Implementation of Superscalar Processor using Verilog

Done by Naman Kalra, EE23B032

Under the supervision of Prof. Jaynarayan Tudu

Super-Scalar Processor Design and Implementation Thesis

- **Branch Prediction**
- **Tomasulo's Algorithm**
- **Dynamic Speculation and Recovery**

Contents

1 Introduction

A superscalar processor is a type of CPU architecture designed to increase performance by executing multiple instructions simultaneously. Unlike scalar processors, which execute one instruction per clock cycle, superscalar processors can issue and execute several instructions per cycle, leveraging multiple execution units. This capability allows them to exploit Instruction-Level Parallelism (ILP) more effectively, which is crucial for enhancing computational throughput and efficiency.

1.1 Instruction Fetch Unit (IFU)

Components:

- **Instruction Cache:** Stores frequently accessed instructions to minimize delays due to memory access. It often uses a cache hierarchy (L1, L2) to speed up fetching.
- **Branch Target Buffer (BTB):** Caches the target addresses of recently executed branches to speed up instruction fetching for branch instructions.
- **Instruction Queue:** Holds fetched instructions before they are passed to the decode stage. It often operates as a FIFO queue to ensure instructions are processed in the order they are fetched.

1.2 Instruction Decode Unit (IDU)

Components:

- **Decoder:** Converts instructions into signals that specify the operation to be performed and the necessary execution units. It also identifies the source and destination registers.
- **Dispatch Logic:** Determines how decoded instructions are allocated to different execution units. It may use FIFO buffers to manage the order of dispatch.

1.3 Reservation Stations

Components:

- **Instruction Buffers:** Store instructions until all operands are available. They act like a FIFO queue where instructions wait in the order they were dispatched.
- **Tagging Mechanism:** Each reservation station entry includes tags to identify which instructions are waiting for which operands. This mechanism helps in tracking dependencies.

1.4 Execution Units

Components:

- **Integer Units:** Handle arithmetic and logical operations on integer data. They operate independently of floating-point units and memory units.

- **Floating-Point Units (FPUs):** Perform operations on floating-point numbers. They are optimized for complex mathematical calculations.

- **Load/Store Units:** Manage memory operations, including loading data from and storing data to memory.

1.5 Common Data Bus (CDB)

Components:

- **Result Bus:** Carries results from execution units to reservation stations and the reorder buffer. It operates as a broadcast bus where results are broadcast to all waiting reservation stations.

- **Operand Forwarding:** Results on the CDB are used to satisfy pending operand requests in reservation stations. This ensures that instructions waiting for operands can proceed once the results are available.

1.6 Reorder Buffer (ROB)

Components:

- **Queue:** Manages instructions that have completed execution but need to be committed in order. It operates as a FIFO queue to ensure that results are committed in the original program order.

- **Commit Logic:** Controls the process of writing results to the architectural state. It ensures that results are committed in the correct order and handles the retirement of instructions from the ROB.

1.7 Branch Prediction Unit (BPU)

Components:

- **Branch History Table (BHT):** Stores recent branch outcomes to predict the direction of branches. It helps in reducing stalls caused by branch instructions.

- **Pattern History Table (PHT):** Tracks patterns of branch behavior to improve prediction accuracy. It often works with the BHT to refine predictions.

1.8 Speculation Management

Components:

- **Checkpointing:** Periodically saves the processor state to allow recovery from incorrect speculations. This is crucial for maintaining correct program execution when speculative instructions are involved.

- **Rollback Mechanism:** Reverts the processor to a previously saved state if speculative execution is found to be incorrect. This mechanism helps in discarding incorrect results and resuming correct execution.

2 Design and Verilog Implementation of Various Blocks

2.1 Branch Prediction Verilog Codes

In a superscalar processor, branch prediction is crucial for maintaining high instruction throughput by minimizing pipeline stalls due to branch instructions. The following sections describe the Verilog implementation of various components involved in branch prediction, including the Branch Target Buffer (BTB), Prediction Table, and the 2-bit saturating counter mechanism.

2.1.1 Branch Target Buffer (BTB)

The BTB is used to predict the target address of a branch instruction. It caches the target addresses of recently executed branches to speed up the instruction fetch process.

```
module BTB(
    output reg [31:0] BTB_Target,
    input [31:0] PC,
    input [31:0] BTB_Addr,
    input [31:0] BTB_Entry);
    reg [31:0] BTB_Table [0:15];
    integer i;
    initial begin
        for (i = 0; i < 16; i = i + 1) begin
            BTB_Table[i] = 32'b0;
        end
    end
    always @(PC) begin
        BTB_Target = BTB_Table[PC[3:0]];
    end
    always @(BTB_Addr or BTB_Entry) begin
        BTB_Table[BTB_Addr[3:0]] = BTB_Entry; // Update the BTB entry
    end
endmodule
```

Listing 1: BTB: Branch Target Buffer for caching branch target addresses.

2.1.2 Prediction Table

The Prediction Table stores the state of 2-bit saturating counters used for branch prediction. It helps in predicting whether a branch will be taken or not based on past behavior.

```
module PredictionTable(output takenOut,
    input [3:0] index,
    input [3:0] wIndex,input takenIn);
    reg [1:0] PT [0:15];
    reg [1:0] CS, CS_update;
    wire [1:0] NS;
    integer i;
    initial begin
        for (i = 0; i <= 15; i = i + 1) begin
```

```
            PT[i] = 2'b10; // Initial state
        end
    end
    always @(index) begin
        CS = PT[index];
    end
    always @(wIndex) begin
        CS_update = PT[wIndex];
    end
    always @(NS or wIndex) begin
        PT[wIndex] = NS;
    end
    Prediction_2bit pred(
        .NS(NS),
        .takenOut(takenOut),
        .taken(takenIn),
        .CS_read(CS),
        .CS_update(CS_update));
endmodule
```

Listing 2: Prediction Table for branch prediction.

2.1.3 Prediction_2bit

The Prediction_2bit module implements a 2-bit saturating counter mechanism used for branch prediction. This counter can be in one of four states, providing a more accurate prediction compared to a simple 1-bit predictor.

```
module Prediction_2bit(
    output reg [1:0] NS,output reg takenOut,
    input [1:0] CS_read,
    input [1:0] CS_update,input taken);
    parameter ST = 2'b00, T = 2'b01, NT =2'b10, SNT = 2'b11;
    always @(CS_read) begin
        case (CS_read)
            ST: takenOut = 1;
            T: takenOut = 1;
            NT: takenOut = 0;
            SNT: takenOut = 0;
        endcase
    end
    always @(CS_update or taken) begin
        case (CS_update)
            ST: NS = taken ? ST : T;
            T: NS = taken ? ST : SNT;
            NT: NS = taken ? ST : SNT;
            SNT: NS = taken ? NT : SNT;
        endcase
    end
endmodule
```

Listing 3: 2-bit Saturating Counter Mechanism for branch prediction.

2.1.4 1-bit Multiplexer

A 1-bit multiplexer selects between different branch predictions based on a selection signal.

```
module mux1bit(
    output out,
    input i0,
    input i1,
    input sel);
    assign out = sel ? i1 : i0;
endmodule
```

Listing 4: 1-bit Multiplexer for selecting between predictions.

2.1.5 Branch Predictor

The Branch Predictor module integrates various prediction strategies, including local and global history-based methods, to generate branch predictions.

```
module BranchPredictor(
    output prediction,
    input [31:0] PC,
    input [3:0] wPCindex,
    input taken);
    wire [3:0] lhtIndex, wlhtIndex;
    wire [1:0] localPrediction, globalPrediction, combinedPrediction;
    wire [1:0] localHistory, globalHistory;
    // Global History Shift Register
    reg [31:0] GHSR;
    always @(wPCindex or taken) begin
        GHSR <= {GHSR[30:0], taken};
    end
    LocalHistoryTable lht(
        .rPCindex(PC[3:0]),
        .wPCindex(wPCindex),
        .taken(taken),
        .lptIndex(lhtIndex),
        .wlptIndex(wlhtIndex)
    );
    // Prediction Tables
    PredictionTable lpt(
        .takenOut(localPrediction),
        .index(lhtIndex),
        .wIndex(wlhtIndex),
        .takenIn(taken)
    );
    PredictionTable gpt(
        .takenOut(globalPrediction),
        .index(GHSR[3:0]),
        .wIndex(GHSR[3:0]),
        .takenIn(taken)
    );
```

```
    PredictionTable cpt(
        .takenOut(combinedPrediction),
        .index({GHSR[3:0], lhtIndex}),
        .wIndex({GHSR[3:0], lhtIndex}),
        .takenIn(taken)
    );
    // Multiplexer for final prediction
    mux1bit mux(
        .out(prediction),
        .i0(localPrediction),
        .i1(globalPrediction),
        .sel(combinedPrediction)
    );
endmodule
```

Listing 5: Branch Predictor using multiple prediction strategies.

2.1.6 LocalHistoryTable (LHT)

The LocalHistoryTable maintains the local history of branch outcomes for prediction purposes.

```
module LocalHistoryTable(
    output reg [3:0] lptIndex,
    output reg [3:0] wlptIndex,
    input [3:0] rPCindex,
    input [3:0] wPCindex,
    input taken
);
    reg [3:0] LHT [0:15];
    integer i;
    initial begin
        for (i = 0; i < 16; i = i + 1) begin
            LHT[i] = 4'b0;
        end
    end
    always @(rPCindex) begin
        lptIndex = LHT[rPCindex];
    end
    always @(wPCindex or taken) begin
        LHT[wPCindex] = {LHT[wPCindex][2:0], taken};
    end
endmodule
```

Listing 6: Local History Table for managing branch outcomes.

2.1.7 Branch Prediction Workflow Explanation

1. **Branch Instruction Check:**

   - **Check BTB:** The Branch Instruction (BIA) is first checked in the Branch Target Buffer (BTB) to see if it matches any stored entries. If found, the BTB

provides the corresponding Branch Target Address (BTA) and indicates whether the branch instruction was previously encountered.

2. **Branch Prediction Determination:**

   - **Predict Outcome:** If the BTB indicates a hit, the Branch Predictor uses the address information (PCToPredict) to determine the branch prediction. This involves using local and global history information to predict whether the branch will be taken or not.

3. **Local History Table (LHT):**

   - **Maintain History:** The LocalHistoryTable (LHT) maintains local branch history, providing context for predictions based on recent branch behavior. PredictionTables use this context, alongside global history, to make predictions.

4. **2-Bit Predictor:**

   - **Refine Prediction:** The 2-Bit Predictor refines the prediction by evaluating the current state and updating it based on the actual outcome of the branch, allowing for more accurate predictions over time.

5. **Final Prediction Selection:**

   - **Select Prediction:** Finally, the mux1bit module selects the most appropriate prediction from multiple sources, such as local, global, or combined predictors, ensuring the final branch prediction is based on the most reliable information.

By integrating these modules, the branch prediction system enhances the CPU's ability to forecast branch behavior accurately, improving instruction throughput.

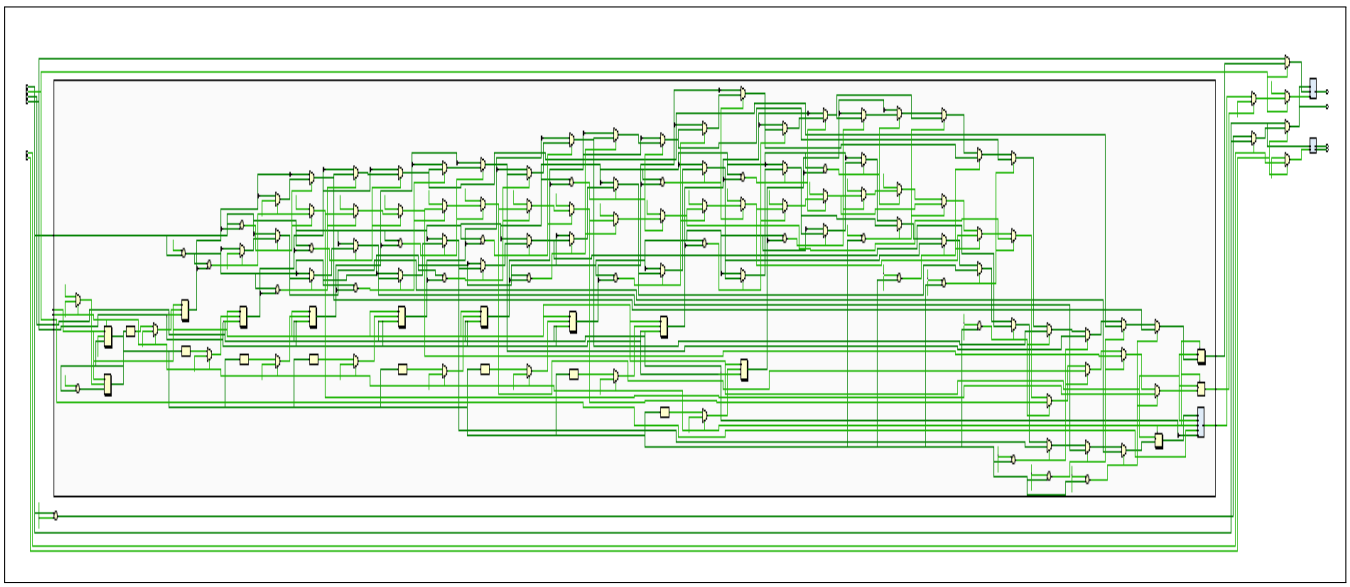

Figure 1: RTL Design of Branch Predictor Unit - BTB

2.2 Instruction Fetch Unit

The Instruction Fetch Unit (IFU) is responsible for fetching instructions from memory based on the current Program Counter (PC). It also handles branch prediction to fetch the correct instructions for branch targets. Below is the Verilog implementation of the Instruction Fetch Unit, including its components and workflow.

