Department of Electrical & Computer Engineering

AIR UNIVERSITY

PROJECT REPORT

Digital Systems Design

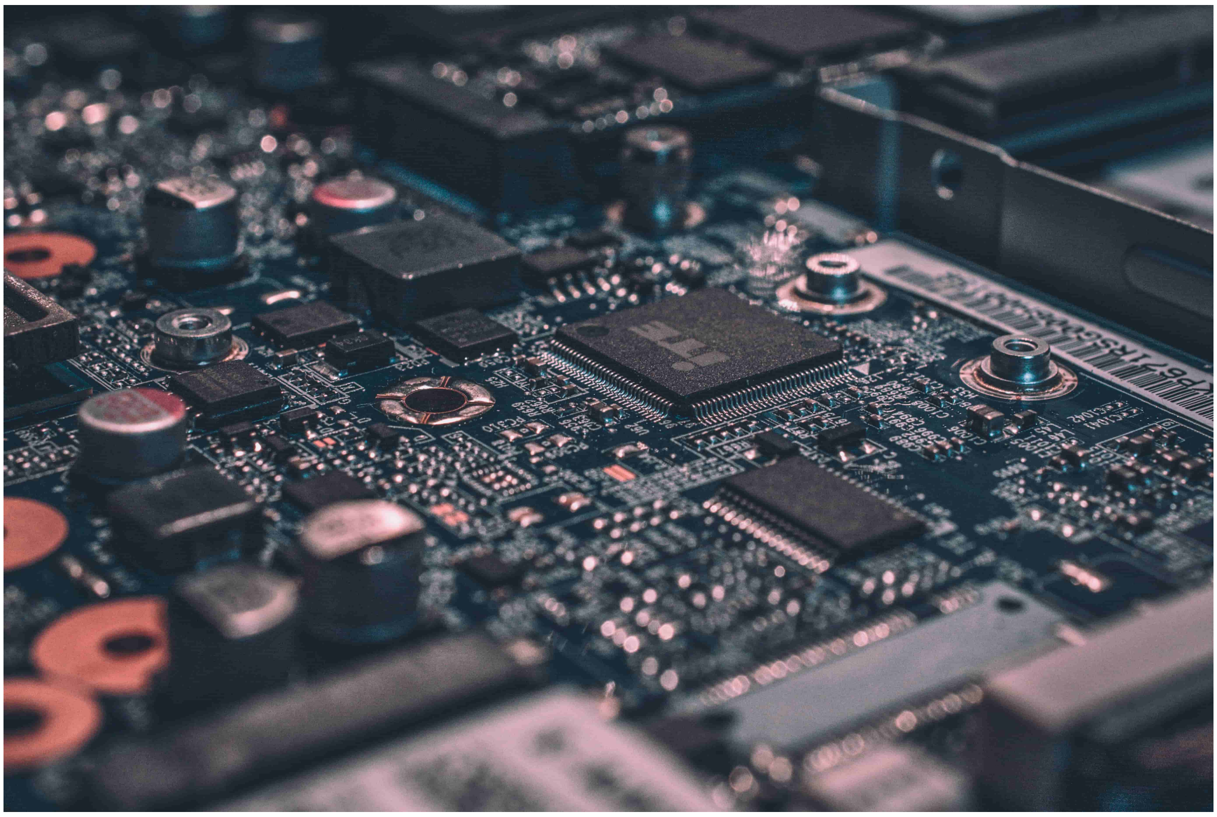
Instructor Name

Sir Usman Hameed

Submitted by

Muhammad Burhan Ahmed
Javeria Razzaq
Muhammad Ali Raza

05/20/2024



Implementation of 8-Bit RISC SPM Processor

ABSTRACT

This project aims to design a 8 bit RISC SPM processor which is capable enough of executing basic instructions. This project involves developing a simplified, yet efficient, 8-bit RISC processor that emphasizes the core concepts of instruction execution, memory hierarchy, and processor design. The processor utilizes a Stored Program Memory architecture, where instructions and data are stored in a single memory space, facilitating streamlined processing and reduced complexity. The project also explores memory management techniques and the interface between the processor and memory components. Through simulation and testing, the performance and functionality of the 8-bit RISC processor are evaluated. The datapath consists of three blocks which include the control unit, the memory unit and the processor. The processor is designed using a Verilog HDL and simulated using Xilinx software. The final implementation of the processor is tested using the test bench.

Contents

1 OBJECTIVES **4**
- 1.0.1 Software Used : 4

2 INTRODUCTION **4**
- 2.0.1 **RISC SPM Key Features:** 4
- 2.1 Instruction Types 4
  - 2.1.1 Short Instruction 4
  - 2.1.2 Long Instruction 5
- 2.2 Control Signals 5
  - 2.2.1 **Blocks of RISC SPM** 6
- 2.3 Processor Architecture 6
- 2.4 Processing Unit 7
  - 2.4.1 Register File 7
  - 2.4.2 Instruction Register(IR) 8
  - 2.4.3 Program Counter(PC) 8
  - 2.4.4 Multiplexers(MUXes) 9
  - 2.4.5 ALU (Arithmetic Logic Unit) 9
- 2.5 Memory Unit 10
- 2.6 Control Unit(CU) 10

3 Verilog Module Code **12**

4 RTL View: **28**
- 4.0.1 Top Module Explaination 28
- 4.0.2 RAM Explaination 29
- 4.0.3 Processor Explaination 30
- 4.0.4 ALU Explaination 31
- 4.0.5 Control Unit Explaination 32
- 4.0.6 3 x 1 MUX Explaination 33
- 4.0.7 5 x 1 MUX Explaination 34
- 4.0.8 Program Counter Explaination 35
- 4.0.9 Instruction Register Explaination 36
- 4.0.10 Address Register Explaination 37
- 4.0.11 Register Unit Explaination 38

5 Test Bench **39**

6 Conclusion **40**

7 References **41**

1 OBJECTIVES

- Designing simplified RISC architecture.
- Implementation using Verilog hardware language.
- Ensuring correctness and performance through simulation.
- Achieving a single-cycle execution model for all instructions, optimizing for speed.

1.0.1 Software Used :

- Xilinx

2 INTRODUCTION

RISC Single Process Multiple Data (SPM)
A Reduced Instruction Set Computer Single Process Multiple Data is a processor that executes single instruction across multiple data elements. The Architecture is designed for data parallel operation commonly found in scientific computing and etc. Basically it is used when we want to process multiple data concurrently.

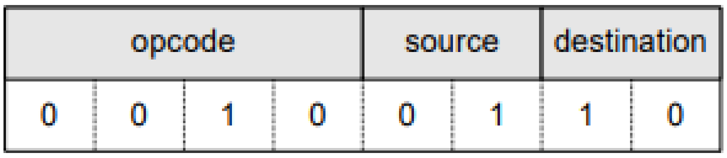
2.0.1 RISC SPM Key Features:

- A RISC store-program machine (SPM) consists of three functional units : a processor, a controller and memory
- Program instructions and data are stored in memory
- Instructions are fetched from memory synchronously, decoded and executed

2.1 Instruction Types

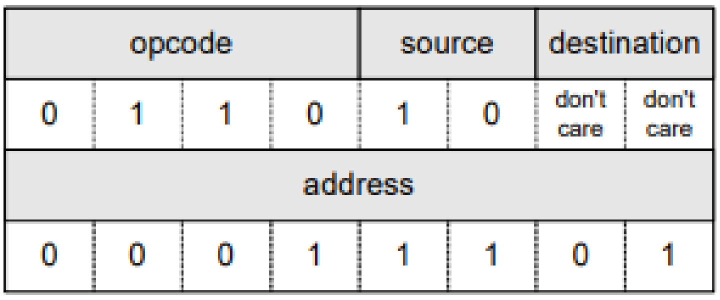
1. Short Instruction
2. Long Instruction

2.1.1 Short Instruction



opcode				source		destination	
0	0	1	0	0	1	1	0

2.1.2 Long Instruction



opcode				source		destination	
0	1	1	0	1	0	don't care	don't care
address							
0	0	0	1	1	1	0	1

2.2 Control Signals

mention all 12 signals along with purpose/usage

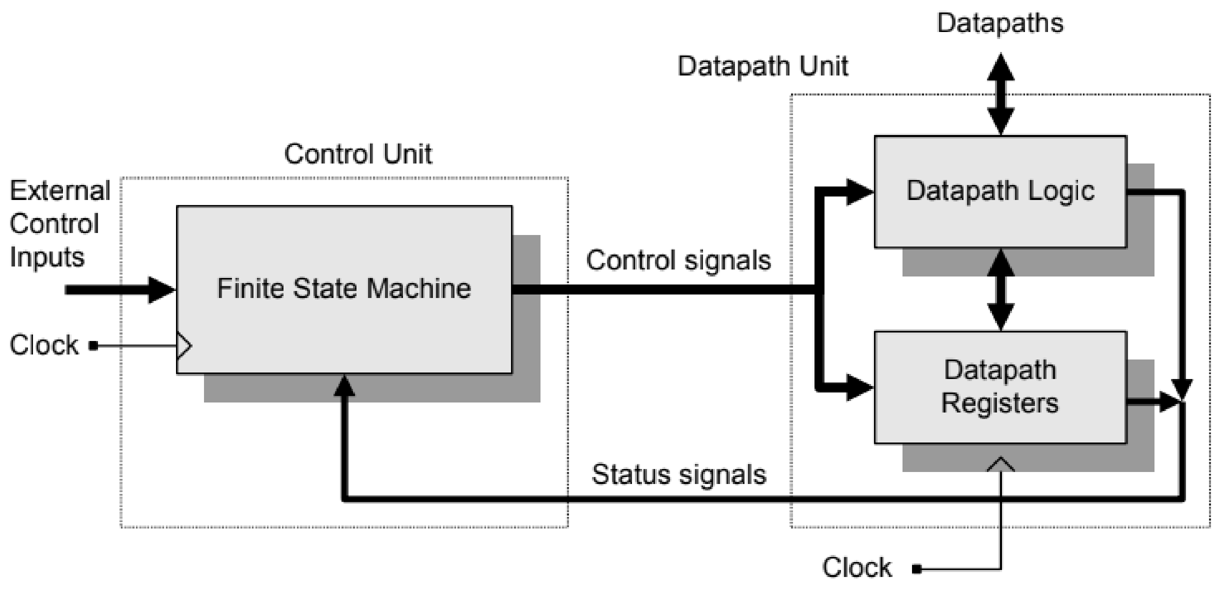
1. **Clock Signal (CLK):** This signal synchronizes the operation of the processor and ensures that actions occur at the appropriate times during each clock cycle.
2. **Reset (RESET):** Resets the processor to a known state, typically clearing registers and setting the program counter to the reset address.
3. **Instruction Fetch (IF):** Indicates that the processor is fetching the next instruction from memory.
4. **Instruction Decode (ID):** Signals that the fetched instruction is being decoded to determine the operation to be performed.
5. **Memory Read (MEM_RD):** Indicates that the processor is reading data from memory.
6. **Memory Write (MEM_WR):** Signals that the processor is writing data to memory.
7. **Register Write (REG_WR):** Indicates that the processor is writing data to one of its internal registers.
8. **ALU Operation (ALU_OP):** Specifies the operation to be performed by the Arithmetic Logic Unit (ALU), such as addition, subtraction, logical AND, logical OR, etc.
9. **Branch (BRANCH):** Signals that the current instruction is a branch instruction, and the processor should update the program counter accordingly.
10. **Jump (JUMP):** Indicates that the current instruction is a jump instruction, directing the processor to jump to a different location in the program.
11. **Load (LOAD):** Signals that the current instruction is a load instruction, and the processor should retrieve data from memory.
12. **Store (STORE):** Indicates that the current instruction is a store instruction, and the processor should store data into memory.

