CPU Project Final Report

ELEC 374: Digital Systems Engineering

Group 15

XXXXXXXX - XXXXXXXX

Julien Chagnon - 20390465

April 2nd, 2026

Abstract

This report presents the design, simulation, and FPGA implementation of a 32-bit single-bus RISC processor developed across four incremental phases as part of the ELEC 374 Digital Systems Engineering laboratory project at Queen's University. The processor was designed entirely in Verilog using Quartus Prime and functionally verified in ModelSim before being deployed on the DE0-CV development board. The datapath is built around a shared 32-bit single-bus architecture with sixteen general-purpose registers, dedicated HI/LO registers for multiply/divide results, a synchronous 512×32-word RAM, and a Booth radix-4 multiplier and non-restoring divider implemented as purely combinational blocks. The control unit is a hardwired Moore FSM that automatically sequences all fetch, decode, and instruction-specific execution states.

Table of Contents

Abstract 1

1. Project Specification 3

2. Project Design and Implementation 4

Datapath 4

3. Evaluation Results 6

3.1. Maximum frequency of operation in the simulation and on the FPGA chip 6

3.2. Average Cycle Per Instruction 6

3.3. Percentage of chip area used in design 7

4. Discussion 8

5. Conclusion and future work 9

Appendix I - Printout of the Schematic i

Appendix II - Functional simulation results for Phase 4 ii

Appendix III - Printout of the contents of memory for Phase 4 iii

Appendix IV – Pictures of FPGA board showing the displays/switches when running program in Phase 4 iv

Appendix V - Final Verilog code v

1. Project Specification

The objective of this project was to design and implement a simplified Reduced Instruction Set Computer (RISC) processor. The processor was required to support a basic instruction set, a single bus datapath architecture, memory access capabilities, and input/output operations. The design process involved progressively developing the processor through multiple phases, beginning with datapath construction and culminating in full hardware implementation on an FPGA board.

The Mini SRC processor architecture consists of a 32-bit datapath connected by a shared system bus. The datapath includes sixteen 32-bit general-purpose registers that store operands and intermediate values during program execution. In addition to the general-purpose registers, the processor contains several specialized registers including the program counter, instruction register, memory address register and memory data register. Temporary registers are used to support arithmetic operations within the ALU and HI and LO registers are used to store the results of multiplication and division operations. The bus architecture allows data to be transferred between components, with only one transaction occurring on the bus at a time.

The processor's arithmetic logic unit (ALU) was required to support a range of arithmetic and logical operations including addition, subtraction, logical AND and OR, shift and rotation operations, negation and bitwise NOT. The architecture also required support for multiplication and division instructions, with the results stored in the dedicated HI and LO registers. These operations allow the processor to perform the computations required by the instruction set.

The processor was also required to interface with a memory subsystem consisting of RAM, MAR and MDR to manage memory access. Memory operations include instructions such as load, load immediate and store which allow data to be transferred between memory and the processor registers. To support these instructions the architecture uses the MAR to specify memory locations and the MDR to temporarily hold data being transferred between memory and the processor.

Control flow instructions were also included in the specification to allow programs to alter execution order. Conditional branch instructions enable the processor to make decisions based on register values, allowing branching when conditions such as zero, non-zero, positive or negative values are met. This functionality allows the processor to execute loops and conditional statements.

In addition to computation and memory access, the Mini SRC processor was required to support input and output operations. Input and output ports allow the processor to interact

with external hardware devices, enabling data to be read from external inputs and written to output devices. This functionality is necessary for demonstrating processor operation on the FPGA board.

Finally, the processor design was required to be verified through simulation and implemented on the FPGA board. The FPGA implementation allows the processor to execute a provided test program while interacting with physical hardware components such as switches, LEDs and seven segment displays, demonstrating correct processor operation.

2. Project Design and Implementation

Datapath

The starting point for the datapath was the required single-bus structure, where all major components share one internal bus and only one source can drive that bus at a time. We first built the register and bus system, including the sixteen general-purpose registers and the main special-purpose registers such as the PC, IR, Y, Zhigh, Zlow, HI, and LO. The bus multiplexer was then used so that any enabled source could place its value onto the bus while destination registers captured that value on the clock edge. This served as our base structure for register transfers and instruction execution.

The arithmetic path was then added by connecting the ALU to the datapath through the Y and Z registers. One operand is first stored in Y, the second operand comes from the bus, and the ALU result is captured in Zlow or in Zhigh and Zlow for 64-bit results. HI and LO were added to store multiplication and division outputs. The next step was building the memory subsystem which was integrated using MAR, MDR, and RAM so that load and store instructions could be handled through the same single bus datapath.

As the project phases progressed, the datapath was upgraded so it could operate from real instruction words rather than manually setting register values. A select_encode block was added so the register fields inside the instruction register could be used to choose Ra, Rb, and Rc, and those fields were then decoded into the correct register input and output lines. don’t use sign extension path. Immediate values from the instruction register were also sign-extended so they could be used correctly by immediate, load/store, and branch instructions. Input and output registers were added to the datapath so it could interface with the FPGA board. By the final phase, the datapath was fully built so the processor supported register operations, ALU execution, memory access, HI/LO transfers, branching support, and I/O within the required single-bus structure.

Control Unit

The control unit was implemented in Phase 3, after the datapath had been built enough to

execute complete micro-operations correctly, this allowed the team to bring together and automate the features developed in Phases 1 and 2. In the earlier phases, the datapath registers had been initialized by manually asserting control signals in testbenches, but Phase 3 required the CPU to fetch, decode, and execute instructions automatically from memory. To meet that requirement, the control unit was implemented as a finite state machine using Method 1 from the phase 3 documentation that was provided. Using this approach, each state directly drives the required datapath control signals, which made the instruction sequencing easier to debug.

The FSM begins with a shared fetch sequence and then moves into instruction-specific execution states. In the implemented design, fetch moves the PC into MAR, increments the PC through the ALU and Z path, reads the instruction through MDR, and loads the instruction into IR. Because of the memory and IR update on clock edges, a separate decode state was added before execution begins. This allowed the opcode to be decoded only after the fetched instruction had been loaded into IR. Once the IR opcode is decoded, the control unit branches into the correct sequence for arithmetic, immediate, memory, branch, jump, multiply/divide, or input/output instructions.

Arithmetic and logical instructions load one operand into Y, use the second operand from the bus, store the ALU result in Zlow, and write it back into the destination register. Immediate instructions use the same pattern but take the second operand from the sign-extended constant path. Load and store instructions first compute an effective address, then complete the transfer through MAR, MDR, and RAM. Branch instructions use the CON FF block to evaluate the required condition and decide whether the PC should be updated. Multiply and divide write their 64-bit results into Zhigh and Zlow and then into HI and LO, while mfhi and mflo move those values back to the register file. jal required a special case because it always writes the return address into R12, so that register was driven directly in that state. Reset and halt were also included in the FSM, so reset clears the processor and returns it to the starting state, while halt or stop places it in a halted state with Run low.

Final Implementation

After the datapath and control unit were completed, they were connected with a top level CPU wrapped, this was essentially the last step in the main design so that the processor could be treated as a single module So this wrapper brought the datapath and control unit together into one CPU module and made the main external signals available, including clock, reset, stop, input data, output data, and the run signal. That integration step allowed the CPU to be tested by simply applying reset and clock and then letting the machine-code program execute from the preloaded instructions in RAM.

In the final phase, the CPU was connected to the DE0-CV board through a separate board-level top module. This module maps the FPGA clock to the CPU, the push-buttons to reset and stop, the lower slide switches to the input port, an LED to the run signal, and the low byte of the output port to the seven-segment displays. The RAM image was also updated with the supplied Phase 3 and Phase 4 programs and required data values. So the teams final CPU had automatic FSM-based control and a fully integrated processor running complete programs in simulation and on FPGA hardware.

3. Evaluation Results

3.1. Maximum frequency of operation in the simulation and on the FPGA chip

All Verilog testbenches across phases 1-4 use a clock period of 20ns, corresponding to a simulation clock of 50MHz. The declaration line is seen below:

```
forever #10 Clock = ~Clock;
```

As for the FPGA chip, the Phase 4 top-level file instantiates a clock_divider module that divides the board's oscillator by 25, in other words toggling the output every 25 input cycles. Since the DE0-CV uses a 50MHz clock (CLOCK_50), and the fact a complete output cycles requires two toggles for a clock cycle, it is calculated that the divider produces a CPU clock of 1MHz.

$$F_{cpu} = \frac{50\,MHz}{(2\,x\,25)} = 1\,MHz$$

Consulting output file CPU_Project.sta.rpt, Quartus Prime static timing analysis at the slow 1100mV 85°C worst-case corner reports a maximum achievable frequency of 8.24 MHz on the CPU clock domain (clock_divider:clkdiv_u|clk_out) and 142.61 MHz on the board reference clock (CLOCK_50). Since the design operates at 1 MHz on the board, all timing constraints are met with substantial margin.

3.2. Average Cycle Per Instruction

The per-instruction cycle count is seen in the table below, note that all instructions share a common five state fetch-decode cycle (FETCH0-FETCH3 + DECODE).

Instruction Class	**Instructions**	**Cycles**
Register move / port	MFHI, MFLO, IN, OUT, NOP	6
Jump	JR	6
Halt	HALT	6
Unary ALU	NEG, NOT	7

Jump and Link	JAL	7
Branch (Not taken)	BR	7
Branch (Taken)	BR	9
Binary ALU	ADD, SUB, AND, OR, SHR, SHRA, SHL, ROR, ROL, ADDI, ANDI, ORI, LDI	8
Multiply/Divide	MUL, DIV	9
Store	ST	10
Load	LD	11

Branch timing differs because BR first evaluates the condition in BR4 through CONout, then only performs PC update steps if the condition is true (See control_unit.v). If false, control returns directly to fetch in BR5 (7 total cycles). If true, the CPU must execute two extra operations to form and write the target address (PC + C) in BR6 and BR7 (9 total cycles).

Cycle counts are also high due to single-bus and synchronous RAM design, which adds clocked latency, so instruction fetch and data memory access require extra capture states.

3.3. Percentage of chip area used in design

All figures are from Quartus Prime output files after compilation. A summary table with data taken from CPU_Project.fit.summary showing top-level totals is included below.

Resource	Used	Available	Utilisation
Logic (ALMs)	2 859	18 480	15%
Dedicated registers	943	36 960	3%
IO pins	39	224	17%
Block RAM	2	308	<1%

The design occupies 15% of the device's logic, leaving ample headroom. The largest consumer of logic is the ALU at 3819 of the 4367 total combinational look-up tables in the design, driven mainly by the combinational divider (1286) and Booth radix-4 multiplier (1183).

The 943 total registers include 512 flip-flops for the sixteen 32-bit general-purpose registers, 320 flip-flops for the remaining datapath registers (PC, IR, HI, LO, Y, Zlow, Zhigh, InPort, OutPort), and 81 for the control unit FSM and enable signals.

4. Discussion

By working in parallel on different modules, collaborating on harder implementations, and retrofitting older modules such as the ALU and datapath to ensure functionality with newer modules at each phase, our mini SRC CPU was able to reach full operation. Our design employs a single shared 32 bit bus through which all register-register, register-ALU, and memory data transfers are routed. This architecture minimizes complexity as the bus multiplexer can be implemented as a large OR of individual signals. This comes at the cost of performance however, as every instruction requires multiple bus cycles to stage operands through the Y register before computing a result and writing it back.

The longest combinational path in the datapath and a dominant contributor to the frequency ceiling of the CPU clock is the project's 32 bit adder.v implemented as a ripple carry adder. In the worst case (0x7FFFFFFF + 0x00000001), the carry propagation must pass through all 32 stages. In hindsight, a carry lookahead or carry select adder would significantly reduce carry propagation delay.

The ALU also includes a Booth radix-4 multiplier and a non-restoring divider, both implemented as purely combinational blocks. This means the full 64-bit product or 32 bit quotient and remainder are computed within a single clock cycle. The trade-off is silicon area, as the divider alone accounts for 1286 combinational look-up tables and the multiplier a further 1183, making them by far the largest component in the design.

As for physical FPGA implementation, this project employs the DE0-CV device 5CEBA4F23C7 with top-level entity cpu_de0_top.v. Which connects the CPU to the physical IO of the DE0-CV. It instantiates a clock divider that scales the board's 50MHz onboard oscillator down to 1MHz. The eight onboard slide switches are routed to the CPU's input port, and the output port drives the red LEDs and the two rightmost seven-segment displays. The design uses only 15% of the device's available ALMs and less than 1% of its block RAM, so the Cyclone V is well within capacity for the full CPU.

One of the main challenges during the project was moving from manually controlled instruction testbenches to a fully automated CPU. In the earlier phases, the modules could be tested by manually asserting the required control signals for each instruction, which made easy to test and verify individual instruction performance. The more difficult part was turning that same behavior into a control unit that could generate all of those signals automatically in the correct order while also handling instruction fetch, decode, and memory timing.

We also had difficulty in Phase 4 when moving from our simulations on Quartus to the FPGA board. Up until that point, everything had been tested in simulation, so this was our first time working with the design on actual hardware. We had challenges both with the board-level code and with connecting to the FPGA itself. We had trouble getting the USB-Blaster to connect properly, and the compilation process to generate the .sof file was taking a long time and had been stopping well before it completed. We overcame the connection issue by adjusting some security settings on our laptops so the drivers could work properly, and the .sof generation naturally worked out as we spent more time on Phase 4 and became more comfortable with the tools and the hardware requirements.

