of 1,585 (1.51%). **reg_file (register_file)**:

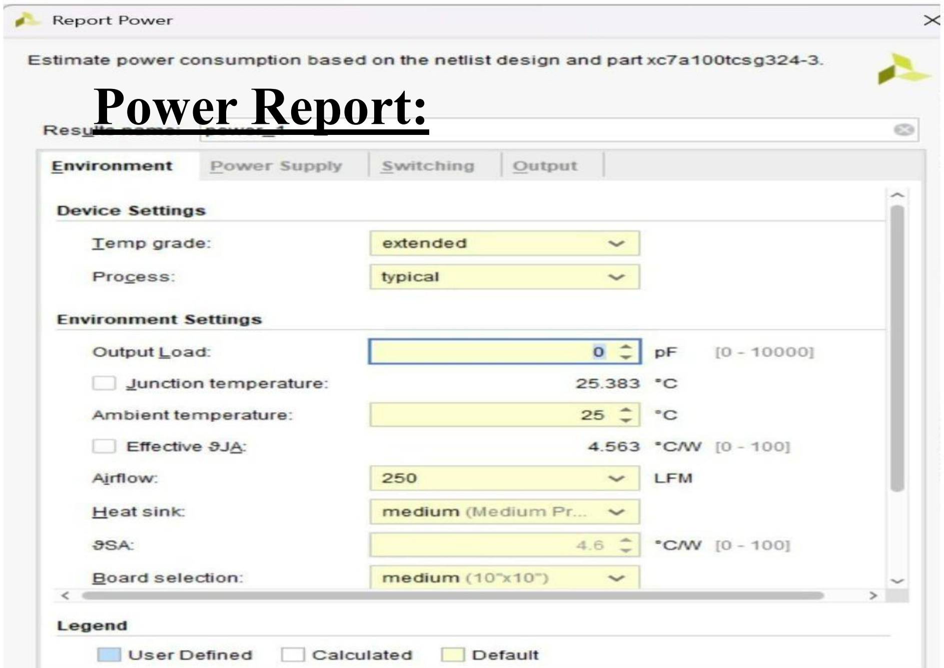
Submodule for register file.

- **Slice LUTs**: 88 out of 63,400 (0.14%). **ssd**

(seven_segment_display):

Submodule for 7-segment display. ○ **Slice LUTs**: 11 out of 63,400 (0.02%). ○ **Slice Registers**: 17 out of 126,800 (0.01%).

Power Report:



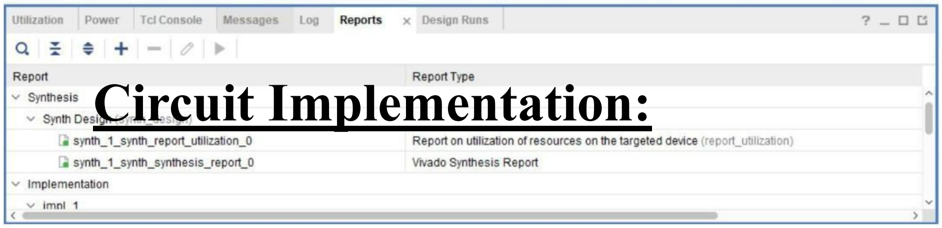
Power Report Explanation:

Target Device: xc7a100tcsg324-3 FPGA.
Results Name: power_1.
Device Settings:

- **Temp grade**: extended (default).
- **Process**: typical (default).

Environment Settings:

- **Output Load**: 0 pF (user-defined, range 0–10000 pF).
- **Junction Temperature**: 25.383 °C (calculated).
- **Ambient Temperature**: 25 °C (default).
- **Effective θJA**: 4.563 °C/W (calculated, range 0–100 °C/W).
- **Airflow**: 250 LFM (default, linear feet per minute).
- **Heat Sink**: medium (Medium Power, Thermal Solution) (default).
- **θSA**: 4.6 °C/W (default, range 0–100 °C/W).
- **Board Selection**: medium (10x10") (default).



Circuit Implementation:

Constraint File Code:

Constraints file for Nexys A7 XC7A100T-1CSG324C FPGA

Maps the MIPS processor design to the Nexys A7 board based on schematic

Clock signal (100 MHz)

set_property -dict { PACKAGE_PIN E3 IOSTANDARD LVCMOS33 } [get_ports { clk }]; # System clock create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports { clk }];

Reset signal (using CPU reset button)

set_property -dict { PACKAGE_PIN U9 IOSTANDARD LVCMOS33 } [get_ports { reset }]; # CPU Reset button (BTNRES)

LEDs (16 LEDs: LD0 to LD15)

set_property -dict { PACKAGE_PIN T8 IOSTANDARD LVCMOS33 } [get_ports { led[0] }]; # LD0

set_property -dict { PACKAGE_PIN V9 IOSTANDARD LVCMOS33 } [get_ports { led[1] }]; # LD1

set_property -dict { PACKAGE_PIN R8 IOSTANDARD LVCMOS33 } [get_ports { led[2] }]; # LD2

set_property -dict { PACKAGE_PIN T6 IOSTANDARD LVCMOS33 } [get_ports { led[3] }]; # LD3

