RISC – V ISA Processor Pipelined Implementation using Verilog

Done by Naman Kalra, B Tech Second Year Student, Under Prof. Vikramkumar puri Sir

RISC-V ISA Processor Implementation Thesis

Made by Naman Kalra, EE23B032

Project Part 2

Pipelined Implementation -> Pipeline Hazards Detection -> Solutions of Detected Hazards

❖ **Pipelined implementation of a processor is essential for improving its performance and efficiency. Here's why pipelining is important:**

1. **Increased Throughput**: Pipelining allows multiple instructions to be processed simultaneously at different stages of execution. Instead of completing one instruction before starting the next, the processor can work on several instructions at once, each at a different stage of completion. This increases the number of instructions completed per unit of time, thus improving the overall throughput of the processor.
2. **Reduced Latency**: In a non-pipelined processor, each instruction has to be executed sequentially from start to finish. Pipelining breaks this process into stages, such as fetching, decoding, executing, and writing back. By overlapping these stages for multiple instructions, the time required to complete an instruction can be reduced, effectively lowering the latency for each instruction.
3. **Better Utilization of Resources**: Pipelining allows different parts of the processor (like the ALU, memory access units, and registers) to be used more efficiently. While one instruction is being executed, other instructions can be in the stages of fetching or decoding, keeping various components of the processor busy and minimizing idle time.
4. **Higher Clock Speeds**: Pipelining can enable higher clock speeds because each stage in the pipeline can be designed to take less time, allowing the processor to clock at higher frequencies. Each stage does less work compared to a non-pipelined design, which can make it easier to achieve higher clock rates.

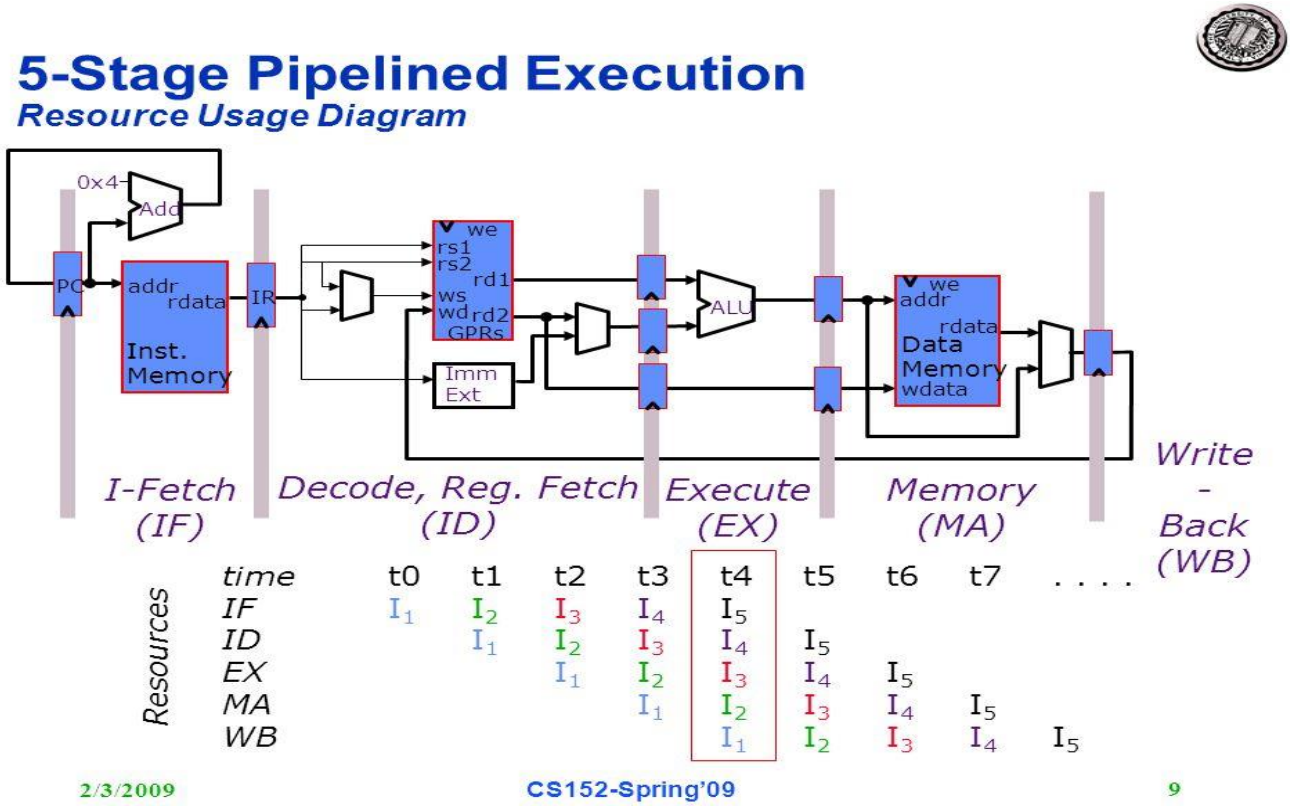


Fig 1.1 – Pictorial Representation of pipelined Processor

❖ **In a register-based pipeline processor, the concept revolves around dividing instruction execution into multiple stages, each handling a specific task and connected by registers.**

1. **Stages**: The pipeline typically consists of stages like Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM), and Write Back (WB). Each stage performs a distinct part of the instruction process.
2. **Registers**: Between these stages, registers (pipeline registers) temporarily hold data and intermediate results, ensuring smooth data flow and maintaining instruction continuity.
3. **Parallelism**: Multiple instructions are processed simultaneously at different stages, increasing overall throughput and efficiency

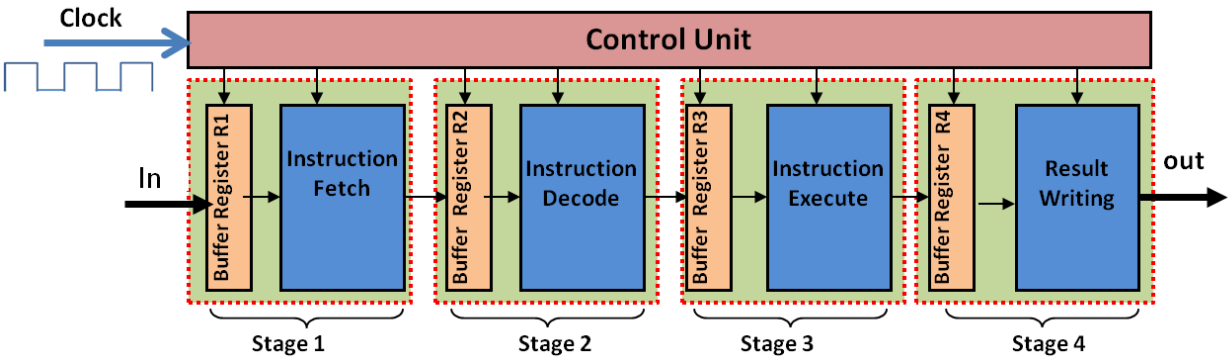


Fig 1.2 – Register Stages

1. Sequential Logic

Sequential logic involves circuits where the output depends on the current inputs and the previous state, which is stored in memory elements like flip-flops. It relies on clock signals to synchronize changes in state.

- **Pipeline Stages**: In a RISC processor, the processing of instructions is typically divided into multiple stages (like Instruction Fetch, Instruction Decode, Execution, Memory Access, and Write Back). Each of these stages involves sequential logic, as the state of the processor is updated on clock edges.
- **Registers**: The processor's registers (like the Program Counter, pipeline registers, and general-purpose registers) are sequential elements that store data and control information across clock cycles.
- **Clocked Operation**: The overall operation of a RISC processor is synchronized with a clock. Instructions move through different stages of the pipeline, and each stage is clocked, meaning it updates and processes data on the rising or falling edge of the clock signal.

2. Combinational Logic

Combinational logic refers to circuits where the output is purely a function of the current inputs, without any memory or state. The output changes immediately with changes in input.

- **ALU Operations**: The Arithmetic Logic Unit (ALU) in a RISC processor performs combinational logic operations. For example, adding two numbers, performing logical operations, or comparing values are done in combinational logic.
- **Instruction Decoding**: The instruction decode process involves combinational logic to extract opcode and operands from the instruction and generate control signals based on the instruction's type.

3. Synchronized Clock

Synchronized clock refers to the use of a clock signal to synchronize various elements of the processor, ensuring that all parts of the processor operate in a coordinated manner.

- **Pipeline Synchronization**: In a pipelined RISC processor, different stages of the pipeline are synchronized with a common clock. This ensures that data moves smoothly from one stage to the next, with each stage updating its state on the clock edge.
- **Clocked Sequential Logic**: The registers, memory, and pipeline stages are all synchronized with the clock. This means that updates to these elements occur at specific intervals dictated by the clock signal.

Summary

- **RISC ISA** itself is not inherently combinational or sequential; rather, it is a specification that defines the instructions and their formats. The implementation of a RISC processor based on this ISA will use both combinational and sequential logic to execute instructions.
- **Sequential Logic**: Used for managing state and data flow through the pipeline stages of the processor, with operations synchronized by the clock.
- **Combinational Logic**: Used for immediate calculations and operations, like those performed by the ALU, without regard to clock cycles.
- **Synchronized Clock**: The entire processor operation, including both sequential and combinational elements, is synchronized by a clock signal to ensure coordinated operation and data integrity.

Let's begin ……..

First of all, we will define each block and intermediate registers separately and then join all of them together.

❖ Instruction Fetch IF Stage -

It contains various blocks such as PC counter, Adder for PC increment, Instruction Memory, IFID Pipeline Register, Pipeline Flush and Hazard detection unit. Brief explanation is mentioned below -

1. **Program Counter (PC)**:
   - **Role**: Holds the address of the next instruction to be fetched.
   - **Operation**: Updates with the address of the next instruction, either sequentially or based on branch/jump operations.
2. **Adder for PC Increment**:
   - **Role**: Computes the address of the next sequential instruction.
   - **Operation**: Adds a constant value (typically 4) to the current PC value to generate the address of the next instruction.
3. **Instruction Memory**:
   - **Role**: Stores the program instructions.
   - **Operation**: Retrieves and outputs the instruction located at the address specified by the PC.
4. **IFID Pipeline Register**:
   - **Role**: Stores the fetched instruction and the PC value.
   - **Operation**: Latches the instruction and PC value from the fetch stage to be passed to the decode stage.
5. **Pipeline Flush**:
   - **Role**: Clears or invalidates instructions in the pipeline if a branch or jump instruction alters the control flow.
   - **Operation**: Ensures that incorrect instructions are removed from the pipeline when a control hazard is detected.
6. **Hazard Detection Unit**:
   - **Role**: Identifies potential data hazards and determines if stalling is needed.
   - **Operation**: Monitors for hazards that may require delaying instruction processing to avoid incorrect execution.

These modules work together to ensure that instructions are accurately fetched and properly handled in a pipelined processor.

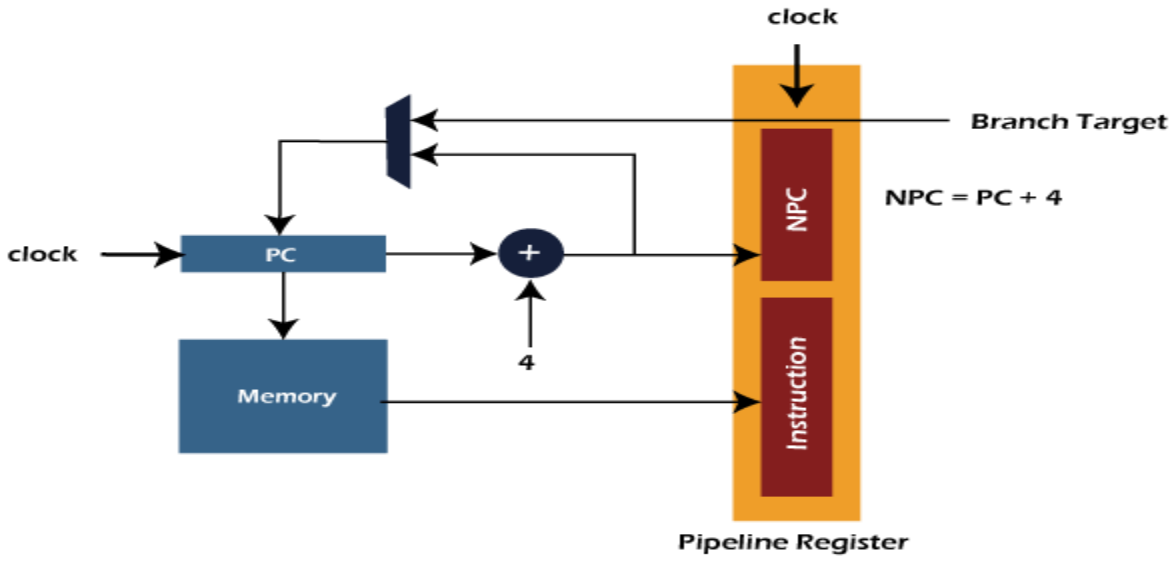


Fig 1.3 - Circuit Overview of Instruction Fetch Unit

❖ Program Counter Module

- **Verilog Code –**

```
module program_counter(
  input [63:0] PC_in,
  input clk,
  input reset,
  input stall,
  output reg [63:0] PC_out);

  always @(posedge clk or posedge reset)
    begin
      if (reset==1'b1)
        begin
          PC_out = 64'd0;
        end
      else if (stall == 1'b0)
        begin
          PC_out = PC_in;
        end
    end
endmodule
```

- **RTL Design of PC –**

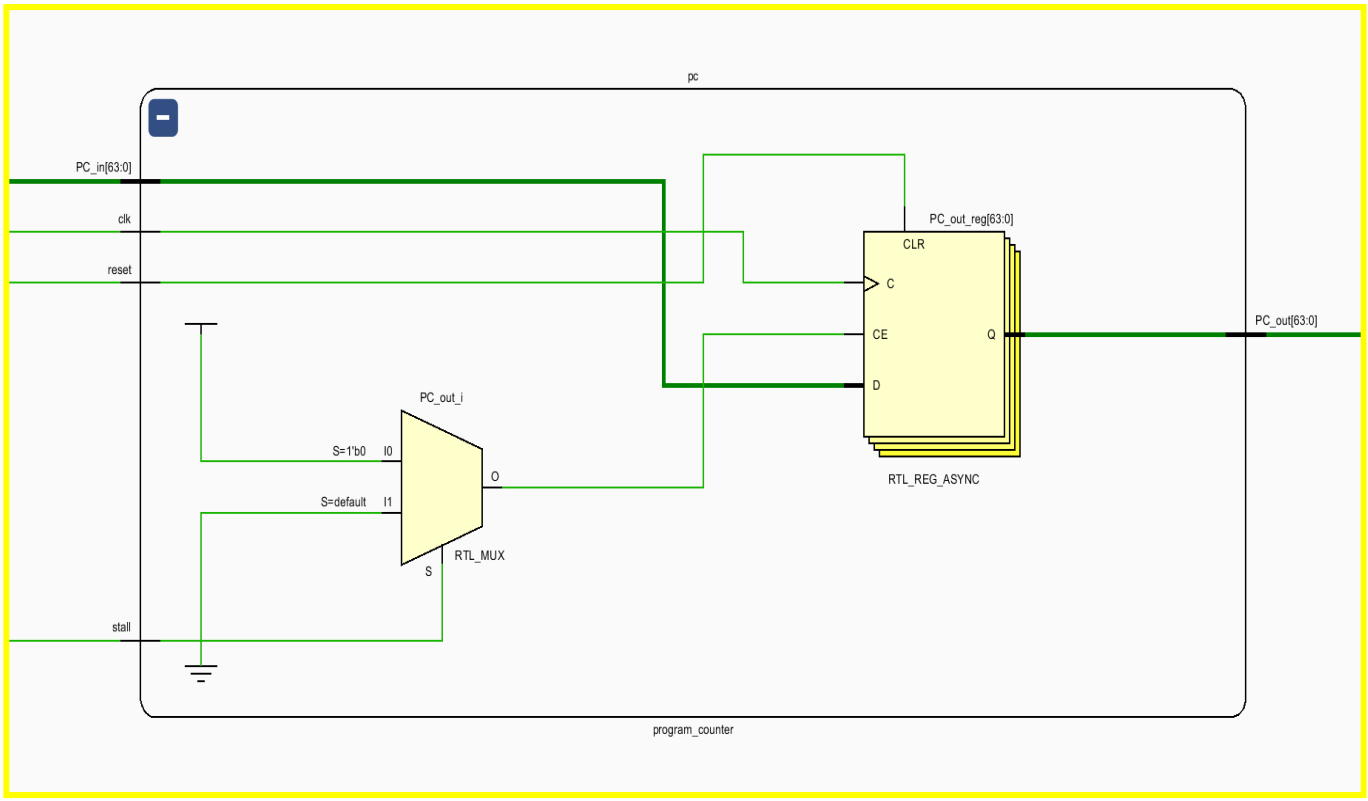


Fig 1.4 – RTL Elaborated Design of PC module

❖ Adder for PC Increment

- **Verilog Code –**

```
module adder (
  input  [63:0] p,      // Current PC value
  input  [63:0] q,      // Value to be added (typically 4 for sequential instructions)
  output [63:0] out     // Result of the addition
);

  assign out = p + q;   // Perform the addition
endmodule
```

- **RTL Design of adder –**

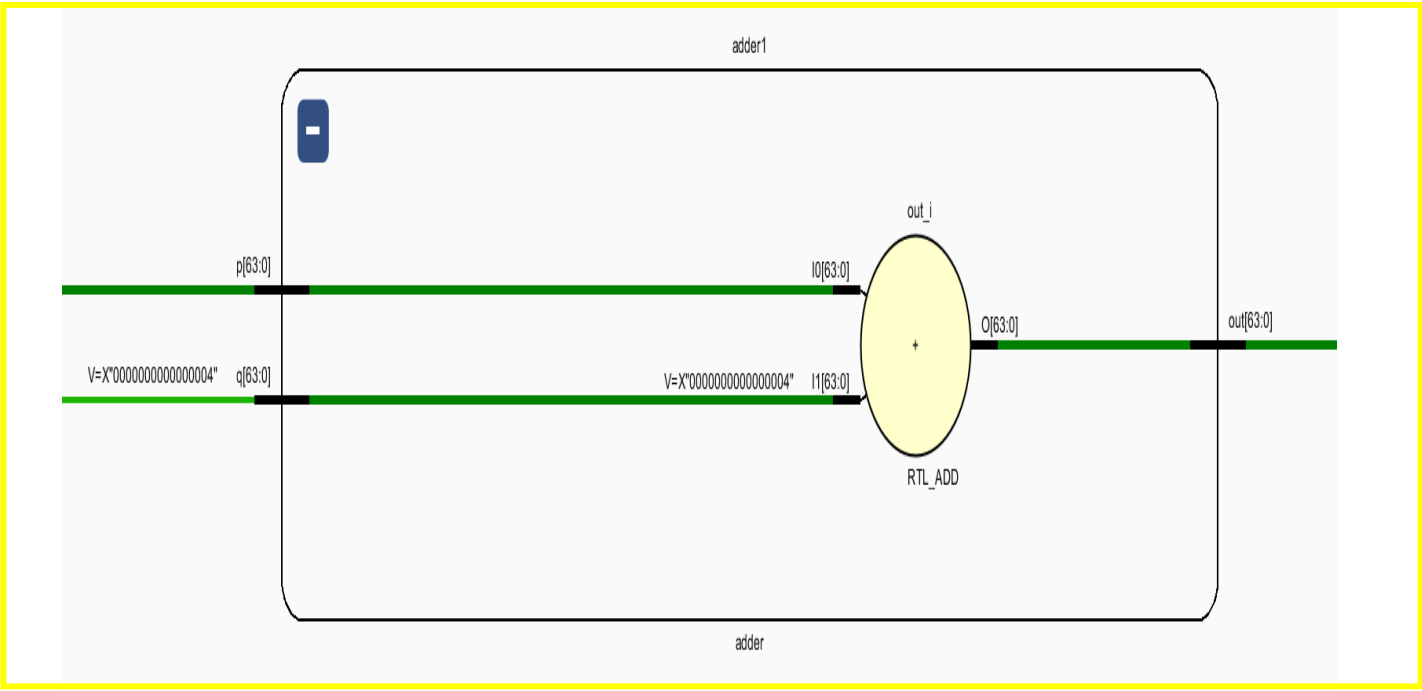


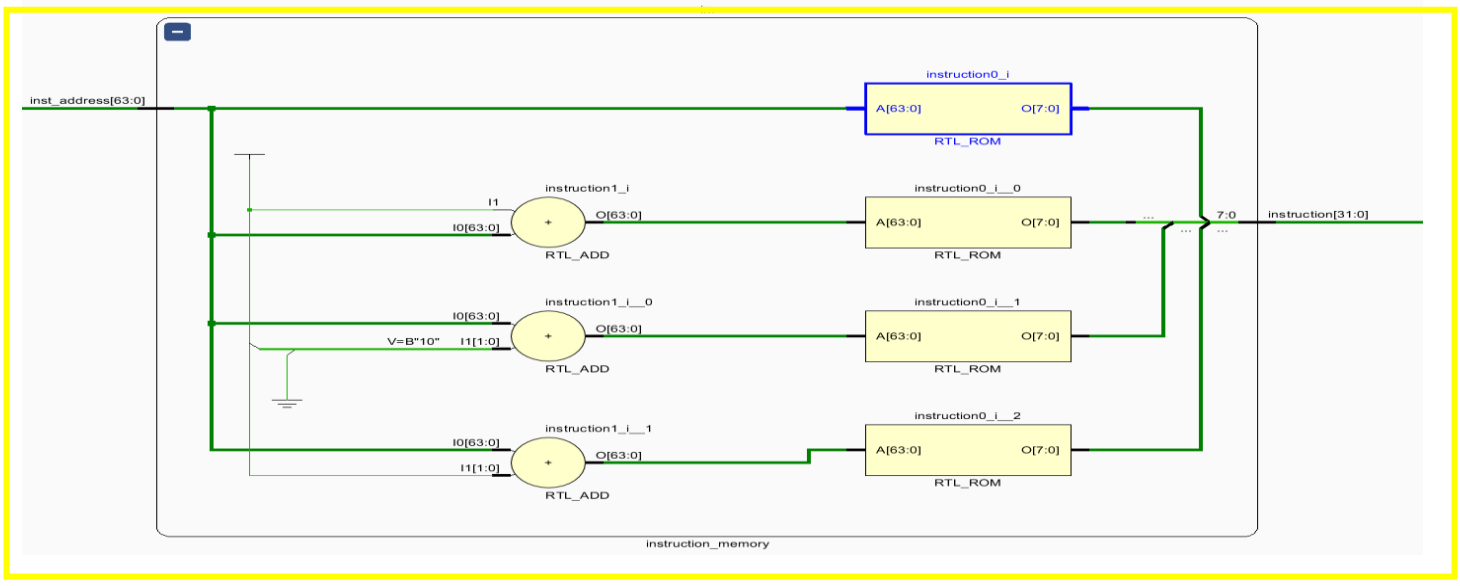
Fig 1.5 - RTL Design of adder

❖ Instruction Memory

- **Verilog Code –**

```
module instruction_memory(
  input [63:0] inst_address,
  output reg [31:0] instruction);
  reg [7:0] inst_mem[87:0];
  initial
    begin
      {inst_mem[3], inst_mem[2], inst_mem[1], inst_mem[0]} = 32'h00000913;//1
      {inst_mem[7], inst_mem[6], inst_mem[5], inst_mem[4]} = 32'h00000433;//2
      {inst_mem[11], inst_mem[10], inst_mem[9], inst_mem[8]} = 32'h04b40863;//3
      {inst_mem[15], inst_mem[14], inst_mem[13], inst_mem[12]} = 32'h00800eb3;//4
      {inst_mem[19], inst_mem[18], inst_mem[17], inst_mem[16]} = 32'h000409b3;//5
```

```
      {inst_mem[23], inst_mem[22], inst_mem[21], inst_mem[20]} =
32'h013989b3;//6
      {inst_mem[27], inst_mem[26], inst_mem[25], inst_mem[24]} =
32'h013989b3;//7
      {inst_mem[31], inst_mem[30], inst_mem[29], inst_mem[28]} =
32'h013989b3;//8
      {inst_mem[35], inst_mem[34], inst_mem[33], inst_mem[32]} =
32'h02be8663;//9
      {inst_mem[39], inst_mem[38], inst_mem[37], inst_mem[36]} =
32'h001e8e93;//10
      {inst_mem[43], inst_mem[42], inst_mem[41], inst_mem[40]} =
32'h00898993;//11
      {inst_mem[47], inst_mem[46], inst_mem[45], inst_mem[44]} =
32'h00093d03;//12
      {inst_mem[51], inst_mem[50], inst_mem[49], inst_mem[48]} =
32'h0009bd83;//13
      {inst_mem[55], inst_mem[54], inst_mem[53], inst_mem[52]} =
32'h01bd4463;//14
      {inst_mem[59], inst_mem[58], inst_mem[57], inst_mem[56]} =
32'hfe0004e3;//15
      {inst_mem[63], inst_mem[62], inst_mem[61], inst_mem[60]} =
32'h01a002b3;//16
      {inst_mem[67], inst_mem[66], inst_mem[65], inst_mem[64]} =
32'h01b93023;//17
      {inst_mem[71], inst_mem[70], inst_mem[69], inst_mem[68]} =
32'h0059b023;//18
      {inst_mem[75], inst_mem[74], inst_mem[73], inst_mem[72]} =
32'hfc000ce3;//19
      {inst_mem[79], inst_mem[78], inst_mem[77], inst_mem[76]} =
32'h00140413;//20
      {inst_mem[83], inst_mem[82], inst_mem[81], inst_mem[80]} =
32'h00890913;//21
      {inst_mem[87], inst_mem[86], inst_mem[85], inst_mem[84]} =
32'hfa000ae3;//22
    end
  always @ (inst_address)
    begin
      instruction[7:0] = inst_mem[inst_address+0];
      instruction[15:8] = inst_mem[inst_address+1];
      instruction[23:16] = inst_mem[inst_address+2];
      instruction[31:24] = inst_mem[inst_address+3];
    end
endmodule
```



