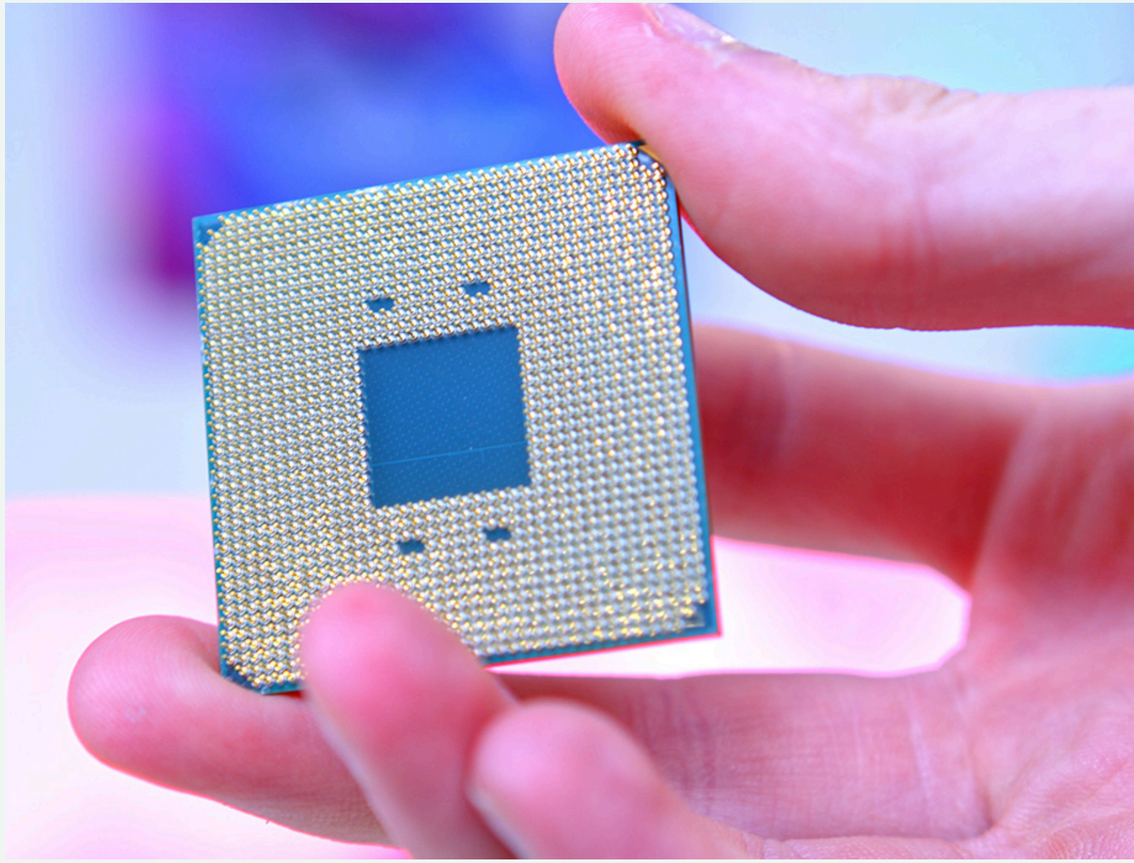
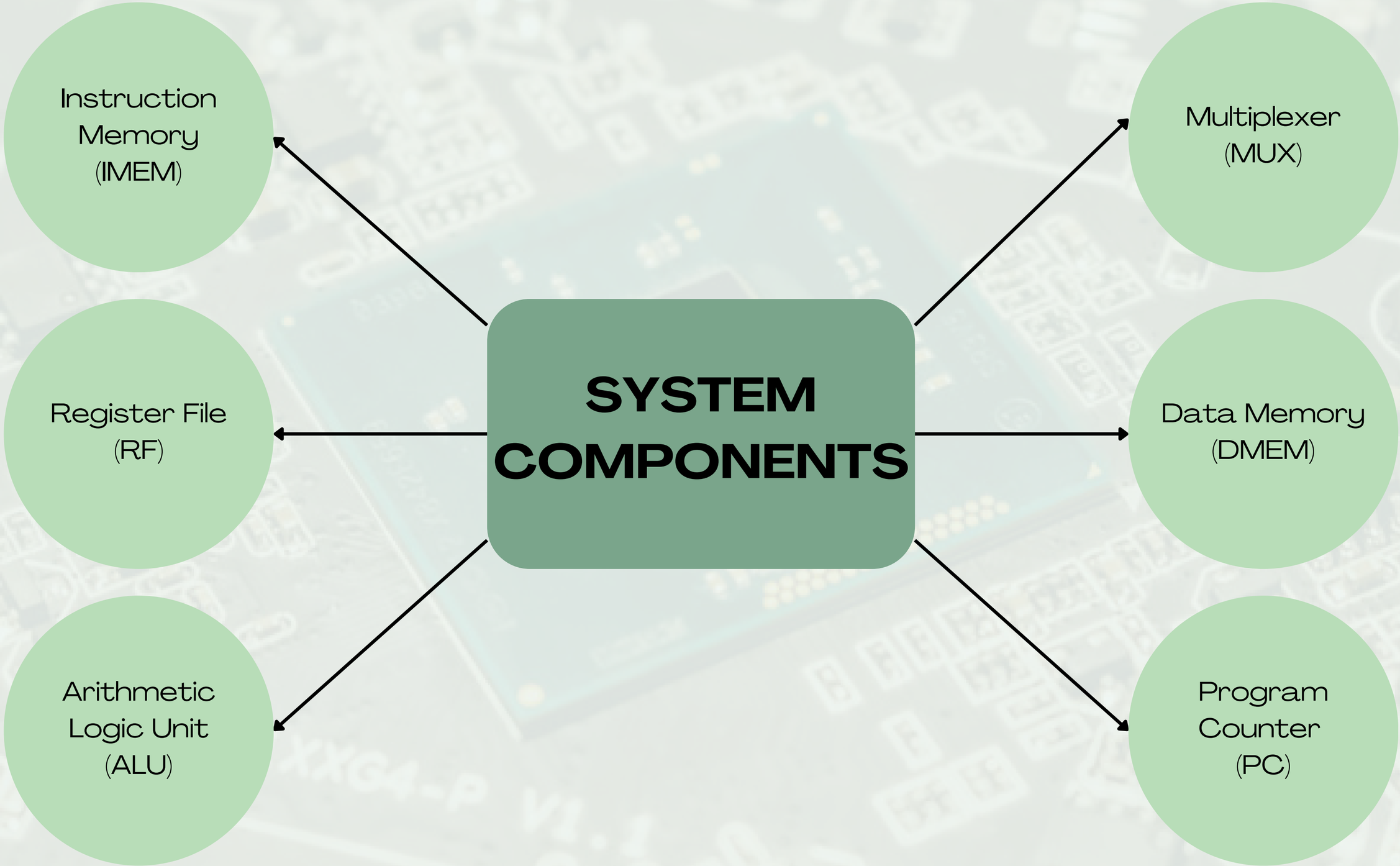
ADSD-RISC

# DESIGN OF A 16-BIT CPU


| | |
|---|---|
| Yeap Wei Shen | 20000375 |
| Tharshen A/L Subramaniam | 20001074 |
| Tang Jin Hang | 20001336 |


Instruction Memory (IMEM)
Register File (RF)
Arithmetic Logic Unit (ALU)
SYSTEM COMPONENTS
Multiplexer (MUX)
Data Memory (DMEM)
Program Counter (PC)

# READ ONLY MEMORY (IMEM)

## DESIGN

```
module IMEM (
  input [15:0] pc_output_to_imem_input,          // 16-bit address input
  input OE,                                      // Output Enable
  output reg [15:0] imem_output_to_rf_input     // 16-bit data output
);

    // Memory array: 512 words, each word is 16 bits (2 bytes)
    reg [15:0] rom [0:511];  // 512 words, each 16 bits

    // Initializing ROM with hardcoded binary values (simulation purposes)
    initial begin
        // Hardcode the ROM values for simulation (this replaces the file read)
        rom[0] = 16'b1010000101100010;  // ROM[0] = 0xA1B2 (little-endian)
        rom[1] = 16'b1100001111010100;  // ROM[1] = 0xC3D4 (little-endian)
        rom[2] = 16'b0101111101101110;  // ROM[2] = 0x5F7E (little-endian)
        rom[3] = 16'b0111110110001100;  // ROM[3] = 0x7D8C (little-endian)
      rom[4] = 16'b1010000101100010;  // ROM[0] = 0xA1B2 (little-endian)
      rom[5] = 16'b1100001111010100;  // ROM[1] = 0xC3D4 (little-endian)
      rom[6] = 16'b0101111101101110;  // ROM[2] = 0x5F7E (little-endian)
      rom[7] = 16'b0111110110001100;
        // ... continue initializing for all required memory addresses
        // Ensure to initialize all 512 entries as needed
    end

    // Always block to access memory
    always @ (pc_output_to_imem_input or OE) begin
        if (OE == 1) begin
            // Read 16-bit word from the given address (no need for byte combining)
            imem_output_to_rf_input = rom[pc_output_to_imem_input];
        end else begin
            // High impedance state when OE is 0
            imem_output_to_rf_input = 16'bz;
        end
    end

endmodule
```

