Implementation of a RISC-V Processor in Verilog

International Institute of Information Technology - Hyderabad

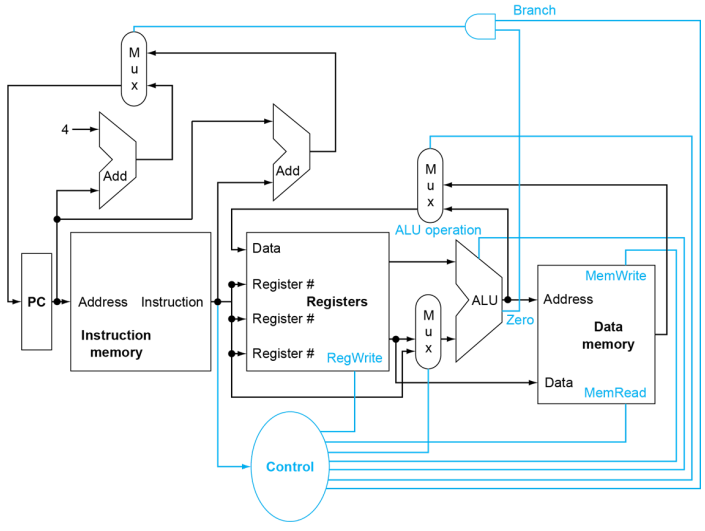
Anshitha Penubolu, anshitha.penubolu@students.iiit.ac.in , 2023102014
Guru Shwadeep, guru.dornadula@research.iiit.ac.in , 2023112020
Harsh Kapoor, harsh.kapoor@research.iiit.ac.in , 2023112004

***Abstract*— This paper presents the design and implementation of a RISC-V processor using Verilog, incorporating both sequential and pipelined architectures. The processor executes fundamental RISC-V instructions, including add, sub, and, or, ld, sd, and beq. The pipelined design employs a five-stage architecture—Fetch, Decode, Execute, Memory, and Write Back—with pipeline registers to enhance instruction throughput. The results demonstrate improved efficiency in the pipelined implementation compared to the sequential design, making it a viable approach for RISC-V-based computing systems.**

This project implements a RISC-V processor using Verilog, supporting both sequential and pipelined architectures. The sequential processor executes one instruction at a time, while the pipelined design improves performance through a five-stage execution model: Fetch, Decode, Execute, Memory, and Write Back. Pipeline registers enable parallel execution, and hazard mitigation techniques like data forwarding and branch prediction ensure correctness. The processor supports key RISC-V instructions (add, sub, and, or, ld, sd, beq) and is verified using unit tests, assembly programs, and automated testbenches.

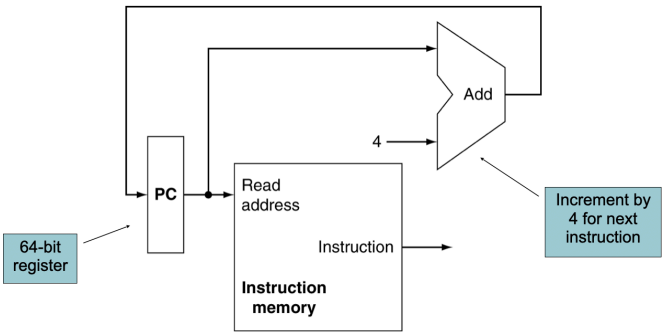
SEQUENTIAL DESIGN

The following diagram is a basic overview of the Processor we are trying to implement, and the different modules we will be implementing. It is a 5-stage design (not to be confused with a 5 stage pipeline) - Fetch, Decode, Execute, Memory Access, and Write Back.



A. Fetch

The **fetch stage** retrieves instructions from memory and updates the **Program Counter (PC)**. The PC module ensures sequential execution and handles branching.



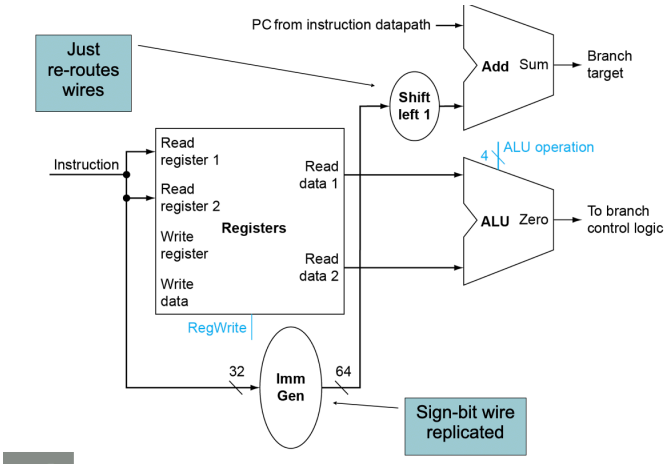
PC Module Functionality

- On **reset (rst)**, instruction fetching is disabled (ce = 0), and PC is set to 0.
- Once reset is deactivated, **PC increments by 4** each cycle to fetch the next instruction.
- If a **branch (Branch = 1)** occurs, **PC is updated to Addr**, redirecting execution.

```
module PC(
    input   wire        clk,
    input   wire        rst,
    input   wire        Branch,  // if branch or not
    input   wire[31:0]  Addr,    // target address

    output  reg         ce,
    output  reg [31:0]  PC
);

always @ (posedge clk) begin
    if (rst)
        ce <= 1'b0;
    else
        ce <= 1'b1;
end

always @ (posedge clk) begin
    if (!ce)
        PC <= 32'b0;
    else if (Branch)
        PC <= Addr;
    else
        PC <= PC + 4'h4;
end

endmodule
```

B. ***Decode***

The **decode stage** translates the fetched instruction into control signals and extracts relevant operand values. It identifies the instruction type, determines register dependencies, and prepares the data for execution.