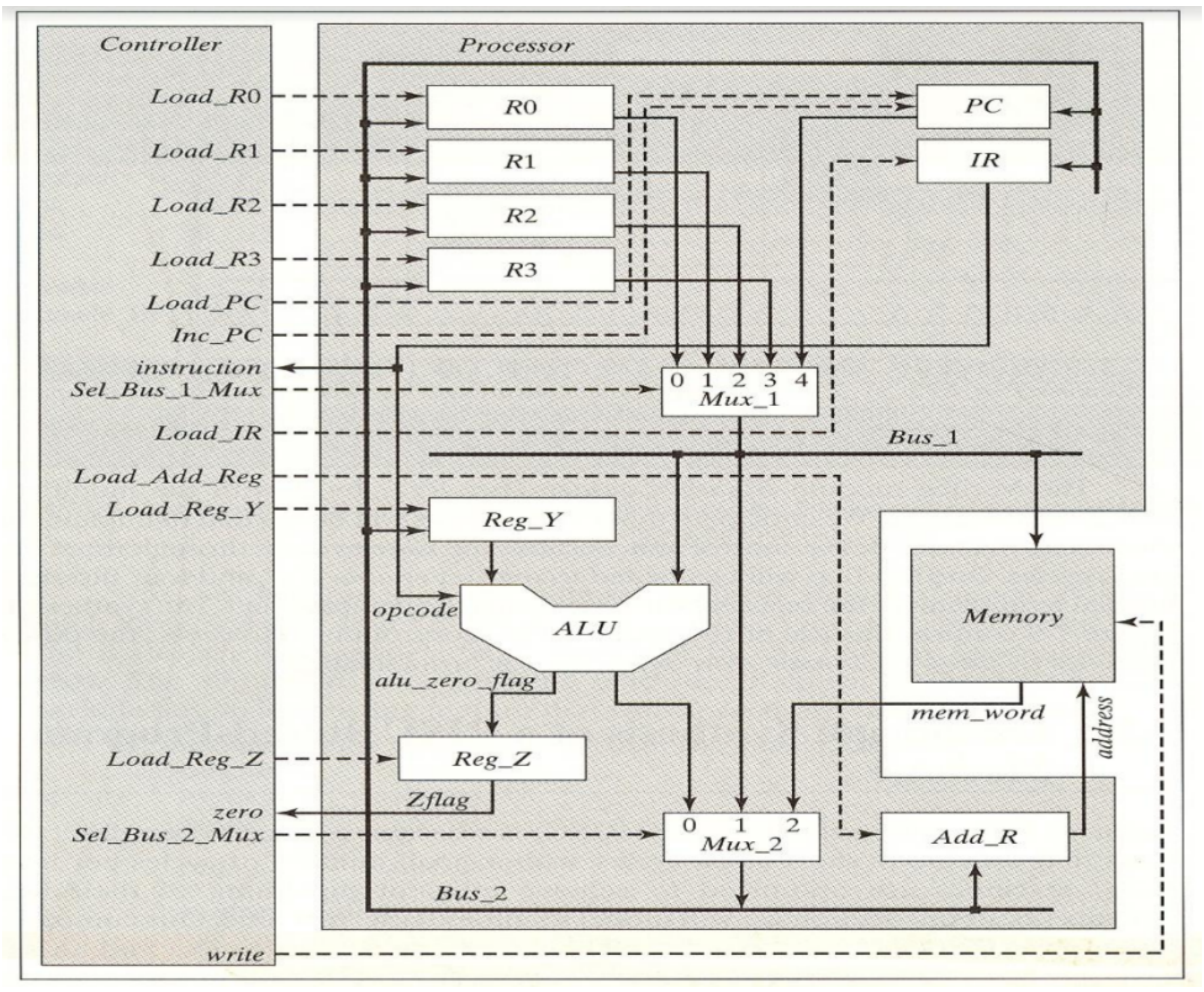
2.2.1 Blocks of RISC SPM

RISC SPM data-path is composed of the following components:

- Processing Unit:
  - Address Register
  - Instruction Register
  - Program Counter
  - Multiplexers
  - Arithmetic Logic Unit
  - Bus1
  - Bus2
  - Register file
- Control Unit
- Memory Unit (RAM)

2.3 Processor Architecture





2.4 Processing Unit

2.4.1 Register File

The Register File (RF) within a computer's CPU serves as a vital repository for a set of registers dedicated to data storage and manipulation. This high-speed memory unit offers swift access to the CPU's general-purpose registers. Typically, the RF encompasses an array of registers, each capable of storing a fixed-size binary value. The number of registers varies depending on the CPU architecture, typically ranging from 8 to 32 or more, with each register identified by a unique index or name. Functionally, the RF serves to provide temporary storage for data during program execution, enabling the CPU to swiftly access and manipulate data without resorting to the slower main memory. These registers are directly accessible by the CPU's arithmetic and logic unit (ALU) for executing arithmetic, logical operations, and data movement. Structured as a bank of registers, the RF assigns specific roles or purposes to each register. Common types include general-purpose registers, capable of storing any data type, and special-purpose registers, designed for tasks like managing program counters, stack pointers, or status flags. Given its integral role, efficient utilization of the RF significantly impacts a computer system's performance. By storing frequently accessed data in registers, the CPU reduces reliance on memory accesses, thereby enhancing overall execution speed.

2.4.2 Instruction Register(IR)

The Instruction Register (IR) is a critical component within a computer's CPU (Central Processing Unit) that holds the current instruction being executed. It is integral to the instruction cycle, particularly in the fetching and decoding of instructions. When the CPU retrieves an instruction from memory, it stores it in the Instruction Register. The control unit of the CPU then uses the contents of the IR to decode the instruction, determining the necessary actions to be taken.

During the decoding process, the control unit extracts the opcode and any additional operands or addressing modes specified by the instruction. The IR typically has a fixed size that corresponds to the length of instructions supported by the CPU architecture. It often consists of multiple fields or bit positions representing different parts of the instruction, such as the opcode, register identifiers, memory addresses, or immediate values.

Once the instruction is decoded, the control unit uses the information from the IR to direct the execution of the instruction. This may involve accessing registers, performing arithmetic or logical operations, or interacting with memory locations. The contents of the IR can change throughout the instruction cycle as the CPU moves through the various stages of instruction execution.

2.4.3 Program Counter(PC)

The Program Counter (PC) is a crucial component in the execution of computer programs. This special register holds the memory address of the next instruction that the processor will fetch and execute. By tracking the program's current position in memory, the PC determines the sequence in which instructions are carried out.

For those with a technical background, the Program Counter is a hardware register that stores the address of the upcoming instruction in memory. After each instruction is fetched, the PC is incremented to point to the next instruction, thus maintaining the correct order of execution. The PC is essential for controlling the program's flow, ensuring that instructions are processed in the right sequence.

For a non-technical audience, the Program Counter can be described as a ”pointer” that guides the computer through a program. It keeps track of the computer's progress through a series of steps or instructions, ensuring that each step is completed before moving on to the next. Think of it as a roadmap that directs the computer to the next instruction, ensuring everything is executed in the correct order.

In adapting the explanations for different audiences, the following writing decisions were made:
Technical Version: Focused on the specific details of the Program Counter, such as its function as a hardware register and its interaction with memory addresses.
Non-Technical Version: Used analogies and simpler terms to explain the concept, making it more accessible to those without a technical background, avoiding overwhelming technical jargon.

2.4.4 Multiplexers(MUXes)

Multiplexers, commonly known as MUXes, are digital electronic devices that select and route one of several input signals to a single output based on control inputs. These control inputs determine which input signal is transmitted to the output, making multiplexers essential in various digital systems and circuits for efficient data routing and selection.

The primary function of a multiplexer is to combine multiple input signals into a single output signal according to the control inputs. A multiplexer consists of two main components: data inputs and control inputs. The number of data inputs varies depending on the specific multiplexer and its configuration.

Control inputs, typically in binary format, determine which data input is selected and routed to the output. The number of control inputs corresponds to the number of selectable inputs. For instance, a 2-to-1 multiplexer has one control input, which can be either 0 or 1, and two data inputs. The control input decides which data input is passed to the output.

Multiplexers can be constructed using various logic gates, such as AND, OR, and NOT gates. The selection of the input signal is based on the combination of logic levels applied to the control inputs, and the output of a multiplexer matches the logic level of the selected input.

Multiplexers have numerous applications in digital systems, including data routing, data selection, and signal switching. In communication systems, for example, multiplexers are used to select different data streams for transmission over a shared channel. In memory systems, they enable the selection of specific memory locations.

In summary, multiplexers are versatile digital devices used to select and route one of several input signals to a single output. They offer a compact and efficient method for managing data selection and routing in digital systems.

2.4.5 ALU (Arithmetic Logic Unit)

The Arithmetic Logic Unit (ALU) is a crucial component of a computer's central processing unit (CPU), responsible for executing arithmetic operations (such as addition, subtraction, multiplication, and division) and logical operations (such as AND, OR, NOT) on binary data.

The ALU receives inputs from the CPU's registers and performs the required operations based on control signals. It processes binary data, represented as bits (0s and 1s), and can handle arithmetic operations on integer values, bitwise operations on binary data, and logical operations to evaluate conditions and make decisions.

The ALU comprises various subcomponents, including arithmetic circuits, logic gates, multiplexers, and control units, which collaborate to execute the desired arithmetic or logical operations. It is designed for parallel data processing, enabling efficient and high-speed computations.

Integral to the CPU's instruction execution process, the ALU performs calculations and data manipulations according to instructions fetched from memory. The results of ALU operations are typically stored in registers or sent to other components for further processing.

In summary, the ALU performs arithmetic and logical operations on binary data within a computer's CPU, playing a critical role in executing instructions and performing the computations necessary for the computer system to function.