5. Conclusion and future work

This project was a valuable experience in designing, testing, and implementing a processor across multiple phases. It brought together concepts from datapath design, control logic, simulation, and FPGA hardware integration, and completing the project through the final phase was also a nice achievment as only a small number of teams fully completed the project.

One possible improvement would be to use assembler-generated memory files directly instead of manually encoding the sample programs machine-code values into RA

Appendix I - Printout of the Schematic

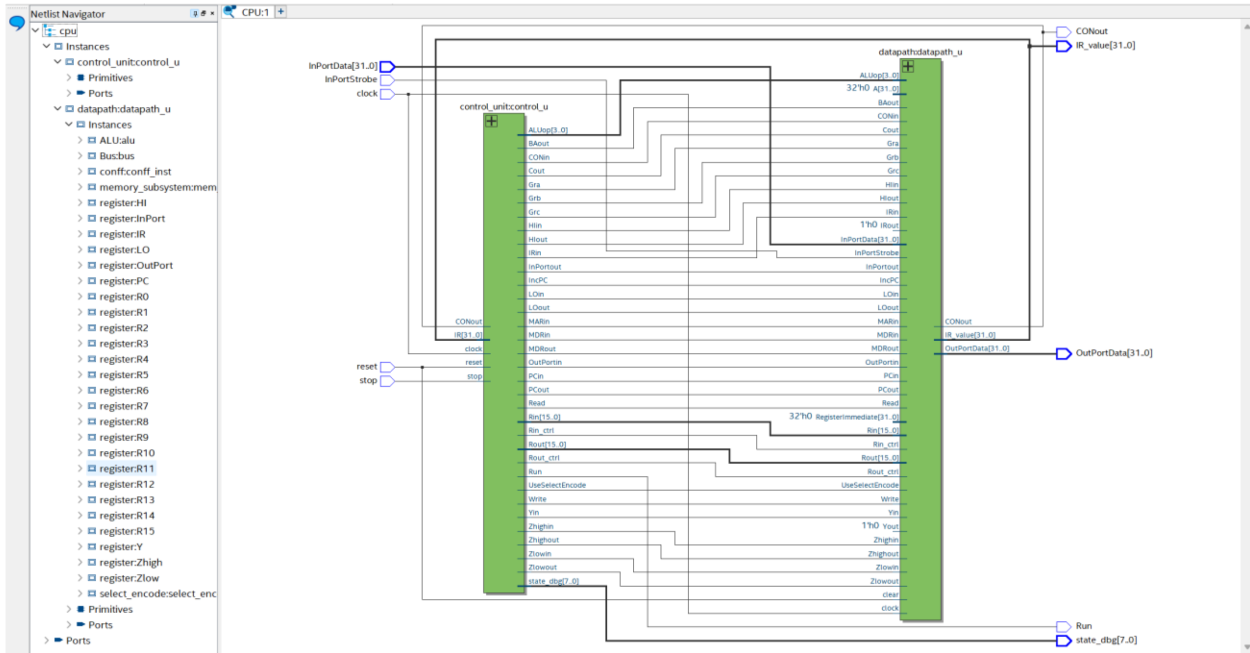

Figure 1 Final schematic

Appendix II - Functional simulation results for Phase 4

Functional simulation was used to verify the correct operation of the Mini SRC CPU after integrating the datapath, control unit and I/O functionality. The simulation was performed using the CPU testbench to ensure that instructions were fetched, decoded and executed correctly according to the provided test program. All register updates, control signals, memory operation and the behaviour of the processor during loop execution were verified. The following figures show waveform outputs from the simulation. These waveforms illustrate the final register values after execution of the test program and demonstrate the correct operation of the loop and control flow instructions implemented in the processor.

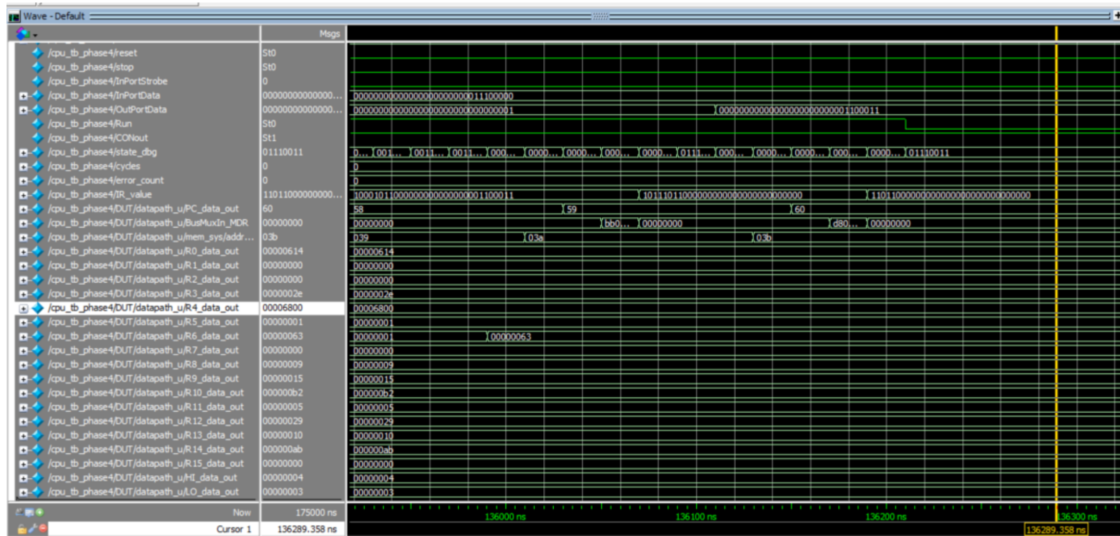

Figure 2 Final values for Phase 4

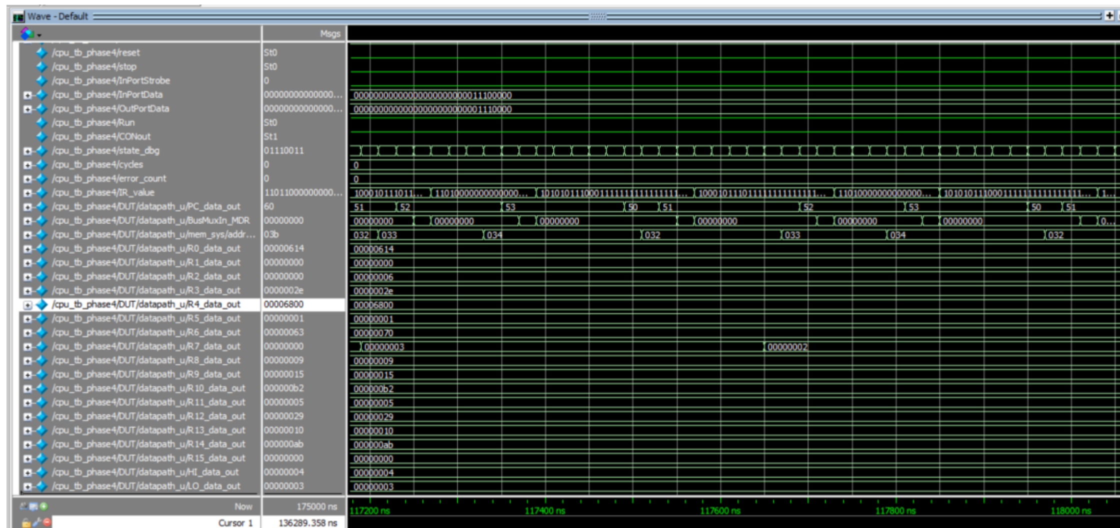

Figure 3 Loop behaviour for Phase 4

Appendix III - Printout of the contents of memory for Phase 4

For simulation purposes, the value stored at memory location 0x88 was changed from 0000FFFF to 00000005. The original value creates a very long delay loop intended for the FPGA hardware operation which results in excessively long simulation times. Reducing the value to five allowed the program to complete in a reasonable simulation timeframe while preserving the intended functionality. The waveform shown below confirms that the relevant memory locations were correctly initialized and accessed during the program. The values at addresses 0x89, 0xA3, and 0x88 demonstrate the expected memory reads and writes performed by the CPU while executing the test program.

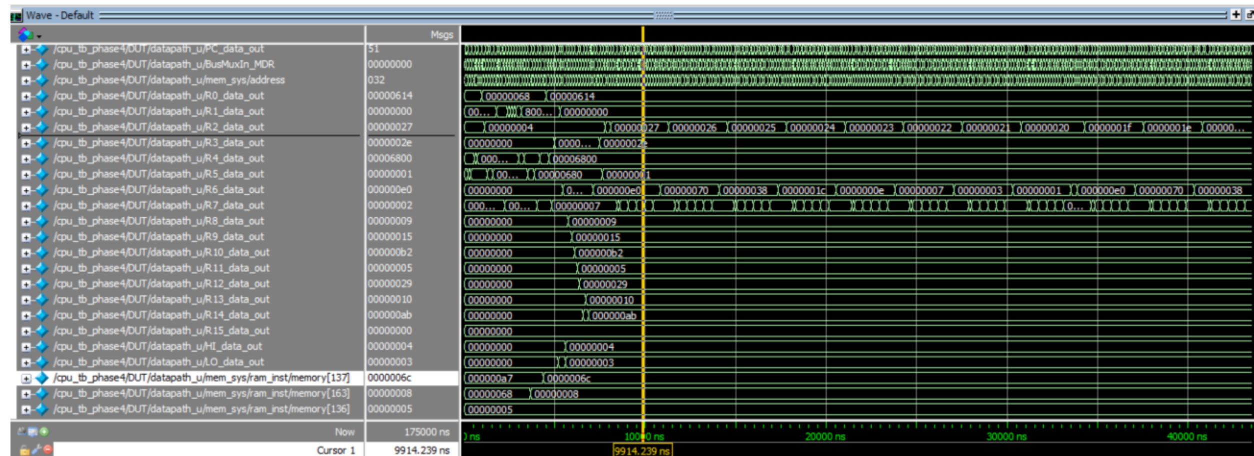

Figure 4 Memory locations 0x89, 0xA3 and 0x88 for Phase 4

Appendix IV – Pictures of FPGA board showing the displays/switches when running program in Phase 4

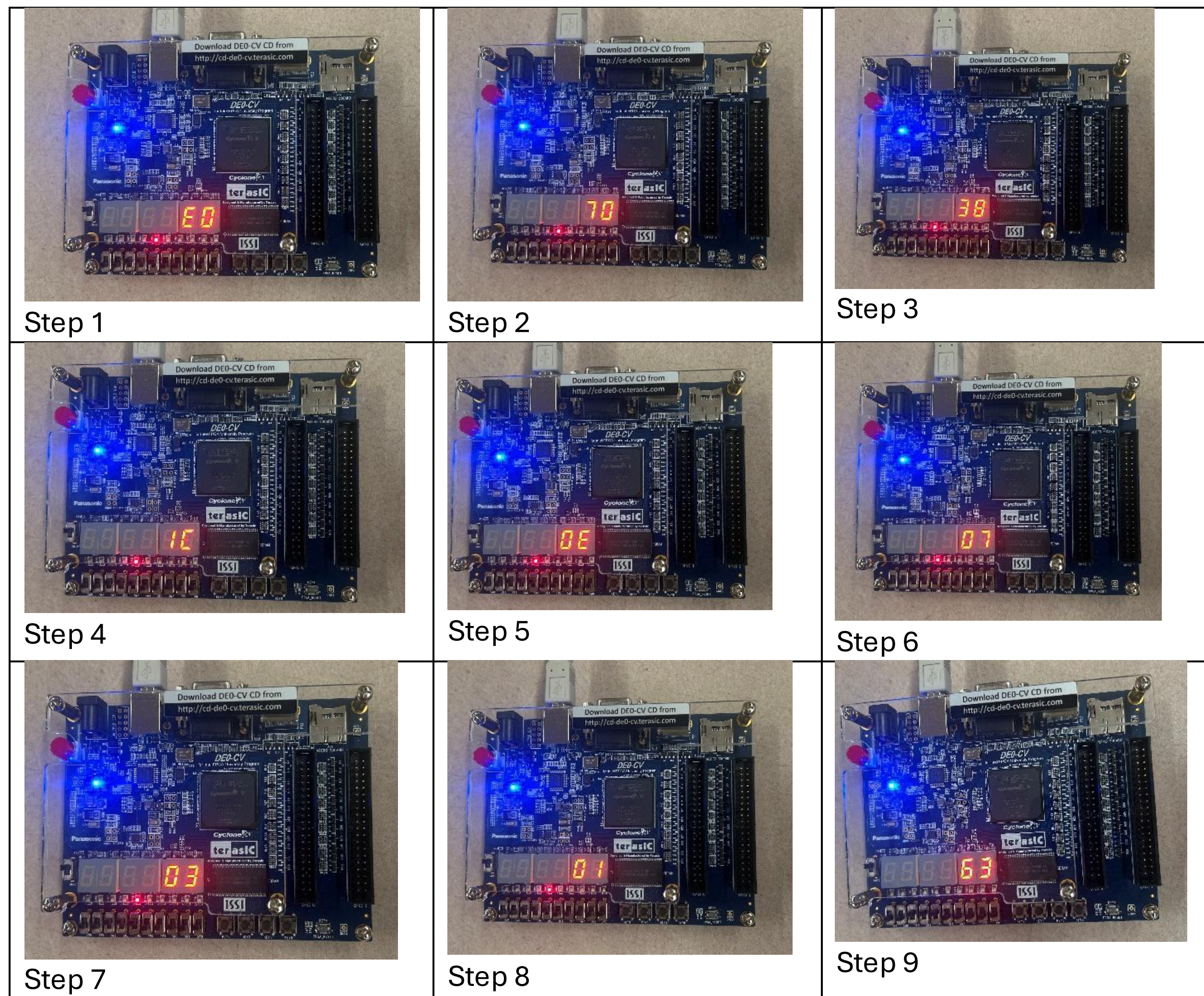

Figure 5: Pictures of each FPGA 7-segment display output

Appendix V - Final Verilog code

Files are pasted in order seen in table:

