RISC V Processor Non - Pipelined Implementation using Verilog

Done By Naman Kalra, B Tech Second Year Student, Under Prof. Vikramkumar Pudi Sir

32 Bits RISC V ISA Processor Implementation Thesis

Made By Naman Kalra, EE23B032

Project will progress in following steps

Non Pipelined -> Pipelined -> Algorithms to solve Pipelined hazards -> Vector Processors -> Processor Extension

- ***Non Pipelined Implementation of RISC V Processor***

- Overall Pictorial Representation of Processor :-

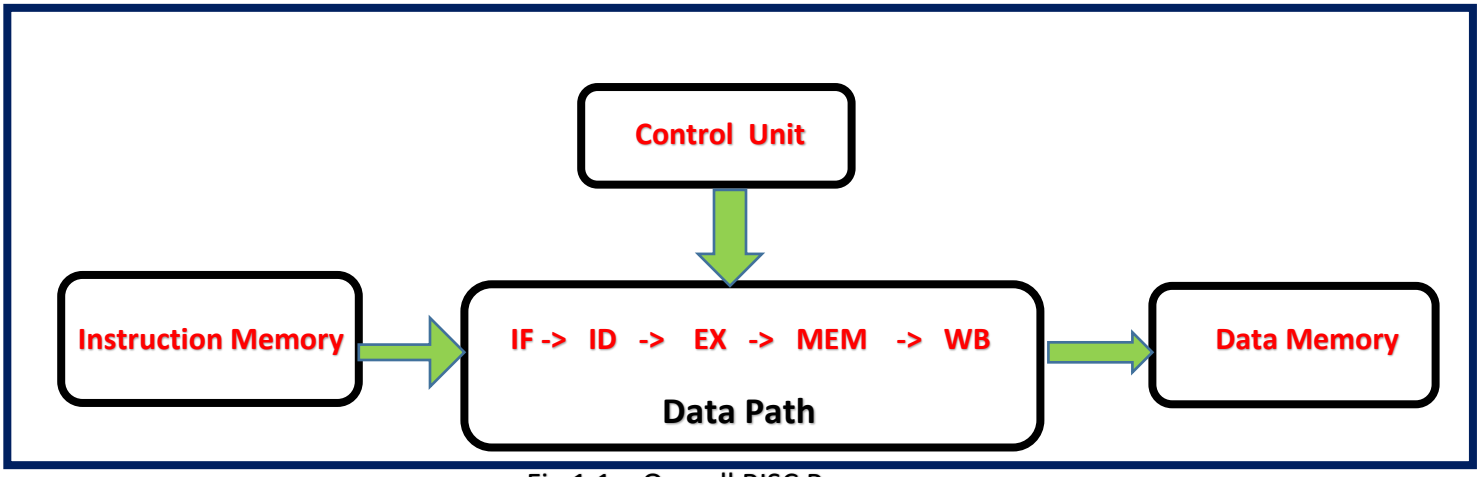


Fig 1.1 – Overall RISC Processor

- **Control Unit**: Decodes the instruction from the instruction memory and generates control signals to direct the operations of the data path components (e.g., ALU, registers).

- **Data Path**: Executes the instructions by performing arithmetic and logic operations using the ALU, managing data flow between registers and the data memory based on control signals. It consist of various blocks i.e. IF, ID, EX, MEM, WB to complete the instruction.

- **Instruction and Data Memory**: The instruction memory stores the program instructions, while the data memory holds data values; the control unit fetches instructions from instruction memory, and the data path reads/writes data to/from data memory based on executed instruction

- Hierarchy Design of Memory :- In order to save time, we add cache memory in which the recently used data is stored.

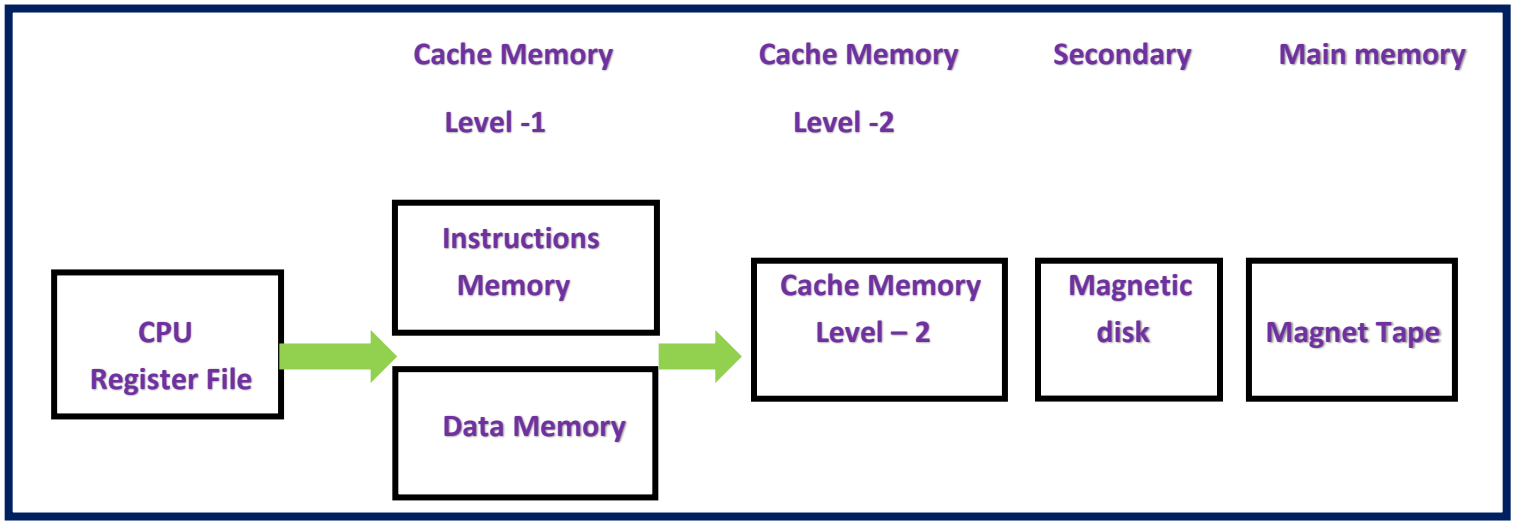


Fig 1.2 - Hierarchy design flow chart for memory

- Memory Interface with CPU :-

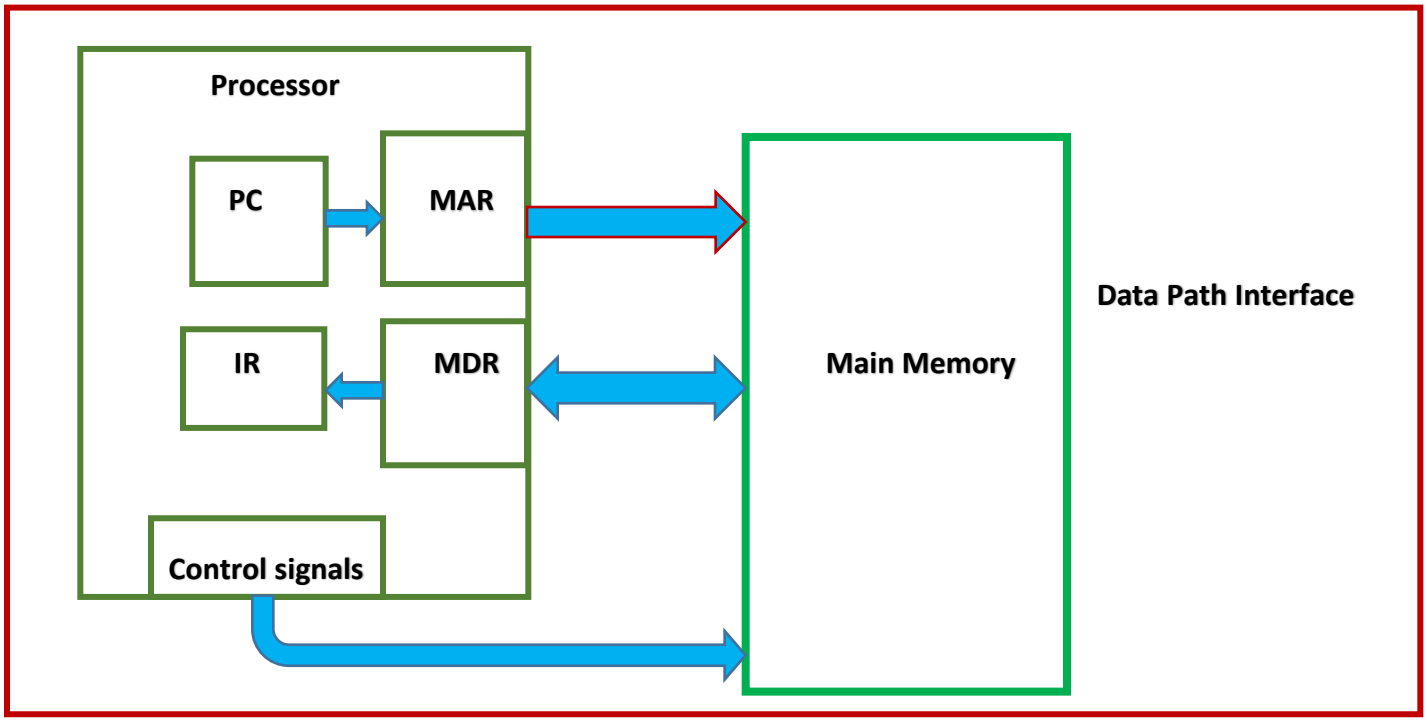


Fig 1.3 - Interface between memory and CPU

➢ Working Flow :-

1. **PC to MAR**: The PC holds the address of the next instruction to be fetched. During instruction fetch, the address in the PC is placed into the MAR.
2. **MAR to Memory**: The MAR sends the address to the memory. Memory uses this address to read the instruction or data from the corresponding location.
3. **Memory to MDR**: The data read from memory is loaded into the MDR. For instructions, this data is an instruction code; for data operations, it's the data to be processed.
4. **MDR to IR**: The content of the MDR, which is the instruction code, is loaded into the Instruction Register (IR) for decoding and execution.
5. **IR to Processor**: The processor decodes the instruction in the IR to determine the required operations and generates appropriate control signals for the data path.
6. **Processor to Registers**: The processor uses control signals to read from or write to registers as specified by the decoded instruction, affecting computation or data storage.
7. **Control Signals**: These signals coordinate the operation of the entire system, including loading addresses into MAR, reading from/writing to memory, transferring data between MDR and IR, and performing operations in the processor and registers.

In summary, the PC provides the address, MAR accesses memory, MDR transfers data, IR holds the instruction, and the processor executes the instruction while control signals manage the entire process.

➢ Implementation of Design Blocks and Verilog codes :-

a) ***Instruction Fetch Unit / IF*** *–> This is the first block in data path used to fetch instruction from Memory by accessing the address stored in PC. Also increase PC to next instruction just by adding 4 and then using Multiplexer selects new PC value between branch or increased PC.*

1. Program Counter (PC)

- **Circuit**: The PC is a register that holds the address of the next instruction to be fetched.
- **Action**: The PC outputs the current address to be used for fetching the instruction from memory.

2. PC to Memory Addressing

- **Circuit**: Directly connects the output of the PC to the memory address lines.
- **Action**: The address held in the PC is used to access the memory and fetch the instruction.

3. Memory Read Operation

- **Circuit**: Memory unit connected to the address lines.
- **Action**: Memory retrieves the instruction stored at the address provided by the PC and places it on the data bus.

4. Memory to Instruction Register (IR)

- **Circuit**: The output of the memory unit is connected to the IR.
- **Action**: The instruction fetched from memory is loaded into the Instruction Register (IR):
    - **IR Load Signal**: A control signal loads the instruction data from memory into the IR.

5. Increment Program Counter (PC)

- **Circuit**: An adder is used to compute the address of the next instruction.
- **Action**: The PC is incremented by the size of the instruction (e.g., 4 bytes):
    - **Adder**: Adds the instruction size (4 bytes) to the current PC value to generate the address for the next instruction.

6. Multiplexer for Next PC Selection

- **Circuit**: A multiplexer selects between the incremented PC address and a branch target address (if applicable).
- **Action**:
    - **Adder Output**: Provides the incremented PC value.
    - **Branch Target Address**: If the current instruction is a branch, the branch target address is used.
    - **MUX Selection Signal**: Controls whether to select the incremented PC value or the branch target address.

7. Update PC

- **Circuit**: PC register is updated based on the output of the multiplexer.
- **Action**: The selected address (either the incremented PC or the branch target) is loaded back into the PC for the next fetch cycle.

