尊嘟假嘟组

Design Report

Creation Date：2023.8.31

Version： 6.0

The project is open sourced at: Cao1014/16-bits-multi-cycle-CPU: CPU Design project for the course "Application and Design of Digital Logic" at Glasgow College, UESTC . (github.com)

University of Electronic Science and Technology of China

Chengdu, Sichuan

Document History

Date	Version	Change	Editor
20230906	1.0	Update Specification and Overall Design for Design Report v1.0	曹正阳
20230920	2.0	Update Datapath Submodule and Modify Section 2 for Design Report v2.0	张睿宁
20231011	3.0	Update Control Unit and Modify the Datapath Submodule for Design Report v3.0	张鹏飞
20231018	4.0	Update Ins_ROM, Data_RAM module submodules, Clock, Reset & Initial part and Led_display module, Modify the Control Unit Submodule for Design Report v4.0	吴高翔
20231115	5.0	Update Verification & Test and Modify Previous Content for Design Report v5.0	李奕廷
20231201	6.0	Update Conclusion & Future work, Revise and Typeset the Entire Content for Design Report v6.0	全体

Abbreviation

Abbreviation	Full Name
CPU	Central Processing Unit
PC	Program Counter
ALU	Arithmetic Logic Unit
IR	Instruction Register
I/O	Input/Output
ROM	Read Only Memory
RAM	Random Access Memory
ILA	Integrated Logic Analyzer

Abstract

This project outlines the successful design and implementation of a 16-bit multi-cycle CPU, featuring arithmetic, logic, jump, and load/store instructions. Utilizing Vivado simulation, RTL circuit synthesis, and on-board verification with a Zynq7000 FPGA development board, the CPU's functionality was demonstrated, and results were displayed through LEDs and a seven-segment digital display tube. With key metrics such as a 16-bit width, multi-cycle operation, 16 instructions, and a Harvard architecture, the CPU finds applications in embedded systems. The user manual provides insights into power consumption, signals, supported instructions, word length, memory connection, and peripherals. The project's success was confirmed through successful simulation, synthesis, and verification aligned with expectations. Future work may involve enhancing the CPU with button-controlled jumps, conditional branching, and potential pipelining through D-triggers, aiming to broaden its capabilities and performance.

Key Words: 16-bit multi-cycle CPU, Vivado simulation, load/store, Zynq7000 FPGA board

Contents

Figures

Tables

1. Team information

Team Name	尊嘟假嘟组			
Team Members	**Student number**	**Name**	**Task**	**Scores distribution**
	2021190905003	曹正阳	Overall and LED peripheral design & Debug	2
	2021190905027	李奕廷	Instruction set design & Testbench design	2
	2021190905016	吴高翔	Rom/Ram and LED peripheral design & Test	2
	2021190905029	张睿宁	Datapath design & Test	2
	2021190905007	张鹏飞	Controller design & Debug	2

2. Specification

2.1. Requirements analysis

The purpose of this project [1] is to design a 16 bits multi-cycle CPU [1], its basic functions include arithmetic operations, logic operations, jump instructions and Load/Store instructions. We want it to interact with external registers to load and store data. We plan to use Vivado simulation first and then synthesize the RTL circuit. Finally, the on-board verification is completed on the Zynq7000 FPGA development board, and the results of the operation are displayed on its LED and seven-segment digital display tube. Here are some basic design metrics for our CPU.

Table 1 Basic design metrics of our CPU

Basic design metrics	Description
Width	16 bits
Working Type	Multicycle
Number of ins	16
Pipeline	Non-pipeline
Architecture	Harvard

2.2. Application Scenarios

This CPU can be used in many embedded systems, such as counters, calculators, or low-power devices such as game consoles. A simple user manual is as follows.

Table 2 User Manual

Total On-chip Power	0.109W
Input Signals	clk, rst, en_in
Output Signals	led_flowing
Supported instructions	16 instructions, see "Instruction set" for details
Word length	16bits
Memory connection	Harvard
Peripheral	1*Rom, 1*Ram, 1*Led_flowing_reg

3. Design & Implementation

3.1. Overall

3.1.1. Function

The produce can be used to execute 16 instructions based on RISC-V [1]. The instruction set is shown in Table 3.

Table 3 Instruction set

opcode	rd	rs	imm	asm	description
0000	rd	unused	imm	movi rd, imm	R[rd]=imm=imm+0
0001	rd	rs	unused	mov rd, rs	R[rd]=R[rs]=R[rs]+0
0010	rd	unused	imm	addi rd, imm	R[rd]=R[rd]+imm
0011	rd	rs	unused	add rd, rs	R[rd]=R[rd]+R[rs]
0100	rd	rs	unused	sub rd, rs	R[rd]=R[rd]-R[rs]
0101	rd	unused	imm	andi rd, imm	R[rd]=R[rd]&imm
0110	rd	rs	unused	and rd, rs	R[rd]=R[rd]&R[rs]
0111	rd	rs	unused	or rd, rs	R[rd]=R[rd] | R[rs]
1000	rd	unused	unused	not rd	R[rd]= ~R[rd]
1001	rd	rs	unused	xor rd, rs	R[rd]=R[rd]^R[rs]
1010	rd	unused	imm	slli rd, imm	R[rd]=R[rd]<<imm
1011	rd	unused	imm	srli rd, imm	R[rd]=R[rd]>>imm
1100	rd	unused	imm	srai rd, imm	R[rd]=R[rd]>>> imm
1101	unused	unused	imm	jump imm	pc=imm
1110	rd	unused	imm	ld rd, imm	R[rd]=Mem[imm]
1111	rd	unused	imm	st rd, imm	Mem[imm]=rd

Illustration:

The initial four digits represent the opcode, determining the instruction to be executed. Following this, the subsequent four digits align with the register's position, determining the specific register to access and store.

The final eight digits, known as immediate numbers, supply a value under particular conditions.

3.1.2. Input/Output

The interface of this chip is defined in Table 4.

Table 4 Input/Output signals

Signal name	I/O	Width (bits)	Function Description
clk	I	1	clk signal (posedge)
rst	I	1	asynchronous system reset (negedge)
en_in	I	1	system enable (active high)
led_flowing	O	8	the output of the cpu (active high), which is intended to drive the eight LED pins of the FPGA

3.1.3. Architecture

The Architecture of this CPU can be summarized as: Control unit + Datapath + Ins_Rom + Data_Ram + Led_flowing_reg [1]. The top level hardware structure is shown in Figure 1.

错误!未找到引用源。

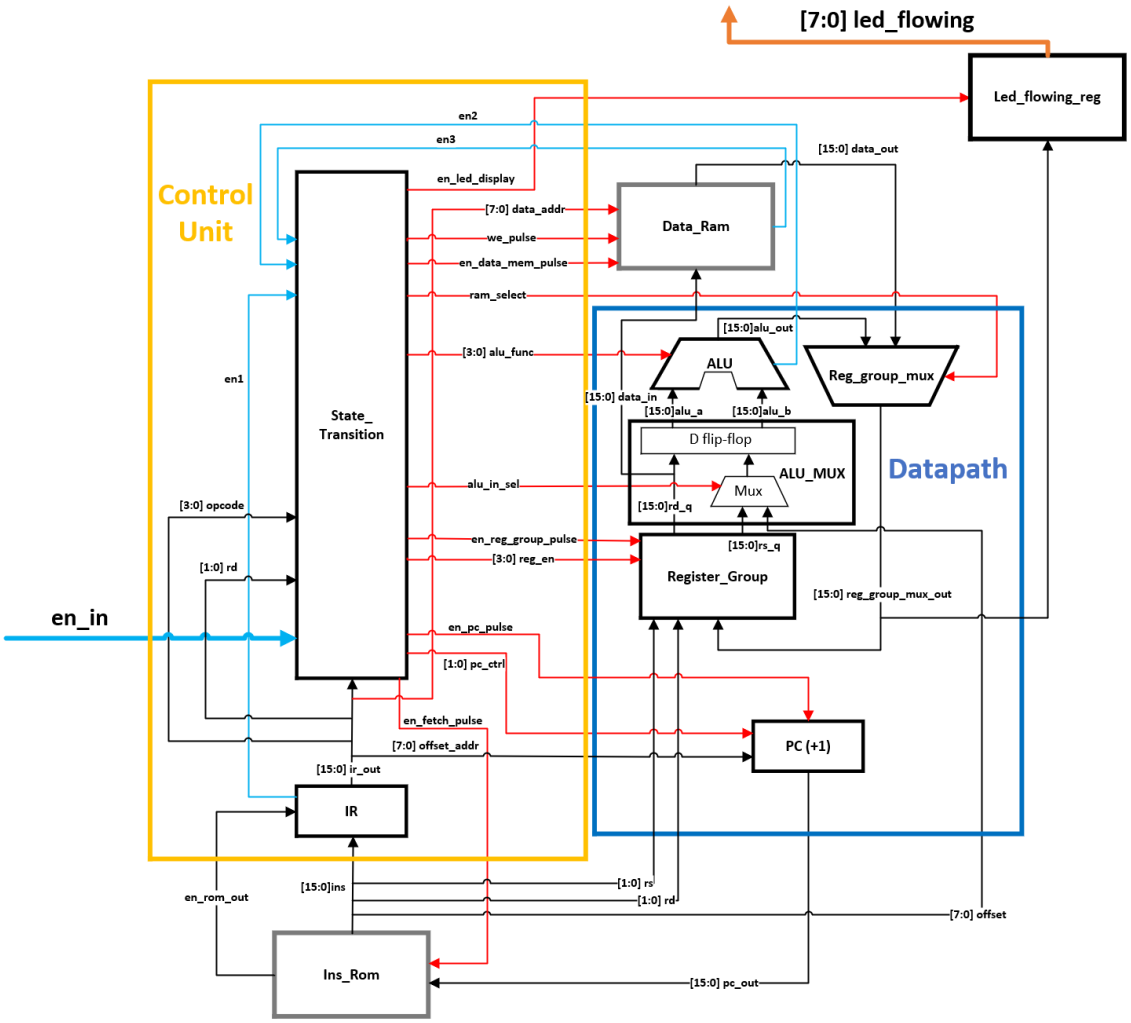

Figure 1 Top Level Hardware Architecture

3.1.4. Dataflow

Dataflows for different instructions are shown in Figure 2-5.