2.2.1 Instruction Memory

Description: The `InstructionMemory` module provides access to instructions stored in memory. It is byte-addressable and initialized from a file.

```
`include "mux.v"
`include "BranchPrediction.v"

module InstructionMemory(
    input [15:0] PC,
    input en,
    output reg [31:0] instr1,
    output reg [31:0] instr2
);
    reg [31:0] instructions [0:1023];

    always @(PC) begin
        if (en) begin
            instr1 <= instructions[PC];
            instr2 <= instructions[PC + 1];
        end
    end

    initial begin
        $readmemh("dataDep.dat", instructions);
    end
endmodule
```

Listing 7: InstructionMemory: Access to instruction memory.

2.2.2 Fetch Unit

Description: The `Fetch` module controls the fetching of instructions based on the Program Counter and branch target. It integrates the `InstructionMemory` and `BranchTargetBuffer` modules and uses a multiplexer to select the next PC based on branch prediction.

```
module Fetch(
    input PCSrc,
    input hlt,
    input PCWrite,
    input [15:0] PC,
    input [15:0] branchTarget,
    input [15:0] wBIA,
    output [31:0] instr1,
    output [31:0] instr2,
    output [15:0] NPC
```

```
);
    parameter PC_WIDTH = 16;

    reg [PC_WIDTH-1:0] nxtPC;
    reg en = 0;
    wire [PC_WIDTH-1:0] nextPC;
    wire prediction;
    wire [15:0] predictedTarget;
    reg [15:0] target;
    reg nextPC_sel;

    always @(*) begin
        if (hlt) begin
            en = 0;
            nxtPC = PC;
        end else if (PCWrite) begin
            nxtPC = PC + 2;
            en = 1;
        end else begin
            nxtPC = PC;
            en = 0;
        end
    end

    InstructionMemory im(
        .PC(PC),
        .en(en),
        .instr1(instr1),
        .instr2(instr2)
    );

    mux16bit mux(
        .out(NPC),
        .i0(nxtPC),
        .i1(target),
        .sel(nextPC_sel)
    );

    always @(branchTarget or predictedTarget or hit) begin
        if (PCSrc) begin
            target = branchTarget;
            nextPC_sel = PCSrc;
        end else begin
            target = predictedTarget;
            if (hit) begin
                nextPC_sel = prediction;
            end else begin
                nextPC_sel = 0;
            end
        end
    end
```

```
62
63      BranchTargetBuffer btb(
64          .predictedTarget(predictedTarget),
65          .hit(hit),
66          .prediction(prediction),
67          .PC(PC),
68          .wBIA(wBIA),
69          .branchTarget(branchTarget),
70          .PCSrc(PCSrc)
71      );
72  endmodule
```

Listing 8: Fetch: Fetching instructions based on PC and branch prediction.

2.2.3 Branch Target Buffer

Description: The `BranchTargetBuffer` module helps in predicting the target of branch instructions.

```
1  module BranchTargetBuffer(
2      output reg [15:0] predictedTarget,
3      output reg hit,
4      output reg prediction,
5      input [15:0] PC,
6      input [15:0] wBIA,
7      input [15:0] branchTarget,
8      input PCSrc
9  );
10
11 endmodule
```

Listing 9: BranchTargetBuffer: Predicting branch target addresses.

2.2.4 Instruction Fetch Workflow

1. Instruction Fetch

The Fetch module is responsible for managing the Program Counter (PC) and deciding the next instruction to be fetched. It evaluates the control signals to determine whether to halt instruction fetching or update the PC. If the `hlt` signal is asserted, it halts the fetch process, freezing the PC at its current value. When the processor is allowed to continue (`hlt` is not asserted) and `PCWrite` is enabled, the PC is incremented to the address of the next instruction (assuming a 2-byte instruction width). If `PCWrite` is not enabled, the PC does not change. The Fetch module computes the `nxtPC` value based on these conditions and sets an enable signal to the InstructionMemory module for fetching instructions.

2. Instruction Memory Access

The InstructionMemory module handles reading instructions from memory. It accesses the memory array using the address provided by the PC value. When enabled, it retrieves the instruction located at the address specified by the PC and also reads

the instruction at the subsequent address (PC + 1) to provide two consecutive instructions.

3. Branch Target Handling

The BranchTargetBuffer (BTB) predicts the target address for branch instructions to improve instruction fetching efficiency. It uses historical data to guess where a branch instruction will jump to, if taken. When a branch instruction is encountered, the BTB compares it against stored branch addresses and predicts a target address based on past behavior. If the prediction is successful (indicated by the `hit` signal), the predicted target address is used. Otherwise, the actual branch target address is used.

4. Multiplexer Selection

It takes the computed next PC address and the branch target address from the BTB and selects one based on the `nextPC_sel` signal. This signal is determined by branch prediction results and control signals. If branch prediction is successful (`PCSrc` is true and the prediction is accurate), the multiplexer selects the predicted branch target address. Otherwise, it uses the next sequential address.

5. Update and Fetch

Once the multiplexer selects the appropriate PC address (either the next sequential address or the branch target address), the PC is updated to this new value. This updated PC is then used to fetch the next set of instructions from the Instruction-Memory. By updating the PC and fetching instructions accordingly, the processor continues executing instructions seamlessly, even when handling branches or other control flow changes.

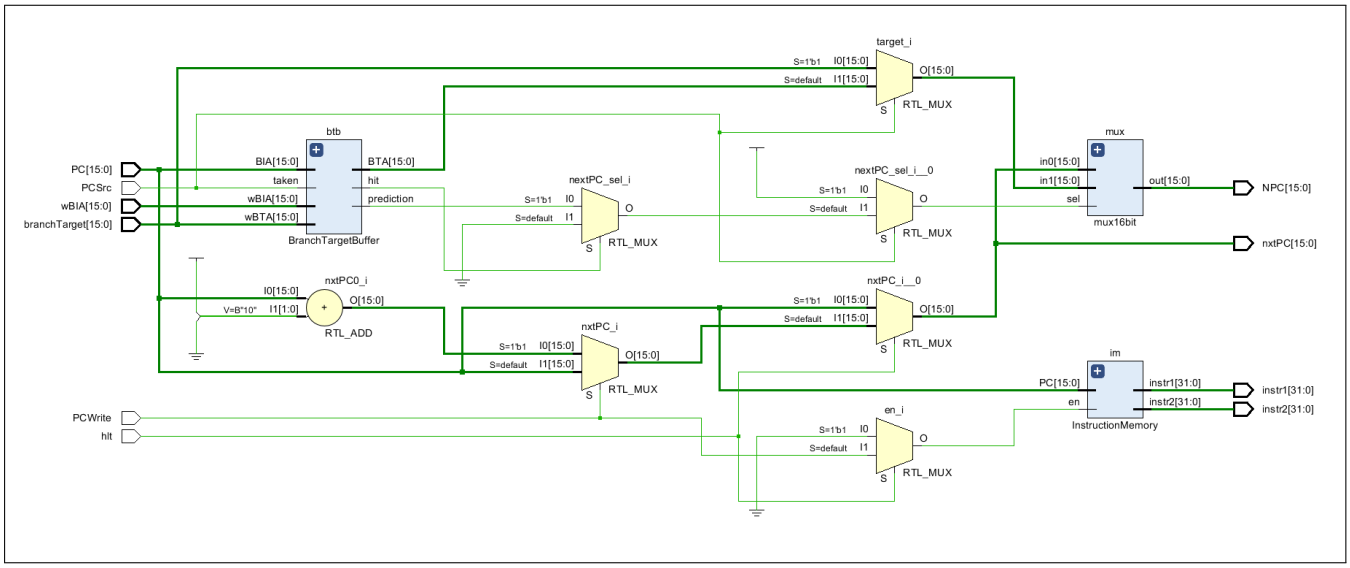

Figure 2: RTL Design of Fetch Unit

2.3 Pipeline Registers and Buffers

Pipeline registers and buffers play a vital role in pipelined processors, facilitating the transfer of data and control signals between stages. They ensure that each stage of the pipeline operates independently and efficiently. Below are the Verilog implementations for several pipeline registers and buffers used in a typical pipelined processor architecture.

2.3.1 Program Counter Register Description: Description:

The `PCReg` module stores the current Program Counter (PC) value. It updates its output when the `PCWrite` signal is asserted, allowing for the PC to advance or be set to a new value.

```
module PCReg(
    output reg [15:0] PCOut,
    input [15:0] PCIn,
    input clk, rst, PCWrite);
    always @(posedge clk or posedge rst) begin
        if (rst)
            PCOut <= 16'b0000000000000000;
        else if (PCWrite)
            PCOut <= PCIn;
    end
endmodule
```

Listing 10: PCReg: Program Counter Register.

2.3.2 Instruction Fetch/Decode Register Description: Description:

The `IFIDReg` module holds intermediate data between the Instruction Fetch and Decode stages. It stores the PC+4 value, fetched instructions, and the next PC selection signal.

```
module IFIDReg(
    output reg [15:0] PCplus4Out,
    output reg [31:0] instrOut1, instrOut2,
    output reg nextPC_selOut,
    input [15:0] PCplus4In,
    input [31:0] instrIn1, instrIn2,
    input clk, IFIDWrite, IFFlush, nextPC_selIn);
    always @(posedge clk) begin
        if (IFIDWrite) begin
            if (IFFlush) begin
                instrOut1 <= 32'b0;
                instrOut2 <= 32'b0;
            end else begin
                instrOut1 <= instrIn1;
                instrOut2 <= instrIn2;
            end
            PCplus4Out <= PCplus4In;
            nextPC_selOut <= nextPC_selIn;
        end
    end
endmodule
```

Listing 11: IFIDReg: Instruction Fetch/Decode Register.