❖ IFID Pipeline Register

- **Verilog Code –**

```
module IFID
  (
    input clk,
    input reset,
    input [31:0] instruction,
    input [63:0] A, //a
    input flush,
    input IFIDWrite,
    output reg [31:0] inst,//instruction out,
    output reg [63:0] a_out
  );
  always @(posedge clk)
    begin
      if (reset == 1'b1 || flush == 1'b1)
        begin
          inst = 32'b0;
          a_out = 64'b0;
        end
      else if (IFIDWrite == 1'b0)
        begin
          inst = instruction;
          a_out = A;
        end
    end
endmodule
```

- **RTL Design of IFIF Register**

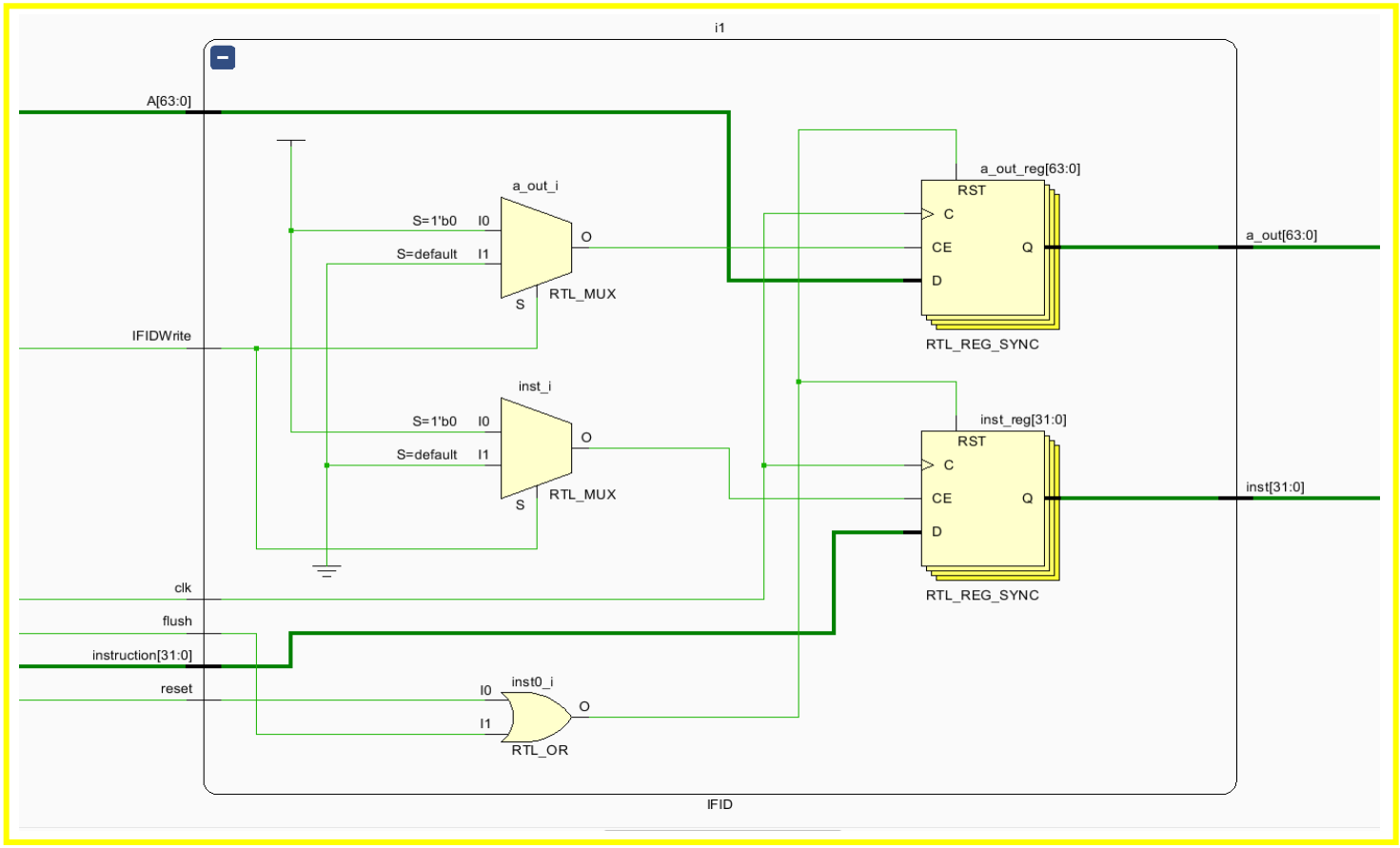


Fig 1.6 - RTL Design of IFIF Register

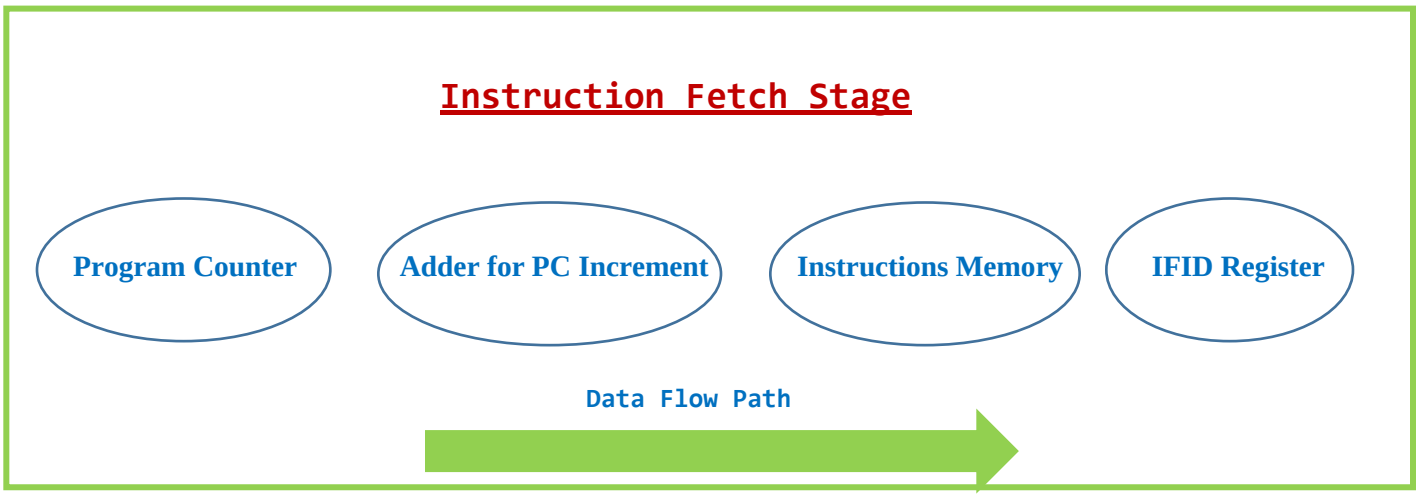


Fig 1.7 – Data flow diagram for IF Stage

❖ Code Explanation

1. Program Counter Module

Purpose: The `program_counter` module maintains the current program counter (PC) value, which indicates the address of the next instruction to be fetched.

- **Inputs:**
  - `PC_in`: New value to be loaded into the PC.
  - `clk`: Clock signal to synchronize updates.
  - `reset`: Signal to reset the PC to 0.
  - `stall`: Signal to indicate whether the PC should be updated or not.
- **Output:**
  - `PC_out`: Current value of the PC.

Behavior:

- On a positive edge of the clock or a positive edge of the reset signal:
  - If `reset` is high, `PC_out` is set to 0.
  - If `stall` is low, `PC_out` is updated with `PC_in`.

2. Adder for PC Increment

Purpose: The `adder` module adds two 64-bit values. Typically, this is used to increment the program counter to fetch the next instruction.

- **Inputs:**
  - `p`: Current PC value.
  - `q`: Value to be added (usually 4 for sequential instructions).
- **Output:**
  - `out`: Result of adding `p` and `q`.

Behavior:

- It performs a simple addition: `out = p + q`.

3. Instruction Memory

Purpose: The **`instruction_memory`** module holds the instruction set and provides instructions based on the address input.

- **Inputs:**
  - **`inst_address`:** Address to fetch the instruction from.
- **Output:**
  - **`instruction`:** 32-bit instruction fetched from memory.

Behavior:

- The memory is initialized with predefined instructions.
- When `inst_address` changes, the corresponding 32-bit instruction is output.

4. IFID Pipeline Register

Purpose: The `IFID` pipeline register stores the instruction fetched and the current PC value for use in the next stage of the pipeline (Instruction Fetch to Instruction Decode).

- **Inputs:**
  - `clk`: Clock signal for synchronization.
  - `reset`: Signal to reset the register values.
  - `instruction`: Instruction fetched from memory.
  - `A`: Current PC value.
  - `flush`: Signal to discard current values and reset the register.
  - `IFIDWrite`: Signal to control whether to write to the register or not.
- **Outputs:**
  - `inst`: Stored instruction.
  - `a_out`: Stored PC value.

Behavior:

- On the rising edge of the clock:
  - If `reset` or `flush` is high, the register is reset to 0.
  - If `IFIDWrite` is low, the `instruction` and `A` values are stored in the register.

❖ **Simulation Results RTL Design in Xilinx Software**

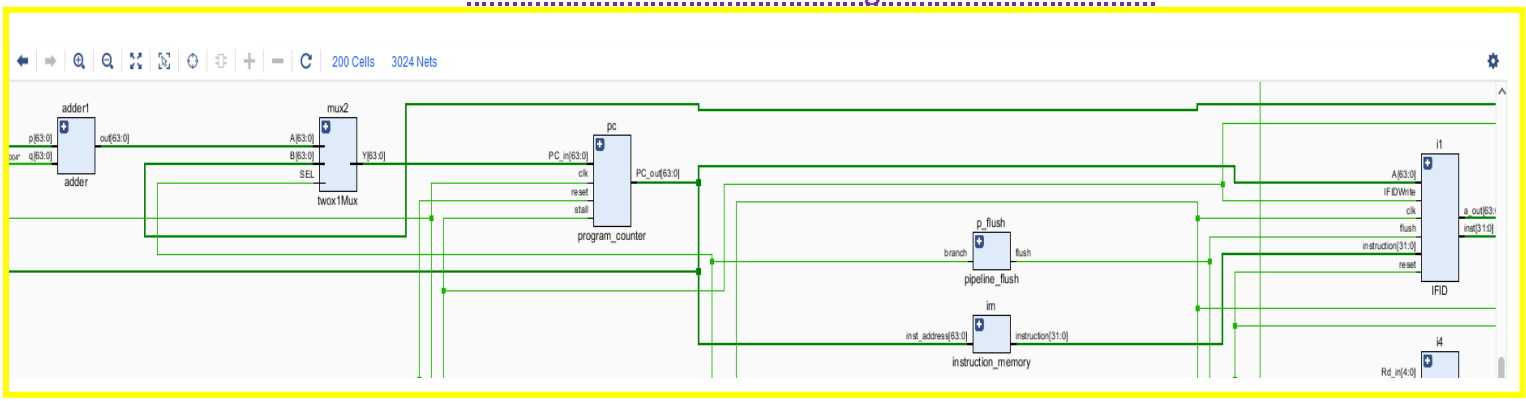


Fig 1.8 – Register Transfer Level Design of IF unit

❖ **Instruction Decode Stage:** In the Instruction Decode (ID) stage of a RISC-V processor, several key blocks and modules are used to decode the fetched instruction, extract relevant fields, and prepare necessary control signals for execution.

1. Instruction Parser (Instruction Decoder)

- **Function:** Extracts fields from the fetched instruction.
- **Inputs:** 32-bit instruction.
- **Outputs:**
  - Opcode (7 bits)
  - Function codes (funct3, funct7)
  - Source registers (rs1, rs2)
  - Destination register (rd)
  - Immediate value (if applicable)

2. Control Unit (CU)

- **Function:** Generates control signals based on the opcode and function codes extracted from the instruction.
- **Inputs:**
  - Opcode
  - Function codes (funct3, funct7)
- **Outputs:**
  - Control signals (e.g., branch, memread, memtoreg, memwrite, regwrite, ALUsrc, ALUop)

3. Immediate Data Extractor

- **Function:** Extracts and extends immediate values from the instruction.
- **Inputs:** 32-bit instruction.
- **Outputs:** 64-bit extended immediate value.

4. Register File

- **Function:** Provides read and write access to the general-purpose registers.
- **Inputs:**
  - Read register addresses (rs1, rs2)
  - Write register address (rd)
  - Write data
  - Write enable signal
- **Outputs:**
  - Read data from registers (readdata1, readdata2)

5. Forwarding Unit

- **Function:** Determines whether the data needed by the current instruction should be forwarded from previous stages rather than fetched from the register file.
- **Inputs:**
  - Register addresses for read operations
  - Register addresses for write operations
- **Outputs:**
  - Forwarding control signals

6. Hazard Detection Unit

- **Function:** Detects and handles hazards (e.g., data hazards) that may occur due to instruction dependencies.
- **Inputs:**
  - Information about instructions (e.g., register addresses, control signals)
- **Outputs:**
  - Stall signals
  - Hazard resolution signals

7. Branching Unit

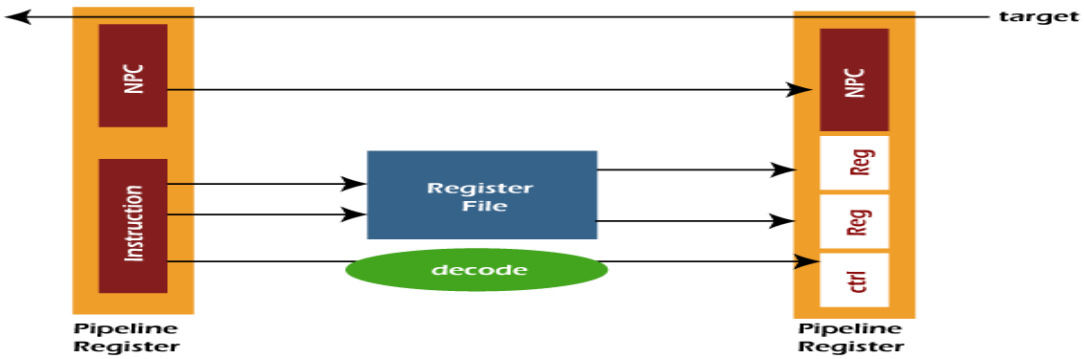
- **Function:** Computes branch targets and determines whether a branch should be taken.
- **Inputs:**
  - Branch condition signals (e.g., result of comparison)
  - Immediate value (for branch offset)
- **Outputs:**
  - Branch target address
  - Branch decision

8. Pipeline Registers (e.g., IFID, IDEX)

- **Function:** Store intermediate data between pipeline stages to maintain state and ensure correct operation of the pipeline.
- **Inputs:**
  - Data from the previous stage (e.g., fetched instruction, program counter value)
- **Outputs:**
  - Data passed to the next stage (e.g., instruction and PC for decode stage)

Summary :-The Instruction Decode (ID) stage is crucial for interpreting instructions and preparing them for execution. The key components involved are:

- **Instruction Parser:** Extracts instruction fields.
- **Control Unit:** Generates control signals.
- **Immediate Data Extractor:** Handles immediate values.
- **Register File:** Provides access to registers.
- **Forwarding Unit:** Handles data forwarding.
- **Hazard Detection Unit:** Manages hazards.
- **Branching Unit:** Computes branch decisions and targets.
- **Pipeline Registers:** Maintain state across pipeline stages



❖ Instruction Parser Block

- **Verilog Code :-**

```
module instruction_parser(
  input [31:0] instruction,
  output [6:0] opcode,
  output [4:0] rd,
  output [4:0] rs1,
  output [4:0] rs2,
  output [2:0] funct3,
  output [6:0] funct7
);
  assign opcode = instruction[6:0];
  assign rd = instruction[11:7];
  assign rs1 = instruction[19:15];
  assign rs2 = instruction[24:20];
  assign funct3 = instruction[14:12];
  assign funct7 = instruction[31:25];
endmodule
```

➢ RTL Design of Parser Block :-

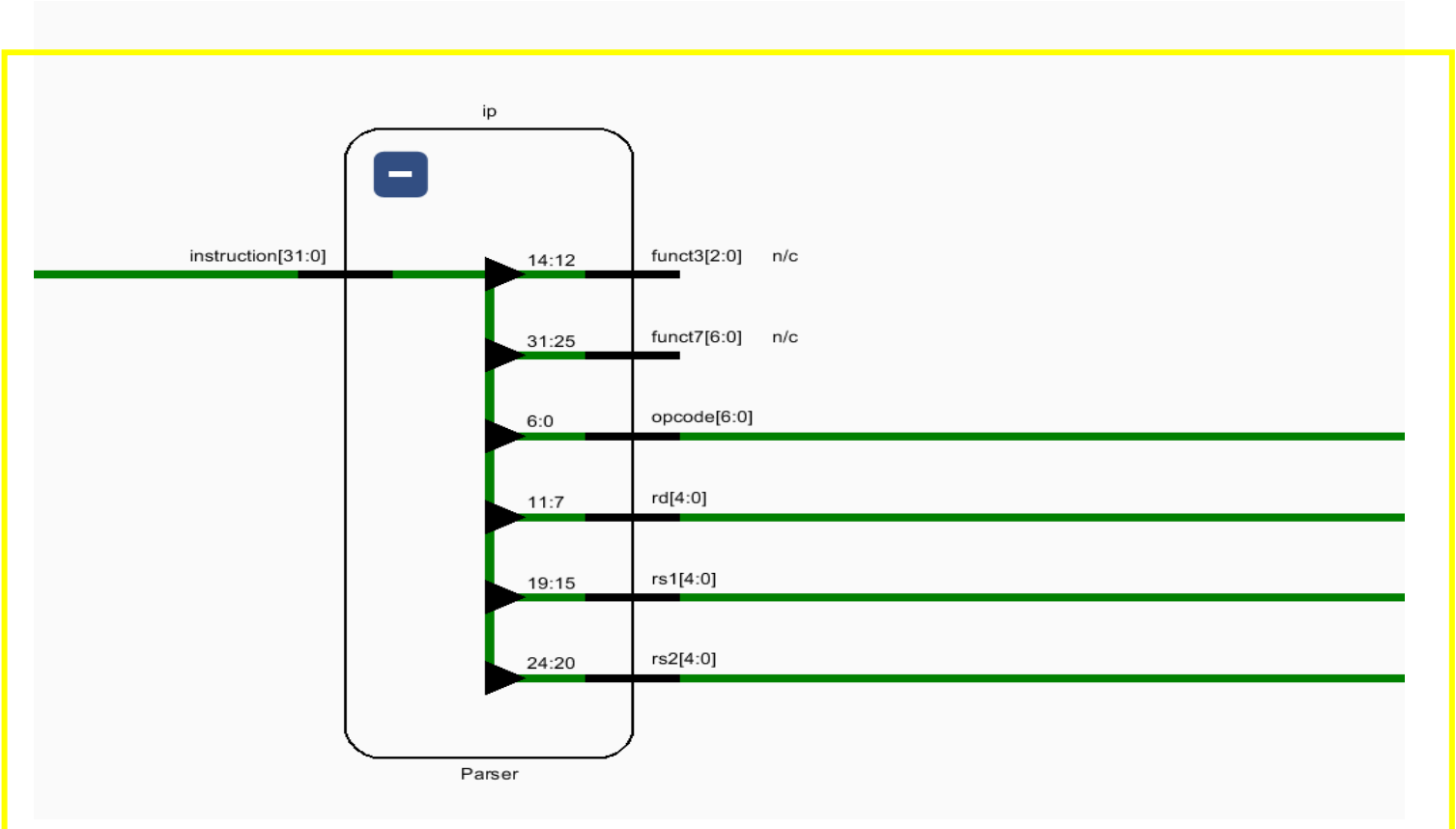


Fig 1.9 - RTL Design of Parser Block

❖ Control Unit

- **Verilog Code :-**

```
module CU(
  input [6:0] opcode,
  input stall,
  output reg branch,
  output reg memread,
  output reg memtoreg,
  output reg memwrite,
  output reg aluSrc,
  output reg regwrite,
  output reg [1:0] Aluop
);
  always @(*)
    begin
      // Default values
      aluSrc = 1'b0;
      memtoreg = 1'b0;
      regwrite = 1'b0;
      memread = 1'b0;
      memwrite = 1'b0;
      branch = 1'b0;
      Aluop = 2'b00;

      if (stall) begin
        // If stall is active, override all control signals to halt the pipeline
        aluSrc = 1'b0;
        memtoreg = 1'b0;
        regwrite = 1'b0;
        memread = 1'b0;
        memwrite = 1'b0;
        branch = 1'b0;
        Aluop = 2'b00;
      end else begin
        case (opcode)
          7'b0000011: begin // Load instructions
            aluSrc = 1'b1;
            memtoreg = 1'b1;
            regwrite = 1'b1;
            memread = 1'b1;
            memwrite = 1'b0;
            branch = 1'b0;
            Aluop = 2'b00;
          end
          7'b0100011: begin // Store instructions
            aluSrc = 1'b1;
            memtoreg = 1'bx; // Don't care
            regwrite = 1'b0;
            memread = 1'b0;
            memwrite = 1'b1;
            branch = 1'b0;
            Aluop = 2'b00;
          end
          7'b0110011: begin // R-type instructions
            aluSrc = 1'b0;
            memtoreg = 1'b0;
            regwrite = 1'b1;
            memread = 1'b0;
            memwrite = 1'b0;
```

```
              branch = 1'b0;
              Aluop = 2'b10;
            end
            7'b1100011: begin // Branch instructions
              aluSrc = 1'b0;
              memtoreg = 1'bx; // Don't care
              regwrite = 1'b0;
              memread = 1'b0;
              memwrite = 1'b0;
              branch = 1'b1;
              Aluop = 2'b01;
            end
            7'b0010011: begin // I-type instructions
              aluSrc = 1'b1;
              memtoreg = 1'b0;
              regwrite = 1'b1;
              memread = 1'b0;
              memwrite = 1'b0;
              branch = 1'b0;
              Aluop = 2'b00;
            end
            default: begin
              // Default case to handle unknown opcodes
              aluSrc = 1'b0;
              memtoreg = 1'b0;
              regwrite = 1'b0;
              memread = 1'b0;
              memwrite = 1'b0;
              branch = 1'b0;
              Aluop = 2'b00;
            end
          endcase
        end
      end
  endmodule
```