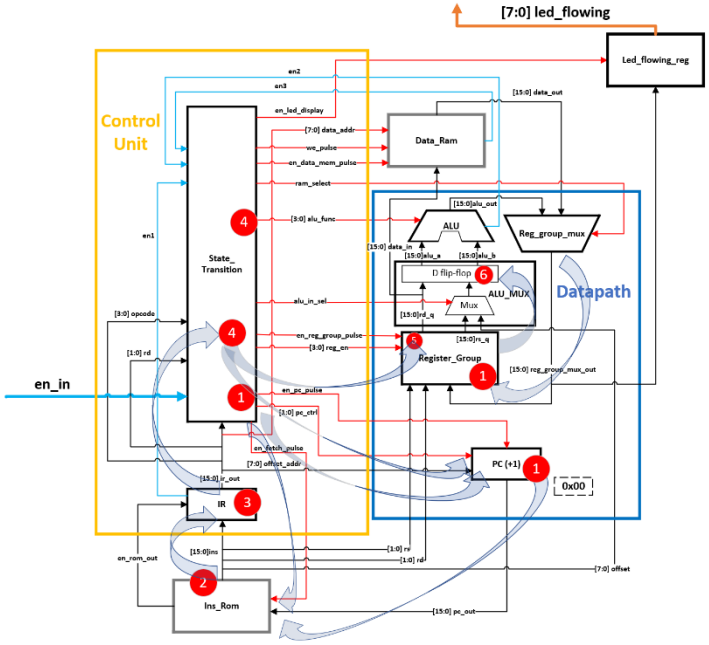

Figure 2 Dataflow for Arithmetic and Logic Instructions

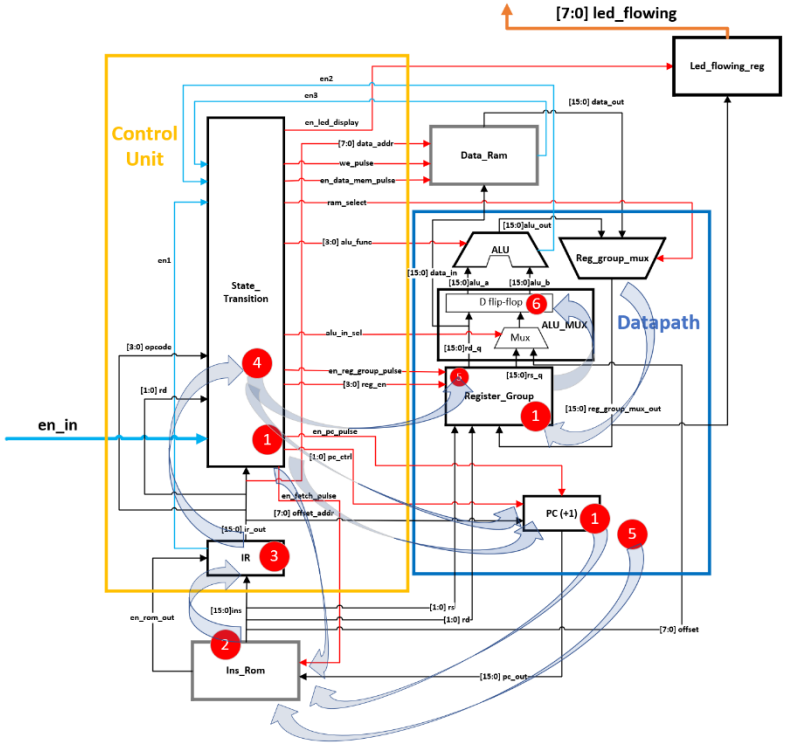

Figure 3 Dataflow for Jump Instruction

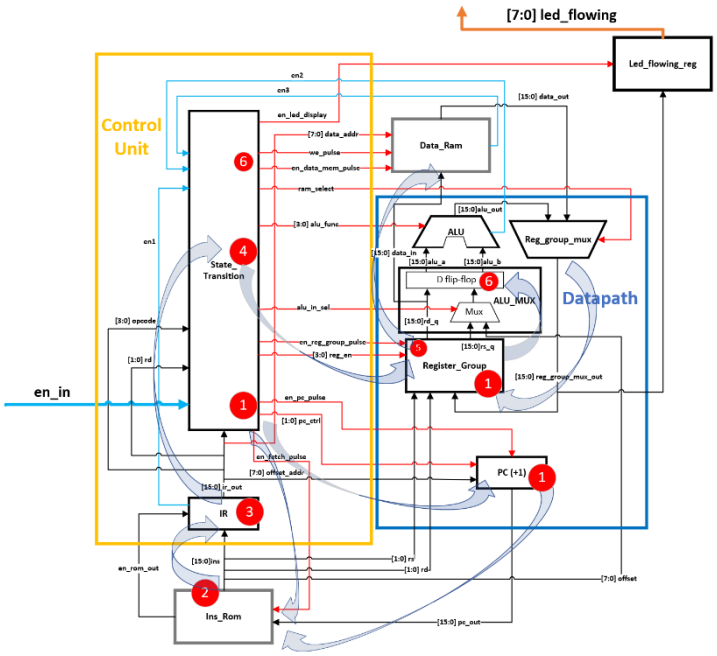

Figure 4 Dataflow for Load Instruction

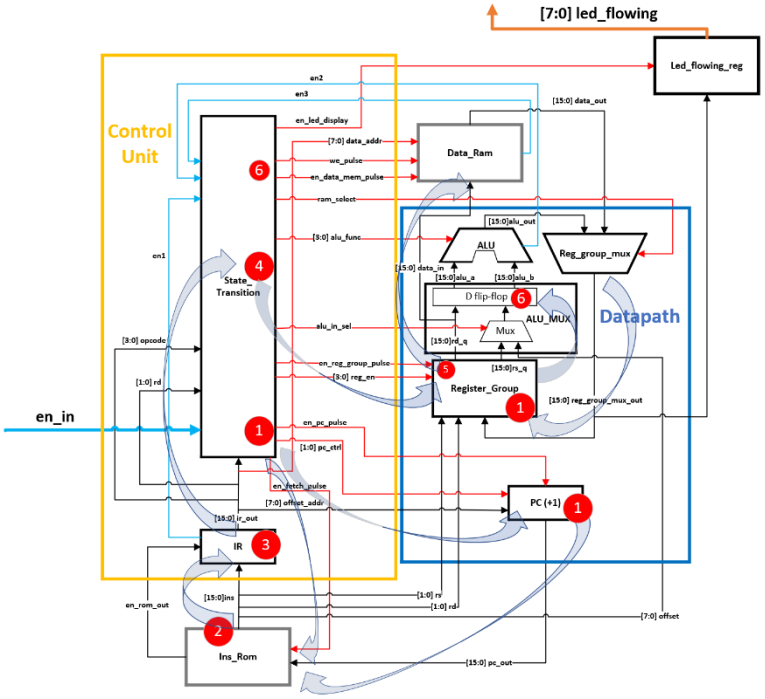

Figure 5 Dataflow for Store Instruction

3.1.5. Code Excerpt

```
module cpu(clk, rst, en_in, led_flowing);   //Defines the I/O of the cpu

input clk, rst, en_in;
output reg [7:0] led_flowing;
```

```
// ILA (Integrated Logic Analyzer) for debugging purposes
ila_0 my_lia_debug (
    .clk(clk), // input wire clk
    .probe0(rst), // input wire [0:0]   probe0
    .probe1(clk_slow), // input wire [0:0]   probe1
    .probe2(reg_group_mux_out), // input wire [15:0]   probe2
    .probe3(led_flowing) // input wire [7:0]   probe3
);

// Instantiate components
data_path data_path1 (......);
control_unit control_unit1(......);
instruction_memory_module instruction_memory1 (......);
reg_group_mux reg_group_mux1(......);
data_memory_module data_memory1(......);

endmodule
```

3.2. Datapath submodule

3.2.1. Function

The datapath module is responsible for the data operation, including three sub-sub modules, which are program counter, register group, algorithm and logic operation unit (ALU) and ALU multiplexer [1]. These sub-sub modules allow the datapath section to carry out logic and algorithm operations listed in the instruction set, enable it to store calculation results, write to or read from the random access memory (RAM) and determine the address of the next instruction for the CPU.

3.2.2. Input/Output

The I/O of Datapath is defined in Table 5.

Table 5 Input/Output signals of Datapath

Signal name	I/O	Width (bits)	Function Description
alu_func	I	4	Determine the specific type of operation. It is just the same as the opcode, corresponding to specific command.
alu_in_sel	I	1	Determine the second operand between immediate number and number from register. When it equals to 1, choose number from register.
reg_en	I	4	One-hot code to determine the register that will be written. r1 corresponds to 0001, r2 meaning 0010, and so on.
rd	I	2	Address of register containing the first operand and when rd is unused it will be set to 0
rs	I	2	Address of register containing the second operand. When rs is unused it will be set to 0.
offset	I	8	The immediate number, possibly serving as the second

			operand.
offset_addr	I	8	Immediate number that serves as the program counter's address increment when the instruction is JUMP.
pc_ctrl	I	2	The control signal to determine whether the program counter auto-incrementing or adding the immediate number.
en_pc_pulse	I	1	Enable signal to activate the program counter.
en_in	I	1	Enable signal to activate the register group.
clk	I	1	Clock signal, positive edge effective.
rst	I	1	Asynchronous reset signal (negedge)
pc_out	O	16	The address output from the program counter.
en_out	O	1	When alu is enabled, return en_out=1 to the state machine.
rd_data	O	16	The updated data of the rd register, which is used as input to the dataram.
reg_group_mux_out	I	16	The result of reg_group_mux, which is obtained from either alu_out or data_out.
ram_select	I	1	Control signal choose the source o f the data that will be written to the register group. between the output of ALU and the output of RAM. When 0, ALU output is chosen.

3.2.3. Architecture

The datapath structure is shown in Figure 6.

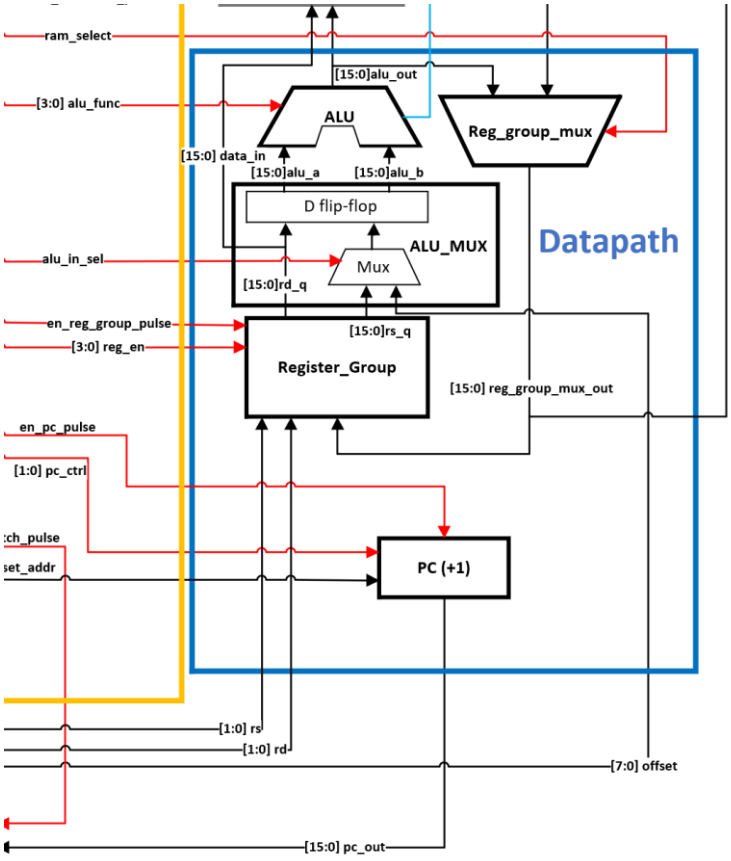

Figure 6 Datapath Architecture

3.2.4. Dataflow

The dataflow for adding instruction is shown in Figure 7.

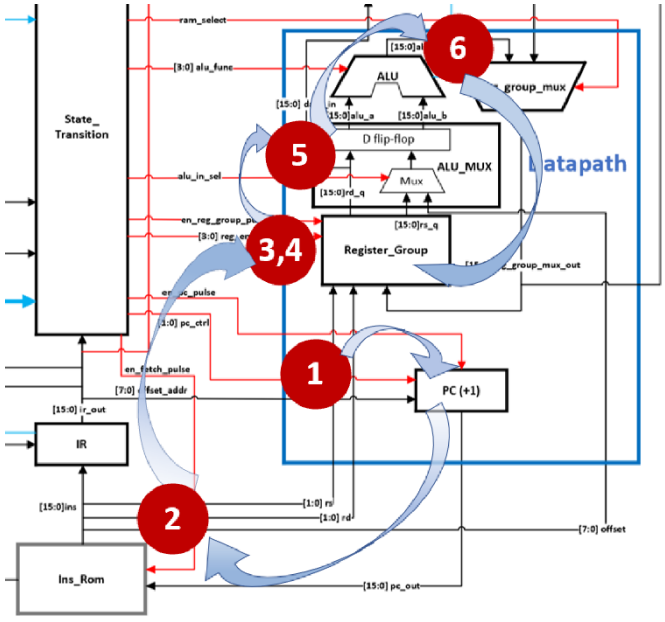

Figure 7 Dataflow chart

3.2.5. Program Counter (PC) sub-submodule

3.2.5.1. Function:

Determine and store the address of next instruction, finally sending the address to the Read Only Memory (ROM).

3.2.5.2. Input/output:

The I/O of Program Counter is defined in Table 6.

Table 6 Input/Output signals of Program Counter

Signal name	I/O	Width (bits)	Function Description
clk	I	1	Clock signal (posedge)
rst	I	1	Asynchronous reset signal (negedge)
en_in	I	1	Enable signal to activate PC.
pc_ctrl	I	2	The control signal to determine whether the program counter auto-incrementing or adding the immediate number. Detailed information is show below.
offset_addr	I	8	Immediate number that serves as the program counter's address increment according to the command.
pc_out	O	16	The next instruction that will be executed.

The detailed signal set is shown in Figure 8.

pc_ctrl	pc_out
00	pc_out
01	pc_out +1
10	offset_addr

Figure 8 pc_ctrl signal and its corresponding output

3.2.5.3. Architecture:

PC architecture is shown in the Figure 9.

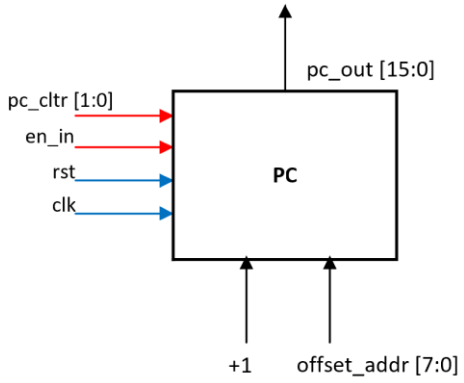

Figure 9 PC architecture

3.2.5.4. Dataflow

PC dataflow is shown in the Figure 10.

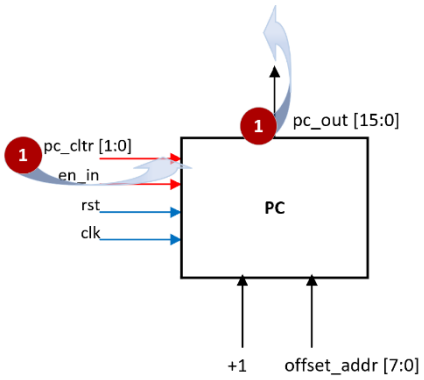

Figure 10 PC dataflow

3.2.5.5. Code excerpt:

```
// program counter output address according to the control signal
case (pc_ctrl)
        2'b01:
        begin
          pc_out <= pc_out + 1;   //en_pc_out <= 0;
        end
        2'b10:
        begin
          pc_out <= {8'b00000000,offset_addr[7:0]};   //en_pc_out <= 1;
        end
        default:
        begin
          pc_out <= pc_out; //en_pc_out <= 0;
        end
      endcase
```

3.2.6. Register group sub-submodule

3.2.6.1. Function:

Store the results of arithmetic and logic operations, store the data loaded from the RAM, keep the operands from the instruction and provide ALU module with stored operands.

3.2.6.2. Input/ Output:

The I/O of register group is defined in Table 7.

Table 7 Input/Output signals of register group

Signal name	I/O	Width (bits)	Function Description
clk	I	1	Clock signal (posedge)
rst	I	1	Asynchronous reset signal (negedge)
en_in	I	1	Enable signal to activate register group
reg_en	I	4	One-hot code to determine the register that will be written. r1 corresponds to 0001, r2 meaning 0010, and so on.

d_in	I	16	Data that will be written to the register group
rs	I	2	Address of register containing the second operand and the detailed address set is shown as follows. When rs is unused it will be set to 0.
rd	I	2	Address of register containing the first operand and the detailed address set is shown as follows. When rd is unused it will be set to 0.
rd_q	O	16	When rd is used, it is the first operand or it is set to 0
rs_q	O	16	When rs is used, it is the second operand or it is 0.

Detailed codes of registers are shown in Figure 11.

rd/rs	rd_q/rs_q
00	R0
01	R1
02	R2
03	R3

Figure 11 Register code set

3.2.6.3. Architecture

Register group architecture is shown in the Figure 12.