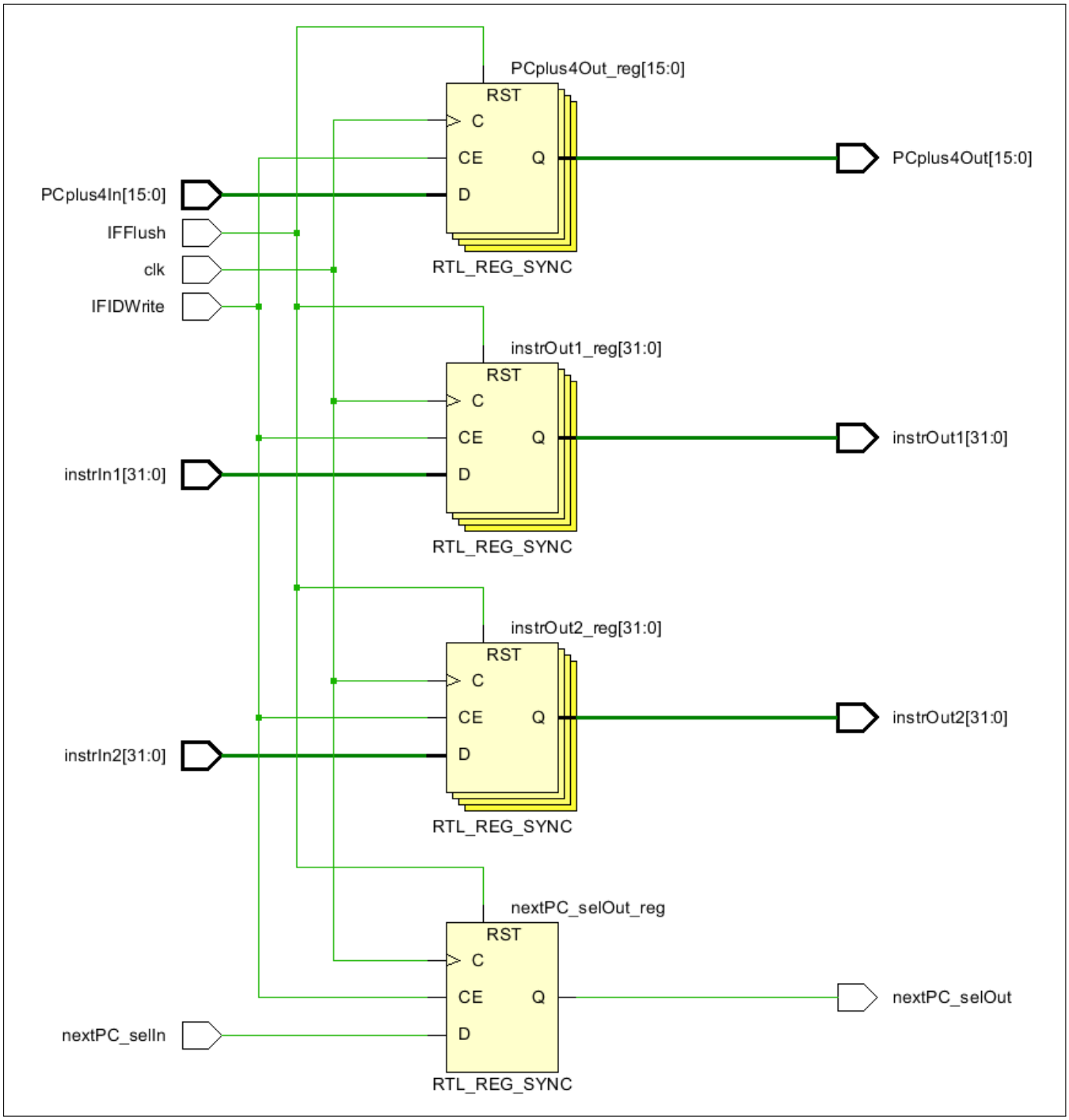

Figure 3: RTL Design of IFID

2.3.3 Dispatch Buffer Description: The `DispatchBuffer` module stores and forwards data between the decode stage and the execution stage. It handles register tags, data values, immediate values, control signals, and other necessary information.

```
module DispatchBuffer(
    output reg [3:0] rstag1out, rstag2out, rstag3out, rstag4out,
    output reg [15:0] dataRs1out, dataRt1out, dataRs2out, dataRt2out, imm1out
    output reg [5:0] ctrl1out, ctrl2out,
    output reg [3:0] robDest1out, robDest2out,
    output reg [2:0] func1out, func2out,
```

```
7       output reg spec1out, spec2out, nextPC_selOut,
8       input [3:0] rstag1in, rstag2in, rstag3in, rstag4in,
9       input [15:0] dataRs1in, dataRt1in, dataRs2in, dataRt2in, imm1in, imm2in,
10      input [5:0] ctrl1in, ctrl2in,
11      input [3:0] robDest1in, robDest2in,input [2:0] func1, func2,input clk, s
12      always @(posedge clk) begin
13          if (dispatchWrite) begin
14              if (flush) begin
15                  rstag1out <= 4'b0;
16                  rstag4out <= 4'b0;
17                  dataRs1out <= 16'b0;
18                  dataRt1out <= 16'b0;
19                  dataRs2out <= 16'b0;
20                  dataRt2out <= 16'b0;
21                  imm1out <= 16'b0;
22                  imm2out <= 16'b0;
23                  ctrl1out <= 6'b0;
24                  ctrl2out <= 6'b0;
25                  robDest1out <= 4'b0;
26                  robDest2out <= 4'b0;
27                  func1out <= 3'b0;
28                  func2out <= 3'b0;
29                  spec1out <= 1'b1;
30                  spec2out <= 1'b1;
31              end else begin
32                  rstag1out <= rstag1in;
33                  rstag2out <= rstag2in;
34                  rstag3out <= rstag3in;
35                  rstag4out <= rstag4in;
36                  dataRs1out <= dataRs1in;
37                  dataRt1out <= dataRt1in;
38                  dataRs2out <= dataRs2in;
39                  dataRt2out <= dataRt2in;
40                  imm1out <= imm1in;
41                  imm2out <= imm2in;
42                  ctrl1out <= ctrl1in;
43                  ctrl2out <= ctrl2in;
44                  robDest1out <= robDest1in;
45                  robDest2out <= robDest2in;
46                  func1out <= func1;
47                  func2out <= func2;
48                  spec1out <= spec1in;
49                  spec2out <= spec2in;
50                  PCplus2out <= PCplus2in;
51              end
52              nextPC_selOut <= nextPC_selIn;
53          end
54      end
55  endmodule
```

Listing 12: DispatchBuffer: Dispatch Buffer.

2.3.4 Execute Buffer Description **Description:** The `ExecuteBuffer` module holds data and control information related to the execution stage, including CDB data and entry tags.

```
module ExecuteBuffer(
    output reg [41:0] CDBData,
    output reg [1:0] entryTCout_I, entryTCout_LS,
    output reg [2:0] clearRSEntry,
    input [20:0] int_datain, MUL_datain, LS_datain,
    input [1:0] entryTCin_I, entryTC_LS,
    input clk
);
    always @(posedge clk) begin
        if (int_datain[20]) begin
            CDBData[20:0] = int_datain;
            clearRSEntry[0] = 1;
            entryTCout_I = entryTCin_I;
            if (LS_datain[20]) begin
                CDBData[41:21] = LS_datain;
                clearRSEntry[1] = 1;
                entryTCout_LS = entryTC_LS;
                clearRSEntry[2] = 0;
            end else if (MUL_datain[20]) begin
                CDBData[41:21] = MUL_datain;
                clearRSEntry[2] = 1;
                clearRSEntry[1] = 0;
            end
        end else if (LS_datain[20]) begin
            CDBData[20:0] = LS_datain;
            clearRSEntry[1] = 1;
            clearRSEntry[0] = 0;
            entryTCout_I = 2'bxx;
            entryTCout_LS = entryTC_LS;
            if (MUL_datain[20]) begin
                CDBData[41:21] = MUL_datain;
                clearRSEntry[2] = 1;
                clearRSEntry[1] = 0;
            end
        end else if (MUL_datain[20]) begin
            CDBData[20:0] = MUL_datain;
            clearRSEntry[2] = 1;
            clearRSEntry[0] = 0;
            clearRSEntry[1] = 0;
        end
    end
endmodule
```

Listing 13: ExecuteBuffer: Execute Buffer.

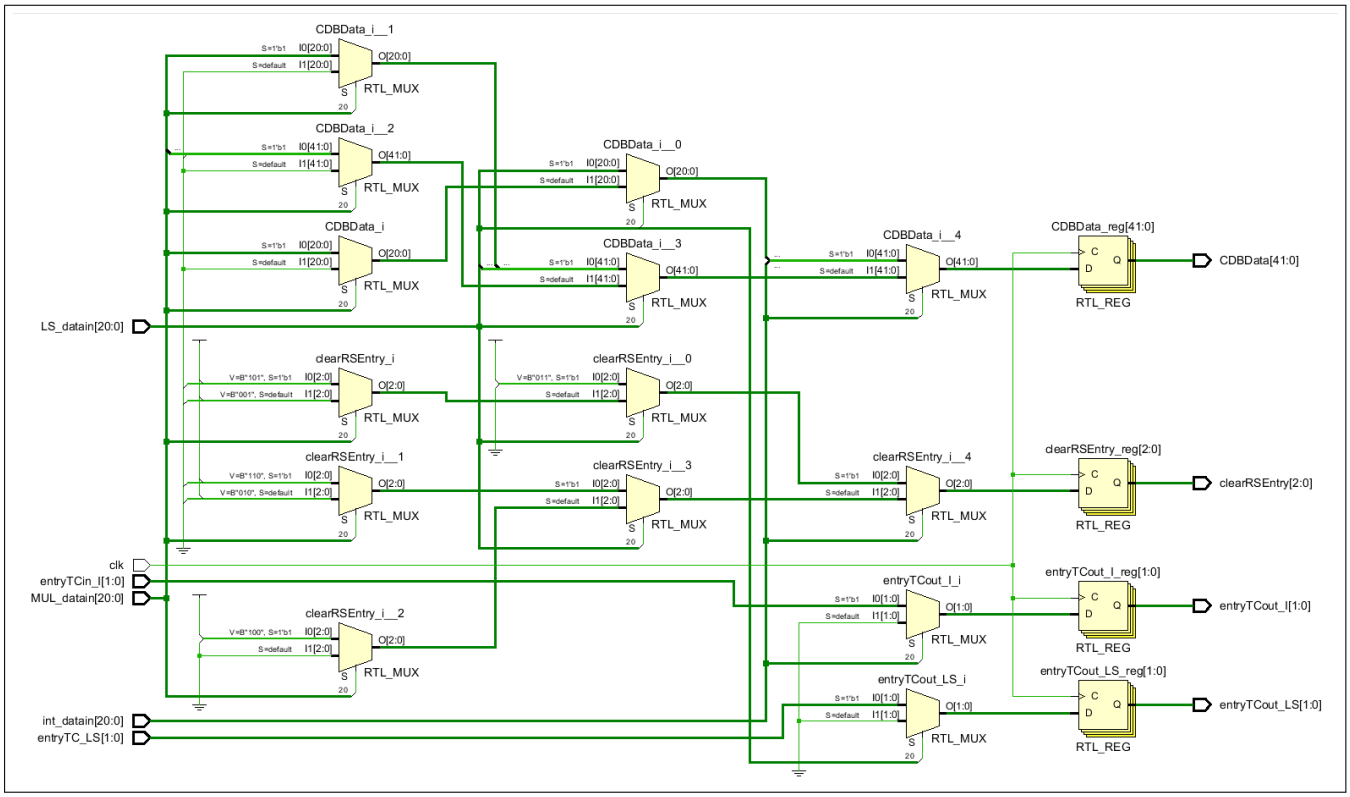

Figure 4: RTL Design of Execution Buffer

2.3.5 Data Flow Overview

- **Program Counter Update:** The `PCReg` module is responsible for maintaining and updating the Program Counter (PC). It updates the PC based on control signals such as `PCWrite` and `rst`. If the `rst` (reset) signal is active, it sets the PC to zero. Otherwise, it updates the PC with a new value if `PCWrite` is asserted. This updated PC value determines the address for fetching the next instruction.

- **Instruction Fetch:** The updated PC value from `PCReg` is used by the `InstructionMemory` module to fetch instructions from memory. The `InstructionMemory` module reads the instructions from a predefined memory file or array based on the current PC value. The fetched instructions are then passed on to the next stage of the pipeline.

- **Instruction Fetch to Decode:** The `IFIDReg` module acts as a pipeline register that holds the fetched instructions and the address of the next instruction (`PCplus4`). It transfers this information from the fetch stage to the decode stage. This is crucial for maintaining the continuity of instructions as they pass through the pipeline.

- **Instruction Decode:** During the decode stage, the `DispatchBuffer` processes the instruction data, including decoding register operands, immediate values, control signals, and other necessary information. It manages the information flow from the decode stage to the execution stage. It also handles scenarios where instructions may need to be flushed (cleared) due to control hazards or changes in execution flow.

- **Execution:** The `IDEXReg` module stores data and control signals as they move from the decode stage to the execution stage. It includes operands, immediate values, and

control signals required for the execution of arithmetic or logical operations. This data is used by the execution units to perform the actual computations.

- **Execute to Memory:** After the execution stage, results such as the output of arithmetic operations and any data to be written to memory are stored in the `EXMEMReg` module. This module holds the results and control signals necessary for the memory access stage, including addresses for memory operations and the destination register for results.
- **Memory to Write-Back:** In the memory stage, data that has been read from memory or computed by the ALU (Arithmetic Logic Unit) is stored in the `MEMWBReg` module. This data is then prepared for writing back to the register file. The `MEMWBReg` module holds data that will be used to update the registers and ensures that results from memory operations or computations are correctly passed to the write-back stage.
- **Write-Back:** Finally, the write-back stage updates the register file with the results stored in `MEMWBReg`. This includes writing data read from memory or the result of ALU computations into the appropriate registers. This step ensures that the results of operations are available for future instructions.

2.4 Control Unit

2.4.1 Verilog Code

```
module ControlUnit(
    input [5:0] opcode,
    input [5:0] functCode,
    output reg sw, lw, r, branch, jmp, hlt,
    output reg [2:0] func // 1xx - mul, 000 - ADD, 001 - SUB, 010 - AND, 011
);

    parameter R = 6'b000000,
    LW = 6'b100011,
    SW = 6'b101011,
    BEQ = 6'b000100,
    HLT = 6'b111111,
    JMP = 6'b000010;
    parameter ADD = 6'b100000,
    SUB = 6'b100010,
    AND = 6'b100100,
    OR = 6'b100101,
    SLT = 6'b101010,
    MUL = 6'b100001;

    initial begin
        hlt = 0;
    end

    always @(opcode or functCode) begin
        case (opcode)
            R: begin
                r = 1;
                sw = 0;
                lw = 0;
                branch = 0;
                jmp = 0;
                hlt = 0;
                if (functCode == ADD)
                    func = 3'b000;
                else if (functCode == SUB)
                    func = 3'b001;
                else if (functCode == AND)
                    func = 3'b010;
                else if (functCode == OR)
                    func = 3'b011;
                else if (functCode == MUL)
                    func = 3'b100;
            end
            LW: begin
                r = 0;
                sw = 0;
                lw = 1;
```

```
                branch = 0;
                jmp = 0;
                hlt = 0;
            end
            SW: begin
                r = 0;
                sw = 1;
                lw = 0;
                branch = 0;
                jmp = 0;
                hlt = 0;
            end
            BEQ: begin
                r = 0;
                sw = 0;
                lw = 0;
                branch = 1;
                jmp = 0;
                hlt = 0;
            end
            JMP: begin
                r = 0;
                sw = 0;
                lw = 0;
                branch = 0;
                jmp = 1;
                hlt = 0;
            end
            HLT: begin
                r = 0;
                sw = 0;
                lw = 0;
                branch = 0;
                jmp = 0;
                hlt = 1;
            end
        endcase
    end
endmodule
```

Listing 14: ControlUnit: Control Unit Module.

2.4.2 Explanation The `ControlUnit` module generates control signals for the processor based on the opcode and function code of the current instruction. Here is a detailed breakdown:

Inputs:

- `opcode` [5:0]: The 6-bit opcode field of the instruction that determines the instruction type.

- `functCode` [5:0]: The 6-bit function code field of the instruction, used to specify

operations for R-type instructions.

Outputs:

- `sw`: A signal indicating a store word (SW) instruction.
- `lw`: A signal indicating a load word (LW) instruction.
- `r`: A signal indicating a register (R-type) instruction.
- `branch`: A signal indicating a branch instruction (BEQ).
- `jmp`: A signal indicating a jump instruction (JMP).
- `hlt`: A signal indicating a halt instruction (HLT).
- `func` [2:0]: A 3-bit function code specifying the exact operation for R-type instructions (e.g., ADD, SUB).