- RTL Design of Control Unit

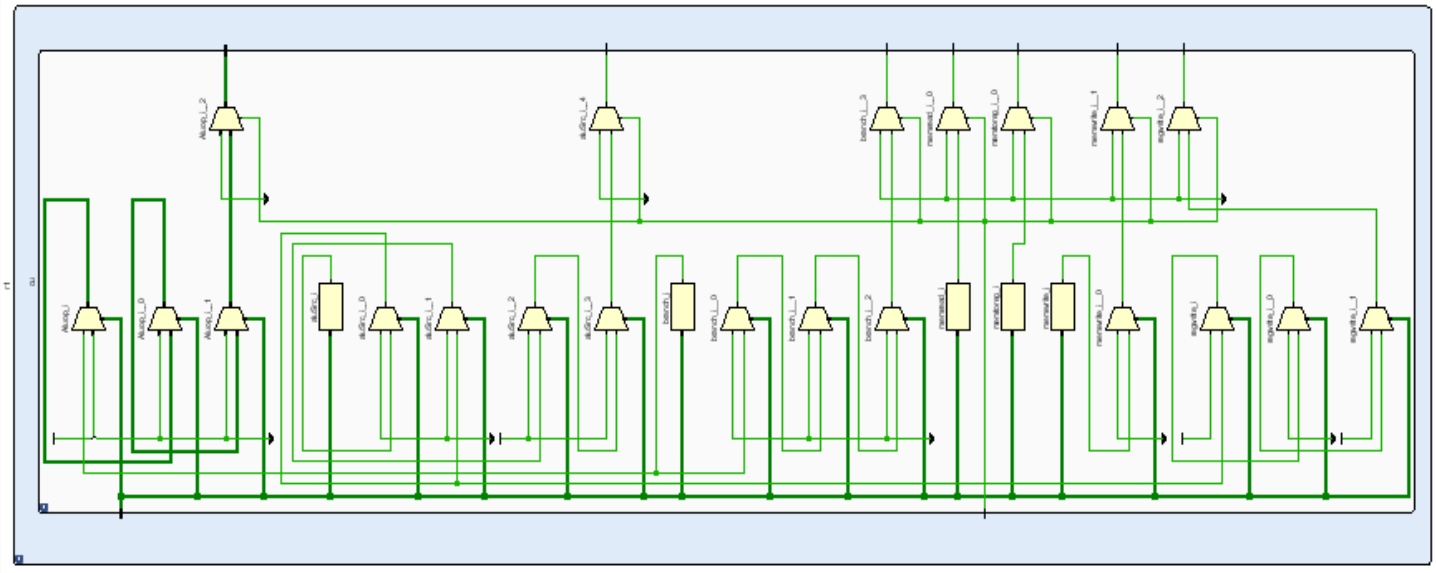


Fig 2.0 – RTL Design of Control Unit

❖ Immediate Data Extractor

- **Verilog Code :-**

```
module data_extractor
(
    input [31:0] instruction,
    output reg [63:0] imm_data
);
  always @(*)
  begin
    // Default value
    imm_data = 64'b0;
    case (instruction[6:5])
      2'b00: // I-type format (Immediate value in bits [31:20])
        begin
          imm_data[11:0] = instruction[31:20];
        end
      2'b01: // S-type format (Immediate value from bits [31:25] and
[11:7])
        begin
          imm_data[11:0] = {instruction[31:25], instruction[11:7]};
        end
      2'b11: // B-type format (Immediate value from bits [31], [7],
[30:25], [11:8])
        begin
          imm_data[11:0] = {instruction[31], instruction[7],
instruction[30:25], instruction[11:8]};
        end
      default: // Handle unexpected cases
        begin
          imm_data = 64'b0; // or some error handling
        end
    endcase
    // Sign-extend immediate data to 64 bits
    imm_data = {{52{imm_data[11]}}, imm_data[11:0]};
  end
endmodule
```

- RTL Design for Immediate Extractor :-

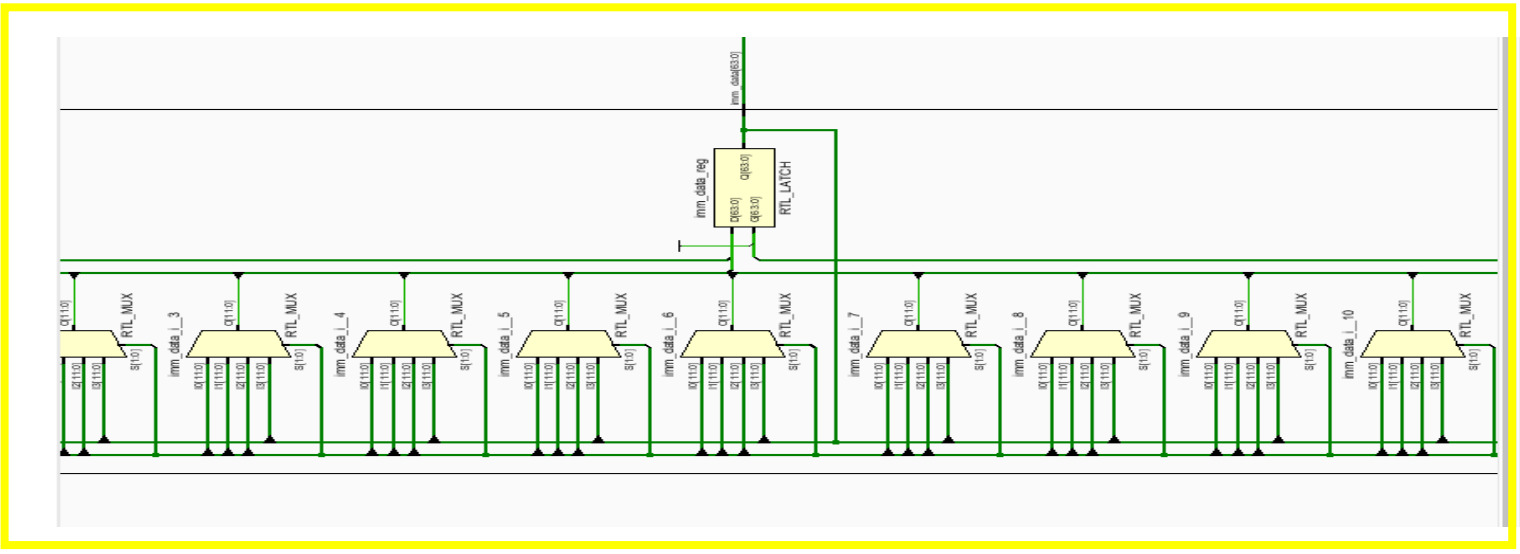


Fig 2.1 – RTL Design for IMM_Extractor

❖ Register File

- **Verilog Code :-**

```
module registerFile(
  input clk,
  input reset,
  input [4:0] rs1,
  input [4:0] rs2,
  input [4:0] rd,
  input [63:0] writedata,
  input reg_write,
  output reg [63:0] readdata1,
  output reg[63:0] readdata2,
  output [63:0] r8,
  output [63:0] r19,
  output [63:0] r20,
  output [63:0] r21,
  output [63:0] r22
);
  integer i;
  reg [63:0] registers [31:0];
  initial
    begin
      registers[0] = 64'd0;
      registers[1] = 64'd0;
      registers[2] = 64'd0;
      registers[3] = 64'd0;
      registers[4] = 64'd0;
      registers[5] = 64'd0;
      registers[6] = 64'd0;
      registers[7] = 64'd0;
      registers[8] = 64'd0;
      registers[9] = 64'd0;
      registers[10] = 64'd0;
      registers[11] = 64'd8;
      registers[12] = 64'd0;
      registers[13] = 64'd0;
      registers[14] = 64'd0;
      registers[15] = 64'd0;
      registers[16] = 64'd0;
      registers[17] = 64'd0;
      registers[18] = 64'd0;
      registers[19] = 64'd0;
      registers[20] = 64'd0;
      registers[21] = 64'd0;
      registers[22] = 64'd0;
      registers[23] = 64'd0;
      registers[24] = 64'd0;
      registers[25] = 64'd0;
      registers[26] = 64'd0;
      registers[27] = 64'd0;
      registers[28] = 64'd0;
      registers[29] = 64'd0;
      registers[30] = 64'd0;
      registers[31] = 64'd0;
    end
  assign r8 = registers[8];
  assign r19 = registers[19];
  assign r20 = registers[20];
```

```
    assign r21 = registers[26];
    assign r22 = registers[27];

    always @ (*)
      begin
        if (reset == 1'b1)
          begin
            readdata1 = 64'd0;
            readdata2 = 64'd0;
          end
        else
          begin
            readdata1 = registers[rs1];
            readdata2 = registers[rs2];
          end
      end
    always@(negedge clk)
      begin
        if (reg_write == 1)
          registers[rd] = writedata;
      end
  endmodule
```

❖ Branching Unit

- **Verilog Code :-**

```
  module branching_unit
    (
     input [2:0] funct3,
      input [63:0] readData1,
      input [63:0] b,
     output reg addermuxselect);
    initial
      begin
        addermuxselect = 1'b0;
      end
    always @(*)
    begin
        case (funct3)
          3'b000:
            begin
              if (readData1 == b)
                addermuxselect = 1'b1;
              else
                addermuxselect = 1'b0;
              end
           3'b100:
                 begin
                if (readData1 < b)
                addermuxselect = 1'b1;
              else
                addermuxselect = 1'b0;
              end
          3'b101:
            begin
              if (readData1 > b)
                  addermuxselect = 1'b1;
             else
                addermuxselect = 1'b0;
            end
        endcase
```

```
        end
endmodule
```

- **RTL Design of Branch Module**

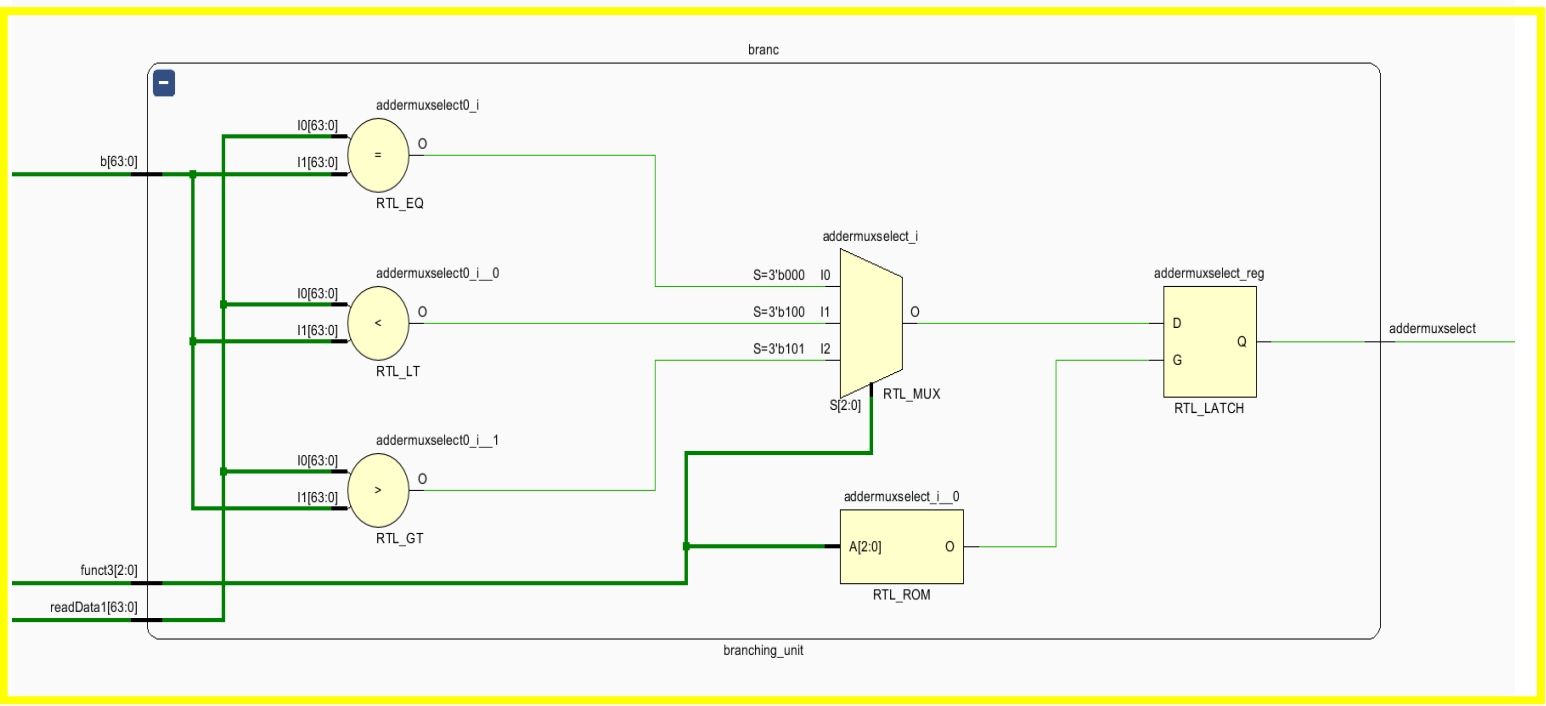


Fig 2.2 – Branch Module

❖ IDEX Pipeline Register

- **Verilog Code :-**

```
module IDEX(
input clk,reset,
input [3:0] funct4_in,//funct4 of instruction from instruction memory
input [63:0] A_in,//adder input, ouput of IFID carried forward
input [63:0] readdata1_in, //from regwrite
input [63:0] readdata2_in,//from regwrite
input [63:0] imm_data_in,//from data extractor
input [4:0] rs1_in,//from instruction parser
input [4:0] rs2_in, //from instruction parser
input [4:0] rd_in, //from instruction parser
input branch_in,memread_in,memtoreg_in,memwrite_in,aluSrc_in,regwrite_in,
//from control unit
input [1:0] Aluop_in,
input flush,
output reg [63:0] a,
output reg [4:0] rs1,
output reg [4:0] rs2,
output reg [4:0] rd,
output reg [63:0] imm_data,
output reg [63:0] readdata1, //2bit mux
output reg [63:0] readdata2, //2bit mux
output reg [3:0] funct4_out,
output reg Branch,Memread,Memtoreg, Memwrite, Regwrite,Alusrc,
output reg [1:0] aluop);
always @ (posedge clk)
begin
if (reset == 1'b1 || flush == 1'b1)
```

```
begin
a = 64'b0;
rs1 = 5'b0;
rs2 = 5'b0;
rd = 5'b0;
imm_data = 64'b0;
readdata1 = 64'b0;
readdata2 = 64'b0;
funct4_out = 4'b0;
Branch = 1'b0;
Memread = 1'b0;
Memtoreg =1'b0;
Memwrite = 1'b0;
Regwrite = 1'b0;
Alusrc = 1'b0;
aluop = 2'b0;
end
else
begin
a = A_in;
rs1 = rs1_in;
rs2 = rs2_in;
rd = rd_in;
imm_data = imm_data_in;
readdata1 = readdata1_in;
readdata2 = readdata2_in;
funct4_out = funct4_in;
Branch = branch_in;
Memread = memread_in;
Memtoreg = memtoreg_in;
Memwrite = memwrite_in;
Regwrite = regwrite_in;
Alusrc = aluSrc_in;
aluop = Aluop_in;
end
end
endmodule
```

- **RTL Design for ID/IE Register –**

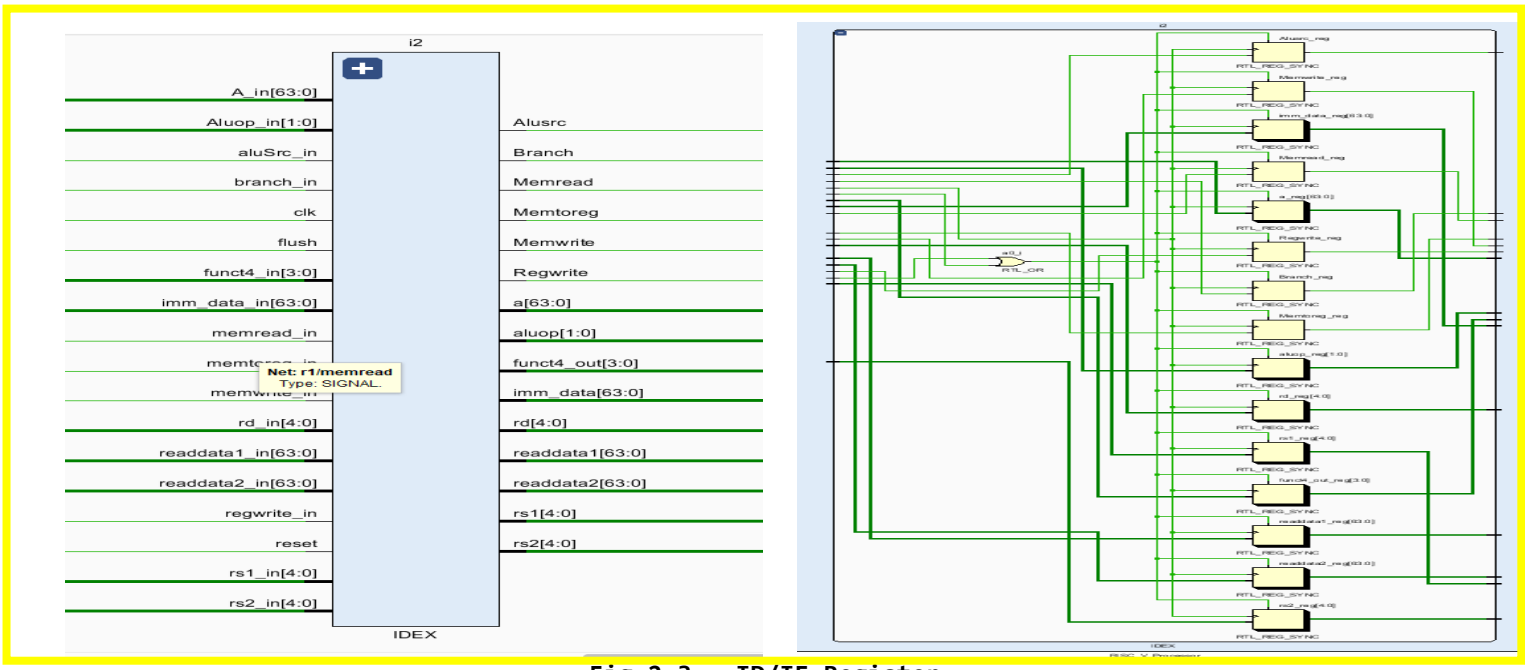


Fig 2.3 – ID/IE Register

❖ Codes Explanation –

- **Instruction Parser**:

  - **Purpose**: Extracts specific fields from a 32-bit instruction.
  - **Outputs**:
    - `opcode`: Instruction operation code.
    - `rd`: Destination register.
    - `rs1`, `rs2`: Source registers.
    - `funct3`: Function code for ALU operations.
    - `funct7`: Additional function code for specific instructions.

- **Control Unit (CU)**:

  - **Purpose**: Generates control signals based on the `opcode` from the instruction and a `stall` signal.
  - **Outputs**:
    - `branch`: Indicates if a branch instruction is present.
    - `memread`: Signal to read from memory.
    - `memtoreg`: Signal to write memory data to a register.
    - `memwrite`: Signal to write data to memory.
    - `aluSrc`: Selects the second operand for the ALU.
    - `regwrite`: Signal to write data to a register.
    - `Aluop`: ALU operation code to determine the type of ALU operation.

- **Immediate Data Extractor**:

  - **Purpose**: Extracts and sign-extends immediate data from the instruction based on its format.
  - **Outputs**:
    - `imm_data`: Sign-extended immediate value extracted from the instruction.

- **Register File**:

  - **Purpose**: Holds 32 registers and provides read and write access.
  - **Inputs**:
    - `rs1`, `rs2`: Source registers for read operations.
    - `rd`: Destination register for write operations.
    - `writedata`: Data to be written to the register.
    - `reg_write`: Control signal to enable writing.
  - **Outputs**:
    - `readdata1`, `readdata2`: Values read from registers `rs1` and `rs2`.
    - `r8`, `r19`, `r20`, `r21`, `r22`: Specific registers for external access.

- **Branching Unit**:

  - **Purpose**: Determines if a branch condition is met based on `funct3` and input values.
  - **Outputs**:
    - `addermuxselect`: Control signal indicating whether to select the branch address.