Section	Files
Phase 1 – ALU submodules	adder, subtractor, neg, and_or, not_32, shift_left, shift_right, shift_right_arithmetic, rol, ror, booth_multiplier, divider
Phase 1 - Datapath	register, sign_extend, bus, alu, datapath
Phase 2	MAR, MDR, conff, select_encode, ram_512x32, memory_subsystem,
Phase 3	control_unit, cpu
Phase 4	cpu_de0_top
Testbenches	ld_tb, cpu_tb

adder.v

```
module adder (A, B, Result);

    input  [31:0] A, B;
    output [31:0] Result;

    reg [31:0] Result;
    reg [32:0] LocalCarry;

    integer i;

    always @(A or B)
    begin
        LocalCarry = 33'd0;
        for (i = 0; i < 32; i = i + 1)
        begin
            Result[i]      = A[i] ^ B[i] ^ LocalCarry[i];
            LocalCarry[i+1] = (A[i] & B[i]) | (LocalCarry[i] & (A[i] | B[i]));
        end
    end

endmodule
```

subtractor.v

```
module subtractor (
    input [31:0] A,
    input [31:0] B,
    output reg [31:0] Result
);
    reg [32:0] Carry;
    wire [31:0] B_inv; //contains one's complement of B

    integer i;
```

```
    assign B_inv = ~B; // one's complement

    always @(*) begin
        Carry[0] = 1'b1; // +1 for two's complement

        for (i = 0; i < 32; i = i + 1) begin
            Result[i] = A[i] ^ B_inv[i] ^ Carry[i]; // calculates sum
            Carry[i+1] = (A[i] & B_inv[i]) | (Carry[i] & (A[i] ^ B_inv[i])); //
calculates carry
        end
    end

endmodule
```

neg.v

```
module neg(
      output wire [31:0] Rz,
      input wire [31:0] Ra
      );

      wire [31:0] not_out;

      not_32 not_32( //flip
            .A(Ra),
            .result(not_out)
      );

      adder adder( //add 1
            .A(not_out),
            .B(32'h00000001),
            .Result(Rz)
      );

endmodule
```

and_or.v

```
module and_or(
      input wire [31:0] A, B,
      input wire selection,
      output wire [31:0] result
);

      assign result = (selection == 1)? A & B : A | B;

endmodule
```

not_32.v

```
module not_32(
    input  wire [31:0] A,
    output wire [31:0] result
);

    assign result = ~A;

endmodule
```

shift_left.v

```
module shift_left
(
    input  wire [31:0] data_in,       // Data to be shifted
    input  wire [31:0]  shift_amount,  // Number of bits to shift
    output wire [31:0] data_out       // Shifted output
);

    assign data_out = data_in << shift_amount;
endmodule
```

shift_right.v

```
module shift_right (
    input  wire [31:0] data_in,       // Data to be shifted
    input  wire [31:0]  shift_amount,  // Number of bits to shift
    output wire [31:0] data_out       // Shifted output
);
    assign data_out = data_in >> shift_amount;
endmodule
```

shift_right_arithmetic.v

```
module shift_right_arithmetic
(
    input  wire signed [31:0] data_in,      // Data to be shifted
    input wire [31:0]         shift_amount,     // Number of bits to shift
    output wire signed [31:0] data_out      // Shifted output
);

    assign data_out = data_in >>> shift_amount;
endmodule
```

rol.v

```
module rol(
    output reg[31:0] Rz,
    input wire[31:0] Ra,
    input wire[31:0] RotateBits
```

```
);

      always@(*)
            begin
                  case(RotateBits)
                        5'd1 : Rz <= {Ra[30:0], Ra[31]};
                        5'd2 : Rz <= {Ra[29:0], Ra[31:30]};
                        5'd3 : Rz <= {Ra[28:0], Ra[31:29]};
                        5'd4 : Rz <= {Ra[27:0], Ra[31:28]};
                        5'd5 : Rz <= {Ra[26:0], Ra[31:27]};
                        5'd6 : Rz <= {Ra[25:0], Ra[31:26]};
                        5'd7 : Rz <= {Ra[24:0], Ra[31:25]};
                        5'd8 : Rz <= {Ra[23:0], Ra[31:24]};
                        5'd9 : Rz <= {Ra[22:0], Ra[31:23]};
                        5'd10: Rz <= {Ra[21:0], Ra[31:22]};
                        5'd11: Rz <= {Ra[20:0], Ra[31:21]};
                        5'd12: Rz <= {Ra[19:0], Ra[31:20]};
                        5'd13: Rz <= {Ra[18:0], Ra[31:19]};
                        5'd14: Rz <= {Ra[17:0], Ra[31:18]};
                        5'd15: Rz <= {Ra[16:0], Ra[31:17]};
                        5'd16: Rz <= {Ra[15:0], Ra[31:16]};
                        5'd17: Rz <= {Ra[14:0], Ra[31:15]};
                        5'd18: Rz <= {Ra[13:0], Ra[31:14]};
                        5'd19: Rz <= {Ra[12:0], Ra[31:13]};
                        5'd20: Rz <= {Ra[11:0], Ra[31:12]};
                        5'd21: Rz <= {Ra[10:0], Ra[31:11]};
                        5'd22: Rz <= {Ra[9:0], Ra[31:10]};
                        5'd23: Rz <= {Ra[8:0], Ra[31:9]};
                        5'd24: Rz <= {Ra[7:0], Ra[31:8]};
                        5'd25: Rz <= {Ra[6:0], Ra[31:7]};
                        5'd26: Rz <= {Ra[5:0], Ra[31:6]};
                        5'd27: Rz <= {Ra[4:0], Ra[31:5]};
                        5'd28: Rz <= {Ra[3:0], Ra[31:4]};
                        5'd29: Rz <= {Ra[2:0], Ra[31:3]};
                        5'd30: Rz <= {Ra[1:0], Ra[31:2]};
                        5'd31: Rz <= {Ra[0], Ra[31:1]};
                        default: Rz <= Ra;
                  endcase
            end
endmodule
```

ror.v

```
module ror(
    output reg[31:0] Rz,
    input wire[31:0] Ra,
    input wire[31:0] RotateBits
);
    always@(*)
        begin
            case(RotateBits)
```

```
                    5'd31:      Rz = {Ra[30:0], Ra[31]};
                    5'd30:      Rz = {Ra[29:0], Ra[31:30]};
                    5'd29:      Rz = {Ra[28:0], Ra[31:29]};
                    5'd28:      Rz = {Ra[27:0], Ra[31:28]};
                    5'd27:      Rz = {Ra[26:0], Ra[31:27]};
                    5'd26:      Rz = {Ra[25:0], Ra[31:26]};
                    5'd25:      Rz = {Ra[24:0], Ra[31:25]};
                    5'd24:      Rz = {Ra[23:0], Ra[31:24]};
                    5'd23:      Rz = {Ra[22:0], Ra[31:23]};
                    5'd22:      Rz = {Ra[21:0], Ra[31:22]};
                    5'd21:      Rz = {Ra[20:0], Ra[31:21]};
                    5'd20:      Rz = {Ra[19:0], Ra[31:20]};
                    5'd19:      Rz = {Ra[18:0], Ra[31:19]};
                    5'd18:      Rz = {Ra[17:0], Ra[31:18]};
                    5'd17:      Rz = {Ra[16:0], Ra[31:17]};
                    5'd16:      Rz = {Ra[15:0], Ra[31:16]};
                    5'd15:      Rz = {Ra[14:0], Ra[31:15]};
                    5'd14:      Rz = {Ra[13:0], Ra[31:14]};
                    5'd13:      Rz = {Ra[12:0], Ra[31:13]};
                    5'd12:      Rz = {Ra[11:0], Ra[31:12]};
                    5'd11:      Rz = {Ra[10:0], Ra[31:11]};
                    5'd10:      Rz = {Ra[9:0], Ra[31:10]};
                    5'd9:       Rz = {Ra[8:0], Ra[31:9]};
                    5'd8:       Rz = {Ra[7:0], Ra[31:8]};
                    5'd7:       Rz = {Ra[6:0], Ra[31:7]};
                    5'd6:       Rz = {Ra[5:0], Ra[31:6]};
                    5'd5:       Rz = {Ra[4:0], Ra[31:5]};
                    5'd4:       Rz = {Ra[3:0], Ra[31:4]};
                    5'd3:       Rz = {Ra[2:0], Ra[31:3]};
                    5'd2:             Rz = {Ra[1:0], Ra[31:2]};
                    5'd1:             Rz = {Ra[0], Ra[31:1]};
                    default: Rz = Ra;
            endcase
        end

endmodule
```

Phase 1 – ALU: booth_multiplier.v

```
module booth_multiplier (
    input  signed [31:0] multiplicand,
    input  signed [31:0] multiplier,
    output reg    signed [63:0] product // 64-bit product (32 x 32 bits)
);

    integer i;
    reg signed [33:0] pp;          // partial product (added 2 bits for shifts)
    reg signed [63:0] acc;         // accumulator (running sum of partial
products)
    reg [33:0] m_ext;              // sign-extended multiplicand
    reg [33:0] mult_ext;           //multiplier with extra 0 bit appended
```

```
    always @(*) begin
        // Sign extend multiplicand
        m_ext = { {2{multiplicand[31]}}, multiplicand };

        // append extra 0 bit to multiplier
        mult_ext = { multiplier, 1'b0 };

        //initialize the accumulator (starting sum = 0)
        acc = 64'd0;

        //Booth loop (32/2 = 16 iterations)
        for (i = 0; i < 16; i = i + 1) begin
            case (mult_ext[2*i +: 3]) //examine 3 bits at a time
                3'b000,
                3'b111: pp = 34'd0;            //  no change add 0
                3'b001,
                3'b010: pp =  m_ext;           // multiply by +1
                3'b011: pp =  m_ext <<< 1;     // multiply by +2 (shift by
one)
                3'b100: pp = -(m_ext <<< 1);   // multiply by -2 (2s
complement and shift by one )
                3'b101,
                3'b110: pp = -m_ext;           // multiply by -1
                default: pp = 34'd0;           // default is no change
            endcase

            //Shift the partial product and add to accumulator
            acc = acc + (pp <<< (2*i));
        end

        product = acc;
    end

endmodule
```

divider.v

```
module divider(
    input wire [31:0] dividend,
    input wire [31:0] divisor,
    output wire [63:0] result
);
    wire dividend_neg = dividend[31];
    wire divisor_neg = divisor[31];
    wire sign_q = dividend_neg ^ divisor_neg;
    wire sign_r = dividend_neg;

    //Take 2s complement if negative
    wire [31:0] dividend_abs = dividend_neg ? (~dividend + 1'b1) : dividend;
    wire [31:0] divisor_abs = divisor_neg ? (~divisor + 1'b1) : divisor;
```

```
reg [31:0] quotient;
reg [31:0] remainder;
reg [31:0] Q;
reg signed [32:0] A;
reg signed [32:0] M;
integer i;

//Non-restoring Division
always @(*) begin
    quotient = 32'b0;
    remainder = 32'b0;
    Q = dividend_abs;
    A = 33'sd0;
    M = {1'b0, divisor_abs};

    if (divisor == 32'b0) begin
        quotient = 32'b0;
        remainder = dividend;
    end else begin
        for (i = 0; i < 32; i = i + 1) begin

            //Shift left
            A = {A[31:0], Q[31]};
            Q = {Q[30:0], 1'b0};

            //Add/Substract M
            if (A[32] == 0)
                A = A - M;
            else
                A = A + M;

            //Set quotient bit
            if (A[32] == 0)
                Q[0] = 1'b1;
            else
                Q[0] = 1'b0;
        end

        //Final check for remainder
        if (A[32] == 1)
            A = A + M;

        quotient = Q;
        remainder = A[31:0];

        if (sign_q)
            quotient = ~quotient + 1'b1;
        if (sign_r)
            remainder = ~remainder + 1'b1;
    end
end
```

```
    assign result = {remainder, quotient};
endmodule
```

register.v

```
module register #( parameter DATA_WIDTH_IN = 32, DATA_WIDTH_OUT = 32, INIT =
32'h0 )(
    input clear, clock, enable,
    input [DATA_WIDTH_IN-1:0] BusMuxOut,
    output wire [DATA_WIDTH_OUT-1:0] BusMuxIn
);

reg [DATA_WIDTH_IN-1:0] q;
initial q = INIT;
always @ (posedge clock)
        begin
            if (clear) begin
                q <= {DATA_WIDTH_IN{1'b0}};
            end
            else if (enable) begin
            q <= BusMuxOut;
            end
        end
    assign BusMuxIn = q[DATA_WIDTH_OUT-1:0];
endmodule
```

sign_extend.v

```
module sign_extend (
    input  [31:0] IR,               // input instruction register (IR)
    input         Cout,             // control signal
    output [31:0] C_sign_extended   // output the sign-extended value of C
);

    wire [18:0] C;                  // Extract the 19-bit immediate value C
from the instruction register (IR)

    assign C = IR[18:0];           // C is the lower 19 bits of the
instruction register (IR)

    assign C_sign_extended = Cout ?                   // if Cout is selected
do the sign extension

                             {{13{C[18]}}, C} :         //repeat the sign bit
(C[18]) 13 times and concatenate it with C (the 13 upper bits are the sign
extension)
                             32'b0;                     // if Cout is not
selected output 32 bits of 0

endmodule
```