2.5 Memory Unit

The Memory Unit, commonly known as the main memory or RAM (Random Access Memory), is a crucial component of a computer system where data is stored and retrieved during program execution. Separate from the CPU, it holds both program instructions and data that are actively being processed.

Data Memory is organized into addressable storage locations, each capable of holding a fixed number of bits or bytes of data. These locations are accessed using memory addresses, unique identifiers for each storage spot.

Data Memory plays a vital role in program execution, storing variables, arrays, structures, and other data types used by the CPU. When the CPU needs to read or write data, it sends the appropriate memory address to Data Memory, and the requested data is transferred between the memory and CPU registers.

The size of Data Memory determines how much data can be stored and accessed by the CPU, typically measured in bytes or kilobytes. The capacity varies depending on the specific computer system.

In modern computer architectures, Data Memory is usually implemented as dynamic random-access memory (DRAM) or static random-access memory (SRAM). DRAM offers higher storage capacity but slower access times, while SRAM provides faster access times but lower storage density.

In summary, Data Memory serves as temporary storage for data actively processed by the CPU, enabling data manipulation and retrieval during program execution. It is an essential component of a computer system.

2.6 Control Unit(CU)

The Control Unit (CU) is a vital part of a computer's central processing unit (CPU), responsible for coordinating and controlling the CPU's operations. It ensures that instructions are executed in the correct sequence and that data is accurately processed and transferred.

The Control Unit performs several key functions:

- **Instruction Decoding:** The CU receives instructions from memory and decodes them to determine the specific operations to be performed. It interprets the instruction's opcode (operation code) and identifies the required data operands. - **Instruction Sequencing:** The CU ensures that instructions are executed in the correct order. It controls the flow of instructions, determining which instruction should be executed next based on the program counter and the current instruction being executed. **Operand Fetching:** The CU retrieves the data operands required by the instructions from memory or registers. It fetches data from the appropriate memory locations or registers and makes it available for processing by the CPU. **Execution Control:** The CU controls the execution of arithmetic and logical operations within the CPU. It activates the appropriate functional units, such as the Arithmetic Logic Unit (ALU), to perform the required computations. **Register Transfer:** The CU manages the transfer of data between registers, memory, and other CPU components. It ensures that data is moved to the correct locations and that intermediate results are stored appropriately. **Branch and Jump Handling:** The CU handles branching and jumping instructions, determining the correct address to branch or jump to based on the conditions specified in the instructions. It updates the program counter accordingly.

The Control Unit uses a combination of logic circuits, microcode, and control signals to carry out its functions. It interacts closely with other CPU components, such as the ALU, registers, and memory, to execute instructions and process data.

Overall, the Control Unit is crucial for the execution of instructions and the control of data flow within the CPU. It ensures that instructions are properly decoded, executed, and coordinated, allowing for the orderly execution of programs and the processing of data.

3 Verilog Module Code

- Main Data path:

```
module top(
  clk,
  clr
);
    parameter DATAWIDTH = 8;
    parameter sel1_size = 3;
    parameter sel2_size = 2;
    wire [sel1_size - 1:0] sel_bus1_mux;
    wire [sel2_size - 1:0] sel_bus2_mux;
    input clk, clr;

    //Data nets
    wire zero;
    wire [DATAWIDTH - 1:0] instruction, address, bus1, mem_word;

    //Control nets
    wire ld_r0, ld_r1, ld_r2, ld_r3, ld_pc, inc_pc, ld_ir;
    wire ld_address_reg, ld_reg_y, ld_reg_z;
    wire write;

  processing_unit processor(
    instruction,
    zero,
    address,
    bus1,
    mem_word,
    ld_r0,
    ld_r1,
    ld_r2,
    ld_r3,
    ld_pc,
    inc_pc,
    sel_bus1_mux,
    ld_ir,
    ld_address_reg,
    ld_reg_y,
    ld_reg_z,
    sel_bus2_mux,
    clk,
    clr
    );

    control_unit controller(
    ld_r0,
    ld_r1,
    ld_r2,
    ld_r3,
    ld_pc,
    inc_pc,
    sel_bus1_mux,
    sel_bus2_mux,
    ld_ir,
    ld_address_reg,
    ld_reg_y,
    ld_reg_z,
    write,
    instruction,
    zero,
    clk,
    clr
    );

    memory_unit RAM(
    .data_out(mem_word),
    .data_in(bus),
    .address(address),
    .clk(clk),
    .write(write)
    );

endmodule
```

- Processing Unit:

```
module processing_unit (
    instruction,
    zero_flag,
    address,
    bus1,
    mem_word,
    ld_r0,
    ld_r1,
    ld_r2,
    ld_r3,
    ld_pc,
    inc_pc,
    sel_bus1_mux,
    ld_ir,
    ld_address_reg,
    ld_reg_y,
    ld_reg_z,
    sel_bus2_mux,
    clk,
    clr
);

// Parameters declaration
```

```verilog
parameter DATAWIDTH = 8;
parameter opcode_size = 4;
parameter sel1_size = 3;
parameter sel2_size = 2;

// Ports declaration

output [DATAWIDTH - 1:0] instruction, address, bus1;
output zero_flag;

input [DATAWIDTH - 1:0] mem_word;
input ld_r0, ld_r1, ld_r2, ld_r3, ld_pc, inc_pc;

input [sel1_size - 1:0] sel_bus1_mux;
input [sel2_size - 1:0] sel_bus2_mux;

input ld_ir, ld_address_reg, ld_reg_z, ld_reg_y, clk, clr;

// Intermediate wires declaration

wire ld_r0, ld_r1, ld_r2, ld_r3;
wire [DATAWIDTH - 1:0] bus2;
wire [DATAWIDTH - 1:0] out_r0, out_r1, out_r2, out_r3;
wire [DATAWIDTH - 1:0] pc_count, out_reg_y, alu_out;
wire alu_zero_flag;
wire [opcode_size - 1:0] opcode = instruction [DATAWIDTH - 1:DATAWIDTH -
    opcode_size];

// Instantiations of register units, flipflops, address register, instruction
    register, program counter, ALU and Multiplexers

register_unit R0 (out_r0, bus2, ld_r0, clk, clr);
register_unit R1 (out_r1, bus2, ld_r1, clk, clr);
register_unit R2 (out_r2, bus2, ld_r2, clk, clr);
register_unit R3 (out_r3, bus2, ld_r3, clk, clr);
register_unit REG_Y (out_reg_y, bus2, ld_reg_y, clk, clr);
dff REG_Z (zero_flag, alu_zero_flag, ld_reg_z, clk, clr);
address_register ADDR_REG (address, bus2, ld_address_reg, clk, clr);
instruction_register IR (instruction, bus2, ld_ir, clk, clr);
program_counter PC (pc_count, bus2, ld_pc, inc_pc, clk, clr);
mux_5channel MUX1 (bus1, out_r0, out_r1, out_r2, out_r3, pc_count,
    sel_bus1_mux);
mux_3channel MUX2 (bus2, alu_out, bus1, mem_word, sel_bus2_mux);
alu_risc ALU (alu_zero_flag, alu_out, out_reg_y, bus1, opcode);

endmodule
```

– Register File :

```
module register_unit (
    data_out,
    data_in,
    load,
    clk,
    clr
);

parameter DATAWIDTH = 8;
output reg [DATAWIDTH - 1:0] data_out;
input [DATAWIDTH - 1:0] data_in;
input load, clk, clr;

// Clear is an active low signal

always @ (posedge clk or negedge clr)
begin
    if(!clr)
        data_out <= 0;
    else if (load)
      data_out <= data_in;
end

endmodule
```

```
module dff (
   output reg data_out,
   input data_in,
   input load,
   input clk,
   input clr
);

  always @ (posedge clk or negedge clr)
  begin
      if(!clr)
        data_out <= 0;
      else if (load) begin
        data_out <= data_in;
    end
  end

endmodule
```

– Instruction Register :

```
module instruction_register(
    data_out,
    data_in,
    load,
    clk,
    clr
);

parameter DATAWIDTH = 8;
output reg [DATAWIDTH - 1:0] data_out;
input [DATAWIDTH - 1:0] data_in;
input load, clk, clr;

// Clear is an active low signal

always @ (posedge clk or negedge clr)
begin
    if(!clr)
        data_out <= 0;
    else if (load) begin
        data_out <= data_in;
    end
end

endmodule
```

– Program Counter :

```
module program_counter(
    count,
    data_in,
    ld_pc,
    inc_pc,
    clk,
    clr
);

parameter DATAWIDTH = 8;
output reg [DATAWIDTH - 1:0] count;
input [DATAWIDTH - 1:0] data_in;
input ld_pc, inc_pc, clk, clr;

// Clear is an active low signal

always @ (posedge clk or negedge clr)
begin
    if(!clr)
        count <= 0;
    else if (ld_pc)
        count <= data_in;
    else if (inc_pc)
        count <= count + 1;
end
```

```
endmodule
```

– Multiplexers :

```
module mux_3channel (
    mux_out,
    in1,
    in2,
    in3,
    select
);

parameter DATAWIDTH = 8;
output [DATAWIDTH - 1:0] mux_out;
input [DATAWIDTH - 1:0] in1, in2, in3;
input [1:0] select;

assign mux_out = (select == 0) ? in1 : (select == 1)
                               ? in2 : (select == 2)
                               ? in3 : 'bx;

endmodule
```

```
module mux_5channel (
    mux_out,
    in1,
    in2,
    in3,
    in4,
    in5,
    select
);

parameter DATAWIDTH = 8;
output [DATAWIDTH - 1:0] mux_out;
input [DATAWIDTH - 1:0] in1, in2, in3, in4, in5;
input [2:0] select;

assign mux_out = (select == 0) ? in1 : (select == 1)
                               ? in2 : (select == 2)
                               ? in3 : (select == 3)
                               ? in4 : (select == 4)
                               ? in5 : 'bx;

endmodule
```

– Arithmetic Logic Unit :

```
module alu_risc (
    alu_zero_flag,
    alu_out,
    data_1,
    data_2,
    select
);

  /* ALU Instruction              Action
    ADD                           Adds datapaths to form data_1 + data_2
    SUB                           Subtracts datapaths to form data_1 -
       data_2
    AND                           Takes bitwise and of datapaths data_1 and
       data_2
    NOT                           Takes bitwise boolean complement of data_1
  */

  parameter DATAWIDTH = 8;
  parameter opcode_size = 4;

  // Op-codes
  parameter NOP = 4'b0000;
  parameter ADD = 4'b0001;
  parameter SUB = 4'b0010;
  parameter AND = 4'b0011;
  parameter NOT = 4'b0100;
  parameter RD = 4'b0101;
  parameter WR = 4'b0110;
  parameter BR = 4'b0111;
  parameter BRZ = 4'b1000;

  //Port declarations
  output alu_zero_flag;
  output reg [DATAWIDTH - 1:0] alu_out;
  input [DATAWIDTH - 1:0] data_1, data_2;
  input [opcode_size - 1:0] select;

  assign alu_zero_flag = ~|alu_out;

  always @ (select or data_1 or data_2)
  begin
      case(select)
        NOP: alu_out = 0;
        ADD: alu_out = data_1 + data_2;
        SUB: alu_out = data_1 - data_2;
        AND: alu_out = data_1 & data_2;
        NOT: alu_out = ~data_2; // WTF is going on
        default: alu_out = 0;
      endcase
  end
endmodule
```