Functionality of the ID Module

- Extracts **source (rs1, rs2) and destination (rd) registers** from the instruction.
- Determines the **ALU operation (ALUop)** based on instruction type.
- Generates **control signals** for register reads (RegRead1, RegRead2) and writes (WriteReg).
- Computes **immediate values (imm)** for instructions requiring them.
- Calculates **branch target addresses (BranchAddr)** and determines whether a **branch should be taken (Branch)**.

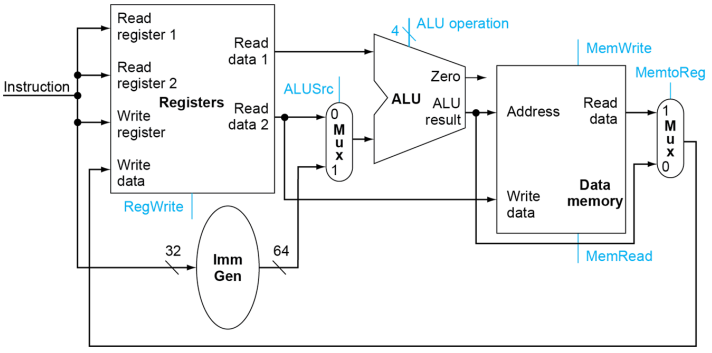
```
always @ (*) begin
    if (rst)
        Branch <= 1'b0;
    else if (inst_i[6:0] ==
       7'b1101111 || (inst_i[6:0] == 7'b1100011
       && inst_i[13:12] == 2'b0))  // jal, beq, blt
        Branch <= 1'b1;
    else
        Branch <= 1'b0;
end

always @ (*) begin
    if (rst)
        WriteData <= 5'b0;
    else
        WriteData <= rd_addr;
end
```

```
always @ (*) begin
    if (rst)
        RegAddr1 <= 5'b0;
    else
        RegAddr1 <= rs1_addr;
end

always @ (*) begin
    if (rst)
        RegAddr2 <= 5'b0;
    else
        RegAddr2 <= rs2_addr;
end

always @ (*) begin
    if (rst)
        Reg1 <= 32'b0;
    else if (RegRead1)
        Reg1 <= RegData1;
    else
        Reg1 <= imm;
end

always @ (*) begin
    if (rst)
        Reg2 <=  32'b0;
    else if (RegRead2)
        Reg2 <= RegData2;
    else
        Reg2 <= imm;
end
```

C. ***Execute***

The **execute (EX) stage** performs arithmetic and logical operations based on the instruction type. In this modified design, **data forwarding** is integrated to handle data hazards, ensuring correct values are used without unnecessary stalls.

Functionality of the EX Module

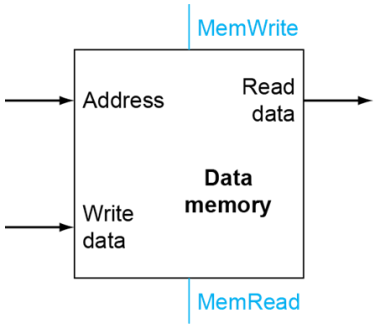


- **ALU Operation:**
  - The ALU performs the computation using **operands** to ensure the correct values are used.
  - ALU has Add, Sub, Addi, And, Or, Xor functionalities in the arithmetic domain.

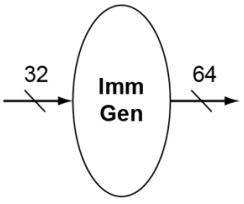
- It also has SLL, SRL, SLU and other operations as well as shifting operations.
- During Store, Load and No-op it has an all 0 output as the ALU is not supposed to generate an output at those commands.
- It also passes the values for Load and Store values.

D. ***Memory***

The **memory (MEM) stage** handles **load (LW) and store (SW) operations**, reading from or writing to memory based on control signals. This module ensures that memory operations are correctly executed within the pipeline.



a. Data memory unit



b. Immediate generation unit

Functionality of the Memory Module

- **Reading from Memory (Dm1, Dm2)**
  - If an address (Am1, Am2) is provided, the module retrieves **4 bytes** from memory and stores them in Dm1 or Dm2.
- **Writing to Memory (RW = 1)**
  - If RW = 1, the module writes **4 bytes** from Dm3 into memory at address Am2.

E. ***Write Back***

The **write back (WB) stage** is the final step in the execution cycle, responsible for updating the register file with the results of computations or memory operations.

Functionality of the WB Module

- If **IsWb = 1**, the instruction writes back to a register.
- If **IsCall = 1**, the return address (`pc_current + 4`) is written to register `x1` (typically used for function calls).
- If **IsLd = 1**, the result from memory (`LdResult`) is written back.
- Otherwise, the **ALU computation (`AluResult`)** is stored in the destination register (`Rd`).