- ***Circuit of IF block :-***

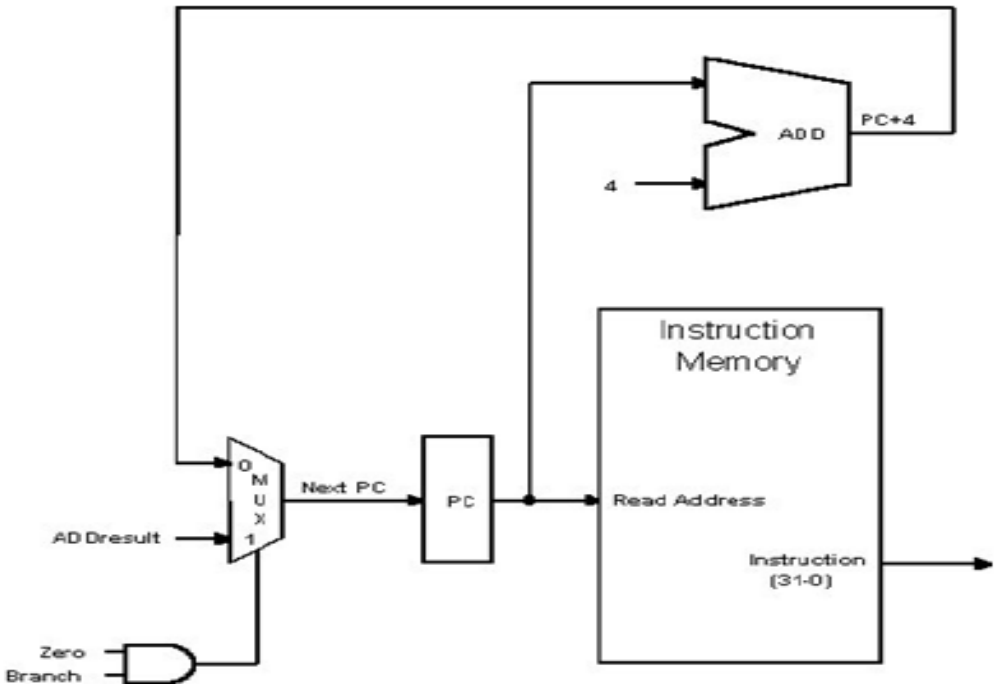


Fig 1.4 – Block circuit of IF Stage

- *Verilog Code for Standard IF unit for simulation in Xilinx Vivado Software :-*

```
module IFU (
input wire clock,
// Clock signal
input wire reset,
// Reset signal
input wire branch_taken,
// Signal indicating if a branch is taken
input wire [31:0] branch_target,
// Branch target address
output reg [31:0] Instruction_Code  );
// Output instruction code from memory

reg [31:0] PC;
// 32-bit program counter
reg [31:0] next_pc;
// Next PC value to be selected by MUX

INST_MEM instr_mem (
// Instantiate the instruction memory module
.Address(PC),
// Connect PC to memory address input
.Reset(reset),
// Connect reset signal to memory
.Instruction_Code(Instruction_Code)
// Output instruction code
);

always @* begin
if (branch_taken) begin
next_pc = branch_target;
// Select branch target if branch is taken
end else begin
next_pc = PC + 4;
// Otherwise, increment PC by 4
```

```
end
end

// Program counter update logic
always @(posedge clock or posedge reset) begin
if (reset) begin
PC <= 32'b0;
// On reset, initialize PC to zero
end else begin
PC <= next_pc;
// Update PC with the selected next address
end
end
endmodule
```

Simulation Design Result in Xilinx :-

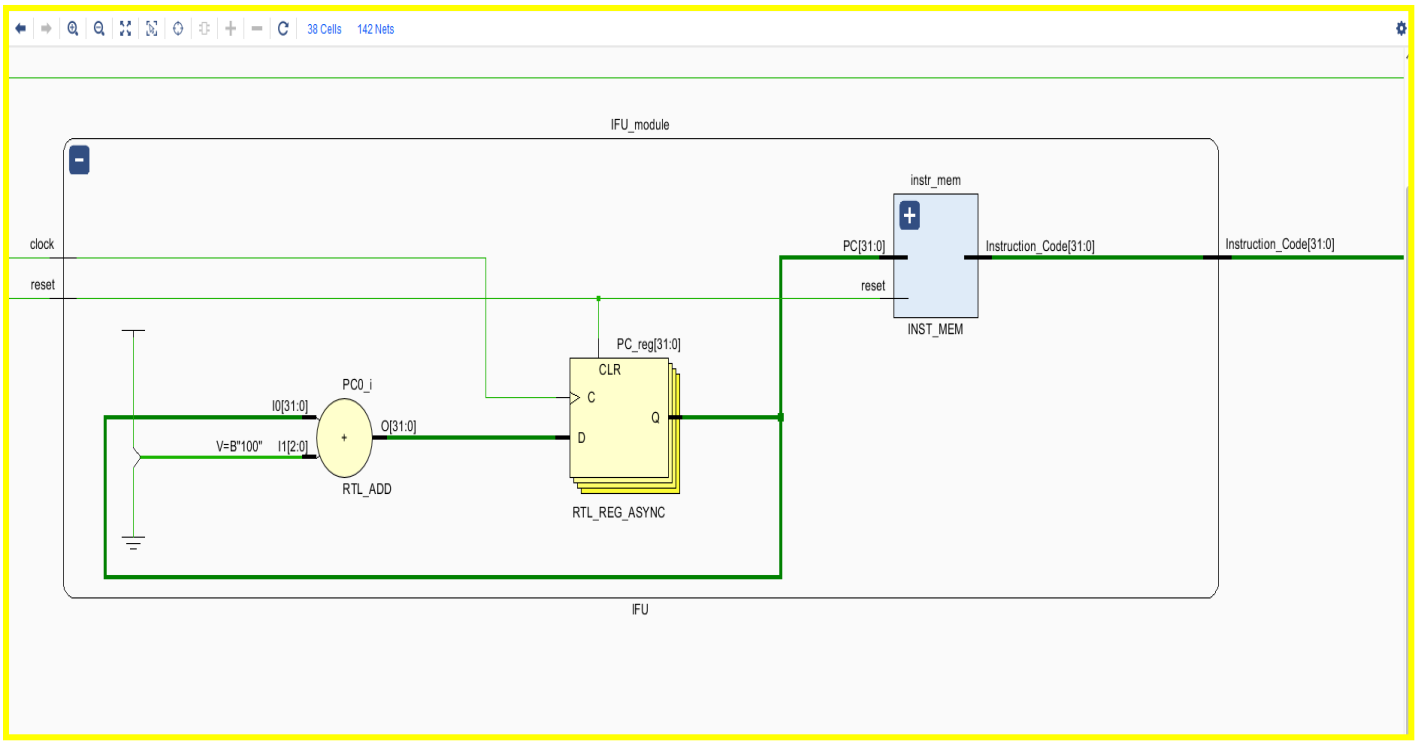


Fig 1.5 - RTL Elaborated Design of IF Unit (By Naman Kalra, EE23B032)

b) Instruction Decode Block / ID :-

1. Instruction Input

- The ID stage receives the instruction from the Instruction Register of Instruction Fetch (IF) stage. This instruction is generally a 32-bit value, which includes various fields such as the opcode, source registers, destination registers, immediate values, and function codes.

2. Instruction Fields Extraction

- **Opcode**: Determines the type of instruction and which operation to perform. There are various types of instruction in RISK ISA which are mentioned below :-

- **R-Type**: Operations on registers (arithmetic, logical, shift).
- **I-Type**: Operations with immediate values and memory access (load/store, branch).
- **J-Type**: Jump operations (unconditional jumps, jumps with link).

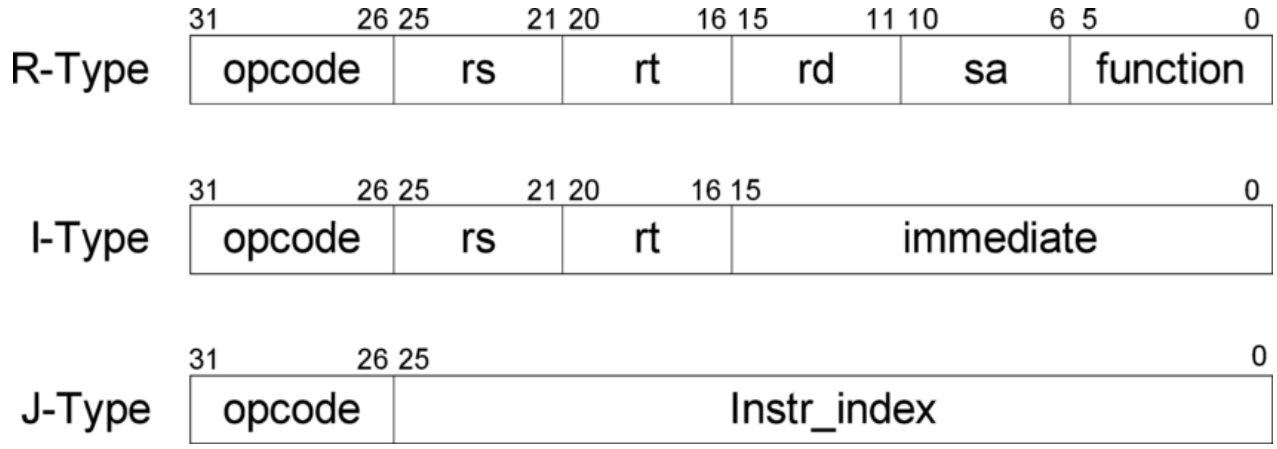


Fig 1.6 – Instruction Types of RISK ISA

- **Source Registers (rs1, rs2)**: Identifies the registers to be read for operands.
- **Destination Register (rd)**: Specifies which register will receive the result of the operation.
- **Immediate Value**: Provides a constant value used in the instruction.
- **Function Code (funct3, funct7)**: Specifies the exact variant of the operation, especially for instructions with similar opcodes.

3. Control Signal Generation

- Based on the opcode and function codes, the ID stage generates control signals for the execution stage. These control signals may include:
  - **ALU Operation**: Specifies the operation to be performed by the ALU (Arithmetic Logic Unit).
  - **Register File Read/Write**: Signals to read from or write to the register file.
  - **Branch Control**: Indicates if the instruction involves a branch or jump.
  - **Memory Access**: Determines if the instruction requires memory read/write operations.

4. Register File Access

- **Read Registers**: The ID stage uses the source register addresses (rs1 and rs2) to read data from the register file.
- **Prepare Data**: The data from these registers is prepared for the execution stage, along with any immediate values.

5. Immediate Value Handling

- For instructions that use an immediate value, the ID stage extracts and sign-extends this value as necessary. Immediate values are often used for operations like addition or subtraction with constants.

6. Branch Prediction and Address Calculation

- For branch instructions, the ID stage may calculate the target address and perform branch prediction based on the opcode and immediate value. This information is used by the control unit to decide if a branch should be taken.

❖ Instruction Decoder Block Circuit

➢ Interface Between Instruction Stage Fetch and ALU :-

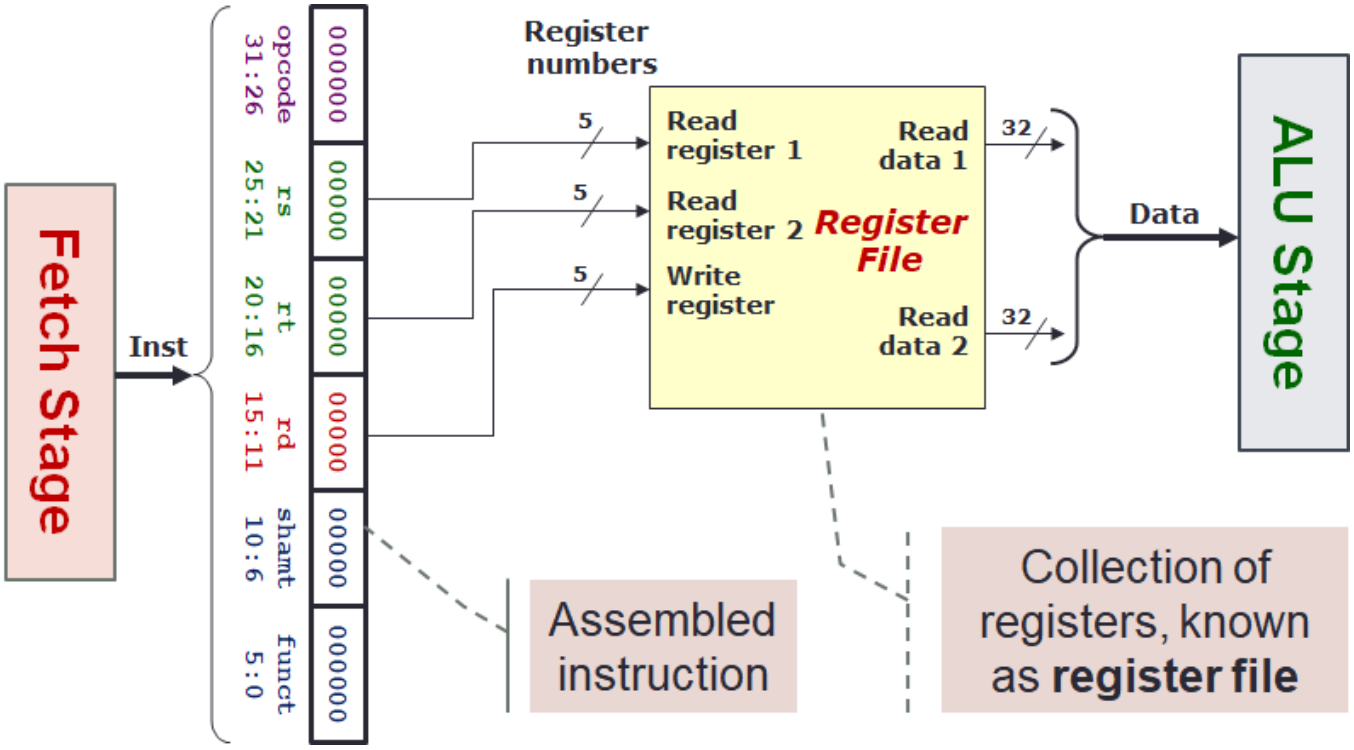


Fig 1.7 - Interface Between Fetch and ALU

➢ Interface between Control Unit and Data Path :-

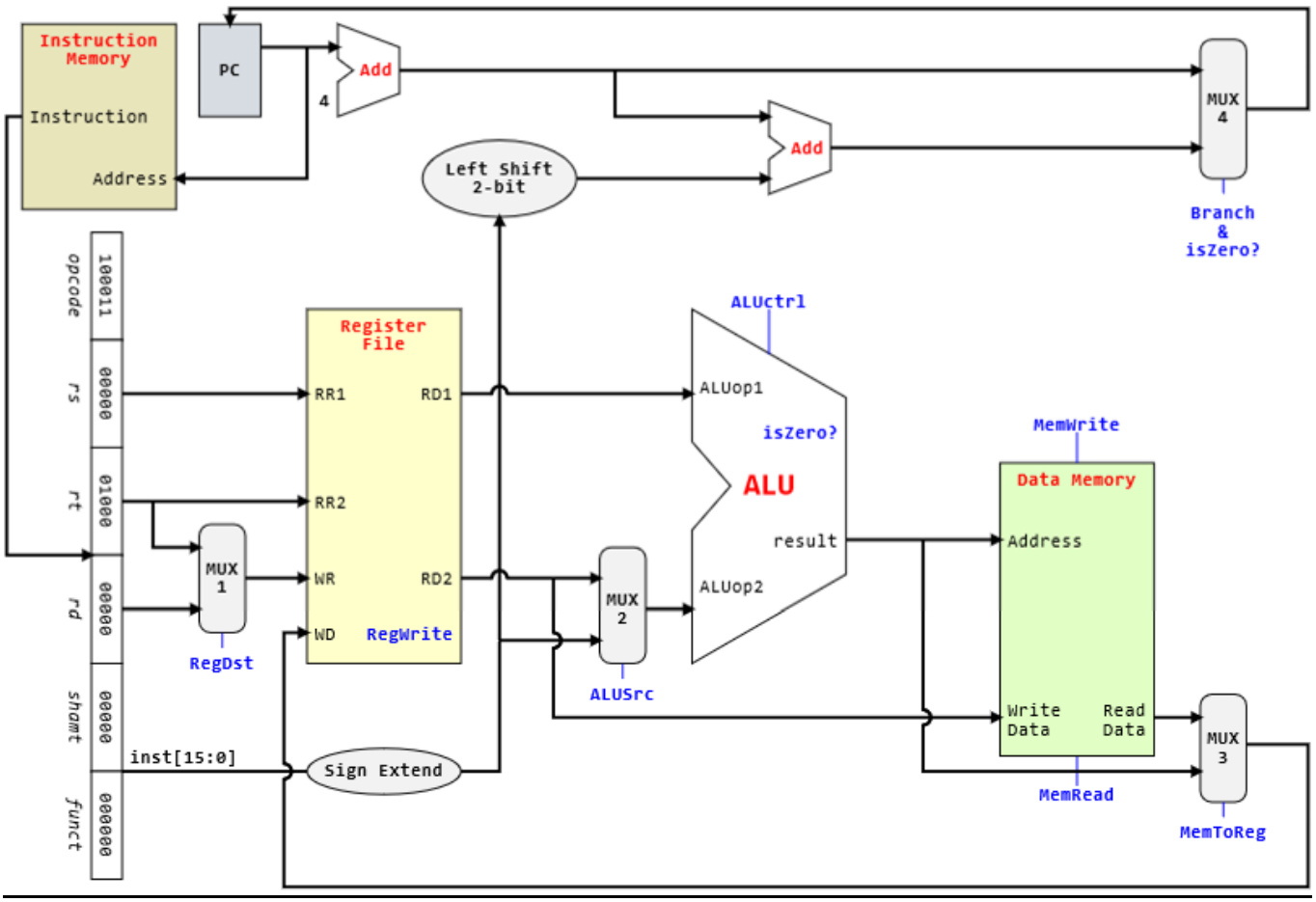


Fig 1.8 - Interface between Control Unit and Data Path

❖ MUX for Selecting the Next PC Value

Inputs to the MUX:

1. **PC + 4**: The address of the next sequential instruction.
2. **Branch Target Address**: The target address if a branch is taken.