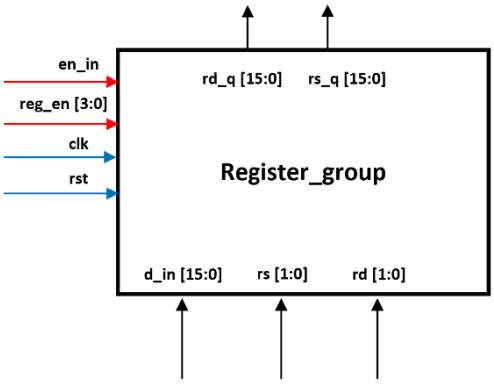

Figure 12 Register Group Architecture

3.2.6.4. Dataflow:

Register Group dataflow is shown in the Figure 13.

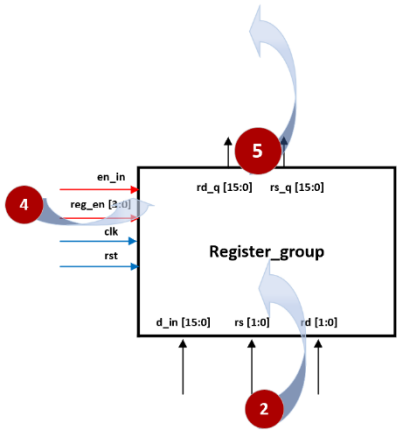

Figure 13 Register Group dataflow

3.2.6.5. Code:

```
case({rd[1:0],rs[1:0]})
   4'b0000:
   begin
      rd_q <= q0;
      rs_q <= q0;
   end
   4'b0001:
   begin
        rd_q <= q0;
        rs_q <= q1;
   end
   4'b0010:
   begin
        rd_q <= q0;
        rs_q <= q2;
   end
  ...... ......
   default:
   begin
      rd_q <= 0000000000000000;
      rs_q <= 0000000000000000;
   end
 endcase
```

3.2.7. ALU&ALU mux sub-submodule

3.2.7.1. Functions:

(1) ALU_Mux:

Choose the second operand between the output rs from register group and the immediate number offset.

(2) ALU:

Logical and numerical operations according to the specific instruction executed.

3.2.7.2. Input/Output

The I/O of ALU_MUX and ALU are defined in Table 8 and Table 9.

(1) ALU_MUX

Table 8 Input/Output signals of ALU_MUX

Signal Name	I/O	Width (bits)	Function description
clk	I	1	Clock signal (posedge)
rst	I	1	Asynchronous reset signal (negedge)
rd_q	I	16	The first operand from the register group. When rd is unused it is set to 0.
rs_q	I	16	Register output, possibly the second operand.
offset	I	8	Immediate number from the instruction.
alu_in_sel	I	1	Determine the second operand between immediate number and register output. When it equals to 1, choose number from register.
alu_a	O	16	The first operand
alu_b	O	16	The second operand, either the immediate number or the register output.

(2) ALU

Table 9 Input/Output signals of ALU

Signal Name	I/O	Width (bits)	Function description
rst	I	1	Asynchronous reset signal (negedge)
alu_func	I	4	Determine the specific type of operation. It is just the same as the opcode, corresponding to specific command.
alu_a	I	16	The first operand
alu_b	I	16	The second operand
alu_out	O	16	Final operation results. For Jump, Load and Store instructions, it is 0.

3.2.7.3. Architecture

ALU & ALU_MUX architecture are shown in the Figure 14.

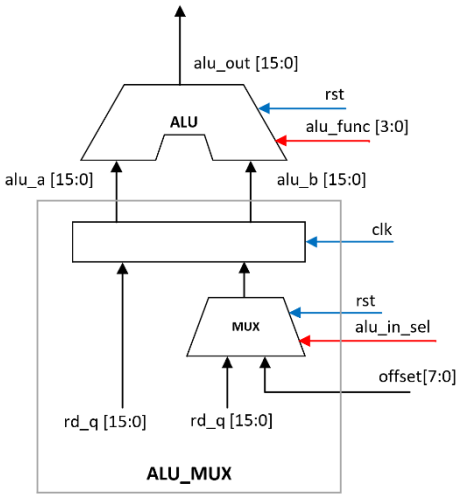

Figure 14 ALU & ALU_MUX architecture

3.2.7.4. Dataflow

ALU & ALU_MUX dataflow is shown in the Figure 15.

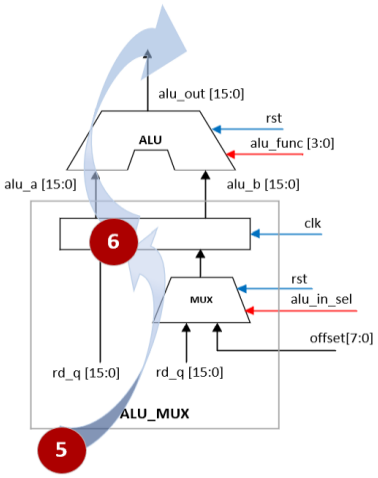

Figure 15 ALU & ALU_MUX dataflow

3.2.7.5. Code excerpt:

(1) ALU_MUX

```
if(alu_in_sel ==1'b0)
    alu_b <= {{8{offset[7]}},offset[7:0]};
else
    alu_b <= rs_q;
```

(2) ALU

```
case(alu_func)
                        `movi: alu_out = alu_b;
                        `mov: alu_out = alu_b;
                        `addi: alu_out = alu_a + alu_b;
                        `add: alu_out = alu_a + alu_b;
```

```
            `sub: alu_out = alu_a - alu_b;
            `andi: alu_out = alu_a & alu_b;
            `and_: alu_out = alu_a & alu_b;
            `or_: alu_out = alu_a | alu_b;
            `not_: alu_out = ~alu_a;
            `xor_: alu_out = alu_a ^ alu_b;
            `slli: alu_out = alu_a << alu_b;
            `srli: alu_out = alu_a >> alu_b;
            `srai: alu_out = alu_a >>> alu_b;
            `jump: alu_out = 16'b0000000000000000;
            `load: alu_out = 16'b0000000000000000;
            `store: alu_out = 16'b0000000000000000;
        default: alu_out   = 16'b0000000000000000;
        endcase
```

3.2.8. Reg _group_mux sub-submodule

3.2.8.1. Function

Determine the data that will be written to the register group, which selects between the RAM and ALU output.

3.2.8.2. Input/Output

The I/O of reg_group_mux_out are defined in Table 10.

Table 10 Input/Output signals of reg_group_mux_out

Signal Name	I/O	Width (bits)	Function Description
ram_select	I	1	Control signal selecting the data that will be written to the register group between the output of ALU and the output of RAM. When 0, ALU output is chosen.
alu_out	I	16	Output from ALU
data_out	I	16	Output from RAM
reg_group_mux_out	O	16	The chosen result which is written to the register group

3.2.8.3. Architecture

The reg_group_mux_out arhietecture is shown in Figure 16.

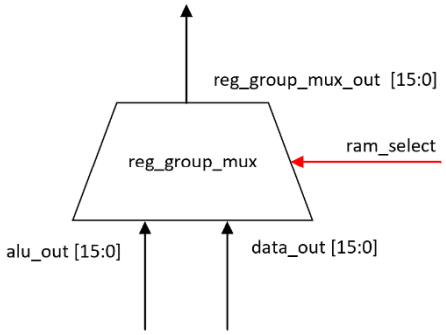

Figure 16 reg_group_mux_out arhietecture

3.2.8.4. Dataflow

The reg_group_mux_out dataflow is shown in the Figure 17.

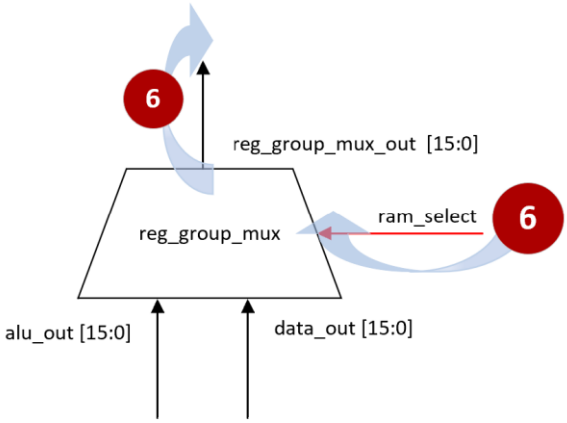

Figure 17 reg_group_mux_out dataflow

3.2.8.5. Code

```
if(ram_select==1'b0)
                    begin
                        reg_group_mux_out = alu_out;
                    end
                else
                    begin
                        reg_group_mux_out = data_out;
                    end
```

3.2.9. Code Excerpt for Datapath

```
module data_path ( clk, rst, reg_group_mux_out, offset_addr, en_pc_pulse, pc_ctrl,
offset,                                    en_in, reg_en, alu_in_sel, alu_func, en_out, pc_out, rd,
rs,
                rd_data, alu_out);
```

```
pc pc1(
        .clk(clk),
        .rst(rst),
        .en_in(en_pc_pulse),
        .pc_ctrl(pc_ctrl),
        .offset_addr(offset_addr),
        .pc_out(pc_out)
    );

reg_group reg_group1( ..... );
alu_mux alu_mux1( ..... );
alu alu1 ( ..... );
```

3.3. Control Unit submodule

3.3.1. Function

The Control Unit (CU) is a key component of a CPU responsible for coordinating and managing the execution of instructions [1]. Its main functions include:

Instruction Fetch: The Control Unit retrieves instructions from the rom, typically, and prepares them for execution.

Instruction Decoding: It decodes the instructions obtained from memory to determine the specific operation to be performed and the operands involved.

Instruction Execution: The Control Unit directs the execution of the decoded instructions by issuing control signals to other parts of the CPU, such as the Arithmetic Logic Unit (ALU) and the registers.

1. Timing and Control: It manages the timing and sequencing of operations within the CPU, ensuring that instructions are executed in the correct order and at the appropriate times.
2. Coordination of Data Movement: The Control Unit oversees the movement of data between the CPU's various components, including registers, ALU, and memory, to facilitate the execution of instructions.

In summary, the Control Unit plays a crucial role in managing the flow of instructions and data within the CPU, ensuring that instructions are fetched, decoded, and executed in a coordinated and timely manner.

3.3.2. Input/Output

Table 11 Input/Output signals of Control Unit

Signal name	I/O	Width (bits)	Function Description
clk	I	1	Clock signal (posedge)
rst	I	1	Asynchronous reset signal (negedge)
ins	I	16	Instruction input according to the instruction set
en	I	1	Enable signal to activate the module
en_alu	I	1	Enable signal to indicate alu has finished operation demanded by instruction

en_ram_out	I	1	Enable signal that indicates ram has sent the value that needs loading
en_rom_out	I	1	Enable signal that indicates rom has sent the instruction into control unit
offset_addr	O	8	The immediate number in the command that will be sent to the program counter
data_addr	O	8	Output the read/write address to ram
alu_func	O	4	Determine the specific type of operation that will be executed. It is just the same as the opcode, so operation selected just corresponds to the command
alu_in_sel	O	1	Determine the second operand between immediate number and number from register
pc_ctrl	O	2	The control signal to determine whether the program counter auto-incrementing or adding the immediate number
reg_en	O	4	Address signal into register group to determine which register should receive the value from operation or load
en_rom_in	O	1	Enable the instruction rom
en_group_pulse	O	1	Enable the register_ group module
en_pc_pulse	O	1	Enable the program counter module
en_data_mem_pulse	O	1	Enable the ram
we_pulse	O	1	0 controls ram read, 1 controls ram write
ram_select	O	1	0 controls that reg_group_mux selects data from alu and 1 controls that reg_group_mux selects data from ram
en_led_display	O	1	Enable the update of LED lights

3.3.3. Architecture

The control unit structure is shown in Figure 18 and Figure 19.