- **IDEX Pipeline Register**:

- **Purpose**: Stores values and control signals between the Instruction Decode and Execute stages.
- **Inputs**:
  - Data and control signals from the ID stage.
  - `flush`: Clears the register on pipeline flush.
- **Outputs**:
  - Data and control signals forwarded to the Execute stage

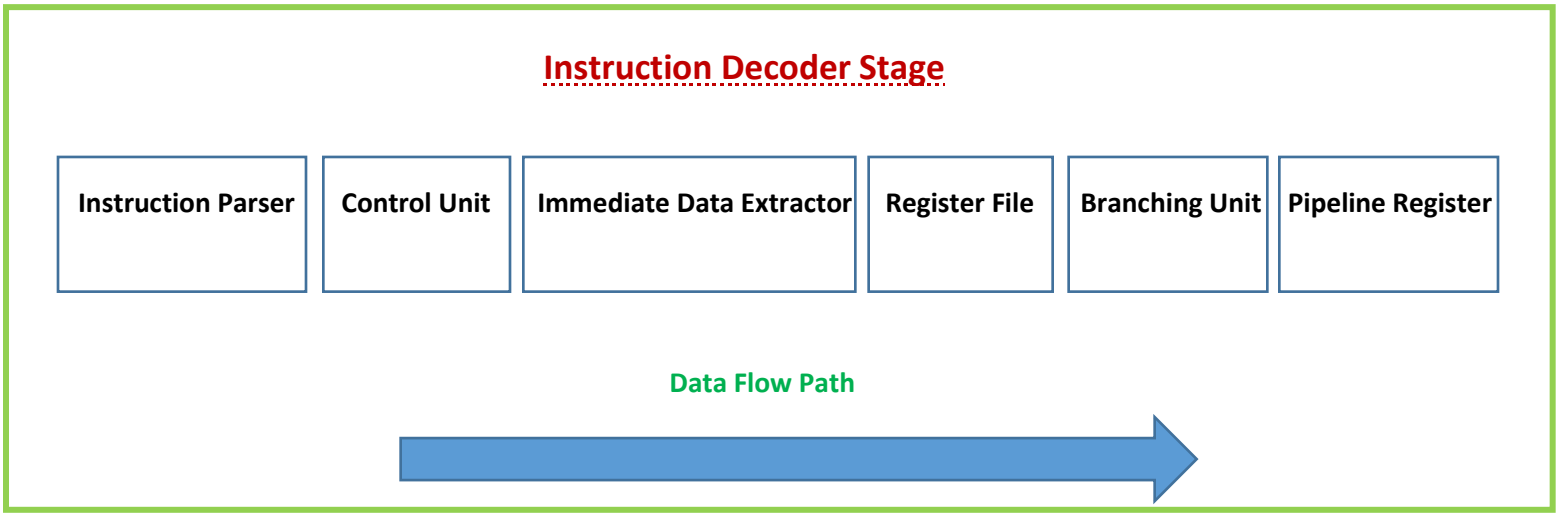


Fig 2.4 – Data Flow diagram for Instruction Decode Stage

- **Execution stage – contains different small blocks which work as a combine unit to give results.**

1. Forwarding Unit

Purpose: Resolves data hazards by forwarding results from one pipeline stage to another, ensuring instructions dependent on prior results receive the correct data without unnecessary delays.

Components:

- **Inputs:**
  - Results from the EXMEM stage (write-back result and register).
  - Results from the MEMWB stage (write-back result and register).
- **Outputs:**
  - Forwarding control signals to the ALU inputs for selecting between forwarded data and original data.

Operation: The forwarding unit ensures that if an instruction in the IDEX stage depends on results from the EXMEM or MEMWB stage, the correct data is used. This avoids stalls by directly forwarding the necessary data to the execution stage.

2. ALU Control

Purpose: Generates control signals that direct the ALU on what operation to perform based on the instruction's opcode and function codes.

Components:

- **Inputs:**
  - ALU operation code from the control unit.
  - Function field from the instruction decoder.
- **Outputs:**
  - ALU operation code, which determines the specific operation the ALU should perform.

Operation: The ALU Control unit converts high-level operation codes and function fields into specific ALU control signals, ensuring the ALU performs the correct arithmetic or logical operation.

3. ALU (Arithmetic Logic Unit)

Purpose: Performs arithmetic and logical operations on the data inputs based on control signals provided by the ALU Control unit.

Components:

- **Inputs:**
  - Operand a (result from the forwarding unit or register file).
  - Operand b (immediate value or register data, based on ALU source control).
  - ALU operation code.

- **Outputs:**
  - ALU result.
  - Zero flag indicating if the result is zero.

Operation: The ALU executes the specified operation on its inputs and provides the result along with status flags (e.g., zero flag) for use in subsequent stages.

4. Branching Unit

Purpose: Handles branch instructions by computing the branch target address and determining whether the branch should be taken based on the outcome of the ALU operation.

Components:

- **Inputs:**
  - Computed address from the EXMEM stage.
  - Branch condition flags and inputs from the ALU.
  - Branch control signals.
- **Outputs:**
  - Branch decision.
  - Target address for the branch if taken.

Operation: The Branching Unit calculates the branch target address and decides if the current instruction sequence should be altered based on branch conditions.

5. Three-by-One Mux (for data forwarding)

Purpose: Selects the correct data to be forwarded to the ALU from multiple sources (e.g., the output of the register file, results from the EXMEM stage, and results from the MEMWB stage).

Components:

- **Inputs:**
  - Data from the register file.
  - Data from the EXMEM stage.
  - Data from the MEMWB stage.
- **Outputs:**
  - Selected data for the ALU.

Operation: This multiplexer selects the correct operand based on forwarding control signals to ensure the ALU receives the correct data.

6. Two-by-One Mux (for ALU input selection)

Purpose: Selects between the immediate value and a register value to be used as the second operand for the ALU.

Components:

- **Inputs:**
  - Immediate value extracted from the instruction.
  - Data from the register file.

- **Outputs:**
  - Selected ALU operand.

Operation: This multiplexer selects the appropriate operand for the ALU based on whether the instruction uses an immediate value or a register value.

7. Adder (for branch target calculation)

Purpose: Computes the target address for branch instructions by adding the offset to the current program counter value.

Components:

- **Inputs:**
  - Current PC value.
  - Offset from the immediate value shifted left.
- **Outputs:**
  - Branch target address.

Operation: The adder calculates the address to which the PC should jump if a branch is taken, ensuring the correct address is used to fetch the next instruction.

8. EXMEM Pipeline Register

Purpose: Stores intermediate results and control signals between the execution stage and the memory access stage.

Components:

- **Inputs:**
  - ALU result.
  - Branch target address.
  - Control signals (e.g., MemRead, MemWrite, branch decision).
- **Outputs:**
  - Data to be written to memory.
  - Address for memory operations.
  - Control signals for memory access.

Operation: This register holds the results and control signals from the execution stage, making them available for the memory access stage.

Summary

The execution stage of a CPU pipeline involves:

- **Forwarding Unit:** Resolves data hazards by forwarding results to the ALU.
- **ALU Control:** Generates control signals for ALU operations.
- **ALU:** Performs arithmetic or logical operations.
- **Branching Unit:** Computes branch targets and decisions.
- **Three-by-One Mux:** Forwards data from multiple sources to the ALU.
- **Two-by-One Mux:** Selects between immediate values and register values for the ALU.
- **Adder:** Calculates branch target addresses.

- **EXMEM Pipeline Register:** Stores intermediate results and control signals for the memory access stage.

➢ In Depth Explanation of each block of Pipelined RISK V ISA Processor

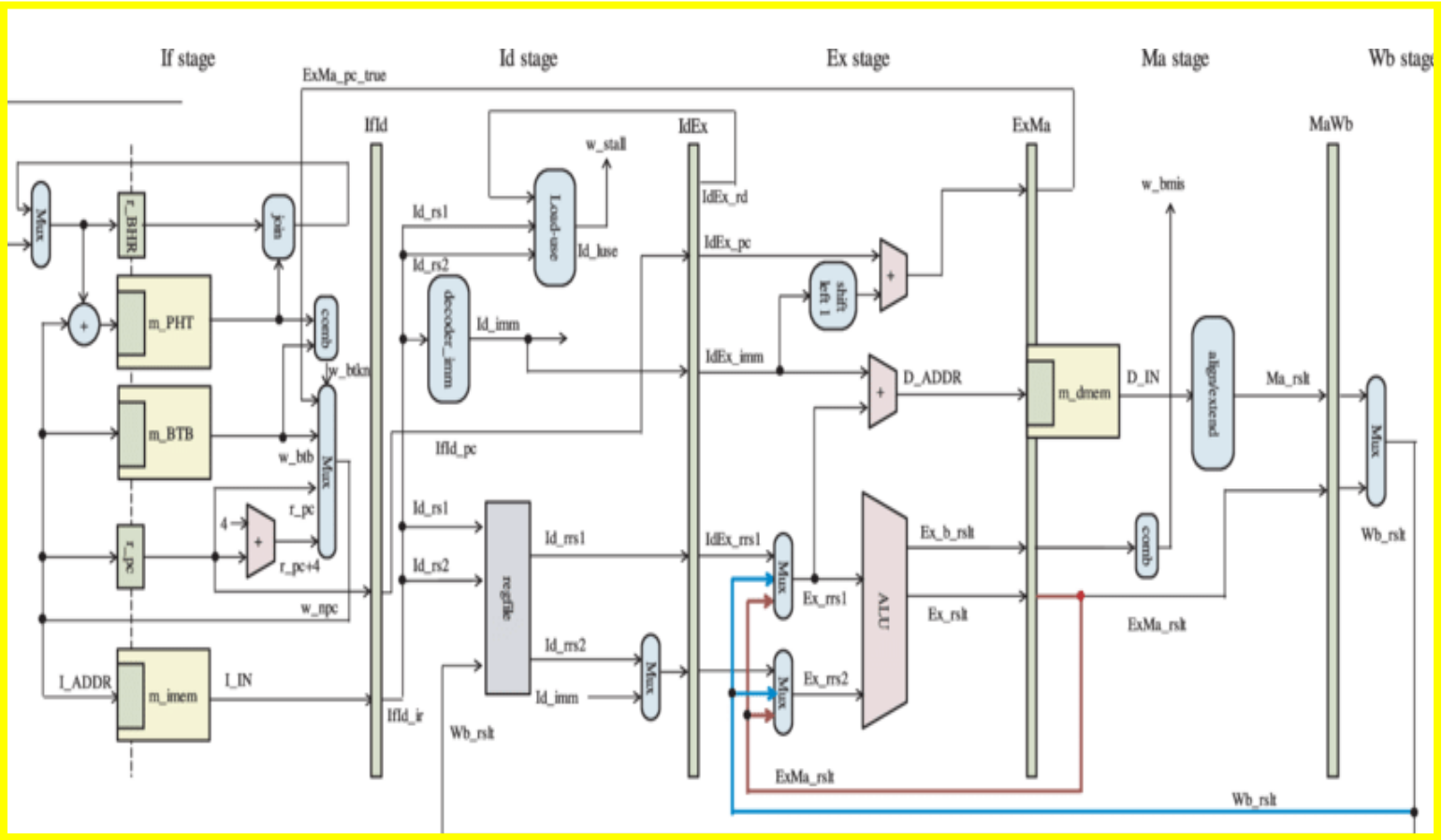


Fig 2.5 – Detailed Circuit Representation of 5 stage pipelined RISC-V ISA Processor

❖ Forwarding Unit

- **Verilog Code –**

```
module ForwardingUnit
  (
    input [4:0] RS_1, //ID/EX.RegisterRs1
    input [4:0] RS_2, //ID/EX.RegisterRs2
    input [4:0] rdMem, //EX/MEM.Register Rd
    input [4:0] rdWb, //MEM/WB.RegisterRd

    input regWrite_Wb, //MEM/WB.RegWrite
    input regWrite_Mem, // EX/MEM.RegWrite
    output reg [1:0] Forward_A,
    output reg [1:0] Forward_B
 );

  always @(*)
    begin
         if ( (rdMem == RS_1) & (regWrite_Mem != 0 & rdMem !=0))
          begin
               Forward_A = 2'b10;
          end
         else
          begin
            // Not condition for MEM hazard
            if ((rdWb== RS_1) & (regWrite_Wb != 0 & rdWb != 0) & ~((rdMem == RS_1) &(regWrite_Mem != 0 & rdMem !=0)  )  )
              begin
                Forward_A = 2'b01;
              end
            else
              begin
                Forward_A = 2'b00;
              end
          end

        if ( (rdMem == RS_2) & (regWrite_Mem != 0 & rdMem !=0) )
          begin
            Forward_B = 2'b10;
          end
        else
          begin
            // Not condition for MEM Hazard
            if ( (rdWb == RS_2) & (regWrite_Wb != 0 & rdWb != 0) & ~((regWrite_Mem != 0 & rdMem !=0 ) & (rdMem == RS_2) ) )
              begin
                Forward_B = 2'b01;
              end
            else
              begin
                Forward_B = 2'b00;
              end
          end
```

- `end`
- `endmodule`

➢ RTL Design of Forwarding Unit :-

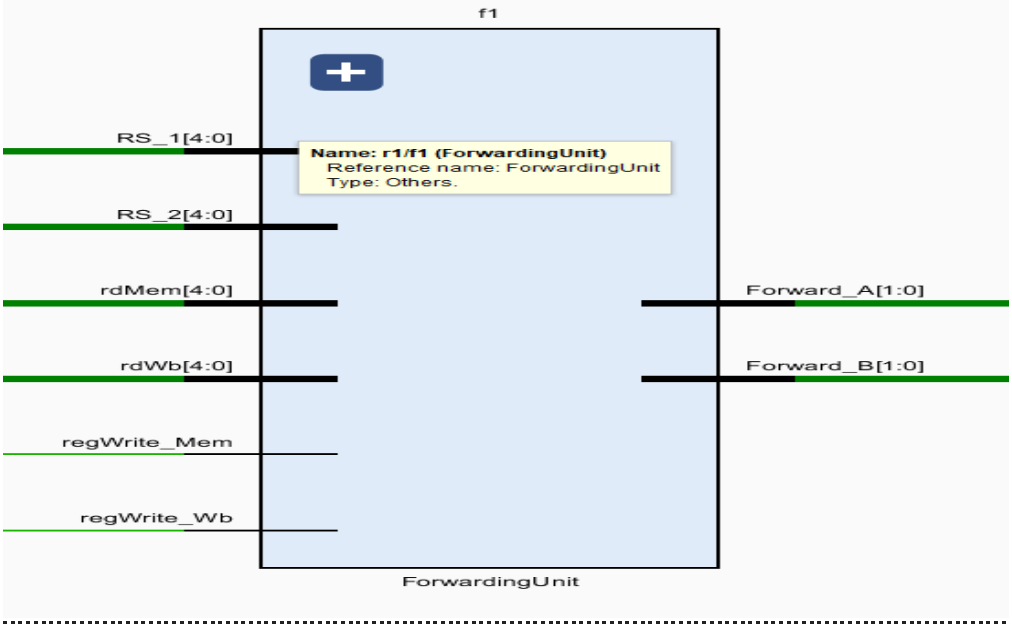


Fig 2.6 – Forwarding module

- **Code Explanation –**

Purpose

The `ForwardingUnit` is designed to handle data hazards in a pipelined processor. Data hazards occur when an instruction depends on the result of a previous instruction that has not yet completed. To avoid pipeline stalls and ensure smooth execution, the `ForwardingUnit` decides if the result of a previous instruction should be forwarded to the current instruction.

Inputs

1. **RS_1:** Source register 1 from the `ID/EX` stage (the instruction currently being decoded).
2. **RS_2:** Source register 2 from the `ID/EX` stage.
3. **rdMem:** Destination register from the `EX/MEM` stage (where the result of an arithmetic operation is stored before writing to the register file).
4. **rdWb:** Destination register from the `MEM/WB` stage (where the result of a memory read or ALU operation is written back to the register file).
5. **regWrite_Wb:** Register write enable signal from the `MEM/WB` stage, indicating whether the `MEM/WB` stage is performing a write operation.
6. **regWrite_Mem:** Register write enable signal from the `EX/MEM` stage, indicating whether the `EX/MEM` stage is performing a write operation.

Outputs

1. **Forward_A:** Control signal for forwarding data to source register 1 (RS_1).

2. **Forward_B:** Control signal for forwarding data to source register 2 (RS_2).

Forwarding Logic

For Source Register 1 (`RS_1`):

- **Forward_A Calculation:**
  - **Condition 1:** If the destination register from `EX/MEM` (`rdMem`) matches `RS_1`, and the `EX/MEM` stage is performing a write operation (`regWrite_Mem` is non-zero), set `Forward_A` to `2'b10`. This means forward data from the `EX/MEM` stage.
  - **Condition 2:** If the destination register from `MEM/WB` (`rdWb`) matches `RS_1`, and the `MEM/WB` stage is performing a write operation (`regWrite_Wb` is non-zero), and the first condition is not met, set `Forward_A` to `2'b01`. This means forward data from the `MEM/WB` stage.
  - **Else:** If neither condition is met, set `Forward_A` to `2'b00`. This means no forwarding is needed for `RS_1`.

For Source Register 2 (`RS_2`):

- **Forward_B Calculation:**
  - **Condition 1:** If the destination register from `EX/MEM` (`rdMem`) matches `RS_2`, and the `EX/MEM` stage is performing a write operation (`regWrite_Mem` is non-zero), set `Forward_B` to `2'b10`. This means forward data from the `EX/MEM` stage.
  - **Condition 2:** If the destination register from `MEM/WB` (`rdWb`) matches `RS_2`, and the `MEM/WB` stage is performing a write operation (`regWrite_Wb` is non-zero), and the first condition is not met, set `Forward_B` to `2'b01`. This means forward data from the `MEM/WB` stage.
  - **Else:** If neither condition is met, set `Forward_B` to `2'b00`. This means no forwarding is needed for `RS_2`.

❖ ALU Control Unit

- **Verilog Code –**

```
module alu_control(
  input [1:0] Aluop,
  input [3:0] funct,
  output reg [3:0] operation);

  always @ (*)
    begin
      if (Aluop==2'b01)
        begin
          operation = 4'b0110;
        end
      if (Aluop==2'b00)
        begin
          operation = 4'b0010;
        end
      if (Aluop==2'b10)
        begin
          if (funct == 4'b0000)
            begin
              operation = 4'b0010;
            end
          if (funct == 4'b0111)
            begin
              operation = 4'b0000;
            end
          if (funct == 4'b1000)
            begin
              operation = 4'b0110;
            end
          if (funct == 4'b0110)
            begin
              operation = 4'b0001;
            end
        end
    end
endmodule
```

- **Code Explanation –**

The `alu_control` module is used to generate the ALU operation code (`operation`) based on the `Aluop` and `funct` inputs. Here's a breakdown of its functionality:

Inputs

1. **Aluop (2-bit):**
   - This signal indicates the type of ALU operation to perform. It is derived from the control unit and typically specifies whether the operation is an arithmetic or logical operation.
2. **funct (4-bit):**
   - This signal provides additional details about the ALU operation, often used to specify the exact operation to perform within a broader category.

Output

1. **operation (4-bit):**
   - This signal specifies the actual ALU operation that should be performed. The `operation` code will control the ALU's behavior.

Functionality

1. **When `Aluop` is `2'b01`:**
   - The module sets `operation` to `4'b0110`, which typically represents a subtraction operation.
2. **When `Aluop` is `2'b00`:**
   - The module sets `operation` to `4'b0010`, which typically represents an addition operation.
3. **When `Aluop` is `2'b10`:**
   - The module checks the value of `funct` to determine the specific ALU operation:
     - **If `funct` is `4'b0000`:** Sets `operation` to `4'b0010` (addition).
     - **If `funct` is `4'b0111`:** Sets `operation` to `4'b0000` (AND operation).
     - **If `funct` is `4'b1000`:** Sets `operation` to `4'b0110` (subtraction).
     - **If `funct` is `4'b0110`:** Sets `operation` to `4'b0001` (OR operation).

- **RTL Design for ALU control :-**

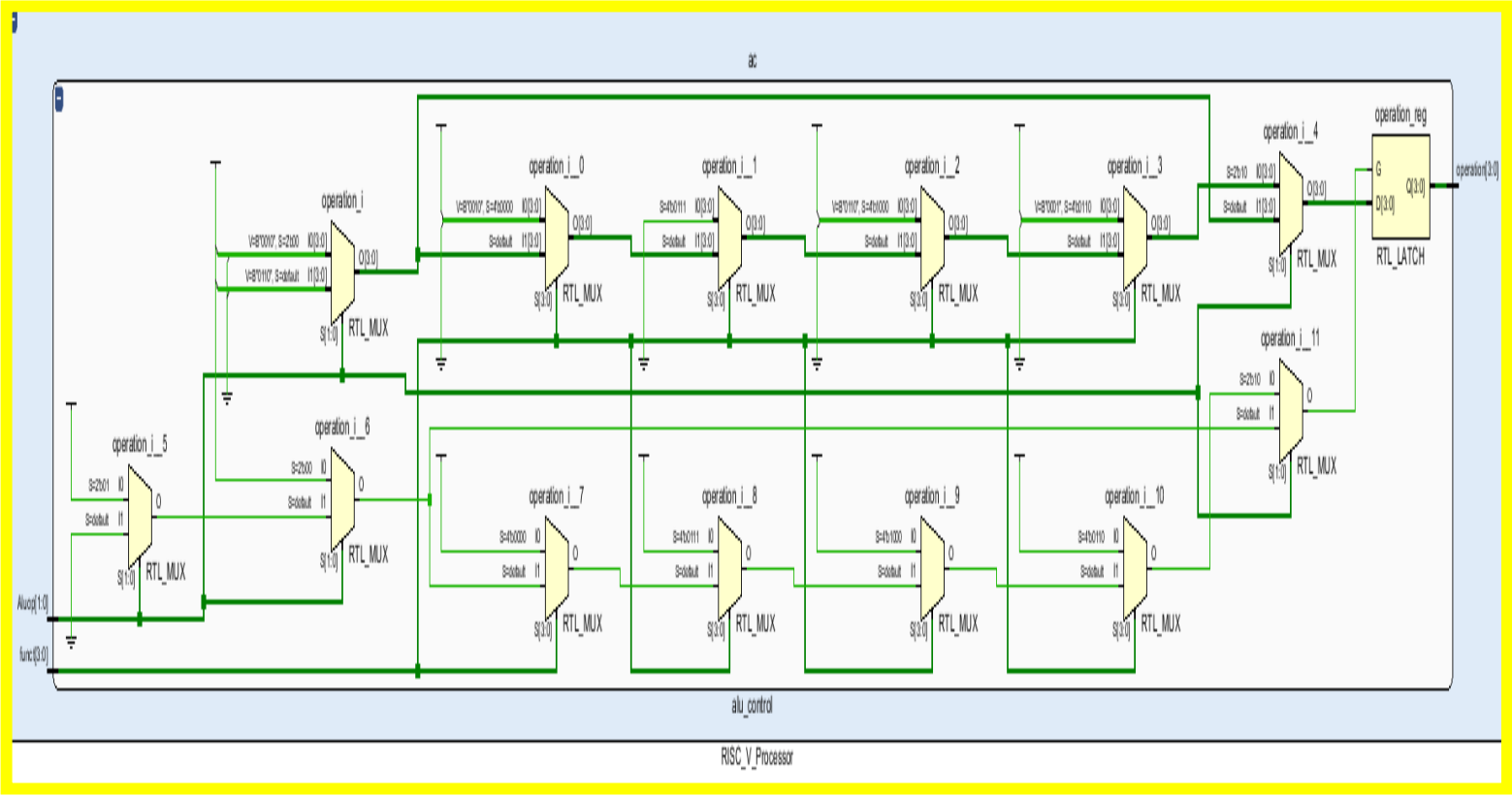


Fig 2.7 – ALU_ctrl

❖ Arithmetic Logic Unit

- **Verilog Code –**

```
module Alu64
  (
    input [63:0] a,b,
    input [3:0] ALuop,
    output reg [63:0] Result,
    output reg zero

  );

  always @(*)
    begin
      case (ALuop)
        4'b0000: Result = a & b;
        4'b0001: Result = a | b;
        4'b0010: Result = a + b;
        4'b0110: Result = a - b;
        4'b1100: Result = ~(a | b); //nor
      endcase
      if (Result == 0)
        zero = 1;
      else
        zero = 0;
    end
endmodule
```

