RISC V Project :

تمرین کامپیوتری شماره ۴ **معماری کامپیوتر**
نیم‌سال دوم **۱۴۰۳-۱۴۰۲**
زمان تحویل: **روز یکشنبه ۲۰ خرداد ۱۴۰۳ (ساعت ۲۳:۵۹)**

هدف از این تمرین کامپیوتری پیاده‌سازی پایپ‌لاین پردازنده‌ی RISC-V با محموعه دستورات زیر است:

- R-Type: add, sub, and, or, slt, sltu
- I-Type: lw, addi, xori, ori, slti, sltiu, jalr
- S-Type: sw
- J-Type: jal
- B-Type: beq, bne, blt, bge
- U-Type: lui

- مدارهای لازم برای تشخیص و برطرف کردن انواع هازاردهایی که در کلاس درس بحث شد را در طراحی خود بگنجانید.

برای تست پردازنده‌ی خود، یک برنامه بنویسید که بزرگترین عنصر یک آرایه‌ی ۱۰ عنصری از اعداد صحیح علامت‌دار ۳۲ بیتی را پیدا کند.

روش ارزیابی:

- پیاده‌سازی پردازنده ۱۰۰ نمره دارد
  - ۲۵ نمره طراحی مسیر داده و واحد کنترل (به همراه کد وریلاگ باید بارگذاری شود)
  - ۱۵ نمره روش کدینگ (مسیر داده به صورت ساختاری و واحد کنترل به صورت ترکیبی)
  - ۴۰ نمره صحت طراحی با برنامه‌ی طراحی شده توسط شما
  - ۲۰ نمره صحت طراحی با برنامه‌ی طراحی شده توسط دستیاران آموزشی

Mohammad Taha Majlesi
810101504

Mohammad Sina Parvisi
810101492

Project Description

Objective:

The objective of this project is to design and implement a RISC-V pipeline processor capable of executing a specific set of instructions. The processor should include mechanisms for detecting and resolving pipeline hazards. The functionality of the processor will be validated using a program that finds the largest element in an array of 10 signed 32-bit integers.

Instruction Types:

The processor should support the following instruction types:

- **R-Type:** Register-to-register operations such as add, sub, and, or, slt, and sltu.
- **I-Type:** Immediate operations such as lw, addi, xori, ori, slti, and sltiu.
- **S-Type:** Store operations such as sw.
- **J-Type:** Jump operations such as jal.
- **B-Type:** Branch operations such as beq, bne, blt, and bge.
- **U-Type:** Upper immediate operations such as lui.

Supported Instructions:

The processor should handle the following RISC-V instructions:

- **R-Type:** add, sub, and, or, slt, sltu
- **I-Type:** lw, addi, xori, ori, slti, sltiu, jalr
- **S-Type:** sw
- **J-Type:** jal
- **B-Type:** beq, bne, blt, bge
- **U-Type:** lui

Pipeline Stages:

The processor pipeline should include the following stages:

1. **Instruction Fetch (IF):** Fetch the instruction from memory.
2. **Instruction Decode (ID):** Decode the instruction and read the registers.
3. **Execute (EX):** Perform the operation specified by the instruction.
4. **Memory Access (MEM):** Access memory for load and store instructions.
5. **Write Back (WB):** Write the result back to the register file.

Hazard Detection and Resolution:

The processor should include circuits to detect and resolve different types of hazards:

- **Data Hazards:** Forwarding and stalling mechanisms should be implemented to handle data dependencies.
- **Control Hazards:** Branch prediction and flushing mechanisms should be implemented to handle control dependencies.

Testing:

To test the processor, write a program that finds the largest element in an array of 10 signed 32-bit integers. This program will be used to verify the correct functionality of the processor.

Evaluation Criteria:

- **Design of Data Path and Control Unit (25 points):** The design should include a clear and efficient implementation of the data path and control unit in Verilog.
- **Coding Methodology (15 points):** The data path should be implemented structurally, and the control unit should be implemented combinationally.
- **Functionality with Provided Program (40 points):** The processor should correctly execute the program written by the student.
- **Functionality with Additional Tests (20 points):** The processor should correctly execute additional programs designed by the teaching assistants.