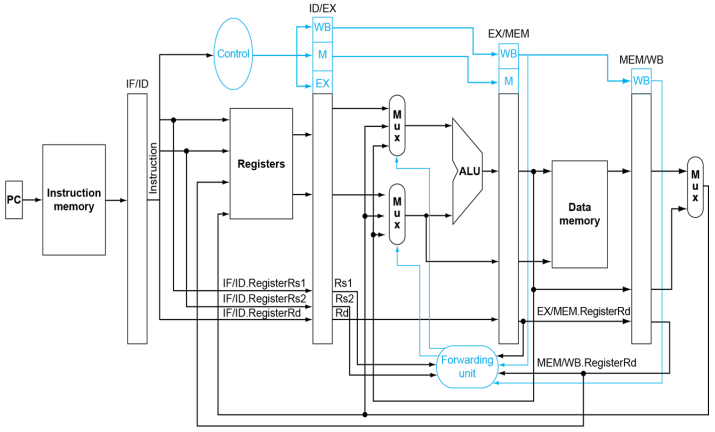
```
always @ (*) begin
    if (rst)
        WriteBackNum <= 5'b0;
    else
        WriteBackNum <= MemWriteNum;
end

always @ (*) begin
    if (rst)
        WriteBackReg <= 1'b0;
    else
        WriteBackReg <= MemWriteReg;
end

always @ (*) begin
    if (rst)
        WriteBackData <= 32'b0;
    else
        WriteBackData <= MemWriteData;
end
```

E. ***Challenges***

- **Instruction Dependency:** Sequential execution requires each instruction to complete before the next starts, causing delays.
- **Memory Access Timing:** Ensuring correct load and store operations without conflicts.
- **Register Write Timing:** Managing correct data propagation and avoiding overwrites.

PIPELINED DESIGN

The following diagram is a basic overview of the Processor we are trying to implement, and the different modules we will be implementing. It is a 5-stage pipelined design- Fetch, Decode, Execute, Memory Access, and Write Back, with a proper control unit, forwarding circuit and a branch prediction unit.



A. Instruction Fetch (IF)

The Instruction Fetch (IF) module is responsible for retrieving instructions from memory for execution. At its core, this module maintains and updates the Program Counter (PC), ensuring that the processor knows which instruction to fetch next. It also incorporates essential control mechanisms such as branch handling, pipeline stalls, and branch prediction to optimize execution flow and minimize delays.

```
always @ (posedge clk) begin
  if (rst)
    ce <= 1'b0;
  else
    ce <= 1'b1;
end

always @ (posedge clk) begin
  if (!ce)
    PC <= 32'b0;
  else if (!(stall[0])) begin
    if (Branch)
      PC <= Addr;
    else if (PreBranch)
      PC <= PreAddr;
    else
      PC <= PC + 4'h4;
  end
end
```

Key Functionalities:

- **Instruction Address Management:**
  - Maintains the current instruction address (PC).
  - Increments the PC sequentially (PC + 4) for normal instruction flow.
- **Branch Handling:**
  a. If a branch instruction is taken (Branch = 1), the PC updates to the correct branch target (Addr).
  b. If branch prediction is active (PreBranch = 1), the PC speculatively updates to the predicted address (PreAddr).
- **Pipeline Stall Support:**
  a. When a stall signal (stall[0] = 1) is detected, the PC remains unchanged, preventing incorrect instruction fetching.
- **Clock and Reset Control:**
  a. The PC is reset (PC = 0) when rst = 1, disabling instruction fetch.
  b. Once reset is deasserted, the module resumes normal execution

B. IF / ID

The PC_ID module serves as a pipeline register between the Instruction Fetch (IF) stage and the Instruction Decode (ID) stage in our pipelined RISC-V processor. Its primary role is to hold and transfer the instruction address (PC) and instruction data (Inst) from the IF stage to the ID stage while handling stalls and resets. Additionally, it propagates branch prediction information to assist in speculative execution.

```
always @ (*) begin
    if (rst)
        idPC <= 32'b0;
    else if (!stall[1])
        idPC <= ifPC;
    else if (stall[2:1] ==2'b01)
        idPC <= 32'b0;
end

always @ (*) begin
    if (rst)
        idInst <= 32'b0;
    else if (!stall[1])
        idInst <= ifInst;
    else if (stall[2:1] ==2'b01)
        idInst <= 32'b0;
end
```

```
always @ (*) begin
    if (rst)
        Predict_o <= 1'b0;
    else if (!stall[1])
        Predict_o <= Predict_i;
    else if (stall[2:1] ==2'b01)
        Predict_o <= 1'b0;
end
```

Key Functionalities:

- **Instruction Address Handling (idPC)**
    - Transfers the instruction fetch address (ifPC) to the decode stage (idPC).
    - On reset (rst = 1), idPC is cleared to **0**.
    - If no stall is detected (stall[1] = 0), the module updates idPC with the latest ifPC.
    - If a pipeline flush is needed (stall[2:1] = 2'b01), idPC is reset to **0** to discard incorrect instructions.

- **Instruction Data Handling (idInst)**
    a. Transfers the fetched instruction (ifInst) to the decode stage (idInst).
    b. On reset, idInst is cleared to **0**.
    c. If no stall is present, idInst updates normally.
    d. If a flush is required, idInst is reset to **0**, preventing incorrect instruction execution.