- Control Unit :

```verilog
module control_unit(
    ld_r0,
    ld_r1,
    ld_r2,
    ld_r3,
    ld_pc,
    inc_pc,
    sel_bus1_mux,
    sel_bus2_mux,
    ld_ir,
    ld_address_reg,
    ld_reg_y,
    ld_reg_z,
    write,
    instruction,
    zero,
    clk,
    clr
);

parameter DATAWIDTH = 8;
parameter opcode_size = 4;
parameter state_size = 4;

parameter src_size = 2;
parameter dest_size = 2;
parameter sel1_size = 3;
parameter sel2_size = 2;

// State codes

parameter idle = 0;
parameter fetch1 = 1;
parameter fetch2 = 2;
parameter decode = 3;
parameter execute = 4;
parameter read1 = 5;
parameter read2 = 6;
parameter write1 = 7;
parameter write2 = 8;
parameter branch1 = 9;
parameter branch2 = 10;
parameter halt = 11;

// Op-codes

parameter NOP = 0;
parameter ADD = 1;
parameter SUB = 2;
parameter AND = 3;
parameter NOT = 4;
parameter RD = 5;
parameter WR = 6;
parameter BR = 7;
parameter BRZ = 8;

// Source & Destination codes

parameter R0 = 0;
parameter R1 = 1;
parameter R2 = 2;
parameter R3 = 3;

output reg ld_r0, ld_r1, ld_r2, ld_r3, ld_pc, inc_pc, ld_ir, ld_address_reg,
    ld_reg_y, ld_reg_z, write;
output [sel1_size - 1:0] sel_bus1_mux;
output [sel2_size - 1:0] sel_bus2_mux;
input [DATAWIDTH - 1:0] instruction;
input zero, clk, clr;

reg [state_size - 1:0] state, next_state;
reg error_flag, sel_alu, sel_bus1, sel_mem, sel_r0, sel_r1, sel_r2, sel_r3,
    sel_pc;

wire [opcode_size - 1:0] opcode = instruction [DATAWIDTH - 1:DATAWIDTH -
    opcode_size];
wire [src_size - 1:0] src = instruction [src_size + dest_size - 1:dest_size];
wire [dest_size - 1:0] dest = instruction [dest_size - 1:0];

// Mux selectors

assign sel_bus1_mux [sel1_size - 1:0] = sel_r0 ? 0 : sel_r1 ? 1 : sel_r2 ? 2 :
    sel_r3 ? 3 : sel_pc ? 4 : 3'bx;
assign sel_bus2_mux [sel2_size - 1:0] = sel_alu ? 0 : sel_bus1 ? 1 : sel_mem ?
    2 : 2'bx;

always @ (posedge clk or negedge clr)
begin
    if(!clr)
        state <= idle;
    else
        state <= next_state;
end

always @(state or opcode or src or dest or zero) begin
    sel_r0 = 0;
    sel_r0 = 0;
    sel_r0 = 0;
    sel_r0 = 0;
    sel_r0 = 0;
```

```
ld_r0 = 0;
ld_r1 = 0;
ld_r2 = 0;
ld_r3 = 0;

ld_pc = 0;
ld_ir = 0;
ld_address_reg = 0;
ld_reg_y = 0;
ld_reg_z = 0;
inc_pc = 0;
sel_bus1 = 0;
sel_alu = 0;
sel_mem = 0;
write = 0;
error_flag = 0;
next_state = state;

case(state)
    idle: next_state = fetch1;
    fetch1: begin
        next_state = fetch2;
        sel_pc = 1;
        sel_bus1 = 1;
        ld_address_reg = 1;
    end

    fetch2: begin
        next_state = decode;
        sel_mem = 1;
        ld_ir = 1;
        inc_pc = 1;
    end

    decode: case (opcode)
                NOP: next_state = fetch1;
                ADD, SUB, AND: begin
                    next_state = execute;
                    sel_bus1 = 1;
                    ld_reg_y = 1;
                case (src)
                    R0: sel_r0 = 1;
                    R1: sel_r1 = 1;
                    R2: sel_r2 = 1;
                    R3: sel_r3 = 1;
                    default: error_flag = 1;
                endcase
                end

                NOT: begin
                    next_state = fetch1;
    ld_reg_z = 1;
    sel_bus1 = 1;
    sel_alu = 1;
    case (src)
    R0: sel_r0 = 1;
    R1: sel_r1 = 1;
    R2: sel_r2 = 1;
    R3: sel_r3 = 1;
    default: error_flag = 1;
    endcase

case (dest)
    R0: ld_r0 = 1;
    R1: ld_r1 = 1;
    R2: ld_r2 = 1;
    R3: ld_r3 = 1;
    default: error_flag = 1;
endcase
end
RD: begin
    next_state = read1;
    sel_pc = 1;
    sel_bus1 = 1;
    ld_address_reg = 1;
end

WR: begin
    next_state = write1;
    sel_pc = 1;
    sel_bus1 = 1;
    ld_address_reg = 1;
end

BR: begin
    next_state = branch1;
    sel_pc = 1;
    sel_bus1 = 1;
    ld_address_reg = 1;
end

BRZ: if(zero == 1) begin
    next_state = branch1;
    sel_pc = 1;
    sel_bus1 = 1;
    ld_address_reg = 1;
end
else begin
    next_state = fetch1;
    inc_pc = 1;
end
default: next_state = halt;
```

```
        endcase

execute: begin
    next_state = fetch1;
    ld_reg_z = 1;
    sel_alu = 1;
    case(dest)
        R0: begin sel_r0 = 1; ld_r0 = 1; end
        R1: begin sel_r1 = 1; ld_r1 = 1; end
        R2: begin sel_r2 = 1; ld_r2 = 1; end
        R3: begin sel_r3 = 1; ld_r3 = 1; end
        default: error_flag = 1;
endcase
end

read1: begin
    next_state = read2;
    sel_mem = 1;
    ld_address_reg = 1;
    inc_pc = 1;
end

write1: begin
    next_state = write2;
    sel_mem = 1;
    ld_address_reg = 1;
    inc_pc = 1;
end

read2: begin
    next_state = fetch1;
    sel_mem = 1;
    case (dest)
        R0: ld_r0 = 1;
        R1: ld_r1 = 1;
        R2: ld_r2 = 1;
        R3: ld_r3 = 1;
        default: error_flag = 1;
    endcase
end

write2: begin
    next_state = fetch1;
    write = 1;
    case (src)
    R0: sel_r0 = 1;
    R1: sel_r1 = 1;
    R2: sel_r2 = 1;
    R3: sel_r3 = 1;
    default: error_flag = 1;
    endcase
```

```
        end

        branch1: begin
            next_state = branch2;
            sel_mem = 1;
            ld_address_reg = 1;
        end

        branch2: begin
            next_state = fetch1;
            sel_mem = 1;
            ld_pc = 1;
        end

        halt: next_state = halt;
        default: next_state = idle;
endcase
end
endmodule
```

- Memory Unit (RAM) :

```
// A simple unit of memory can be arranged as an array of D flipflops

module memory_unit (
    data_out,
    data_in,
    address,
    clk,
    write
);

  parameter DATAWIDTH = 8;
  parameter memory_size = 256;

  output wire [DATAWIDTH - 1:0] data_out;
  input [DATAWIDTH - 1:0] data_in;
  input [DATAWIDTH - 1:0] address;
  input clk, write;

  reg [DATAWIDTH - 1:0] memory [memory_size - 1:0];

  assign data_out = memory[address];

  always @ (posedge clk)
  begin
      if(write)
      memory[address] <= data_in;
  end
  //assign data_out = memory[address];
```

```
endmodule
```

- TestBench:

```
`timescale 1ns / 1ps

////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 21:54:54 05/26/2024
// Design Name: top
// Module Name: D:/Bwork/DSD/CEP/RISC SPM/RISC/Testing.v
// Project Name: RISC
// Target Device:
// Tool versions:
// Description:
//
// Verilog Test Fixture created by ISE for module: top
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////

module Testing #(parameter word_size = 8)();
  reg rst;
  wire clk;
  reg [8:0]k;
  Clock_Unit M1(clk);
  top M2(clk, rst);

  // define probes
  wire [word_size-1: 0]word0, word1, word2, word3, word4, word5, word6, word7,
     word8, word9, word10, word11, word12, word13,
                      word14,word128, word129, word130, word131, word132,
                         word133, word134, word135, word136, word137, word255,
                      word138, word139, word140;

  assign    word0 = M2.RAM.memory[0],
            word1 = M2.RAM.memory[1],
            word2 = M2.RAM.memory[2],
            word3 = M2.RAM.memory[3],

            word4 = M2.RAM.memory[4],
            word5 = M2.RAM.memory[5],
            word6 = M2.RAM.memory[6],
            word7 = M2.RAM.memory[7],
            word8 = M2.RAM.memory[8],
            word9 = M2.RAM.memory[9],
            word10 = M2.RAM.memory[10],
            word11 = M2.RAM.memory[11],
            word12 = M2.RAM.memory[12],
            word13 = M2.RAM.memory[13],
            word14 = M2.RAM.memory[14],
            word128 = M2.RAM.memory[128],
            word129 = M2.RAM.memory[129],
            word130 = M2.RAM.memory[130],
            word131 = M2.RAM.memory[131],
            word132 = M2.RAM.memory[132],
            word133 = M2.RAM.memory[133],
            word134 = M2.RAM.memory[134],
            word135 = M2.RAM.memory[135],
            word136 = M2.RAM.memory[136],
            word137 = M2.RAM.memory[137],
            word138 = M2.RAM.memory[138],
            word140 = M2.RAM.memory[140],
            word255 = M2.RAM.memory[255];

   initial #2800 $finish;

   // Flush Memory
   initial begin: Flush_Memory
   #2 rst = 0;
   for (k=0; k<=255; k=k+1)
      M2.RAM.memory[k] = 0;
      #10 rst = 1;
    end

   initial begin: Load_program
   #5
                                              // opcode_src_dest
   M2.RAM.memory[0] = 8'b0000_00_00;       // NOP
   M2.RAM.memory[1] = 8'b0101_00_10;       // Read 130 to R2
   M2.RAM.memory[2] = 130;
   M2.RAM.memory[3] = 8'b0101_00_11;      // Read 131 to R3
   M2.RAM.memory[4] = 131;
   M2.RAM.memory[5] = 8'b0101_00_01;      // Read 128 to R1
   M2.RAM.memory[6] = 128;
   M2.RAM.memory[7] = 8'b0101_00_00;      // Read 129 to R0
   M2.RAM.memory[8] = 129;
   M2.RAM.memory[9] = 8'b0010_00_01;      // Sub R1-R0 to R1
   M2.RAM.memory[10]= 8'b1000_00_00;      // BRZ
   M2.RAM.memory[11] = 134;             // Holds address for BRZ
   M2.RAM.memory[12]= 8'b0001_10_11;      // Add R2+R3 to R3 // BR
   M2.RAM.memory[13] = 8'b0111_00_11;     // BR
   M2.RAM.memory[14] = 140;
   // Load data
```