bus.v

```
module Bus (
    //Mux
      //all possible sources that can drive the bus
    input [31:0] BusMuxInR0,
    input [31:0] BusMuxInR1,
    input [31:0] BusMuxInR2,
    input [31:0] BusMuxInR3,
    input [31:0] BusMuxInR4,
    input [31:0] BusMuxInR5,
    input [31:0] BusMuxInR6,
    input [31:0] BusMuxInR7,
    input [31:0] BusMuxInR8,
    input [31:0] BusMuxInR9,
    input [31:0] BusMuxInR10,
    input [31:0] BusMuxInR11,
    input [31:0] BusMuxInR12,
    input [31:0] BusMuxInR13,
    input [31:0] BusMuxInR14,
    input [31:0] BusMuxInR15,

      //special registers that can also drive the bus
    input [31:0] BusMuxInPC,
    input [31:0] BusMuxInZlow,
    input [31:0] BusMuxInZhigh,
    input [31:0] BusMuxInMDR,
      input [31:0] BusMuxInIR,
    input [31:0] BusMuxInHI,
    input [31:0] BusMuxInLO,
      input [31:0] BusMuxInY,
    input [31:0] BusMuxInC,
    input [31:0] BusMuxInInPort,



    //Encoder
      // if one of these is a one the registers value will be placed on the
bus
    input R0out, R1out, R2out, R3out, R4out, R5out, R6out, R7out, R8out, R9out,
R10out, R11out, R12out, R13out, R14out, R15out,
    input PCout, MDRout, IRout, Zlowout, Zhighout, HIout, LOout, Yout, Cout,
InPortout,

    output wire [31:0] BusMuxOut
);

      // internal selected bus value
    reg [31:0] q;
```

```
    always @(*) begin
        q = 32'b0;
        if (R0out)    q = BusMuxInR0;
        if (R1out)    q = BusMuxInR1;
        if (R2out)    q = BusMuxInR2;
        if (R3out)    q = BusMuxInR3;
        if (R4out)    q = BusMuxInR4;
        if (R5out)    q = BusMuxInR5;
        if (R6out)    q = BusMuxInR6;
        if (R7out)    q = BusMuxInR7;
        if (R8out)    q = BusMuxInR8;
        if (R9out)    q = BusMuxInR9;
        if (R10out)   q = BusMuxInR10;
        if (R11out)   q = BusMuxInR11;
        if (R12out)   q = BusMuxInR12;
        if (R13out)   q = BusMuxInR13;
        if (R14out)   q = BusMuxInR14;
        if (R15out)   q = BusMuxInR15;
        if (PCout)    q = BusMuxInPC;
        if (MDRout)   q = BusMuxInMDR;
        if (HIout)    q = BusMuxInHI;
        if (LOout)    q = BusMuxInLO;
        if (Zlowout)  q = BusMuxInZlow;
        if (Zhighout) q = BusMuxInZhigh;
            if (Yout)      q = BusMuxInY;
            if (Cout)     q = BusMuxInC;
            if (InPortout) q = BusMuxInInPort;
            if (IRout)    q = BusMuxInIR;
    end

    assign BusMuxOut = q;

endmodule
```

alu.v

```
module ALU(
      input wire [31:0] A, B,
      input wire [3:0] op,
      output reg [63:0] result
);

      //wires to hold the output of each ALU operation
      wire [31:0] and_result, or_result;
      wire [31:0] not_result, add_result, sub_result;
      wire [31:0] shr_result, shra_result, shl_result;
      wire [31:0] neg_result;
      wire [31:0] ror_result, rol_result;
      wire signed [63:0] mul_result;
      wire [63:0] div_result;
```

```
	//instantiate each ALU operation module
	and_or or_instance(A, B, 1'b0, or_result);
	and_or and_instance(A, B, 1'b1, and_result);
	not_32 not_instance(B, not_result);
	adder add_instance(A, B, add_result);
	subtractor sub_instance(A, B, sub_result);
	shift_right shr_instance(A, B, shr_result);
	shift_right_arithmetic shra_instance(A, B, shra_result);
	shift_left shl_instance(A, B, shl_result);
	neg neg_instance(neg_result, B);
	booth_multiplier mul_instance(A, B, mul_result);
	divider div_instance(A, B, div_result);
	ror ror_instance(ror_result, A, B);
	rol rol_instance(rol_result, A, B);


	//assign ror_result = (B[4:0] == 5'd0) ? A : ((A >> B[4:0]) | (A <<
(6'd32 - {1'b0, B[4:0]})));
	//assign rol_result = (B[4:0] == 5'd0) ? A : ((A << B[4:0]) | (A >>
(6'd32 - {1'b0, B[4:0]})));

	// op mapping used by datapath testbenches:
	// 0: OR, 1: AND, 2: NOT(B), 3: ADD, 4: SUB,
	// 5: SHR, 6: SHRA, 7: SHL, 8: ROR, 9: ROL, 10: NEG(B),
	// 11: MUL, 12: DIV({remainder, quotient})
	always @(*) begin
		case(op)
			0:	result = {32'b0, or_result};
			1:	result = {32'b0, and_result};
			2: result = {32'b0, not_result};
			3: result = {32'b0, add_result};
			4: result = {32'b0, sub_result};
			5: result = {32'b0, shr_result};
			6: result = {32'b0, shra_result};
			7: result = {32'b0, shl_result};
			8: result = {32'b0, ror_result};
			9: result = {32'b0, rol_result};
			10: result = {32'b0, neg_result};
			11: result = mul_result;
			12: result = div_result;
			default: result = 64'b0;
		endcase
	end

endmodule
```

datapath.v

```
module datapath(
	input  wire clock, clear,
	input  wire [31:0] A,
```

```
      input  wire [31:0] RegisterImmediate,
      input  wire Read,
    input  wire Write,
      input  wire [3:0] ALUop,


      input  wire [15:0] Rin,   // R0in ... R15in
      input  wire [15:0] Rout , // R0out ... R15out

      input  wire MARin,
      input  wire PCin, PCout,
      input  wire IRin, IRout,
      input  wire Yin,  Yout,
      input  wire MDRin, MDRout,
      input  wire HIin, HIout,
      input  wire LOin, LOout,
      input wire Zhighin, Zlowin, Zhighout, Zlowout,
    input wire BAout,
    input wire IncPC,

    //select encode controls
    input wire UseSelectEncode,
    input wire Gra, Grb, Grc,
    input wire Rin_ctrl, Rout_ctrl,
    input wire Cout,

    //i/o port controls
    input wire InPortout,
    input wire OutPortin,
    input wire InPortStrobe,
    input wire [31:0] InPortData,
    output wire [31:0] OutPortData,

    // CON FF: branch condition latch
    input wire CONin,
    output wire CONout,

    //used to send instruction register value to the control unit
    output wire [31:0] IR_value
);


wire [31:0] BusMuxOut, BusMuxInRZ, BusMuxInRA, BusMuxInRB, BusMuxIn_MDR,
Mdatain;
wire [63:0] zregin;

//General Purpose Registers
wire [31:0] R0_data_out;
wire [31:0] R1_data_out;
wire [31:0] R2_data_out;
wire [31:0] R3_data_out;
wire [31:0] R4_data_out;
```

```
wire [31:0] R5_data_out;
wire [31:0] R6_data_out;
wire [31:0] R7_data_out;
wire [31:0] R8_data_out;
wire [31:0] R9_data_out;
wire [31:0] R10_data_out;
wire [31:0] R11_data_out;
wire [31:0] R12_data_out;
wire [31:0] R13_data_out;
wire [31:0] R14_data_out;
wire [31:0] R15_data_out;

//Special Registers
wire [31:0] PC_data_out;
wire [31:0] IR_data_out;
wire [31:0] Y_data_out;
wire [31:0] HI_data_out;
wire [31:0] LO_data_out;

wire [31:0] Zlow_data_out;
wire [31:0] Zhigh_data_out;
wire [31:0] InPort_data_out;
wire [31:0] OutPort_data_out;

//for revising register R0
wire [31:0] R0_bus_out;         //modified bus output
wire [15:0] Rin_decoded;
wire [15:0] Rout_decoded;
wire [31:0] C_sign_extended;
wire        select_encode_enable;
wire [15:0] Rin_internal;
wire [15:0] Rout_internal;

//if BAout is 1 then output 0 onto the bus instead of R0's value
//then R0 can be used as a zero register
assign R0_bus_out = BAout ? 32'b0 : R0_data_out;
assign select_encode_enable = (UseSelectEncode === 1'b1);
assign Rin_internal  = select_encode_enable ? Rin_decoded  : Rin;
assign Rout_internal = select_encode_enable ? Rout_decoded : Rout;
assign OutPortData = OutPort_data_out;
assign IR_value = IR_data_out;

select_encode select_encode_u (
    .IR(IR_data_out),
    .Gra(Gra),
    .Grb(Grb),
    .Grc(Grc),
    .Rin(Rin_ctrl),
    .Rout(Rout_ctrl),
    .BAout(BAout),
    .Cout(Cout),
    .Rin_decoded(Rin_decoded),
```

```
    .Rout_decoded(Rout_decoded),
    .C_sign_extended(C_sign_extended)
);

// Devices

register R0(
    .clear(clear),
    .clock(clock),
    .enable(Rin_internal[0]),
    .BusMuxOut(BusMuxOut),
    .BusMuxIn(R0_data_out)
);
register R1(
    .clear(clear),
    .clock(clock),
    .enable(Rin_internal[1]),
    .BusMuxOut(BusMuxOut),
    .BusMuxIn(R1_data_out)
);
register R2(
    .clear(clear),
    .clock(clock),
    .enable(Rin_internal[2]),
    .BusMuxOut(BusMuxOut),
    .BusMuxIn(R2_data_out)
);
register R3(
    .clear(clear),
    .clock(clock),
    .enable(Rin_internal[3]),
    .BusMuxOut(BusMuxOut),
    .BusMuxIn(R3_data_out)
);
register R4(
    .clear(clear),
    .clock(clock),
    .enable(Rin_internal[4]),
    .BusMuxOut(BusMuxOut),
    .BusMuxIn(R4_data_out)
);
register R5(
    .clear(clear),
    .clock(clock),
    .enable(Rin_internal[5]),
    .BusMuxOut(BusMuxOut),
    .BusMuxIn(R5_data_out)
);
register R6(
    .clear(clear),
    .clock(clock),
    .enable(Rin_internal[6]),
```

```
    .BusMuxOut(BusMuxOut),
    .BusMuxIn(R6_data_out)
);
register R7(
    .clear(clear),
    .clock(clock),
    .enable(Rin_internal[7]),
    .BusMuxOut(BusMuxOut),
    .BusMuxIn(R7_data_out)
);
register R8(
    .clear(clear),
    .clock(clock),
    .enable(Rin_internal[8]),
    .BusMuxOut(BusMuxOut),
    .BusMuxIn(R8_data_out)
);
register R9(
    .clear(clear),
    .clock(clock),
    .enable(Rin_internal[9]),
    .BusMuxOut(BusMuxOut),
    .BusMuxIn(R9_data_out)
);
register R10(
    .clear(clear),
    .clock(clock),
    .enable(Rin_internal[10]),
    .BusMuxOut(BusMuxOut),
    .BusMuxIn(R10_data_out)
);
register R11(
    .clear(clear),
    .clock(clock),
    .enable(Rin_internal[11]),
    .BusMuxOut(BusMuxOut),
    .BusMuxIn(R11_data_out)
);
register R12(
    .clear(clear),
    .clock(clock),
    .enable(Rin_internal[12]),
    .BusMuxOut(BusMuxOut),
    .BusMuxIn(R12_data_out)
);
register R13(
    .clear(clear),
    .clock(clock),
    .enable(Rin_internal[13]),
    .BusMuxOut(BusMuxOut),
    .BusMuxIn(R13_data_out)
);
```

```
register R14(
    .clear(clear),
    .clock(clock),
    .enable(Rin_internal[14]),
    .BusMuxOut(BusMuxOut),
    .BusMuxIn(R14_data_out)
);
register R15(
    .clear(clear),
    .clock(clock),
    .enable(Rin_internal[15]),
    .BusMuxOut(BusMuxOut),
    .BusMuxIn(R15_data_out)
);


register PC(
    .clear(clear),
    .clock(clock),
    .enable(PCin),
    .BusMuxOut(BusMuxOut),
    .BusMuxIn(PC_data_out)
);
register IR(
    .clear(clear),
    .clock(clock),
    .enable(IRin),
    .BusMuxOut(BusMuxOut),
    .BusMuxIn(IR_data_out)
);
register Y(
    .clear(clear),
    .clock(clock),
    .enable(Yin),
    .BusMuxOut(BusMuxOut),
    .BusMuxIn(Y_data_out)
);
register HI(
    .clear(clear),
    .clock(clock),
    .enable(HIin),
    .BusMuxOut(BusMuxOut),
    .BusMuxIn(HI_data_out)
);
register LO(
    .clear(clear),
    .clock(clock),
    .enable(LOin),
    .BusMuxOut(BusMuxOut),
    .BusMuxIn(LO_data_out)
);
```

```
// output port register: captures bus when OutPortin is asserted
register OutPort(
    .clear(clear),
    .clock(clock),
    .enable(OutPortin),
    .BusMuxOut(BusMuxOut),
    .BusMuxIn(OutPort_data_out)
);


register InPort(
    .clear(clear),
    .clock(clock),
    .enable(InPortStrobe),
    .BusMuxOut(InPortData),
    .BusMuxIn(InPort_data_out)
);

register Zlow(
    .clear(clear),
    .clock(clock),
    .enable(Zlowin),
    .BusMuxOut(zregin[31:0]),
    .BusMuxIn(Zlow_data_out)
);
register Zhigh(
    .clear(clear),
    .clock(clock),
    .enable(Zhighin),
    .BusMuxOut(zregin[63:32]),
    .BusMuxIn(Zhigh_data_out)
);
memory_subsystem mem_sys (
    .clk(clock),
    .clear(clear),
    .MARin(MARin),
    .MDRin(MDRin),
    .MDRout(MDRout),
    .Read(Read),
    .Write(Write),
    .BusMuxOut(BusMuxOut),
    .Mdatain(Mdatain),
    .BusMuxIn_MDR(BusMuxIn_MDR)
);

// ALU wires
wire [63:0] alu_result;

wire [31:0] alu_A = IncPC ? 32'd1 : Y_data_out;

ALU alu (
    .A(alu_A),
```

```
    .B(BusMuxOut),
    .op(ALUop),
    .result(alu_result)
);

// ALU output to Z
assign zregin = alu_result;

// Bus
Bus bus(
    // Temp registers
//    .BusMuxInRZ(BusMuxInRZ),
//    .BusMuxInRA(BusMuxInRA),
//    .BusMuxInRB(BusMuxInRB),

    // General-purpose registers
    .BusMuxInR0(R0_bus_out),            //put modified R0 bus output here
    .BusMuxInR1(R1_data_out),
    .BusMuxInR2(R2_data_out),
    .BusMuxInR3(R3_data_out),
    .BusMuxInR4(R4_data_out),
    .BusMuxInR5(R5_data_out),
    .BusMuxInR6(R6_data_out),
    .BusMuxInR7(R7_data_out),
    .BusMuxInR8(R8_data_out),
    .BusMuxInR9(R9_data_out),
    .BusMuxInR10(R10_data_out),
    .BusMuxInR11(R11_data_out),
    .BusMuxInR12(R12_data_out),
    .BusMuxInR13(R13_data_out),
    .BusMuxInR14(R14_data_out),
    .BusMuxInR15(R15_data_out),

    // Special registers
    .BusMuxInPC(PC_data_out),

      .BusMuxInZlow(Zlow_data_out),
      .BusMuxInZhigh(Zhigh_data_out),

      .BusMuxInMDR(BusMuxIn_MDR),
    .BusMuxInIR(IR_data_out),
    .BusMuxInY(Y_data_out),
    .BusMuxInHI(HI_data_out),
    .BusMuxInLO(LO_data_out),
    .BusMuxInC(C_sign_extended),
    .BusMuxInInPort(InPort_data_out),



       .R0out(Rout_internal[0]),
    .R1out(Rout_internal[1]),
    .R2out(Rout_internal[2]),
```

```
    .R3out(Rout_internal[3]),
    .R4out(Rout_internal[4]),
    .R5out(Rout_internal[5]),
    .R6out(Rout_internal[6]),
    .R7out(Rout_internal[7]),
    .R8out(Rout_internal[8]),
    .R9out(Rout_internal[9]),
    .R10out(Rout_internal[10]),
    .R11out(Rout_internal[11]),
    .R12out(Rout_internal[12]),
    .R13out(Rout_internal[13]),
    .R14out(Rout_internal[14]),
    .R15out(Rout_internal[15]),


        .PCout(PCout),
    .MDRout(MDRout),
        .IRout(IRout),
    .HIout(HIout),
    .LOout(LOout),
    .Yout (Yout),
    .Cout(select_encode_enable & Cout),
    .InPortout(InPortout),


        .Zlowout(Zlowout),
        .Zhighout(Zhighout),


    // Output

    .BusMuxOut(BusMuxOut)
);


conff conff_inst (
    .BusMuxOut(BusMuxOut),
    .IR_C2(IR_data_out[20:19]),
    .CONin(CONin),
    .clk(clock),
    .clear(clear),
    .CONout(CONout)
);

endmodule
```