Branch Instruction Handling:

1. **Branch Condition Check**:
   - The `Zero` flag is used to determine if the branch condition is met (e.g., in the case of a `BEQ` instruction, `Zero` is set if two registers are equal).
   - The Control Unit uses the `Zero` flag to set the `Branch_taken` signal, which indicates whether the branch should be taken based on the branch condition.
2. **Control Signals**:
   - **Branch**: Indicates whether the current instruction is a branch instruction. If this signal is asserted, it implies that a branch instruction is in progress.
   - **Branch_taken**: Set if the branch condition (checked using `Zero` or other condition checks) is satisfied.

3. **MUX Selection**:
    - **Branch Signal**: The `Branch` signal tells the MUX whether the instruction is a branch.
    - **Branch_taken Signal**: Determines if the branch should be taken based on the condition (e.g., `Zero` flag).

- **Branch Target Address** if both `Branch` is asserted and `Branch_taken` is true (the branch condition is met).
- **PC + 4** if the `Branch` is not asserted or the branch condition is not met.

- **Branch Offset Field**: Specifies a number of instructions to jump, not bytes.
- **Shift Left by 2**: Converts the instruction offset into bytes (because each instruction is 4 bytes).
- **Add to PC + 4**: To get the target address relative to the instruction that follows the branch, ensuring correct branching

➢ Verilog Code for various blocks

• **Instruction Fetch (IF) Stage**: Fetches instructions and updates the PC.

• **Instruction Decode (ID) Stage**: Decodes the instruction, generates control signals, and prepares signals for the ALU.

• **Control Unit**: Generates control signals based on the instruction opcode and function codes.

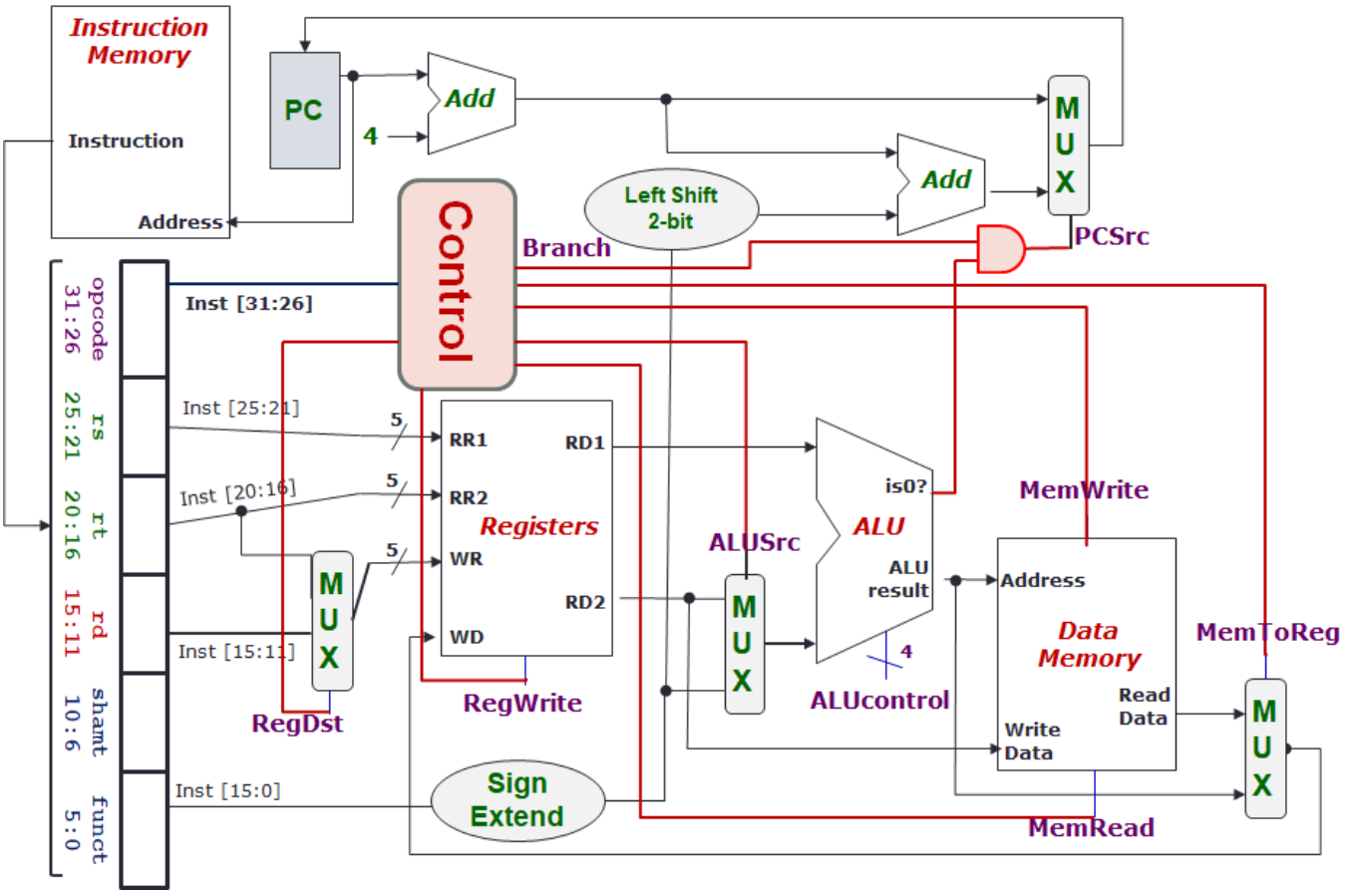


Fig 1.9 – Detailed Representation of Data Path of RISC-V ISA Processor

❖ *Types of Control Signals-*

1. **ALUSrc (ALU Source)**
   - **Function**: Determines the source of the second operand for the ALU.
   - **Value**:
     - `0`: The second operand is taken from a register (for R-type instructions).
     - `1`: The second operand is an immediate value (for I-type instructions).
2. **MemRead (Memory Read)**
   - **Function**: Controls whether the processor reads data from memory.
   - **Value**:
     - `0`: No memory read operation.
     - `1`: Read data from memory.
3. **MemWrite (Memory Write)**
   - **Function**: Controls whether the processor writes data to memory.
   - **Value**:
     - `0`: No memory write operation.
     - `1`: Write data to memory.
4. **RegWrite (Register Write)**
   - **Function**: Controls whether data is written to the register file.
   - **Value**:
     - `0`: No write to register.
     - `1`: Write data to the specified register.
5. **Branch**
   - **Function**: Controls whether a branch operation should be taken.
   - **Value**:
     - `0`: No branch.
     - `1`: Perform a branch if the condition is met.
6. **Jump**
   - **Function**: Controls whether a jump operation should be performed.
   - **Value**:
     - `0`: No jump.
     - `1`: Perform a jump to a new address.
7. **ALUOp (ALU Operation)**
   - **Function**: Specifies the operation that the ALU should perform.
   - **Value**: Encoded control signal to perform operations like addition, subtraction, logical AND, etc

❖ Control Path Verilog Code :-

```
module ControlUnit (
    input wire [5:0] opcode,        // Opcode from instruction
    input wire [5:0] funct,         // Function code (for R-type instructions)
    output reg ALUSrc,              // ALU source (0: Register, 1: Immediate)
    output reg MemRead,             // Memory read enable
    output reg MemWrite,            // Memory write enable
    output reg RegWrite,            // Register write enable
    output reg Branch,              // Branch enable
    output reg Jump,                // Jump enable
    output reg [3:0] ALUOp           // ALU operation code
);
    // Decode instruction and generate control signals
    always @(*) begin
        case(opcode)
            6'b000000: begin  // R-type instructions
                RegWrite = 1;

                ALUSrc = 0;
```

```
                MemRead = 0;
                MemWrite = 0;
                Branch = 0;
                Jump = 0;
                case(funct)
                    6'b100000: ALUOp = 4'b0010; // ADD
                    6'b100010: ALUOp = 4'b0110; // SUB
                    6'b100100: ALUOp = 4'b0000; // AND
                    6'b100101: ALUOp = 4'b0001; // OR
                    default:   ALUOp = 4'b0000; // Default to AND
                endcase
            end
            6'b100011: begin  // LW (Load Word)
                RegWrite = 1;
                ALUSrc = 1;
                MemRead = 1;
                MemWrite = 0;
                Branch = 0;
                Jump = 0;
                ALUOp = 4'b0010; // ADD
            end
            6'b101011: begin  // SW (Store Word)
                RegWrite = 0;
                ALUSrc = 1;
                MemRead = 0;
                MemWrite = 1;
                Branch = 0;
                Jump = 0;
                ALUOp = 4'b0010; // ADD
            end
            6'b000100: begin  // BEQ (Branch if Equal)
                RegWrite = 0;
                ALUSrc = 0;
                MemRead = 0;
                MemWrite = 0;
                Branch = 1;
                Jump = 0;
                ALUOp = 4'b0110; // SUB
            end
            6'b000010: begin  // J (Jump)
                RegWrite = 0;
                ALUSrc = 0;
                MemRead = 0;
                MemWrite = 0;
                Branch = 0;
                Jump = 1;
                ALUOp = 4'bxxxx; // Don't care for jump
            end
            default: begin
                // Default case
                RegWrite = 0;
                ALUSrc = 0;
                MemRead = 0;
                MemWrite = 0;
                Branch = 0;
                Jump = 0;
                ALUOp = 4'b0000;
            end
        endcase
    end
endmodule
```