## TESTBENCH

```
// Display header
$display("Time\taddr\t\t\t\tOE\tdata_out");

// Initialize the simulation signals
pc_output_to_imem_input = 16'b0;   // Start with address 0
OE = 0;                            // Initially disable Output Enable (OE = 0)

// Apply some test cases
#10;  // Wait for 10 time units

// Test Case 1: OE = 1, address 0 (Expect ROM[0] and ROM[1] combined as big-endian)
OE = 1;  // Enable output
pc_output_to_imem_input = 16'b0000000000000000; // Address 0
#10;  // Wait for 10 time units
$display("%0t\t%b\t\t%b\t%b", $time, pc_output_to_imem_input, OE, imem_output_to_rf_input);

// Test Case 2: OE = 1, address 1 (Expect ROM[1] and ROM[2] combined as big-endian)
pc_output_to_imem_input = 16'b0000000000000001; // Address 1
#10;
$display("%0t\t%b\t\t%b\t%b", $time, pc_output_to_imem_input, OE, imem_output_to_rf_input);
```

## RESULTS

| Time | addr | OE | data_out |
|---|---|---|---|
| 20 | 0000000000000000 | 1 | 1010000101100010 |
| 30 | 0000000000000001 | 1 | 1100001111010100 |
| 40 | 0000000000000010 | 1 | 0101111101101110 |
| 50 | 0000000000000011 | 0 | zzzzzzzzzzzzzzzz |

# REGISTER FILE (RF)

## DESIGN

```
module RF(rs,rt,rd,data_in,write_en,rst,clk,rs_data_out,rt_data_out,leds);
  input [3:0] rs,rt;          //4-bit source register
  input [3:0] rd;             //4-bit destination register
  input [15:0] data_in;       //16-bit data input
  input write_en, rst;        //1-bit control signal
  input clk;                  //clock signal

  output [7:0]leds;           //8-bit output for LED control at r15
  output [15:0] rs_data_out, rt_data_out;       //16-bit output for rs and rt

    // Declare 16 registers, each 16 bits wide
    reg [15:0] registers [15:0];

    // Asynchronous read for rs and rt
    assign rs_data_out = registers[rs];
    assign rt_data_out = registers[rt];



    // LEDs connected to the least significant 8 bits of register 15 (r15)
    assign leds = registers[15][7:0]; // Use r15[7:0] for controlling LEDs


    // Synchronous write operation
    always @(posedge clk or posedge rst) begin
      registers[0] <= 16'b0;          // set r0 is always 0
        if (rst) begin
            // Reset all registers to 0 when reset is high, except r0
            //registers[0] <= 16'b0;  // r0 is always zero no need to reset
            registers[1] <= 16'b0;
            registers[2] <= 16'b0;
            registers[3] <= 16'b0;
            registers[4] <= 16'b0;
            registers[5] <= 16'b0;
            registers[6] <= 16'b0;
            registers[7] <= 16'b0;
            registers[8] <= 16'b0;
            registers[9] <= 16'b0;
            registers[10] <= 16'b0;
            registers[11] <= 16'b0;
            registers[12] <= 16'b0;
            registers[13] <= 16'b0;
            registers[14] <= 16'b0;
            registers[15] <= 16'b0;
        end
        else if (write_en) begin
            // Write data to registers, except for r0 (which is read-only and always 0)
            if (rd != 4'b0000) begin
                registers[rd] <= data_in;  // Write data to register rd, unless it's r0
            end
            // Note: r0 will always stay 0, no matter what is written to it.
        end
    end
```

## TESTBENCH

```
// Initialize signals
clk = 0;
rst = 0;
write_en = 0;
rs = 0;
rt = 0;
rd = 0;
data_in = 0;

// Apply reset
#5 rst = 1;  // Assert reset
#10 rst = 0; // Deassert reset
$display("Testbench reset complete");

// Test 1: Write to a normal register (e.g., r1)
#10;
write_en = 1;         // Enable write
rd = 4'b0001;         // Select register r1
data_in = 16'b0001001000110100;  // Data to write (binary)
#10;                  // Wait for one clock cycle
write_en = 0;         // Disable write
$display("Test 1: Write to r1 complete");

// Test 2: Read from r1
#10;
rs = 4'b0001;         // Select register r1
rt = 4'b0001;         // Select register r0 (for testing)
#10;
```

## RESULTS

```
At time 0,      data_in=0000000000000000, rd=0000,  rs = 0000, rt = 0000, rs_data_out = xxxxxxxxxxxxxxxx, rt_data_out = xxxxxxxxxxxxxxxx, leds = xxxxxxxx
At time 5,      data_in=0000000000000000, rd=0000,  rs = 0000, rt = 0000, rs_data_out = 0000000000000000, rt_data_out = 0000000000000000, leds = 00000000
Testbench reset complete
At time 25,     data_in=0001001000110100, rd=0001,  rs = 0000, rt = 0000, rs_data_out = 0000000000000000, rt_data_out = 0000000000000000, leds = 00000000
Test 1: Write to r1 complete
At time 45,     data_in=0001001000110100, rd=0001,  rs = 0001, rt = 0001, rs_data_out = 0001001000110100, rt_data_out = 0001001000110100, leds = 00000000
Test 2: Read from r1 complete
```

# DATA MEMORY (DMEM)

## DESIGN

```
`timescale 1ns / 1ps
module dmem (
    input wire clk,                     // Clock input
    input wire [15:0] alu_output_to_dmem_addr,          // 16-bit address
    input wire [15:0] rf_output_rt_to_alu_input,       // 16-bit data_in
    input wire RW_,                    // Read/Write control (1 for read, 0 for write)
    input wire CS,                     // Chip select
    output reg [15:0] dmem_output_to_rf_input       // 16-bit data_out
);

    // Memory array (1KB = 1024 bytes = 512 16-bit words)
    // Implement as byte addressable but with 16-bit ports
  reg [15:0] memory [0:511];       // 1KB byte-addressable memory

    // Tri-state output control
    always @(*) begin
        if (!CS) begin
            // High impedance state when chip select is inactive
            dmem_output_to_rf_input  = 16'bz;
        end
        else if (RW_) begin
            // Read operation
          if (alu_output_to_dmem_addr < 511) begin
                dmem_output_to_rf_input = memory[alu_output_to_dmem_addr];
            end
            else begin
                // Address beyond memory space
                dmem_output_to_rf_input  = 16'b0;
            end
        end
        else begin
            // During write operation, output is high impedance
            dmem_output_to_rf_input  = 16'bz;
        end
    end

    // Write operation (synchronized to clock)
    always @(posedge clk) begin
        if (CS && !RW_ && alu_output_to_dmem_addr < 511) begin
            memory[alu_output_to_dmem_addr] = rf_output_rt_to_alu_input;
        end
    end
endmodule
```

## TESTBENCH

```
        // Test Case 1: Write to address 0x0000
        $display("Test Case 1: Write to address 0x0000");
        CS = 1'b1;
        RW_ = 1'b0;
        alu_output_to_dmem_addr = 16'h0000;
        rf_output_rt_to_alu_input = 16'hABCD;
        @(posedge clk);
        #1;  // Wait for write to complete

        // Test Case 2: Read from address 0x0000
        $display("Test Case 2: Read from address 0x0000");
        RW_ = 1'b1;
        #5;  // Wait for read to stabilize
        if (dmem_output_to_rf_input === 16'hABCD)
            $display("PASS: Read data matches written data");
        else
            $display("FAIL: Read data %h doesn't match
written data %h", dmem_output_to_rf_input, 16'hABCD);
```

## RESULTS

```
Test Case 1: Write to address 0x0000
Test Case 2: Read from address 0x0000
PASS: Read data matches written data
```

# PROGRAM COUNTER (PC)

## DESIGN

```
1  `timescale 1ns / 1ps
2
3  module pc(pc_load,
4            alu_output_to_rf_input,
5            reset,
6            clk,
7            pc_output_to_imem_input);
8    input pc_load, reset, clk;
9    input [15:0] alu_output_to_rf_input; //16 bits data_in
10   output reg [15:0] pc_output_to_imem_input; //16 bits data_out
11
12   always @(posedge clk or negedge reset)
13     begin
14       if (!reset) //Active low configuration
15         // Reset the output to zero
16         pc_output_to_imem_input <=16'b0;
17       else if (pc_load)
18         //Load data_in to data_out
19         pc_output_to_imem_input <= alu_output_to_rf_input;
20     end
21 endmodule
```

## TESTBENCH

```
// Task to check expected output
task check_output(input [15:0] expected);
  begin
    #1; // Small delay to allow output to settle
    if (pc_output_to_imem_input === expected)
      $display("TEST PASSED at time %0t | Expected: %d | Got: %d", $time, expected, pc_output_to_imem_input);
    else
      $display("TEST FAILED at time %0t | Expected: %d | Got: %d", $time, expected, pc_output_to_imem_input);
  end
endtask

initial begin
  // Initialize signals
  clk = 0; reset = 0; pc_load = 0;
  alu_output_to_rf_input = 16'd5;

  #15 reset = 1;  // Release reset
  #20 check_output(16'd0);  // Expecting 0 after reset

  // Test Case 1: Load value 5
  #20 pc_load = 1;
  #20 pc_load = 0;
  #10 check_output(16'd5); // Expecting 5 after loading

  // Test Case 2: Load new value 10
  #20 alu_output_to_rf_input = 16'd10;
  #20 pc_load = 1;
  #20 pc_load = 0;
  #10 check_output(16'd10); // Expecting 10 after loading
```