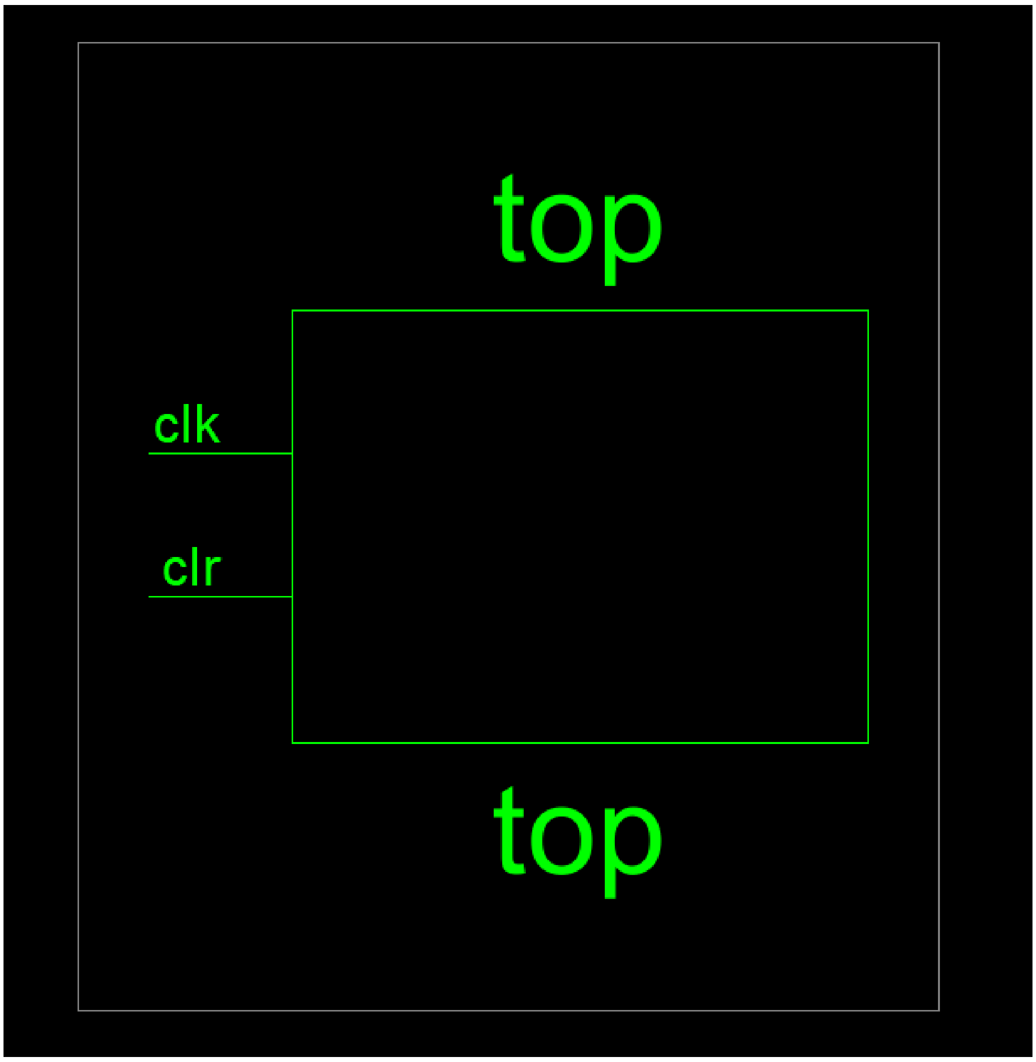
```
   M2.RAM.memory[128] = 6;
   M2.RAM.memory[129] = 1;
   M2.RAM.memory[130] = 2;
   M2.RAM.memory[131] = 0;
   M2.RAM.memory[134] = 139;
   M2.RAM.memory[139] = 8'b1111_00_00;     // HALT
   M2.RAM.memory[140] = 9;                 // Recycle
  end

endmodule

module Clock_Unit (output reg clock);
  parameter delay = 0;
  parameter half_cycle = 10;
  initial begin #delay clock = 0;
  forever #half_cycle clock = ~clock; end
endmodule
```

4 RTL View:

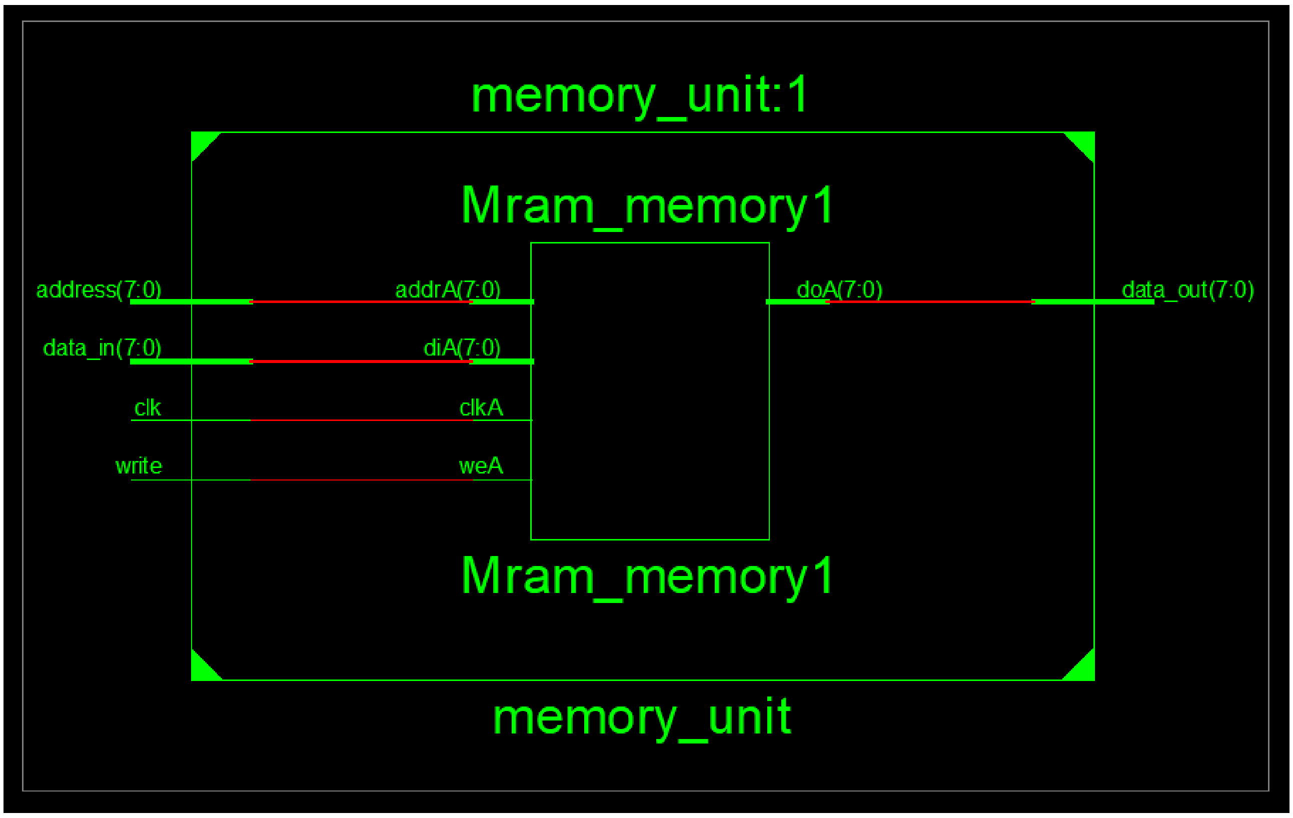
Top Module RTL



4.0.1 Top Module Explaination

The top module acts as the orchestrator of the digital system by instantiating and interconnecting the processing unit, control unit, and memory unit. It facilitates the flow of data and control signals between these modules, ensuring the coherent execution of instructions. As the central hub, the top module coordinates the interaction among the various components, enabling the system to function smoothly and efficiently.

Random Access Memory RTL



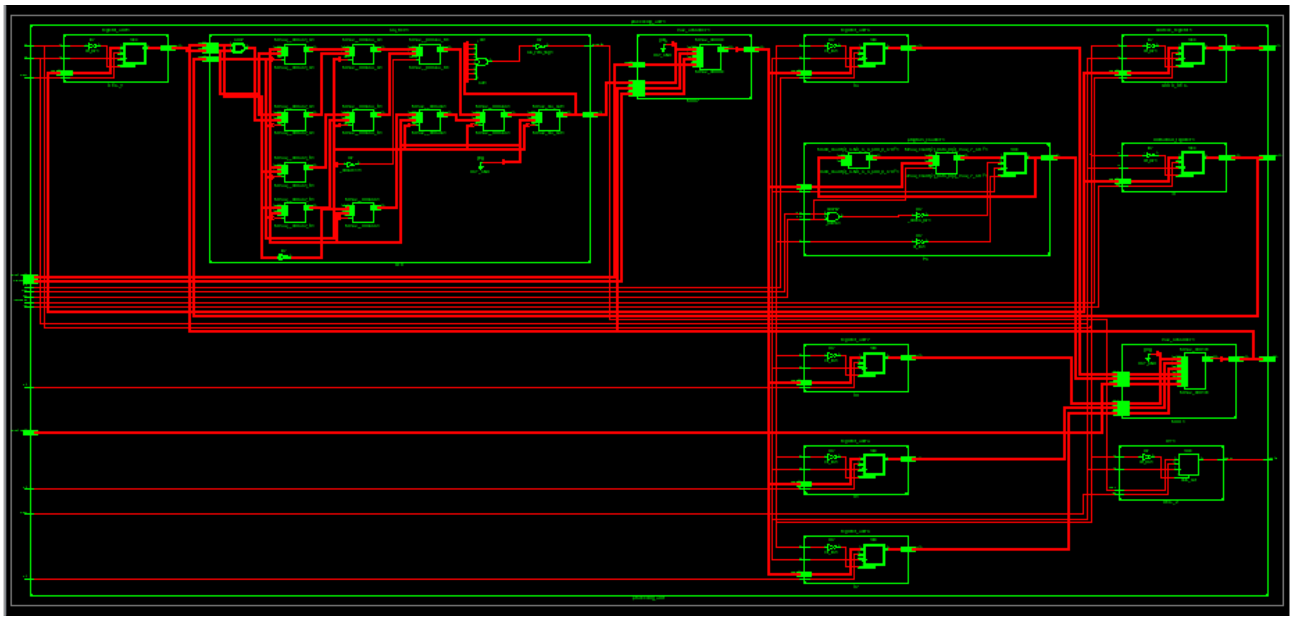
4.0.2 RAM Explaination

Read Operation: When the write signal is not asserted, the memory outputs data (data out) from the specified address.