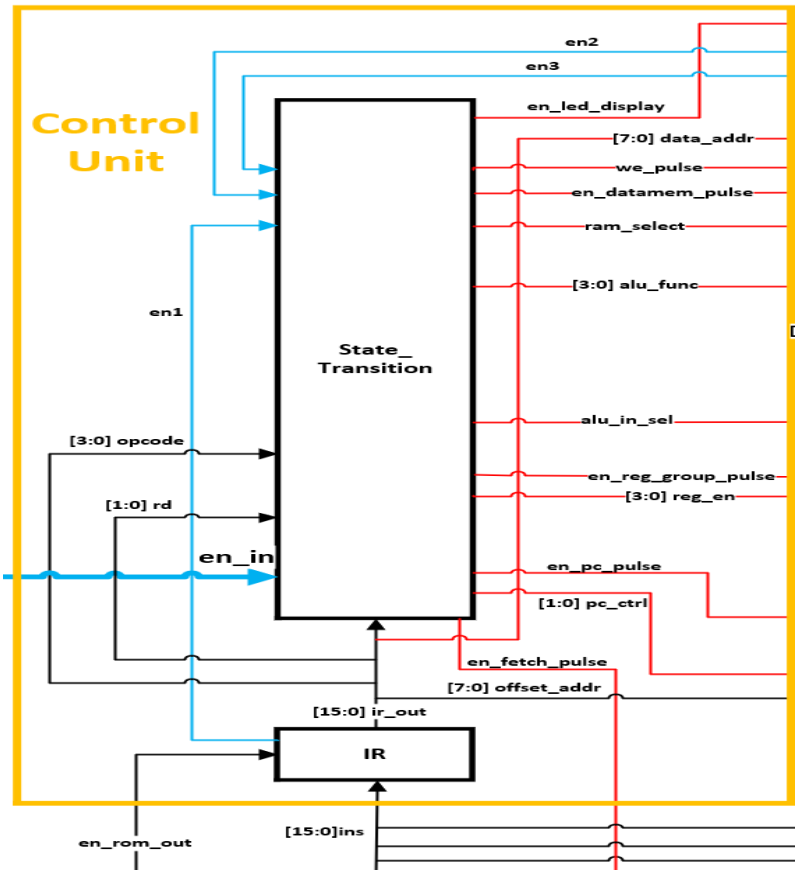

Figure 18 Control Unit Architecture

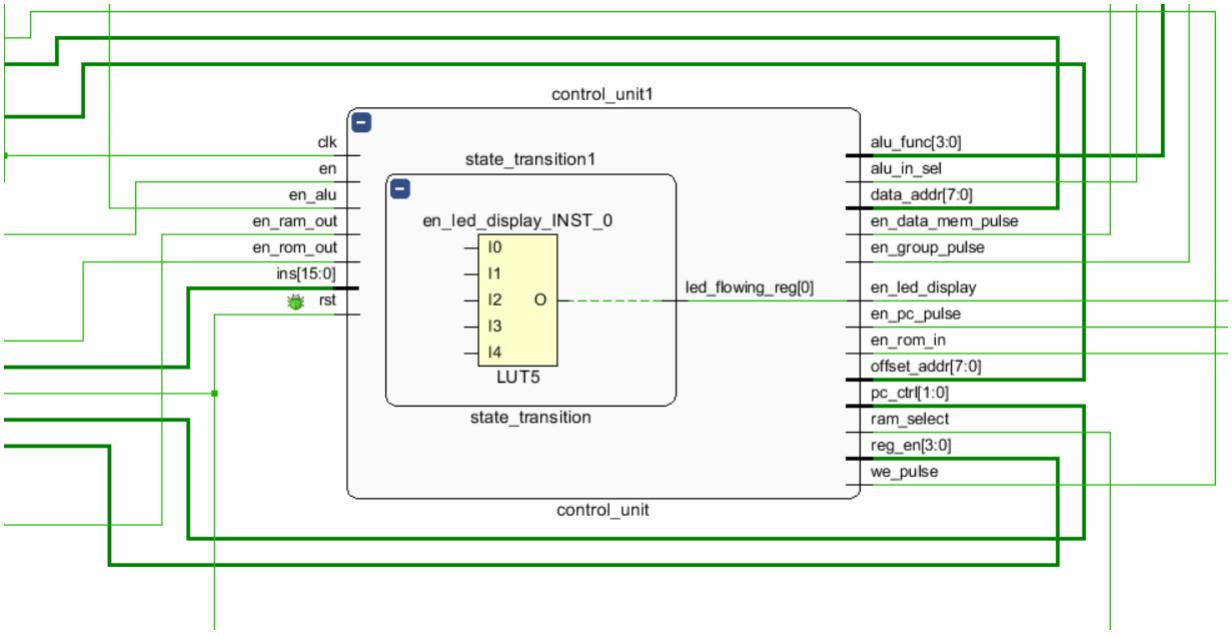

Figure 19 RTL Schematic of Control Unit

3.3.4. Dataflow

The dataflow diagram is shown in Figure 20Figure 19.

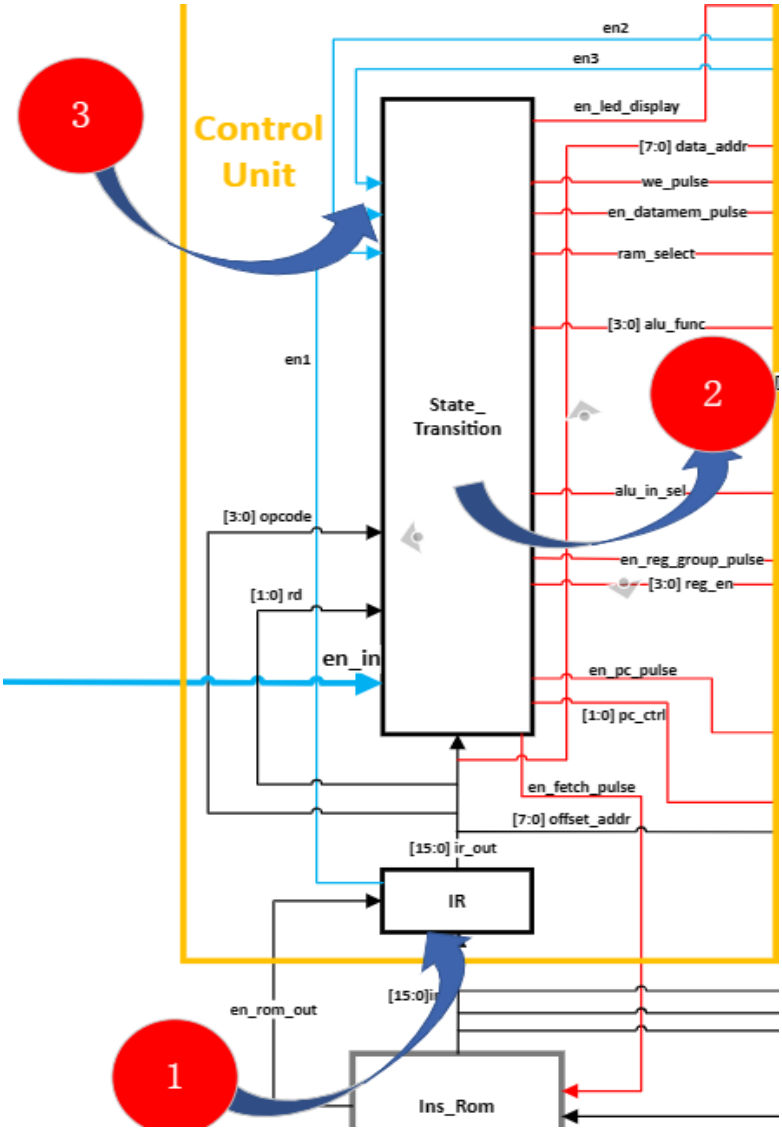

Figure 20 Dataflow diagram of Control Unit

3.3.5. Instruction Register sub-submodule

3.3.5.1. Function

Temporarily store the instruction from Rom.

3.3.5.2. Input/Output

Table 12 Input/Output signals of instruction register

Signal name	I/O	Width (bits)	Function Description
clk	I	1	Clock signal (posedge)
rst	I	1	Asynchronous reset signal (negedge)
ins	I	16	Instruction input according to the instruction set.
en_in	I	1	Enable signal to activate instruction register
en_out	O	1	Delay one clock cycle to return back the value of input enable signal

ir_out	O	16	Delay one clock cycle to send out the input instruction

3.3.5.3. Architecture

Figure 21 Instruction Register Architecture

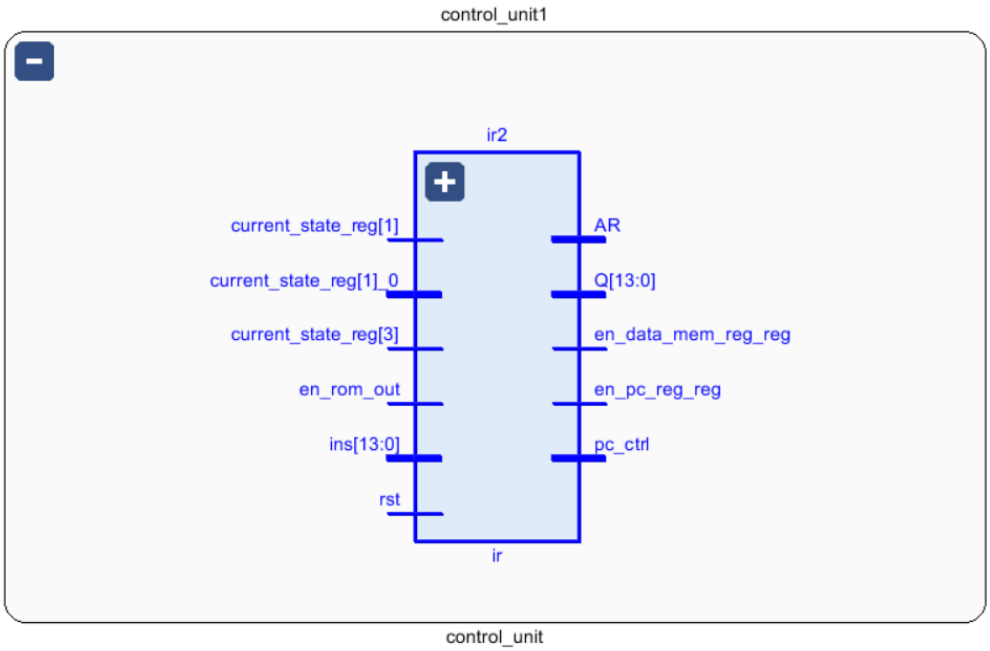

Figure 22 RTL Schematic of Instruction Register

3.3.5.4. Dataflow

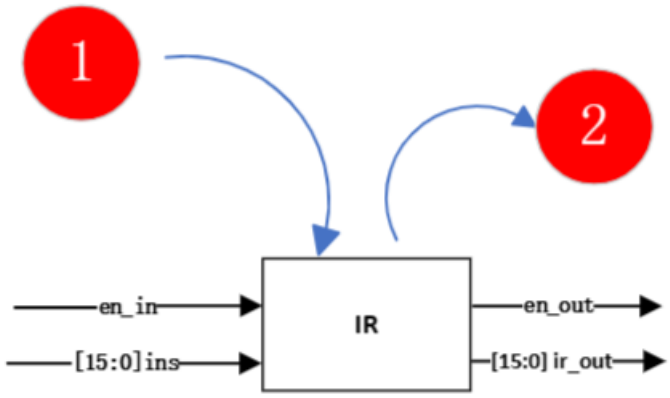

Figure 23 Dataflow diagram of Instruction Register

3.3.5.5. Code

```
always @ (*) begin
    // Do the following when the clock rises or resets the falling edge

    if (!rst) begin
        // If the reset signal is low
        ir_out = 16'b0000000000000000; // Reset the IR output register to all zeros
        en_out = 1'b0;                 // The output enable signal is set to high, indicating invalid data
    end
    else if (en_in) begin
            // If the input enable signal is high
              en_out = 1'b1; // The output enables the signal to remain high, indicating that the data is valid
              ir_out = ins;   // Copy the input instruction to the IR output register
        end
        else
              en_out = 1'b0; // If the input enable signal is low, the output enable signal is set to low,
     // indicating invalid data
        end
    endmodule
```

3.3.6. State Machine sub-submodule

3.3.6.1. Function

The execution of instruction includes Address, Decode, Execute, and Writeback steps; Help the processor understand and control state changes in other modules to respond to asynchronous events.

3.3.6.2. Input/Output

Table 13 Input/Output signals of state machine

Signal name	I/O	Width (bits)	Function Description
clk	I	1	Clock signal (posedge)
rst	I	1	Asynchronous reset signal (negedge)
en_in	I	1	Only when it is 1 does the cpu start to work
en1	I	1	The returned enable signal to indicate instruction has been sent to state machine for entering Decode state
en2	I	1	The returned enable signal to indicate alu has finished operation for entering Write_back state
en3	I	1	The returned enable signal to indicate ram has sent the value that needs loading for entering Write_back state
rd	I	2	The register address of first operand which decides the value of reg_en
opcode	I	4	The operational code of instruction that decides Decode state entering which Execute state
en_fetch_pulse	O	1	Enable signal that starts fetching instruction by enabling Rom
en_group_pulse	O	1	Enable signal that activates Datapath by enabling register group first
en_pc_pulse	O	1	Enable signal that makes Program Counter send out the address of next instruction to Rom
en_data_mem_pulse	O	1	Enable signal that activates Ram to load or store data
we_pulse	O	1	Write enable signal that determines whether Ram loads or stores data
ram_select	O	1	Select signal that determines whether the output of alu or data from Ram to be loaded is sent to register group
alu_in_sel	O	1	Select signal that determines whether the second operand into Alu_mux is from register or offset value
en_led_display	O	1	Enable signal that makes 8-bits LED present the last eight digits of data into register group in Write_back state
pc_ctrl	O	2	Select signal that determines whether the address of next instruction to add 1 directly or be offset address
reg_en	O	4	Address signal into register group to determine which register should receive the value from operation or load
alu_func	O	4	Control signal that controls Alu to operate which instruction