## RESULTS

```
Time = 56000 | Reset = 1 | PC Load = 1 | Input =     5 | PC Output =     0
Time = 70000 | Reset = 1 | PC Load = 1 | Input =     5 | PC Output =     5
Time = 76000 | Reset = 1 | PC Load = 0 | Input =     5 | PC Output =     5
TEST PASSED at time 87000 | Expected:     5 | Got:     5
```

```
Time = 107000 | Reset = 1 | PC Load = 0 | Input =    10 | PC Output =     5
Time = 127000 | Reset = 1 | PC Load = 1 | Input =    10 | PC Output =     5
Time = 130000 | Reset = 1 | PC Load = 1 | Input =    10 | PC Output =    10
Time = 147000 | Reset = 1 | PC Load = 0 | Input =    10 | PC Output =    10
TEST PASSED at time 158000 | Expected:    10 | Got:    10
```

# ARITHMETIC LOGIC UNIT (ALU)

## DESIGN

```
//`timescale 1ns / 1ps

module ALU(
    input [15:0] rf_output_rs_to_alu_input, // ALU gets rs from RF
    input [15:0] rf_output_rt_to_alu_input, // ALU gets rt from RF
    input [3:0] imem_output_to_rf_input_opcode, // Opcode from IMEM (instruction)
    output reg [15:0] alu_output_to_rf_input, // ALU result goes to RF
    output reg Zero, Negative, Overflow
);
    reg signed [15:0] signed_rs, signed_rt, signed_Result;

    always @(*) begin
        // Default values
        Zero = 0;
        Negative = 0;
        Overflow = 0;
        signed_rs <= rf_output_rs_to_alu_input;
        signed_rt <= rf_output_rt_to_alu_input;

        case (imem_output_to_rf_input_opcode)
            4'b0000: begin // ADD
                signed_Result = signed_rs + signed_rt;
                Overflow = ((signed_rs[15] == signed_rt[15]) && (signed_Result[15] != signed_rs[15]));
                alu_output_to_rf_input = signed_Result;
            end
            4'b0001: begin // SUB
                signed_Result = signed_rs - signed_rt;
                Overflow = ((signed_rs[15] != signed_rt[15]) && (signed_Result[15] != signed_rs[15]));
                alu_output_to_rf_input = signed_Result;
            end
            4'b0010: alu_output_to_rf_input = signed_rs | signed_rt; // OR
            4'b0011: alu_output_to_rf_input = signed_rs & signed_rt; // AND
            4'b0100: alu_output_to_rf_input = signed_rs <<< rf_output_rt_to_alu_input[3:0]; // SHL (Arithmetic Shift Left)
            4'b0101: alu_output_to_rf_input = $signed(rf_output_rs_to_alu_input) >>> rf_output_rt_to_alu_input[3:0]; // SHR (Arithmetic Shift Right)
            4'b0110: alu_output_to_rf_input = {rf_output_rs_to_alu_input[14:0], rf_output_rs_to_alu_input[15]}; // ROL (Rotate Left)
            4'b0111: alu_output_to_rf_input = {rf_output_rs_to_alu_input[0], rf_output_rs_to_alu_input[15:1]}; // ROR
            4'b1000: alu_output_to_rf_input = ~rf_output_rs_to_alu_input; // NOT
            4'b1111: alu_output_to_rf_input = signed_rs + {{12{rf_output_rt_to_alu_input[3]}}, rf_output_rt_to_alu_input[3:0]}; // ADDI (Sign-extended)
            default: alu_output_to_rf_input = 16'b0;
        endcase

        // Zero flag
        Zero = (alu_output_to_rf_input == 16'b0) ? 1'b1 : 1'b0;

        // Negative flag
        Negative = alu_output_to_rf_input[15];
    end
endmodule
```

| Opcode | Instruction | RTL Operation |
|---|---|---|
| R-type (Register Instructions) | | |
| 0000 | add, rs, rt, rd | rd <- rs + rt |
| 0001 | sub rs, rt, rd | rd <- rs - rt |
| 0010 | or rs, rt, rd | rd <- rs \| rt (bitwise OR) |
| 0011 | and rs, rt, rd | rd <- rs & rt (bitwise AND) |
| S-type (Shift Instructions) | | |
| 0100 | shl rs, rt, #imm4 | rt <- arithmetic shift left rs by #imm4 bits |
| 0101 | shr rs, rt, #imm4 | rt <- arithmetic shift right rs by #imm4 bits |
| 0110 | rol rs, rt | rt <- {rs[14:0], rs[15]} |
| 0111 | ror rs, rt | rt <- {rs[0], rs[15:1]} |
| 1000 | not rs, rt | rt <- ~rs (bitwise negation) |
| 1111 | addi rs, rt, #imm4 | rt <- rs + {12{imm4[3], imm4}} |

**Overflow:**

**(+A) + (+B) = -C (addition)**

**(+A) - (-B) = -C (subtraction)**

# ARITHMETIC LOGIC UNIT (ALU)

## TESTBENCH

```
// Test ADD (Overflow case)
rf_output_rs_to_alu_input = 16'h7FFF;
rf_output_rt_to_alu_input = 16'h0001;
imem_output_to_rf_input_opcode = 4'b0000; #10;

// Test SUB (Overflow case)
rf_output_rs_to_alu_input = 16'h8000;
rf_output_rt_to_alu_input = 16'h0001;
imem_output_to_rf_input_opcode = 4'b0001; #10;

// Test OR
rf_output_rs_to_alu_input = 16'h00F0;
rf_output_rt_to_alu_input = 16'h000F;
imem_output_to_rf_input_opcode = 4'b0010; #10;

// Test AND
rf_output_rs_to_alu_input = 16'hFFFF;
rf_output_rt_to_alu_input = 16'h0F0F;
imem_output_to_rf_input_opcode = 4'b0011; #10;

// Test Shift Left
rf_output_rs_to_alu_input = 16'h0001;
rf_output_rt_to_alu_input = 16'h0004;
imem_output_to_rf_input_opcode = 4'b0100; #10;

// Test Shift Right
rf_output_rs_to_alu_input = 16'h8000;
rf_output_rt_to_alu_input = 16'h0004;
imem_output_to_rf_input_opcode = 4'b0101; #10;
```

```
// Test Rotate Left
rf_output_rs_to_alu_input = 16'h8001;
imem_output_to_rf_input_opcode = 4'b0110; #10;

// Test Rotate Right
rf_output_rs_to_alu_input = 16'h8001;
imem_output_to_rf_input_opcode = 4'b0111; #10;

// Test NOT
rf_output_rs_to_alu_input = 16'hFFFF;
imem_output_to_rf_input_opcode = 4'b1000; #10;

// Test ADDI (Sign-extended immediate addition)
rf_output_rs_to_alu_input = 16'h0005;
rf_output_rt_to_alu_input = 16'h000F;
imem_output_to_rf_input_opcode = 4'b1111; #10;

// Extra Tests:
// Test ADD (Negative result case)
rf_output_rs_to_alu_input = 16'b0000000000000000;
rf_output_rt_to_alu_input = 16'b1000000000000000;
imem_output_to_rf_input_opcode = 4'b0000; #10;

// Test SUB (Negative result case)
rf_output_rs_to_alu_input = 16'b1111111111111111;
rf_output_rt_to_alu_input = 16'b0111111111111111;
imem_output_to_rf_input_opcode = 4'b0001; #10;
```

| Opcode | Instruction | RTL Operation |
|---|---|---|
| R-type (Register Instructions) | | |
| 0000 | add, rs, rt, rd | rd <- rs + rt |
| 0001 | sub rs, rt, rd | rd <- rs - rt |
| 0010 | or rs, rt, rd | rd <- rs \| rt (bitwise OR) |
| 0011 | and rs, rt, rd | rd <- rs & rt (bitwise AND) |
| S-type (Shift Instructions) | | |
| 0100 | shl rs, rt, #imm4 | rt <- arithmetic shift left rs by #imm4 bits |
| 0101 | shr rs, rt, #imm4 | rt <- arithmetic shift right rs by #imm4 bits |
| 0110 | rol rs, rt | rt <- {rs[14:0], rs[15]} |
| 0111 | ror rs, rt | rt <- {rs[0], rs[15:1]} |
| 1000 | not rs, rt | rt <- ~rs (bitwise negation) |
| 1111 | addi rs, rt, #imm4 | rt <- rs + {12{imm4[3], imm4}} |