Operation:

- `R-type Instructions:` When the opcode corresponds to an R-type instruction (`R`), the `r` signal is set, and other signals are cleared. The `func` output is set according to the `functCode` to specify the operation (e.g., ADD, SUB, OR).
- `LW Instruction:` For a load word instruction (`LW`), the `lw` signal is set, and all other control signals are cleared.
- `SW Instruction:` For a store word instruction (`SW`), the `sw` signal is set, and other signals are cleared.
- `Branch Instruction:` For a branch instruction (BEQ), the `branch` signal is set, and other signals are cleared.
- `Jump Instruction:` For a jump instruction (JMP), the `jmp` signal is set, and other signals are cleared.
- `Halt Instruction:` For a halt instruction (HLT), the `hlt` signal is set, and all other signals are cleared.

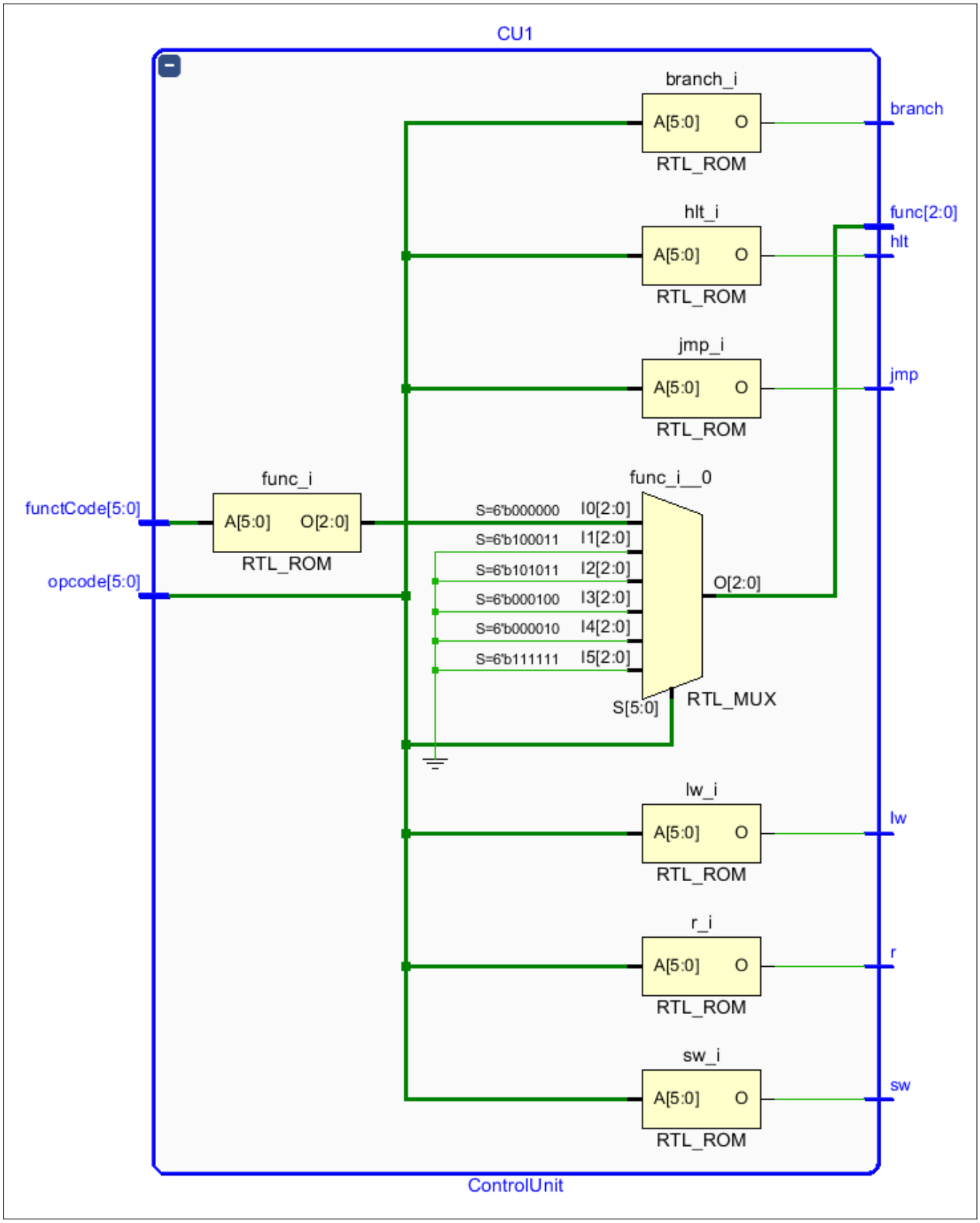

Figure 5: RTL Design of Control Unit

2.5 Instructions Decode Unit

2.5.1 Register File The `RegisterFile` module manages the storage and retrieval of data for the processor's registers. It handles read and write operations based on control signals and provides the necessary data for other modules. This module also supports a mechanism for reading register statuses and updating their values based on various control signals.

```
module RegisterFile(
    input regRead, clk, hlt, busy1, busy2,
    input [1:0] regWrite,
    input [4:0] readAddr1, readAddr2, readAddr3, readAddr4,
    writeAddr1, writeAddr2, destBT1, destBT2,input [3:0] tag1,
    tag2, input [15:0] writeData1,
    writeData2,output reg [20:0] readData1, readData2, readData3,
    readData4,output reg read
);

    parameter DATA_WIDTH = 16;
    parameter ARF_WIDTH = DATA_WIDTH + 5;

    initial read = 0;

    reg [ARF_WIDTH - 1:0] registers[0:31];

    always @(posedge clk) begin
        if (regRead) begin
            readData1 = registers[readAddr1];
            readData2 = registers[readAddr2];
            readData3 = registers[readAddr3];
            readData4 = registers[readAddr4];
            registers[destBT1][20:16] = {tag1, busy1};
            registers[destBT2][20:16] = {tag2, busy2};
            read = ~read;
        end
    end

    always @(posedge clk) begin
        if (regWrite[0])
            registers[writeAddr1][15:0] <= writeData1;
    end

    always @(posedge clk) begin
        if (regWrite[1])
            registers[writeAddr2][15:0] <= writeData2;
    end

    initial begin
        registers[0] = 21'b000000000000000000000;
        $readmemh("loadReg1.dat", registers);
    end
```

```
44
45      integer i;
46      initial begin
47          for (i=0; i<14; i=i+1)
48              $display($time, "␣␣reg␣%d␣=␣%d", i, registers[i][15:0]);
49          #200 for (i=1; i<12; i=i+1)
50              $display($time, "␣␣reg␣%d␣=␣%d", i, registers[i][15:0]);
51      end
52
53      always @(registers[7])
54          $display($time, "␣␣reg␣7␣=␣%d", registers[7][15:0]);
55
56      always @(registers[11])
57          $display($time, "␣␣reg␣11␣=␣%d", registers[11][15:0]);
58
59      always @(registers[8])
60          $display($time, "␣␣reg␣8␣=␣%d", registers[8][15:0]);
61
62      always @(registers[5])
63          $display($time, "␣␣reg␣5␣=␣%d", registers[5][15:0]);
64
65      always @(registers[2])
66          $display($time, "␣␣reg␣2␣=␣%d", registers[2][15:0]);
67  endmodule
```

Listing 15: RegisterFile: Register File Module.

2.5.2 Decode The `decode` module interprets the instruction fields to generate control signals and extract relevant data. It interfaces with the `ControlUnit` to produce control signals based on the opcode and function code. It also determines if speculative execution is required based on the control signals from both instructions.

```
1   module decode(
2       input [31:0] instr1, instr2,
3       output [4:0] rs1, rt1, rd1, rs2, rt2, rd2,
4       output [5:0] ctrl1, ctrl2,
5       output [2:0] func1, func2,
6       output [15:0] immediate1, immediate2,
7       output reg spec1, spec2
8   );
9
10      wire [5:0] opcode1, opcode2;
11      wire [15:0] imm1, imm2;
12
13      assign opcode1 = instr1[31:26];
14      assign imm1 = instr1[15:0];
15      assign rs1 = instr1[25:21];
16      assign rt1 = instr1[20:16];
17      assign rd1 = instr1[15:11];
18
19      assign opcode2 = instr2[31:26];
```

```
20      assign imm2 = instr2[15:0];
21      assign rs2 = instr2[25:21];
22      assign rt2 = instr2[20:16];
23      assign rd2 = instr2[15:11];
24
25      ControlUnit CU1(ctrl1[5],
26      ctrl1[4], ctrl1[3],
27      ctrl1[2], ctrl1[1],
28      ctrl1[0], func1,
29      opcode1, instr1[5:0]);
30      ControlUnit CU2(ctrl2[5],
31      ctrl2[4], ctrl2[3],
32      ctrl2[2], ctrl2[1],
33      ctrl2[0], func2,
34      opcode2, instr2[5:0]);
35      reg speculative;
36      initial begin
37          speculative = 0;
38          spec1 = 0;
39          spec2 = 0;
40      end
41      always @(ctrl1 or ctrl2) begin
42          if (speculative == 0) begin
43              if (ctrl1[2]) begin
44                  spec1 = 0;
45                  spec2 = 1;
46                  speculative = 1;
47              end else if (ctrl2[2]) begin
48                  speculative = 1;
49                  spec1 = 0;
50                  spec2 = 0;
51              end else
52                  speculative = 0;
53          end else begin
54              spec1 = 1;
55              spec2 = 1;
56          end
57      end
58  endmodule
```

Listing 16: decode: Decode Module.

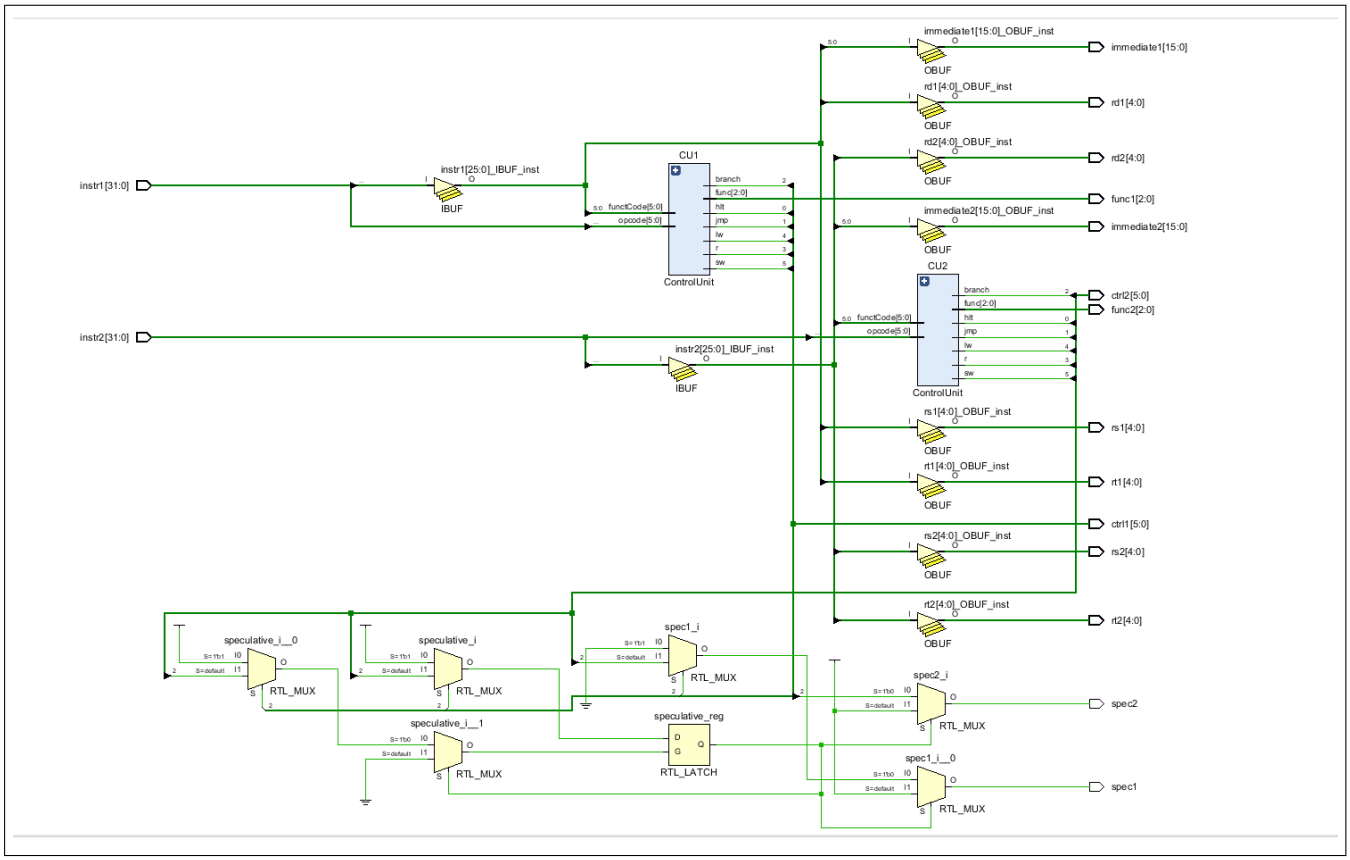

Figure 6: RTL Design of Decode Unit

2.5.3 Source Read The `SourceRead` module determines the source data for instructions based on whether the data is valid or needs to be retrieved from the reorder buffer.

```
module SourceRead(
    output reg [15:0] srcData,
    output reg [3:0] srcTag,
    input [16:0] regData, robData,
    input [3:0] robTag,
    input W);
    always @(W) begin
        #1 if (regData[16]) begin
            if (robData[16]) begin
                srcData = robData[15:0];
                srcTag = 4'b0;
            end else begin
                srcTag = robTag;
            end
        end else begin
            srcData <= regData[15:0];
            srcTag = 4'b0;
        end
    end
endmodule
```

Listing 17: SourceRead: Source Read Module.