❖ RTL Design of Control Path :-

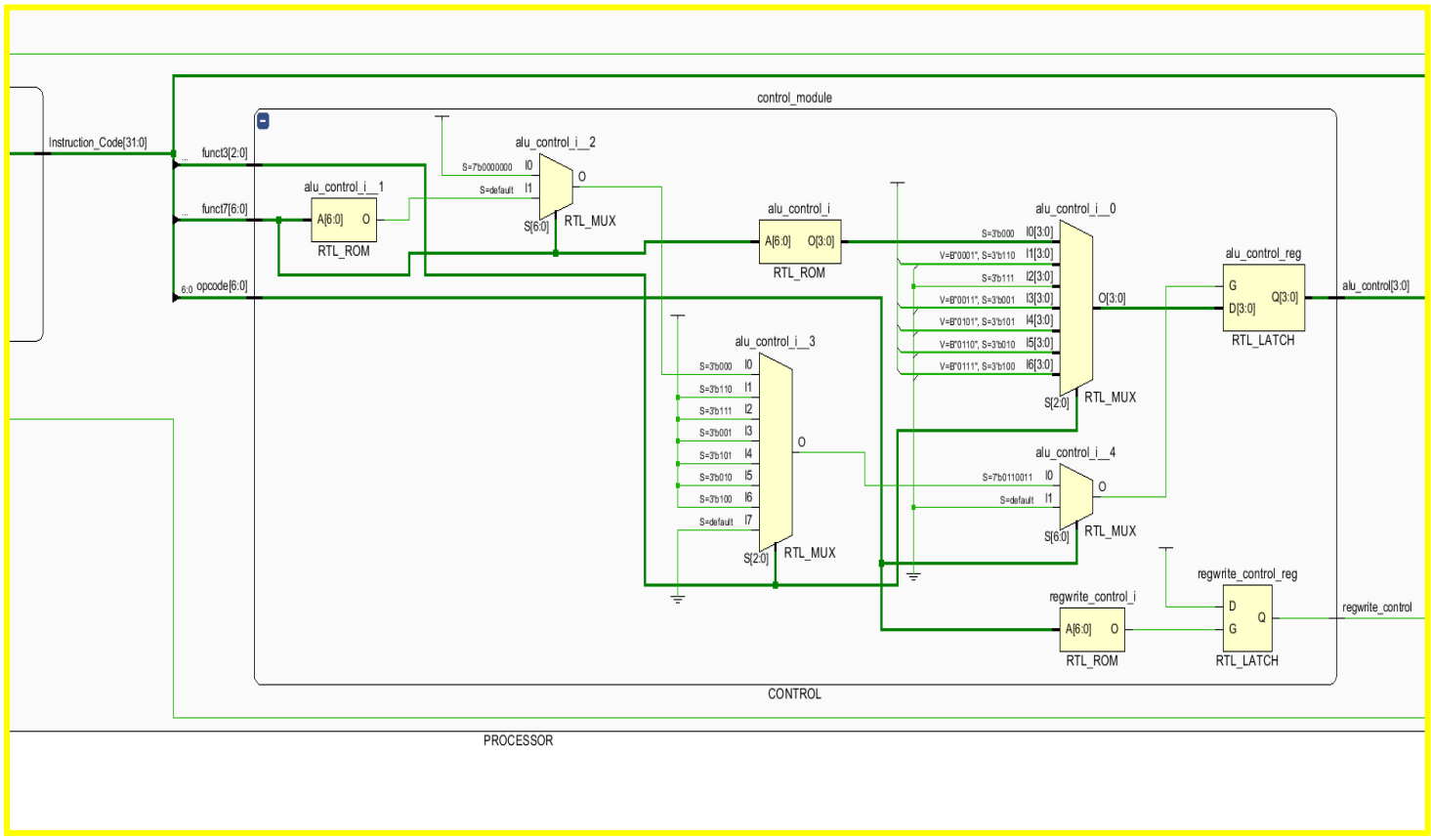


Fig 2.0 - RTL Design of Control Path

❖ Showing Interface between all modules by using Xilinx Software RTL Design

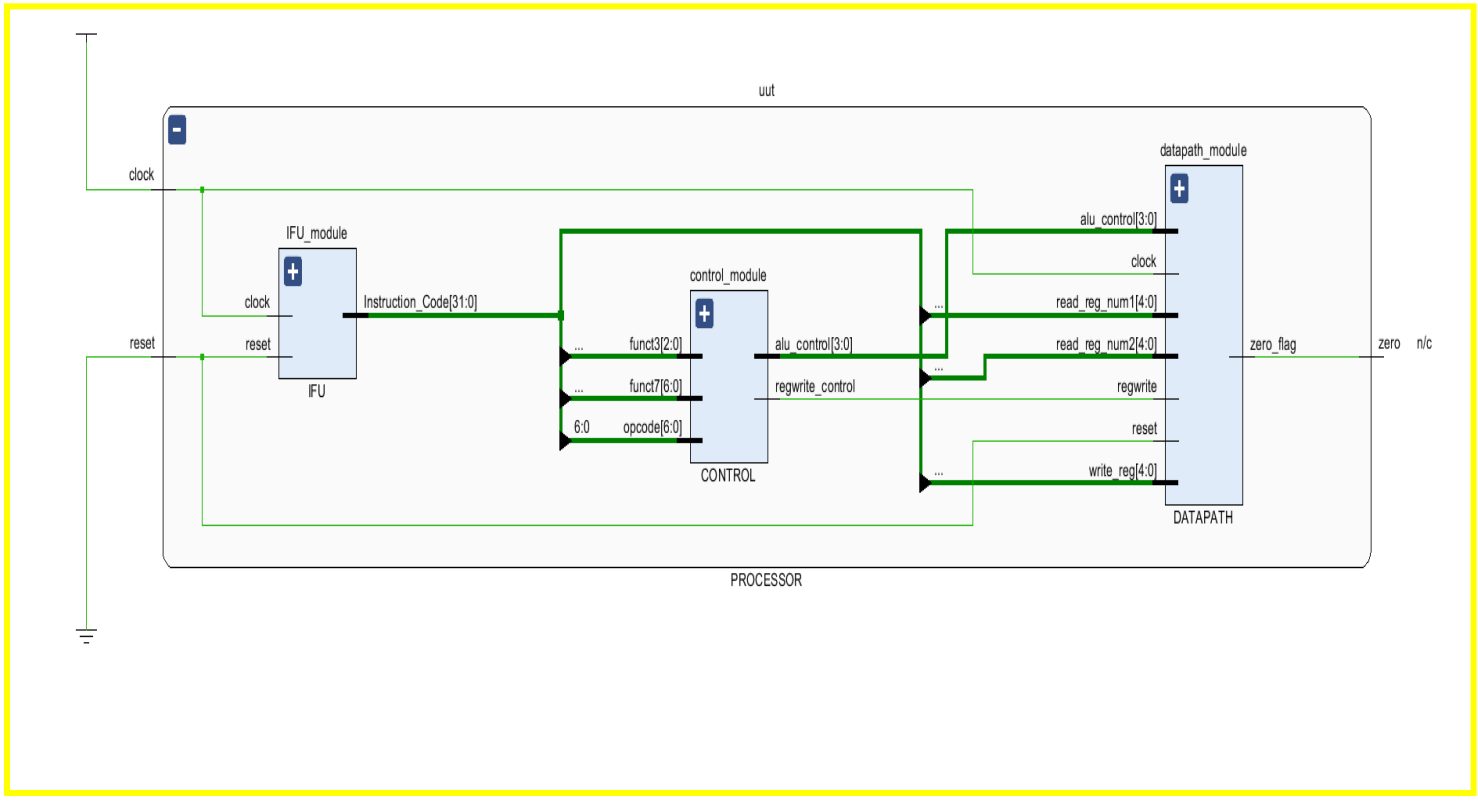


Fig 2.1 – Interface between IFU Module, Control Module and Data Path Module

c) Register File of Processor :-

• **General-Purpose Registers (GPRs)**: Used for typical arithmetic and logic operations, data transfer between instructions, and temporary storage.
• **Program Counter (PC)**: Directs the flow of execution by pointing to the next instruction to be executed.
• **Special Registers (CSRs)**: Manage processor-specific control and status operations, including handling exceptions and interrupts.
• **Floating-Point Registers**: Handle operations involving floating-point arithmetic.
• **Vector Registers**: Used for vectorized operations in parallel computing.

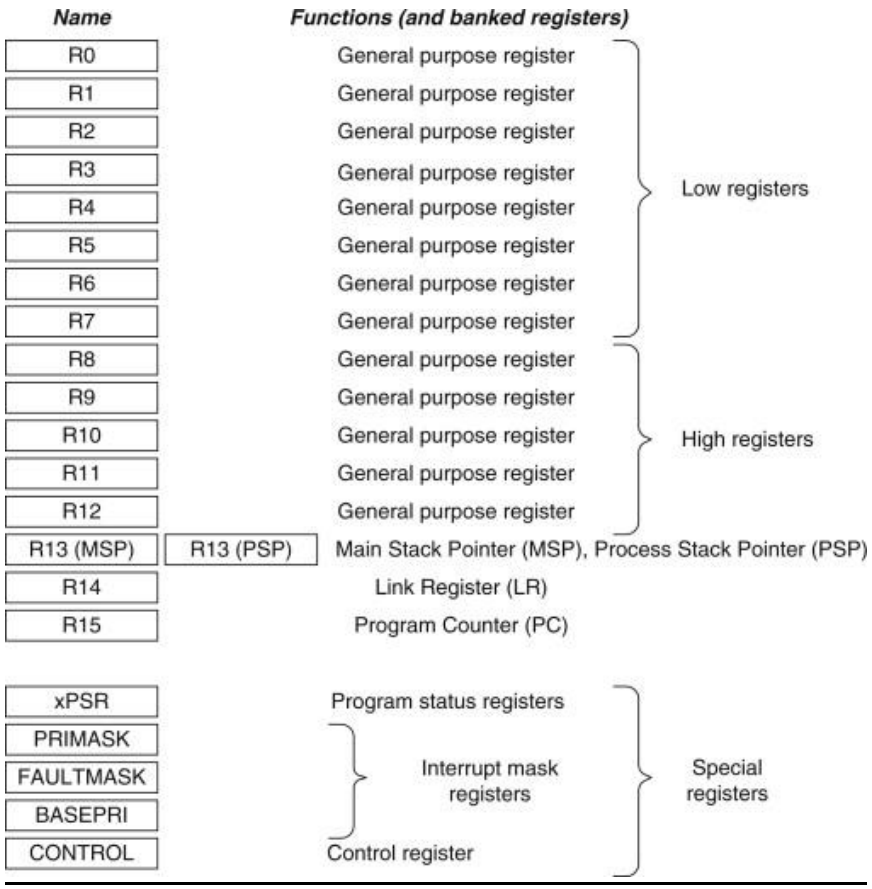


Fig 2.1– Types of Registers in CPU

1. General-Purpose Registers (GPRs)

x0 - x31 (32 Registers)

- **Functionality**: These registers are used for various operations, such as holding intermediate data, operands for instructions, and results of computations.
- **Special Roles**:
  - **x0**: Hardwired to zero. This register always contains the value 0 and cannot be written to.
  - **x1 (ra)**: Return Address. Used to hold the return address in function calls.
  - **x2 (sp)**: Stack Pointer. Points to the current top of the stack in memory.
  - **x3 (gp)**: Global Pointer. Used for accessing global variables.
  - **x4 (tp)**: Thread Pointer. Used in multi-threaded contexts to point to thread-local storage.
  - **x5 - x7**: Temporary registers. Used by instructions for intermediate values and are not preserved across function calls.
  - **x8 (s0/fp)**: Saved Register / Frame Pointer. Used to save the old value of the frame pointer, and also used as a frame pointer in function calls.

- **x9 - x15**: Saved Registers. Used to save values that need to be preserved across function calls.
- **x16 - x17**: Temporary registers.
- **x18 - x27**: Saved Registers.
- **x28 - x31**: Temporary registers.

2. Program Counter (PC)

- **Functionality**: Holds the address of the current instruction being executed. It is incremented or modified based on control flow instructions (like jumps and branches).

3. Special Registers

- **Status Register**: Includes various flags and control bits related to the processor's state (in some RISC architectures, the status register is combined with the control registers).
- **Control and Status Registers (CSRs)**: Used to manage processor state and configuration. These registers handle various control and status operations such as:
  - **mstatus**: Machine Status Register. Contains status bits related to processor state.
  - **mtvec**: Machine Trap-Vector Base Address Register. Holds the address where exception or interrupt handlers are located.
  - **mie**: Machine Interrupt Enable Register. Controls the enablement of machine-level interrupts.
  - **mtdeleg**: Machine Trap Delegation Register. Configures which types of traps are delegated to lower privilege levels.
  - **mcause**: Machine Cause Register. Records the cause of the last trap.
  - **mepc**: Machine Exception Program Counter. Contains the address of the instruction that caused the exception.
  - **mscratch**: Machine Scratch Register. Used for temporary storage of data across trap handling.

4. Floating-Point Registers (FPRs)

- **f0 - f31 (32 Registers)**
- **Functionality**: Used for floating-point computations. They are used in conjunction with floating-point instructions to perform arithmetic and data manipulation in floating-point format.

5. Vector Registers

- **v0 - v31 (32 Registers, in vector extensions)**
- **Functionality**: Used in vector processing extensions for performing operations on vectors of data. These are available in the RISC-V vector extension and are used for SIMD (Single Instruction, Multiple Data) operations.

6. Floating-Point Control and Status Registers

- **fcsr**: Floating-Point Control and Status Register. Contains flags and control bits related to floating-point operations.

7. Privilege-Level Registers

- Registers that are used to manage and configure different privilege levels in RISC-V, such as user mode and machine mode.

❖ Verilog Code for Registers File of CPU

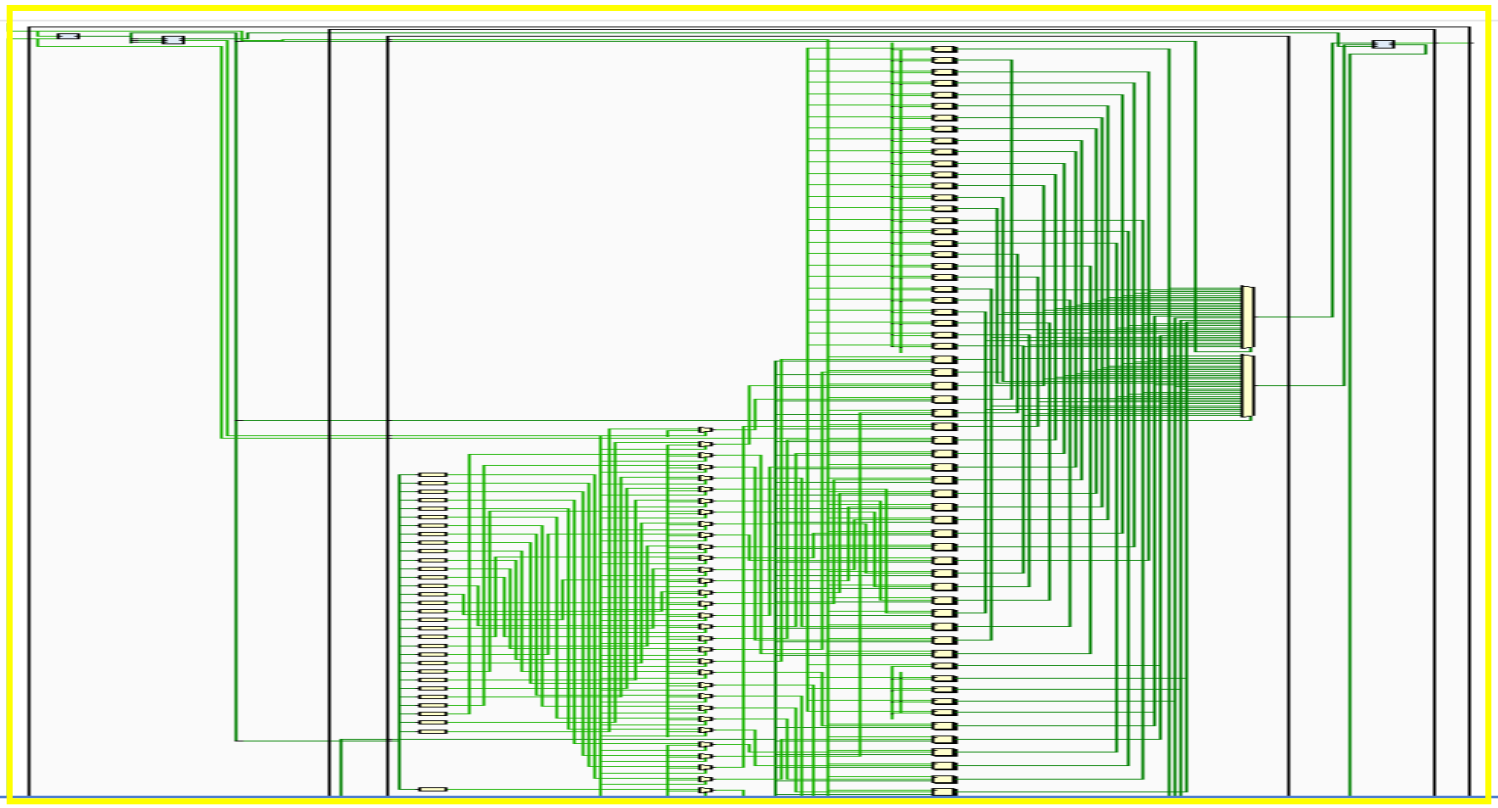
```
module RegisterFile (
    input wire clk,                 // Clock signal
    input wire rst,                 // Reset signal
    input wire reg_write,           // Register write enable
    input wire [4:0] rs1,           // Source register 1 (5-bit for 32 registers)
    input wire [4:0] rs2,           // Source register 2 (5-bit for 32 registers)
    input wire [4:0] rd,            // Destination register (5-bit for 32 registers)
    input wire [31:0] write_data,   // Data to write to the register file
    output wire [31:0] read_data1,  // Data read from register rs1
    output wire [31:0] read_data2   // Data read from register rs2
);

    reg [31:0] regfile [31:0];      // 32 x 32-bit registers

    // Asynchronous read for rs1 and rs2
    assign read_data1 = (rs1 != 5'b0) ? regfile[rs1] : 32'b0; // Read rs1 register if rs1 != 0
    assign read_data2 = (rs2 != 5'b0) ? regfile[rs2] : 32'b0; // Read rs2 register if rs2 != 0

    // Synchronous write
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            // Reset all registers to 0
            integer i;
            for (i = 0; i < 32; i = i + 1) begin
                regfile[i] <= 32'b0;
            end
        end else if (reg_write && rd != 5'b0) begin
            // Write data to the destination register if reg_write is enabled and rd != 0
            regfile[rd] <= write_data;
        end
    end

endmodule
```