- **Branch Prediction Propagation (Predict_o)**
    a. Passes branch prediction information (Predict_i) to the next stage (Predict_o).
    b. On reset, Predict_o is set to **0**.
    c. If the pipeline is not stalled, Predict_o updates with Predict_i.
    d. If a flush occurs, Predict_o is reset, ensuring incorrect predictions are discarded.

C. Instruction Decode (ID)

The Instruction Decode (ID) module is responsible for decoding the fetched instruction and extracting key control signals required for execution. It retrieves register values from the register file, determines the type of operation, and generates control signals that guide the Execution stage. The module handles register read and write operations, immediate extraction, and forwarding mechanisms when necessary. Additionally, it ensures proper hazard detection to maintain pipeline integrity.

Key Functionalities:

- **Instruction Decoding**: Parses the instruction to identify opcode, function codes, source, and destination registers.
- **Register File Access**: Reads register values based on source register addresses.
- **Immediate Generation**: Extracts immediate values for arithmetic and memory operations.
- **Control Signal Generation**: Produces signals required for ALU operations, memory access, and write-back.
- **Hazard Detection**: Identifies and handles data hazards through forwarding or stalling mechanisms.

```
/* Notes:
 ********************************************************************************
 * Inst * ALUSrc * MemToReg * RegWrite * MemRead * MemWrite * Branch * ALUOp1 * ALUOp0 *
 ********************************************************************************
 *   R  *    0    *    0    *    1    *    0    *    0    *    0    *    1    *    0   *
 ********************************************************************************
 *  lw  *    1    *    1    *    1    *    1    *    0    *    0    *    0    *    0   *
 ********************************************************************************
 *  sw  *    1    *    x    *    0    *    0    *    1    *    0    *    0    *    0   *
 ********************************************************************************
 * beq  *    0    *    x    *    0    *    0    *    0    *    1    *    0    *    1   *
 ********************************************************************************
 */
```

D. ID / EX

The ID/EX pipeline register is responsible for storing intermediate values between the Instruction Decode (ID) and Execution (EX) stages in the pipeline. It ensures that decoded instruction information, including register values and control signals, is properly transferred to the execution stage. Additionally, it facilitates RS1 and RS2 forwarding, which helps in handling data dependencies by keeping track of source registers and forwarding their values when necessary. The inclusion of stall control ensures that the pipeline remains synchronized and prevents unintended overwrites.

```
always @(posedge clk) begin
    if (rst) begin
        exALUop   <= 5'b0;
        exReg1    <= 32'b0;
        exReg2    <= 32'b0;
        exWriteNum <= 5'b0;
        exWriteReg <= 1'b0;
        exRS1     <= 5'b0;
        exRS2     <= 5'b0;
        exInst    <= 32'b0;
    end else if (!stall[2]) begin
        exALUop   <= idALUop;
        exReg1    <= idReg1;
        exReg2    <= idReg2;
        exWriteNum <= idWriteNum;
        exWriteReg <= idWriteReg;
        exRS1     <= idRS1;
        exRS2     <= idRS2;
        exInst    <= idInst;
    end
end
```

Key Functionalities:

- **Pipeline Data Storage**: Holds ALU operation codes, register values, and write-back signals between ID and EX stages.
- **Register Forwarding**: Passes RS1 and RS2 values to help resolve data hazards in subsequent pipeline stages.
- **Stall Handling**: Uses a stall signal to prevent updates when a stall is detected, maintaining pipeline integrity.
- **Instruction Transfer**: Passes the current instruction to the execution stage for reference and debugging.
- **Reset Mechanism**: Clears all stored values when a reset signal is asserted.

```
always @(posedge clk) begin
    if (rst) begin
        exALUop   <= 5'b0;
        exReg1    <= 32'b0;
        exReg2    <= 32'b0;
        exWriteNum <= 5'b0;
        exWriteReg <= 1'b0;
        exRS1     <= 5'b0;
        exRS2     <= 5'b0;
        exInst    <= 32'b0;
    end else if (!stall[2]) begin
        exALUop   <= idALUop;
        exReg1    <= idReg1;
        exReg2    <= idReg2;
        exWriteNum <= idWriteNum;
        exWriteReg <= idWriteReg;
        exRS1     <= idRS1;
        exRS2     <= idRS2;
        exInst    <= idInst;
    end
end
```