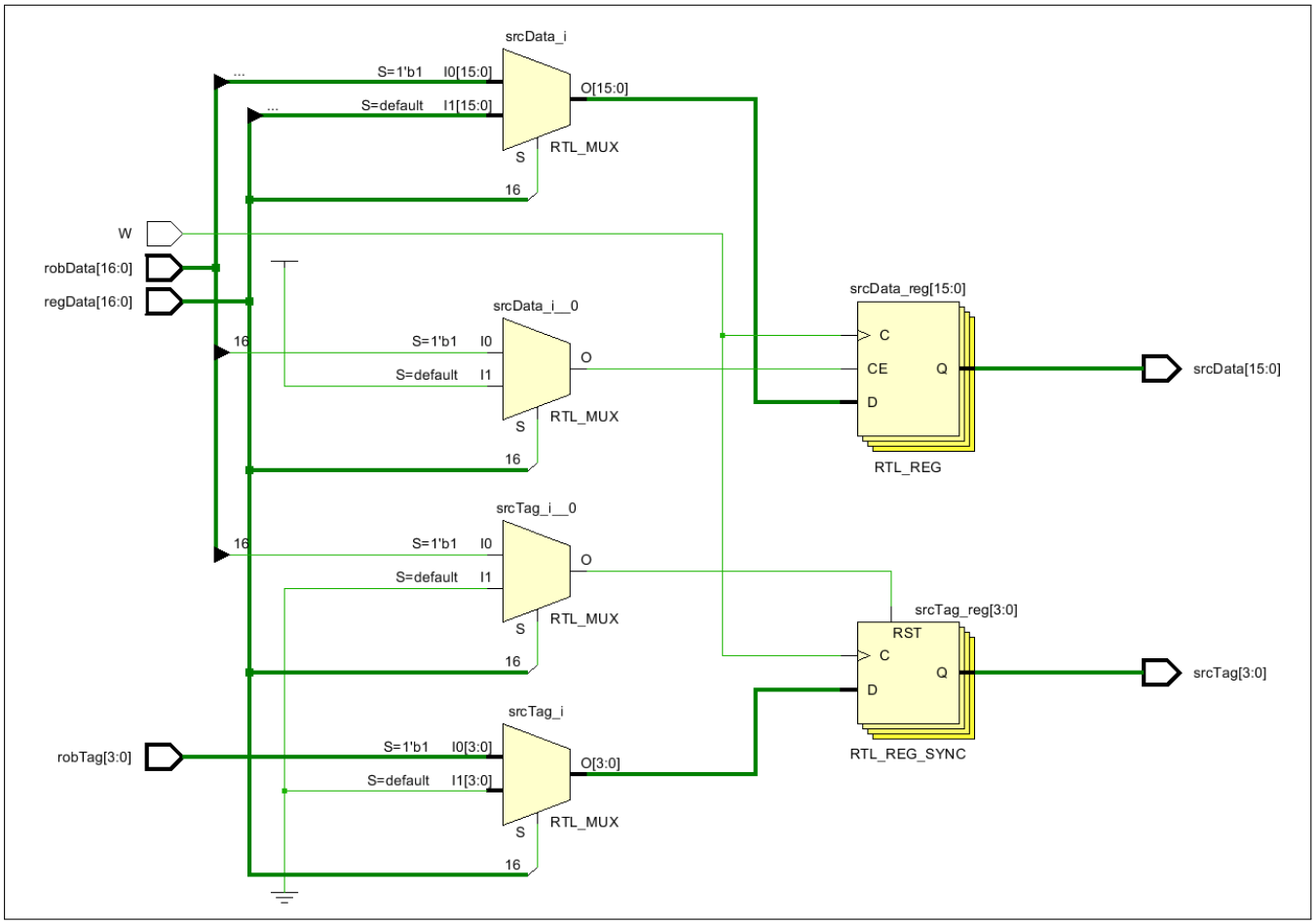

Figure 7: RTL Design of Source Read

2.5.4 Find Type The `FindType` module classifies instructions based on control signals. It determines the type of instruction (e.g., arithmetic, load/store, branch) to aid in directing the flow of execution and handling different types of instructions appropriately.

```
module FindType(
    output reg [1:0] type,
    input [5:0] ctrl
);

    always @(ctrl) begin
        if (ctrl[3] || ctrl[4])
            type = 2'b00;
        else if (ctrl[5])
            type = 2'b01;
        else if (ctrl[2])
            type = 2'b10;
        else if (ctrl[1])
            type = 2'b11;
    end
endmodule
```

Listing 18: FindType: Find Type Module.

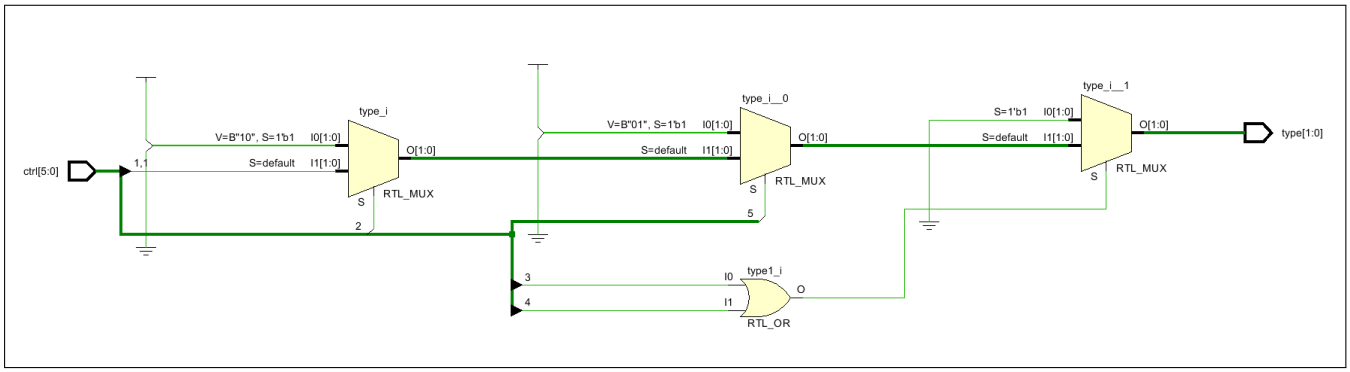

Figure 8: RTL Design of Control Unit

2.6 Instruction Execution Unit

The **Instruction Execution Unit** is responsible for executing the instructions after they have been decoded. This section details the components involved in executing instructions, including their respective Verilog implementations and functionalities.

2.6.1 Reservation Station for Integer Instructions

The `ReservationStation` module is used to hold instructions waiting for execution. It stores the instructions along with their operands and status until they are executed. This module manages the reservations for different functional units and ensures that instructions are executed in the correct order.

```
module ReservationStationInt(
    output reg [15:0] rsOut, rtOut,
    output reg [3:0] robTagOut,
    output reg [1:0] funcOut, entryToBeClearedOut,
    output reg stall, issued,
    input [15:0] rs1in, rt1in, rs2in, rt2in,
    input [3:0] tag_rs1in, tag_rt1in, tag_rs2in, tag_rt2in,
    input [1:0] func1in, func2in, load, entryToBeClearedIn,
    input [5:0] robDest1, robDest2,
    input clk, clearRSEntry, flush,
    input [41:0] CDBData);
    parameter SPEC_WIDTH = 4;
    reg [SPEC_WIDTH-1:0] spec1, spec2;
    reg [48:0] resEntries [0:3];
    wire [3:0] busyBits;
    wire [1:0] index1, index2;
    wire [1:0] full;
    reg [1:0] head;
    initial begin
        spec1 = {SPEC_WIDTH{1'b0}};
        spec2 = {SPEC_WIDTH{1'b0}};
        resEntries[0] = 49'b0;
        resEntries[1] = 49'b0;
        resEntries[2] = 49'b0;
        resEntries[3] = 49'b0;
        head = 2'b0;
        stall = 0;
    end
    assign busyBits = {resEntries[(head + 2'b11) % 4][47], resEntries[(head
    allocateUnit au(index1, index2, full, busyBits, clk);
    always @(posedge clk) begin
        if (load[0] && (full == 2'b00 || full == 2'b01) && (rt1in >= 0 && rs
            resEntries[index1] = {spec1, 1'b1, func1in, robDest1, tag_rt1in,
            resEntries[index1][0] = (tag_rs1in == 4'b0 && tag_rt1in == 4'b0)
        end
        if (load[1] && full == 2'b00) begin

            resEntries[index2] = {spec2, 1'b1, func2in, robDest2, tag_rt2in,
            head = (head + 1) % 4;
            resEntries[index2][0] = (tag_rs2in == 4'b0 && tag_rt2in == 4'b0)
```

```
41          end
42      end
43      always @(posedge clk) begin
44          if (flush) begin
45              resEntries[0] <= 49'b0;
46              resEntries[1] <= 49'b0;
47              resEntries[2] <= 49'b0;
48              resEntries[3] <= 49'b0;
49          end
50      end
51      always @(full) begin
52          stall = (full == 2'b11);
53      end
54      always @(posedge clk) begin
55          if (clearRSEntry) begin
56              resEntries[entryToBeClearedIn][47] <= 1'b0;
57              resEntries[entryToBeClearedIn][0] <= 1'b0;
58          end
59      end
60      always @(posedge clk) begin
61          casex ({resEntries[3][0], resEntries[2][0], resEntries[1][0], resEnt
62              4'bxxx1: begin
63                  rsOut <= resEntries[0][16:1];
64                  rtOut <= resEntries[0][36:21];
65                  robTagOut <= resEntries[0][44:41];
66                  funcOut <= resEntries[0][46:45];
67                  entryToBeClearedOut <= 2'b00;
68                  issued <= 1'b1;
69              end
70              4'bxx10: begin
71                  rsOut <= resEntries[1][16:1];
72                  rtOut <= resEntries[1][36:21];
73                  robTagOut <= resEntries[1][44:41];
74                  funcOut <= resEntries[1][46:45];
75                  entryToBeClearedOut <= 2'b01;
76                  issued <= 1'b1;
77              end
78              4'bx100: begin
79                  rsOut <= resEntries[2][16:1];
80                  rtOut <= resEntries[2][36:21];
81                  robTagOut <= resEntries[2][44:41];
82                  funcOut <= resEntries[2][46:45];
83                  entryToBeClearedOut <= 2'b10;
84                  issued <= 1'b1;
85              end
86              4'b1000: begin
87                  rsOut <= resEntries[3][16:1];
88                  rtOut <= resEntries[3][36:21];
89                  robTagOut <= resEntries[3][44:41];
90                  funcOut <= resEntries[3][46:45];
91                  entryToBeClearedOut <= 2'b11;
```

```
92                  issued <= 1'b1;
93              end
94              default: begin
95                  rsOut <= 16'b0;
96                  rtOut <= 16'b0;
97                  robTagOut <= 4'b0;
98                  issued <= 1'b0;
99              end
100         endcase
101     end
102     wire [15:0] oper01, oper02, oper11, oper12, oper21, oper22, oper31, oper
103     reg [1:0] tmIndex;
104     tagMatch tm1(oper01, resEntries[0][20:17], CDBData);
105     tagMatch tm2(oper02, resEntries[0][40:37], CDBData);
106     tagMatch tm3(oper11, resEntries[1][20:17], CDBData);
107     tagMatch tm4(oper12, resEntries[1][40:37], CDBData);
108     tagMatch tm5(oper21, resEntries[2][20:17], CDBData);
109     tagMatch tm6(oper22, resEntries[2][40:37], CDBData);
110     tagMatch tm7(oper31, resEntries[3][20:17], CDBData);
111     tagMatch tm8(oper32, resEntries[3][40:37], CDBData);
112     always @(oper01 or oper02 or oper11
113     or oper12 or oper21 or oper22 or oper31 or oper32) begin
114         case (tmIndex)
115             2'b00: begin
116                 resEntries[0][20:1] <= {4'b0, oper01};
117                 resEntries[0][40:21] <= {4'b0, oper02};
118             end
119             2'b01: begin
120                 resEntries[1][20:1] <= {4'b0, oper11};
121                 resEntries[1][40:21] <= {4'b0, oper12};
122             end
123             2'b10: begin
124                 resEntries[2][20:1] <= {4'b0, oper21};
125                 resEntries[2][40:21] <= {4'b0, oper22};
126             end
127             2'b11: begin
128                 resEntries[3][20:1] <= {4'b0, oper31};
129                 resEntries[3][40:21] <= {4'b0, oper32};
130             end
131         endcase
132     end
133     always @(resEntries[tmIndex]) begin
134         resEntries[tmIndex][0]
135         = (resEntries[tmIndex][20:17] == 4'b0 && resEntries[tmIndex]
136         [40:37] == 4'b0);
137     end
138 endmodule
```

Listing 19: Reservation Modules for Integer Instructions

Explanation: The `ReservationStation` module manages a reservation station for integer operations, handling the allocation, clearing, and issuing of entries based on var-

ious inputs and control signals.