❖ Test Bench for Registers File

```
module RegisterFile_tb;

    reg clk;
    reg rst;
    reg reg_write;
    reg [4:0] rs1;
    reg [4:0] rs2;
    reg [4:0] rd;
    reg [31:0] write_data;
    wire [31:0] read_data1;
    wire [31:0] read_data2;

    // Instantiate the Register File
    RegisterFile rf (
        .clk(clk),
        .rst(rst),
        .reg_write(reg_write),
        .rs1(rs1),
        .rs2(rs2),
        .rd(rd),
```

```
        .write_data(write_data),
        .read_data1(read_data1),
        .read_data2(read_data2)
    );

    // Clock generation
    always #5 clk = ~clk;

    // Test procedure
    initial begin
        // Initialize signals
        clk = 0;
        rst = 1;
        reg_write = 0;
        rs1 = 0;
        rs2 = 0;
        rd = 0;
        write_data = 0;

        // Reset the Register File
        #10 rst = 0;

        // Write data to register 1
        #10 reg_write = 1;
        rd = 5'b00001;
        write_data = 32'h12345678;
        #10 reg_write = 0; // Disable write

        // Read data from register 1
        rs1 = 5'b00001;
        #10;

        // Write data to register 2
        #10 reg_write = 1;
        rd = 5'b00010;
        write_data = 32'habcdef01;
        #10 reg_write = 0; // Disable write

        // Read data from register 2
        rs2 = 5'b00010;
        #10;

        // Finish simulation
        #20 $finish;
    end

    // Monitor changes
    initial begin
        $monitor("Time: %0d, Reg[1]: %h, Reg[2]: %h", $time, read_data1, read_data2);
    end

endmodule
```

❖ RTL Design for Register File



❖ Overall design of 32 Bits 32 registers Reg File

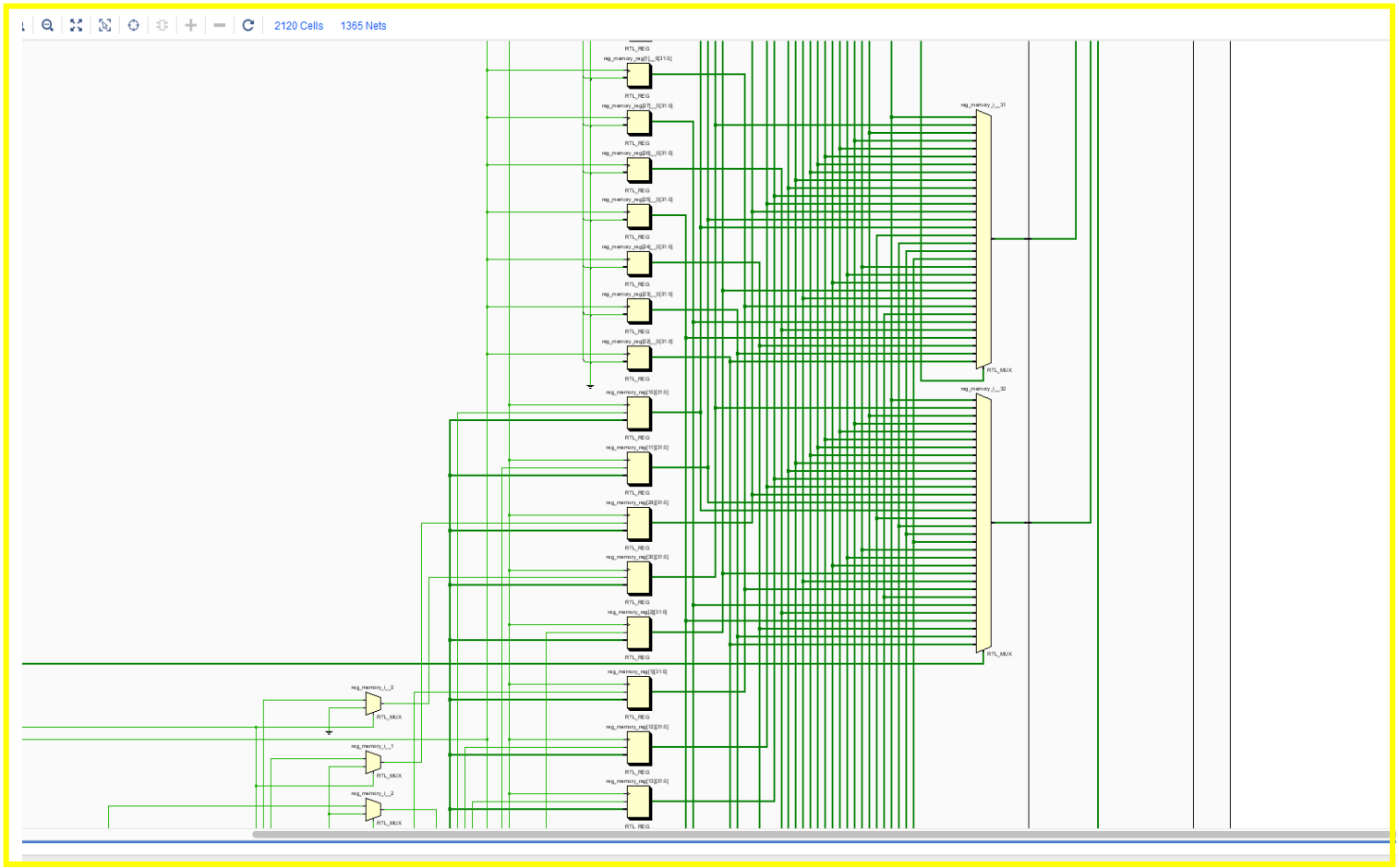


Fig 2.2 – Register File of Processor RTL View

d) Instruction Execute Unit / EX –

- **ALU Operation**
  - The Arithmetic Logic Unit (ALU) performs the required operation based on the control signals and the operands provided.
  - The ALU may perform operations such as addition, subtraction, logical AND/OR, etc.
- **Address Calculation**
  - For instructions that require memory access (like Load or Store), the effective address is calculated. This may involve adding an immediate value to a register value.

❖ Arithmetic Logic Unit/ ALU ->

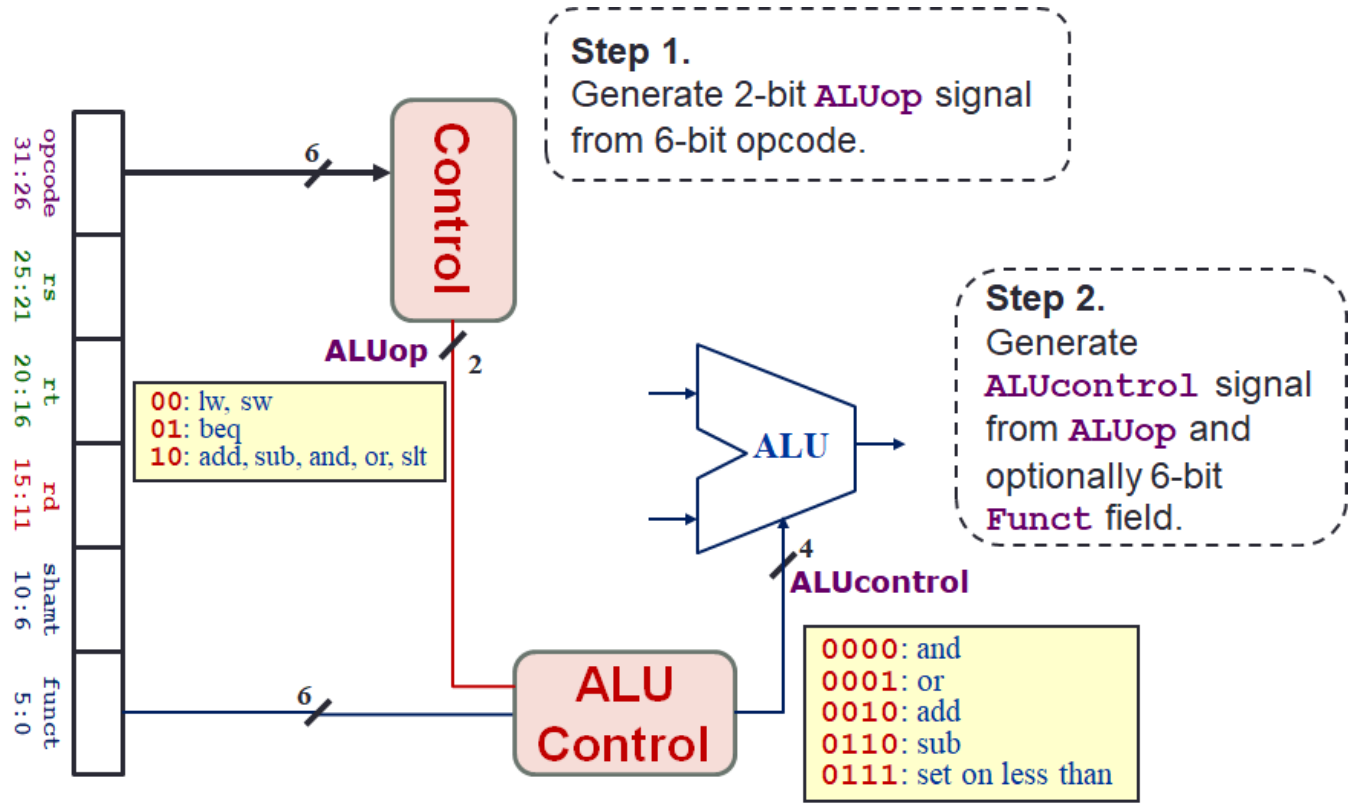


Fig 2.3 –Interface between ALU and Control Unit

➢ ALU Structure

1. **Operands**: The ALU takes two inputs, often referred to as operands. These can be:
   - **Register Values**: Data fetched from registers.
   - **Immediate Values**: Constant values directly specified in the instruction.
2. **ALU Control Unit**: This unit generates the necessary control signals to dictate the specific operation the ALU should perform.
3. **ALU Operations**: Based on the control signals, the ALU performs the operation and outputs the resul

- **ALU Operation**:

- **Operands**: Read values from registers R2 and R3.
- **Operation**: Perform addition as specified by the `ALUOp` signal.
- **Result**: Output the result to a specified register (R1).

- **Control Signals**:

1) ALUSrc

Purpose

- **ALUSrc** determines the source of the second operand for the ALU. It decides whether the ALU's second operand comes from a register or is an immediate value specified in the instruction.

Function

- **Operand Selection**: Controls whether the ALU will use a register value or an immediate value for its second operand.

- **Signal**: ALUSrc is a single-bit signal.
  - `0`: The second operand is taken from a register.
  - `1`: The second operand is an immediate value.

2) ALUOp

Purpose

- **ALUOp** is used to specify the type of operation the ALU should perform, but it doesn't specify the exact operation directly. Instead, it provides a higher-level instruction that helps determine what kind of operation is needed.

Function

- **Operation Encoding**: It encodes a general type of ALU operation based on the instruction's opcode and, sometimes, the funct field for more specific operations.
- **Higher-Level Control**: ALUOp is used to select among a set of predefined operation types, such as arithmetic or logical operations.

3) ALUControl (ALUctr)

Purpose

- **ALUControl** is used to directly specify the exact operation that the ALU should perform, such as addition, subtraction, logical AND, or logical OR.

Function

- **Operation Specification**: It provides a detailed control signal that directly dictates the ALU's operation based on the `ALUOp` and, if applicable, the funct code for R-type instructions.
- **Exact Control**: It converts the higher-level `ALUOp` signal into specific operations that the ALU can execute.

❖ How to get ALUctr from ALUop :-

- **`funct` Field:** Specifies the exact ALU operation required for R-type instructions by working in conjunction with `ALUOp`.

- **ALUControl Signal:** This signal is determined by decoding both `ALUOp` and the `funct` fields, enabling the ALU to perform the correct operation as specified by the instruction