Deliverables:

- **Verilog Code:** Complete Verilog code for the processor design, including all pipeline stages and hazard detection mechanisms.
- **Test Program:** A program written in assembly to find the largest element in an array of 10 signed 32-bit integers.
- **Documentation:** Detailed documentation explaining the design, implementation, and testing of the processor.

Submission Deadline:

- **Date:** Sunday, June 20, 1403
- **Time:** 11:59 PM

This project aims to solidify your understanding of processor design and pipelining concepts by implementing a functional RISC-V pipeline processor with comprehensive hazard handling mechanisms.

Assembly Code :

at first we have instruction memory :

machine code :

```
03
24
00
00
13
03
40
00
93
23
83
02
63
8e
03
00
83
24
03
00
b3
a0
84
00
63
84
00
00
33
04
90
00
13
03
43
00
6f
f0
5f
fe
37
35
00
00
ef
05
20
00
```

assembly of this :

```
03 24 00 00
13 03 40 00
93 23 83 02
63 8e 03 00
```

```
83 24 03 00
b3 a0 84 00
63 84 00 00
33 04 90 00
13 03 43 00
6f f0 5f fe
37 35 00 00
ef 05 20 00
```

Now, let's convert each one into RISC-V assembly:

1. `03 24 00 00` ⇒ `lw x8, 0(x0)`
2. `13 03 40 00` ⇒ `addi x6, x0, 64`
3. `93 23 83 02` ⇒ `addi x7, x7, 40`
4. `63 8e 03 00` ⇒ `beq x0, x7, label`
5. `83 24 03 00` ⇒ `lw x8, 0(x6)`
6. `b3 a0 84 00` ⇒ `add x9, x9, x8`
7. `63 84 00 00` ⇒ `beq x8, x0, label`
8. `33 04 90 00` ⇒ `add x8, x2, x0`
9. `13 03 43 00` ⇒ `addi x6, x8, 12`
10. `6f f0 5f fe` ⇒ `jal x0, label`
11. `37 35 00 00` ⇒ `lui x10, 0x35`
12. `ef 05 20 00` ⇒ `jal x11, 32`

Here's the RISC-V assembly code:

```
    lw x8, 0(x0)
    addi x6, x0, 64
    addi x7, x7, 40
    beq x0, x7, label
    lw x8, 0(x6)
    add x9, x9, x8
    beq x8, x0, label
    add x8, x2, x0
    addi x6, x8, 12
    jal x0, label
    lui x10, 0x35
    jal x11, 32

label:
    nop
```

ArithmeticLogicUnit:

```
module ArithmeticLogicUnit (input signed [31:0] operandA, operandB, input [2:0] controlSignal, output reg [
    always @(operandA, operandB, controlSignal) begin
        case (controlSignal)
```

```
            3'b000:  result = operandA + operandB;
            3'b001:  result = operandA - operandB;
            3'b010:  result = operandA & operandB;
            3'b011:  result = operandA | operandB;
            3'b100:  result = operandA ^ operandB;
            3'b101:  result = operandA < operandB ? 1'b1 : 1'b0;
            3'b110:  result = $unsigned(operandA) < $unsigned(operandB) ? 1'b1 : 1'b0;
            default: result = {32{1'b0}};
        endcase
    end
    assign zeroFlag = ~|result;
endmodule
```

ArithmeticLogicUnit Module for 32-bit Arithmetic and Logic Operations

Overview

The `ArithmeticLogicUnit` module performs various arithmetic and logic operations on two input operands and produces a 32-bit result.

Module Interface

Inputs

- **operandA [31:0]**: The first 32-bit signed operand.
- **operandB [31:0]**: The second 32-bit signed operand.
- **controlSignal [2:0]**: A 3-bit control signal that determines the operation to be performed.

Outputs

- **result [31:0]**: The 32-bit result of the ALU operation.
- **zeroFlag**: A flag that is asserted (set to 1) if `result` is zero.

Operation

The `ArithmeticLogicUnit` module performs different operations based on the `controlSignal`:

- **3'b000**: Addition (`operandA + operandB`)
- **3'b001**: Subtraction (`operandA - operandB`)
- **3'b010**: Bitwise AND (`operandA & operandB`)
- **3'b011**: Bitwise OR (`operandA | operandB`)
- **3'b100**: Bitwise XOR (`operandA ^ operandB`)
- **3'b101**: Set on less than (`operandA < operandB ? 1 : 0`)
- **3'b110**: Set on less than unsigned (`$unsigned(operandA) < $unsigned(operandB) ? 1 : 0`)
- **default**: Sets `result` to zero (`32'b0`)

The `zeroFlag` output is asserted if the result of the ALU operation (`result`) is zero.

ControlDecoder :

```
module ControlDecoder(input jalrFlag, jumpFlag, branchFlag, zeroFlag, aluResultFlag, input [2:0] funcCode,
    assign pcSource = (jalrFlag) ? 2'b10 :
        ((jumpFlag) ||
        (branchFlag && funcCode == 3'b000 && zeroFlag) || // beq
```

```
        (branchFlag && funcCode == 3'b001 && ~zeroFlag) || // bne
        (branchFlag && funcCode == 3'b100 && aluResultFlag) || // blt
        (branchFlag && funcCode == 3'b101 && ~aluResultFlag)) ? 2'b01 : // bge
        2'b00;
endmodule
```

ControlDecoder Module for Program Counter Source Selection

Overview

The `ControlDecoder` module determines the source for the program counter (`pcSource`) based on various control signals and the function code.

Module Interface

Inputs

- **jalrFlag**: Flag indicating a JALR (Jump And Link Register) instruction.
- **jumpFlag**: Flag indicating a jump instruction.
- **branchFlag**: Flag indicating a branch instruction.
- **zeroFlag**: Flag indicating the result of an ALU operation is zero.
- **aluResultFlag**: Flag indicating the result of an ALU operation.
- **funcCode [2:0]**: The 3-bit function code from the instruction.

Outputs

- **pcSource [1:0]**: The source for the program counter, determining the next instruction to be executed.

Operation

The `ControlDecoder` module sets the `pcSource` output based on the following conditions:

- **jalrFlag**: If `jalrFlag` is set, `pcSource` is set to `2'b10`.
- **jumpFlag**: If `jumpFlag` is set, `pcSource` is set to `2'b01`.
- **Branch Conditions**:
  - **BEQ (Branch if Equal)**: If `branchFlag` is set, `funcCode` is `3'b000`, and `zeroFlag` is set, `pcSource` is set to `2'b01`.
  - **BNE (Branch if Not Equal)**: If `branchFlag` is set, `funcCode` is `3'b001`, and `zeroFlag` is not set, `pcSource` is set to `2'b01`.
  - **BLT (Branch if Less Than)**: If `branchFlag` is set, `funcCode` is `3'b100`, and `aluResultFlag` is set, `pcSource` is set to `2'b01`.
  - **BGE (Branch if Greater Than or Equal)**: If `branchFlag` is set, `funcCode` is `3'b101`, and `aluResultFlag` is not set, `pcSource` is set to `2'b01`.

If none of the above conditions are met, `pcSource` is set to `2'b00`.

ControlUnit:

```
`define R_TYPE 7'b0110011
`define I_TYPE 7'b0010011
`define LW 7'b0000011
`define JALR 7'b1100111
`define S_TYPE 7'b0100011
`define J_TYPE 7'b1101111
`define B_TYPE 7'b1100011
`define U_TYPE 7'b0110111
```

```
`define ADD 10'b0000000000
`define SUB 10'b0100000000
`define AND 10'b0000000111
`define OR 10'b0000000110
`define SLT 10'b0000000010
`define SLTU 10'b0000000011
`define LW_FUNC 3'b010
`define ADDI 3'b000
`define XORI 3'b100
`define ORI 3'b110
`define SLTI 3'b010
`define JALR_FUNC 3'b000
`define SLTIU 3'b011
`define BEQ 3'b000
`define BNE 3'b001
`define BLT 3'b100
`define BGE 3'b101

module ControlUnit(input [2:0] function3, input [6:0] function7, opcode, output reg memWrite, aluSrc, regWr
    wire [9:0] functionn;
    assign functionn = {function7, function3};

    always @(function3, function7, opcode) begin
        {memWrite, aluSrc, regWrite, jump, branch, jalr, resultSrc, aluControl, immSrc} = 14'b0000000000000

        case (opcode)
            `R_TYPE: begin
                regWrite = 1'b1;
                case (functionn)
                    `ADD: aluControl = 3'b000;
                    `SUB: aluControl = 3'b001;
                    `AND: aluControl = 3'b010;
                    `OR: aluControl = 3'b011;
                    `SLT: aluControl = 3'b101;
                    `SLTU: aluControl = 3'b110;
                endcase
            end

            `LW: {regWrite, resultSrc, aluSrc} = 4'b1011;

            `I_TYPE: begin
                {aluSrc, regWrite} = 2'b11;
                case (function3)
                    `ADDI:;
                    `XORI: aluControl = 3'b100;
                    `ORI: aluControl = 3'b011;
                    `SLTI: aluControl = 3'b101;
                    `SLTIU: aluControl = 3'b110;
                endcase
            end

            `JALR: {jalr, aluSrc, resultSrc, regWrite} = 5'b11101;

            `S_TYPE: {immSrc, aluSrc, memWrite} = 5'b00111;

            `J_TYPE: {resultSrc, immSrc, regWrite, jump} = 7'b1001011;

            `B_TYPE: begin
                {branch, immSrc} = 4'b1011;
                case (function3)
```

```
                    `BEQ: aluControl = 3'b001;
                    `BNE: aluControl = 3'b001;
                    `BLT: aluControl = 3'b101;
                    `BGE: aluControl = 3'b101;
                endcase
            end

            `U_TYPE: {resultSrc, immSrc, regWrite} = 6'b111001;
        endcase
    end
endmodule
```

ControlUnit Module for RISC-V Control Logic

Overview

The `ControlUnit` module decodes the instruction opcode and function fields to generate control signals for the RISC-V processor.

Module Interface

Inputs

- **function3 [2:0]**: The 3-bit function field.
- **function7 [6:0]**: The 7-bit function field.
- **opcode [6:0]**: The 7-bit opcode field.

Outputs

- **memWrite**: Control signal to enable memory write.
- **aluSrc**: Control signal to select the ALU source.
- **regWrite**: Control signal to enable register write.
- **jump**: Control signal for jump instructions.
- **branch**: Control signal for branch instructions.
- **jalr**: Control signal for JALR instructions.
- **resultSrc [1:0]**: Control signal to select the result source.
- **aluControl [2:0]**: Control signal to select the ALU operation.
- **immSrc [2:0]**: Control signal to select the immediate source.

Operation

The `ControlUnit` module generates control signals based on the input opcode and function fields to control the execution of instructions in the RISC-V processor.

- **R_TYPE (7'b0110011)**: Generates control signals for R-type instructions.
  - **ADD (10'b0000000000)**: ALU performs addition.
  - **SUB (10'b0100000000)**: ALU performs subtraction.
  - **AND (10'b0000000111)**: ALU performs bitwise AND.
  - **OR (10'b0000000110)**: ALU performs bitwise OR.
  - **SLT (10'b0000000010)**: ALU performs set on less than (signed).
  - **SLTU (10'b0000000011)**: ALU performs set on less than (unsigned).
- **LW (7'b0000011)**: Generates control signals for load word instructions.
- **I_TYPE (7'b0010011)**: Generates control signals for I-type instructions.

    - **ADDI (3'b000)**: ALU performs addition with immediate.
    - **XORI (3'b100)**: ALU performs bitwise XOR with immediate.
    - **ORI (3'b110)**: ALU performs bitwise OR with immediate.
    - **SLTI (3'b010)**: ALU performs set on less than immediate (signed).
    - **SLTIU (3'b011)**: ALU performs set on less than immediate (unsigned).
- **JALR (7'b1100111)**: Generates control signals for JALR instructions.
- **S_TYPE (7'b0100011)**: Generates control signals for store instructions.
- **J_TYPE (7'b1101111)**: Generates control signals for jump instructions.
- **B_TYPE (7'b1100011)**: Generates control signals for branch instructions.
    - **BEQ (3'b000)**: ALU performs branch if equal.
    - **BNE (3'b001)**: ALU performs branch if not equal.
    - **BLT (3'b100)**: ALU performs branch if less than.
    - **BGE (3'b101)**: ALU performs branch if greater than or equal.
- **U_TYPE (7'b0110111)**: Generates control signals for upper immediate instructions.

ControlUnit Module for RISC-V Control Logic

Overview

The `ControlUnit` module decodes the instruction opcode and function fields to generate control signals for the RISC-V processor.

Module Interface

Inputs

- **function3 [2:0]**: The 3-bit function field.
- **function7 [6:0]**: The 7-bit function field.
- **opcode [6:0]**: The 7-bit opcode field.

Outputs

- **memWrite**: Control signal to enable memory write.
- **aluSrc**: Control signal to select the ALU source.
- **regWrite**: Control signal to enable register write.
- **jump**: Control signal for jump instructions.
- **branch**: Control signal for branch instructions.
- **jalr**: Control signal for JALR instructions.
- **resultSrc [1:0]**: Control signal to select the result source.
- **aluControl [2:0]**: Control signal to select the ALU operation.
- **immSrc [2:0]**: Control signal to select the immediate source.

Operation

The `ControlUnit` module generates control signals based on the input opcode and function fields to control the execution of instructions in the RISC-V processor.

- **R_TYPE (7'b0110011)**: Generates control signals for R-type instructions.
    - **ADD (10'b0000000000)**: ALU performs addition.
    - **SUB (10'b0100000000)**: ALU performs subtraction.
    - **AND (10'b0000000111)**: ALU performs bitwise AND.
    - **OR (10'b0000000110)**: ALU performs bitwise OR.

    - **SLT (10'b0000000010)**: ALU performs set on less than (signed).
    - **SLTU (10'b0000000011)**: ALU performs set on less than (unsigned).
- **LW (7'b0000011)**: Generates control signals for load word instructions.
- **I_TYPE (7'b0010011)**: Generates control signals for I-type instructions.
    - **ADDI (3'b000)**: ALU performs addition with immediate.
    - **XORI (3'b100)**: ALU performs bitwise XOR with immediate.
    - **ORI (3'b110)**: ALU performs bitwise OR with immediate.
    - **SLTI (3'b010)**: ALU performs set on less than immediate (signed).
    - **SLTIU (3'b011)**: ALU performs set on less than immediate (unsigned).
- **JALR (7'b1100111)**: Generates control signals for JALR instructions.
- **S_TYPE (7'b0100011)**: Generates control signals for store instructions.
- **J_TYPE (7'b1101111)**: Generates control signals for jump instructions.
- **B_TYPE (7'b1100011)**: Generates control signals for branch instructions.
    - **BEQ (3'b000)**: ALU performs branch if equal.
    - **BNE (3'b001)**: ALU performs branch if not equal.
    - **BLT (3'b100)**: ALU performs branch if less than.
    - **BGE (3'b101)**: ALU performs branch if greater than or equal.
- **U_TYPE (7'b0110111)**: Generates control signals for upper immediate instructions.