MAR.v

```
module MAR (
    input clk,
    input clear,
```

```
    input MARin,                  // enable signal
    input [31:0] bus_mux_out,     // data from bus
    output [8:0] address          // to RAM
);

    reg [31:0] MAR_reg;               // internal register to hold the value of
MAR

    always @(posedge clk) begin
        if (clear)                    // on clear set MAR to all zeros
            MAR_reg <= 32'b0;
        else if (MARin)               //when MAR_in is enabled
            MAR_reg <= bus_mux_out; //load the value from the bus into MAR
    end

    // Only lower 9 bits go to RAM because RAM has 512 (2^9) addresses
    assign address = MAR_reg[8:0];

endmodule
```

MDR.v

```
module MDR (
    input clk,
    input clear,
    input MDRin,                  // load enable
    input MDRout,                 // drive bus enable
    input [31:0] bus_mux_out,     //data from CPU bus
    input [31:0] Mdatain,         //data from RAM
    input Read,                   //selects RAM input
    output [31:0] BusMuxIn_MDR,   //to Bus
    output [31:0] data_to_RAM     //to RAM
);

    reg [31:0] MDR_reg;

    // Load logic
    always @(posedge clk) begin
        if (clear)
            MDR_reg <= 32'b0;
        else if (MDRin) begin
            if (Read)
                MDR_reg <= Mdatain;       // load from RAM
            else
                MDR_reg <= bus_mux_out;   // load from bus
        end
    end

    // drive bus only when MDRout = 1 otherwise drive 0
    assign BusMuxIn_MDR = MDRout ? MDR_reg : 32'b0;
```

```
    //data going to RAM for write
    assign data_to_RAM = MDR_reg;

endmodule
```

conff.v

```
// CON FF: latches the branch condition result on posedge clk when CONin is
asserted.
// IR_C2 = IR[20:19], the C2 condition field from the branch instruction.
module conff(
    input [31:0] BusMuxOut,   // Ra value on the bus \
    input [1:0]  IR_C2,       // branch condition
    input        CONin,
    input        clk,
    input        clear,
    output reg   CONout       // 1 = branch taken, held until next CONin
);

    always @(posedge clk, posedge clear) begin
        if (clear)
            CONout <= 0;
        else if (CONin) begin
            CONout <= 0; // default: condition not met
            case (IR_C2)
                2'b00: if (BusMuxOut == 0)        CONout <= 1; // brzr: branch
if zero
                2'b01: if (BusMuxOut != 0)        CONout <= 1; // brnz: branch
if nonzero
                2'b10: if (BusMuxOut[31] == 0)    CONout <= 1; // brpl: branch
if positive
                2'b11: if (BusMuxOut[31] == 1)    CONout <= 1; // brmi: branch
if negative
            endcase
        end
    end

endmodule
```

select_encode.v

```
module select_encode(
    // full 32-bit instruction currently stored in ir
    input  wire [31:0] IR,

    // these pick which ir field Ra, Rb, or Rc should be used as the register
index
    input  wire Gra,
    input  wire Grb,
    input  wire Grc,
```

```
    // control bits that enable decoded write/read register lines
    input  wire Rin,
    input  wire Rout,

    // base address mode; this also enables the decoded rout path
    input  wire BAout,

    // controls whether the immediate constant path is active in sign_extend
    input  wire Cout,

    // one-hot write enables for r0..r15
    output wire [15:0] Rin_decoded,
    // one-hot read enables for r0..r15
    output wire [15:0] Rout_decoded,

    // 32-bit immediate constant output built from ir[18:0]
    output wire [31:0] C_sign_extended
);

    // extract the 3 register fields from fixed ir bit ranges
    wire [3:0] Ra = IR[26:23];
    wire [3:0] Rb = IR[22:19];
    wire [3:0] Rc = IR[18:15];

    // AND each register with its control field, OR all fields to feed into 4
to 16 decoder
    wire [3:0] reg_sel = ({4{Gra}} & Ra) | ({4{Grb}} & Rb) | ({4{Grc}} & Rc);

    // convert selected 4-bit index into one-hot 16-bit decode
    wire [15:0] dec = (16'b1 << reg_sel);

    // decoded rout path should turn on if either rout or baout is asserted
    wire rout_enable = Rout | BAout;

    // gate decoded outputs with the control enables
    assign Rin_decoded  = Rin         ? dec : 16'b0;
    assign Rout_decoded = rout_enable ? dec : 16'b0;

    // use the existing sign_extend module instead of duplicating that logic
here
    sign_extend sign_extend_u (
        .IR(IR),
        .Cout(Cout),
        .C_sign_extended(C_sign_extended)
    );

endmodule
```

ram_512x32.v

```
module ram_512x32 (
```

```
    input clk,
    input read,
    input write,
    input [8:0] address,        // 9 bits adresses
    input [31:0] data_in,       // write RAM
    output reg [31:0] data_out  // read RAM
);

    reg [31:0] memory [0:511];      //512 words of 32 bits each
    integer i;

    //phase 4 sample program starting at mem location 0x0
    initial begin
        for (i = 0; i < 512; i = i + 1)
            memory[i] = 32'b0;

        memory[9'h000] = 32'h8A800043; // ldi  R5, 0x43
        memory[9'h001] = 32'h8AA80006; // ldi  R5, 6(R5)
        memory[9'h002] = 32'h82000089; // ld   R4, 0x89
        memory[9'h003] = 32'h8A200004; // ldi  R4, 4(R4)
        memory[9'h004] = 32'h8027FFF8; // ld   R0, -8(R4)
        memory[9'h005] = 32'h89000004; // ldi  R2, 4
        memory[9'h006] = 32'h8A800087; // ldi  R5, 0x87
        memory[9'h007] = 32'hAA980003; // brmi R5, 3
        memory[9'h008] = 32'h8AA80005; // ldi  R5, 5(R5)
        memory[9'h009] = 32'h80AFFFFD; // ld   R1, -3(R5)
        memory[9'h00A] = 32'hD0000000; // nop
        memory[9'h00B] = 32'hA8900002; // brpl R1, 2
        memory[9'h00C] = 32'h89A80007; // ldi  R3, 7(R5)
        memory[9'h00D] = 32'h8B9FFFFC; // ldi  R7, -4(R3)
        memory[9'h00E] = 32'h03A90000; // add  R7, R5, R2
        memory[9'h00F] = 32'h48880003; // addi R1, R1, 3
        memory[9'h010] = 32'h70880000; // neg  R1, R1
        memory[9'h011] = 32'h78880000; // not  R1, R1
        memory[9'h012] = 32'h5088000F; // andi R1, R1, 0xF
        memory[9'h013] = 32'h3A010000; // ror  R4, R0, R2
        memory[9'h014] = 32'h58A00005; // ori  R1, R4, 5
        memory[9'h015] = 32'h2A090000; // shra R4, R1, R2
        memory[9'h016] = 32'h22A90000; // shr  R5, R5, R2
        memory[9'h017] = 32'h928000A3; // st   0xA3, R5
        memory[9'h018] = 32'h42810000; // rol  R5, R0, R2
        memory[9'h019] = 32'h1B900000; // or   R7, R2, R0
        memory[9'h01A] = 32'h12280000; // and  R4, R5, R0
        memory[9'h01B] = 32'h93A00089; // st   0x89(R4), R7
        memory[9'h01C] = 32'h082B8000; // sub  R0, R5, R7
        memory[9'h01D] = 32'h32290000; // shl  R4, R5, R2
        memory[9'h01E] = 32'h8B800007; // ldi  R7, 7
        memory[9'h01F] = 32'h89800019; // ldi  R3, 0x19
        memory[9'h020] = 32'h69B80000; // mul  R3, R7
        memory[9'h021] = 32'hC0800000; // mfhi R1
        memory[9'h022] = 32'hCB000000; // mflo R6
        memory[9'h023] = 32'h61B80000; // div  R3, R7
```

```
        memory[9'h024] = 32'h8C380002; // ldi  R8, 2(R7)
        memory[9'h025] = 32'h8C9FFFFC; // ldi  R9, -4(R3)
        memory[9'h026] = 32'h8D300003; // ldi  R10, 3(R6)
        memory[9'h027] = 32'h8D880005; // ldi  R11, 5(R1)
        memory[9'h028] = 32'h9D000000; // jal  R10
        memory[9'h029] = 32'hB3000000; // in   R6
        memory[9'h02A] = 32'h93000077; // st   0x77, R6
        memory[9'h02B] = 32'h8980002E; // ldi  R3, 0x2E
        memory[9'h02C] = 32'h8A800001; // ldi  R5, 1
        memory[9'h02D] = 32'h89000028; // ldi  R2, 40
        memory[9'h02E] = 32'hBB000000; // out  R6
        memory[9'h02F] = 32'h8917FFFF; // ldi  R2, -1(R2)
        memory[9'h030] = 32'hA9000008; // brzr R2, 8
        memory[9'h031] = 32'h83800088; // ld   R7, 0x88
        memory[9'h032] = 32'h8BBFFFFF; // ldi  R7, -1(R7)
        memory[9'h033] = 32'hD0000000; // nop
        memory[9'h034] = 32'hAB8FFFFD; // brnz R7, -3
        memory[9'h035] = 32'h23328000; // shr  R6, R6, R5
        memory[9'h036] = 32'hAB0FFFF7; // brnz R6, -9
        memory[9'h037] = 32'h83000077; // ld   R6, 0x77
        memory[9'h038] = 32'hA1800000; // jr   R3
        memory[9'h039] = 32'h8B000063; // ldi  R6, 0x63
        memory[9'h03A] = 32'hBB000000; // out  R6
        memory[9'h03B] = 32'hD8000000; // halt

              //Initialize memory locations 0x88, 0x89, and 0xA3
        memory[9'h089] = 32'h000000A7;
        memory[9'h0A3] = 32'h00000068;
        memory[9'h088] = 32'h0000FFFF;

        memory[9'h0B2] = 32'h07450000; // add  R14, R8, R10
        memory[9'h0B3] = 32'h0ECD8000; // sub  R13, R9, R11
        memory[9'h0B4] = 32'h0F768000; // sub  R14, R14, R13
        memory[9'h0B5] = 32'hA6000000; // jr   R12
    end

    always @(posedge clk) begin
        if (write)
            memory[address] <= data_in;//write to RAM

        if (read)
            data_out <= memory[address];//read from RAM
    end

endmodule
```