## RESULTS

```
Time = 00000 | rs = 0111111111111111 ( 32767) | rt = 0000000000000001 (     1) | Opcode = 0000 | Result = 1000000000000000 (-32768) | Zero = 0 | Negative = 1 | Overflow = 1
Time = 05000 | rs = 1000000000000000 (-32768) | rt = 0000000000000001 (     1) | Opcode = 0001 | Result = 0111111111111111 ( 32767) | Zero = 0 | Negative = 0 | Overflow = 1
Time = 10000 | rs = 0000000011110000 (   240) | rt = 0000000000001111 (    15) | Opcode = 0010 | Result = 0000000011111111 (   255) | Zero = 0 | Negative = 0 | Overflow = 0
Time = 15000 | rs = 1111111111111111 (    -1) | rt = 0000111100001111 (  3855) | Opcode = 0011 | Result = 0000111100001111 (  3855) | Zero = 0 | Negative = 0 | Overflow = 0
Time = 20000 | rs = 0000000000000001 (     1) | rt = 0000000000000100 (     4) | Opcode = 0100 | Result = 0000000000010000 (    16) | Zero = 0 | Negative = 0 | Overflow = 0
Time = 25000 | rs = 1000000000000000 (-32768) | rt = 0000000000000100 (     4) | Opcode = 0101 | Result = 1111100000000000 ( -2048) | Zero = 0 | Negative = 1 | Overflow = 0
Time = 30000 | rs = 1000000000000001 (-32767) | rt = 0000000000000100 (     4) | Opcode = 0110 | Result = 0000000000000011 (     3) | Zero = 0 | Negative = 0 | Overflow = 0
Time = 35000 | rs = 1000000000000001 (-32767) | rt = 0000000000000100 (     4) | Opcode = 0111 | Result = 1100000000000000 (-16384) | Zero = 0 | Negative = 1 | Overflow = 0
Time = 40000 | rs = 1111111111111111 (    -1) | rt = 0000000000000100 (     4) | Opcode = 1000 | Result = 0000000000000000 (     0) | Zero = 1 | Negative = 0 | Overflow = 0
Time = 45000 | rs = 0000000000000101 (     5) | rt = 0000000000001111 (    15) | Opcode = 1111 | Result = 0000000000000100 (     4) | Zero = 0 | Negative = 0 | Overflow = 0
Time = 50000 | rs = 0000000000000000 (     0) | rt = 1000000000000000 (-32768) | Opcode = 0000 | Result = 1000000000000000 (-32768) | Zero = 0 | Negative = 1 | Overflow = 0
Time = 55000 | rs = 1111111111111111 (    -1) | rt = 0111111111111111 ( 32767) | Opcode = 0001 | Result = 1000000000000000 (-32768) | Zero = 0 | Negative = 1 | Overflow = 0
```

# 2 TO 1 MULTIPLEXERS (MUX)

## DESIGN

```
//`timescale 1ns / 1ps
module mux_4bit_2to1 (
    input [3:0] I0,     // First 4-bit input
    input [3:0] I1,     // Second 4-bit input
    input S,            // Select signal
    output reg [3:0] Y // 4-bit output
);
    always @(*) begin
        if (S == 0)
            Y = I0;     // Select I0 when S=0
        else
            Y = I1;     // Select I1 when S=1
    end
endmodule

module mux_16bit_2to1 (
    input [15:0] I0,     // First 16-bit input
    input [15:0] I1,     // Second 16-bit input
    input S,            // Select signal
    output reg [15:0] Y // 16-bit output
);
    always @(*) begin
        if (S == 0)
            Y = I0;     // Select I0 when S=0
        else
            Y = I1;     // Select I1 when S=1
    end
endmodule

```

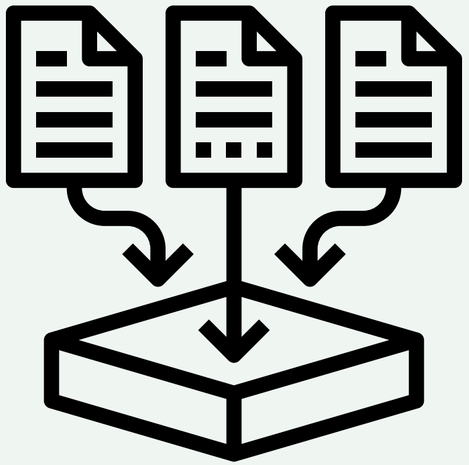
## TESTBENCH

```
// Test case 1: S=0, should select I0
I0_4bit = 4'b0001; I1_4bit = 4'b1110; S_4bit = 0;
I0_16bit = 16'h1234; I1_16bit = 16'h4321; S_16bit = 0;
#10;

// Test case 2: S=1, should select I1
S_4bit = 1; S_16bit = 1;
#10;
```

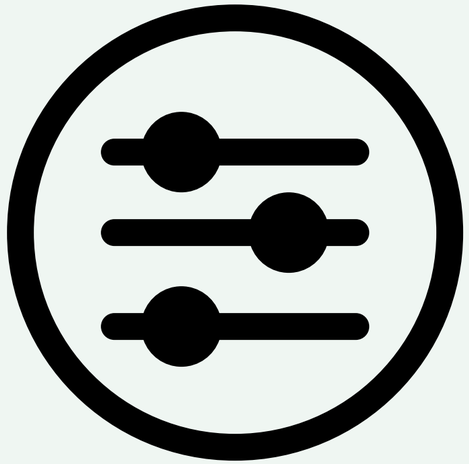
## RESULTS

```
Time = 00000
4-bit MUX -> S = 0 | I0 = 0001 | I1 = 1110 | Y = 0001
16-bit MUX -> S = 0 | I0 = 1234 | I1 = 4321 | Y = 1234


Time = 10000
4-bit MUX -> S = 1 | I0 = 0001 | I1 = 1110 | Y = 1110
16-bit MUX -> S = 1 | I0 = 1234 | I1 = 4321 | Y = 4321
```