Write Operation: When the write signal is asserted, data from the data input (data in) is written into memory at the specified address on the positive edge of the clock (clk).

Synchronization: All operations are synchronized with the clock signal (clk) to ensure proper timing and consistency in data handling.

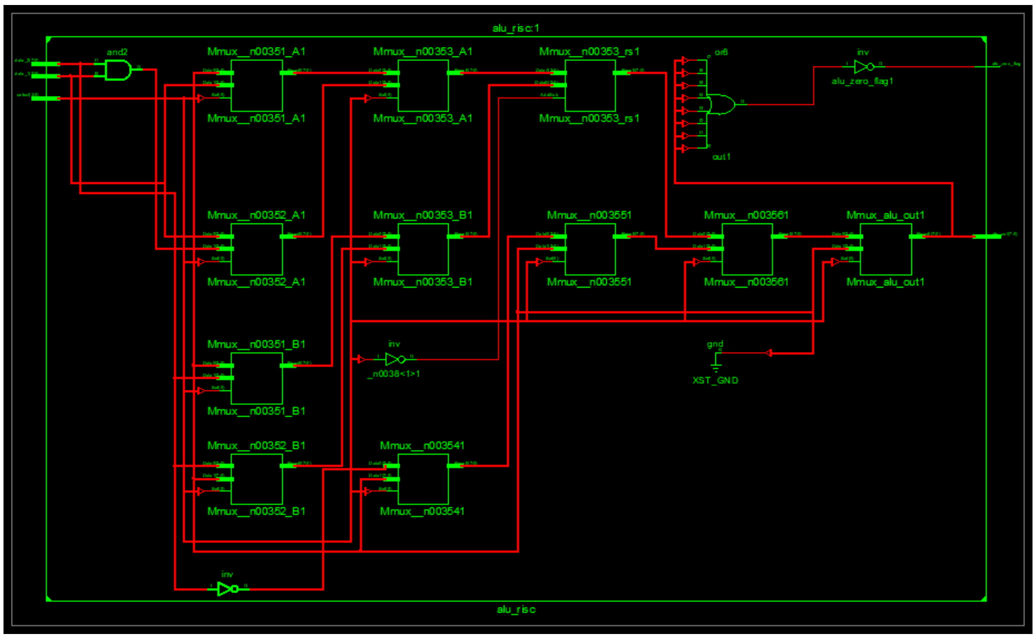
Processor RTL



4.0.3 Processor Explaination

The "Parameters" section defines configurations for data width and sizes of various components, while the "Ports" section declares input and output ports for communication with other modules, encompassing signals for instructions, data, control, clock, and clear. In the "Intermediate Wires" section, wires are declared to facilitate data flow between different parts of the processing unit. The "Instantiations" section involves the instantiation of sub-modules for storing data, managing program flow, executing operations, and selecting inputs. Finally, in the "Functionality" section, operations such as reading instructions, executing arithmetic and logic operations, and managing register values based on control signals and clock are performed.

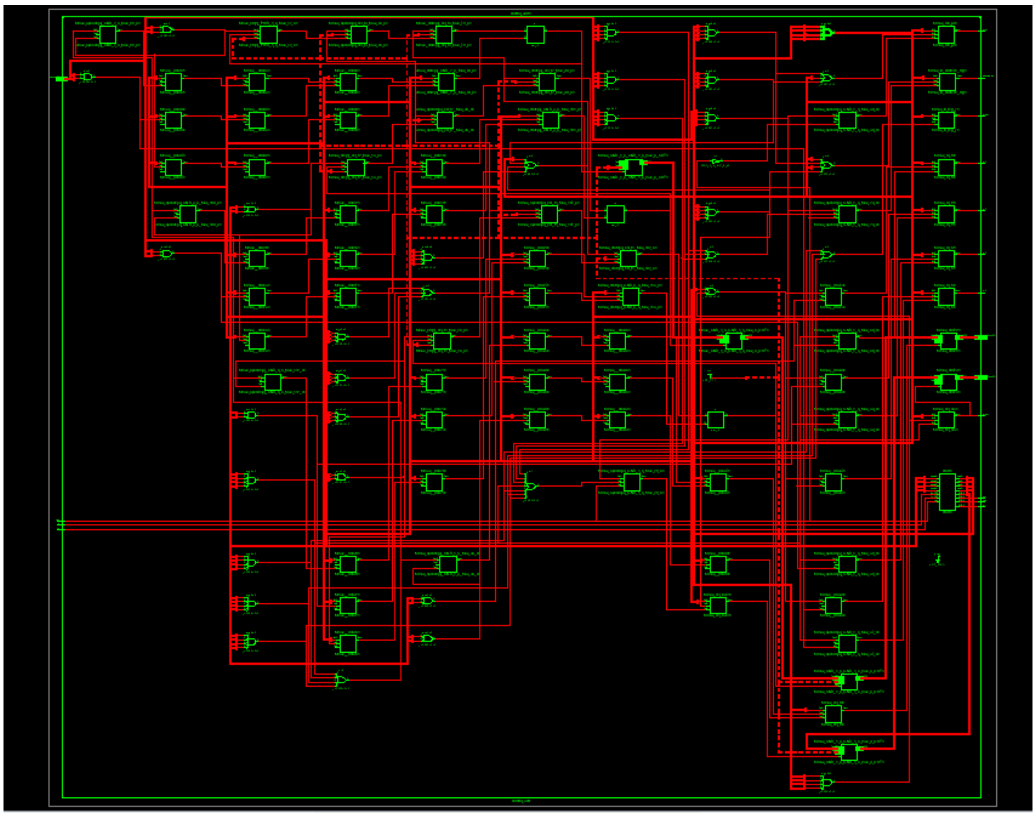
Arithmetic Logic Unit RTL



4.0.4 ALU Explaination

The ALU operates by determining the ALU operation based on the select input opcode and computes the ALU result (alu out) accordingly. It also calculates the alu zero flag based on the result of the ALU operation. Supporting operations such as ADD, SUB, AND, and NOT with specific opcodes, the ALU handles each opcode case using a case statement within an always block.

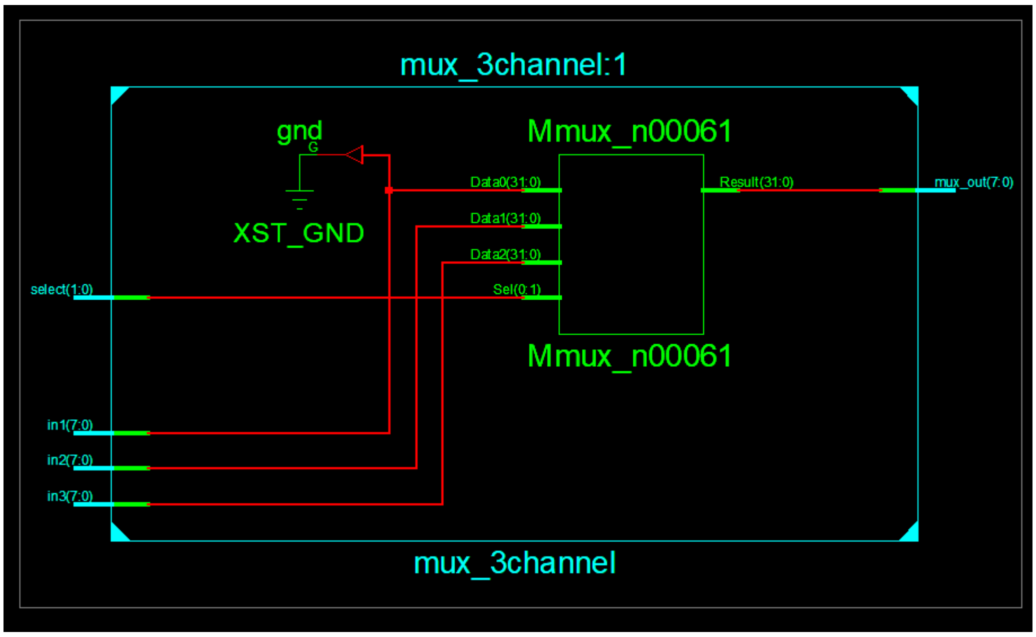
Control Unit Unit RTL



4.0.5 Control Unit Explaination

The control unit, a Verilog module within a processor, manages control signals crucial for its operation. It oversees tasks such as register loading, multiplexer selection, memory operations, and state transitions. Supporting various states including idle, fetch, decode, execute, read, write, branch, and halt, it handles op-codes like NOP, ADD, SUB, AND, NOT, RD, WR, BR, and BRZ. Responding to signals such as clock (clk), clear (clr), instruction input (instruction), and zero flag (zero), the control unit utilizes a state machine architecture to facilitate conditional state transitions and signal generation corresponding to different instruction execution stages.