2.6.2 Allocate Unit The `AllocateUnit` module is responsible for determining which entry in the reservation station will be used for a new instruction. It ensures that there is proper allocation of entries based on their availability and manages the corresponding indices.

```
module allocateUnit(
    input clk,
    input [3:0] in,
    output reg [1:0] index1, index2,
    output reg [1:0] full
);
    always @(negedge clk) begin
        // Code for allocating reservation station entries
    end
endmodule
```

Listing 20: AllocateUnit: Allocate Unit Module.

Explanation: The `AllocateUnit` module handles the assignment of available entries in the reservation station to new instructions. It updates the ‘index1‘ and ‘index2‘ outputs with the indices of the allocated entries and uses the ‘full‘ signal to indicate whether there is space available.

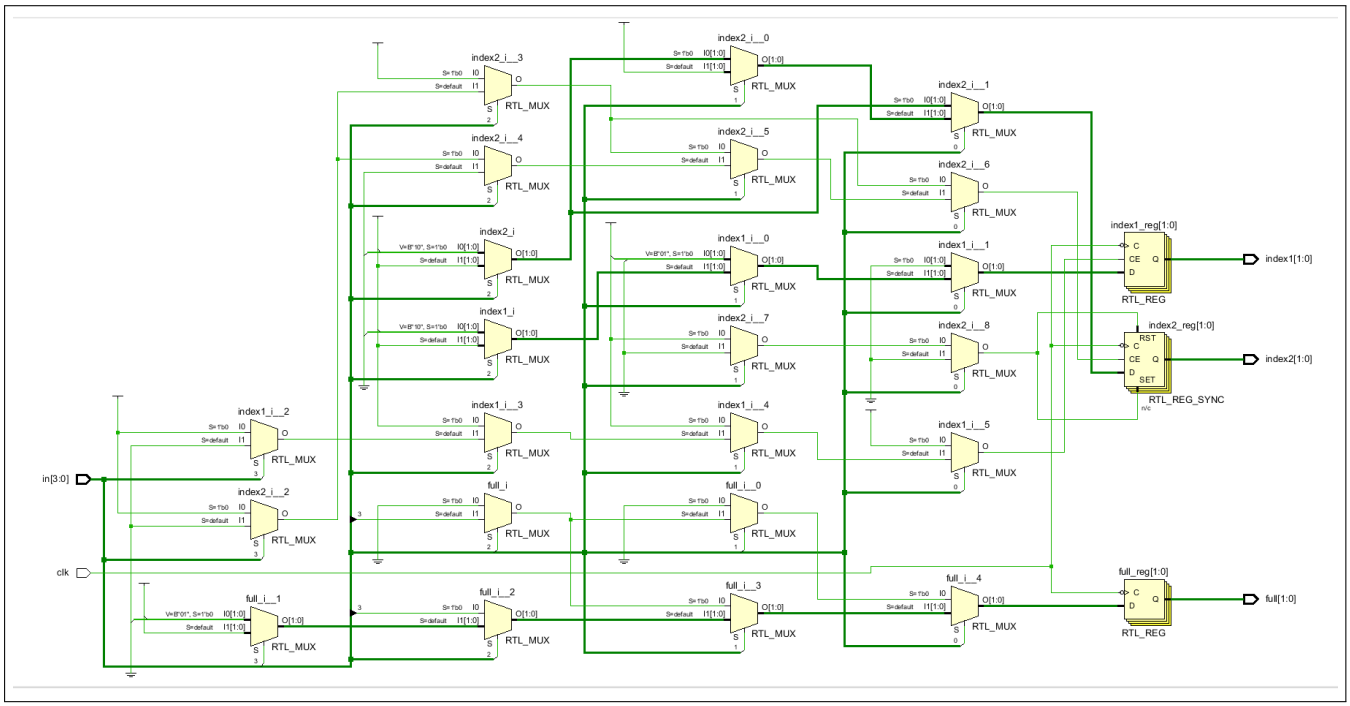

Figure 9: RTL Design of Allocate Unit

2.6.3 Tag match The `TagMatch` module compares instruction tags with those in the Common Data Bus (CDB) to determine if there is a match. It updates the operands based on the tag matches to ensure that instructions have the correct data for execution.

```
module tagMatch(
    output reg [15:0] operand,
    input [3:0] tag,
    input [41:0] CDBData
);
    always @(*) begin
        if (tag == CDBData[19:16])
            operand <= CDBData[15:0];
        else if (tag == CDBData[40:37])
            operand <= CDBData[36:21];
    end
endmodule
```

Listing 21: TagMatch: Tag Match Module.

Explanation: The `TagMatch` module is crucial for ensuring that operands used in instructions are up-to-date. It checks if the tag of an operand matches any tag in the CDB and updates the operand with the corresponding data if a match is found.

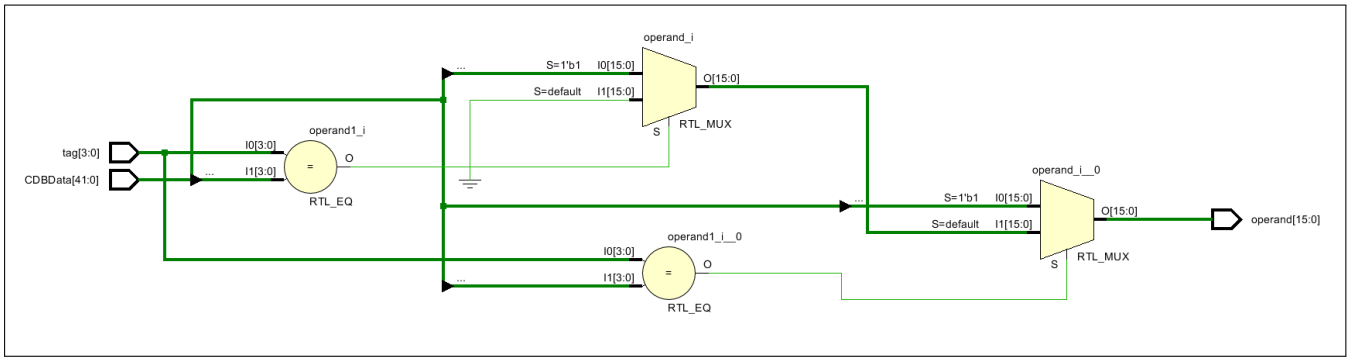

Figure 10: RTL Design of Tag Match

2.6.4 Store Data Buffer (SDB) The `StoreDataBuffer` (SDB) module temporarily holds data for store operations that have not yet been completed. It manages read and write operations and tracks the readiness of the data within the buffer.

```
module StoreDataBuffer(
    input [2:0] readIndex1, readIndex2,
    input [15:0] wAddr, wData,
    input store,
    output reg [31:0] readData1, readData2,
    output [2:0] wIndex,
    output reg [1:0] ready,
    output full
);
    reg [32:0] sdb [0:7];
    always @(store) begin
        // Code for handling store operations
    end
```

```
    always @(readIndex1 or readIndex2) begin
        // Code for reading data from SDB
    end
endmodule
```

Listing 22: : Store Data Buffer Module.

Explanation: The `StoreDataBuffer` module manages data that is involved in store operations. It handles the writing of data into the buffer and provides read operations to access the stored data. The 'ready' output indicates if the data is ready to be accessed, while the 'full' signal shows if the buffer is at capacity.

2.6.5 Reorder Buffer (ROB) The `ReOrderBuffer` (ROB) module maintains the order of instruction completion, ensuring that instructions are completed and written back in the correct sequence. It handles the final write-back stage and manages any branch corrections needed.

```
module ReOrderBuffer1(
    input clk, flush, correction,
    input [3:0] tag1, tag2, tag3, tag4,
    input [15:0] wData1, wData2,
    input [4:0] dest1, dest2,
    input [1:0] type1, type2,
    input [3:0] correctionIndex,
    output reg [27:0] data1, data2, data3, data4,
    output reg [3:0] index1, index2,
    output reg [1:0] wbType1, wbType2,
    output reg [20:0] wbData1, wbData2,
    output reg stall
);
    reg [27:0] robEntries[0:15];
    always @(posedge clk) begin
        // Code for handling ROB operations
    end
    always @(correctionIndex) begin
        // Code for handling branch corrections
    end
    always @(flush) begin
        // Code for flushing ROB entries
    end
endmodule
```

Listing 23: : Reorder Buffer Module.

Explanation: The `ReOrderBuffer` (ROB) ensures that instructions are executed and written back in the correct order. It handles write-backs and manages any corrections needed for mispredicted branches. The 'flush' signal clears the buffer entries as needed, while the 'correction' input deals with branch corrections. The 'stall' output indicates whether the ROB is busy and cannot accept new instructions.

Summary: Each component of the Instruction Execution Unit plays a crucial role in the accurate and efficient execution of instructions. The Reservation Station and Allocate Unit manage instruction scheduling and allocation, while the Tag Match, Store Data Buffer, and Reorder Buffer ensure correct data handling and final result processing.

2.7 Functional Units

2.7.1 Integer Functional Unit The `IntegerFU` module performs basic arithmetic and logical operations on 16-bit inputs. It takes two operands (`a` and `b`), a control signal (`ctrl`) to select the operation, and a destination tag (`destTag`).

```
module IntegerFU(CDBout, a, b, ctrl, destTag, issued);
    input [15:0] a, b;
    input [1:0] ctrl;
    input [3:0] destTag;
    input issued;
    output [20:0] CDBout;
    reg [15:0] out;
    assign CDBout = {issued, destTag, out};
    always @(*)
    begin
        case(ctrl)
            2'b00: out = a + b;
            2'b01: out = a - b;
            2'b10: out = a & b;
            2'b11: out = a | b;
        endcase
    end
endmodule
```

Listing 24: IntegerFU: Integer Functional Unit.

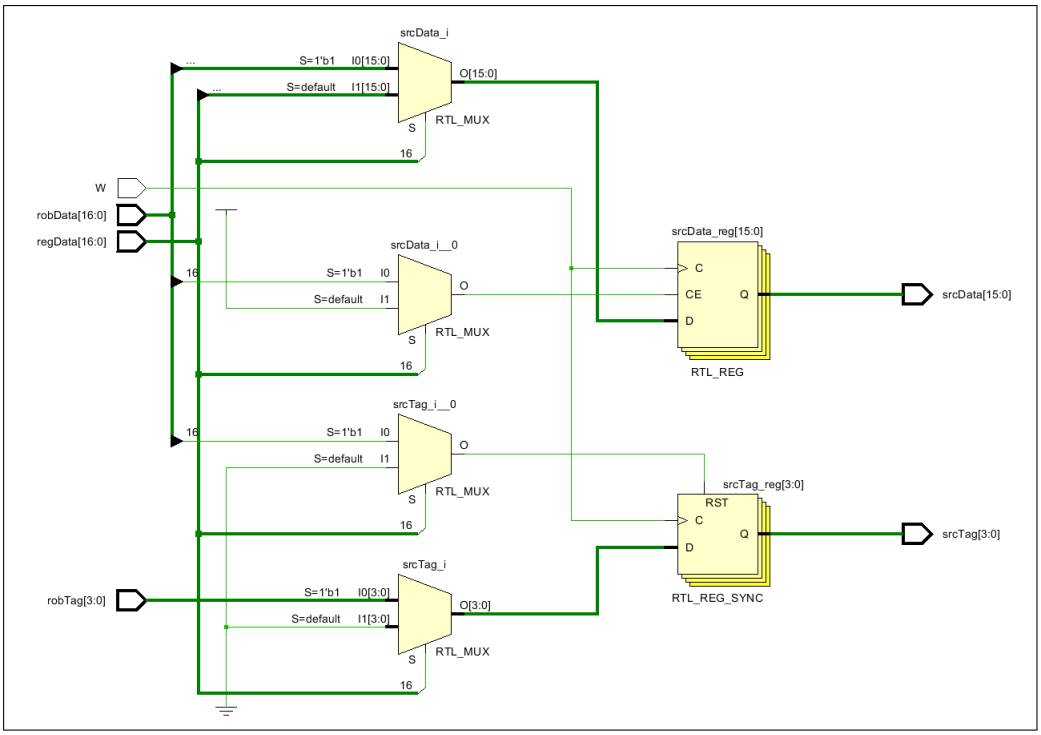

Figure 11: RTL Design of Integer Functional Unit

2.7.2 Multiply Unit The `multiply` module handles the multiplication of two 8-bit numbers. It computes the product and outputs the result along with the issued status and destination tag in a 21-bit format. The multiplication operation is delayed by 40 time units.

```
module multiply(CDBout, a, b, destTag, issued);
    input [7:0] a, b;
    input issued;
    input [3:0] destTag;
    output [20:0] CDBout;
    wire [15:0] tmp;
    assign tmp = a * b;
    assign #40 CDBout = {issued, destTag, tmp};
endmodule
```

Listing 25: Multiply: Multiply Unit.