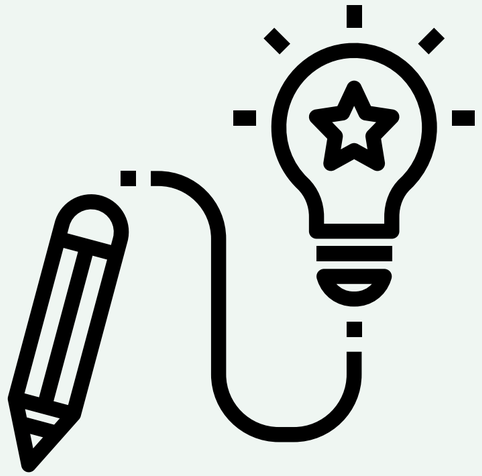
# SYSTEM INTEGRATION



Datapath



Controller



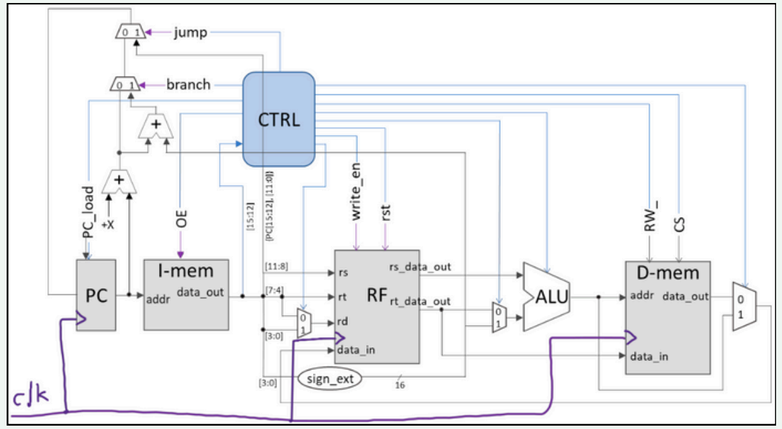
Top Level

# DATAPATH

## DESIGN

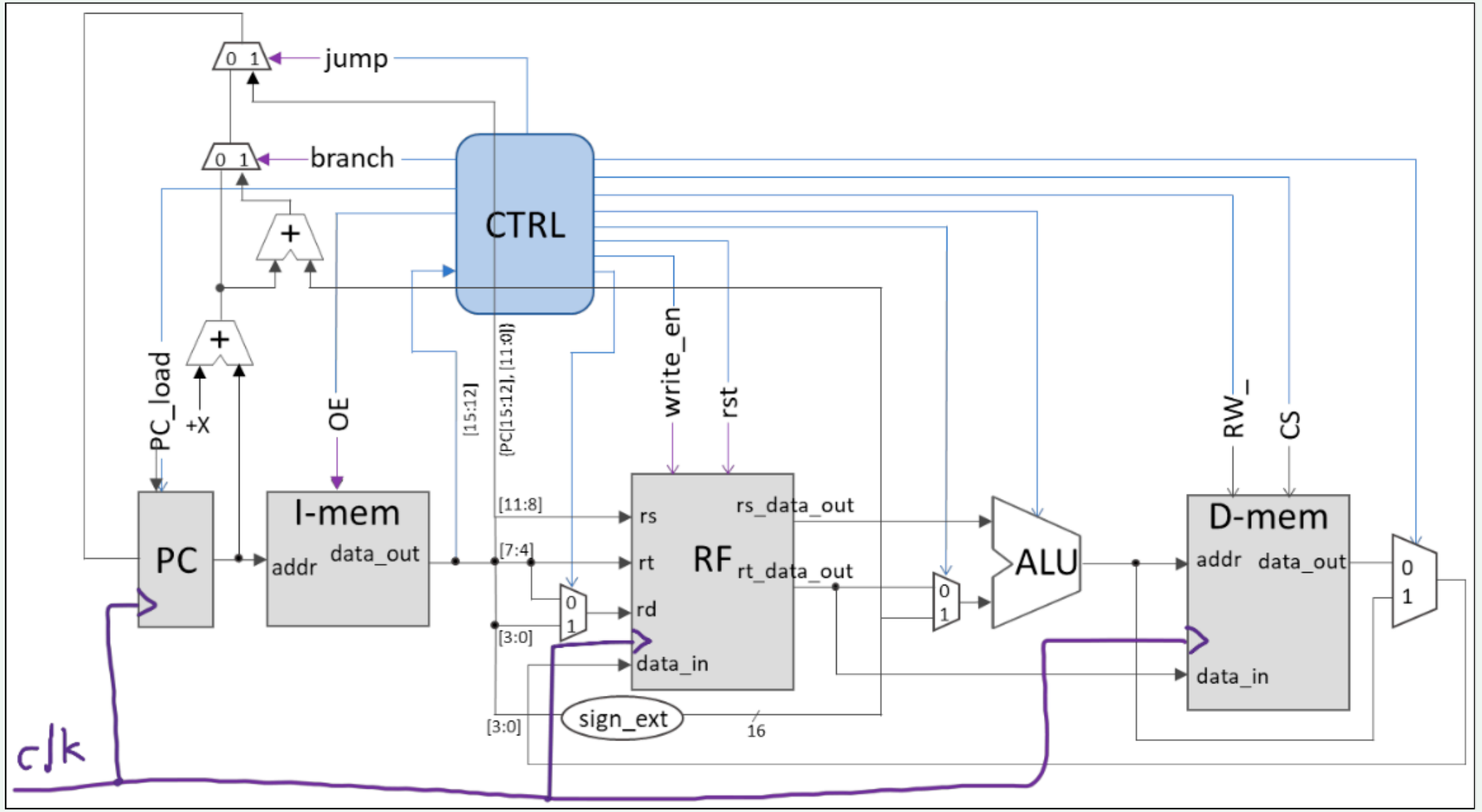
```
//`timescale 1ns / 1ps
module Datapath(
    input clk,
    input reset,
    input pc_load,
    input write_en,
    input RW_,
    input CS,
    input OE,
    input ctrl_output_to_mux_rf_selected_rd_input,
    input ctrl_output_to_mux_alu_selected_rt_input,
    input ctrl_output_to_mux_rf_selected_data_in_input,
    input [3:0] ctrl_output_to_alu_input_opcode,
    input ctrl_output_to_mux_branch,
    input ctrl_output_to_mux_jump,
    output [3:0] imem_output_to_control_input_opcode,
    output Zero,
    output Negative,
    output Overflow,
    output [7:0] LEDs
);

    wire [15:0] pc_output_to_imem_input;
    wire [15:0] imem_output_instruction_register_IR;
    wire [15:0] rf_output_rs_to_alu_input;
    wire [15:0] rf_output_rt_to_alu_input;
    wire [15:0] alu_output_to_rf_input;
    wire [15:0] dmem_output_to_rf_input;
    wire [15:0] imem_output_to_alu_input;
    wire [15:0] alu_output_to_dmem_addr;
    wire [3:0] rf_selected_rd;
    wire [15:0] alu_selected_rt, rf_selected_data_in;
    wire [15:0] mux_branch_output_to_mux_jump_input;
    wire [15:0] mux_jump_output_to_pc_input;

    assign imem_output_to_alu_input =
    {{12{imem_output_instruction_register_IR[3]}},
    imem_output_instruction_register_IR[3:0]};

    assign imem_output_to_control_input_opcode =
    imem_output_instruction_register_IR[15:12];

    shortint addition_pc_output_plus_2,
    addition_pc_output_plus_2_plus_imem_output_to_alu_input;

    assign addition_pc_output_plus_2 = pc_output_to_imem_input + 16'b01;

    assign addition_pc_output_plus_2_plus_imem_output_to_alu_input =
    addition_pc_output_plus_2 + imem_output_to_alu_input;

    // Instantiate Program Counter (PC)
    pc PC (
        .pc_load(pc_load),
        .alu_output_to_rf_input(mux_jump_output_to_pc_input),
        .reset(reset),
        .clk(clk),
        .pc_output_to_imem_input(pc_output_to_imem_input)
    );

    // Instantiate Instruction Memory (IMEM)
    IMEM ROM (
        .pc_output_to_imem_input(pc_output_to_imem_input),
        .OE(OE),
        .imem_output_to_rf_input(imem_output_instruction_register_IR)
    );

    // Instantiate Register File (RF)
    RF RegisterFile (
        .imem_output_to_rf_input_rs(imem_output_instruction_register_IR[11:8]), // rs register
        .imem_output_to_rf_input_rt(imem_output_instruction_register_IR[7:4]),  // rt register
        .imem_output_to_rf_input_rd(rf_selected_rd),         // rd register
        .alu_output_to_rf_input(rf_selected_data_in),   // Data to RF from DMEM
        .write_en(write_en),
        .rst(reset),
        .clk(clk),
        .rf_output_rs_to_alu_input(rf_output_rs_to_alu_input),
        .rf_output_rt_to_alu_input(rf_output_rt_to_alu_input),
        .leds(LEDs)
    );

    // Instantiate ALU
    ALU alu (
        .rf_output_rs_to_alu_input(rf_output_rs_to_alu_input),
        .rf_output_rt_to_alu_input(alu_selected_rt),
        .imem_output_to_rf_input_opcode(ctrl_output_to_alu_input_opcode),
        .alu_output_to_rf_input(alu_output_to_rf_input),
        .Zero(Zero),
        .Negative(Negative),
        .Overflow(Overflow)
    );

    // Instantiate Data Memory (DMEM)
    dmem DataMemory (
        .clk(clk),
        .alu_output_to_dmem_addr(alu_output_to_rf_input),  // Address from ALU
        .rf_output_rt_to_alu_input(rf_output_rt_to_alu_input),  // Data input
        .RW_(RW_),  // Read/Write control
        .CS(CS),  // Chip Select
        .dmem_output_to_rf_input(dmem_output_to_rf_input) // Data out to RF
    );

    mux_4bit_2to1 mux_rf_selected_rd (
        .I0(imem_output_instruction_register_IR[7:4]),      // Option 0: Bits [7:4]
        .I1(imem_output_instruction_register_IR[3:0]),      // Option 1: Bits [3:0]
        .S(ctrl_output_to_mux_rf_selected_rd_input),    // Select signal
        .Y(rf_selected_rd)                              // Output of MUX
    );

    mux_16bit_2to1 mux_alu_selected_rt (
        .I0(rf_output_rt_to_alu_input),
        .I1(imem_output_to_alu_input),
        .S(ctrl_output_to_mux_alu_selected_rt_input),
        .Y(alu_selected_rt)
    );

    mux_16bit_2to1 mux_rf_selected_data_in (
        .I0(dmem_output_to_rf_input),
        .I1(alu_output_to_rf_input),
        .S(ctrl_output_to_mux_rf_selected_data_in_input),
        .Y(rf_selected_data_in)
    );

    mux_16bit_2to1 mux_branch (
        .I0(addition_pc_output_plus_2),
        .I1(addition_pc_output_plus_2_plus_imem_output_to_alu_input),
        .S(ctrl_output_to_mux_branch),
        .Y(mux_branch_output_to_mux_jump_input)
    );

    mux_16bit_2to1 mux_jump (
        .I0(mux_branch_output_to_mux_jump_input),
        .I1({{4{imem_output_instruction_register_IR[11]}},
         imem_output_instruction_register_IR[11:0]}),
        .S(ctrl_output_to_mux_jump),
        .Y(mux_jump_output_to_pc_input)
    );

endmodule
```

## BLOCK DIAGRAM



# DATAPATH

## BLOCK DIAGRAM



# CONTROLLER

## DESIGN

```
module controller (
    input clk,
    input rst,
    input wire [3:0] opcode,   // 4-bit opcode from instruction
    input Zero,
    input Negative,
    input Overflow,
    output reg       pc_load,
    output reg       i_mem_oe,
    output reg       rf_mux_sel, //mux before going into rf
    output reg       rf_write_en, //write_en for rf
    output reg       alu_mux_sel, //mux before going into alu
    output reg [3:0] alu_opcode,     // ALU operation selection
    output reg       d_mem_rw_,  //rw_ for dmem
    output reg       d_mem_cs,  //cs for dmem
    output reg       data_out_mux, //data_out from dmem into mux
    output reg       branch,     // Branch signal
    output reg       jump        // Jump signal
);

always @(*) begin
    // Default values
  if (!rst) begin
    pc_load = 1'b0;
    i_mem_oe = 1'b0;
    rf_mux_sel = 1'b1;
    rf_write_en = 1'b0;
    alu_mux_sel = 1'b0;
    alu_opcode = 4'b0000;
    d_mem_rw_ = 1'b0;
    d_mem_cs = 1'b0;
    data_out_mux = 1'b1;
    branch = 1'b0;
    jump = 1'b0;
  end

  else begin
    pc_load = 1'b0;
    i_mem_oe = 1'b1;
    rf_mux_sel = 1'b1;
    rf_write_en = 1'b0;
    alu_mux_sel = 1'b0;
    alu_opcode = 4'b0000;
    d_mem_rw_ = 1'b0;
    d_mem_cs = 1'b0;
    data_out_mux = 1'b1;
    branch = 1'b0;
    jump = 1'b0;
```