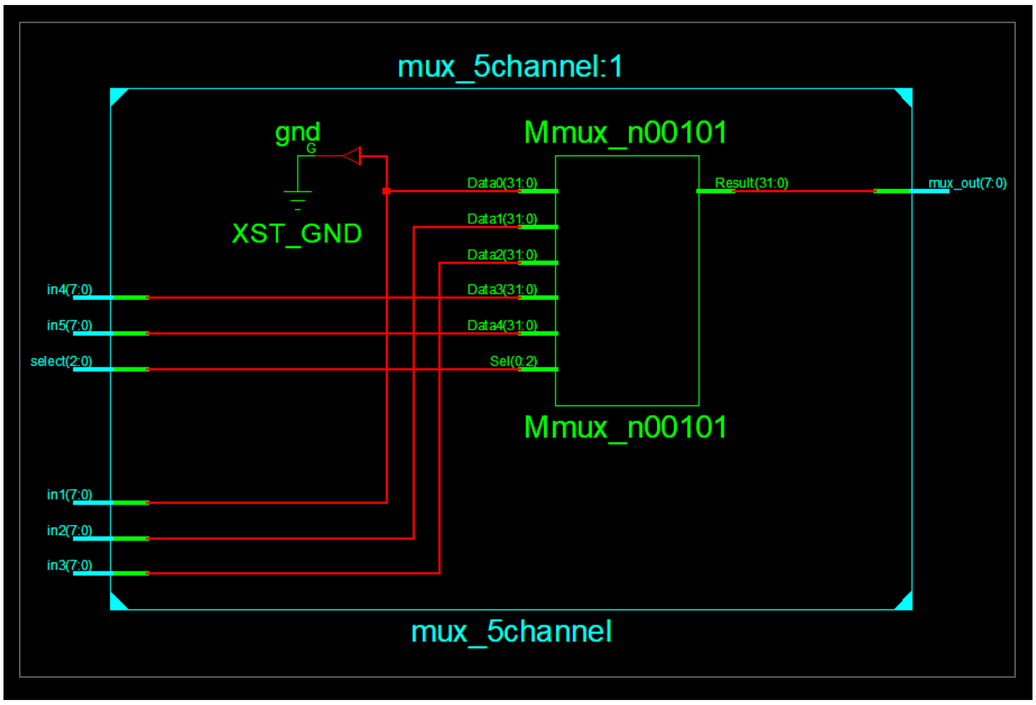
3-Multiplexer Channel RTL



4.0.6 3 x 1 MUX Explaination

The mux 3-channel module is a parameterized 3-to-1 multiplexer with a typically set DATAWIDTH of 8 bits. It functions by selecting between three input signals (in1, in2, and in3) based on the select input. The output, mux out, reflects the chosen input, with select values 0, 1, and 2 corresponding to in1, in2, and in3, respectively. Selecting invalid values leads to an output of 'bx', representing an undefined value.

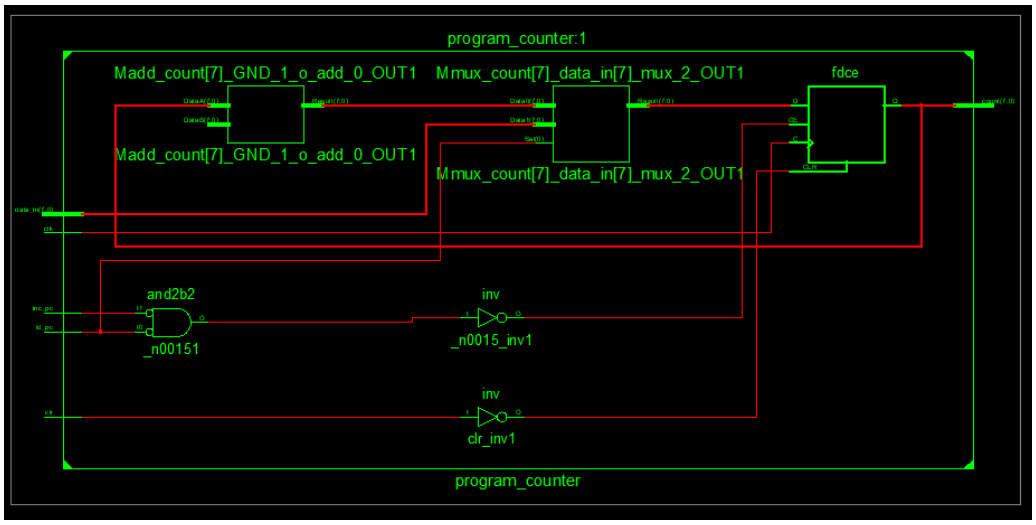
5-Multiplexer Channel RTL



4.0.7 5 x 1 MUX Explaination

The mux 5-channel Verilog module implements a 5-to-1 multiplexer, typically supporting a DATAWIDTH parameter set to 8 bits. Inputs in1 through in5 are multiplexed based on the 3-bit select input, with the output, mux out, reflecting the chosen input. Each value of select corresponds to one of the input signals. In cases of invalid select values, the output results in 'bx', indicating an undefined value.

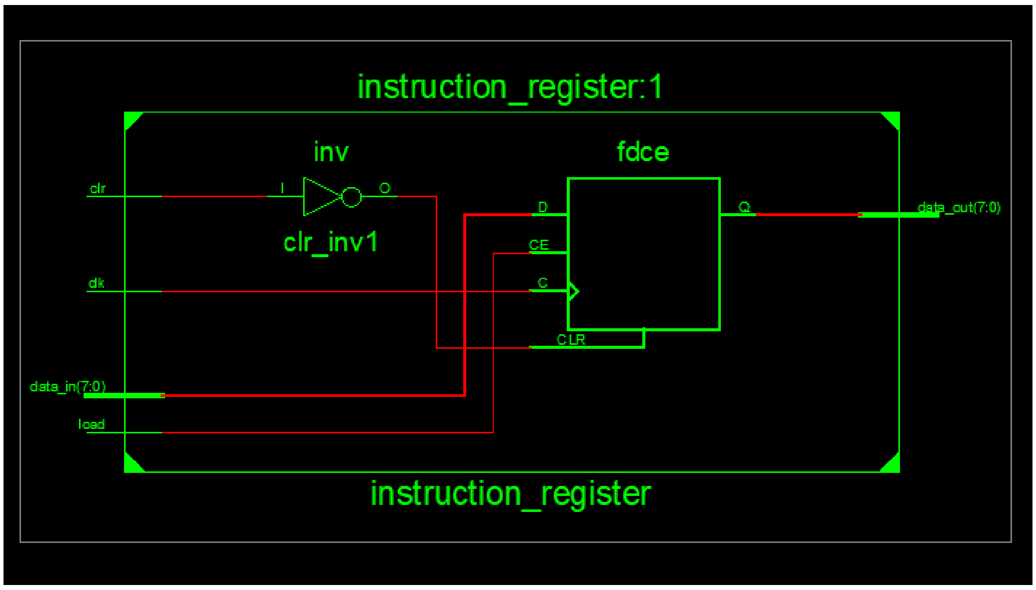
Program Counter RTL


program_counter:1
Madd_count[7]_GND_1_o_add_0_OUT1
Mmux_count[7]_data_in[7]_mux_2_OUT1
fdce
Madd_count[7]_GND_1_o_add_0_OUT1
Mmux_count[7]_data_in[7]_mux_2_OUT1
and2b2
_n00151
inv
_n0015_inv1
inv
clr_inv1
program_counter

4.0.8 Program Counter Explaination

The program counter, a Verilog module, is responsible for implementing a program counter, typically operating on a data width specified by the parameter DATAWIDTH, often set to 8 bits. Inputs include data in for setting the counter, ld pc to load new data, inc pc to increment the counter, clk for the clock signal, and clr for clearing the counter. The counter value updates on positive clock edges or negative clear edges, with the counter resetting to 0 when clr is active low. When ld pc is active, the counter loads the data in, and when inc pc is active, the counter increments by 1.

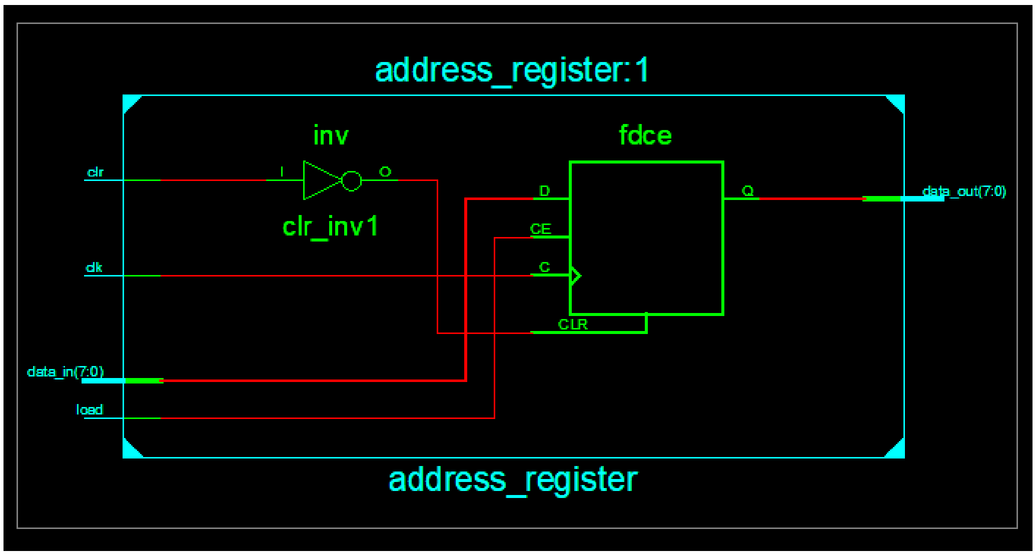
Instruction Register RTL



4.0.9 Instruction Register Explaination

The instruction register, implemented as a Verilog module, functions as a register with a supported data width specified by the parameter DATAWIDTH, often set to 8 bits. Its inputs comprise data in for setting the register, load for loading new data, clk for the clock signal, and clr for clearing the register. The register's value updates on positive clock edges or negative clear edges, and it resets to 0 when clr is active low. Upon activation of the load signal, the register loads the data in.