- **RTL Design for ALU –**

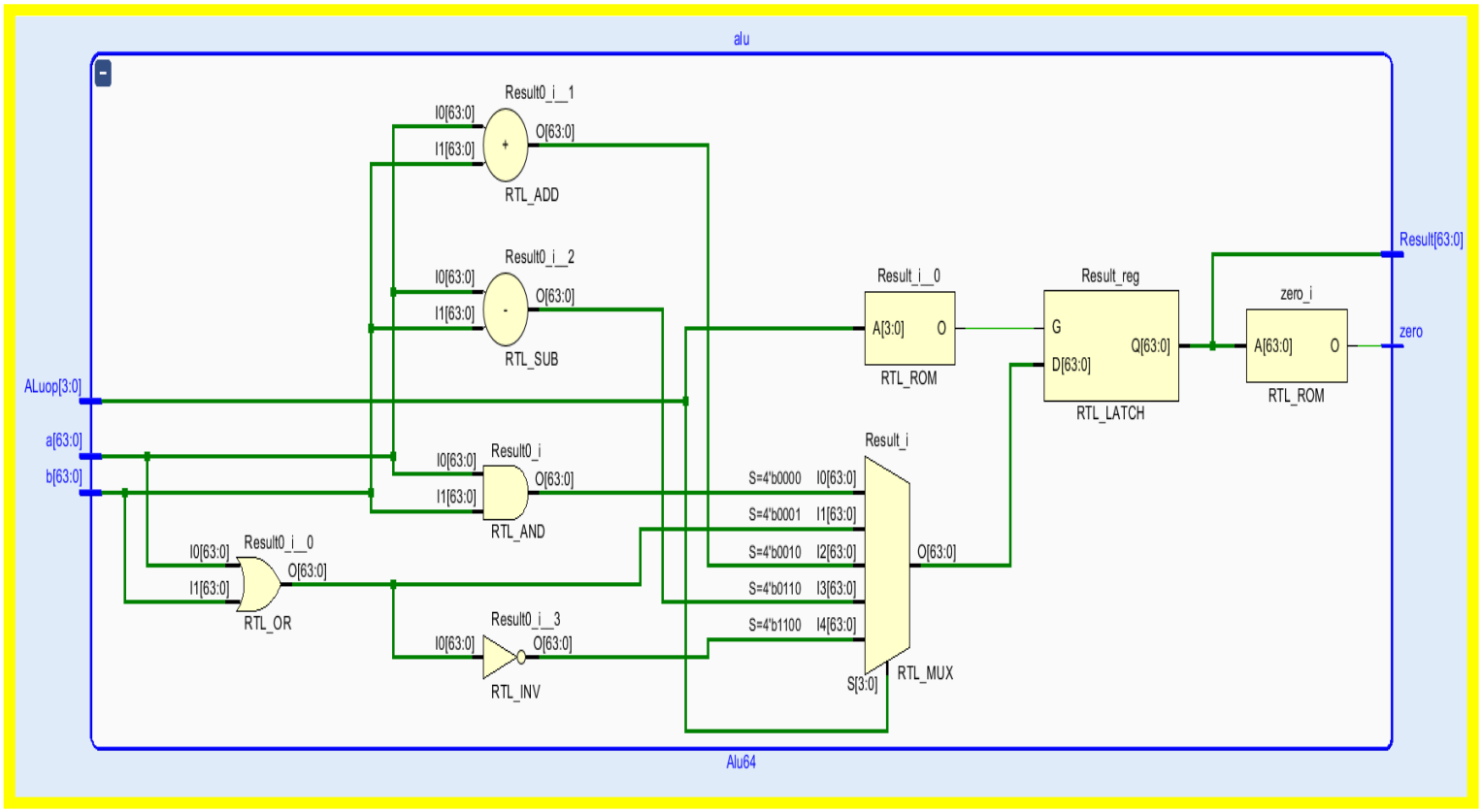


Fig 2.8 – ALU

❖ Branching Unit

- **Verilog Code :-**

```
module branching_unit
  (
   input [2:0] funct3,
    input [63:0] readData1,
    input [63:0] b,
   output reg addermuxselect
  );

  initial
    begin
      addermuxselect = 1'b0;
    end

  always @(*)
  begin
      case (funct3)
        3'b000:
          begin
            if (readData1 == b)
              addermuxselect = 1'b1;
            else
              addermuxselect = 1'b0;
            end
         3'b100:
               begin
              if (readData1 < b)
              addermuxselect = 1'b1;
            else
              addermuxselect = 1'b0;
            end
        3'b101:
          begin
            if (readData1 > b)
                addermuxselect = 1'b1;
             else
              addermuxselect = 1'b0;
            end
       endcase
      end
endmodule
```

- **Code Explanation :-**

Inputs

1. **funct3 (3-bit):**
   - This field specifies the type of branch operation (e.g., equality, less than, greater than) and is part of the instruction encoding.
2. **readData1 (64-bit):**
   - This is the value from a register that will be compared against another value to make a branch decision.
3. **b (64-bit):**
   - This is the value to which `readData1` is compared. It typically comes from the immediate value or a different source depending on the instruction.

Output

1. **addermuxselect (1-bit):**
    - This output determines whether the branch should be taken (`1`) or not (`0`). It is used to select the correct input for the program counter adder, affecting the branch decision.

Functionality

- **Initialization:**
    - `addermuxselect` is initialized to `0` to ensure a default value at startup.
- **Branch Decision Logic:**
    - **Equality (`funct3 = 3'b000`):**
        - Compares `readData1` and `b`. If they are equal, `addermuxselect` is set to `1` (branch taken). If not equal, it is set to `0` (branch not taken).
    - **Less Than (`funct3 = 3'b100`):**
        - Compares if `readData1` is less than `b`. If true, `addermuxselect` is set to `1` (branch taken). Otherwise, it is set to `0` (branch not taken).
    - **Greater Than (`funct3 = 3'b101`):**
        - Compares if `readData1` is greater than `b`. If true, `addermuxselect` is set to `1` (branch taken). Otherwise, it is set to `0` (branch not taken).

- **RTL Design for Branching Unit :-**

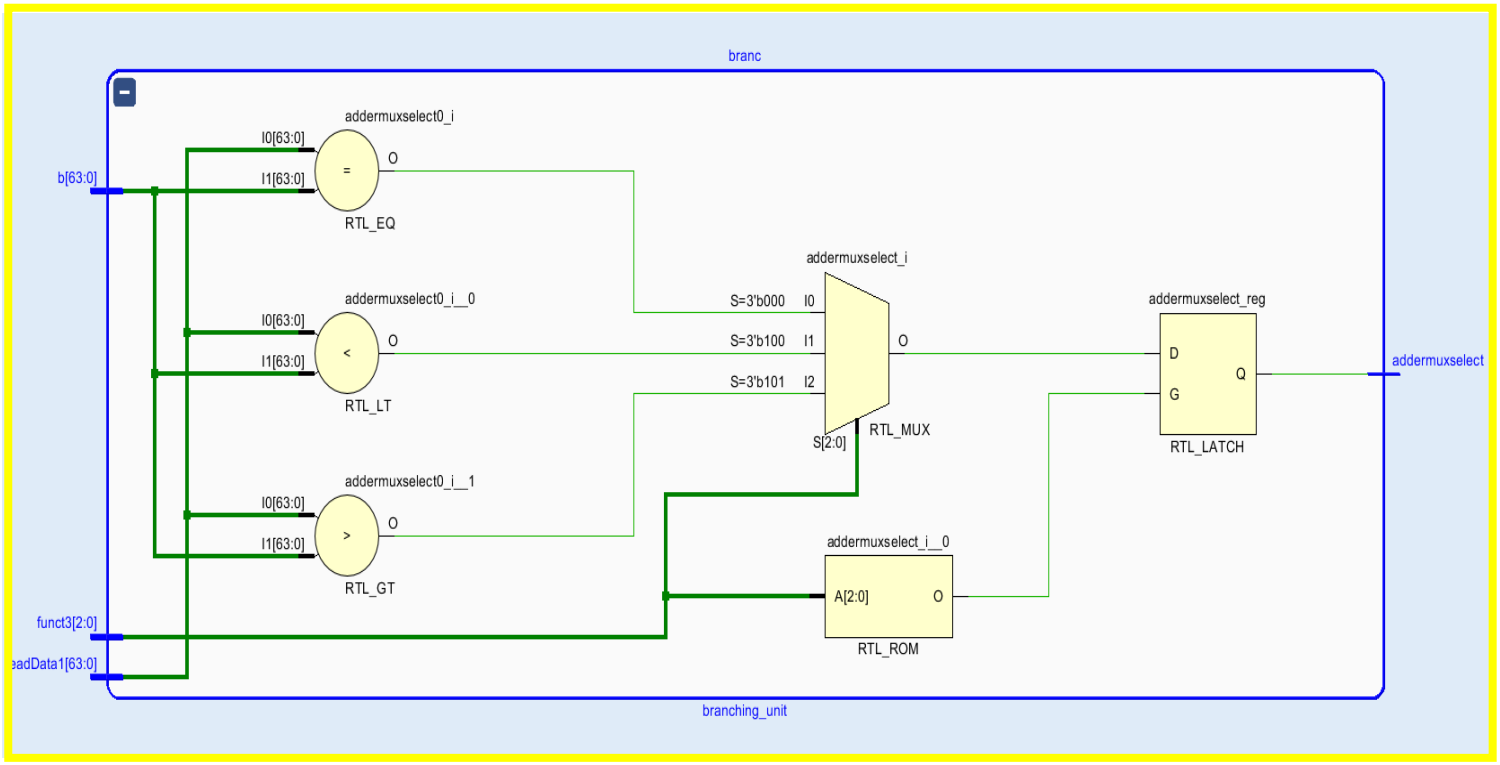


Fig 2.9 – Branching Unit

❖ Three-by-One Mux

- **Verilog Code -**

```
module ThreebyOneMux
  (
    input[63:0] a,
    input[63:0] b,
    input[63:0] c,
    input [1:0] sel,
    output reg [63:0] out
  );
  always @(*)
    begin
      case(sel[1:0])
        2'b00: out = a;
        2'b01: out = b;
        2'b10: out = c;
      endcase
    end
endmodule
```

- **Code Explanation –**

Inputs

1. **a (64-bit):** First input to the multiplexer.
2. **b (64-bit):** Second input to the multiplexer.
3. **c (64-bit):** Third input to the multiplexer.
4. **sel (2-bit):** Select signal used to choose which of the three inputs (`a`, `b`, or `c`) is routed to the output.

Output

1. **out (64-bit):** The output of the multiplexer, which is one of the inputs selected based on the `sel` signal.

Functionality

- **Multiplexer Operation:**
  - The `always @(*)` block ensures the multiplexer evaluates and updates its output whenever any of its inputs or select signals change.
  - **Case Statement:**
    - **`2'b00`:** If `sel` is `00`, the output `out` is set to `a`.
    - **`2'b01`:** If `sel` is `01`, the output `out` is set to `b`.
    - **`2'b10`:** If `sel` is `10`, the output `out` is set to `c`.
    - **Default Case:** If `sel` is `11`, no action is defined explicitly, but the multiplexer will retain the last valid output due to the `always` block behavior.

❖ Two-by-One Mux (for ALU input selection)

- **Verilog Code –**

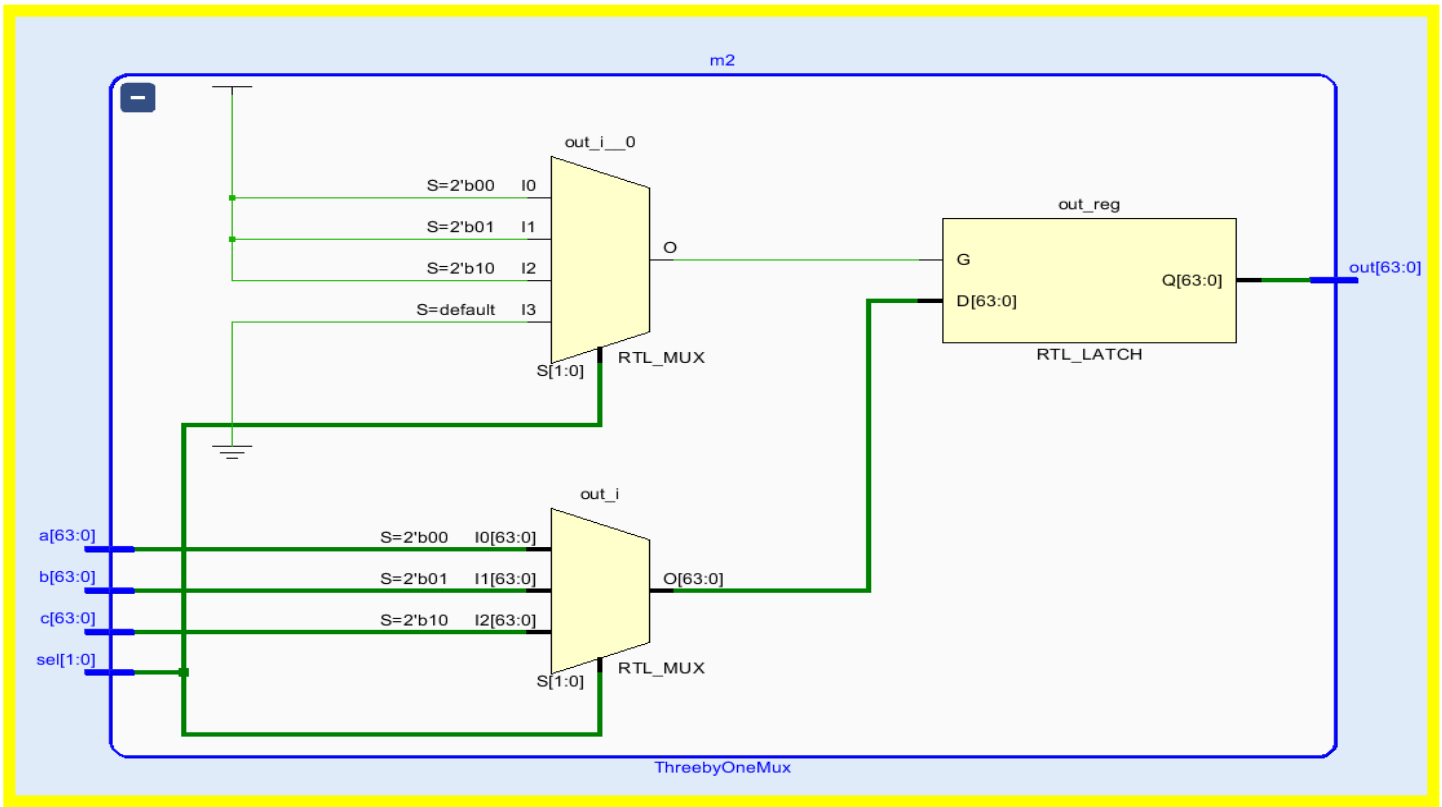
```
module twox1Mux(
  input [63:0] A,B,
  input SEL,
  output reg[63:0] Y);
  always @ (A or B or SEL)
    begin
      if (SEL==0)
        Y=A;
      else
        Y=B;
    end
endmodule
```

- **Code Explanation –**

Functionality

- **Always Block:**
  - The `always @ (A or B or SEL)` block means that the block will execute whenever any of its inputs (`A`, `B`, or `SEL`) change.
  - **Conditional Statement:**
    - **If `SEL` is `0`:** The output `Y` is set to `A`.
    - **If `SEL` is `1`:** The output `Y` is set to `B`.

- **RTL design 2_1 Mux and 3_1 Mux :-**



- **RTL for 2_1 Mux :-**

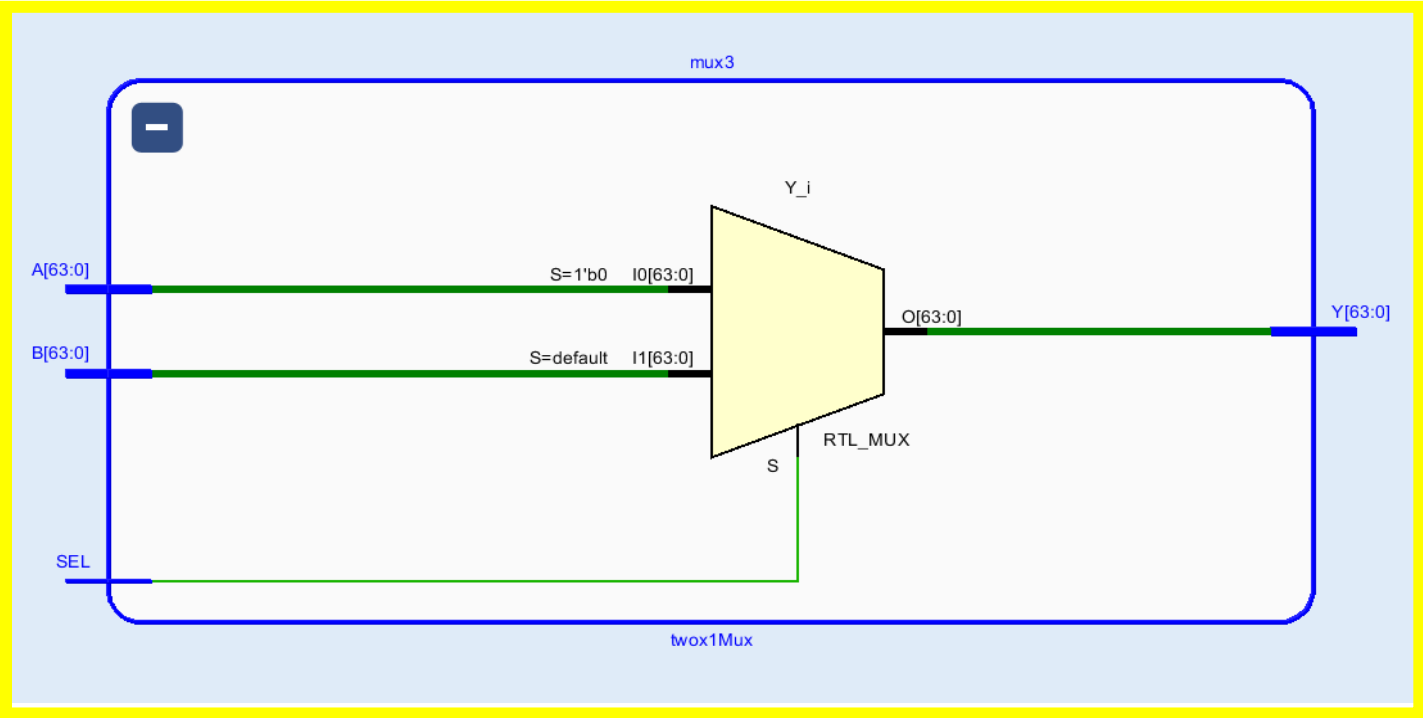


Fig 3.0 – 2_1 and 3_1 Mux

❖ Adder for Branch Calculation

- **Verilog Code –**

```
module adder (
    input [63:0] p,      // First operand (e.g., Program Counter or base address)
    input [63:0] q,      // Second operand (e.g., immediate value for branching)
    output [63:0] out    // Sum of p and q (branch target address)
);

    // Perform the addition
    assign out = p + q;

endmodule
```

- **Code Explanation –**

1. **Inputs:**
   - **p (64-bit):** The first operand. In the context of branch target calculation, this is typically the current value of the program counter (PC) or another address.
   - **q (64-bit):** The second operand. This is usually an immediate value or offset that you want to add to `p` to calculate the branch target address.
2. **Output:**

- **out (64-bit):** The result of the addition. This represents the new address to which the program counter should be set for branching.

Functionality

- The module uses the `assign` statement to continuously add the two 64-bit inputs (`p` and `q`) and drive the result to the `out` output.
- This operation effectively calculates the branch target address by adding the offset (`q`) to the base address (`p`).

- **RTL Design for Adder :**

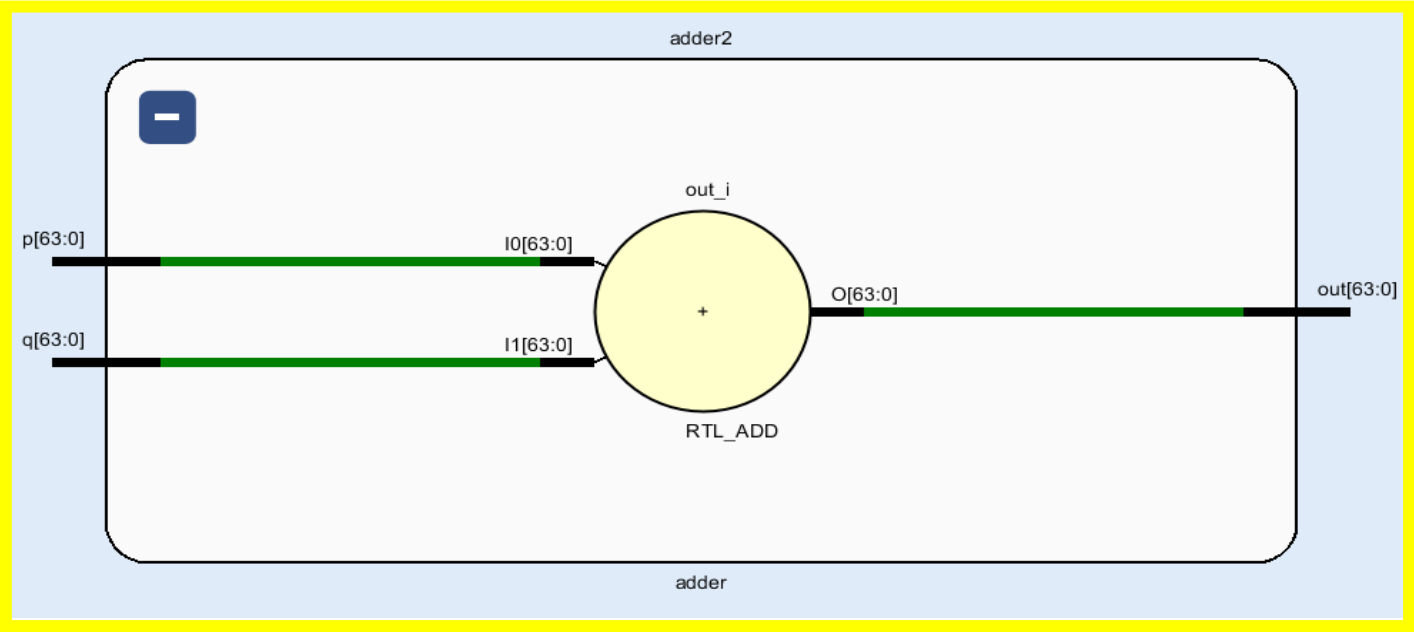


Fig 3.1 – Adder Unit

❖ EX/MEM Pipeline Register

- **Verilog Code –**

```
module EXMEM (
    input clk,              // Clock signal
    input reset,            // Reset signal
    input flush,            // Flush signal to clear pipeline register
    input [63:0] ALU_result_in, // Result from ALU
    input [63:0] write_data_in, // Data to be written to memory
    input [63:0] branch_target_in, // Address for branching (if
applicable)
    input [4:0] rd_in,      // Destination register
    input branch_in,        // Branch control signal
    input memread_in,       // Memory read control signal
    input memtoreg_in,      // Memory-to-register control signal
    input memwrite_in,      // Memory write control signal
    input regwrite_in,      // Register write control signal
    output reg [63:0] ALU_result_out, // Output ALU result
    output reg [63:0] write_data_out, // Output data to be written to
memory
    output reg [63:0] branch_target_out, // Output branch target address
```

```
    output reg [4:0] rd_out, // Output destination register
    output reg branch_out,   // Output branch control signal
    output reg memread_out,  // Output memory read control signal
    output reg memtoreg_out, // Output memory-to-register control signal
    output reg memwrite_out, // Output memory write control signal
    output reg regwrite_out  // Output register write control signal
);

    // Sequential block for updating pipeline register on the rising edge of the clock
    always @(posedge clk or posedge reset) begin

        if (reset) begin
            // Clear the pipeline register on reset
            ALU_result_out <= 64'b0;
            write_data_out <= 64'b0;
            branch_target_out <= 64'b0;
            rd_out <= 5'b0;
            branch_out <= 1'b0;
            memread_out <= 1'b0;
            memtoreg_out <= 1'b0;
            memwrite_out <= 1'b0;
            regwrite_out <= 1'b0;

        end else if (flush) begin
            // Clear the pipeline register on flush
            ALU_result_out <= 64'b0;
            write_data_out <= 64'b0;
            branch_target_out <= 64'b0;
            rd_out <= 5'b0;
            branch_out <= 1'b0;
            memread_out <= 1'b0;
            memtoreg_out <= 1'b0;
            memwrite_out <= 1'b0;
            regwrite_out <= 1'b0;

        end else begin
            // Update the pipeline register with input values
            ALU_result_out <= ALU_result_in;
            write_data_out <= write_data_in;
            branch_target_out <= branch_target_in;
            rd_out <= rd_in;
            branch_out <= branch_in;
            memread_out <= memread_in;
            memtoreg_out <= memtoreg_in;
            memwrite_out <= memwrite_in;
            regwrite_out <= regwrite_in;
        end
    end

endmodule
```