## R-TYPE

```
        // R-type instructions (add, sub, or, and)
        4'b0000: begin //Add
            pc_load = 1'b1;
            i_mem_oe = 1'b1;
            rf_mux_sel = 1'b1;
            rf_write_en = 1'b1;
            alu_mux_sel = 1'b0;
            alu_opcode = 4'b0000;
            d_mem_rw_ = 1'b0;
            d_mem_cs = 1'b0;
            data_out_mux = 1'b1;
            branch = 1'b0;
            jump = 1'b0;
        end

        4'b0001: begin //Sub
            pc_load = 1'b1;
            i_mem_oe = 1'b1;
            rf_mux_sel = 1'b1;
            rf_write_en = 1'b1;
            alu_mux_sel = 1'b0;
            alu_opcode = 4'b0001;
            d_mem_rw_ = 1'b0;
            d_mem_cs = 1'b0;
            data_out_mux = 1'b1;
            branch = 1'b0;
            jump = 1'b0;
        end

        4'b0010: begin //Or
            pc_load = 1'b1;
            i_mem_oe = 1'b1;
            rf_mux_sel = 1'b1;
            rf_write_en = 1'b1;
            alu_mux_sel = 1'b0;
            alu_opcode = 4'b0010;
            d_mem_rw_ = 1'b0;
            d_mem_cs = 1'b0;
            data_out_mux = 1'b1;
            branch = 1'b0;
            jump = 1'b0;
```

## S-TYPE

```
// S-type (shift, rotate, NOT, addi)
4'b0100: begin //shift left
    pc_load = 1'b1;
    i_mem_oe = 1'b1;
    rf_mux_sel = 1'b0;
    rf_write_en = 1'b1;
    alu_mux_sel = 1'b1;
    alu_opcode = 4'b0100;
    d_mem_rw_ = 1'b0;
    d_mem_cs = 1'b0;
    data_out_mux = 1'b1;
    branch = 1'b0;
    jump = 1'b0;
end

4'b0101: begin //shift right
    pc_load = 1'b1;
    i_mem_oe = 1'b1;
    rf_mux_sel = 1'b0;
    rf_write_en = 1'b1;
    alu_mux_sel = 1'b1;
    alu_opcode = 4'b0101;
    d_mem_rw_ = 1'b0;
    d_mem_cs = 1'b0;
    data_out_mux = 1'b1;
    branch = 1'b0;
    jump = 1'b0;
end

4'b0110: begin //rotate left
    pc_load = 1'b1;
    i_mem_oe = 1'b1;
    rf_mux_sel = 1'b0;
    rf_write_en = 1'b1;
    alu_mux_sel = 1'b1;
    alu_opcode = 4'b0110;
    d_mem_rw_ = 1'b0;
    d_mem_cs = 1'b0;
    data_out_mux = 1'b1;
    branch = 1'b0;
    jump = 1'b0;
end
```

# CONTROLLER

## B-TYPE

```
        4'b1001: begin //Beq
            pc_load = 1'b1;
            i_mem_oe = 1'b1;
            rf_mux_sel = 1'b0;
            rf_write_en = 1'b0;
            alu_mux_sel = 1'b0;
            alu_opcode = 4'b0001;
            d_mem_rw_ = 1'b0;
            d_mem_cs = 1'b0;
            data_out_mux = 1'b0;
//              branch = 1'b0;
            jump = 1'b0;
          if (Zero) branch = 1'b1;
        end

        4'b1010: begin //Blt
            pc_load = 1'b1;
            i_mem_oe = 1'b1;
            rf_mux_sel = 1'b0;
            rf_write_en = 1'b0;
            alu_mux_sel = 1'b0;
            alu_opcode = 4'b0001;
            d_mem_rw_ = 1'b0;
            d_mem_cs = 1'b0;
            data_out_mux = 1'b0;
//              branch = 1'b0;
            jump = 1'b0;
          if (Negative) branch = 1'b1;
        end

        4'b1011: begin //Bgt
            pc_load = 1'b1;
            i_mem_oe = 1'b1;
            rf_mux_sel = 1'b0;
            rf_write_en = 1'b0;
            alu_mux_sel = 1'b0;
            alu_opcode = 4'b0001;
            d_mem_rw_ = 1'b0;
            d_mem_cs = 1'b0;
            data_out_mux = 1'b0;
//              branch = 1'b0;
            jump = 1'b0;
          if (!Zero && !Negative) branch = 1'b1;
        end
```

## L-TYPE

```
// L-type (Load and Store)
4'b1100: begin //Ld
    pc_load = 1'b1;
    i_mem_oe = 1'b1;
    rf_mux_sel = 1'b0;
    rf_write_en = 1'b1;
    alu_mux_sel = 1'b1;
    alu_opcode = 4'b0000;
    d_mem_rw_ = 1'b1; //read
    d_mem_cs = 1'b1;
    data_out_mux = 1'b0;
    branch = 1'b0;
    jump = 1'b0;
end

4'b1101: begin //St
    pc_load = 1'b1;
    i_mem_oe = 1'b1;
    rf_mux_sel = 1'bx; //dont care
    rf_write_en = 1'b0;
    alu_mux_sel = 1'b1;
    alu_opcode = 4'b0000;
    d_mem_rw_ = 1'b0; //write
    d_mem_cs = 1'b1;
    data_out_mux = 1'bx; //dont care
    branch = 1'b0;
    jump = 1'b0;
end
```

## J-TYPE

```
// J-type (Jump)
4'b1110: begin
    pc_load = 1'b1;
    i_mem_oe = 1'b1;
    rf_mux_sel = 1'bx; //dont care
    rf_write_en = 1'b0;
    alu_mux_sel = 1'b0;
    alu_opcode = 4'b0000;
    d_mem_rw_ = 1'b0;
    d_mem_cs = 1'b0;
    data_out_mux = 1'bx; //dont care
    branch = 1'b0;
    jump = 1'b1;
end
```

# TOP LEVEL DESIGN

## DESIGN

```
`timescale 1ns / 1ps
`include "control.sv"
`include "datapath.sv"
`include "alu.sv"
`include "dmem.sv"
`include "mux.sv"
`include "pc.sv"
`include "rom.sv"
`include "rf.sv"

module TopLevel (
    input logic clk,           // Clock input
    input logic reset,         // Reset input
    output logic [7:0] LEDs    // LEDs output from RF module (for debugging or status)
);

    // Internal signals for connecting controller and datapath
    logic [3:0] opcode;  // Opcode fetched from IMEM
    logic Zero, Negative, Overflow;

    // Control signals from controller
    logic pc_load;
    logic i_mem_oe;
    logic rf_mux_sel;
    logic rf_write_en;
    logic alu_mux_sel;
    logic [3:0] alu_opcode;
    logic d_mem_rw_;
    logic d_mem_cs;
    logic data_out_mux;
    logic branch;
    logic jump;

    // Additional control signals for datapath
    logic ctrl_output_to_mux_rf_selected_rd_input;
    logic ctrl_output_to_mux_alu_selected_rt_input;
    logic ctrl_output_to_mux_rf_selected_data_in_input;
    logic [3:0] ctrl_output_to_alu_input_opcode;
```

```
    // Instantiate the datapath module
    Datapath dp_inst (
        .clk(clk),
        .reset(reset),
        .pc_load(pc_load),                          // Connected to controller
        .write_en(rf_write_en),                     // Connected to controller
        .RW_(d_mem_rw_),                            // Connected to controller
        .CS(d_mem_cs),                              // Connected to controller
        .OE(i_mem_oe),                              // Connected to controller
        .ctrl_output_to_mux_rf_selected_rd_input(rf_mux_sel), // Connected to controller
        .ctrl_output_to_mux_alu_selected_rt_input(alu_mux_sel), // Connected to controller
        .ctrl_output_to_mux_rf_selected_data_in_input(data_out_mux), // Connected to controller
        .ctrl_output_to_alu_input_opcode(alu_opcode), // Connected to controller
        .ctrl_output_to_mux_branch(branch),  // Connected to controller
        .ctrl_output_to_mux_jump(jump),      // Connected to controller
        .imem_output_to_control_input_opcode(opcode),  // Opcode output to controller
        .Zero(Zero),                                // Zero flag to controller
        .Negative(Negative),                            // Negative flag to controller
        .Overflow(Overflow),                            // Overflow flag to controller
        .LEDs(LEDs)                                 // Debugging output
    );

    // Instantiate the controller module
    controller ctrl_inst (
        .clk(clk),
        .rst(reset),
        .opcode(opcode),   // Opcode comes from IMEM in Datapath
        .Zero(Zero),                                // Zero flag to controller
        .Negative(Negative),                            // Negative flag to controller
        .Overflow(Overflow),
        .pc_load(pc_load),
        .i_mem_oe(i_mem_oe),
        .rf_mux_sel(rf_mux_sel),
        .rf_write_en(rf_write_en),
        .alu_mux_sel(alu_mux_sel),
        .alu_opcode(alu_opcode),
        .d_mem_rw_(d_mem_rw_),
        .d_mem_cs(d_mem_cs),
        .data_out_mux(data_out_mux),
        .branch(branch),
        .jump(jump)
    );

endmodule
```