E. Execute (EX)

The Execution (EX) stage is a critical component of the RISC-V processor pipeline. It performs arithmetic and logical operations using the ALU (Arithmetic Logic Unit) and resolves data hazards with a forwarding mechanism. The EX stage receives operands from the ID/EX pipeline register and processes them based on the decoded ALU operation (ALUop_i). Forwarding logic ensures correct data is used, even when dependencies exist between instructions in the pipeline. The EX stage also determines the result that will be passed to the next pipeline stage, either for memory access or writeback.

```
ForwardingUnit FU(
    .ID_EX_RS1(RS1),
    .ID_EX_RS2(RS2),
    .EX_MEM_RD(EX_MEM_RD),
    .MEM_WB_RD(MEM_WB_RD),
    .EX_MEM_RegWrite(EX_MEM_RegWrite),
    .MEM_WB_RegWrite(MEM_WB_RegWrite),
    .ForwardA(ForwardA),
    .ForwardB(ForwardB)
);

// Forwarding Mux for Oprend1
always @(*) begin
    case (ForwardA)
        2'b10: ForwardedOprend1 = EX_MEM_ALUResult;
        2'b01: ForwardedOprend1 = MEM_WB_Data;
        default: ForwardedOprend1 = Oprend1;
    endcase
end

// Forwarding Mux for Oprend2
always @(*) begin
    case (ForwardB)
        2'b10: ForwardedOprend2 = EX_MEM_ALUResult;
        2'b01: ForwardedOprend2 = MEM_WB_Data;
        default: ForwardedOprend2 = Oprend2;
    endcase
end
```

```
ADD_32 u_add (
    .A(ForwardedOprend1),
    .B(ForwardedOprend2),
    .Sum(add_result),
    .Cout(dummy_cout)
);

SUB_32 u_sub (
    .A(ForwardedOprend1),
    .B(ForwardedOprend2),
    .Diff(sub_result),
    .Cout(dummy_cout)
);

SLL_32 u_sll (
    .in(ForwardedOprend1),
    .shamt(ForwardedOprend2[4:0]),
    .out(sll_result)
);

SRL_32 u_srl (
    .in(ForwardedOprend1),
    .shamt(ForwardedOprend2[4:0]),
    .out(srl_result)
);

XOR_32 u_xor (
    .A(ForwardedOprend1),
    .B(ForwardedOprend2),
    .Y(xor_result)
);

OR_32 u_or (
    .A(ForwardedOprend1),
    .B(ForwardedOprend2),
    .Y(or_result)
);

AND_32 u_and (
    .A(ForwardedOprend1),
    .B(ForwardedOprend2),
    .Y(and_result)
);
```

Key Functionalities:

- **Arithmetic and Logical Operations**: Supports addition, subtraction, bitwise AND, OR, XOR, shift left logical (SLL), shift right logical (SRL), and shift right arithmetic (SRA).
- **Forwarding Mechanism**: Resolves data hazards by selecting the correct operand values from previous pipeline stages (EX/MEM or MEM/WB).
- **Pipeline Control**: Maintains correct data flow in the presence of hazards by handling forwarded values dynamically.
- **ALU Execution**: Computes the result based on the given instruction and forwards it to the **EX/MEM** register for further processing.
- **Instruction Execution for Branching and Memory Operations**: Processes instructions such as JAL, BEQ, BLT, LW, and SW, ensuring proper computation for address calculations and conditional execution.

```
always @(*) begin
       if (rst) begin
           ALUResult = 32'b0;
           WriteDataNum_o = 5'b0;
           WriteReg_o = 1'b0;
           WriteData_o = 32'b0;
       end else begin
           case (ALUop_i)
               5'b10000: WriteData_o = Oprend1 + Oprend2; // JAL (example)
               5'b10001: WriteData_o = Oprend1 + Oprend2; // BEQ (example)
               5'b10010: WriteData_o = Oprend1 + Oprend2; // BLT (example)
               5'b10100: WriteData_o = Oprend1 + Oprend2; // LW (example)
               5'b10101: WriteData_o = Oprend1 + Oprend2; // SW (example)
               5'b01100: WriteData_o = add_result;       // ADDI
               5'b01101: WriteData_o = add_result;       // ADD
               5'b01110: WriteData_o = sub_result;       // SUB
               5'b01000: WriteData_o = sll_result;       // SLL
               5'b00110: WriteData_o = xor_result;       // XOR
               5'b01001: WriteData_o = srl_result;       // SRL
               5'b00101: WriteData_o = or_result;        // OR
               5'b00100: WriteData_o = and_result;       // AND
               default:  WriteData_o = 32'b0;            // NOP or invalid op
           endcase

           // Assign ALUResult based on the operation
           ALUResult = WriteData_o;
           WriteDataNum_o = WriteDataNum_i;
           WriteReg_o = WriteReg_i;
       end
end
```

F. EX / MEM

The EX/MEM pipeline register is a key stage in the RISC-V pipeline, bridging the Execution (EX) stage and the Memory Access (MEM) stage. It temporarily holds the results of ALU computations, memory addresses, and register write-back data until the memory stage is ready to process them.

Key Functionalities:

- **Data Forwarding from Execution to Memory Stage**
    - Stores and transfers ALU results, memory addresses, and register values to the next stage.
- **Handling of Pipeline Stalls**
    - If stall[3] is asserted, the register holds its current value.
    - If stall[4:3] == 2'b01, the register is flushed, preventing unintended writes.

```
always @ (posedge clk) begin
    if (rst)
        memWriteNum <= 5'b0;
    else if (stall[4:3] == 2'b01)
        memWriteNum <= 5'b0;
    else if (!stall[3])
        memWriteNum <= exWriteNum;
end

always @ (posedge clk) begin
    if (rst)
        memWriteReg <= 1'b0;
    else if (stall[4:3] == 2'b01)
        memWriteReg <= 1'b0;
    else if (!stall[3])
        memWriteReg <= exWriteReg;
end

always @ (posedge clk) begin
    if (rst)
        memWriteData <= 32'b0;
    else if (stall[4:3] == 2'b01)
        memWriteData <= 32'b0;
    else if (!stall[3])
        memWriteData <= exWriteData;
end

always @ (posedge clk) begin
    if (rst)
        memALUop <= 5'b0;
    else if (stall[4:3] == 2'b01)
        memALUop <= 5'b0;
    else if (!stall[3])
        memALUop <= exALUop;
end

always @ (posedge clk) begin
    if (rst)
        memAddr <= 32'b0;
    else if (stall[4:3] == 2'b01)
        memAddr <= 32'b0;
    else if (!stall[3])
        memAddr <= exAddr;
end

always @ (posedge clk) begin
    if (rst)
        memReg <= 32'b0;
    else if (stall[4:3] == 2'b01)
        memReg <= 32'b0;
    else if (!stall[3])
        memReg <= exReg;
end
```