set_property -dict { PACKAGE_PIN T5 IOSTANDARD LVCMOS33 } [get_ports { led[4] }]; # LD4

```
set_property -dict { PACKAGE_PIN T4   IOSTANDARD LVCMOS33 } [get_ports { led[5] }];
# LD5

set_property -dict { PACKAGE_PIN U7   IOSTANDARD LVCMOS33 } [get_ports { led[6] }];
# LD6

set_property -dict { PACKAGE_PIN U6   IOSTANDARD LVCMOS33 } [get_ports { led[7] }];
# LD7

set_property -dict { PACKAGE_PIN V4   IOSTANDARD LVCMOS33 } [get_ports { led[8] }];
# LD8

set_property -dict { PACKAGE_PIN U3   IOSTANDARD LVCMOS33 } [get_ports { led[9] }];
# LD9

set_property -dict { PACKAGE_PIN V1   IOSTANDARD LVCMOS33 } [get_ports { led[10] }];
# LD10

set_property -dict { PACKAGE_PIN R1   IOSTANDARD LVCMOS33 } [get_ports { led[11] }];
# LD11

set_property -dict { PACKAGE_PIN P5   IOSTANDARD LVCMOS33 } [get_ports { led[12] }];
# LD12

set_property -dict { PACKAGE_PIN U1   IOSTANDARD LVCMOS33 } [get_ports { led[13] }];
# LD13

set_property -dict { PACKAGE_PIN R2   IOSTANDARD LVCMOS33 } [get_ports { led[14] }];
# LD14
set_property -dict { PACKAGE_PIN P2   IOSTANDARD LVCMOS33 } [get_ports { led[15] }];
# LD15

## 7-Segment Display

## Segments (CA, CB, CC, CD, CE, CF, CG) - Active low

set_property -dict { PACKAGE_PIN L3   IOSTANDARD LVCMOS33 } [get_ports { seg[0] }];
# CA

set_property -dict { PACKAGE_PIN N1   IOSTANDARD LVCMOS33 } [get_ports { seg[1] }];
# CB

set_property -dict { PACKAGE_PIN L5   IOSTANDARD LVCMOS33 } [get_ports { seg[2] }];
# CC

set_property -dict { PACKAGE_PIN L4   IOSTANDARD LVCMOS33 } [get_ports { seg[3] }];
# CD
```

set_property -dict { PACKAGE_PIN K3   IOSTANDARD LVCMOS33 } [get_ports { seg[4] }]; # CE

set_property -dict { PACKAGE_PIN M2   IOSTANDARD LVCMOS33 } [get_ports { seg[5] }]; # CF

set_property -dict { PACKAGE_PIN L6   IOSTANDARD LVCMOS33 } [get_ports { seg[6] }]; # CG

Anodes (AN0 to AN3) - Active low

set_property -dict { PACKAGE_PIN M1   IOSTANDARD LVCMOS33 } [get_ports { an[0] }]; # AN0 (rightmost digit)

set_property -dict { PACKAGE_PIN L1   IOSTANDARD LVCMOS33 } [get_ports { an[1] }]; # AN1

set_property -dict { PACKAGE_PIN N4   IOSTANDARD LVCMOS33 } [get_ports { an[2] }]; # AN2

set_property -dict { PACKAGE_PIN N2   IOSTANDARD LVCMOS33 } [get_ports { an[3] }]; # AN3 (leftmost digit)

Timing Constraints

Ensure proper timing for the 7-segment display refresh

set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets clk_IBUF]
Additional Settings

set_property CFGBVS VCCO [current_design] set_property

CONFIG_VOLTAGE 3.3 [current_design]

Constraint File Code Explanation:

Purpose: Constraints file for mapping a MIPS processor design to a Nexys A7 FPGA (XC7A100T-1CSG324C).

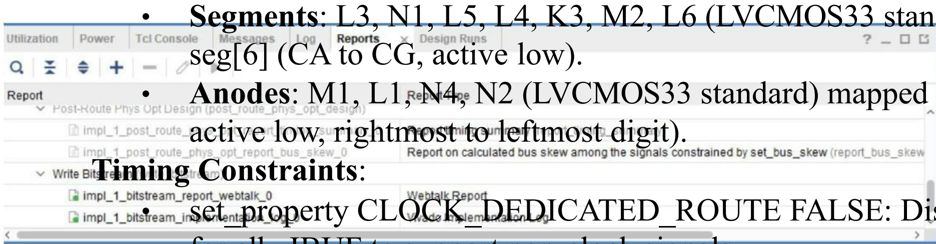
Clock Signal:

- **Pin**: E3, LVCMOS33 standard, mapped to clk (100 MHz system clock).
- **Constraint**: create_clock sets a 10 ns period (100 MHz) with a 50% duty cycle (0 to 5 ns).

Reset Signal:

- **Pin**: U9, LVCMOS33 standard, mapped to reset (CPU reset button, BTNRES). **LEDs**:
- **Pins**: T8, V9, R8, T6, T5, T4, U7, U6, V4, U3, V1, R1, P5, U1, R2, P2 (LVCMOS33 standard).

- **Mapping**: led[0] to led[15] correspond to LD0 to LD15 on the board. **7-Segment Display**:



- **Segments**: L3, N1, L5, L4, K3, M2, L6 (LVCMOS33 standard) mapped to seg[0] to seg[6] (CA to CG, active low).
- **Anodes**: M1, L1, N4, N2 (LVCMOS33 standard) mapped to an[0] to an[3] (AN0 to AN3, active low, rightmost to leftmost digit).

Timing Constraints:

- set_property CLOCK_DEDICATED_ROUTE FALSE: Disables dedicated clock routing for clk_IBUF to support non-clock signals.

Additional Settings:

- set_property CFGBVS VCCO: Sets configuration bank voltage to VCCO.
- set_property CONFIG_VOLTAGE 3.3: Sets configuration voltage to 3.3V.

Function: Defines pin assignments and timing for proper hardware implementation on the Nexys A7 board.

Bitstream Generation: