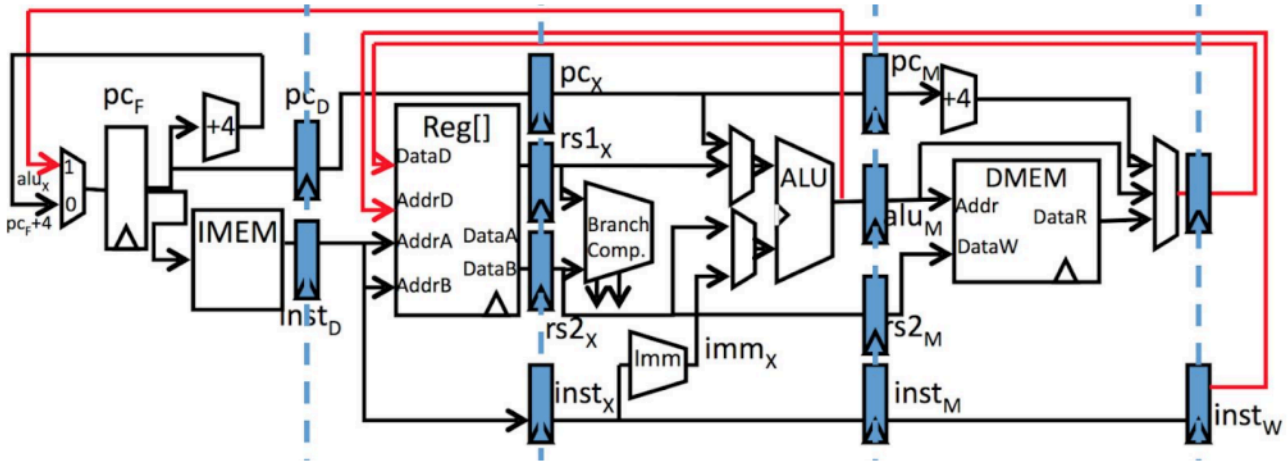
20.10.2024

K Charan
B Tech (ECE)

Problem statement

Verilog code for a 5-stage pipelined RISC-V processor, along with an example of a small arithmetic operation (RV32I)



Verilog code:-

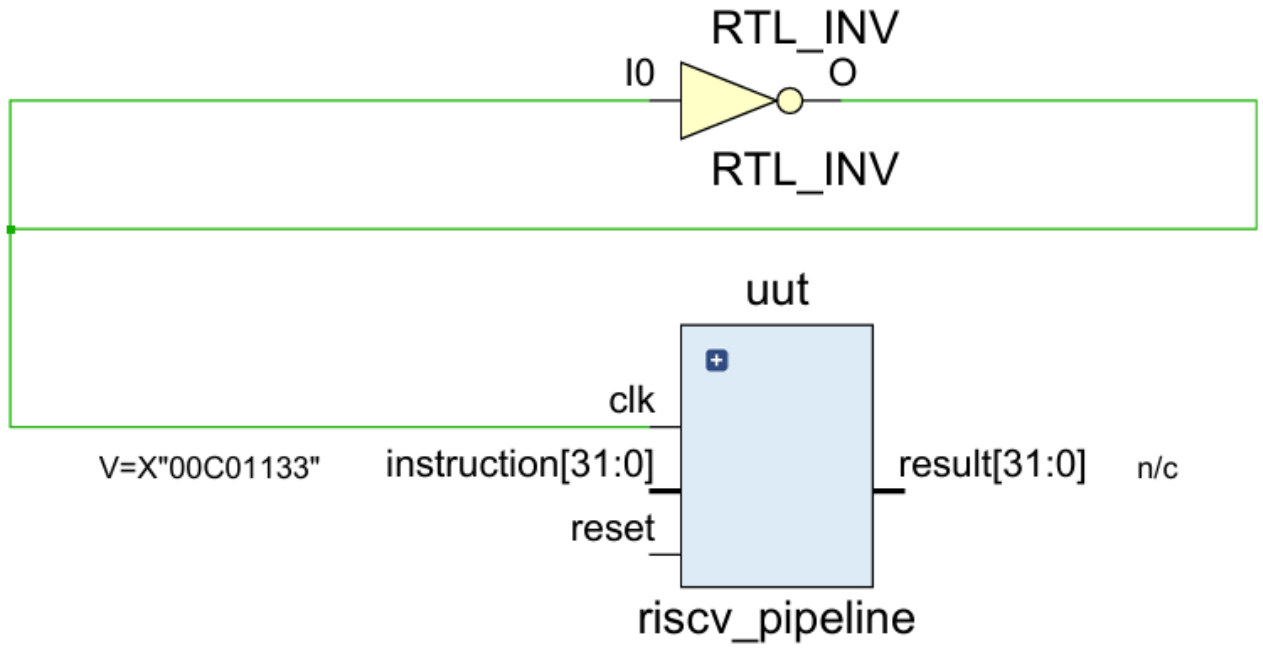
```
`timescale 1ns / 1ps

module riscv_pipeline (
    input clk,                // Clock signal
    input reset,              // Reset signal
    input [31:0] instruction, // Input instruction (32-bit)
    output reg [31:0] result  // Output result (32-bit)
);

// Define pipeline registers
reg [31:0] IF_ID_instruction; // Instruction register between IF and ID stages
reg [31:0] ID_EX_alu_out;     // ALU output register between ID and EX stages
reg [31:0] EX_MEM_alu_out;    // ALU output register between EX and MEM stages
reg [31:0] MEM_WB_data;       // Data register between MEM and WB stages

// Control signals
```

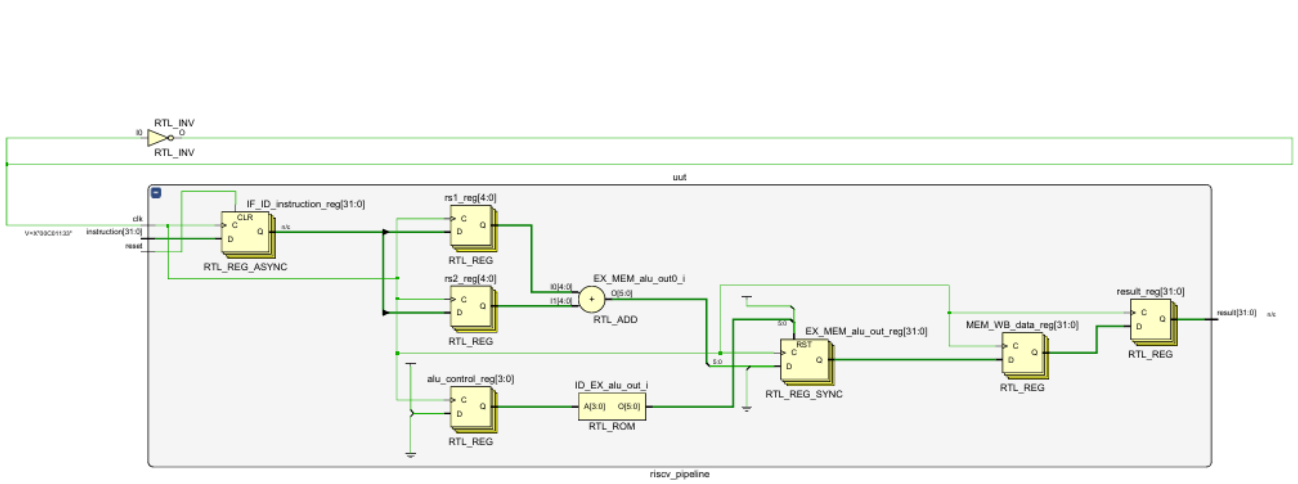
```
reg [4:0] rs1, rs2, rd;       // Source and destination registers
reg [3:0] alu_control;        // ALU control signal

// Register file (for simplicity)
reg [31:0] registers [0:31]; // 32 registers

// ALU operation
always @(*) begin
    case (alu_control)
        4'b0010: ID_EX_alu_out = registers[rs1] + registers[rs2]; // ADD
operation
        default: ID_EX_alu_out = 32'b0;
    endcase
end

// Instruction Fetch (IF) Stage
always @(posedge clk or posedge reset) begin
    if (reset) begin
        IF_ID_instruction <= 32'b0;
    end else begin
        IF_ID_instruction <= instruction; // Fetch instruction
    end
end

// Instruction Decode (ID) Stage
always @(posedge clk) begin
    rd <= IF_ID_instruction[11:7];   // Destination register from instruction
    rs1 <= IF_ID_instruction[19:15];  // Source register 1 from instruction
    rs2 <= IF_ID_instruction[24:20];  // Source register 2 from instruction

    // Control signal for ALU operation (for ADD)
    alu_control <= 4'b0010;
end

// Execute (EX) Stage
always @(posedge clk) begin
    EX_MEM_alu_out <= ID_EX_alu_out; // Pass ALU output to MEM stage
end

// Memory Access (MEM) Stage
always @(posedge clk) begin
    MEM_WB_data <= EX_MEM_alu_out; // Pass data to WB stage
end
```

```
// Write Back (WB) Stage
always @(posedge clk) begin
    result <= MEM_WB_data; // Output final result after write-back stage

    // Write result back to the register file (for demonstration)
    registers[rd] <= result;
end

endmodule

// Testbench for the RISC-V Pipeline Processor
module tb_riscv_pipeline;
    reg clk;
    reg reset;
    reg [31:0] instruction;
    wire [31:0] result;

    riscv_pipeline uut (
        .clk(clk),
        .reset(reset),
        .instruction(instruction),
        .result(result)
    );

    initial begin
        clk = 0;
        reset = 1;
        #10 reset = 0;

        // Example arithmetic operation: ADD x1, x2, x3 -> x1 = x2 + x3
        instruction = {7'b0000000, 5'b00010, 5'b00001, 3'b000, 5'b00001,
7'b0110011}; // Corrected ADD instruction

        #20; // Wait for some time to observe results

        $display("Result of ADD operation: %d", result);

        $finish;
    end

    always #5 clk = ~clk; // Generate clock signal every 5 time units
```

```
endmodule
```

output





Key points

In the provided Verilog code for the RISC-V pipelined processor, the arithmetic operation performed is an **addition** of two registers. Let's break down the relevant parts of the code and explain how the addition is implemented, including the bit sizes involved.

Key Components of the Addition Operation

Register File:

- The register file is defined as:

```
reg [31:0] registers [0:31]; // 32 registers
```

- Here, `registers` is an array of 32 registers, each 32 bits wide. This means each register can hold a value in the range of 00 to 232−1232−1 for unsigned integers or −231−231 to 231−1231−1 for signed integers.
  Instruction Encoding:
- The ADD instruction is encoded as follows:

```
instruction = {7'b0000000, 5'b00010, 5'b00001, 3'b000, 5'b00001, 7'b0110011};
```

- This instruction consists of various fields:
  - **funct7 (7 bits)**: Specifies the operation type (for ADD, this is `0000000`).
  - **rs2 (5 bits)**: The second source register (in this case, `x2`, which is represented as `00010`).
  - **rs1 (5 bits)**: The first source register (in this case, `x1`, which is represented as `00001`).
  - **funct3 (3 bits)**: Specifies the operation subtype (for ADD, this is `000`).
  - **rd (5 bits)**: The destination register where the result will be stored (in this case, also `x1`, represented as `00001`).
  - **opcode (7 bits)**: Specifies the instruction type (for R-type instructions like ADD, this is `0110011`).

Arithmetic Expression

The actual addition occurs in the ALU operation section of the code:

```
always @(*) begin
    case (alu_control)
        4'b0010: ID_EX_alu_out = registers[rs1] + registers[rs2]; // ADD
operation
        default: ID_EX_alu_out = 32'b0;
    endcase
end
```

- **Registers Access**:
  - `registers[rs1]`: This accesses the value stored in register `rs1`.

    - `registers[rs2]`: This accesses the value stored in register `rs2`.
- **Addition Operation**:
    - The expression `registers[rs1] + registers[rs2]` performs a binary addition of the two 32-bit values retrieved from the register file.

Bit Size of Addition

- Each operand (`registers[rs1]` and `registers[rs2]`) is **32 bits** wide.
- In binary addition:
    - If both operands are n bits wide (in this case, n=32n=32), the result can be up to n+1 bits wide due to potential carry-out from the most significant bit.

Example

For example, if we have:

- `registers[rs1] = 32'h00000002` (which is decimal 2)
- `registers[rs2] = 32'h00000003` (which is decimal 3)

The addition would be:

```
ID_EX_alu_out = registers[rs1] + registers[rs2]; // Result will be 5
```

The binary addition would look like this:

```
   0000 0000 0000 0000 0000 0000 0000 0010   // Value in rs1 (2)
+  0000 0000 0000 0000 0000 0000 0000 0011   // Value in rs2 (3)
-------------------------------------------
   0000 0000 0000 0000 0000 0000 0000 0101   // Result in ID_EX_alu_out (5)
```

Conclusion

In summary, the arithmetic expression performs a **32-bit addition** of two values retrieved from registers specified by the instruction. The result is also a **32-bit value**, which can be stored back into a destination register. The addition logic ensures that even though each operand is limited to 3232 bits, it can handle cases where overflow occurs by using an additional bit if necessary.