# SYSTEM VERIFICATION

## TESTBENCH

```
`timescale 1ns / 1ps

module TopLevel_tb;

    // Testbench signals
    logic clk;
    logic reset;
    logic [7:0] LEDs;

    // Instantiate the TopLevel module
    TopLevel uut (
        .clk(clk),
        .reset(reset),
        .LEDs(LEDs)
    );

    // Clock generation
    always begin
        #5 clk = ~clk;  // Clock period of 10ns (100MHz)
    end

    // Stimulus for the testbench
    initial begin
        // Initialize clock and reset
        clk = 0;
        reset = 0;

        // Apply reset
        #10 reset = 0;  // Apply reset
        #10 reset = 1;  // Release reset

        $display("Starting test...");

        // Wait for execution to proceed
        #200;

        // Monitor output
        $display("Final LED Output: %b", LEDs);

        // End of test
        $finish;
    end
```

```
    // Monitor function to display values of important signals at every time step
    initial begin
        $monitor("Time = %t | clk = %b | reset = %b | opcode = %b | LEDs = %b | PC = %d | R1 = %b | R2 =
%b | R5 = %b | R15 = %b| addr = %b| data_in = %b | RW_ = %b| CS = %b | data_out = %b |mux_output = %b",
                 $time, clk, reset, uut.dp_inst.imem_output_to_control_input_opcode, LEDs,
                 uut.dp_inst.PC.pc_output_to_imem_input,
                 uut.dp_inst.RegisterFile.registers[1], uut.dp_inst.RegisterFile.registers[2],
                 uut.dp_inst.RegisterFile.registers[5],
                 uut.dp_inst.RegisterFile.registers[15],
                 uut.dp_inst.DataMemory.alu_output_to_dmem_addr,
                 uut.dp_inst.DataMemory.rf_output_rt_to_alu_input,
                 uut.dp_inst.DataMemory.RW_,
                 uut.dp_inst.DataMemory.CS,
                 uut.dp_inst.DataMemory.dmem_output_to_rf_input,
                 uut.dp_inst.mux_rf_selected_data_in.Y
                );
    end

//  initial begin
//    $dumpfile("dump.vcd");
//    $dumpvars;
//  end

endmodule
```

## RESULTS

```
Starting test...
Time =              20000 | clk = 0 | reset = 1 | opcode = 1111 | LEDs = 00000000 | PC =     0 | R1 = 0000000000000000 | R2 = 0000000000000000 | R5 = 0000000000000000 | R15 = 0000000000000000|
Time =              25000 | clk = 1 | reset = 1 | opcode = 1111 | LEDs = 00000000 | PC =     1 | R1 = 0000000000000000 | R2 = 0000000000000000 | R5 = 0000000000000001 | R15 = 0000000000000000|

Time =              55000 | clk = 1 | reset = 1 | opcode = 1001 | LEDs = 00000111 | PC =     4 | R1 = 0000000000000111 | R2 = 0000000000000111 | R5 = 0000000000000001 | R15 = 0000000000000111|
Time =              60000 | clk = 0 | reset = 1 | opcode = 1001 | LEDs = 00000111 | PC =     4 | R1 = 0000000000000111 | R2 = 0000000000000111 | R5 = 0000000000000001 | R15 = 0000000000000111|
Time =              65000 | clk = 1 | reset = 1 | opcode = 0001 | LEDs = 00000111 | PC =     5 | R1 = 0000000000000111 | R2 = 0000000000000111 | R5 = 0000000000000001 | R15 = 0000000000000111|

Time =              95000 | clk = 1 | reset = 1 | opcode = 0000 | LEDs = 00000111 | PC =     3 | R1 = 0000000000000110 | R2 = 0000000000001110 | R5 = 0000000000000001 | R15 = 0000000000000111|
Time =             100000 | clk = 0 | reset = 1 | opcode = 0000 | LEDs = 00000111 | PC =     3 | R1 = 0000000000000110 | R2 = 0000000000001110 | R5 = 0000000000000001 | R15 = 0000000000000111|
Time =             105000 | clk = 1 | reset = 1 | opcode = 1001 | LEDs = 00001110 | PC =     4 | R1 = 0000000000000110 | R2 = 0000000000001110 | R5 = 0000000000000001 | R15 = 0000000000001110|
Time =             110000 | clk = 0 | reset = 1 | opcode = 1001 | LEDs = 00001110 | PC =     4 | R1 = 0000000000000110 | R2 = 0000000000001110 | R5 = 0000000000000001 | R15 = 0000000000001110|
```

# RESULTS (QUARTUS 2) PIN ASSIGNMENT

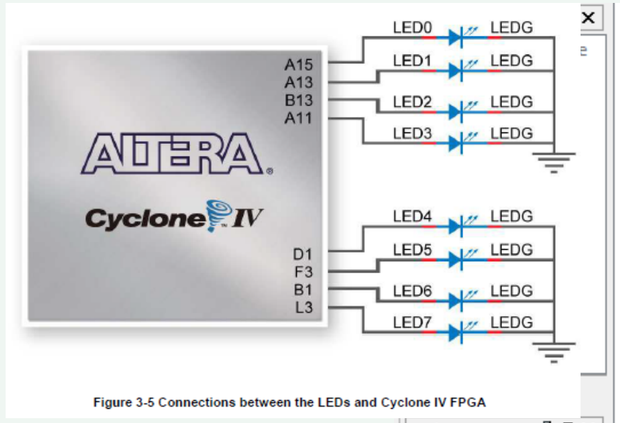
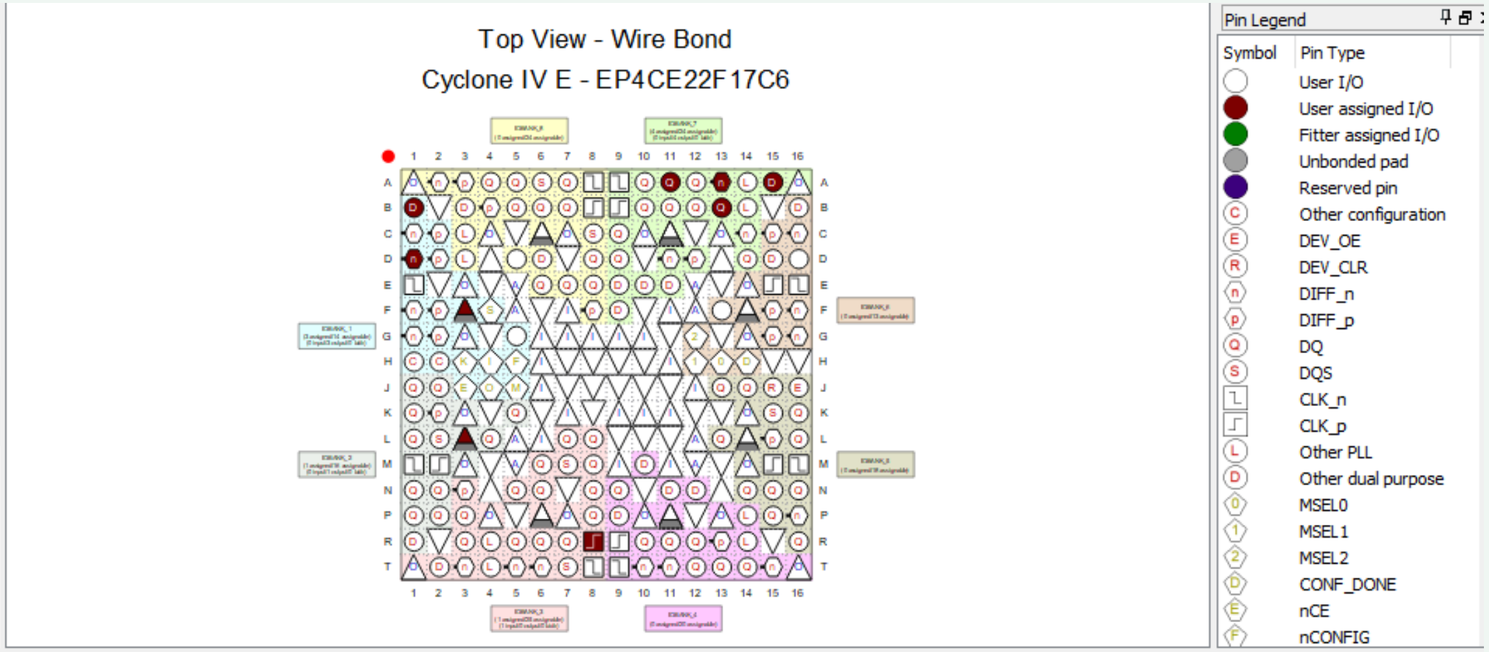
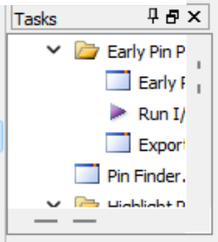



Figure 3-5 Connections between the LEDs and Cyclone IV FPGA









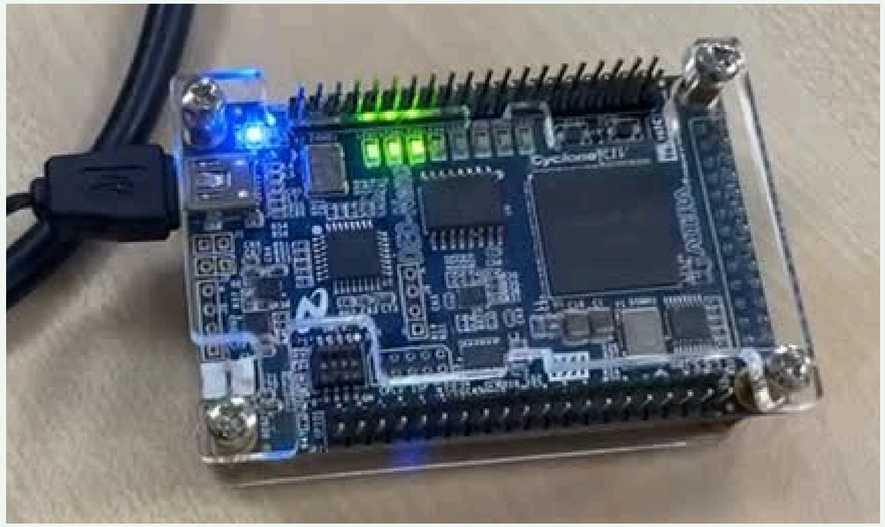

Named: * Edit: Filter: Pins: all

| Node Name | Direction | Location | I/O Bank | VREF Group | Fitter Location | I/O Standard | Reserved | Current Strength | Slew Rate |
|---|---|---|---|---|---|---|---|---|---|
| LEDs[7] | Output | PIN_L3 | 2 | B2_N0 | PIN_L3 | 3.3-V LVTTL | | 8mA (default) | 2 (default) |
| LEDs[6] | Output | PIN_B1 | 1 | B1_N0 | PIN_B1 | 3.3-V LVTTL | | 8mA (default) | 2 (default) |
| LEDs[5] | Output | PIN_F3 | 1 | B1_N0 | PIN_F3 | 3.3-V LVTTL | | 8mA (default) | 2 (default) |
| LEDs[4] | Output | PIN_D1 | 1 | B1_N0 | PIN_D1 | 3.3-V LVTTL | | 8mA (default) | 2 (default) |
| LEDs[3] | Output | PIN_A11 | 7 | B7_N0 | PIN_A11 | 3.3-V LVTTL | | 8mA (default) | 2 (default) |
| LEDs[2] | Output | PIN_B13 | 7 | B7_N0 | PIN_B13 | 3.3-V LVTTL | | 8mA (default) | 2 (default) |
| LEDs[1] | Output | PIN_A13 | 7 | B7_N0 | PIN_A13 | 3.3-V LVTTL | | 8mA (default) | 2 (default) |
| LEDs[0] | Output | PIN_A15 | 7 | B7_N0 | PIN_A15 | 3.3-V LVTTL | | 8mA (default) | 2 (default) |
| clk | Input | PIN_R8 | 3 | B3_N0 | PIN_R8 | 3.3-V LVTTL | | 8mA (default) | |
| reset | Input | | | | PIN_E1 | 3.3-V LVTTL | | 8mA (default) | |

# RESULTS & DEMONSTRATION

## ROM 1 Values

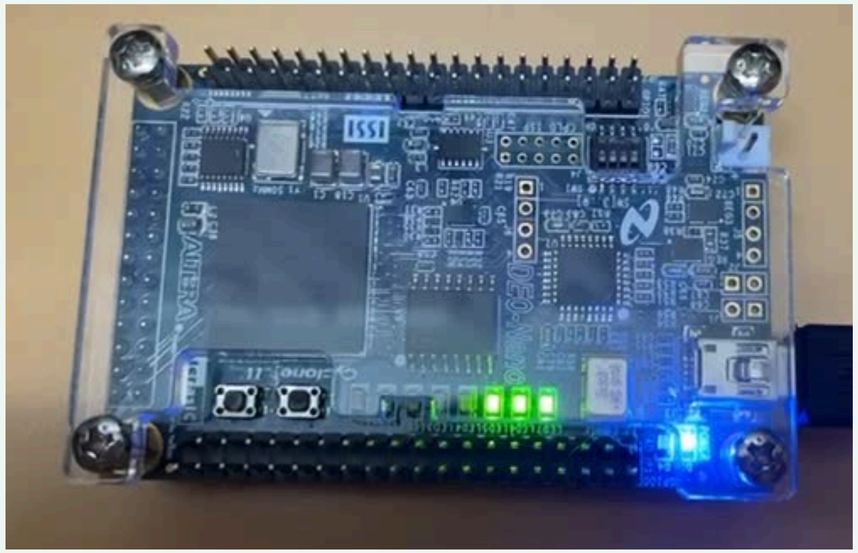
```
    // initialize the rom to a known assembler program
initial
  begin
    rom[0]  <= 16'b1111_0000_0101_0001;  // addi r0, r5 , 0001 //r5=1, to be used for decrementing r1
    rom[1]  <= 16'b1111_0000_0010_0111;  // addi r0, r2 , 0101 //r2=111 for LED display pattern
    rom[2]  <= 16'b1111_0000_0001_0111;  // addi r0, r1 , 0111 //r1=7
    rom[3]  <= 16'b0000_0000_0010_1111;  // add r0, r2 , r15   //r15=r2 write to LED port
    rom[4]  <= 16'b1001_0000_0001_1100;  // beq r0, r1 , -4    //if (r1==0), goto 2
    rom[5]  <= 16'b0001_0001_0101_0001;  // sub r1, r5, r1     //r1=r1-1
    rom[6]  <= 16'b0100_0010_0010_0001;  // shl r2, r2, 1      //r2<<1
    rom[7]  <= 16'b1110_0000_0000_0011;  // jmp 3    //if (r1==0), goto 2
    rom[8]  <= 16'b0000_0000_0000_0000;  // unused
    rom[9]  <= 16'b0000_0000_0000_0000;
    rom[10] <= 16'b0000_0000_0000_0000;
    rom[11] <= 16'b1110_0000_0000_0000;  // jump to 0 in case it ever comes here.
    rom[12] <= 16'b0000_0000_0000_0000;
    rom[13] <= 16'b0000_0000_0000_0000;
    rom[14] <= 16'b0000_0000_0000_0000;
    rom[15] <= 16'b0000_0000_0000_0000;
  end
```



# RESULTS & DEMONSTRATION

## ROM 2 Values

```
    // initialize the rom to a known assembler program
initial
  begin
    rom[0]  = 16'b1111_0000_0101_0001;  // addi r0, r5 , 0001  //r5=1, for decrementing r1
    rom[1]  = 16'b1000_0000_1110_xxxx;  // not r14, r0 , xxxx  //r14=0xffff, for LED display
    rom[2]  = 16'b1111_0000_0010_0111;  // addi r0, r2 , 0111  //r2=111, for LED display
    rom[3]  = 16'b1111_0000_0001_0111;  // addi r0, r1 , 0111  //r1=7, count=7
    rom[4]  = 16'b0011_1110_0010_1111;  // and r14, r2 , r15   //r15=r2 write to LED port using AND
    rom[5]  = 16'b0000_0000_0000_0000;  // sw     //
    rom[6]  = 16'b1001_0000_0001_0100;  // beq r0, r1 , 4      //if (r1==0), goto 2
    rom[7]  = 16'b0001_0001_0101_0001;  // sub r1, r5, r1      //r1--
    rom[8]  = 16'b0100_0010_0010_0001;  // shl r2, r2, 1       //r2<<1
    rom[9]  = 16'b1110_0000_0000_0100;  // jmp 4                //goto 2
    rom[10] = 16'b0000_0000_0000_0000;  // unused
    rom[11] = 16'b1111_0000_0011_0110;  // addi r0, r3, 0110   //r3=6, count = 7
    rom[12] = 16'b0010_0000_0010_1111;  // or r0, r2, r15      // write to LED port using OR
    rom[13] = 16'b1011_0001_0011_0011;  // bgt r1, r3, 3
    rom[14] = 16'b0000_0001_0101_0001;  // add r1, r5, r1      // r1++
    rom[15] = 16'b0101_0010_0010_0001;  // shr r2, r2, 1       //r2>>1
    rom[16] = 16'b1110_0000_0000_1100;  // jmp 12
    rom[17] = 16'b1110_0000_0000_0011;  // jmp 3
    rom[18] = 16'b0000_0000_0000_0000;  // unused
end
```



Terima Kasih

¡Muchas GRACIAS!

# ARIGATOU GOZAIMASU

THANK YOU VERY MUCH