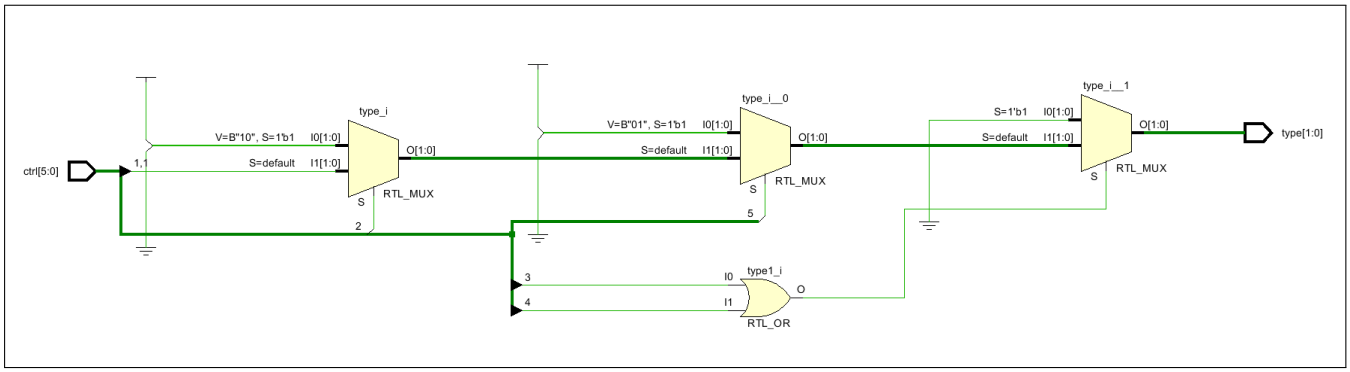

Figure 12: RTL Design of Tag Match

2.7.3 Load/Store Unit The `LoadStore` module is used for address calculations in load and store operations. It performs addition between two 16-bit inputs (`a` and `b`) to compute the effective address for memory operations.

```
module LoadStore(out, a, b);
    input [15:0] a, b;
    output [15:0] out;
    assign out = a + b;
endmodule
```

Listing 26: LoadStore: Load/Store Unit.

2.7.4 Branch Unit The `BranchUnit` module evaluates branch conditions and calculates the branch target address. It compares two 16-bit inputs (`a` and `b`) and, if they are equal and the branch is issued, it computes the new program counter address by adding the immediate value (`imm`) to the current program counter (`PC`).

```
module BranchUnit(PCSrc, branchTarget, a, b, PC, imm, issued);
    input [15:0] a, b, PC, imm;
    input issued;
    output PCSrc;
```

```
    output [15:0] branchTarget;
    assign PCSrc = (issued && a == b) ? 1'b1 : 1'b0;
    assign branchTarget = PCSrc ? PC + imm : PC;
endmodule
```

Listing 27: BranchUnit: Branch Unit.

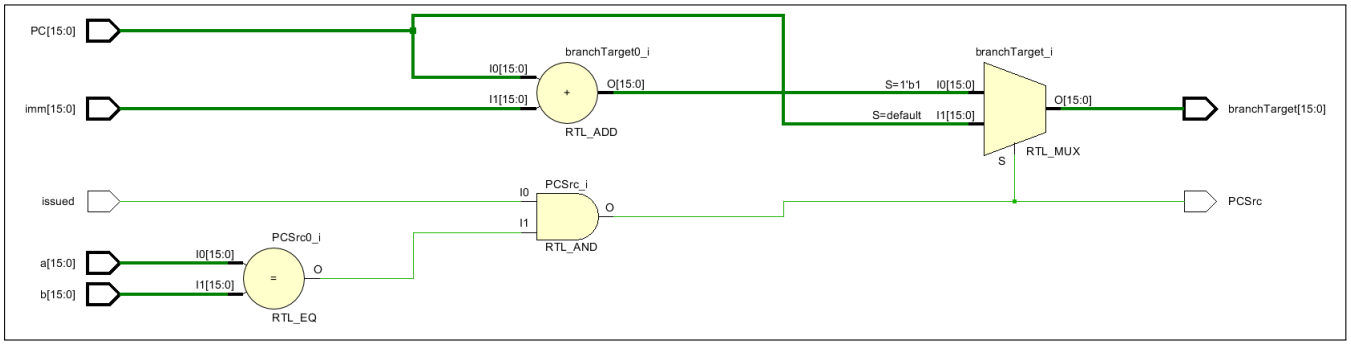

Figure 13: RTL Design of Branch Unit

2.7.5 5-bit Multiplexer The `mux5bit` module is a 5-bit multiplexer that selects between two 5-bit inputs based on a single-bit select signal. It outputs one of the inputs depending on the value of the select signal.

```
module mux5bit(output[4:0] out, input[4:0] i0, input[4:0] i1, input sel);
    assign out = sel ? i1 : i0;
endmodule
```

Listing 28: : 5-bit Multiplexer.

3 Superscalar Processor Design

In this section, we will cover the data flow of a superscalar processor and provide the corresponding Verilog code for its implementation.

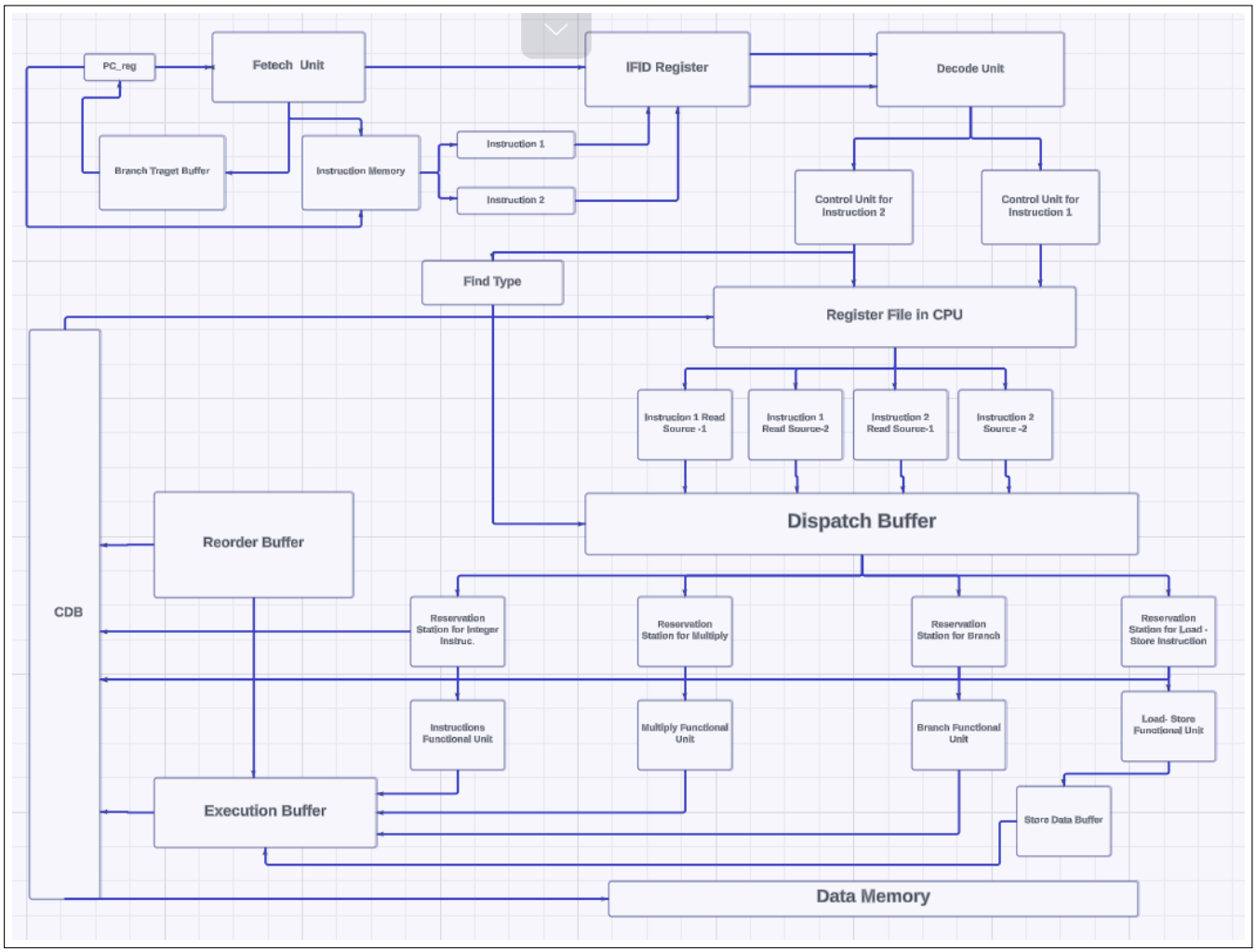

Figure 14: Flow of Data in Superscalar Processor

3.1 Instruction Fetch (IF)

Purpose: Retrieve two instructions per cycle from memory and update the Program Counter (PC).

Components Involved:

- Program Counter (PC)
- Fetch Unit
- Instruction Memory

Flow:

1. The PC provides the address of the current instruction.
2. The Fetch unit retrieves two instructions (instr1 and instr2) from memory based on the PC.

3. These instructions are passed to the IFID register, which holds them until they are needed in the Decode stage.

4. The PC is updated to point to the next set of instructions.

3.2 Instruction Fetch Workflow

Purpose: Decode two instructions to determine their operation types and operands.

Components Involved:

- Decode Unit
- Register File

Flow:

1. The Decode unit processes both instructions simultaneously.
2. It extracts the operation codes, source and destination registers, and immediate values.
3. The Register File is accessed to read the values of the source registers for both instructions.
4. Decoded information and control signals are forwarded to the Dispatch Buffer for both instructions.

3.3 Instruction Dispatch (DIS)

Purpose: Dispatch two instructions to the appropriate reservation stations.

Components Involved:

- Dispatch Buffer
- Reservation Stations

Flow:

1. The Dispatch Buffer holds the decoded instructions and associated data.
2. Each instruction is dispatched to its corresponding reservation station based on the instruction type:
    - Reservation Station for Integer Instructions
    - Reservation Station for Multiplications
    - Reservation Station for Load/Store Operations
    - Reservation Station for Branch Instructions

3.4 Instruction Execution (EX)

Purpose: Execute two instructions in parallel using functional units.

Components Involved:

- Functional Units (e.g., Integer ALU, Multiplier, Load/Store Unit, Branch Unit)
- Reservation Stations

Flow:

1. Reservation stations provide operands to functional units when they are available.
2. Functional units perform the operations for both instructions in parallel.
3. The results are placed in the Execution Buffer for temporary holding before writing back.

3.5 Execution Buffer

Purpose: Buffer the results of executed instructions before they are written back to the register file or memory.

Components Involved:

- Execution Buffer

Flow:

1. The Execution Buffer temporarily holds results from the functional units.
2. Results are tagged with their respective reservation station or instruction identifiers.
3. This ensures that results are properly managed and prepared for broadcasting on the CDB.

3.6 Common Data Bus (CDB)

Purpose: Broadcast results of two executed instructions to all relevant components.

Components Involved:

- CDB
- Reservation Stations
- Reorder Buffer (ROB)

Flow:

1. The CDB broadcasts results from the Execution Buffer.
2. Reservation stations receive these results and update their entries if the result matches their pending instructions.
3. The ROB also updates its entries with the results received on the CDB.

3.7 Store Data Buffer (SDB)

Purpose: Manage data being written to memory, especially for store operations.

Components Involved:

- Store Data Buffer
- Data Memory

Flow:

1. Store instructions place their data into the Store Data Buffer.
2. The buffer holds data until it is written to memory.
3. When ready, the buffer transfers data to the Data Memory.

3.8 Write Back (WB)

Purpose: Commit results to the register file or memory.

Components Involved:

- Register File
- Memory

Flow:

1. The ROB determines when results are ready to be committed.
2. Results are written to the Register File if they are register results.
3. For store operations, the Store Data Buffer handles writing data to memory.

3.9 Commit (COM)

Purpose: Finalize the instruction results and ensure that all instructions are correctly retired from the pipeline.

Components Involved:

- Reorder Buffer (ROB)
- Register File

Flow:

1. The ROB tracks the status and results of instructions in program order.
2. When an instruction's result is confirmed to be correct and no exceptions occurred, it is committed.
3. The results are then written to the Register File or Memory, depending on the instruction type.
4. The ROB updates its entries to indicate that the instruction has been committed, allowing subsequent instructions to proceed.