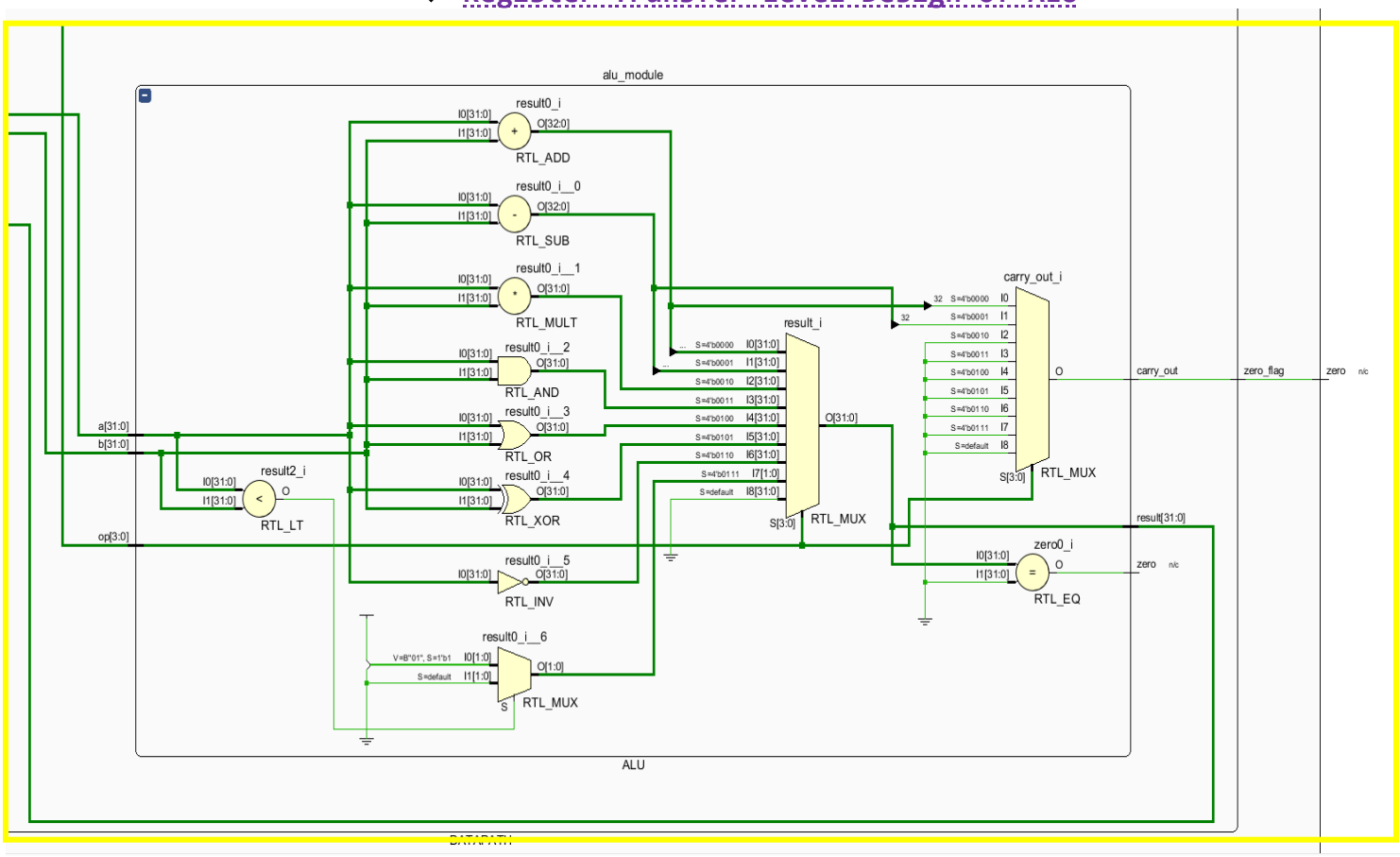
❖ Verilog Code for Standard ALU

```
/*
ALU Control lines | Function
----------------------------
        0000    Bitwise-AND
        0001    Bitwise-OR
        0010 Add (A+B)
        0100 Subtract (A-B)
        1000 Set on less than
        0011    Shift left logical
        0101    Shift right logical
        0110    Multiply
        0111    Bitwise-XOR
*/

module ALU (
    input [31:0] in1,in2,
    input[3:0] alu_control,
    output reg [31:0] alu_result,
    output reg zero_flag
);
    always @(*)
    begin
        // Operating based on control input
        case(alu_control)

        4'b0000: alu_result = in1&in2;
        4'b0001: alu_result = in1|in2;
        4'b0010: alu_result = in1+in2;
        4'b0100: alu_result = in1-in2;
        4'b1000: begin
            if(in1<in2)
            alu_result = 1;
            else
            alu_result = 0;
        end
        4'b0011: alu_result = in1<<in2;
        4'b0101: alu_result = in1>>in2;
        4'b0110: alu_result = in1*in2;
        4'b0111: alu_result = in1^in2;

        endcase

        // Setting Zero_flag if ALU_result is zero
        if (alu_result == 0)
            zero_flag = 1'b1;
        else
            zero_flag = 1'b0;

    end
endmodule
```

❖ Test Bench for Standard ALU

```
`include "ALU.v"
module stimulus ();
reg [31:0] A,B;
reg [3:0] ALUControl;
wire ZERO;
wire [31:0] ALUResult;
// Instantiating modules
ALU
ALU_module(.in1(A),.in2(B),.alu_control(ALUControl),.zero_flag(ZERO),.alu_result(ALUR
esult));
// Setting up waveform
initial
begin
        $dumpfile("output_wave.vcd");
        $dumpvars(0,stimulus);
end
// Monitoring changing values
initial
$monitor($time, "\nInput_1 = %b, \nInput_2 = %b,\nALU_control = %b,\n ALU_result =
%b, Zero_flag = %b\n",A,B,ALUControl,ALUResult,ZERO);
// Test conditions
initial
begin
  A = 23; B = 42;  ALUControl = 4'b0000;
#20 A = 23; B = 42;  ALUControl = 4'b0001;
#20 A = 23; B = 42;  ALUControl = 4'b0010;
#20 A = 23; B = 42;  ALUControl = 4'b0100;
end
// Finish after 150 clock cycles
initial
#150 $finish;
endmodule
```

❖ Register Transfer Level Design of ALU



❖ Optimized Architecture for ALU :-

1. Parallelism and Pipelining

- **Parallel Execution:** Use parallel paths for different operations to reduce latency. For instance, dedicated circuits for addition, subtraction, and multiplication can work simultaneously.
- **Pipelining:** Implement pipelining to overlap the execution of multiple instructions. Each stage of the pipeline performs a part of the operation, improving throughput.

2. Carry Lookahead and Carry Select

- **Carry Lookahead Adder (CLA):** Reduces delay in addition by predicting carry bits rather than waiting for them to be propagated through each bit position. It calculates carry bits in parallel.
- **Carry Select Adder:** Uses multiple adders to compute carry bits in parallel, selecting the correct result based on the actual carry-in bit.

➢ **Carry Look ahead Logic**

The CLA uses the generate and propagate signals to calculate carry-out signals quickly without waiting for the carry to ripple through each bit position. The carry-out of each bit position can be determined using:

- **Carry Generate (C_G):** Indicates whether a carry will be generated at this bit position regardless of the carry-in.
- **Carry Propagate (C_P):** Indicates whether a carry-in will be propagated to the next bit position,

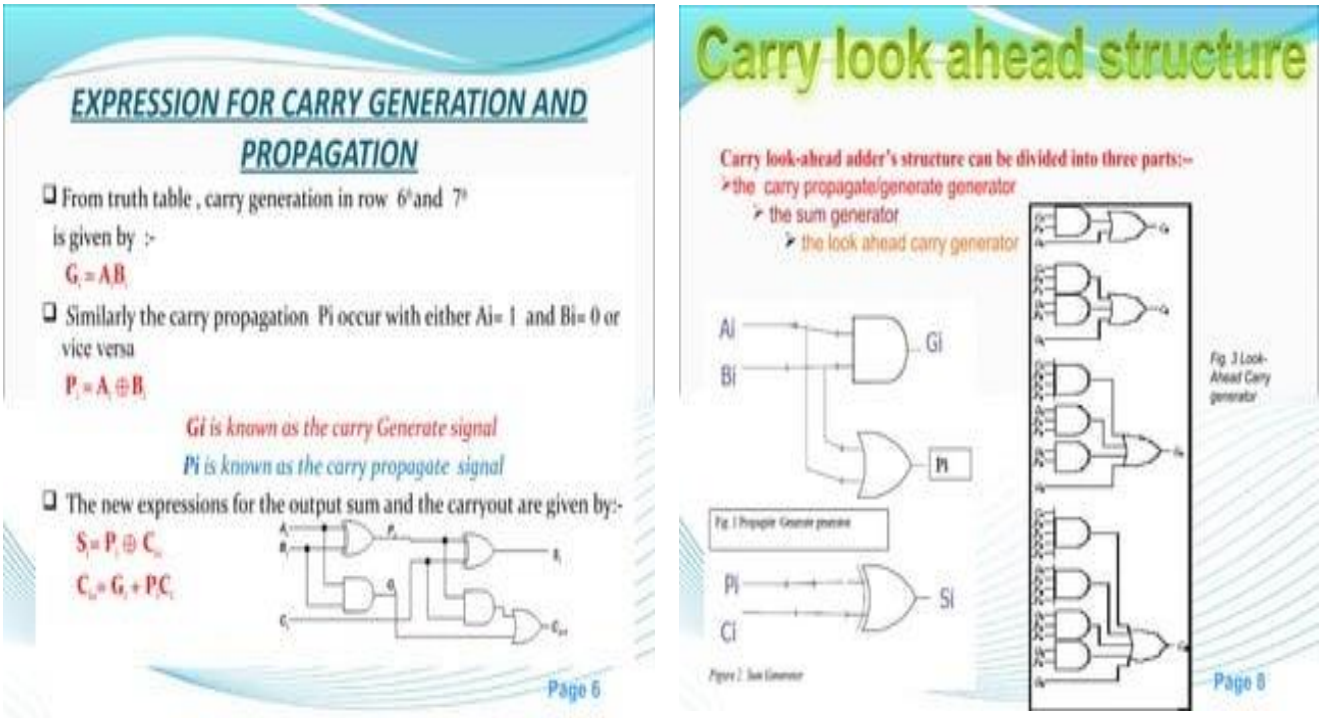


Fig 2.5 – Carry Look Ahead Adder for Optimized ALU

❖ Verilog Code for Carry Look Ahead Adder

```
module CLA4 (
input [3:0] A,           // 4-bit operand A
input [3:0] B,           // 4-bit operand B
input C_in,              // Carry-in
output [3:0] Sum,       // 4-bit sum output
output C_out );            // Carry-out

// Internal wires
wire [3:0] G;            // Generate signals
wire [3:0] P;            // Propagate signals
wire [3:0] C;            // Carry signals

// Generate and propagate calculations
assign G = A & B;        // Generate signals
assign P = A | B;        // Propagate signals

// Carry signals calculations
assign C[0] = G[0] | (P[0] & C_in);   // Carry-out for bit 0
assign C[1] = G[1] | (P[1] & C[0]);   // Carry-out for bit 1
assign C[2] = G[2] | (P[2] & C[1]);   // Carry-out for bit 2
assign C[3] = G[3] | (P[3] & C[2]);   // Carry-out for bit 3

// Sum calculations
assign Sum = A ^ B ^ {C[2:0], C_in}; // Sum bit
// Carry-out
assign C_out = C[3];    // Carry-out of the most significant bit
endmodule
```

❖ Test Bench for CLA

```
module tb_CLA4;

reg [3:0] A;
reg [3:0] B;
reg C_in;
wire [3:0] Sum;
wire C_out;

// Instantiate the CLA4 module
CLA4 cla4 (
    .A(A),
    .B(B),
    .C_in(C_in),
    .Sum(Sum),
    .C_out(C_out)
);

initial begin
    // Initialize inputs
    A = 4'b0000;
    B = 4'b0000;
    C_in = 0;

    // Test case 1
    #10 A = 4'b0011; B = 4'b0101; C_in = 0;
    #10; // Wait and check results

    // Test case 2
    #10 A = 4'b1111; B = 4'b0001; C_in = 0;
```

```
        #10; // Wait and check results

        // Test case 3
        #10 A = 4'b1010; B = 4'b0101; C_in = 1;
        #10; // Wait and check results

        // End simulation
        #10 $finish;
    end

    // Monitor signals
    initial begin
        $monitor("Time = %0d: A = %b, B = %b, C_in = %b, Sum = %b, C_out = %b",
                 $time, A, B, C_in, Sum, C_out);
    end

endmodule
```

❖ RTL Design of Carry Look Ahead Adder :

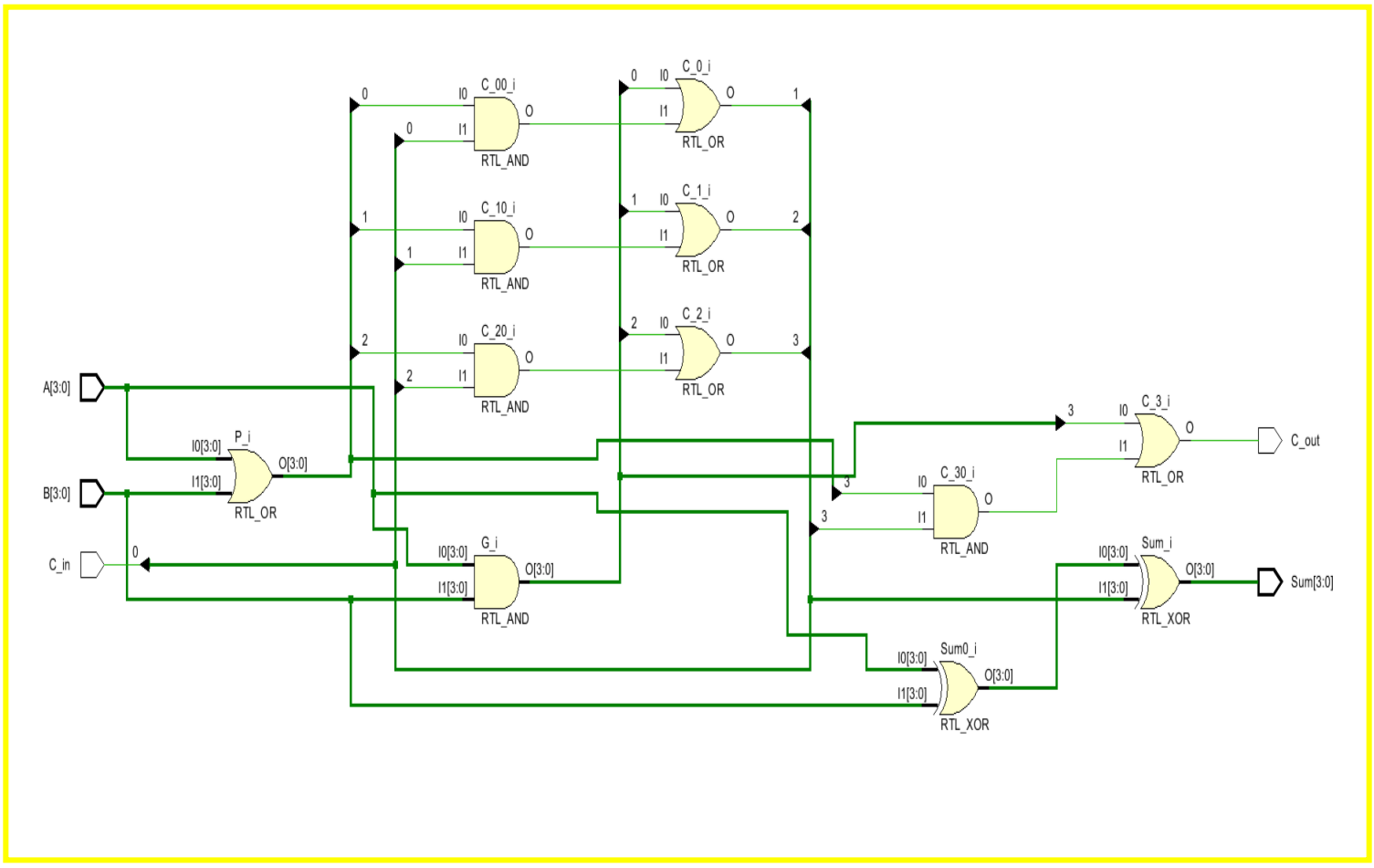


Fig 2.6 – CLA Circuit Design

e) Memory Block / Mem ->

➢ Components of the Memory Block

- **Memory Array:** This is the actual storage element where data is held. It can be implemented using RAM (Random Access Memory) or similar memory structures.
- **Memory Address Register (MAR):** Holds the address in memory from which data is to be read or written.
- **Memory Data Register (MDR):** Holds the data to be written to memory or the data read from memory.
- **Read/Write Control Signals:**
  - **MemRead:** Signal to enable reading from memory.
  - **MemWrite:** Signal to enable writing to memory.