memory_subsystem.v

```
module memory_subsystem (
    input clk,
    input clear,
```

```
    // Control signals
    input MARin,
    input MDRin,
    input MDRout,
    input Read,
    input Write,

    // CPU Bus input
    input [31:0] BusMuxOut,

    output [31:0] Mdatain,

    // Bus output from MDR
    output [31:0] BusMuxIn_MDR
);

    // Internal wires
    wire [8:0] address;
    wire [31:0] ram_out;

    wire [31:0] data_to_RAM;
    wire [31:0] mdmux_out;

    // MAR Instance
    MAR mar_inst (
        .clk(clk),
        .clear(clear),
        .MARin(MARin),
        .bus_mux_out(BusMuxOut),
        .address(address)
    );

    // MDMux logic selects between RAM output and BusMuxOut based on Read
signal
    assign mdmux_out = Read ? Mdatain : BusMuxOut;

    // MDR Instance
    MDR mdr_inst (
        .clk(clk),
        .clear(clear),
        .MDRin(MDRin),
        .MDRout(MDRout),
        .bus_mux_out(BusMuxOut),
        .Mdatain(Mdatain),
        .Read(Read),
        .BusMuxIn_MDR(BusMuxIn_MDR),
        .data_to_RAM(data_to_RAM)
    );

    // RAM Instance
    ram_512x32 ram_inst (
```

```
        .clk(clk),
        .read(Read),
        .write(Write),
        .address(address),
        .data_in(data_to_RAM),
        .data_out(Mdatain)
    );

endmodule
```

control_unit.v

```
`timescale 1ns/10ps

module control_unit (
    input  wire         clock,
    input  wire         reset,
    input  wire         stop,
    input  wire [31:0] IR,
    input  wire         CONout,

    output reg          Run,

    output reg          Read,
    output reg          Write,
    output reg [3:0]    ALUop,

    output reg [15:0]   Rin,
    output reg [15:0]   Rout,

    output reg          MARin,
    output reg          PCin,
    output reg          PCout,
    output reg          IRin,
    output reg          Yin,
    output reg          MDRin,
    output reg          MDRout,
    output reg          HIin,
    output reg          HIout,
    output reg          LOin,
    output reg          LOout,
    output reg          Zhighin,
    output reg          Zlowin,
    output reg          Zhighout,
    output reg          Zlowout,
    output reg          BAout,
    output reg          IncPC,

    output reg          UseSelectEncode,
    output reg          Gra,
    output reg          Grb,
```

```
    output reg          Grc,
    output reg          Rin_ctrl,
    output reg          Rout_ctrl,
    output reg          Cout,

    output reg          InPortout,
    output reg          OutPortin,
    output reg          CONin,

    output wire [7:0]   state_dbg
);

//opcode values for each instruction
localparam [4:0]
    OP_ADD  = 5'b00000,
    OP_SUB  = 5'b00001,
    OP_AND  = 5'b00010,
    OP_OR   = 5'b00011,
    OP_SHR  = 5'b00100,
    OP_SHRA = 5'b00101,
    OP_SHL  = 5'b00110,
    OP_ROR  = 5'b00111,
    OP_ROL  = 5'b01000,
    OP_ADDI = 5'b01001,
    OP_ANDI = 5'b01010,
    OP_ORI  = 5'b01011,
    OP_DIV  = 5'b01100,
    OP_MUL  = 5'b01101,
    OP_NEG  = 5'b01110,
    OP_NOT  = 5'b01111,
    OP_LD   = 5'b10000,
    OP_LDI  = 5'b10001,
    OP_ST   = 5'b10010,
    OP_JAL  = 5'b10011,
    OP_JR   = 5'b10100,
    OP_BR   = 5'b10101,
    OP_IN   = 5'b10110,
    OP_OUT  = 5'b10111,
    OP_MFHI = 5'b11000,
    OP_MFLO = 5'b11001,
    OP_NOP  = 5'b11010,
    OP_HALT = 5'b11011;


//control unit states, fetch/decode states are the same for all instructions
localparam [7:0]
    RESET_STATE = 8'd0,
    FETCH0      = 8'd1,
    FETCH1      = 8'd2,
    FETCH2      = 8'd3,
    FETCH3      = 8'd4,
    DECODE      = 8'd5,
```

```
    ADD4        = 8'd10,
    ADD5        = 8'd11,
    ADD6        = 8'd12,
    SUB4        = 8'd13,
    SUB5        = 8'd14,
    SUB6        = 8'd15,
    AND4        = 8'd16,
    AND5        = 8'd17,
    AND6        = 8'd18,
    OR4         = 8'd19,
    OR5         = 8'd20,
    OR6         = 8'd21,
    SHR4        = 8'd22,
    SHR5        = 8'd23,
    SHR6        = 8'd24,
    SHRA4       = 8'd25,
    SHRA5       = 8'd26,
    SHRA6       = 8'd27,
    SHL4        = 8'd28,
    SHL5        = 8'd29,
    SHL6        = 8'd30,
    ROR4        = 8'd31,
    ROR5        = 8'd32,
    ROR6        = 8'd33,
    ROL4        = 8'd34,
    ROL5        = 8'd35,
    ROL6        = 8'd36,

    ADDI4       = 8'd40,
    ADDI5       = 8'd41,
    ADDI6       = 8'd42,
    ANDI4       = 8'd43,
    ANDI5       = 8'd44,
    ANDI6       = 8'd45,
    ORI4        = 8'd46,
    ORI5        = 8'd47,
    ORI6        = 8'd48,

    NEG4        = 8'd50,
    NEG5        = 8'd51,
    NOT4        = 8'd52,
    NOT5        = 8'd53,

    LDI4        = 8'd60,
    LDI5        = 8'd61,
    LDI6        = 8'd62,
    LD4         = 8'd63,
    LD5         = 8'd64,
    LD6         = 8'd65,
    LD7         = 8'd66,
    LD8         = 8'd67,
```

```
    LD9         = 8'd68,
    ST4         = 8'd69,
    ST5         = 8'd70,
    ST6         = 8'd71,
    ST7         = 8'd72,
    ST8         = 8'd73,

    BR4         = 8'd80,
    BR5         = 8'd81,
    BR6         = 8'd82,
    BR7         = 8'd83,

    JR4         = 8'd90,
    JAL4        = 8'd91,
    JAL5        = 8'd92,

    MUL4        = 8'd100,
    MUL5        = 8'd101,
    MUL6        = 8'd102,
    MUL7        = 8'd103,
    DIV4        = 8'd104,
    DIV5        = 8'd105,
    DIV6        = 8'd106,
    DIV7        = 8'd107,

    MFHI4       = 8'd110,
    MFLO4       = 8'd111,
    IN4         = 8'd112,
    OUT4        = 8'd113,
    NOP4        = 8'd114,
    HALT_STATE  = 8'd115;

reg  [7:0] present_state = RESET_STATE;
reg  [7:0] next_state;
wire [4:0] opcode = IR[31:27];

assign state_dbg = present_state;

always @(posedge clock or posedge reset or posedge stop) begin
    if (reset)
        present_state <= RESET_STATE;
    else if (stop)
        present_state <= HALT_STATE;
    else
        present_state <= next_state;
end

always @(*) begin
    next_state = present_state;

    case (present_state)
        RESET_STATE: next_state = FETCH0;
```

```
        FETCH0:       next_state = FETCH1;
        FETCH1:       next_state = FETCH2;
        FETCH2:       next_state = FETCH3;
        FETCH3:       next_state = DECODE;

        DECODE: begin
            case (opcode)
                OP_ADD:  next_state = ADD4;
                OP_SUB:  next_state = SUB4;
                OP_AND:  next_state = AND4;
                OP_OR:   next_state = OR4;
                OP_SHR:  next_state = SHR4;
                OP_SHRA: next_state = SHRA4;
                OP_SHL:  next_state = SHL4;
                OP_ROR:  next_state = ROR4;
                OP_ROL:  next_state = ROL4;
                OP_ADDI: next_state = ADDI4;
                OP_ANDI: next_state = ANDI4;
                OP_ORI:  next_state = ORI4;
                OP_DIV:  next_state = DIV4;
                OP_MUL:  next_state = MUL4;
                OP_NEG:  next_state = NEG4;
                OP_NOT:  next_state = NOT4;
                OP_LD:   next_state = LD4;
                OP_LDI:  next_state = LDI4;
                OP_ST:   next_state = ST4;
                OP_JAL:  next_state = JAL4;
                OP_JR:   next_state = JR4;
                OP_BR:   next_state = BR4;
                OP_IN:   next_state = IN4;
                OP_OUT:  next_state = OUT4;
                OP_MFHI: next_state = MFHI4;
                OP_MFLO: next_state = MFLO4;
                OP_NOP:  next_state = NOP4;
                OP_HALT: next_state = HALT_STATE;
                default: next_state = HALT_STATE;
            endcase
        end

        ADD4:  next_state = ADD5;
        ADD5:  next_state = ADD6;
        ADD6:  next_state = FETCH0;
        SUB4:  next_state = SUB5;
        SUB5:  next_state = SUB6;
        SUB6:  next_state = FETCH0;
        AND4:  next_state = AND5;
        AND5:  next_state = AND6;
        AND6:  next_state = FETCH0;
        OR4:   next_state = OR5;
        OR5:   next_state = OR6;
        OR6:   next_state = FETCH0;
        SHR4:  next_state = SHR5;
```

```
        SHR5:  next_state = SHR6;
        SHR6:  next_state = FETCH0;
        SHRA4: next_state = SHRA5;
        SHRA5: next_state = SHRA6;
        SHRA6: next_state = FETCH0;
        SHL4:  next_state = SHL5;
        SHL5:  next_state = SHL6;
        SHL6:  next_state = FETCH0;
        ROR4:  next_state = ROR5;
        ROR5:  next_state = ROR6;
        ROR6:  next_state = FETCH0;
        ROL4:  next_state = ROL5;
        ROL5:  next_state = ROL6;
        ROL6:  next_state = FETCH0;

        ADDI4: next_state = ADDI5;
        ADDI5: next_state = ADDI6;
        ADDI6: next_state = FETCH0;
        ANDI4: next_state = ANDI5;
        ANDI5: next_state = ANDI6;
        ANDI6: next_state = FETCH0;
        ORI4:  next_state = ORI5;
        ORI5:  next_state = ORI6;
        ORI6:  next_state = FETCH0;

        NEG4:  next_state = NEG5;
        NEG5:  next_state = FETCH0;
        NOT4:  next_state = NOT5;
        NOT5:  next_state = FETCH0;

        LDI4:  next_state = LDI5;
        LDI5:  next_state = LDI6;
        LDI6:  next_state = FETCH0;
        LD4:   next_state = LD5;
        LD5:   next_state = LD6;
        LD6:   next_state = LD7;
        LD7:   next_state = LD8;
        LD8:   next_state = LD9;
        LD9:   next_state = FETCH0;
        ST4:   next_state = ST5;
        ST5:   next_state = ST6;
        ST6:   next_state = ST7;
        ST7:   next_state = ST8;
        ST8:   next_state = FETCH0;

        BR4:   next_state = BR5;
        BR5:   next_state = CONout ? BR6 : FETCH0;
        BR6:   next_state = BR7;
        BR7:   next_state = FETCH0;

        JR4:   next_state = FETCH0;
        JAL4:  next_state = JAL5;
```

```
        JAL5:  next_state = FETCH0;

        MUL4:  next_state = MUL5;
        MUL5:  next_state = MUL6;
        MUL6:  next_state = MUL7;
        MUL7:  next_state = FETCH0;
        DIV4:  next_state = DIV5;
        DIV5:  next_state = DIV6;
        DIV6:  next_state = DIV7;
        DIV7:  next_state = FETCH0;

        MFHI4: next_state = FETCH0;
        MFLO4: next_state = FETCH0;
        IN4:   next_state = FETCH0;
        OUT4:  next_state = FETCH0;
        NOP4:  next_state = FETCH0;
        HALT_STATE: next_state = HALT_STATE;

        default: next_state = HALT_STATE;
    endcase
end

always @(*) begin
    Run             = 1'b1;

    Read            = 1'b0;
    Write           = 1'b0;
    ALUop           = 4'd3;

    Rin             = 16'b0;
    Rout            = 16'b0;

    MARin           = 1'b0;
    PCin            = 1'b0;
    PCout           = 1'b0;
    IRin            = 1'b0;
    Yin             = 1'b0;
    MDRin           = 1'b0;
    MDRout          = 1'b0;
    HIin            = 1'b0;
    HIout           = 1'b0;
    LOin            = 1'b0;
    LOout           = 1'b0;
    Zhighin         = 1'b0;
    Zlowin          = 1'b0;
    Zhighout        = 1'b0;
    Zlowout         = 1'b0;
    BAout           = 1'b0;
    IncPC           = 1'b0;

    UseSelectEncode = 1'b1;
    Gra             = 1'b0;
```

```
    Grb             = 1'b0;
    Grc             = 1'b0;
    Rin_ctrl        = 1'b0;
    Rout_ctrl       = 1'b0;
    Cout            = 1'b0;

    InPortout       = 1'b0;
    OutPortin       = 1'b0;
    CONin           = 1'b0;

    case (present_state)
        RESET_STATE: begin
        end

        FETCH0: begin
            PCout    = 1'b1;
            MARin    = 1'b1;
            IncPC    = 1'b1;
            ALUop    = 4'd3;
            Zlowin   = 1'b1;
        end

        FETCH1: begin
            Zlowout = 1'b1;
            PCin     = 1'b1;
            Read     = 1'b1;
        end

        FETCH2: begin
            Read     = 1'b1;
            MDRin    = 1'b1;
        end

        FETCH3: begin
            MDRout   = 1'b1;
            IRin     = 1'b1;
        end

        DECODE: begin
        end

        ADD4: begin Grb = 1'b1; Rout_ctrl = 1'b1; Yin = 1'b1; end
        ADD5: begin Grc = 1'b1; Rout_ctrl = 1'b1; ALUop = 4'd3; Zlowin = 1'b1;
end
        ADD6: begin Zlowout = 1'b1; Gra = 1'b1; Rin_ctrl = 1'b1; end

        SUB4: begin Grb = 1'b1; Rout_ctrl = 1'b1; Yin = 1'b1; end
        SUB5: begin Grc = 1'b1; Rout_ctrl = 1'b1; ALUop = 4'd4; Zlowin = 1'b1;
end
        SUB6: begin Zlowout = 1'b1; Gra = 1'b1; Rin_ctrl = 1'b1; end

        AND4: begin Grb = 1'b1; Rout_ctrl = 1'b1; Yin = 1'b1; end
```

```
        AND5: begin Grc = 1'b1; Rout_ctrl = 1'b1; ALUop = 4'd1; Zlowin = 1'b1;
end
        AND6: begin Zlowout = 1'b1; Gra = 1'b1; Rin_ctrl = 1'b1; end

        OR4: begin Grb = 1'b1; Rout_ctrl = 1'b1; Yin = 1'b1; end
        OR5: begin Grc = 1'b1; Rout_ctrl = 1'b1; ALUop = 4'd0; Zlowin = 1'b1;
end
        OR6: begin Zlowout = 1'b1; Gra = 1'b1; Rin_ctrl = 1'b1; end

        SHR4: begin Grb = 1'b1; Rout_ctrl = 1'b1; Yin = 1'b1; end
        SHR5: begin Grc = 1'b1; Rout_ctrl = 1'b1; ALUop = 4'd5; Zlowin = 1'b1;
end
        SHR6: begin Zlowout = 1'b1; Gra = 1'b1; Rin_ctrl = 1'b1; end

        SHRA4: begin Grb = 1'b1; Rout_ctrl = 1'b1; Yin = 1'b1; end
        SHRA5: begin Grc = 1'b1; Rout_ctrl = 1'b1; ALUop = 4'd6; Zlowin = 1'b1;
end
        SHRA6: begin Zlowout = 1'b1; Gra = 1'b1; Rin_ctrl = 1'b1; end

        SHL4: begin Grb = 1'b1; Rout_ctrl = 1'b1; Yin = 1'b1; end
        SHL5: begin Grc = 1'b1; Rout_ctrl = 1'b1; ALUop = 4'd7; Zlowin = 1'b1;
end
        SHL6: begin Zlowout = 1'b1; Gra = 1'b1; Rin_ctrl = 1'b1; end

        ROR4: begin Grb = 1'b1; Rout_ctrl = 1'b1; Yin = 1'b1; end
        ROR5: begin Grc = 1'b1; Rout_ctrl = 1'b1; ALUop = 4'd8; Zlowin = 1'b1;
end
        ROR6: begin Zlowout = 1'b1; Gra = 1'b1; Rin_ctrl = 1'b1; end

        ROL4: begin Grb = 1'b1; Rout_ctrl = 1'b1; Yin = 1'b1; end
        ROL5: begin Grc = 1'b1; Rout_ctrl = 1'b1; ALUop = 4'd9; Zlowin = 1'b1;
end
        ROL6: begin Zlowout = 1'b1; Gra = 1'b1; Rin_ctrl = 1'b1; end

        ADDI4: begin Grb = 1'b1; Rout_ctrl = 1'b1; Yin = 1'b1; end
        ADDI5: begin Cout = 1'b1; ALUop = 4'd3; Zlowin = 1'b1; end
        ADDI6: begin Zlowout = 1'b1; Gra = 1'b1; Rin_ctrl = 1'b1; end

        ANDI4: begin Grb = 1'b1; Rout_ctrl = 1'b1; Yin = 1'b1; end
        ANDI5: begin Cout = 1'b1; ALUop = 4'd1; Zlowin = 1'b1; end
        ANDI6: begin Zlowout = 1'b1; Gra = 1'b1; Rin_ctrl = 1'b1; end

        ORI4: begin Grb = 1'b1; Rout_ctrl = 1'b1; Yin = 1'b1; end
        ORI5: begin Cout = 1'b1; ALUop = 4'd0; Zlowin = 1'b1; end
        ORI6: begin Zlowout = 1'b1; Gra = 1'b1; Rin_ctrl = 1'b1; end

        NEG4: begin Grb = 1'b1; Rout_ctrl = 1'b1; ALUop = 4'd10; Zlowin = 1'b1;
end
        NEG5: begin Zlowout = 1'b1; Gra = 1'b1; Rin_ctrl = 1'b1; end

        NOT4: begin Grb = 1'b1; Rout_ctrl = 1'b1; ALUop = 4'd2; Zlowin = 1'b1;
end
```

```
        NOT5: begin Zlowout = 1'b1; Gra = 1'b1; Rin_ctrl = 1'b1; end

        LDI4: begin Grb = 1'b1; BAout = 1'b1; Yin = 1'b1; end
        LDI5: begin Cout = 1'b1; ALUop = 4'd3; Zlowin = 1'b1; end
        LDI6: begin Zlowout = 1'b1; Gra = 1'b1; Rin_ctrl = 1'b1; end

        LD4: begin Grb = 1'b1; BAout = 1'b1; Yin = 1'b1; end
        LD5: begin Cout = 1'b1; ALUop = 4'd3; Zlowin = 1'b1; end
        LD6: begin Zlowout = 1'b1; MARin = 1'b1; end
        LD7: begin Read = 1'b1; end
        LD8: begin Read = 1'b1; MDRin = 1'b1; end
        LD9: begin MDRout = 1'b1; Gra = 1'b1; Rin_ctrl = 1'b1; end

        ST4: begin Grb = 1'b1; BAout = 1'b1; Yin = 1'b1; end
        ST5: begin Cout = 1'b1; ALUop = 4'd3; Zlowin = 1'b1; end
        ST6: begin Zlowout = 1'b1; MARin = 1'b1; end
        ST7: begin Gra = 1'b1; Rout_ctrl = 1'b1; MDRin = 1'b1; end
        ST8: begin Write = 1'b1; end

        BR4: begin
            Gra       = 1'b1;
            Rout_ctrl = 1'b1;
            CONin     = 1'b1;
        end
        BR5: begin
            if (CONout) begin
                PCout = 1'b1;
                Yin   = 1'b1;
            end
        end
        BR6: begin
            if (CONout) begin
                Cout   = 1'b1;
                ALUop  = 4'd3;
                Zlowin = 1'b1;
            end
        end
        BR7: begin
            if (CONout) begin
                Zlowout = 1'b1;
                PCin    = 1'b1;
            end
        end

        JR4: begin
            Gra       = 1'b1;
            Rout_ctrl = 1'b1;
            PCin      = 1'b1;
        end

        JAL4: begin
            UseSelectEncode = 1'b0;
```

```
            Rin             = 16'h1000;
            PCout           = 1'b1;
        end
        JAL5: begin
            Gra       = 1'b1;
            Rout_ctrl = 1'b1;
            PCin      = 1'b1;
        end

        MUL4: begin Gra = 1'b1; Rout_ctrl = 1'b1; Yin = 1'b1; end
        MUL5: begin Grb = 1'b1; Rout_ctrl = 1'b1; ALUop = 4'd11; Zlowin = 1'b1;
Zhighin = 1'b1; end
        MUL6: begin Zhighout = 1'b1; HIin = 1'b1; end
        MUL7: begin Zlowout = 1'b1; LOin = 1'b1; end

        DIV4: begin Gra = 1'b1; Rout_ctrl = 1'b1; Yin = 1'b1; end
        DIV5: begin Grb = 1'b1; Rout_ctrl = 1'b1; ALUop = 4'd12; Zlowin = 1'b1;
Zhighin = 1'b1; end
        DIV6: begin Zhighout = 1'b1; HIin = 1'b1; end
        DIV7: begin Zlowout = 1'b1; LOin = 1'b1; end

        MFHI4: begin HIout = 1'b1; Gra = 1'b1; Rin_ctrl = 1'b1; end
        MFLO4: begin LOout = 1'b1; Gra = 1'b1; Rin_ctrl = 1'b1; end
        IN4:   begin InPortout = 1'b1; Gra = 1'b1; Rin_ctrl = 1'b1; end
        OUT4:  begin Gra = 1'b1; Rout_ctrl = 1'b1; OutPortin = 1'b1; end

        NOP4: begin
        end

        HALT_STATE: begin
            Run = 1'b0;
        end

        default: begin
            Run = 1'b0;
        end
    endcase
end

endmodule
```