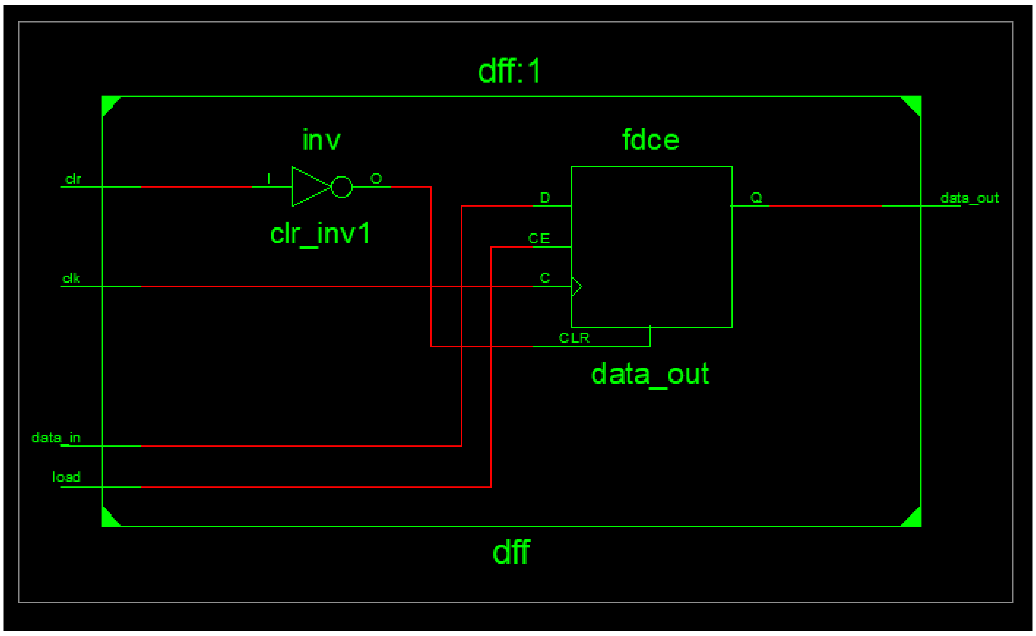
Address Register RTL



4.0.10 Address Register Explaination

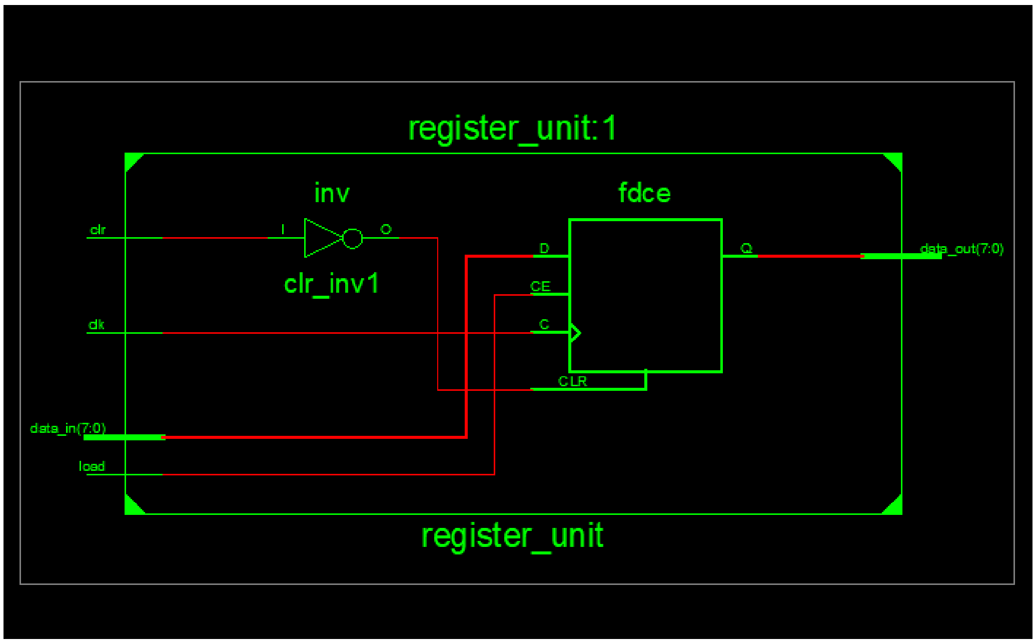
The address register, functioning as a Verilog module to implement a register, supports a data width specified by the parameter DATAWIDTH, commonly set to 8 bits. Its inputs consist of data in for setting the register, load for loading new data, clk for the clock signal, and clr for clearing the register. The register's value is updated on positive clock edges or negative clear edges, resetting to 0 when clr is active low. When load is activated, the register loads the data in.

Zero Register RTL



The dff Verilog module serves as an implementation of a D flip-flop, featuring an output labeled data out and inputs for data in, load, clk, and clr. The flip-flop's value undergoes changes during positive clock edges or negative clear edges, with the output resetting to 0 when clr is active low. Activating load results in the output being updated with the value of data in.

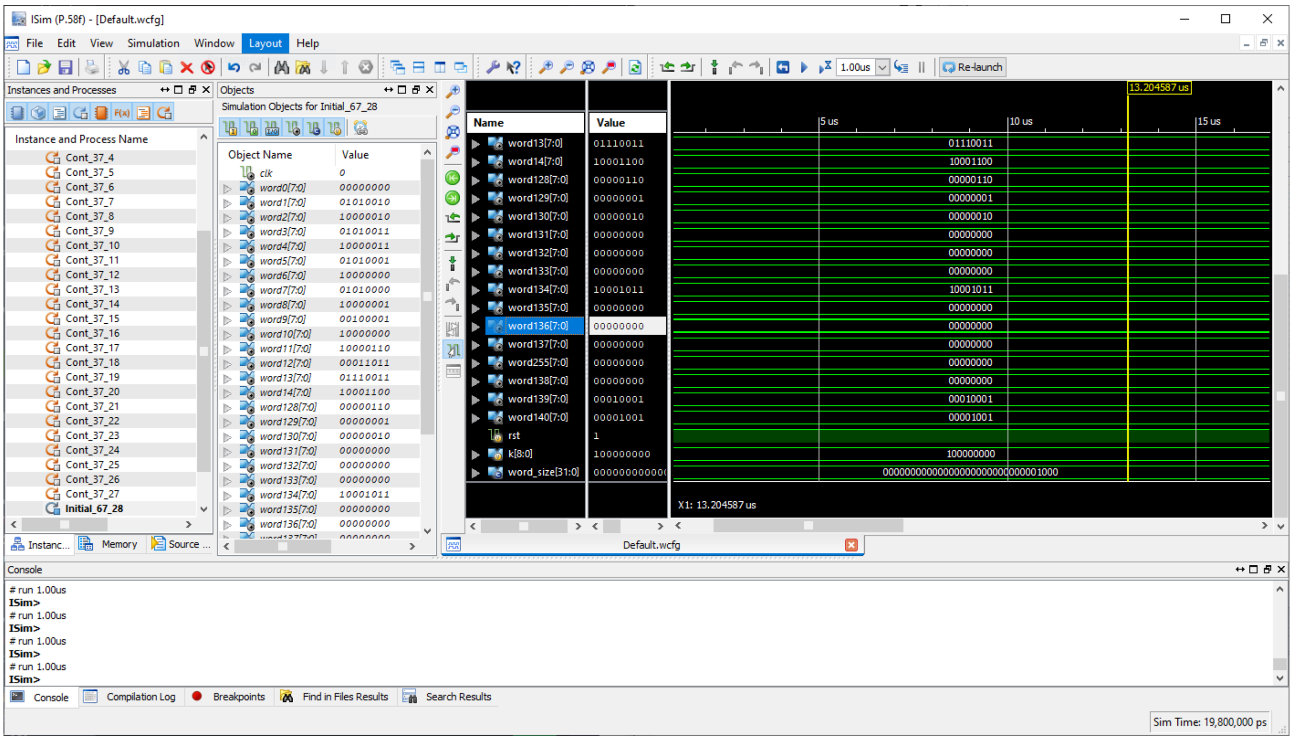
Register Unit RTL



4.0.11 Register Unit Explaination

The register unit, functioning as a Verilog module to implement a register unit, accommodates a data width specified by the parameter DATAWIDTH, often set to 8 bits. Its inputs consist of data in for setting the register, load for loading new data, clk for the clock signal, and clr for clearing the register. The register's value updates during positive clock edges or negative clear edges, resetting to 0 when clr is active low. Upon activation of load, the register loads the incoming data.

5 Test Bench



Instance and Process Name

```
Cont_37_4
Cont_37_5
Cont_37_6
Cont_37_7
Cont_37_8
Cont_37_9
Cont_37_10
Cont_37_11
Cont_37_12
Cont_37_13
Cont_37_14
Cont_37_15
Cont_37_16
Cont_37_17
Cont_37_18
Cont_37_19
Cont_37_20
Cont_37_21
Cont_37_22
Cont_37_23
Cont_37_24
Cont_37_25
Cont_37_26
Cont_37_27
Initial_67_28
```

Object Name	Value
clk	0
word0[7:0]	00000000
word1[7:0]	01010010
word2[7:0]	10000010
word3[7:0]	01010011
word4[7:0]	10000011
word5[7:0]	01010001
word6[7:0]	10000000
word7[7:0]	01010000
word8[7:0]	10000001
word9[7:0]	00100001
word10[7:0]	10000000
word11[7:0]	10000110
word12[7:0]	00011011
word13[7:0]	01110011
word14[7:0]	10001100
word128[7:0]	00000110
word129[7:0]	00000001
word130[7:0]	00000010
word131[7:0]	00000000
word132[7:0]	00000000
word133[7:0]	00000000
word134[7:0]	10001011
word135[7:0]	00000000
word136[7:0]	00000000
word137[7:0]	00000000

Name	Value	5 us / 10 us / 15 us
word13[7:0]	01110011	01110011
word14[7:0]	10001100	10001100
word128[7:0]	00000110	00000110
word129[7:0]	00000001	00000001
word130[7:0]	00000010	00000010
word131[7:0]	00000000	00000000
word132[7:0]	00000000	00000000
word133[7:0]	00000000	00000000
word134[7:0]	10001011	10001011
word135[7:0]	00000000	00000000
word136[7:0]	00000000	00000000
word137[7:0]	00000000	00000000
word255[7:0]	00000000	00000000
word138[7:0]	00000000	00000000
word139[7:0]	00010001	00010001
word140[7:0]	00001001	00001001
rst	1	
k[8:0]	100000000	100000000
word_size[31:0]	000000000000	00000000000000000000000000001000

13.204587 us

X1: 13.204587 us

```
# run 1.00us
ISim>
# run 1.00us
ISim>
# run 1.00us
ISim>
# run 1.00us
ISim>
```

6 Conclusion

In conclusion, designing an 8-bit RISC SPM processor datapath using Xilinx follows a systematic approach and requires careful consideration of various factors. Beginning with the selection of the appropriate RISC SPM ISA variant aligned with project objectives, the instruction set and processor architecture are defined.

The datapath comprises critical components such as the register file, ALU, control unit, multiplexers, and scratchpad memory. Each component needs meticulous design and implementation to ensure the smooth operation of the processor. The design process starts with implementing the instruction fetch stage, followed by instruction decoding, register access, ALU operations, and control signal generation.

Thorough testing is essential to validate the correctness of the RISC SPM processor datapath, with various test cases ensuring accurate instruction execution. Once the design is tested and validated, synthesis using Xilinx tools produces a gate-level netlist for implementation on the target FPGA device.

Performance analysis plays a crucial role in evaluating the efficiency and effectiveness of the RISC SPM processor datapath, with room for optimizations to enhance performance. System-level integration, including interfacing with peripherals and external memory, is vital for comprehensive project implementation.

Throughout the design process, adherence to RISC SPM architecture specifications, good design practices, and leveraging Xilinx design tools and resources are essential for efficient implementation. Detailed guidance and support from relevant textbooks, documentation, and experienced designers enrich the design process, leading to the development of a robust and functional 8-bit RISC SPM processor datapath.

7 References

[1] Christine Connolly. “Pros from touchics”. In: *Industrial Robot: An International Journal* (2008).

[2] Michael H Dickinson. “insight into mechanical design”. In: *Proceedings of the National Academy of Sciences* 96.25 (1999), pp. 14208–14209.

[3] Robert Rosen. “revisited”. In: *The machine as metaphor and tool.* Springer, 1993, pp. 87–100.

[4] René Romain Roth. “The foundation of nics”. In: *Perspectives in bicine* 26.2 (1983), pp. 229–242.