3.3.6.3. Architecture

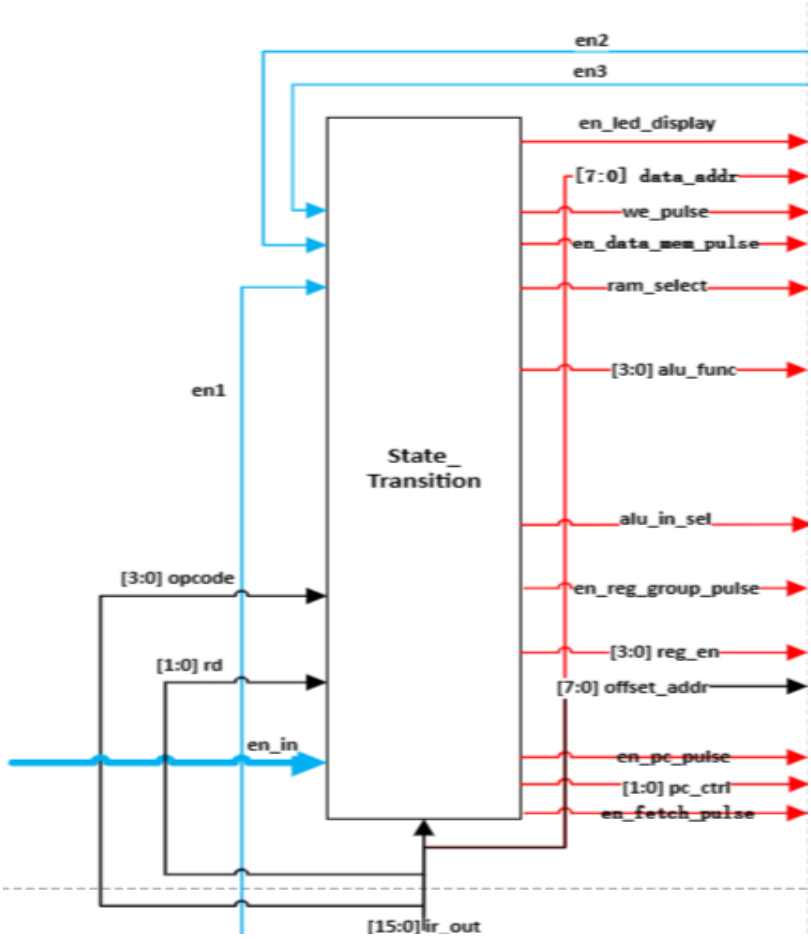

Figure 24 State Machine Architecture

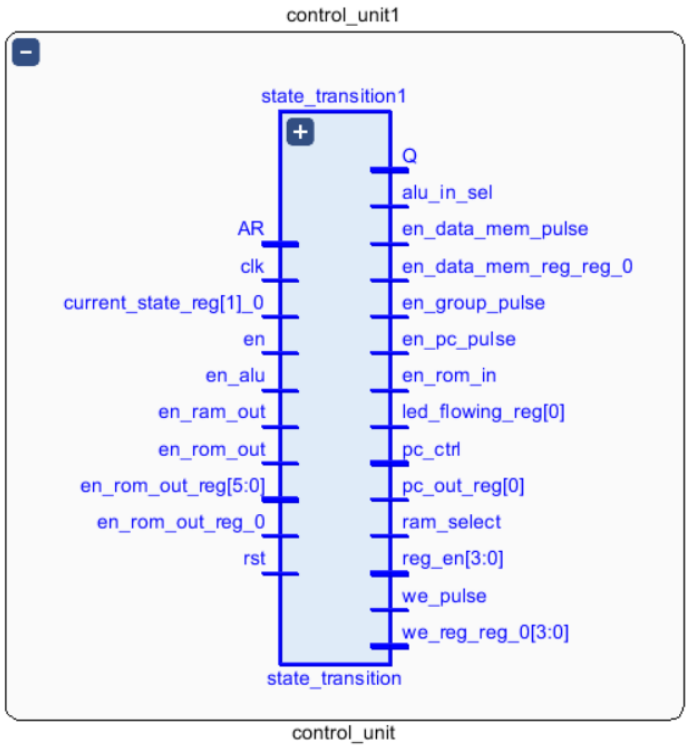

Figure 25 RTL Schematic of Control Unit

3.3.6.4. Dataflow (state-transition)

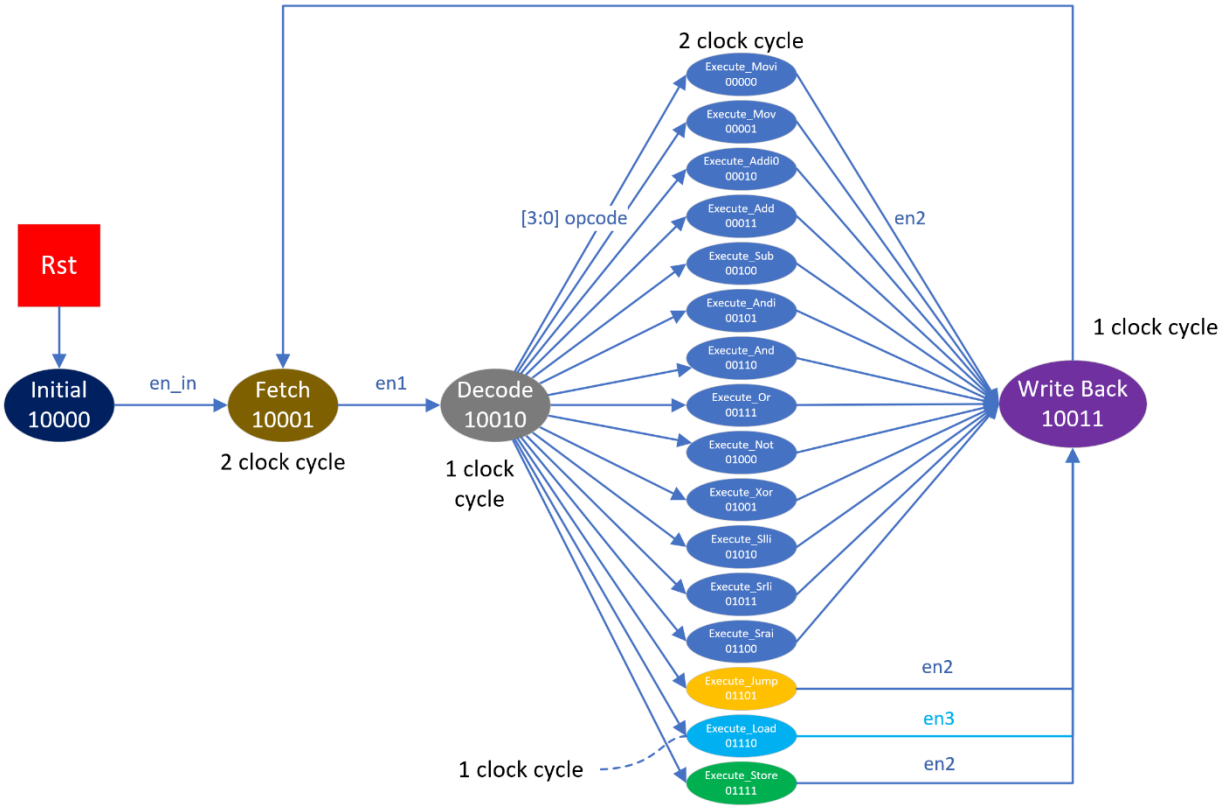

Figure 26 State transition diagram

3.3.6.5 Code

```
always@(*) // Determines the control to start retrieving the enable signal of the decode jump Load
            // and Store state and the subordinate state of each state
    begin
            case(current_state)
              ...
              ...
              ...
                Execute_Jump:
                    begin
                        if(en2) // Indicate the address of next instruction is
                       // prepared and LED lights are ready to be refreshed to 0
                                next_state = Write_back
                        else
                                next_state = current_state;
                    end
                Execute_Load:
                    begin
                        if(en3) // Indicate ram has sent the value that needs
loading
                            next_state = Write_back;
                        else
                            next_state = current_state;
                      end
                Execute_Store:
                        begin
                            if(en2) // Indicate the data that needs storing has
          // been sent into ram and LED lights are ready to be refreshed to 0
                                next_state = Write_back;
                            else
                                next_state = current_state;
                        end

              Write_back: next_state = Fetch; // Start next instruction
              default: next_state = current_state;
              endcase
          end
```

3.3.7. Code Excerpt for Control Unit

```
state_transition state_transition1( // Ports of different signals for state_transition
                .clk(clk) ,
                .rst(rst) ,
                .en_in(en) ,
```

```
                .en1(en_out) , // From ir
                .en2(en_alu) , // From alu
                .en3(en_ram_out) , // From ram
                .rd(ir_out[11:10]) , // address of register for first operand
                .opcode(ir_out[15:12]) ,
                .en_fetch_pulse(en_rom_in), // sent to rom for Fetch
                .en_group_pulse(en_group_pulse), // sent to register_group to start datapath
                .en_pc_pulse(en_pc_pulse) , // enable pc
                .en_data_mem_pulse(en_data_mem_pulse) , // enable ram for load or store
                .we_pulse(we_pulse), // write enable signal to ram
                .ram_select(ram_select),
                .pc_ctrl(pc_ctrl) ,
                .reg_en(reg_en) ,
                .alu_in_sel(alu_in_sel) ,
                .alu_func(alu_func),
                .en_led_display(en_led_display) // refresh LED lights
            );

always @ (en_out,ir_out)
begin // The address sent to pc and ram is directly from offset number
            offset_addr = ir_out[7:0]; // sent to pc functioning in jump instruction
            data_addr = ir_out[7:0]; // sent to ram functioning in load/store instruction
    end
```

3.4. Ins_ROM submodule

3.4.1 Function

The Ins_ROM module is used for outputting instructions, internally instantiating an IP core [1].

For the IP core: The memory type is the single port ROM type. Port A width is 16bit, and Port A depth is 256bit. The operating mode is 'write first', and it employs the enable port type.

This module is used to receive the address signals from the PC. When the enable signal en_rom_in is at a high level, it outputs the instructions stored internally in Ins_ROM corresponding to the PC address, and sends the en_rom_out signal back to the state_transition part of the control_unit as feedback to the state machine. This module also takes the reset signal and clock signal as inputs.

3.4.2 Input/Output

The interface of Ins_Rom module is defined in Table 14.

Table 14 Input/Output signals of Ins_Rom module

Signal name	I/O	Width (bits)	Function Description
clk	I	1	Clock signal (posedge)
rst	I	1	Asynchronous reset signal (negedge)
en_rom_in	I	1	Enable signal that makes Rom give out instruction

addr	I	8	Address signal of instruction from Program Counter
en_rom_out	O	1	Delay one clock cycle to return back the value of input enable signal
ins	O	16	Instruction that is stored in Rom

3.4.3 Architecture

The hardware structure is shown in Figure 27.

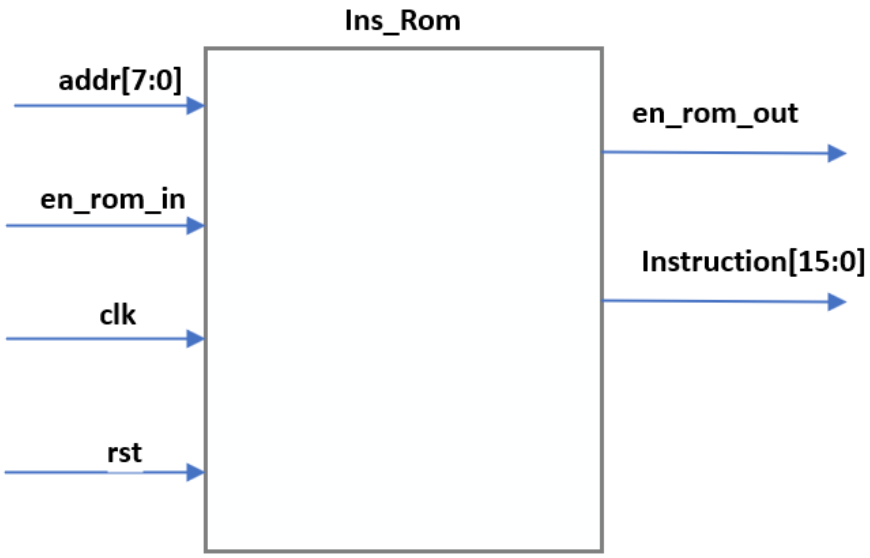

Figure 27 Ins_ROM architecture

The RTL-level is shown in Figure 28.

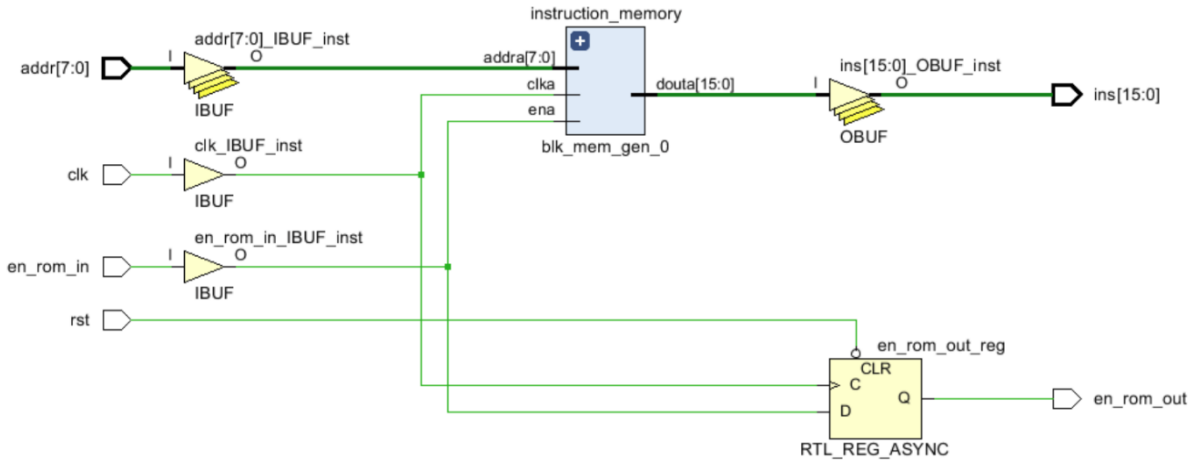

Figure 28 Circuit diagram of Ins_ROM obtained by synthesising

3.4.4 Dataflow

The control flow is shown in Figure 29.
Once the control signal and address bits arrive at the same time, the stored instructions are output in the first clock cycle.

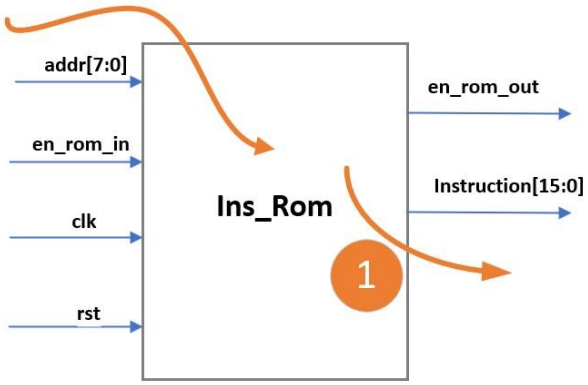

Figure 29 Datapath of Ins_Rom

3.4.5. Code Excerpt

The code for the key parts of this module is shown below, which includes the instantiation of an IP core named "blk_mem_gen_0".

```
// module instruction_memory_module: Instruction Memory Module
// This module represents an instruction memory with read control signals.

module instruction_memory_module (
    input rst,              // Reset signal
    input clk,              // Clock signal
    input en_rom_in,        // Input enable signal for ROM
    input [7:0] addr,       // Address input for ROM
    output [15:0] ins,      // Output instruction
    output reg en_rom_out // Output enable signal for ROM
);

    always @(posedge clk or negedge rst)
        if (!rst)
            en_rom_out <= 0;
        else if (en_rom_in == 1'b1)
            en_rom_out <= 1;
        else
            en_rom_out <= 0;

    // Instantiating the block memory generator for instruction_memory
    // This block memory generator is named "blk_mem_gen_0"
    blk_mem_gen_0 instruction_memory (
        .clka(clk),
        .ena(en_rom_in),
```

```
        .addra(addr),
        .douta(ins)
    );

endmodule
```

3.5. Data_RAM submodule

3.5.1 Function

The Data_RAM module is used for storing or outputting data from the CPU's operations, internally instantiating an IP core [1].