➢ Functionality of the Memory Block

1. Memory Read Operation

- **Address Setup:** The address of the memory location to be read is placed into the MAR.
- **Read Enable:** The `MemRead` signal is asserted (set to 1) to enable the memory read operation.
- **Data Access:** The memory array uses the address in MAR to access the corresponding memory location.
- **Data Retrieval:** The data from the memory location is placed into the MDR.
- **Data Output:** The data in the MDR is sent to the processor or output as required.

2. Memory Write Operation

- **Address Setup:** The address of the memory location to which data will be written is placed into the MAR.
- **Data Setup:** The data to be written to memory is placed into the MDR.
- **Write Enable:** The `MemWrite` signal is asserted (set to 1) to enable the memory write operation.
- **Data Storage:** The memory array writes the data from the MDR to the memory location specified by MAR.

❖ Verilog Code for MEM Block

```
➢   module MEM_BLOCK (
➢   input wire [31:0] address,          // Address for read/write operations
➢   input wire [31:0] write_data,       // Data to write into memory
➢   input wire mem_read,                // Memory read enable signal
➢   input wire mem_write,               // Memory write enable signal
➢   input wire reset,                   // Reset signal
➢   input wire clock,                   // Clock signal
➢   output reg [31:0] read_data        // Data read from memory
➢   );
➢       // Memory storage: 32-bit wide, 256 locations
➢       reg [31:0] memory [255:0];
➢
➢       // Reset and Initialization
➢       always @(posedge reset) begin
➢           // Initialize memory to some known values or zeroes
➢           integer i;
➢           for (i = 0; i < 256; i = i + 1) begin
➢               memory[i] = 32'h0;
➢           end
➢       end
➢
➢       // Memory Read and Write Operations
```

```
    always @(posedge clock) begin
        if (mem_write) begin
            memory[address[31:2]] <= write_data;  // Write data to memory
        end
        if (mem_read) begin
            read_data <= memory[address[31:2]];   // Read data from memory
        end
    end

endmodule
```

❖ RTL Design of Memory Block

f) Write-Back (WB) Stage ->

The WB stage is the final stage in the instruction execution pipeline. Its primary responsibility is to write the results of an instruction back into the register file. This stage typically involves the following operations:

1. **Write Data Selection**: Determine which data should be written back to the register file. This can be the result of an ALU operation, a memory read, or some other computation.
2. **Write Register Selection**: Decide which register will receive the data. This is usually specified by the instruction's destination register field.
3. **Register File Update**: Update the register file with the data selected.

Components of the WB Stage

1. **Data Path Components**:
   - **ALU Result**: The result of arithmetic or logical operations performed in the EX stage.
   - **Memory Data**: Data read from memory if the instruction involves a load operation.
   - **Register Write Address**: The address of the register where the data will be written.
2. **Control Signals**:
   - **`RegWrite`**: A signal that enables writing data to the register file.
   - **`MemToReg`**: A signal used to select whether the data to be written to the register file comes from the ALU result or from memory.

Role of the MUX in the Write-Back Stage

1. **Inputs to the MUX**:
   - **Data from ALU**: The result produced by the ALU after executing an instruction.
   - **Data from Memory**: The data fetched from the data memory during a load operation.
2. **Control Signal (`MemToReg`)**:
   - **`MemToReg` = 0**: Selects the ALU result. This is used for R-type instructions and some I-type instructions where the result is produced by the ALU.
   - **`MemToReg` = 1**: Selects the data from memory. This is used for load instructions where the result of the instruction is obtained from memory.
3. **Output of the MUX**: The selected data is output from the MUX and is then written to the register file.

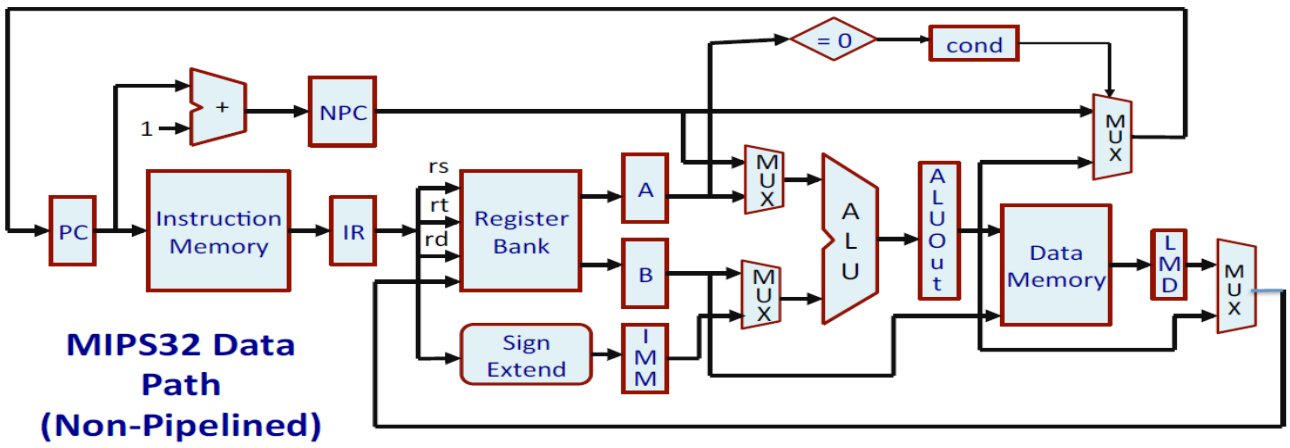


Fig 2.7 – Data Path and Write – Back Stage

- **LMD (Load Memory Data)**

LMD is the data fetched from memory in load instructions. It represents the value retrieved from a memory address during a load operation.

- **IMM (Immediate Value)**

IMM refers to an immediate value directly encoded within an instruction. It is used as an operand in arithmetic or logical operations and as an offset for load/store operations.

❖ **NPC** stands for **Next Program Counter**. It represents the address of the next instruction to be executed in a pipeline or sequence of instructions.

❖ Verilog Code for WB Stage

```
module WriteBackStage(
    input wire [31:0] ALUResult,          // Result from ALU
    input wire [31:0] LoadData,           // Data read from memory
    input wire MemToReg,                  // Control signal to select between memory
data and ALU result
    input wire [4:0] WriteRegAddr,        // Register address to write the result
    output wire [31:0] WriteData,         // Data to be written to the register file
    input wire RegWrite,                  // Control signal to enable writing to the
register file
    input wire clock,                     // Clock signal
    input wire reset,                     // Reset signal
    // Register file ports
    input wire [31:0] ReadData1,          // Data read from register 1
    input wire [31:0] ReadData2           // Data read from register 2
);

    // MUX to select between ALU result and memory data
    assign WriteData = (MemToReg) ? LoadData : ALUResult;

    // Write to register file when RegWrite is enabled
    always @(posedge clock or posedge reset) begin
        if (reset) begin
            // On reset, clear or initialize registers if necessary
        end else if (RegWrite) begin
            // Write data to register file
            // Assuming the register file instance is available and properly
connected
            // This is a placeholder for actual register file write operation
            // reg_file[WriteRegAddr] <= WriteData;
        end
    end

endmodule
```

- In order to represent working of register file and ALU (Execute stage) , we need to make a data path which can interface with Control unit and give desired results.

❖ Data Path of RISK – ISA Processor

➢ Components of a Data path :-

1. **Register File (REG_FILE):**
   - Stores the registers of the processor.
   - Provides read and write access to these registers.
   - Inputs include read register addresses, write register address, data to write, and control signals.
2. **ALU (Arithmetic Logic Unit):**
   - Performs arithmetic and logical operations on the data provided.
   - Takes two inputs (operands), performs an operation as specified by a control signal, and outputs the result.
   - Can also output flags like zero, carry, etc.

➢ Basic Data path Overview :-

Here's a simple overview of a datapath with a register file and an ALU:

1. **Read Data from Register File:**
   - Two registers are read simultaneously based on the provided read register addresses.
2. **ALU Operation:**
   - The data read from the registers are provided to the ALU.
   - The ALU performs an operation based on the ALU control signal.
3. **Write Back:**
   - The result from the ALU can be written back to a register in the register file.

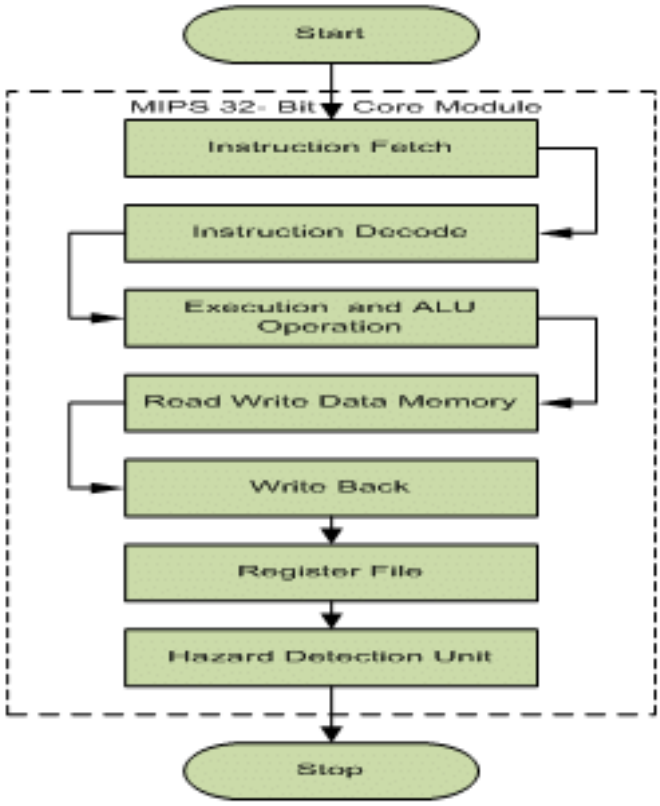


Fig 2.8 - Interconnected Blocks of RISK ISA Processor

❖ Verilog Code for Data Path

```
`include "REG_FILE.v"
`include "ALU.v"

module DATAPATH(
    input [4:0]read_reg_num1,
    input [4:0]read_reg_num2,
    input [4:0]write_reg,
    input [3:0]alu_control,
    input regwrite,
    input clock,
    input reset,
    output zero_flag
);

    // Declaring internal wires that carry data
    wire [31:0]read_data1;
    wire [31:0]read_data2;
    wire [31:0]write_data;

    // Instantiating the register file
    REG_FILE reg_file_module(
    read_reg_num1,
    read_reg_num2,
    write_reg,
    write_data,
    read_data1,
    read_data2,
    regwrite,
    clock,
    reset
    );

    // Instanting ALU
    ALU alu_module(read_data1, read_data2, alu_control, write_data, zero_flag);

endmodule
```

❖ RTL Design of Data Path

❖ Combined Blocks of Non pipelined RISK ISA Processor

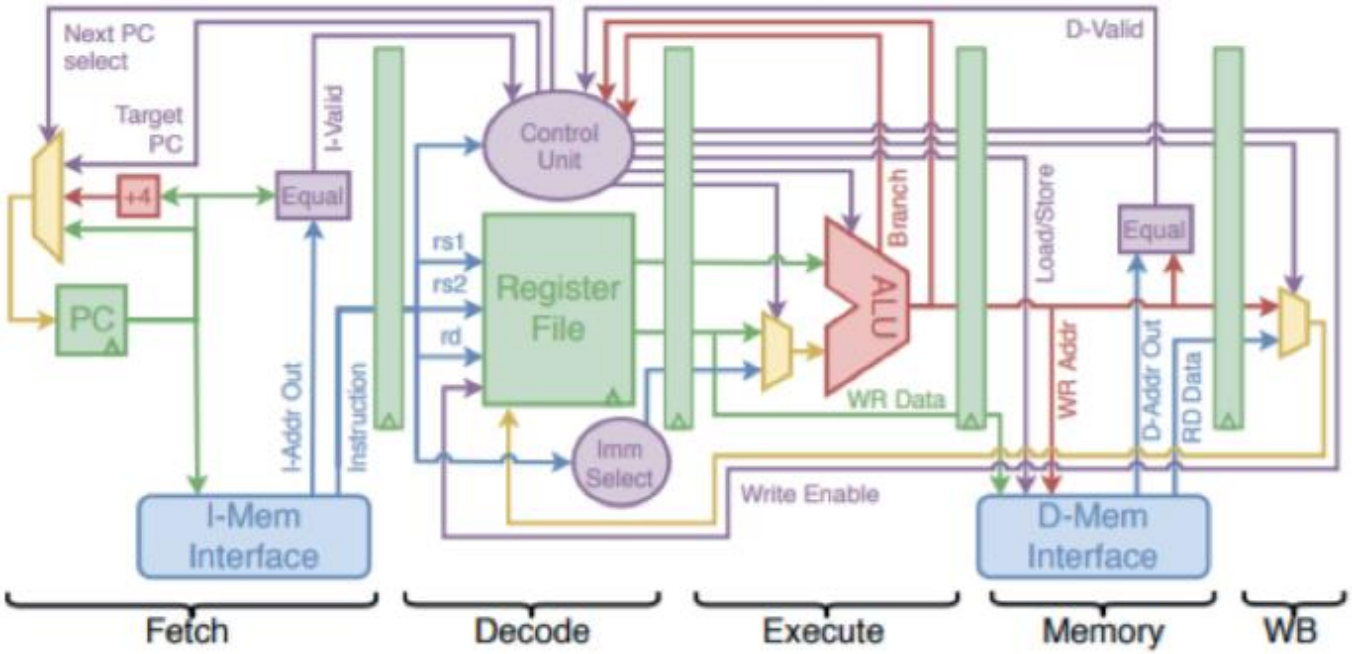


Fig 2.9 – Different Blocks combined Together to make CPU

❖ Verilog Code for Processor

```
➢ `include "CONTROL.v"
➢ `include "DATAPATH.v"
➢ `include "IFU.v"
➢
➢ module PROCESSOR(
➢     input clock,
➢     input reset,
➢     output zero
➢ );
➢
➢     wire [31:0] instruction_code;
➢     wire [3:0] alu_control;
➢     wire regwrite;
➢
➢     IFU IFU_module(clock, reset, instruction_code);
➢
➢     CONTROL control_module(instruction_code[31:25],
  instruction_code[14:12],instruction_code[6:0], alu_control, regwrite);
➢
➢     DATAPATH datapath_module(instruction_code[19:15], instruction_code[24:20],
  instruction_code[11:7], alu_control, regwrite, clock, reset, zero);
➢
➢ endmodule
```

❖ Test Bench for Processor

```
`include "PROCESSOR.v"

module stimulus ();

    reg clock;
    reg reset;
    wire zero;

    // Instantiating the processor!!!
    PROCESSOR test_processor(clock,reset,zero);

    initial begin
        $dumpfile("output_wave.vcd");
        $dumpvars(0,stimulus);
    end

    initial begin
        reset = 1;
        #50 reset = 0;
    end

    initial begin
        clock = 0;
        forever #20 clock = ~clock;
    end

    initial
    #300 $finish;

endmodule
```