G. Memory (MEM)

The MEM stage of the pipeline is responsible for handling memory operations such as load (lw) and store (sw) instructions. It interacts with the data memory and ensures that data is read or written as required.

Key Functionalities:

- **Memory Read and Write Operations**
  - Reads from memory for **load (lw)** instructions.
  - Writes to memory for **store (sw)** instructions.
- **Pipeline Register Forwarding**
  - Receives values from the EX/MEM stage and passes necessary outputs to the MEM/WB stage.
- **Memory Control Signals**
  - MemCE_o (Memory Chip Enable): Enables memory access when a memory instruction is detected.
  - MemWE_o (Memory Write Enable): Determines whether memory is being written

```
always @ (*) begin
    if (rst)
        MemData_o <= 32'b0;
    else if (ALUop_i == 5'b10101)  // sw
        MemData_o <= Reg_i;
end

always @ (*) begin
    if (rst)
        WriteData_o <= 32'b0;
    else begin
        WriteData_o <= WriteData_i;
        if (ALUop_i == 5'b10100)  // lw
            WriteData_o <= MemData_i;
    end
end

always @ (*) begin
    if (rst)
        MemAddr_o <= 32'b0;
    else begin
        if (ALUop_i == 5'b1010x)  // lw or sw
            MemAddr_o <= MemAddr_i;
        else
            MemAddr_o <= 32'b0;
    end
end

always @ (*) begin
    if (rst)
        MemCE_o <= 1'b0;
    else begin
        if (ALUop_i == 5'b1010x)  // lw or sw
            MemCE_o <= 1'b1;
        else
            MemCE_o <= 1'b0;
    end
end
```

H. MEM / WB

The MEM/WB module serves as the final pipeline register between the Memory (MEM) stage and the Write-Back (WB) stage in a pipelined RISC-V processor. It ensures that necessary values are passed from the memory stage to the register file in the final stage.

Key Functionalities:

- **Registers the Write-Back Data**
  - Transfers data from the MEM stage to the WB stage.
- **Handles Pipeline Stalling**
  - Uses stall[5:4] to determine if the pipeline should halt or pass data forward.
- **Preserves Write-Back Information**
  - Keeps track of which register needs to be written and what data should be written.

```
always @ (posedge clk) begin
    if (rst)
        wbWriteNum <= 5'b0;
    else if (stall[5:4] == 2'b01)
        wbWriteNum <= 5'b0;
    else if (!stall[4])
        wbWriteNum <= MemWriteNum;
end

always @ (posedge clk) begin
    if (rst)
        wbWriteReg <= 1'b0;
    else if (stall[5:4] == 2'b01)
        wbWriteReg <= 1'b0;
    else if (!stall[4])
        wbWriteReg <= MemWriteReg;
end

always @ (posedge clk) begin
    if (rst)
        wbWriteData <= 32'b0;
    else if (stall[5:4] == 2'b01)
        wbWriteData <= 32'b0;
    else if (!stall[4])
        wbWriteData <= MemWriteData;
end
```

I. Write Back (WB)

The **WB (Write-Back) module** is the final stage in a pipelined RISC-V processor, where values are written back to the register file.

Key Functionalities:

- **Handles Register Write-Back**
  - Passes MemWriteNum (destination register number), MemWriteReg (write enable), and MemWriteData (data) to the register file.
- **Implements Reset Behavior**
  - Ensures that during a reset (rst is high), all outputs are cleared.

```
always @ (*) begin
    if (rst)
        WriteBackNum <= 5'b0;
    else
        WriteBackNum <= MemWriteNum;
end

always @ (*) begin
    if (rst)
        WriteBackReg <= 1'b0;
    else
        WriteBackReg <= MemWriteReg;
end

always @ (*) begin
    if (rst)
        WriteBackData <= 32'b0;
    else
        WriteBackData <= MemWriteData;
end
```

J. Stall

The STALL module is responsible for controlling pipeline stalls in a RISC-V processor. It generates a stall signal based on pipeline hazards such as load-use dependencies and branch stalls.

Key Functionalities:

- **Handles Pipeline Stalls**
  - If a **load-use hazard** occurs (StallLoad is asserted), it generates a stall for multiple pipeline stages.
  - If a **branch hazard** occurs (StallBranch is asserted), it generates a stall for fewer stages.
- **Implements Reset Behavior**
  - When rst is active, the stall signal is cleared.

```
always @ (*) begin
    if (rst)
        stall <= 6'b0;
    else if (StallBranch)
        stall <= 6'b000010;
    else if (StallLoad)
        stall <= 6'b000111;
    else
        stall <= 6'b0;
end
```

K. Forwarding Unit

The Forwarding Unit helps resolve data hazards in a pipelined RISC-V processor by forwarding values from later stages back to the EX stage, reducing the need for stalls.

```
always @(*) begin
        if (EX_MEM_RegWrite && (EX_MEM_RD != 0) && (EX_MEM_RD == ID_EX_RS1))
            ForwardA = 2'b10;
        else if (MEM_WB_RegWrite && (MEM_WB_RD != 0) && (MEM_WB_RD == ID_EX_RS1))
            ForwardA = 2'b01;
        else
            ForwardA = 2'b00;
end

always @(*) begin
        if (EX_MEM_RegWrite && (EX_MEM_RD != 0) && (EX_MEM_RD == ID_EX_RS2))
            ForwardB = 2'b10;
        else if (MEM_WB_RegWrite && (MEM_WB_RD != 0) && (MEM_WB_RD == ID_EX_RS2))
            ForwardB = 2'b01;
        else
            ForwardB = 2'b00;
end
```

Key Functionalities:

- **Detect Data Hazards**
  - Identifies when an instruction in the EX stage depends on a previous instruction in the EX/MEM or MEM/WB stage.
- **Forward Data to ALU Inputs**
  - Determines whether the ALU operands (RS1 and RS2) should come from:
  - EX/MEM stage (ALU result)
  - MEM/WB stage (write-back data)
  - Register file (no forwarding needed)
- **Control Forwarding Paths**
  - Sets ForwardA and ForwardB to decide which source to use for ALU computation:
  - 2'b10: Forward data from EX/MEM stage (ALU result)
  - 2'b01: Forward data from MEM/WB stage (write-back data)

    - 2'b00: No forwarding, use the register file data
- **Prevent Incorrect Register Reads**
    - Ensures that if forwarding occurs, the processor does not incorrectly use outdated register values**.**
- **Works with the Stall Unit**
    - Forwarding reduces the number of stalls, but load-use hazards (when an instruction depends on a load instruction) may still require stalling.

L. Branch Prediction

The Branch Predictor improves instruction pipeline efficiency by predicting whether a branch instruction will be taken or not before the branch is resolved. This helps in reducing stalls and pipeline flushes.

Key Functionalities:

- **Branch Prediction Mechanism**
    - Detects **branch instructions** (opcode 1100011).
    - Uses **history-based prediction** (global & local history tracking).

Prediction Strategies

- **Global Prediction (GlobalRec & GlobalPre)**
    - Uses a **global history register** (GlobalRec), updated with branch outcomes.
    - Global prediction table (GlobalPre) holds 2-bit **saturating counters** to predict future branches.

- **Local Prediction (LocalAddrRec & LocalPre)**
    - Uses a **local history table** indexed by the instruction's address.
    - Maintains **separate 2-bit counters** for local branch predictions.

- **Prediction Updates (Adaptive Learning)**
    - If the branch **was taken** but **not predicted**, increase confidence (+1).
    - If the branch **was not taken** but **predicted**, decrease confidence (-1).
    - Saturating counters prevent overflows.

```
always @ (*) begin
    if (rst)
        for(i = 0; i < 4096; i = i + 1)
      AllAddrRec[i] = 10'b0;
end

always @ (*) begin
    if (rst)
        for(i = 0; i < 1024; i = i + 1)
      LocalAddrRec[i] <= 10'b0;
end

always @ (*) begin
    if (rst)
        GlobalRec <= 1'b0;
    else
        GlobalRec <= (GlobalRec << 1 | Accept);
end

always @ (*) begin
    if (rst)
        for(i = 0; i < 4096; i = i + 1)
      GlobalPre[i] = 2'b0;
    else if ( Branch && Predict && idSel && (GlobalPre[GlobalRec] < 3) )
        GlobalPre[GlobalRec] = GlobalPre[GlobalRec] + 1;
    else if ( Branch && !Predict && idSel && (GlobalPre[GlobalRec] > 0) )
        GlobalPre[GlobalRec] = GlobalPre[GlobalRec] - 1;
end

always @ (*) begin
    if (rst)
        for(i = 0; i < 1024; i = i + 1)
      LocalPre[i] <= 2'b0;
    else if ( Branch && Predict && !idSel && (LocalPre[LocalAddrRec[idPC[11:2]]] < 3) )
        LocalPre[LocalAddrRec[idPC[11:2]]] <= LocalPre[LocalAddrRec[idPC[11:2]]] + 1;
    else if ( Branch && !Predict && !idSel && (LocalPre[LocalAddrRec[idPC[11:2]]] > 0) )
        LocalPre[LocalAddrRec[idPC[11:2]]] <= LocalPre[LocalAddrRec[idPC[11:2]]] - 1;
end

always @ (*) begin
    if (rst)
        PreBranch <= 1'b0;
    else if (inst_i[6:0] == 7'b1100011)
        PreBranch <= 1'b1;
end

always @ (*) begin
    if (rst)
        PreAddr <= 32'b0;
    else if (inst_i[6:0] == 7'b1100011) begin
        if ( AllPre[AllAddrRec[pc_i[13:2]]][1] ) begin
            if ( GlobalPre[GlobalRec][1] )
                PreAddr <= pc_i + {{20{inst_i[31]}}, inst_i[7], inst_i[30:25], inst_i[11:8],
1'b0};
            else
                PreAddr <= pc_i + 4;
        end
        else begin
            if ( LocalPre[LocalAddrRec[pc_i[11:2]]][1] )
                PreAddr <= pc_i + {{20{inst_i[31]}}, inst_i[7], inst_i[30:25], inst_i[11:8],
1'b0};
            else
                PreAddr <= pc_i + 4;
        end
    end
end

always @ (*) begin
    if (rst)
        PreAccept <= 1'b0;
    else if (inst_i[6:0] == 7'b1100011) begin
        if ( AllPre[AllAddrRec[pc_i[13:2]]][1] ) begin
            if (GlobalPre[GlobalRec][1])
                PreAccept <= 1'b1;
            else
                PreAccept <= 1'b0;
        end
        else begin
            if ( LocalPre[LocalAddrRec[pc_i[11:2]]][1] )
                PreAccept <= 1'b1;
            else
                PreAccept <= 1'b0;
        end
    end
end

always @ (*) begin
  if (rst)
    PreSel <= 1'b0;
  else if (inst_i[6:0] == 7'b1100011)
    if (AllPre[AllAddrRec[pc_i[13:2]]][1])
      PreSel <= 1'b1;
      else
      PreSel <= 1'b0;
    else
        PreSel <= 1'b0;
end
```

Test Cases & Explanation

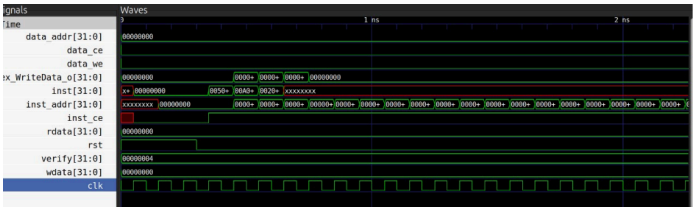
Following are the Test cases ran and their output both in GTK-WAVE and the Terminal outputs -

- **Test Case 1**

00000000010100000000000010010011
00000000101000000000000100010011
00000000001000001000000110110011

addi x1, x0, 5
addi x2, x0, 10
add x3, x2, x1

```
VCD info: dumpfile riscv_soc_tb.vcd opened for output.
x 1 = 00000005
x 2 = 0000000a
x 3 = 0000000f
---      result is          4          ---
```



- **Test Case 2**

00000000010100000000000010010011
00000000101000000000000100010011
00000000001000001000000110110011
00000000001100101010000000100011
00000000010100000000001010010011
00000000010100000000001010010011
00000000000000010101000100000011
01000000001100100000001100110011

addi x1, x0, 5
addi x2, x0, 10
add x3, x1, x2
sw x3, 0(x5)
addi x5, x0, 5
addi x5, x0, 5
lw x4, 0(x5)
sub x6, x4, x3

```
VCD info: dumpfile riscv_soc_tb.vcd opened for output.
x 1 = 00000005
x 2 = 0000000a
x 3 = 0000000f
x 5 = 00000005
x 4 = 00000005
x 5 = 00000005
x 6 = fffffff6
```

- **Test Case 3**

00000000011100000000000010010011
00000000011100000000000100010011
00000000001000001000010001100011
00000000000000000000000110010011
00000000000100000000000110010011

addi x1, x0, 7
addi x2, x0, 7
beq x1, x2, BRANCH
addi x3, x0, 1
BRANCH:
addi x3, x0, 0

```
x 1 = 00000007
x 2 = 00000007
x 3 = 00000000
---      result is          4          ---
```

Work Contribution

Harsh Kapoor & Anshitha Penubolu: Took primary responsibility for writing the Verilog code, ensuring that all modules were implemented correctly and on time. The report was also written by them and split equally.
Guru Shwadeep: Contributed by debugging the code, identifying and fixing issues to improve functionality. Additionally, he integrated the codes and generated GTKWave plots for analysis and created test cases to validate processor execution.

Conclusion

Congratulations! You've just survived a deep dive into the wild world of RISC-V processor design. If you've made it this far, either you're really passionate about computer architecture or you've lost a bet. Either way, let's wrap this up!
This project successfully brings a RISC-V-based processor to life using Verilog, assembling a team of modules that work together like a well-oiled (mostly) pipeline. Each module plays a crucial role—fetching, decoding, executing, memory accessing, and writing back—like a group project where, for once, everyone actually does their part.

Key Takeaways:

- **Pipeline Efficiency**: Thanks to forwarding and stall control, our processor doesn't trip over itself trying to execute instructions. Think of it as dodging traffic jams but for data.
- **Memory Operations**: The MEM and MEM_WB modules handle reads and writes

without losing their minds (or data). No memory leaks, just smooth transactions.

- **Branch Prediction**: Our BRANCH_PRE module does its best to predict the future—sometimes correctly, sometimes like a weather forecast.
- **Write-Back Control**: The WB module makes sure registers get updated properly—because what's the point of computing if you forget the answer?

Final Thoughts:

This processor is a solid first step toward building more complex architectures. Want to add superscalar execution, out-of-order processing, or speculative execution? Go for it! Just remember: with great power comes great debugging sessions.
In the end, this project proves that with some clever logic, careful design, and a few gallons of coffee, you can turn a pile of Verilog code into a working processor. Now go forth and grade us with those sweet sweet A’s.