For the IP core: The memory type is the single port RAM type. Port A write width is 16bit, read width is 16bit, Port A write depth is 256bit, and read depth is 256bit. As there is only one clock frequency, the operating mode is 'not change', thus reducing power consumption. It also uses the enable port type and accepts the enable signal.

This module performs write operations when the CE/we signal is high, storing the input data signal (data_in) at the address received by data_addr. When the CE signal is high and the we signal is low, it performs read operations, sending the data stored at data_addr to the reg_group_mux_out module of data_path, while simultaneously outputting the en_ram_out signal at a high level, returning to the state_transition part of the control_unit as feedback to the state machine. This module also takes the reset signal and clock signal as inputs.

Table 15 Relationship between CE/WE and load/store for RAM

Signal name	Data_direction	Function Description
CE=1,WE=1	input	store_enable mode
CE=1,WE=0	output	load_enable mode

3.5.2 Input/Output

The interface of Ins_Rom module is defined in Table 16.

Table 16 Input/Output signals of Ins_RAM module

Signal name	I/O	Width (bits)	Function Description
clk	I	1	Clock signal (posedge)
rst	I	1	Asynchronous reset signal (negedge)
ce	I	1	Enable signal that activates Ram to load or store data
we	I	1	Write enable signal that determines whether Ram loads or stores data
data_addr	I	8	Address signal of data in Ram
data_in	I	16	Data that needs to be stored in Ram
en_ram_out	O	1	Delay one clock cycle to return back the value of input

			enable signal (ce)
data_out	O	16	Data in Ram that needs to be loaded

3.5.3 Architecture

The hardware structure is shown in Figure 30.

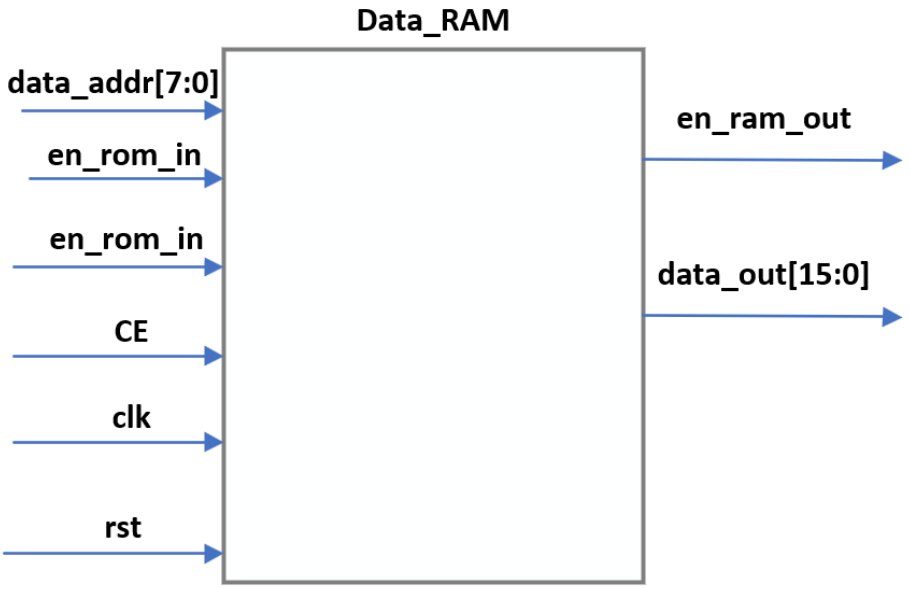

Figure 30 Data_RAM architecture

The RTL-level is shown in Figure 31.

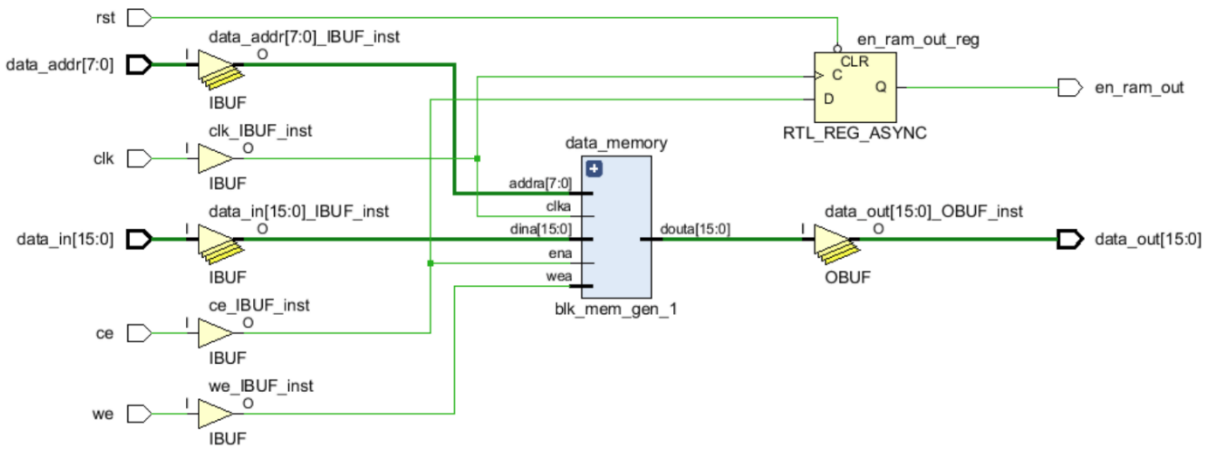

Figure 31 Circuit diagram of Ins_RAM obtained by synthesising

3.5.4 Dataflow

The control flow is shown in Figure X.

In the Stone state (WE=1/CE=1), the data_in data is stored in the address carried by data_addr for one clock cycle. In the Write state (WE=0/CE=1), the address data indicated by addr is read to the data_out signal in one

clock cycle.

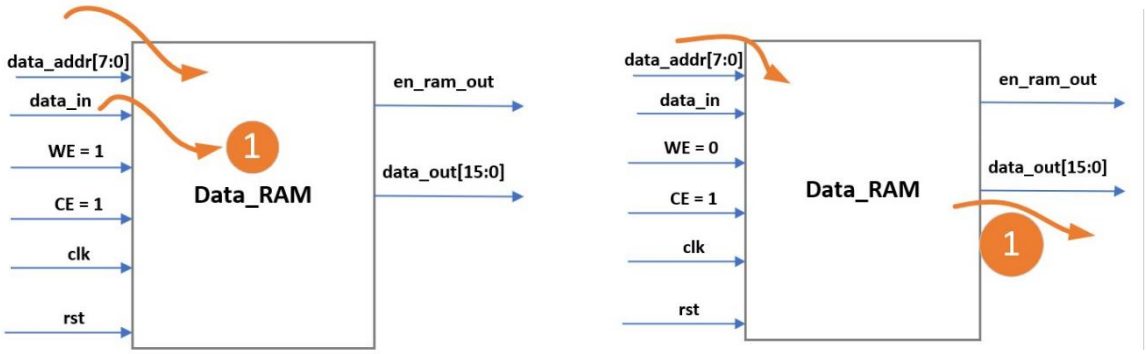

Figure 32 Datapath of Ins_RAM Stone (left) / Load(right)

3.5.5 Code

```
// module data_memory_module: Data Memory Module
// This module represents a data memory with read and write control signals.

module data_memory_module (
    input rst,              // Reset signal
    input clk,              // Clock signal
    input ce,               // Chip Enable signal for RAM
    input we,               // Write Enable signal for RAM
    input [7:0] data_addr,// Address input for RAM
    input [15:0] data_in, // Input data for RAM
    output [15:0] data_out,// Output data from RAM
    output reg en_ram_out // Output enable signal for RAM
);

    always @(posedge clk or negedge rst)
        if (!rst)
            en_ram_out <= 0;
        else if (ce == 1'b1)
            en_ram_out <= 1;
        else
            en_ram_out <= 0;

    // Instantiating the block memory generator for data_memory
    // This block memory generator is named "blk_mem_gen_1"
    blk_mem_gen_1 data_memory (
        .clka(clk),
        .ena(ce),
        .wea(we),
        .addra(data_addr),
```

```
        .dina(data_in),
        .douta(data_out)
    );

endmodule
```

3.6. Clock, Reset & Initial

3.6.1 Clock

The original clock signal comes from the system clock, and in the constraint file, the clock signal for the FPGA board is set to 20ns (50MHz in frequency). The clock's duty cycle is 50%, and the output type is a square wave. To make the results observable to the naked eye, we use a frequency division method to reduce the clock frequency. The clock signal after reduction is 0.2s (5Hz) , which can be observed with the naked eye. The key code is shown below:

```
// 32-bit counter for frequency division
reg [31:0] counter;

// New slow clock
reg clk_slow;

// Division factor, adjust according to your requirements
parameter DIV_FACTOR = 10000000;

// On every rising edge of clk or falling edge of rst
always @(posedge clk or negedge rst)
begin
    if (!rst)
    begin
        // Reset slow clock and counter
        clk_slow <= 0;
        counter <= 0;
    end
    else if (counter == DIV_FACTOR - 1)
    begin
        // Toggle the state of clk_slow
        clk_slow <= ~clk_slow;
        // Reset counter
        counter <= 0;
    end
    else
    begin
        // Increment counter
        counter <= counter + 1;
```

```
    end
end
```

3.6.2 Reset

A reset signal rst is set up to control the CPU reset. During the design process, the falling edge of the reset signal or the rising edge of the clock signal will trigger the assignment of sequential circuits.

In terms of reset type, asynchronous reset is adopted. When a low level reset signal is detected, all CPU tasks will be halted, and all registers and enable signals will be initialized.

The code below has shown the structure of module with reset signal.

```
always @(posedge clk or negedge rst)
if (!rst)
    begin
......
    end
else if ()
    begin
........
    end
else
    begin
........
    end
endmodule
```

3.6.3 Initial

As for the initial state of the CPU, when the en_in signal is detected to be high, the CPU starts executing tasks from the first instruction. During the execution of tasks, a low en_in signal will not affect the CPU's task execution. This is because a high level signal will cause the current_state in the state_transition module to change from initial to Fetch stage, and it will not return to the initial stage subsequently, thus there is no need for the drive of the en_in signal. If you need to stop CPU operations, let en_in sign to be 0.

The code below has shown the control signal sent by control unit during the "Initial" state. All contorl signals are set to be 0.

```
case (next_state)
    Initial:
        begin
            // Disable various modules
            en_fetch = 1'b0;
            en_group = 1'b0;
            en_pc = 1'b0;
            en_data_mem = 1'b0;
            we = 1'b0;
            ram_select = 1'b0;

            // Control signals initialization
```

```
            pc_ctrl = 2'b00;
            reg_en = 4'b0000;
            alu_in_sel = 1'b0;
            alu_func = 4'b0000;

            // Disable LED display module
            en_led_display = 1'b0;
        end
endcase
```

3.7. Led_display module

3.7.1 Function

The Led_display_module can be used to display the results of operations inside the CPU. When the en_display signal is given, the last 8 bits of the CPU's output reg_mux_out signal will be given to the led_flowing register when the next clock rise edge arrives, and the eight bits correspond to the eight semaphores on the FPGA board (and the digital display tubes on them). This module can be used to display the results of operations inside the CPU. When the en_display signal is given, the last 8 bits of the CPU's output reg_mux_out signal will be given to the led_flowing register when the next clock rise edge arrives, and the eight bits correspond to the eight semaphores on the FPGA board (and the digital display tubes on them).

3.7.2 Input/Output

The interface of Led_display module is defined in Table 17.

Table 17 Input/Output signals of Led_display module

Signal name	I/O	Width (bits)	Function Description
en_led_display	I	1	Enable signal of led_display module , which is released at the "Write_back" state
reg_group_mux_out	I	16	The result of the operation from the CPU output
Led_flowing	O	8	Control the bit stream of LED lights every moment

3.7.3 Architecture

The hardware structure is shown in Figure 33.

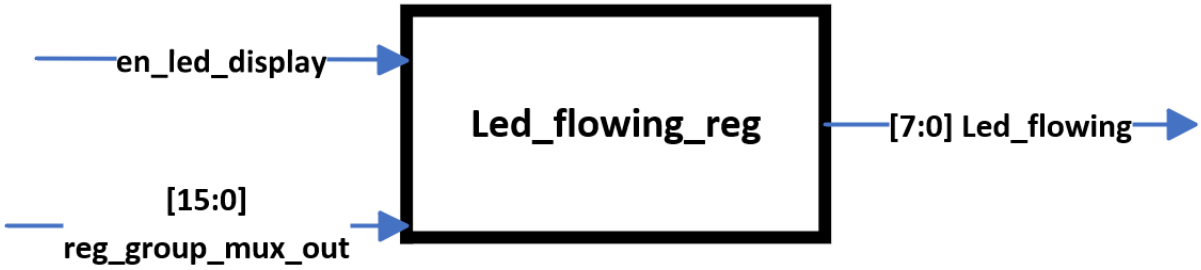

Figure 33 Architecture of Led_flowing_reg

3.7.4. Dataflow

The control flow is shown in Figure 34.