- **RTL Design for EX/MEM Register -**

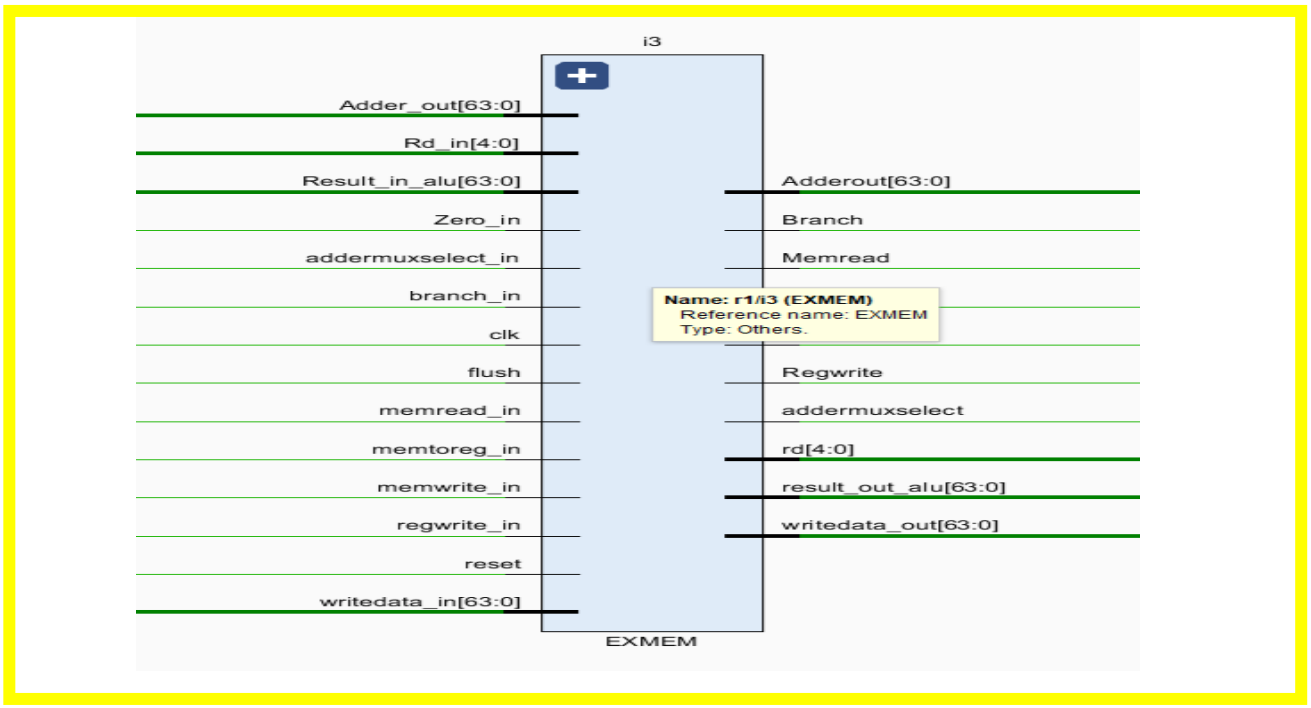
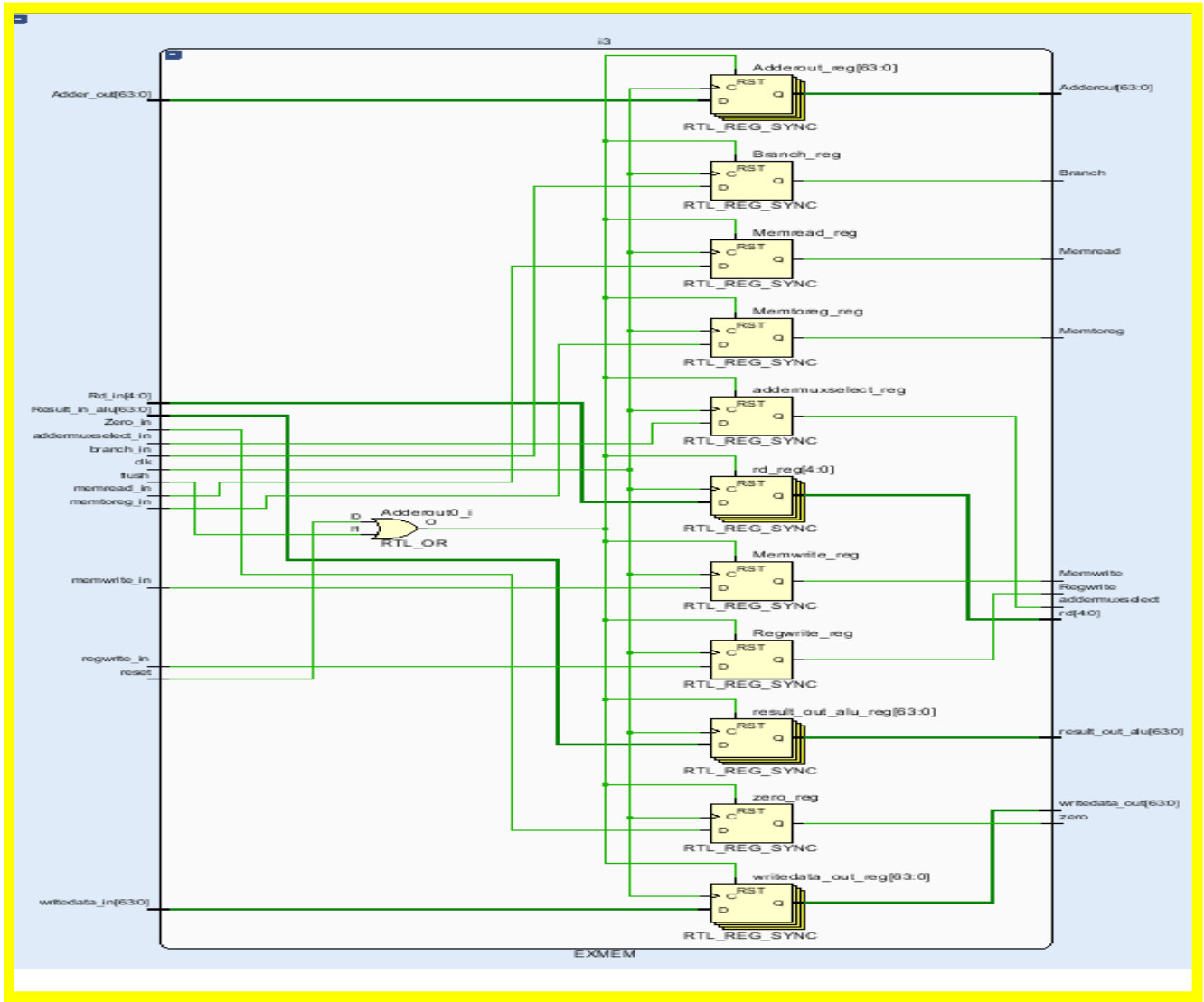


Fig 3.2 – EX/MEM Stage



- Code Explanation –

Inputs

- **clk:** Clock signal to synchronize updates.
- **reset:** Signal to clear the pipeline register, typically used during a system reset.
- **flush:** Signal to clear the pipeline register, often used to handle pipeline stalls or branch mispredictions.
- **ALU_result_in:** The result of the ALU operation, which is to be stored and potentially used for memory access or branching.
- **write_data_in:** Data that is to be written to the memory.
- **branch_target_in:** Address calculated for branching purposes, used if the current instruction results in a branch.
- **rd_in:** Identifier for the destination register where the result will be written back.
- **branch_in:** Control signal indicating whether a branch operation is requested.
- **memread_in:** Control signal indicating whether a memory read operation is requested.
- **memtoreg_in:** Control signal indicating whether the result should be written back to the register file from memory.
- **memwrite_in:** Control signal indicating whether a memory write operation is requested.
- **regwrite_in:** Control signal indicating whether the register file should be updated.

Outputs

- **ALU_result_out:** The output value of the ALU result, which will be used in the MEM stage or for further operations.
- **write_data_out:** Data that will be written to memory, passed to the MEM stage.
- **branch_target_out:** The branch target address, which is used to determine the next instruction address if a branch is taken.
- **rd_out:** Identifier of the destination register, to be updated in the register file if necessary.
- **branch_out:** Indicates if a branch operation is to be performed.
- **memread_out:** Indicates if a memory read operation is to be performed.
- **memtoreg_out:** Indicates if the result should be written back to the register file from memory.
- **memwrite_out:** Indicates if a memory write operation is to be performed.
- **regwrite_out:** Indicates if the register file should be updated with the result.

Operation

- **Clock Trigger:** The pipeline register updates its values on the rising edge of the clock signal (`posedge clk`).
- **Reset Handling:** When the reset signal is active (`posedge reset`), the register clears all its outputs to zero, effectively initializing or resetting the pipeline stage.
- **Flush Handling:** If the flush signal is active, the pipeline register is also cleared, which is useful for handling branch miss predictions or stalls by discarding incorrect or unwanted data.
- **Normal Operation:** When neither reset nor flush is active, the pipeline register updates its outputs with the current values of the inputs. This transfers the ALU results, data, control signals, and addresses from the EX stage to the MEM stage.**Let's move to next stage that is MEM Block ……**

❖ MEM Stage

1. Data Memory

- **Function:** Manages read and write operations to the memory.
- **Operation:** When a write operation is needed, it writes data to the specified memory address. When a read operation is requested, it retrieves data from a specified address. This component directly interacts with the system memory or cache.

2. MEMWB Pipeline Register

- **Function:** Acts as a buffer between the MEM stage and the Writeback (WB) stage.
- **Operation:** It temporarily holds data that is to be written to a register or used for further computation. This includes data read from memory and results from the ALU operations. The pipeline register ensures that data and control signals are correctly transferred from one stage to the next, maintaining consistency and synchronization.

❖ Data Memory

- **Verilog Code –**

```
module data_memory
  (input [63:0] write_data,
   input [63:0] address,
   input memorywrite, clk,memoryread,
   output reg [63:0] read_data,
  output [63:0] element1,
  output [63:0] element2,
  output [63:0] element3,
  output [63:0] element4,
  output [63:0] element5,
  output [63:0] element6,
  output [63:0] element7,
  output [63:0] element8);

  reg [7:0] mem [63:0]; //memory array
  integer i;
  initial
    begin
      for (i=0; i<64; i=i+1)
        begin
          mem[i] = 0;

        end
      mem[0] = 8'd1;
      mem[8] = 8'd2;
      mem[16] = 8'd3;
      mem[24] = 8'd4;
      mem[32] = 8'd5;
      mem[40] = 8'd6;
      mem[48] = 8'd7;
      mem[56] = 8'd8;
    end
  assign element1 =
{mem[7],mem[6],mem[5],mem[4],mem[3],mem[2],mem[1],mem[0]};
```

```
    assign element2 =
  {mem[15],mem[14],mem[13],mem[12],mem[11],mem[10],mem[9],mem[8]};
    assign element3 =
  {mem[23],mem[22],mem[21],mem[20],mem[19],mem[18],mem[17],mem[16]};
    assign element4 =
  {mem[31],mem[30],mem[29],mem[28],mem[27],mem[26],mem[25],mem[24]};
    assign element5 =
  {mem[39],mem[38],mem[37],mem[36],mem[35],mem[34],mem[33],mem[32]};
    assign element6 =
  {mem[47],mem[46],mem[45],mem[44],mem[43],mem[42],mem[41],mem[40]};
    assign element7 =
  {mem[55],mem[54],mem[53],mem[52],mem[51],mem[50],mem[49],mem[48]};
    assign element8 =
  {mem[63],mem[62],mem[61],mem[60],mem[59],mem[58],mem[57],mem[56]};

    always @ (*)
      begin
        if (memoryread)
          begin
            read_data[7:0] = mem[address+0];
            read_data[15:8] = mem[address+1];
            read_data[23:16] = mem[address+2];
            read_data[31:24] = mem[address+3];
            read_data[39:32] = mem[address+4];
            read_data[47:40] = mem[address+5];
            read_data[55:48] = mem[address+6];
            read_data[63:56] = mem[address+7];
          end
      end
    always @(posedge clk)
      begin
        if (memorywrite)
          begin
            mem[address] = write_data[7:0];
            mem[address+1] = write_data[15:8];
            mem[address+2] = write_data[23:16];
            mem[address+3] = write_data[31:24];
            mem[address+4] = write_data[39:32];
            mem[address+5] = write_data[47:40];
            mem[address+6] = write_data[55:48];
            mem[address+7] = write_data[63:56];
          end
      end
  endmodule
```

- **Code Explanation –**

Key Components

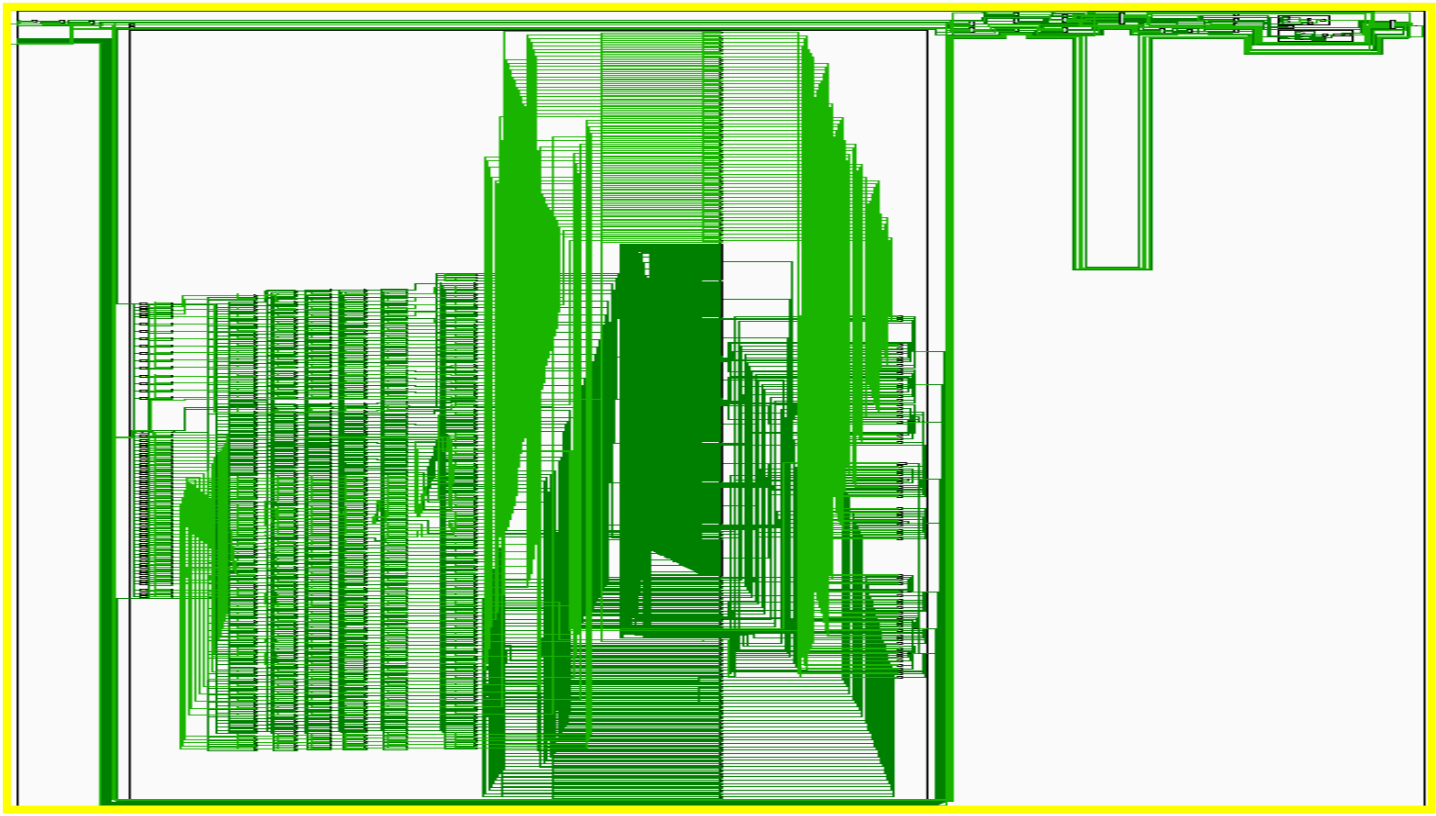
1. **Memory Array:**
   - **Definition:** `reg [7:0] mem [63:0];`
   - **Size:** 64 elements, each 8 bits wide, representing 64 bytes of memory.
   - **Initialization:**
     - The `initial` block sets all memory locations to zero initially.
     - Specific locations are pre-loaded with values to simulate initial conditions.

2. **Inputs:**
    - **write_data [63:0]:** 64-bit input that provides the data to be written into memory.
    - **address [63:0]:** 64-bit input specifying the starting address for read or write operations.
    - **memorywrite:** Control signal that enables writing to memory when active.
    - **clk:** Clock signal used to synchronize write operations.
    - **memoryread:** Control signal that enables reading from memory when active.
3. **Outputs:**
    - **read_data [63:0]:** 64-bit output that provides the data read from memory.
    - **element1 to element8 [63:0]:** Outputs that show the contents of the memory in 64-bit chunks. These outputs help in debugging and visualizing memory data.

Operations

1. **Reading from Memory:**
    - **Control:** When `memoryread` is active, the module reads from the memory.
    - **Process:** The `always @ (*)` block:
        - Reads 8 consecutive bytes from the memory starting at the specified `address`.
        - Combines these bytes into the `read_data` output.
2. **Writing to Memory:**
    - **Control:** When `memorywrite` is active, the module writes to memory.
    - **Process:** The `always @(posedge clk)` block:
        - Writes `write_data` to 8 consecutive memory locations starting at the specified `address`.
        - Each byte of `write_data` is written into successive addresses in the memory array.

- **RTL Design for Data Memory –**



❖ MEM/WB Pipeline Register

- **Verilog Code –**

```
module MEMWB(
  input clk,reset,
  input [63:0] read_data_in,
  input [63:0] result_alu_in, //2 bit 2by1 mux input b
  input [4:0]Rd_in, //EX MEM output
  input memtoreg_in, regwrite_in, //ex mem output as mem wb inputs
  output reg [63:0] readdata, //1bit
  output reg [63:0] result_alu_out,//1bit
  output reg [4:0] rd,
  output reg Memtoreg, Regwrite
);

  always @(posedge clk)
    begin
      if (reset == 1'b1)
        begin
          readdata = 63'b0;
          result_alu_out = 63'b0;
          rd = 5'b0;
          Memtoreg = 1'b0;
          Regwrite = 1'b0;

        end
      else
        begin
         readdata = read_data_in;
          result_alu_out = result_alu_in;
          rd = Rd_in;
          Memtoreg = memtoreg_in;
          Regwrite = regwrite_in;
        end
    end
endmodule
```

- **Code Explanation –**

1. **Inputs:**
   - **`clk`:** Clock signal to synchronize the operations.
   - **`reset`:** Asynchronous reset signal to initialize the pipeline register.
   - **`read_data_in [63:0]`:** 64-bit data read from memory.
   - **`result_alu_in [63:0]`:** 64-bit result from the ALU.
   - **`Rd_in [4:0]`:** 5-bit destination register identifier from the MEM stage.
   - **`memtoreg_in`:** Control signal indicating whether the data to be written back is from memory or the ALU.
   - **`regwrite_in`:** Control signal to enable writing to the register file.
2. **Outputs:**
   - **`readdata [63:0]`:** 64-bit output that provides the data read from memory.
   - **`result_alu_out [63:0]`:** 64-bit output that provides the result from the ALU.
   - **`rd [4:0]`:** 5-bit output indicating the destination register.

- **Memtoreg:** Control signal output that determines if the data to be written back comes from memory or the ALU.
- **Regwrite:** Control signal output to enable or disable writing to the register file.

- **RTL Design for MEM/WB Stage**

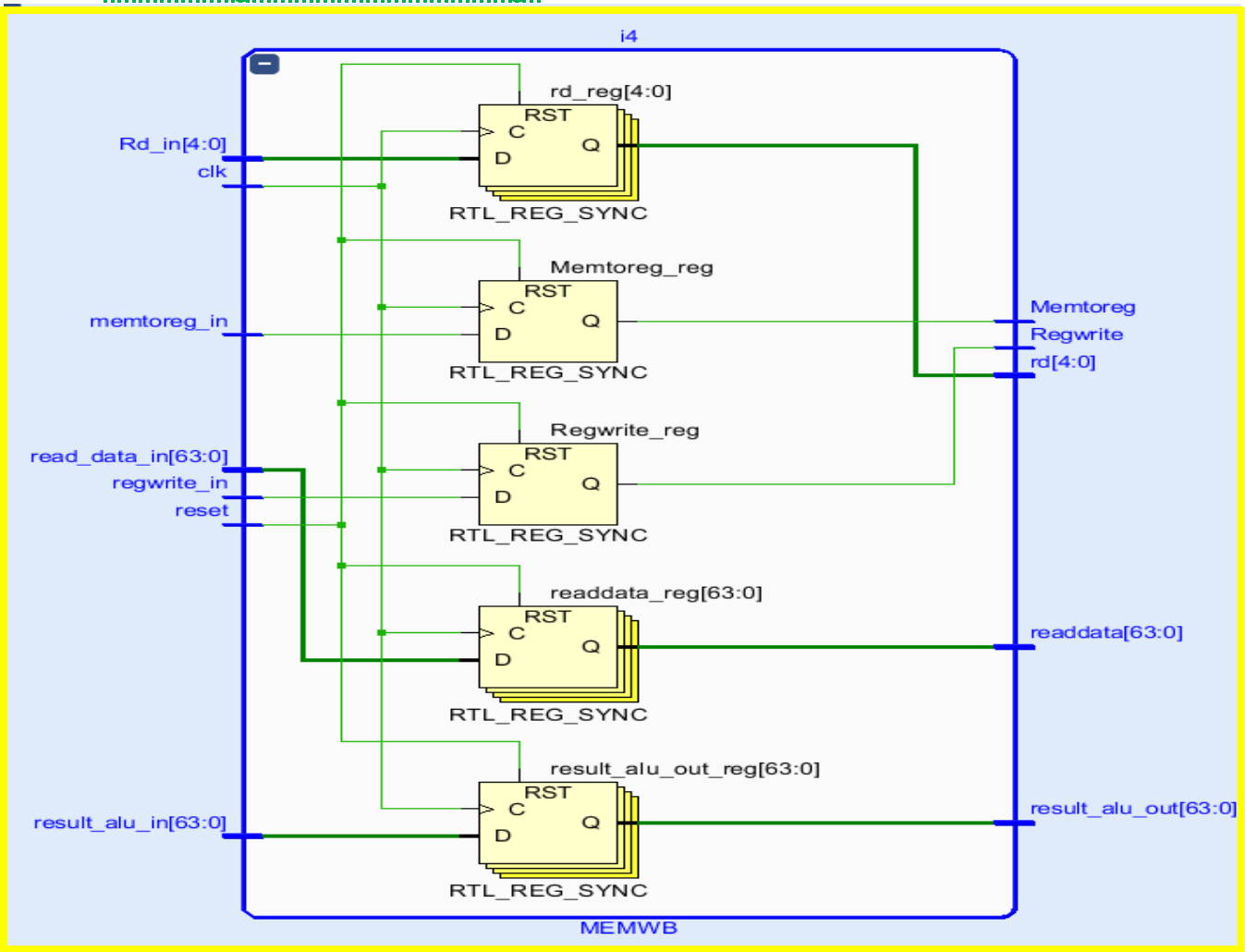


Fig 3.5 - MEM/WB Register

❖ twox1Mux Module (used in mux3)

Operation:

- **Purpose:** This 2-to-1 multiplexer selects between two inputs based on a control signal.
- **Inputs:**
  - `A`: The first input (ALU result).
  - `B`: The second input (data from memory).
  - `SEL`: The control signal (`memwb_memtoreg`) that determines which input is selected.
- **Output:**
  - `Y`: The selected output data which will be written back to the register file.