cpu.v

```
`timescale 1ns/10ps

module cpu (
    input  wire        clock,
    input  wire        reset,
    input  wire        stop,
    input  wire [31:0] InPortData,
    input  wire        InPortStrobe,
```

```
    output wire [31:0] OutPortData,
    output wire        Run,
    output wire        CONout,
    output wire [31:0] IR_value,
    output wire [7:0]  state_dbg
);

wire        Read;
wire        Write;
wire [3:0]  ALUop;
wire [15:0] Rin;
wire [15:0] Rout;
wire        MARin;
wire        PCin;
wire        PCout;
wire        IRin;
wire        Yin;
wire        MDRin;
wire        MDRout;
wire        HIin;
wire        HIout;
wire        LOin;
wire        LOout;
wire        Zhighin;
wire        Zlowin;
wire        Zhighout;
wire        Zlowout;
wire        BAout;
wire        IncPC;
wire        UseSelectEncode;
wire        Gra;
wire        Grb;
wire        Grc;
wire        Rin_ctrl;
wire        Rout_ctrl;
wire        Cout;
wire        InPortout;
wire        OutPortin;
wire        CONin;

control_unit control_u (
    .clock(clock),
    .reset(reset),
    .stop(stop),
    .IR(IR_value),
    .CONout(CONout),
    .Run(Run),
    .Read(Read),
    .Write(Write),
    .ALUop(ALUop),
    .Rin(Rin),
```

```
    .Rout(Rout),
    .MARin(MARin),
    .PCin(PCin),
    .PCout(PCout),
    .IRin(IRin),
    .Yin(Yin),
    .MDRin(MDRin),
    .MDRout(MDRout),
    .HIin(HIin),
    .HIout(HIout),
    .LOin(LOin),
    .LOout(LOout),
    .Zhighin(Zhighin),
    .Zlowin(Zlowin),
    .Zhighout(Zhighout),
    .Zlowout(Zlowout),
    .BAout(BAout),
    .IncPC(IncPC),
    .UseSelectEncode(UseSelectEncode),
    .Gra(Gra),
    .Grb(Grb),
    .Grc(Grc),
    .Rin_ctrl(Rin_ctrl),
    .Rout_ctrl(Rout_ctrl),
    .Cout(Cout),
    .InPortout(InPortout),
    .OutPortin(OutPortin),
    .CONin(CONin),
    .state_dbg(state_dbg)
);

datapath datapath_u (
    .clock(clock),
    .clear(reset),
    .A(32'b0),
    .RegisterImmediate(32'b0),
    .Read(Read),
    .Write(Write),
    .ALUop(ALUop),
    .Rin(Rin),
    .Rout(Rout),
    .MARin(MARin),
    .PCin(PCin),
    .PCout(PCout),
    .IRin(IRin),
    .IRout(1'b0),
    .Yin(Yin),
    .Yout(1'b0),
    .MDRin(MDRin),
    .MDRout(MDRout),
    .HIin(HIin),
    .HIout(HIout),
```

```
    .LOin(LOin),
    .LOout(LOout),
    .Zhighin(Zhighin),
    .Zlowin(Zlowin),
    .Zhighout(Zhighout),
    .Zlowout(Zlowout),
    .BAout(BAout),
    .IncPC(IncPC),
    .UseSelectEncode(UseSelectEncode),
    .Gra(Gra),
    .Grb(Grb),
    .Grc(Grc),
    .Rin_ctrl(Rin_ctrl),
    .Rout_ctrl(Rout_ctrl),
    .Cout(Cout),
    .InPortout(InPortout),
    .OutPortin(OutPortin),
    .InPortStrobe(InPortStrobe),
    .InPortData(InPortData),
    .OutPortData(OutPortData),
    .CONin(CONin),
    .CONout(CONout),
    .IR_value(IR_value)
);

endmodule
```