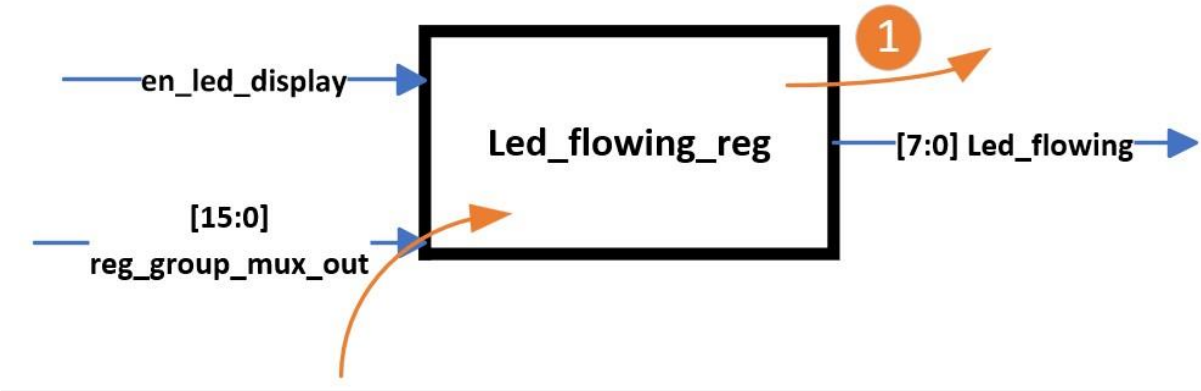

Figure 34 Datapath of Led_flowing_reg

3.7.5. Code

```
// On every rising edge of clk or falling edge of rst
always @(posedge clk or negedge rst)
begin
    // Reset condition: when rst is active
    if (!rst)
    begin
        led_flowing <= 0; // Reset led_flowing
    end
    // Enable LED display when en_led_display is true
    else if (en_led_display)
    begin
        led_flowing <= reg_group_mux_out[7:0]; // Update led_flowing with data
    end
end
```

4. Verification & Test

4.1. Test Plan & TestBench

4.1.1 Test Plan for Datapath

Code Excerpt of tb_datapath:

```
module tb_datapath;

reg [3:0] alu_func, reg_en;                 // ALU function and register enable signals
reg alu_in_sel, en_in, clk, en_pc_pulse, rst; // ALU input selection, enable input, clock, PC pulse, and
reset signals
reg [7:0] offset, offset_addr;          // Offset and offset address for memory access
reg [1:0] pc_ctrl, rd, rs;                // Program counter control, destination register, and source register
reg [15:0] reg_group_mux_out;             // Data input of register group
```

```
wire [15:0] alu_out;                    // ALU output
wire [15:0] pc_out;                     // Program counter output
wire en_out;                            // Output enable signal
wire [15:0] rd_data;                    // Data read from register file

data_path data_path1(
    // Instantiate datapath module...
);

// Test scenarios for the datapath
initial begin
// Instruction 1: R2 = 1
            en_pc_pulse = 1'b0;
            en_in = 1'b0; pc_ctrl = 2'b00; alu_func = 4'b0000; reg_en = 4'b0000;alu_in_sel = 1'b0;
            offset = 8'b00000000; offset_addr = 8'b00000000;rd = 2'b00; rs=2'b00;
            reg_group_mux_out =   16'b0000000000000000;
          #(Tclk*1.25)   pc_ctrl = 2'b01; en_pc_pulse = 1'b1;
          #(Tclk*0.5)    rd = 2'b10; rs = 2'b00; en_pc_pulse = 1'b0; //45
          #(Tclk) pc_ctrl = 2'b00; //55
          #(Tclk) alu_func = 4'b1110; //75
          #(Tclk) reg_en = 4'b0100; reg_group_mux_out = 16'b0000000000000001; //95

// Additional instructions...

// Instruction 19: R0 = Mem[15]
            #(Tclk) pc_ctrl = 2'b01; reg_en = 4'b0000; en_pc_pulse = 1'b1;alu_func = 4'b0000; //2235
            #(Tclk) en_pc_pulse = 1'b0;offset = 8'b00001001; rd = 2'b00; rs = 2'b00; //2255
            #(Tclk) pc_ctrl = 2'b00; offset_addr = 8'b00001001; //2275
            #(Tclk) en_in = 1'b1; alu_func = 4'b1110; alu_in_sel = 1'b0;//2295
            #(Tclk) en_in = 1'b0; //2315
            #(Tclk) reg_en = 4'b0001;reg_group_mux_out =   16'b0000000000000010; alu_in_sel = 1'b0;
//2335
end

endmodule
```

The code snippet excerpted in tb_datapath shows information about all relevant signals except clock and reset and the test flow for the first and last instructions.

This testbench aims to verify the functionality of the datapath module by applying a sequence of instructions. It includes clock generation, reset generation, and specific scenarios for instruction execution. Each instruction is specified with appropriate control signals, and the simulation runs for a defined number of clock cycles before stopping. The testbench checks if the datapath processes the instructions correctly and produces the expected results.

4.1.2 Test Plan for Control Unit

Code Excerpt of tb_control_unit:

```
module tb_control_unit();

reg clk, rst, en, en_alu, en_rom_out, en_ram_out = 0;
reg [15:0] ins; // Instruction loaded from ROM

wire en_rom_in, en_group_pulse, en_pc_pulse, en_data_mem_pulse, we_pulse, ram_select, alu_in_sel,
en_led_display;
wire [1:0] pc_ctrl; // Control signal for the Program Counter
wire [3:0] reg_en, alu_func; // Control signals for Register File and ALU
wire [7:0] offset_addr, data_addr; // Addresses for offset and data in memory
integer i = 0;

control_unit control_unit1(
    // Instantiate the control_unit module...
);

// Enable signal generation
initial
begin
    en = 0;
    #(1.5*Tclk) en = 1;
end

// ALU control signal generation
initial
begin
    en_alu = 0;
    #(0.5*Tclk);
    while (1)
    begin
        #(4*Tclk) en_alu = 1;
        #(Tclk) en_alu = 0;
    end
end

// ROM instruction loading
initial
begin
    en_rom_out = 0;
    ins = 16'b0000000000000000;
```

```
    #(2.5*Tclk) en_rom_out = 1;
    ins = 16'b0000100000000001;
    while (1)
    begin
        // Instruction sequence loading...
    end
end

endmodule
```

The code snippet excerpted in tb_control_unit shows information about all relevant signals except clock and reset and the test flow for ROM instruction loading.

This testbench aims to validate its functionality by applying a sequence of instructions. It includes clock generation, reset generation, and the generation of various control signals like en_alu, en_rom_out, etc. The ROM instructions are loaded into the system to simulate different scenarios. The testbench checks if the control unit processes the instructions correctly and produces the expected control signals.

4.1.3 Test Plan for Overall CPU

Firstly, the CPU is tested using two methods: LED flowing and accumulation from 1 to 16 by giving corresponding ROM instructions. Secondly, the CPU is fully tested by giving a specific sequence of instructions containing all the instructions with specific data memory. The following two tables show the details of instruction memory and data memory we set. During the process we used ILA to grab some of the signals for testing [1].

1. Instruction memory

Table 18 Instructions in ROM

num	opcode	rd	rs	imm	description	expected result
1	1110	10	00 (unused)	0000 0000	Load R2=Mem[0]	0000 0001
2	0000	11	00 (unused)	0000 0010	Movi R3=0000 0010	0000 0010
3	0001	00	10	0000 0000 (unused)	Mov R0=R2	0000 0001
4	0001	01	11	0000 0000 (unused)	Mov R1=R3	0000 0010
5	0010	01	00 (unused)	0000 0100	Addi R1=R1+0000 0100	0000 0110
6	0011	00	01	0000 0000 (unused)	Add R0=R0 + R1	0000 0111
7	0100	00	01	0000 0000 (unused)	Sub R0=R0 - R2	0000 0110

8	1101	01 (unused)	00 (unused)	0000 1001	Jump to the 10th instruction	0000 0000
9	0011	00	10	0000 0000 (unused)	Add R0=R0+R2 (Not execute)	Not execute
10	1000	00	00 (unused)	0000 0000 (unused)	Not R0= ~ R0	1111 1001
11	0101	00	00 (unused)	0000 1001	Andi R0=R0 & (0000 1001)	0000 1001
12	0110	00	01	0000 0000 (unused)	And R0=R0 & R2	0000 0001
13	0111	00	11	0000 0000 (unused)	Or R0=R0 | R3	0000 0011
14	1001	00	10	0000 0000 (unused)	Xor R0 ^ R2	0000 0010
15	1010	00	00 (unused)	0000 0011	Slli R0=R0<<3	0001 0000
16	1011	00	00 (unused)	0000 0010	Srli R0=R0>>2	0000 0100
17	1100	00	00 (unused)	0000 0001	Srai R0=R0>>>1	0000 0010
18	1111	00	00 (unused)	0000 0111	Store Mem[15]=R0	0000 0000
19	1110	00	00 (unused)	0000 0111	Load R0=Mem[15]	0000 0010
20	1101	00 (unused)	00 (unused)	0000 0000	Jump to the 1st instruction	0000 0000

2. Data memory

Table 19 Data in RAM

address	data
0x00	0000 0000 0000 0001
0x01	0000 0000 0000 0010
0x02	0000 0000 0000 0111
0x03	1111 1111 1111 1111
0x04	1111 1111 1111 1111
0x05	1111 1111 1111 1111
0x06	1111 1111 1111 1111
0x07	1111 1111 1111 1111
0x08	1111 1111 1111 1111
0x09	1111 1111 1111 1111
0x0a	1111 1111 1111 1111

0x0b	1111 1111 1111 1111
0x0c	1111 1111 1111 1111
0x0d	1111 1111 1111 1111
0x0e	1111 1111 1111 1111
0x0f	1111 1111 1111 1111

The testing approach involves executing a given sequence of CPU instructions in a Verilog simulation environment to validate the correctness of our CPU design. The Data Memory has also been designed to work with the above tests. The detailed steps are shown in the two tables above.

3. Code Excerpt of tb_cpu:

```
module tb_cpu();
reg             clk,rst,en_in;
wire [7:0] led_flowing; //Output signal on FPGA

// Instantiate the cpu module
cpu test_cpu(
    .clk (clk),
    .rst (rst),
    .en_in (en_in),
    .led_flowing(led_flowing)
);

initial begin
        //define clk
        clk=0;
        forever #(Tclk/2) clk=~clk; // Toggle the clock every half period
        end

initial begin
        //define rst
        rst=0;
        #(Tclk*1)    rst=1; // Assert reset after a certain time
        end

initial begin
        //define en_in
                    en_in = 1'b0;
        #(Tclk*2.5)       en_in = 1'b1; // Set en_in signal after a certain time
end

endmodule
```

The testbench (tb_cpu) is designed to verify the functionality of the cpu module. It sets up the clock signal (clk), reset signal (rst), and an input enable signal (en_in). The testbench generates a clock signal with a period specified by Tclk and initializes the reset and input enable signals. The cpu module is instantiated, and the clock, reset, and enable signals are connected to it. The led_flowing signal is used to observe the output from the cpu.

Overall, the testbench creates the necessary signals, generates a clock, initializes the reset and enable signals, and tests the cpu module's behavior. The led_flowing signal captures the output of the cpu, allowing observation and verification of its functionality.

4.2. TestCase & Result

4.2.1 Simulation Results for Datapath:

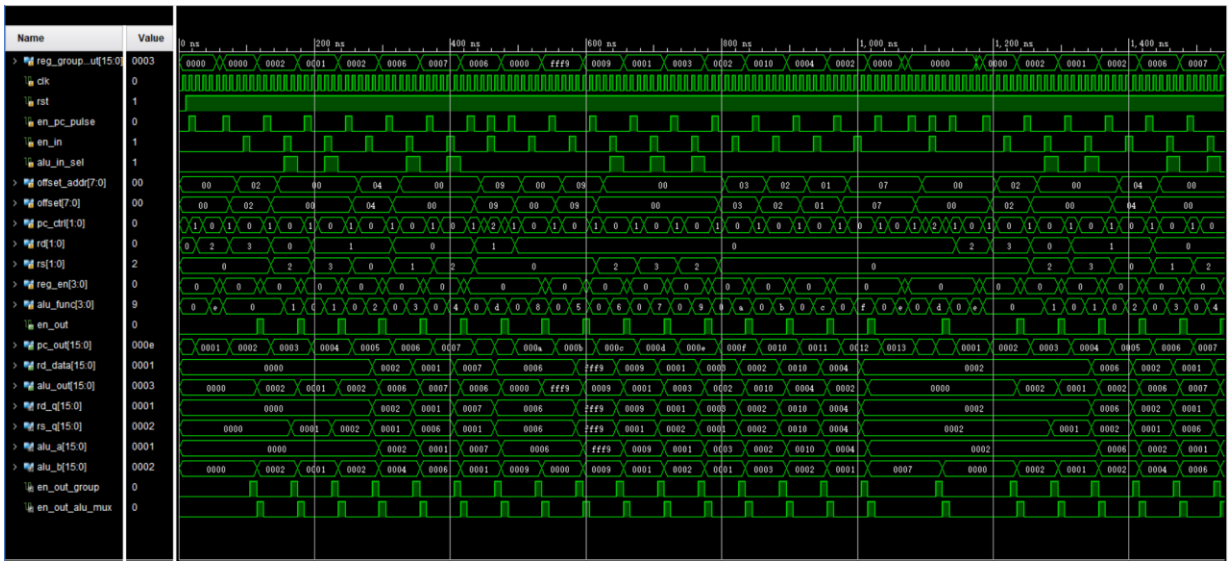

Figure 35 Simulation Results for Datapath

Explanation:
To verify the function, the datapath is given different instructions. The testbench verifies the correct operation of the CPU's data processing components, such as the ALU and register file. The waveform shows proper synchronization of clock signals, appropriate changes in register values, and expected ALU operations. Key points include checking that the ALU produces correct results for various operations and that register values are updated appropriately.