- **Verilog Code –**

```
module twox1Mux(
  input [63:0] A,B,
  input SEL,
  output reg[63:0] Y);
```

```
    always @ (A or B or SEL)
      begin
        if (SEL==0)
          Y=A;
        else
          Y=B;
      end
  endmodule
```

- **RTL Design for `twox1Mux` Module**

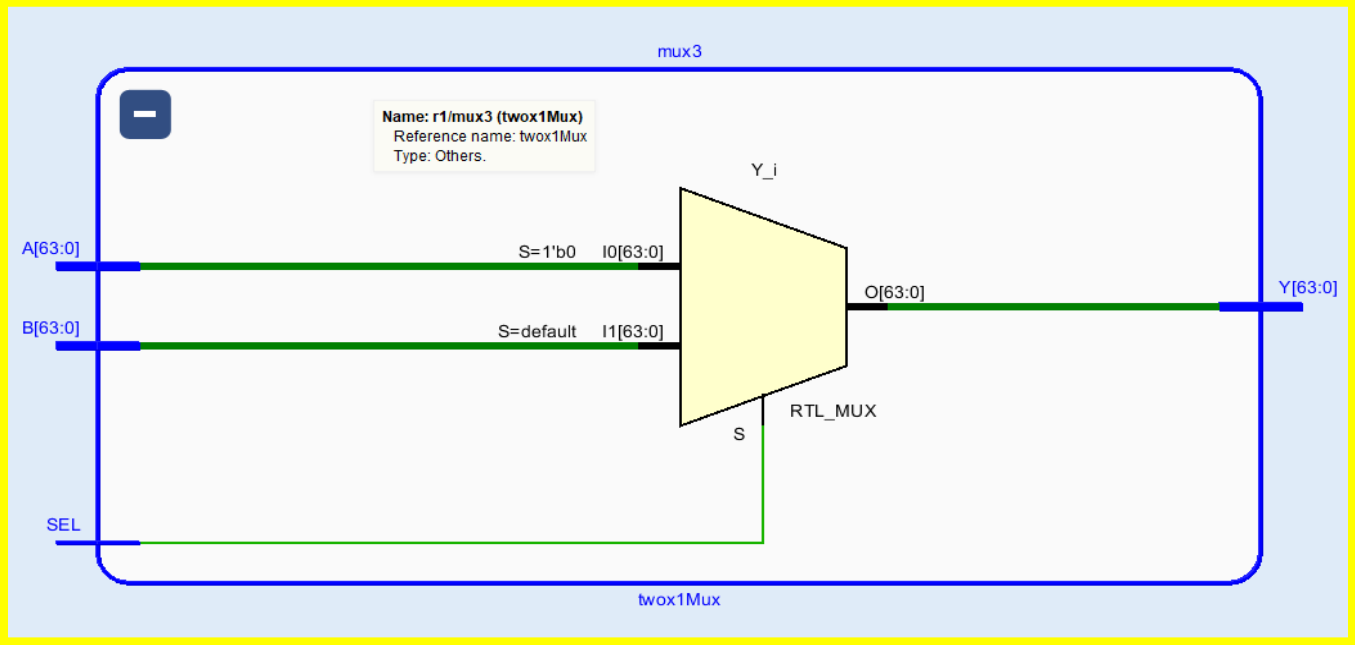


Fig 3.6 – two-1 Mux

❖ Combining all Modules for the Final code of Processor

- **Final Processor Code –**

```
module RISC_V_Processor(input clk, input reset, input wire[63:0] element1,
  input wire[63:0] element2, input wire[63:0] element3,
 input wire[63:0]element4, input wire[63:0] element5,input wire[63:0]
element6,input wire[63:0] element7,input wire[63:0] element8,input stall,
flush);
  wire branch;
  wire memread;
  wire memtoreg;
  wire memwrite;
  wire ALUsrc;
  wire regwrite;
  wire [1:0] ALUop;
  wire regwrite_memwb_out;
  wire [63:0] readdata1, readdata2;
  wire [63:0] r8, r19, r20, r21, r22;
  wire [63:0] write_data;
  wire [63:0] pc_in;
  wire [63:0] pc_out;
  wire [63:0] adderout1;
  wire [63:0] adderout2;
  wire [31:0] instruction;
  wire[31:0] inst_ifid_out;
  wire [6:0] opcode;
  wire [4:0] rd, rs1, rs2;
  wire [2:0] funct3;
  wire [6:0] funct7;
  wire [63:0] imm_data;
  wire [63:0] random;
  wire [63:0] a1;
  wire [4:0] RS1;
  wire [4:0] RS2;
  wire [4:0] RD;
  wire [63:0] d, M1, M2;
  wire Branch;
  wire Memread;
  wire Memtoreg;
  wire Memwrite;
  wire Regwrite;
  wire Alusrc;
  wire [1:0] aluop;
  wire [3:0] funct4_out;
  wire [63:0] threeby1_out1;
  wire[63:0] threeby1_out2;
  wire[63:0]  alu_64_b;
  wire [63:0] write_Data;
  wire [63:0] exmem_out_adder;
  wire exmem_out_zero;
  wire [63:0] exmem_out_result;
  wire [4:0] exmemrd;
  wire BRANCH,MEMREAD,MEMTOREG,MEMEWRITE,REGWRITE;
  wire [63:0] AluResult;
  wire zero;
  wire [3:0] operation;
  wire [63:0] readdata;
  wire[63:0] muxin1,muxin2;
  wire [4:0] memwbrd;
  wire memwb_memtoreg;
```

```
    wire memwb_regwrite;
    wire [1:0] forwardA;
    wire [1:0] forwardB;
    wire addermuxselect;
    wire branch_final;
    pipeline_flush p_flush
    (.branch(branch_final & BRANCH),
      .flush(flush));
    hazard_detection_unit hu
    (.Memread(Memread),
      .inst(inst_ifid_out),
      .Rd(RD),
      .stall(stall));
    program_counter pc
    (.PC_in(pc_in),
      .stall(stall),
      .clk(clk),
      .reset(reset),
      .PC_out(pc_out) );
    instruction_memory im
    (.inst_address(pc_out),
      .instruction(instruction));
    adder adder1
    (.p(pc_out),
      .q(64'd4),
      .out(adderout1));
     IFID i1
    (.clk(clk),
      .reset(reset),
      .IFIDWrite(stall),
      .instruction(instruction),
      .A(pc_out),
      .inst(inst_ifid_out),
      .a_out(random),
      .flush(flush));
    Parser ip
    (.instruction(inst_ifid_out),
      .opcode(opcode),
      .rd(rd),
      .funct3(funct3),
      .rs1(rs1),
      .rs2(rs2),
      .funct7(funct7));
      CU cu
    (.opcode(opcode),
      .branch(branch),
      .memread(memread),
      .memtoreg(memtoreg),
      .memwrite(memwrite),
      .aluSrc(ALUsrc),
      .regwrite(regwrite),
      .Aluop(ALUop),
      .stall(stall));
      data_extractor immextr
    (.instruction(inst_ifid_out),
      .imm_data(imm_data));
    registerFile regfile
    (.clk(clk),
      .reset(reset),
      .rs1(rs1),
      .rs2(rs2),
      .rd(memwbrd),
```

```
      .writedata(write_data),
      .reg_write(memwb_regwrite),
      .readdata1(readdata1),
      .readdata2(readdata2),
      .r8(r8),
      .r19(r19),
      .r20(r20),
      .r21(r21),
      .r22(r22));
      IDEX i2
    (.clk(clk),
      .flush(flush),
      .reset(reset),
      .funct4_in({inst_ifid_out[30],inst_ifid_out[14:12]}),
      .A_in(random),
      .readdata1_in(readdata1),
      .readdata2_in(readdata2),
      .imm_data_in(imm_data),
      .rs1_in(rs1),.rs2_in(rs2),.rd_in(rd),
      .branch_in(branch),.memread_in(memread),.memtoreg_in(memtoreg),
  .memwrite_in(memwrite),.aluSrc_in(ALUsrc),.regwrite_in(regwrite),.Aluop_in(
    ALUop),.a(a1),.rs1(RS1),.rs2(RS2),.rd(RD),.imm_data(d),.readdata1(M1),.read
    data2(M2),.funct4_out(funct4_out),.Branch(Branch),.Memread(Memread),.Memtor
    eg(Memtoreg),.Memwrite(Memwrite),.Regwrite(Regwrite),.Alusrc(Alusrc),.aluop
    (aluop));
    adder adder2
    (.p(a1),
      .q(d << 1),
      .out(adderout2));
    ThreebyOneMux m1

    (.a(M1),.b(write_data),.c(exmem_out_result),.sel(forwardA),.out(threeby1_out1)
    );
    ThreebyOneMux m2
    (.a(M2),.b(write_data),.c(exmem_out_result),.sel(forwardB),.out(threeby1_o
    ut2));
    twox1Mux mux1
    (.A(threeby1_out2),.B(d),.SEL(Alusrc),.Y(alu_64_b));
    Alu64 alu
    (.a(threeby1_out1),
      .b(alu_64_b),
      .ALuop(operation),
      .Result(AluResult),
      .zero(zero));
       alu_control ac
    (.Aluop(aluop),
      .funct(funct4_out),
      .operation(operation));
    EXMEM i3
  (.clk(clk),.reset(reset),.Adder_out(adderout2),.Result_in_alu(AluResult),.Ze
    ro_in(zero),.flush(flush),.writedata_in(threeby1_out2),.Rd_in(RD),
    .addermuxselect_in(addermuxselect),.branch_in(Branch),.memread_in(Memread),.
    memtoreg_in(Memtoreg),.memwrite_in(Memwrite),.regwrite_in(Regwrite),.Adderou
    t(
    exmem_out_adder),.zero(exmem_out_zero),.result_out_alu(exmem_out_result),.wr
    itedata_out(write_Data),.rd(exmemrd),.Branch(BRANCH),.Memread(MEMREAD),.Memt
    oreg(MEMTOREG),.Memwrite(MEMEWRITE),.Regwrite(REGWRITE),.addermuxselect(bran
    ch_final));
    data_memory datamem
    ( .write_data(write_Data),
      .address(exmem_out_result),
      .memorywrite(MEMEWRITE),
```

```
      .clk(clk),
      .memoryread(MEMREAD),
      .read_data(readdata),
      .element1(element1),
      .element2(element2),
      .element3(element3),
      .element4(element4),
      .element5(element5),
      .element6(element6),
      .element7(element7),
      .element8(element8));
    twox1Mux mux2
    (.A(adderout1),.B(exmem_out_adder),.SEL(BRANCH &
branch_final),.Y(pc_in));
    MEMWB i4
    (.clk(clk),.reset(reset),.read_data_in(readdata),
.result_alu_in(exmem_out_result),.Rd_in(exmemrd),.memtoreg_in(MEMTOREG),.r
egwrite_in(REGWRITE),.readdata(muxin1),.result_alu_out(muxin2),.rd(memwbrd
),.Memtoreg(memwb_memtoreg),.Regwrite(memwb_regwrite));
     twox1Mux mux3
    (
      .A(muxin2),.B(muxin1),.SEL(memwb_memtoreg),.Y(write_data));
     ForwardingUnit f1
    (
      .RS_1(RS1),.RS_2(RS2),.rdMem(exmemrd),
      .rdWb(memwbrd),.regWrite_Wb(memwb_regwrite),
      .regWrite_Mem(REGWRITE),
      .Forward_A(forwardA),.Forward_B(forwardB));
    branching_unit branc
(.funct3(funct4_out[2:0]),.readData1(M1),.b(alu_64_b),.addermuxselect(adde
rmuxselect));
endmodule
```