cpu_de0_top.v

```
`timescale 1ns/10ps

module cpu_de0_top (
    input  wire        CLOCK_50,
    input  wire [3:0]  KEY,
    input  wire [9:0]  SW,
    output wire [9:0]  LEDR,
    output wire [6:0]  HEX0,
    output wire [6:0]  HEX1
);

    wire        reset;
    wire        stop;
    wire        cpu_clock;
    wire [31:0] in_port_data;
    wire [31:0] out_port_data;
    wire        run;

    assign reset = ~KEY[0];               // KEY0 active-low
    assign stop  = ~KEY[1];               // KEY1 active-low
    assign in_port_data = {24'b0, SW[7:0]};
```

```
    // Divide 50 MHz board clock for more observable board behavior.
    // Fcpu = 50 MHz / (2 * DIVIDE_BY) = 1 MHz when DIVIDE_BY = 25.
    clock_divider #(
        .DIVIDE_BY(25)
    ) clkdiv_u (
        .clk_in(CLOCK_50),
        // Keep divided clock running during reset so synchronous clears
        // in the datapath can actually sample reset on a clock edge.
        .reset(1'b0),
        .clk_out(cpu_clock)
    );

    cpu cpu_u (
        .clock(cpu_clock),
        .reset(reset),
        .stop(stop),
        .InPortData(in_port_data),
        .InPortStrobe(1'b1),              // continuously capture switches
        .OutPortData(out_port_data),
        .Run(run),
        .CONout(),
        .IR_value(),
        .state_dbg()
    );

    // Drive LEDR with a single assignment to avoid multiple-driver conflicts.
    assign LEDR = {4'b0, run, 5'b0};      // LEDR[5] = Run.Out indicator

    // Display Out.Port[7:0] on HEX1 HEX0
    hex_to_7seg h0 (.nibble(out_port_data[3:0]), .seg(HEX0));
    hex_to_7seg h1 (.nibble(out_port_data[7:4]), .seg(HEX1));

endmodule

module clock_divider #(
    parameter integer DIVIDE_BY = 25
) (
    input  wire clk_in,
    input  wire reset,
    output reg  clk_out
);
    integer count;

    initial begin
        count = 0;
        clk_out = 1'b0;
    end

    always @(posedge clk_in or posedge reset) begin
        if (reset) begin
            count <= 0;
            clk_out <= 1'b0;
```

```
        end
        else if (count == (DIVIDE_BY - 1)) begin
            count <= 0;
            clk_out <= ~clk_out;
        end
        else begin
            count <= count + 1;
        end
    end
endmodule

module hex_to_7seg (
    input  wire [3:0] nibble,
    output reg  [6:0] seg
);
    always @(*) begin
        case (nibble)
            4'h0: seg = 7'b1000000;
            4'h1: seg = 7'b1111001;
            4'h2: seg = 7'b0100100;
            4'h3: seg = 7'b0110000;
            4'h4: seg = 7'b0011001;
            4'h5: seg = 7'b0010010;
            4'h6: seg = 7'b0000010;
            4'h7: seg = 7'b1111000;
            4'h8: seg = 7'b0000000;
            4'h9: seg = 7'b0010000;
            4'hA: seg = 7'b0001000;
            4'hB: seg = 7'b0000011;
            4'hC: seg = 7'b1000110;
            4'hD: seg = 7'b0100001;
            4'hE: seg = 7'b0000110;
            4'hF: seg = 7'b0001110;
            default: seg = 7'b1111111;
        endcase
    end
endmodule
```

ld_tb.v

```
`timescale 1ns/10ps

// ld R7, 0x65 => R7 <- M[R0 + 0x65] = M[0x65]
// R0 is used as base (BAout gates 0 onto bus regardless of R0 contents).
// Expected: R7 = 0x00000084 (preloaded at RAM[0x65] in datapath.v).

module ld_tb;

    reg Clock, clear;

    reg [15:0] Rin, Rout;
```

```
reg PCin, PCout, MARin;
reg MDRin, MDRout;
reg IRin;
reg Yin, Yout;
reg Read, Write;
reg [3:0] ALUop;
reg BAout;
reg Zin, Zlowout;
reg IncPC;

// select/encode controls
reg UseSelectEncode;
reg Gra, Grb, Grc;
reg Rin_ctrl, Rout_ctrl, Cout;

// CON FF
reg CONin;
wire CONout;

// I/O port controls
reg InPortout, OutPortin, InPortStrobe;
reg [31:0] InPortData;
wire [31:0] OutPortData;

// ALU opcodes (must match alu.v case statement)
localparam ALU_ADD = 4'd3;

// ld R7, 0x65: IR[26:23]=Ra=7, IR[22:19]=Rb=0, IR[18:0]=C=0x65
localparam [31:0] LD_INSTR = 32'h03800065;
// RAM[0x65] = 32'h00000084 is preloaded in ram_512x32.v.

parameter Default=4'd0, T0=4'd1, T1=4'd2, T2=4'd3, T3=4'd4,
          T4=4'd5, T5=4'd6, T6=4'd7, T7=4'd8, T8=4'd9, T9=4'd10;

reg [3:0] Present_state = Default;

datapath DUT (
    .clock(Clock),
    .clear(clear),
    .A(32'b0),
    .RegisterImmediate(32'b0),
    .Read(Read),
    .Write(Write),
    .ALUop(ALUop),
    .Rin(Rin),
    .Rout(Rout),
    .MARin(MARin),
    .PCin(PCin),
    .PCout(PCout),
    .IRin(IRin),
    .IRout(1'b0),
```

```
        .Yin(Yin),
        .Yout(Yout),
        .MDRin(MDRin),
        .MDRout(MDRout),
        .HIin(1'b0),
        .HIout(1'b0),
        .LOin(1'b0),
        .LOout(1'b0),
        .Zhighin(1'b0),
        .Zlowin(Zin),
        .Zhighout(1'b0),
        .Zlowout(Zlowout),
        .BAout(BAout),
        .IncPC(IncPC),
        .UseSelectEncode(UseSelectEncode),
        .Gra(Gra),
        .Grb(Grb),
        .Grc(Grc),
        .Rin_ctrl(Rin_ctrl),
        .Rout_ctrl(Rout_ctrl),
        .Cout(Cout),
        .InPortout(InPortout),
        .OutPortin(OutPortin),
        .InPortStrobe(InPortStrobe),
        .InPortData(InPortData),
        .OutPortData(OutPortData),
        .CONin(CONin),
        .CONout(CONout)
    );

    // Preload RAM[0] with ld instruction. RAM[0x65] is preloaded in
datapath.v.
    initial begin
        DUT.mem_sys.ram_inst.memory[9'h000] = LD_INSTR;
    end

    initial begin
        Clock = 0;
        forever #10 Clock = ~Clock;
    end

    always @(posedge Clock) begin
        if (clear)
            Present_state <= Default;
        else begin
            case (Present_state)
                Default : Present_state <= T0;
                T0      : Present_state <= T1;
                T1      : Present_state <= T2;
                T2      : Present_state <= T3;
                T3      : Present_state <= T4;
                T4      : Present_state <= T5;
```

```
                T5      : Present_state <= T6;
                T6      : Present_state <= T7;
                T7      : Present_state <= T8;
                T8      : Present_state <= T9;
                T9      : Present_state <= T9; // hold
            endcase
        end
    end

    always @(*) begin
        Rin  = 16'b0; Rout = 16'b0;
        PCin = 0; PCout = 0; MARin = 0;
        MDRin = 0; MDRout = 0; IRin = 0;
        Yin = 0; Yout = 0;
        Zin = 0; Zlowout = 0;
        Read = 0; Write = 0; BAout = 0;
        ALUop = 4'b0; IncPC = 0;
        UseSelectEncode = 1;
        Gra = 0; Grb = 0; Grc = 0;
        Rin_ctrl = 0; Rout_ctrl = 0; Cout = 0;
        CONin = 0;
        InPortout = 0; OutPortin = 0;
        InPortStrobe = 0; InPortData = 32'b0;

        case (Present_state)
            T0: begin // PCout, MARin, IncPC, Zin
                PCout = 1; MARin = 1; IncPC = 1; ALUop = ALU_ADD; Zin = 1;
            end
            T1: begin // Zlowout, PCin, Read
                Zlowout = 1; PCin = 1; Read = 1;
            end
            T2: begin // Read, MDRin  (MDR latches LD_INSTR from RAM)
                Read = 1; MDRin = 1;
            end
            T3: begin // MDRout, IRin
                MDRout = 1; IRin = 1;
            end
            T4: begin // Grb, BAout, Yin
                Grb = 1; BAout = 1; Yin = 1;
            end
            T5: begin // Cout, ADD, Zin
                Cout = 1; ALUop = ALU_ADD; Zin = 1;
            end
            T6: begin // Zlowout, MARin
                Zlowout = 1; MARin = 1;
            end
            T7: begin // Read  (RAM reads memory[0x65] at posedge T7)
                Read = 1;
            end
            T8: begin // Read, MDRin  (MDR latches 0x84)
                Read = 1; MDRin = 1;
            end
```

```
            T9: begin // MDRout, Gra, Rin
                MDRout = 1; Gra = 1; Rin_ctrl = 1;
            end
        endcase
    end

    initial begin
        clear = 1;
        #20 clear = 0;
    end

    initial begin
        #500;
        $finish;
    end

endmodule
```

cpu_tb.v

```
`timescale 1ns/10ps

module cpu_tb;

    reg Clock;
    reg reset;
    reg stop;
    reg [31:0] InPortData;
    reg InPortStrobe;

    wire [31:0] OutPortData;
    wire Run;
    wire CONout;
    wire [31:0] IR_value;
    wire [7:0] state_dbg;

    integer cycles;
    integer error_count;

    localparam integer MAX_CYCLES = 800;

    cpu DUT (
        .clock(Clock),
        .reset(reset),
        .stop(stop),
        .InPortData(InPortData),
        .InPortStrobe(InPortStrobe),
        .OutPortData(OutPortData),
        .Run(Run),
        .CONout(CONout),
```

```
        .IR_value(IR_value),
        .state_dbg(state_dbg)
    );

    initial begin
        Clock = 0;
        forever #10 Clock = ~Clock;
    end

    task check_value;
        input [8*32-1:0] name;
        input [31:0] actual;
        input [31:0] expected;
        begin
            if (actual !== expected) begin
                error_count = error_count + 1;
                $display("cpu_tb FAILED: %0s = %h expected %h", name, actual,
expected);
            end
            else begin
                $display("cpu_tb PASSED: %0s = %h", name, actual);
            end
        end
    endtask

    initial begin
        reset = 1'b1;
        stop = 1'b0;
        InPortData = 32'b0;
        InPortStrobe = 1'b0;
        cycles = 0;
        error_count = 0;


        #35;
        reset = 1'b0;
    end

    initial begin
        @(negedge reset);

        while ((Run !== 1'b0) && (cycles < MAX_CYCLES)) begin
            @(posedge Clock);
            cycles = cycles + 1;
        end

        if (Run !== 1'b0) begin
            $display("cpu_tb FAILED: CPU did not halt within %0d cycles",
MAX_CYCLES);
            $display("cpu_tb INFO: state=%0d IR=%h PC=%h", state_dbg, IR_value,
DUT.datapath_u.PC_data_out);
            $finish;
```

```
        end

        #1;
        $display("cpu_tb INFO: CPU halted after %0d cycles", cycles);
        $display("cpu_tb INFO: halt state=%0d IR=%h PC=%h", state_dbg,
IR_value, DUT.datapath_u.PC_data_out);

        check_value("R0",  DUT.datapath_u.R0_data_out,  32'h00000614);
        check_value("R1",  DUT.datapath_u.R1_data_out,  32'h00000000);
        check_value("R2",  DUT.datapath_u.R2_data_out,  32'h00000004);
        check_value("R3",  DUT.datapath_u.R3_data_out,  32'h00000019);
        check_value("R4",  DUT.datapath_u.R4_data_out,  32'h00006800);
        check_value("R5",  DUT.datapath_u.R5_data_out,  32'h00000680);
        check_value("R6",  DUT.datapath_u.R6_data_out,  32'h000000AF);
        check_value("R7",  DUT.datapath_u.R7_data_out,  32'h00000007);
        check_value("R8",  DUT.datapath_u.R8_data_out,  32'h00000009);
        check_value("R9",  DUT.datapath_u.R9_data_out,  32'h00000015);
        check_value("R10", DUT.datapath_u.R10_data_out, 32'h000000B2);
        check_value("R11", DUT.datapath_u.R11_data_out, 32'h00000005);
        check_value("R12", DUT.datapath_u.R12_data_out, 32'h00000029);
        check_value("R13", DUT.datapath_u.R13_data_out, 32'h00000010);
        check_value("R14", DUT.datapath_u.R14_data_out, 32'h000000AB);
        check_value("HI",  DUT.datapath_u.HI_data_out,  32'h00000004);
        check_value("LO",  DUT.datapath_u.LO_data_out,  32'h00000003);

        check_value("M[0x89]", DUT.datapath_u.mem_sys.ram_inst.memory[9'h089],
32'h0000006C);
        check_value("M[0xA3]", DUT.datapath_u.mem_sys.ram_inst.memory[9'h0A3],
32'h00000008);

        if (error_count == 0)
            $display("cpu_tb PASSED: full Phase 3 program completed
successfully");
        else
            $display("cpu_tb FAILED: %0d mismatches detected", error_count);

        $finish;
    end

endmodule
```