❖ RTL Design of Overall Non Pipelined RISC ISA Processor

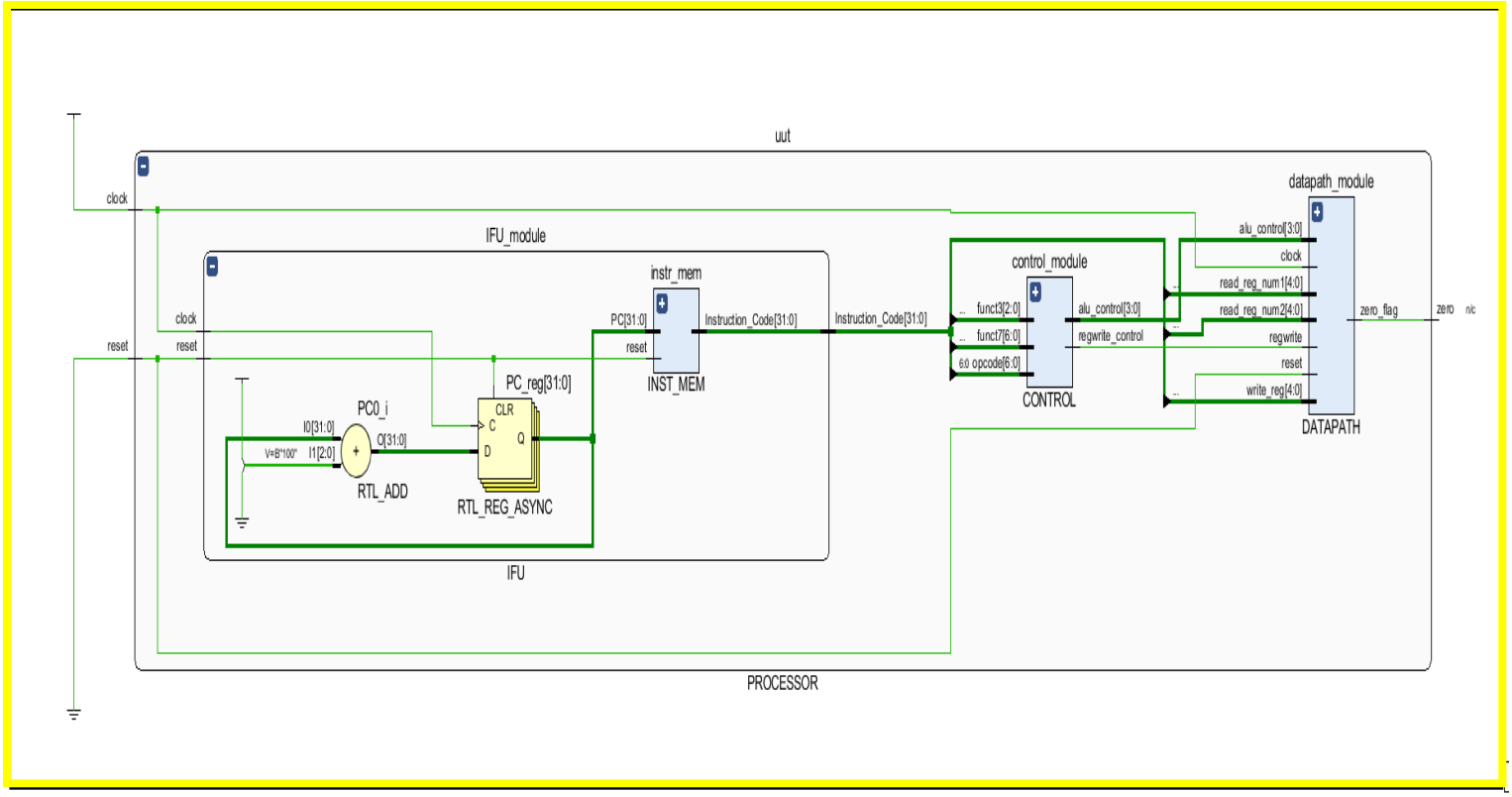


Fig 3.0 – Full design of Processor, includes all small modules (Control, Datapath and IFU module)

❖ Explanation of Code

1. Initialization and Input Handling

- **Clock and Reset Inputs:**
  - **Clock Signal:** Keeps the entire processor synchronized. All state changes and data transfers occur on the clock edges.
  - **Reset Signal:** Initializes or resets the processor. When activated, it sets the processor state to a known initial state.

2. Instruction Fetch

- **Instruction Fetch Unit (IFU):**
  - **Function:** The IFU module fetches the instruction from memory based on the current program counter (PC).
  - **Process:**
  - The `IFU` module takes the `clock` and `reset` signals to synchronize its operations and handle initialization.
  - It outputs the 32-bit `instruction_code`, which represents the current instruction to be executed.

3. Instruction Decoding and Control Signal Generation

- **Control Unit (CONTROL):**
  - **Function:** The `CONTROL` module decodes the fetched instruction to generate control signals required for the ALU and register file.
  - **Process:**
    - The module receives parts of the `instruction_code`:
    - **Opcode (instruction_code[31:25]):** Determines the instruction type.
    - **Funct3 (instruction_code[14:12]):** Provides additional operation-specific details.
    - **Funct7 (instruction_code[6:0]):** Further specifies the exact operation if needed.
    - Based on these inputs, the `CONTROL` module outputs:
    - **ALU Control Signals (`alu_control`):** Direct the ALU on what operation to perform.
    - **Register Write Enable (`regwrite`):** Indicates whether the result should be written to the register file.

4. Data Path Operation

- **Data path (DATA PATH):**
  - **Function:** Executes the instructions by performing operations on data and managing the register file.
  - **Process:**
  - **Register File Access:**
  - Reads data from registers specified by the instruction (`read_reg_num1` and `read_reg_num2`).
  - The `write_reg` signal determines which register will receive the result.
  - **ALU Operation:**
  - The `ALU` module performs arithmetic or logical operations based on the `alu_control` signals.
  - The ALU processes the data from the registers and outputs the result to be written back to the register file if `regwrite` is active.
  - **Zero Flag:** The ALU sets the `zero` flag if the result of its operation is zero. This flag can be used for conditional branching.

5. Interaction Between Modules

- **Instruction Flow:**
  - The `IFU` module fetches the instruction and provides it to the `CONTROL` module.
  - The `CONTROL` module decodes the instruction and generates control signals.
  - The `DATAPATH` module uses these control signals to execute the instruction, interacting with the register file and ALU.

6. Summary of Operation

- **Fetch:** The processor fetches an instruction from memory.
- **Decode:** The processor decodes the instruction to understand what operation is needed and generates corresponding control signals.
- **Execute:** The processor executes the instruction using the ALU and updates the register file if needed.
- **Result:** The processor determines if the result of the ALU operation is zero and updates the state of the registers based on the control signals.

❖ **Simulation Results of Processor in Xilinx Software**

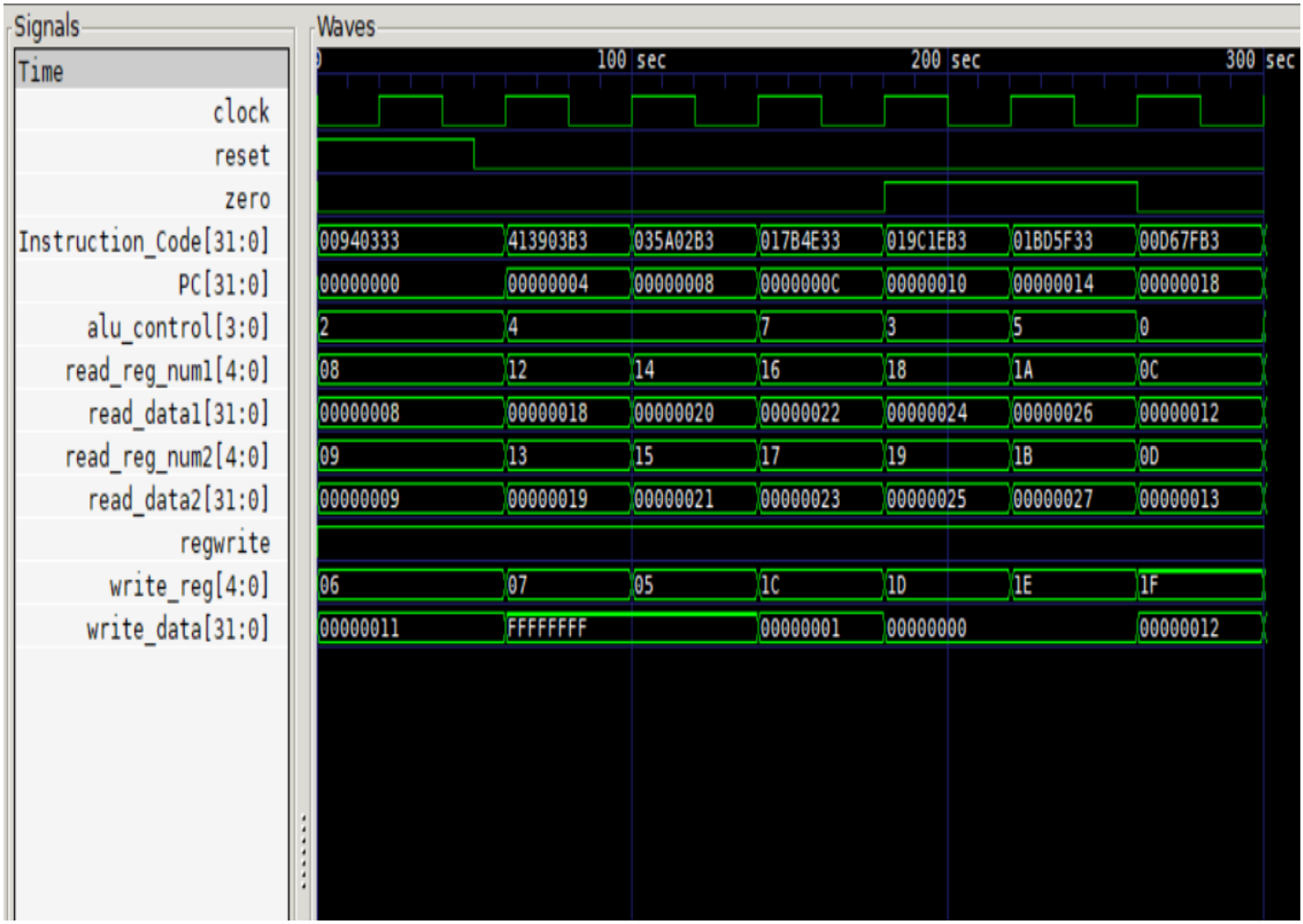


Fig 3.1 – Result in the form of Waveforms

❖ Schematic of RISK-V ISA Non pipelined Processor

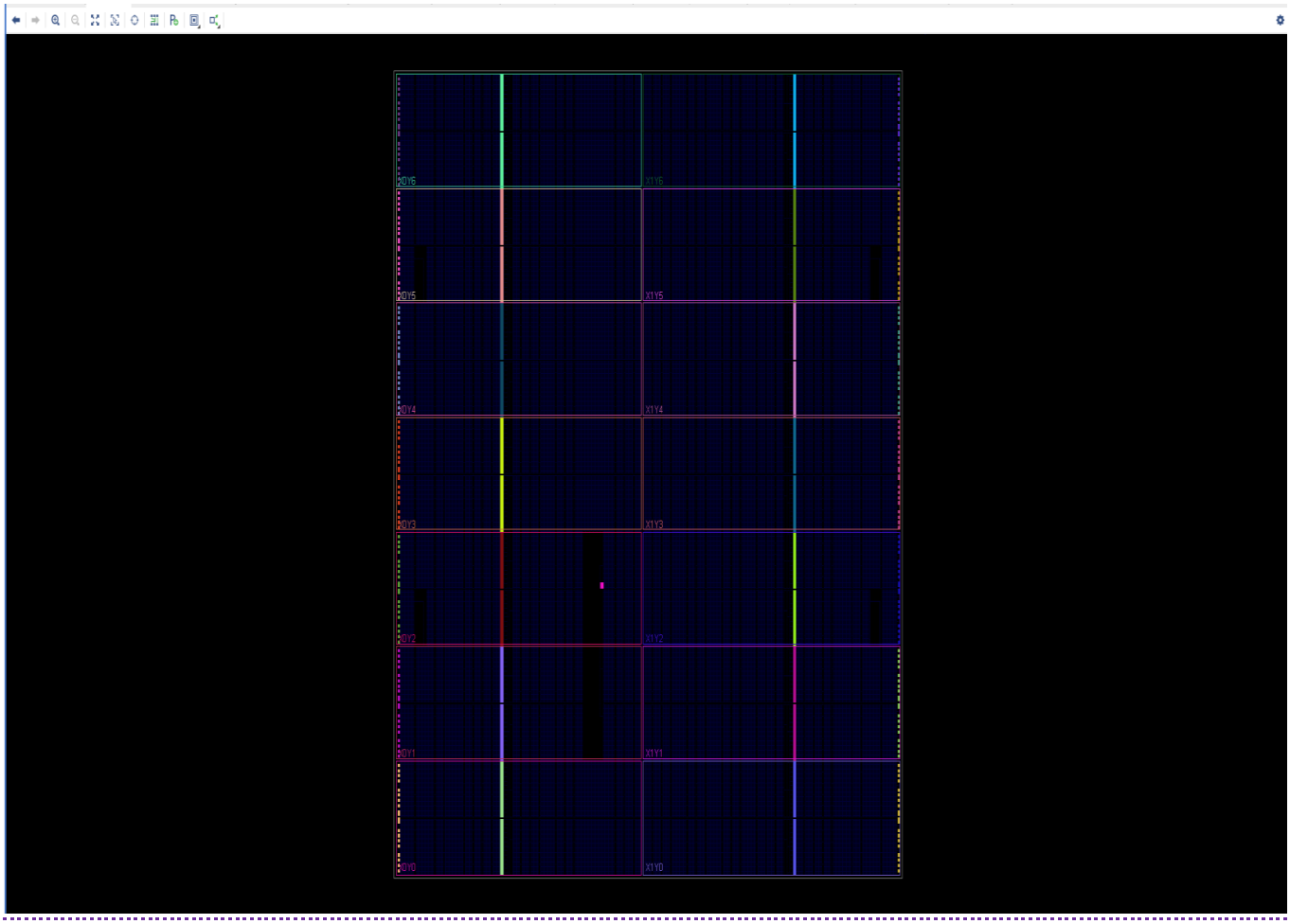


Fig 3.2 – RISC – V ISA Processor Schematic

Fig 3.3 – Zoomed view of schematic