- RTL Design for Pipelined 5 Stage RISK - V ISA Processor

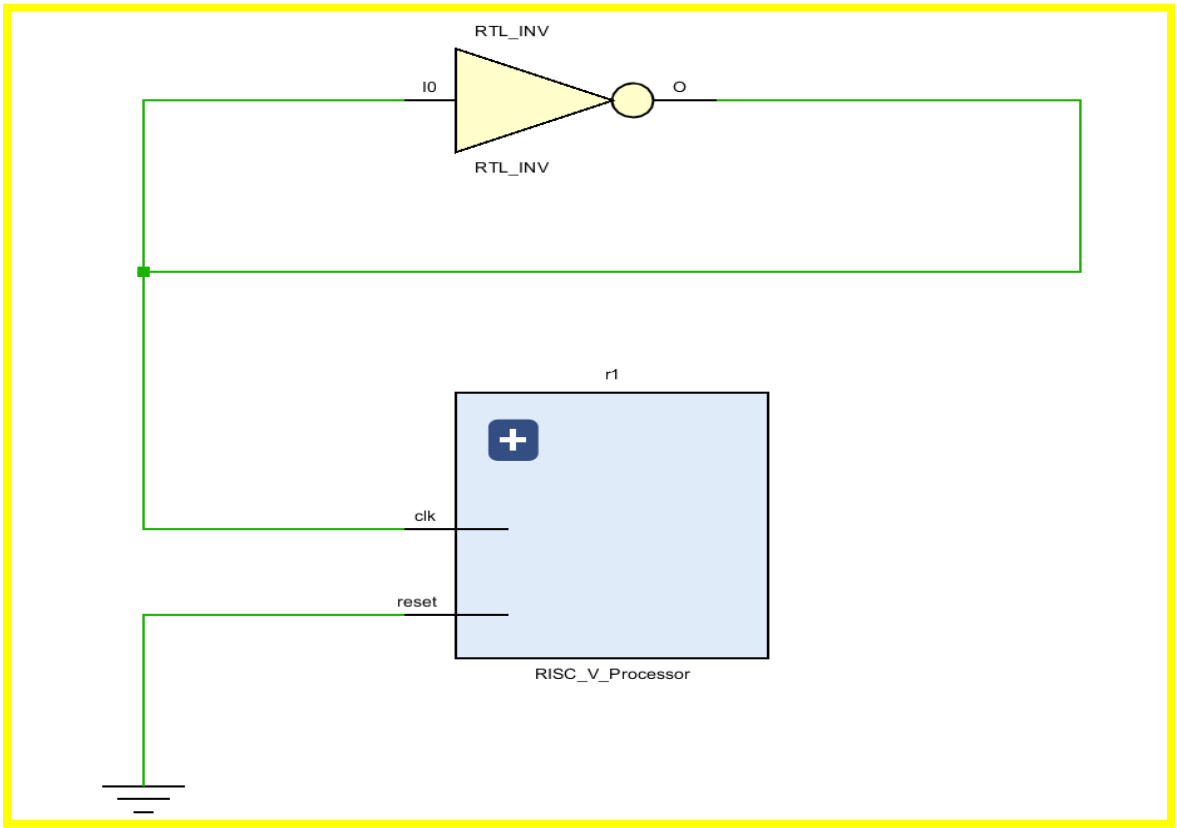


Fig 3.7 – Processor, Clock and Reset RTL Design

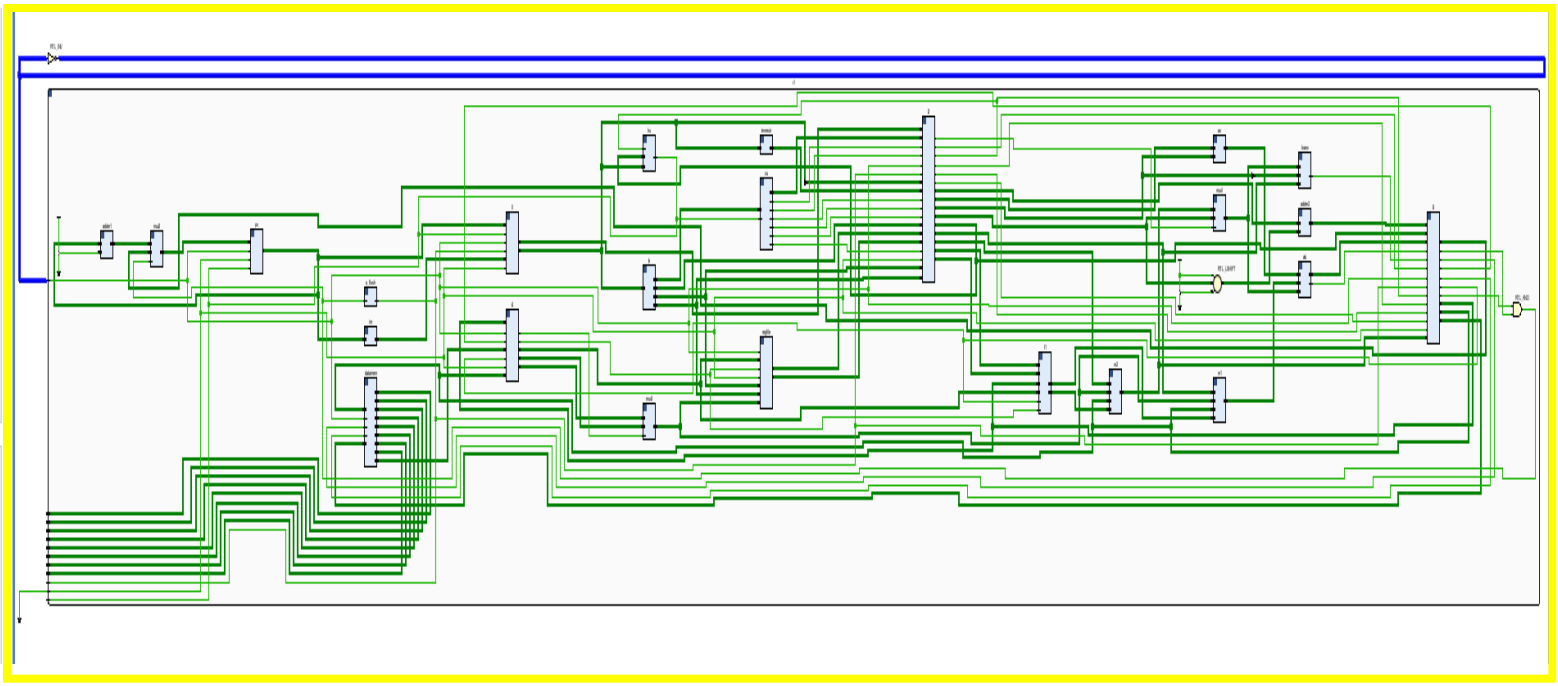


Fig 3.8 – RTL Design for 5 Stage RISC V ISA Pipelined Processor

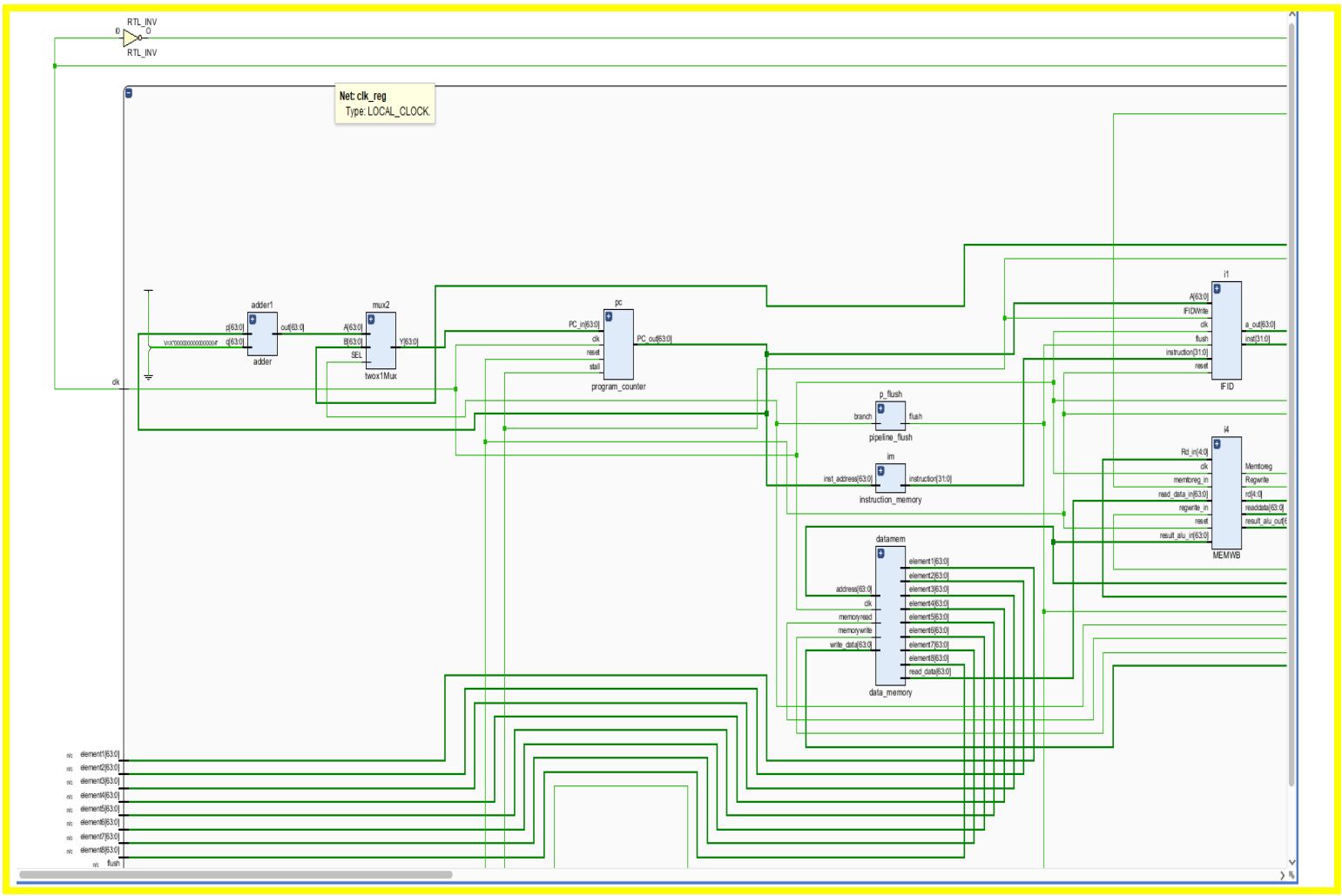


Fig 3.9 – RTL Design for IF Unit

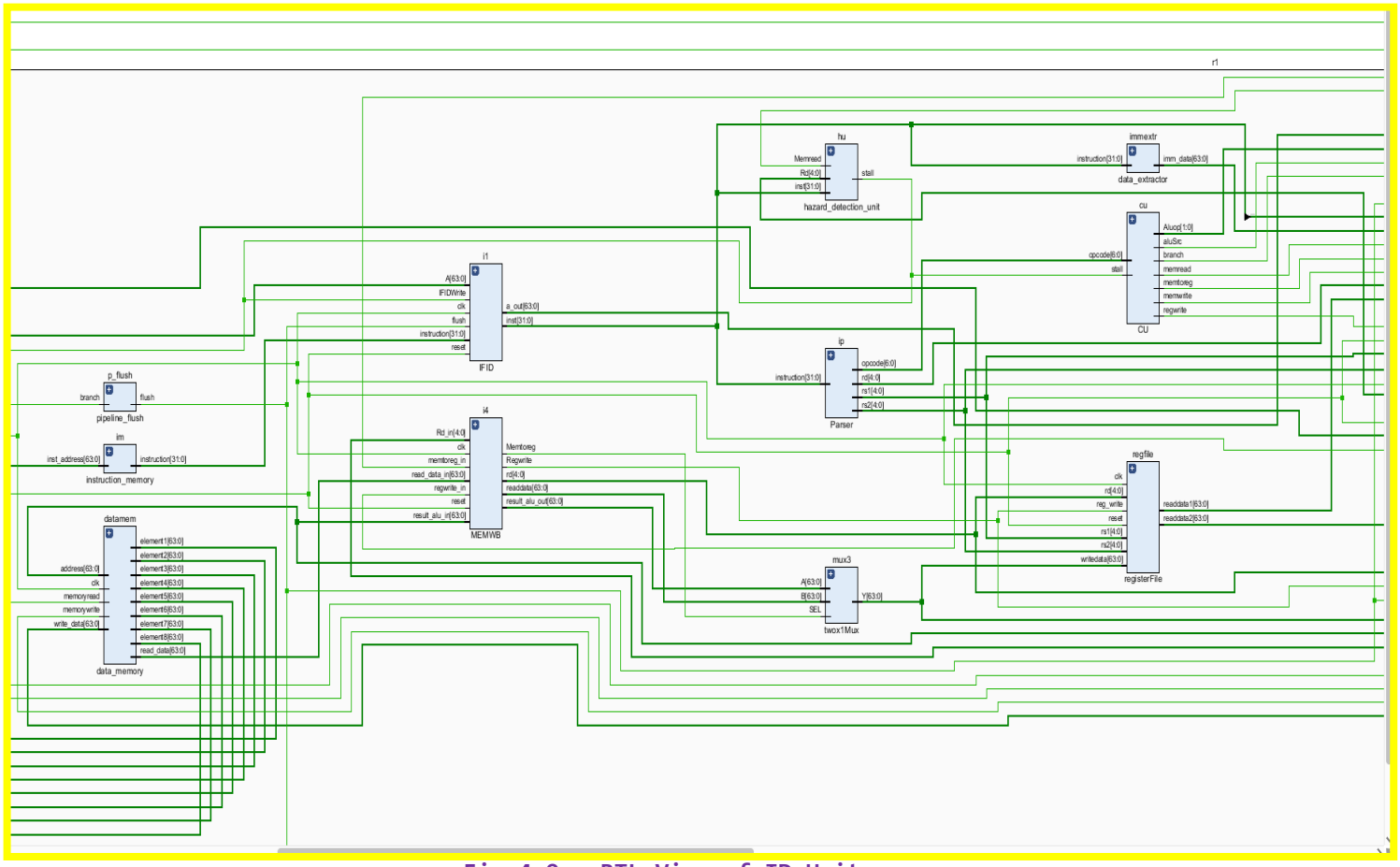


Fig 4.0 – RTL View of ID Unit

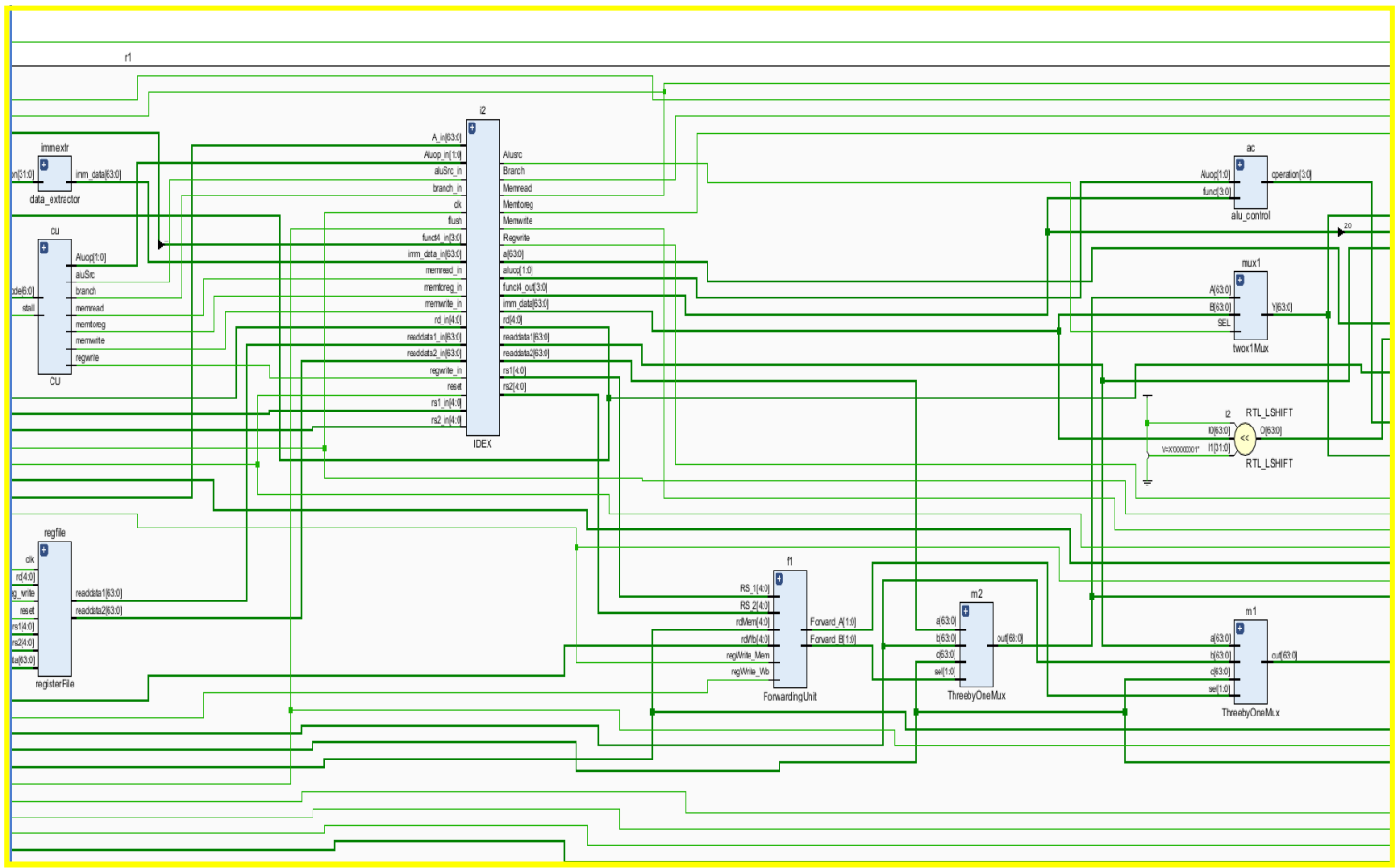


Fig 4.1 – RTL View for EX Unit

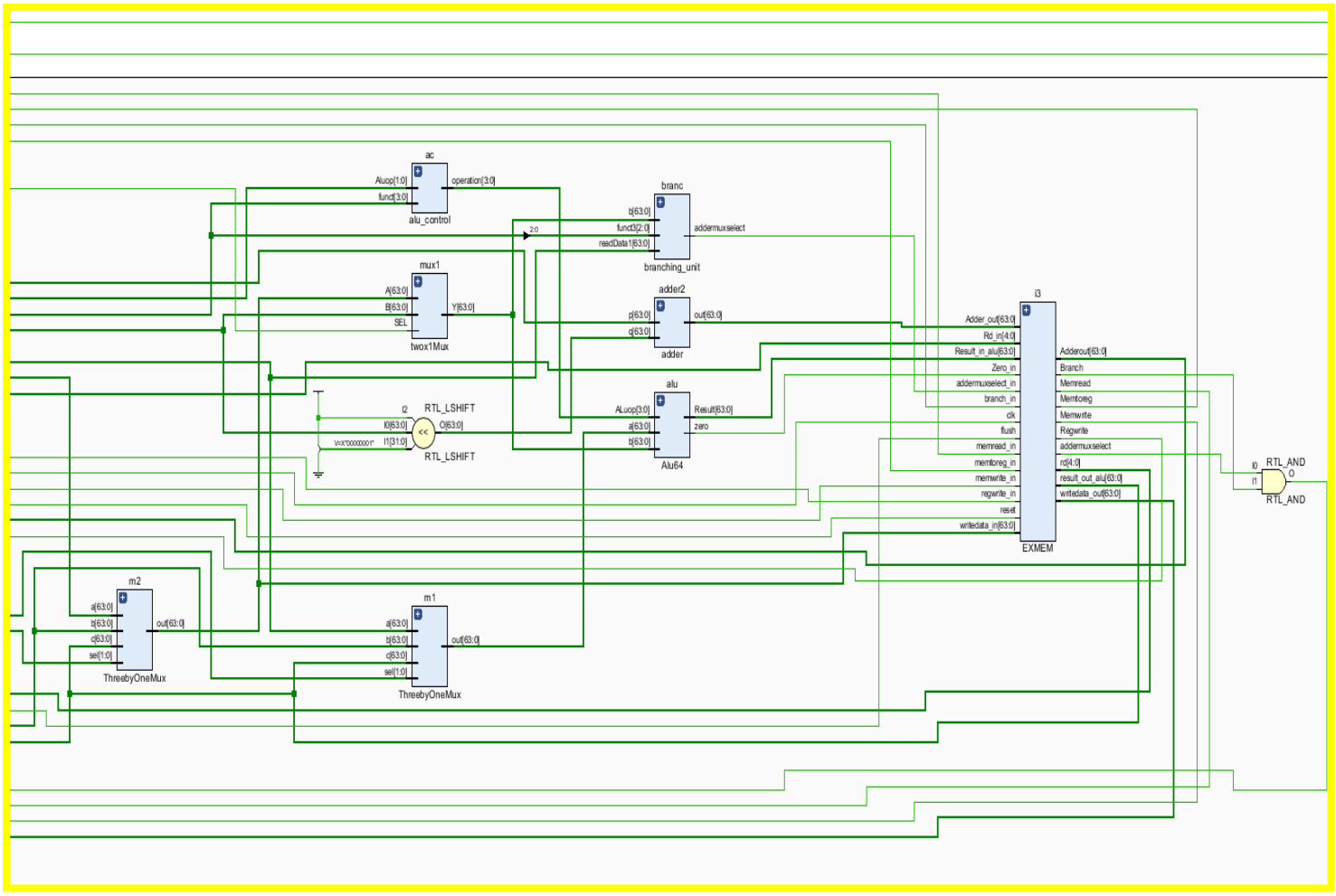


Fig 4.2 – RTL View of EX Unit

- **Code Explanation in depth :-**

➢ Modules and their Roles

1. Program Counter (PC) and Adder

- **Program Counter (`program_counter`)**: Keeps track of the address of the next instruction to be fetched. It increments the address after each instruction fetch unless stalled or flushed.
- **Adder**: Calculates the address of the next instruction (`pc_out + 4`). This is used to determine the next instruction address unless a branch is taken.

2. Instruction Memory

- **Instruction Memory (`instruction_memory`)**: Stores the instructions and provides them based on the address provided by the PC. It outputs the current instruction (`instruction`).

3. IF/ID Pipeline Register

- **IFID Register (`IFID`)**: Stores the instruction fetched from the instruction memory and the PC value for use in the decode stage. It helps in passing the instruction and PC value to the ID stage.

4. Instruction Parser

- **Instruction Parser (`Parser`)**: Extracts fields from the fetched instruction, such as opcode, rd, rs1, rs2, funct3, and funct7. This is crucial for determining the type of instruction and the required operations.

5. Control Unit

- **Control Unit (`CU`)**: Generates control signals based on the opcode from the parser. These signals include branch control, memory read/write, ALU source, register write enable, and ALU operation.

6. Immediate Data Extractor

- **Immediate Data Extractor (`data_extractor`)**: Extracts immediate values from the instruction, which are used for operations like branch offsets and immediate operands in ALU instructions.

7. ID/EX Pipeline Register

- **IDEX Register (`IDEX`)**: Stores decoded instruction fields, register values, and control signals for the execution stage. This register ensures that the pipeline stages are properly isolated and data is passed between stages.

8. Forwarding Unit

- **Forwarding Unit (`ForwardingUnit`)**: Resolves data hazards by forwarding data from the EX or WB stage to the current stage if needed. This prevents incorrect data from being used in computations.

9. ALU and ALU Control

- **ALU (`alu_64bit`)**: Performs arithmetic and logical operations. Takes two inputs and the operation code to compute a result.
- **ALU Control (`alu_control`)**: Decodes the ALU operation required based on the control signals and instruction funct fields.

10. EX/MEM Pipeline Register

- **EXMEM Register (`EXMEM`)**: Stores the results of the ALU computation, memory address, and control signals for the MEM stage. It also stores branch decisions.

11. Data Memory

- **Data Memory (`data_memory`)**: Provides read and write access to data memory. Handles data memory operations based on the address and control signals.

12. MEM/WB Pipeline Register

- **MEMWB Register (`MEMWB`)**: Holds the results from the memory stage and the ALU result for writing back to the register file. This register decides which data (memory read data or ALU result) will be written to the register file.

13. Register File

- **Register File (`reg_file`)**: Stores general-purpose registers. It reads and writes register values based on the signals from the WB stage.

14. Branching Unit

- **Branching Unit (`branching_unit`)**: Determines if a branch should be taken and calculates the branch target address.

15. Hazard Detection Unit

- **Hazard Detection Unit (`hazard_detection_unit`)**: Detects data hazards and inserts stalls as necessary to prevent incorrect execution due to data dependencies.

16. Pipeline Flush

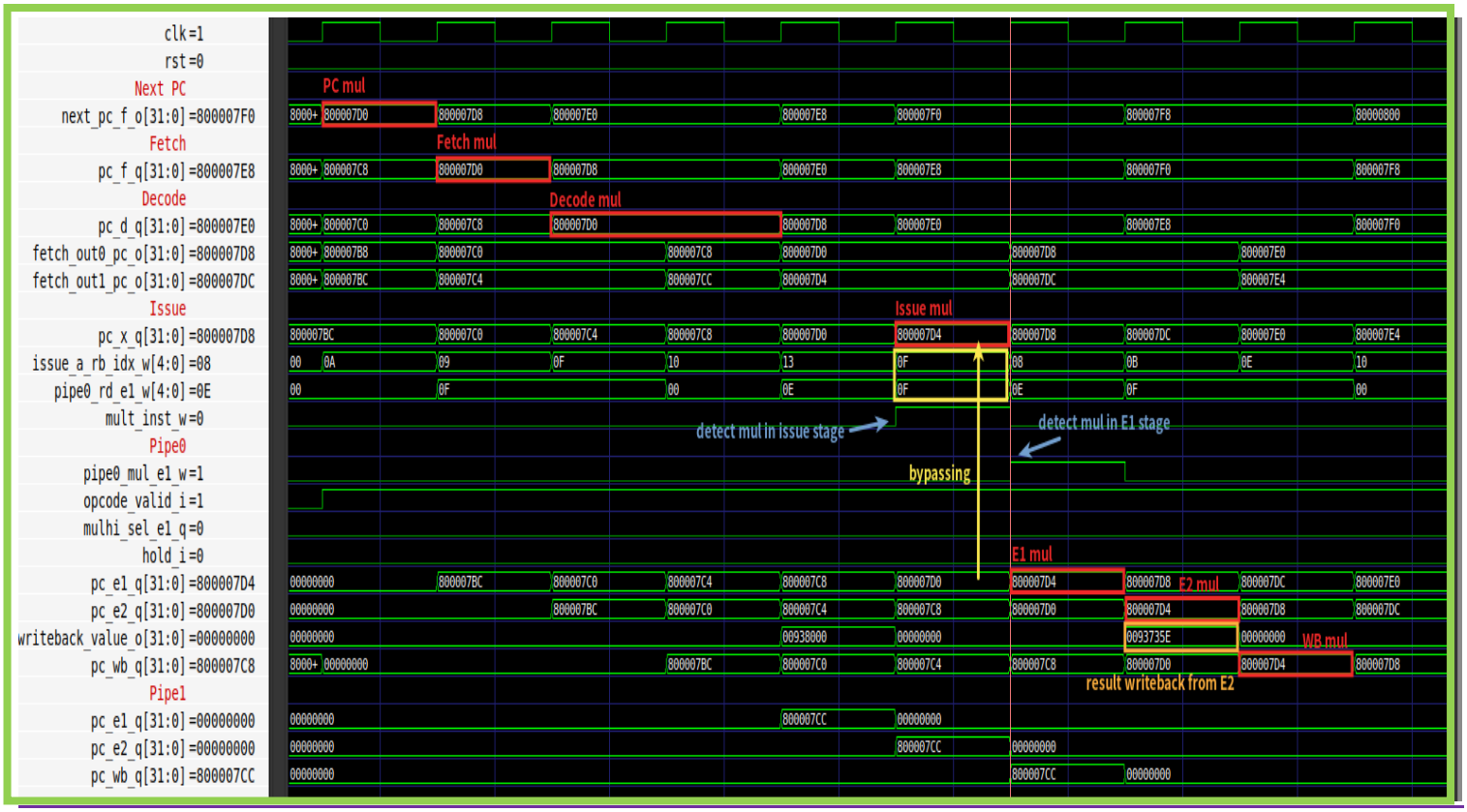
- **Pipeline Flush (`pipeline_flush`)**: Ensures that instructions in the pipeline are discarded if a branch is taken, thus maintaining correct execution.

Pipeline Execution Cycle

Here's how the processor operates over multiple clock cycles:

1. **Instruction Fetch (IF) Stage**
   - The PC outputs the address of the current instruction.
   - The instruction memory fetches the instruction from the address given by the PC.
   - The PC is updated to point to the next instruction (or to the branch target if a branch is taken).
2. **Instruction Decode (ID) Stage**
   - The fetched instruction is decoded to extract the opcode, source and destination registers, and immediate values.
   - The control unit generates control signals based on the decoded instruction.
   - Register values are read from the register file.
   - Immediate values are extracted if needed.
3. **Execution (EX) Stage**
   - The ALU performs operations based on the control signals and operands (register values or immediate values).
   - Forwarding units may provide correct data if there are data hazards.
   - Branching decisions are made based on the ALU results.
4. **Memory Access (MEM) Stage**
   - Data memory operations are performed if needed (read or write).
   - The address for memory operations is computed and the data read from or written to memory.
5. **Write-Back (WB) Stage**
   - The result from the ALU or data memory is written back to the register file.
   - The `MEMWB` register decides which result (ALU or memory data) will be written back based on control signals.

❖ Xilinx Test Bench Results



❖ Schematic Representation of Processor

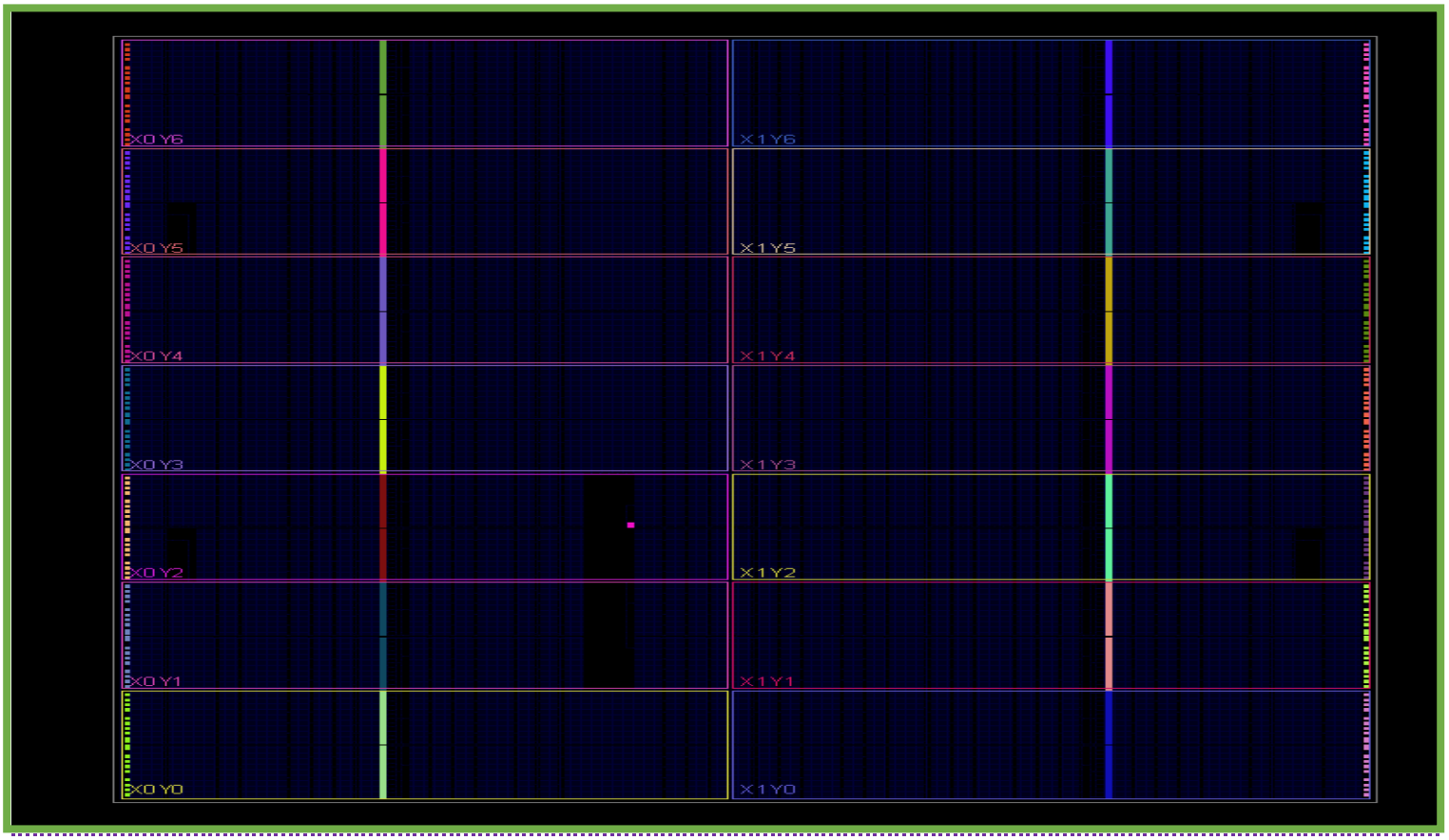


Fig 4.3 – Vivado Software Results for Designed Processor