3.9.1 Verilog Code to implement Superscalar Processor

This `module` module calls all the small modules in the top module.

```
module SuperScalarProcessor(input clk, input rst);
        parameter PC_WIDTH = 16, DATA_WIDTH = 16;
        parameter INSTR_WIDTH = 32;
        parameter MEM_SIZE = 1024; //2^10

        //wire PCSrcTmp;
        reg IFFlush, flush;
        wire PCSrc,
 nextPC_sel_f, nextPC_sel_IFID, nextPC_sel_dis, nextPC_sel_RSB;
 wire[PC_WIDTH-1:0] NPC,
 PCplus2, PCplus2Out, PCOut, PCplus2Out_dis, PCOut_RSB,branchAddress;
        reg [PC_WIDTH-1:0] PC, branchTarget;
        wire[INSTR_WIDTH-1:0] instr1, instr1Out, instr2, instr2Out;
        always@(rst or NPC)
        begin
                if(rst)
                begin
                        PC = 16'b0000000000000000;
                        $display($time, "PC = %d", PC);
                end
                else
                        PC = NPC;
        end
        initial
        begin
                //PCSrc = 0;
                flush = 0;
                IFFlush = 0;
                PCWrite = 1;
                IFIDWrite = 1;
                dispatchWrite = 1;
        end
        reg PCWrite, IFIDWrite, dispatchWrite;
        PCReg reg0(PCOut, PC, clk, rst, PCWrite);
        Fetch f(nextPC_sel_f, NPC, PCplus2, instr1, instr2, (PCSrc || ctrl1[
        IFIDReg reg1(PCplus2Out, instr1Out, instr2Out, nextPC_sel_IFID, PCpl
        wire [20 : 0] readData1, readData2, readData3, readData4;
        wire [15 : 0] dataRs1, dataRt1, dataRs2, dataRt2, imm1, imm2, imm1ou
        wire [3:0] rstag1, rstag2,
        wire [4:0] rs1, rt1, rs2, rt2,
        always@(posedge clk)
                #1 W = 1;
        always@(negedge clk)
                #1 W = 0;
        decode dec(rs1, rt1, rd1, rs2, rt2, rd2, imm1, imm2, ctrl1, ctrl2, f
        RegisterFile rf(read, readData1, readData2, readData3, readData4, 1'
        SourceRead srcRead1(dataRs1, rstag1, readData1[16:0], robdata1[23:7]
        SourceRead srcRead2(dataRt1, rstag2, readData2[16:0], robdata2[23:7]
        SourceRead srcRead3(dataRs2, rstag3, readData3[16:0], robdata3[23:7]
```

```
50          SourceRead srcRead4(dataRt2, rstag4, readData4[16:0], robdata4[23:7]
51          mux5bit mux1(destBT1, rt1, rd1, ctrl1[3]);
52          mux5bit mux2(destBT2, rt2, rd2, ctrl2[3]);
53          FindType ft1(type1, ctrl1);
54          FindType ft2(type2, ctrl2);
55          always@(ctrl1[1] or ctrl2[1])
56          begin
57                  if(ctrl1[1])
58                          branchTarget = imm1;
59                  else if(ctrl2[1])
60                          branchTarget = imm2;
61                  else
62                          branchTarget = branchAddress;
63          end
64          wire [15 : 0] dataRs1out, dataRt1out, dataRs2out, dataRt2out;
65          wire [3:0] rstag1out, rstag2out, rstag3out, rstag4out;
66          wire [5:0] ctrl1out, ctrl2out;
67          reg [1:0] load_I, load_LS, load_M, load_B;
68          wire stall, stall_I, stall_LS, stall_B, stall_M, stall_ROB, LorSout_
69          wire [3:0] robDest1out, robDest2out;
70          DispatchBuffer dispatch_buf(rstag1out, rstag2out, rstag3out, rstag4ou
71   robDest1out, robDest2out, func1out, func2out, spec1out, spec2out, PCplus2Ou
72  imm1, imm2, ctrl1, ctrl2, robIndex1, robIndex2, func1, func2, spec1, spec2,
73          always@(ctrl1out or func1out)
74          begin
75                  if(ctrl1out[5] == 1 || ctrl1out[4] == 1)
76                  begin
77                          load_LS[0] = 1;
78                          LorS1 = ctrl1out[4];
79                          load_I[0] = 0;
80                          load_M[0] = 0;
81                          load_B[0] = 0;
82                  end
83                  else if(ctrl1out[3]) // r type instr
84                  begin
85                          load_LS[0] = 0;
86                          load_I[0] = ~func1out[2];
87                          load_M[0] = func1out[2];
88                          load_B[0] = 0;
89                  end
90                  else if(ctrl1out[2])
91                  begin
92                          load_LS[0] = 0;
93                          load_I[0] = 0;
94                          load_M[0] = 0;
95                          load_B[0] = 1;
96                  end
97                  else if(ctrl1out[0])
98                  begin
99                          load_LS= 2'b0;
100                         load_I = 2'b0;
```

```
                        load_M = 2'b0;
                        load_B = 2'b0;
                end
                else
                begin
                        load_LS[0] = 0;
                        load_I[0] = 0;
                        load_M[0] = 0;
                        load_B[0] = 0;
                end
        end
        always@(ctrl2out or func2out)
        begin
                if(ctrl2out[5] == 1 || ctrl2out[4] == 1)
                begin
                        load_LS[1] = 1;
                        LorS2 = ctrl2out[4];
                        load_I[1] = 0;
                        load_M[1] = 0;
                        load_B[1] = 0;
                end
                else if(ctrl2out[3] == 1)
                begin
                        load_LS[1] = 0;
                        load_I[1] = ~func2out[2];
                        load_M[1] = func2out[2];
                        load_B[1] = 0;
                end
                else if(ctrl2out[2])
                begin
                        load_LS[1] = 0;
                        load_I[1] = 0;
                        load_M[1] = 0;
                        load_B[1] = 1;
                end
                else
                begin
                        load_LS[1] = 0;
                        load_I[1] = 0;
                        load_M[1] = 0;
                        load_B[1] = 0;
                end
        end
        wire [15:0] rsout_I, rtout_I, rsout_M, rtout_M, rsout_LS, rtout_LS,
        wire [3:0] robTagOut_I, robTagOut_M, robTagOut_LS, robTagOut_B;
        wire [41:0] CDBData;
        reg CDBBusy, LorS1, LorS2;
        wire issued_I, issued_LS, issued_M, issued_B;
        ReservationStationInt RS_Int(stall_I, issued_I, entryTCout_I, robTag
        ReservationStation RS_mul(stall_M, issued_M, robTagOut_M, rsout_M, r
        ReservationStationLS RS_ls(stall_LS, issued_LS, entryTCout_LS, robTa
```

```
152         ReservationStation_Branch RS_B(stall_B, issued_B, robTagOut_B, rsout
153         wire [20:0] int_out, mul_out;
154         wire [1:0] aluOp, entryTCin_I, entryTCout_I, entryTCin_LS, entryTCou
155         wire [15:0] LSData, LS_out, loadData;
156         wire [2:0] clearRSEntry;
157         //Functional Units
158         IntegerFU IFU(int_out, rsout_I, rtout_I, aluOp, robTagOut_I, issued_
159         multiply mul(mul_out, rsout_M[7:0], rtout_M[7:0], robTagOut_M, issue
160         LoadStore LS(LS_out, rsout_LS, immOut_LS);
161         BranchUnit BU(PCSrc, branchAddress, rsout_B, rtout_B, PCOut_RSB, imm
162         always@(issued_B or PCSrc)
163         begin
164                 if(issued_B && PCSrc)
165                         flush = 1;
166                 else
167                         flush = 0;
168         end
169         StoreDataBuffer sdb(sdbFull, sdReady, strData1, strData2, strdIndex,
170         DataMemory dm(loadData, LS_out, LorSout_LS, memWrite, strData1[15:0]
171     assign LSData = LorSout_LS ? loadData : {13'b0, strdIndex};
172         ExecuteBuffer EX_buf(CDBData, clearRSEntry, entryTCin_I, entryTCin_LS
173         wire [3:0] robtag1, robtag2, robtag3, robtag4, robIndex1, robIndex2;
174         wire [27:0] robdata1, robdata2, robdata3, robdata4;
175         wire [1:0] type1, type2;
176         wire [20:0] wbData1, wbData2;
177         wire [1:0] wbType1, wbType2;
178         wire [3:0] wIndex1, wIndex2;
179         wire [15:0] wData1, wData2;
180         wire correction;
181         assign correction = ~(PCSrc^nextPC_sel_RSB);
182         assign robtag1 = readData1[20:17];
183         assign robtag2 = readData2[20:17];
184         assign robtag3 = readData3[20:17];
185         assign robtag4 = readData4[20:17];
186         ReOrderBuffer1 rob(stall_ROB, wbData1, wbData2, wbType1, wbType2, ro
187         reg [2:0] strIndex1, strIndex2;
188         reg [4:0] destReg1, destReg2;
189         wire [31:0] strData1, strData2;
190         wire [2:0] strdIndex;
191         wire [1:0] sdReady;
192         wire sdbFull;
193         always@(wbType1 or wbData1)
194         begin
195                 if(wbType1 == 2'b00)
196                 begin
197                         destReg1 = wbData1[4:0];
198                         regWData1 = wbData1[20:5];
199                         regWrite[0] = 1'b1;
200                         memWrite[0] = 1'b0;
201                 end
202                 else if(wbType1 == 2'b01)
```

```
203                 begin
204                         regWrite[0] = 1'b0;
205                         memWrite[0] = 1'b1;
206                         strIndex1 = wbData1[7:5];
207                 end
208                 else
209                 begin
210                         regWrite[0] = 1'b0;
211                         memWrite[0] = 1'b0;
212                 end
213         end
214         always@(wbType2 or wbData2)
215         begin
216                 if(wbType2 == 2'b00)
217                 begin
218                         destReg2 = wbData2[4:0];
219                         regWData2 = wbData2[20:5];
220                         regWrite[1] = 1'b1;
221                         memWrite[1] = 1'b0;
222                 end
223                 else if(wbType2 == 2'b01)
224                 begin
225                         regWrite[1] = 1'b0;
226                         memWrite[1] = 1'b1;
227                         strIndex2 = wbData2[7:5];
228                 end
229                 else
230                 begin
231                         regWrite[1] = 1'b0;
232                         memWrite[1] = 1'b0;
233                 end
234         end
235         always@(stall_I or stall_LS or stall_B or stall_M or stall_ROB)
236         begin
237                 if(stall_I || stall_LS || stall_B || stall_M || stall_ROB)
238                 begin
239                         IFIDWrite = 0;
240                         dispatchWrite = 0;
241                         PCWrite = 0;
242                 end
243                 else
244                 begin
245                         IFIDWrite = 1;
246                         dispatchWrite = 1;
247                         PCWrite = 1;
248                 end
249         end
250         always@(CDBData)
251                 $display($time, "␣Super␣:␣CDBData␣:␣data1␣=␣%d,␣tag1␣=␣%d,␣d
252 endmodule
```

Listing 29: : Superscalar Module.

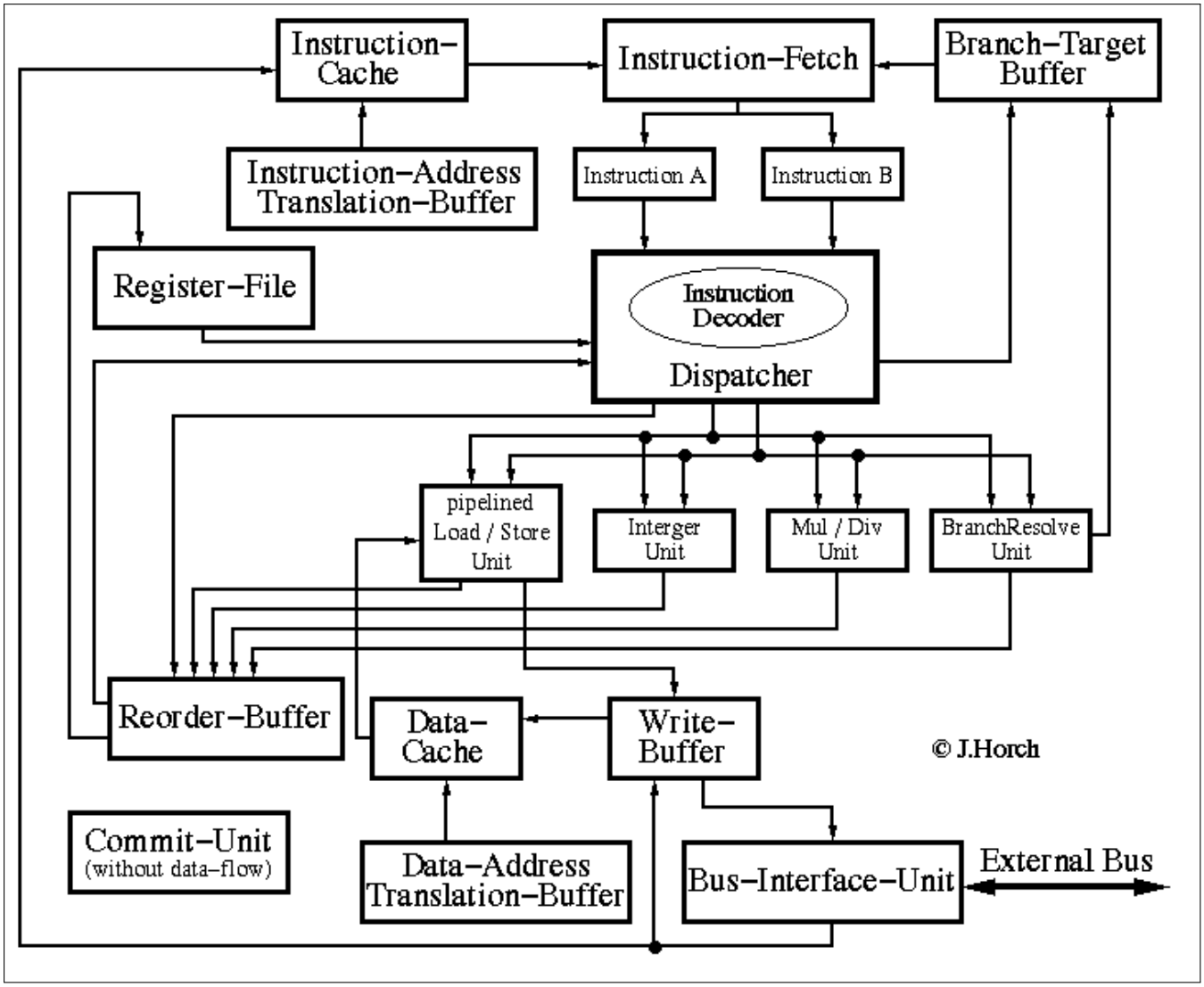

Figure 15: Flow of Data in Superscalar Processor