4.2.2 Simulation Results for Control Unit:

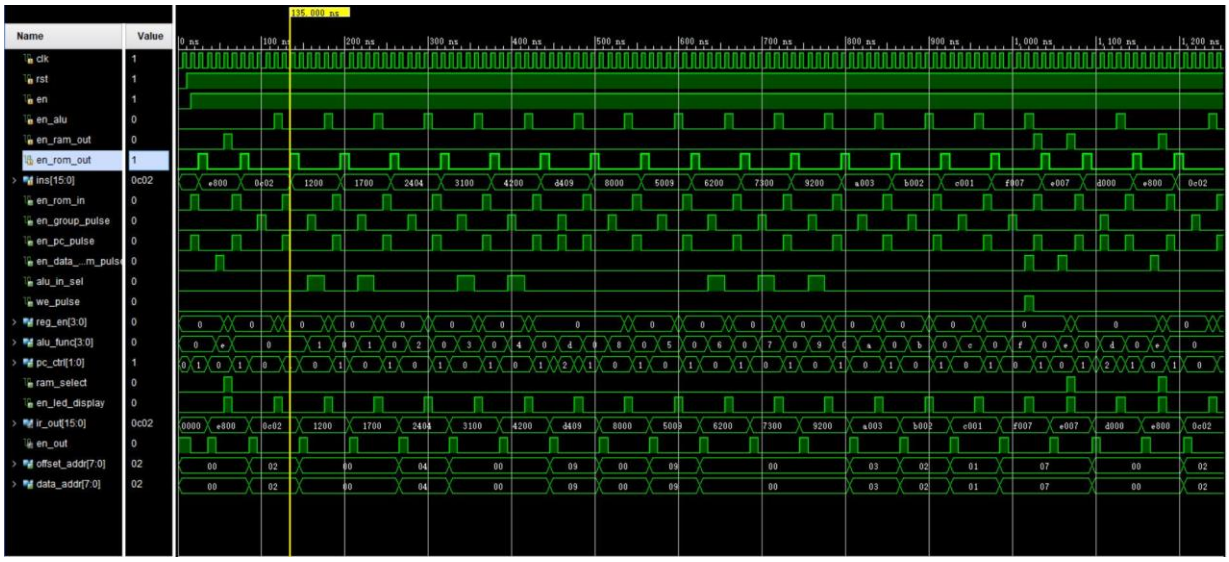

Figure 36 Simulation Results for Control Unit

Explanation:

To verify the function, the control unit is given different instructions. The control unit testbench focuses on validating the correctness of control signals generated by the CPU. The waveform exhibits the generation of control signals for different CPU components, such as the ALU, registers, and memory. Key points include checking that control signals align with the expected behavior for each instruction and that the program counter is incremented or modified correctly.

4.2.3 Simulation Results for CPU:

Total testbench:

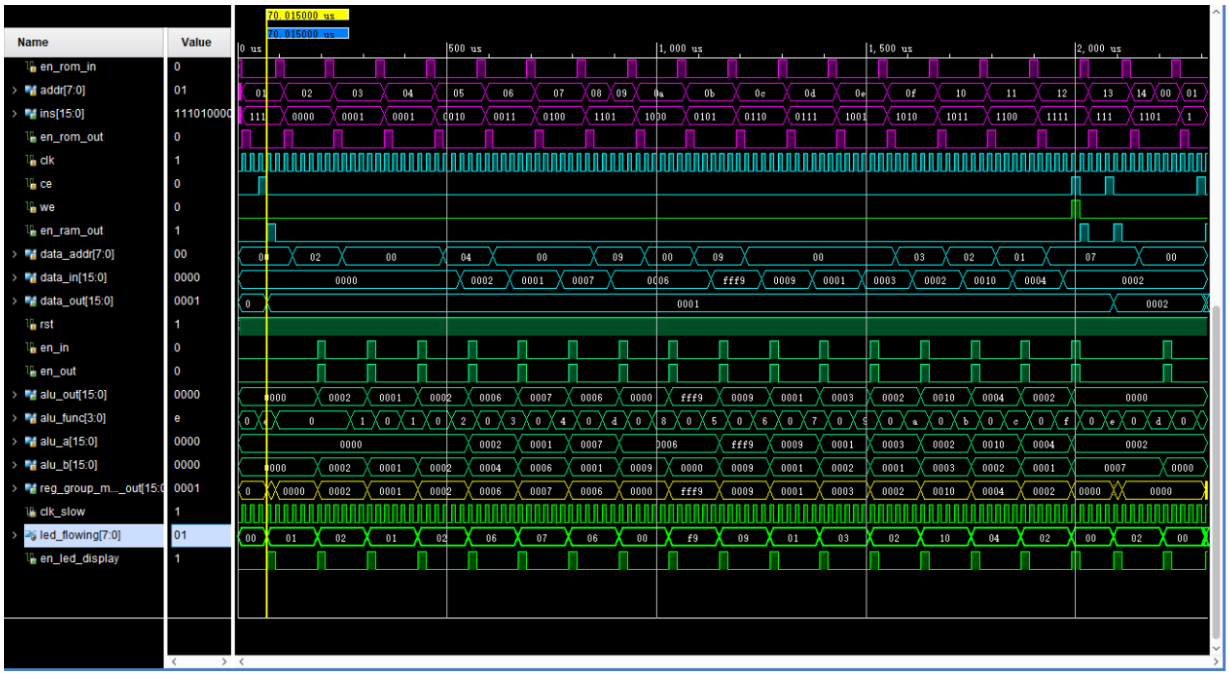

Figure 37 Simulation Results for CPU

Explanation:

The overall CPU testbench combines the datapath and control unit, ensuring their integration functions correctly. The waveform should illustrate the entire CPU operation, including instruction execution, data movement, and control flow. Key points include observing the expected behavior of instructions, proper handling of jump, load and store, and the correct flow of data between the datapath and control unit.

4.2.4. LED flowing:

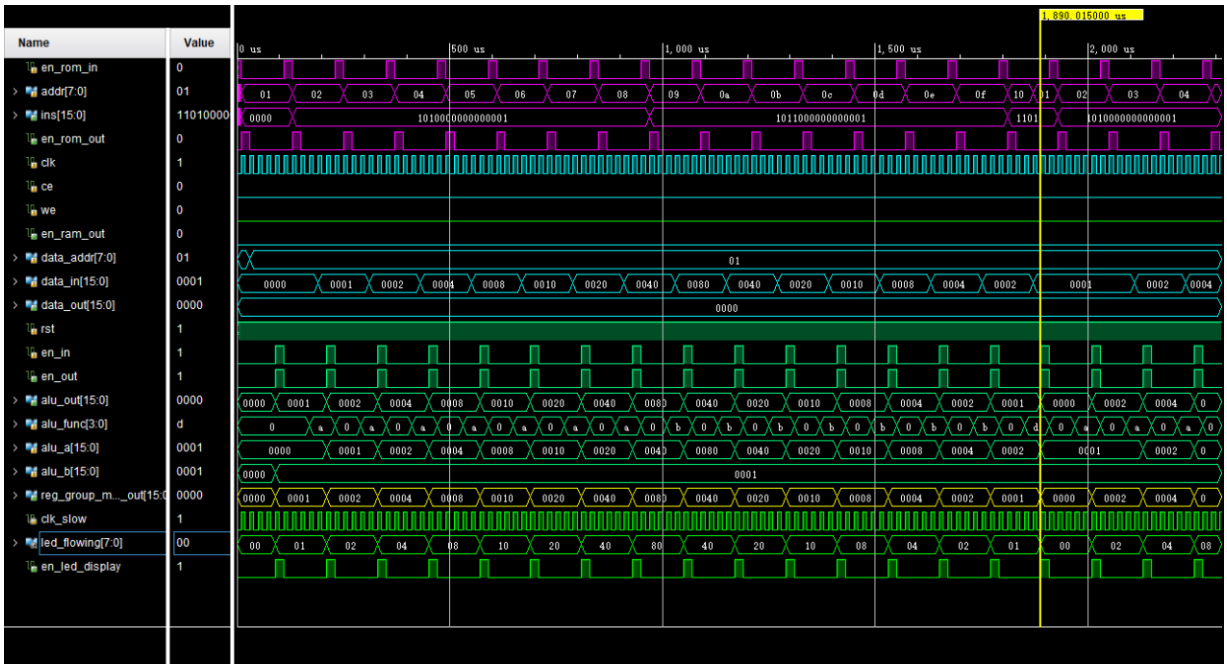

Figure 38 Simulation Result for LED flowing

Key points include the data conversion of led_flowing. The figure shows that the shift instructions work successfully, where the data of led_flowing changes: 8'b00000000, 8'b00000001, 8'b00000010, ... 8'b00000010, 8'b00000001, 8'b00000000.

4.2.5. Add from 1 to 16:

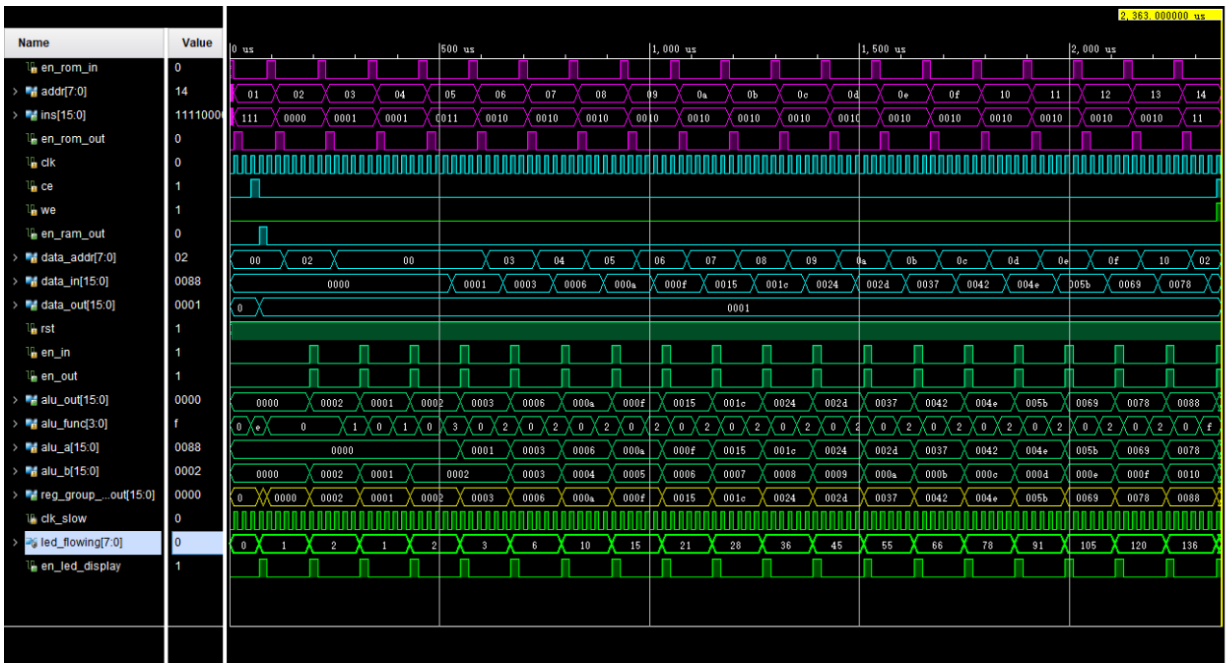

Figure 39 Simulation Result for adding from 1 to 16

Key points include the data conversion of led_flowing. The figure shows that the add, load and relevant instructions work successfully, where the data of led_flowing changes from 1 to 136 in decimal.

5. Conclusion & Future work

5.1. Conclusion

In this project, we successfully designed a 16 bits multi-cycle CPU, which supports arithmetic operations, logic operations, jump, Load/Store, a total of 16 instructions. It uses Harvard architecture and external Rom and Ram connection, by an LED display module display results. We successfully simulated, synthesized, and verified all instructions on the FPGA board using the total testbench. The LED display results were in line with our expectations, proving the effectiveness of our CPU design.

5.2. Future work

Here are some designs that could be improved:

1. It is possible to consider modifying the PC so that the CPU can jump to specific code segments under button control.
2. Introducing conditional branching functionality is feasible.
3. Adding D-trigger to separate different states, further evolving it into a pipelined CPU, can be pursued.

6. References

[1] B. Yan, Introduction for challenging course. [Lecture Slides]. (2023/24, Winter). UoG12021 Digital Logic Design and Application. Chengdu, PRC: Glasgow College, University of Electronic Science and Technology of China.

[2] CSDN, “Computer Composition and Design Lab 3: Multi-Cycle CPU Design.” Blog.csdn.net. (Mar. 01, 2019). Accessed: Dec. 03, 2023. [Online]. Available: https://blog.csdn.net/w_weilan/article/details/84982350

[3] RISC-V, “Specifications.” RISC-V International. (Dec. 03, 2021). Accessed: Dec. 03, 2023. [Online]. Available: https://riscv.org/technical/specifications/

[4] B. Yan, Digital Logic Design and Application Lab Course 4. [Lecture Slides]. (2023/23, Winter). UoG12021 Digital Logic Design and Application. Chengdu, PRC: Glasgow College, University of Electronic Science and Technology of China.

[5] B. Yan, Digital Logic Design and Application Lab Course 2. [Lecture Slides]. (2023/23, Winter). UoG12021 Digital Logic Design and Application. Chengdu, PRC: Glasgow College, University of Electronic Science and Technology of China.

[6] B. Yan, Digital Logic Design and Application Lab Course 3. [Lecture Slides]. (2023/23, Winter). UoG12021 Digital Logic Design and Application. Chengdu, PRC: Glasgow College, University of Electronic Science and Technology of China.

[7] CSDN, “Hands-on guide to creating a RAM IP core in Vivado and verifying it with ILA tools.” Blog.csdn.net. (Oct. 28, 2022). Accessed: Dec. 03, 2023. [Online]. Available: https://blog.csdn.net/y_u_yu_yu_/article/details/127144603

[8] CSDN, “FPGA - Detailed usage of ILA (Logic Analyzer) under Vivado.” Blog.csdn.net. (Oct. 28, 2022). Accessed: Dec. 03, 2023. [Online]. Available: https://blog.csdn.net/unique_ZRF/article/details/127715565