DataPath:

```
module DataPath (
    input clk, MemWriteD, ALUSrcD, RegWriteD, JumpD, BranchD, JalrD, StallF, StallD, FlushD, FlushE,
    input [1:0] ResultSrcD, ForwardAE, ForwardBE, PCSrcE,
    input [2:0] ALUControlD, ImmSrcD,
    output BranchE, JumpE, JalrE, ZeroE, ALUResult0, RegWriteM, RegWriteW,
    output [1:0] ResultSrcE,
    output [2:0] func3D, func3E,
    output [6:0] func7, op,
    output [4:0] Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW,
    output [1:0] ResultSrcM, ResultSrcW
);

    wire [31:0] PCPlus4F, PCTargetE, ALUResultE, PCF_, PCF, InstrF, InstrD, ResultW, RD1D, RD2D, ExtImmD, S
        WriteDataE, ExtImmE, SrcBE, PCE, WriteDataM, ReadDataM, ALUResultW, ReadDataW, PCPlus4W, ExtImmW, P
    wire [4:0] RdD;
    wire [2:0] ALUControlE;
    wire MemWriteM, RegWriteE, MemWriteE, ALUSrcE;

    assign Rs1D = InstrD[19:15];
    assign Rs2D = InstrD[24:20];
    assign RdD = InstrD[11:7];
    assign func3D = InstrD[14:12];
    assign func7 = InstrD[31:25];
    assign op = InstrD[6:0];

    Mux3to1 PCMux(PCSrcE, PCPlus4F, PCTargetE, ALUResultE, PCF_);
    Register PCReg(clk, ~StallF, 1'b0, PCF_, PCF);
    InstructionMemory InstructionMemory(PCF, InstrF);
    SumModule PCAdd4(PCF, 4, PCPlus4F);
    IF_ID_Regs IF_ID_Reg(clk, FlushD, ~StallD, InstrF, PCF, PCPlus4F, InstrD, PCD, PCPlus4D);
```

```
    RegisterFile RegFile(clk, RegWriteW, InstrD[19:15], InstrD[24:20], RdW, ResultW, RD1D, RD2D);
    ImmediateExtend Extend(ImmSrcD, InstrD[31:7], ExtImmD);
    ID_EX_Regs ID_EX_Reg(clk, FlushE, RegWriteD, MemWriteD, ALUSrcD, JumpD, BranchD, JalrD, ResultSrcD, ALU
        func3D, Rs1D, Rs2D, RdD, PCD, ExtImmD, PCPlus4D, RD1D, RD2D, RegWriteE, MemWriteE, ALUSrcE, JumpE,
        ResultSrcE, ALUControlE, func3E, Rs1E, Rs2E, RdE, PCE, ExtImmE, PCPlus4E, RD1E, RD2E);
    Mux4to1 ForwardAMux(ForwardAE, RD1E, ResultW, ALUResultM, ExtImmE, SrcAE);
    Mux4to1 ForwardBMux(ForwardBE, RD2E, ResultW, ALUResultM, ExtImmE, WriteDataE);
    Mux2to1 ALUSrcBMux(ALUSrcE, WriteDataE, ExtImmE, SrcBE);
    SumModule PCImmAdd(PCE, ExtImmE, PCTargetE);
    ArithmeticLogicUnit ALU(SrcAE, SrcBE, ALUControlE, ALUResultE, ZeroE);
    EX_Mem_Regs EX_Mem_Reg(clk, RegWriteE, MemWriteE, ResultSrcE, RdE, ALUResultE, WriteDataE, ExtImmE,
        PCPlus4E, RegWriteM, MemWriteM, ResultSrcM, RdM, ALUResultM, WriteDataM, ExtImmM, PCPlus4M);
    MemoryUnit DataMemory(clk, MemWriteM, ALUResultM, WriteDataM, ReadDataM);
    Mem_WB_Regs Mem_WB_Reg(clk, RegWriteM, ResultSrcM, RdM, ALUResultM, ReadDataM, ExtImmM, PCPlus4M,
        RegWriteW, ResultSrcW, RdW, ALUResultW, ReadDataW, ExtImmW, PCPlus4W);
    Mux4to1 ResultSrcMux(ResultSrcW, ALUResultW, ReadDataW, PCPlus4W, ExtImmW, ResultW);

endmodule
```

DataPath Module for CPU Data Path

Overview

The `DataPath` module implements the data path of a CPU, handling instruction fetch, decode, execute, memory access, and write-back stages.

Module Interface

Inputs

- **clk**: Clock signal.
- **MemWriteD**: Memory write enable signal for the decode stage.
- **ALUSrcD**: ALU source select signal for the decode stage.
- **RegWriteD**: Register write enable signal for the decode stage.
- **JumpD**: Jump signal for the decode stage.
- **BranchD**: Branch signal for the decode stage.
- **JalrD**: JALR signal for the decode stage.
- **StallF**: Stall signal for the fetch stage.
- **StallD**: Stall signal for the decode stage.
- **FlushD**: Flush signal for the decode stage.
- **FlushE**: Flush signal for the execute stage.
- **ResultSrcD [1:0]**: Result source select signal for the decode stage.
- **ForwardAE [1:0]**: Forwarding control signal for ALU source A.
- **ForwardBE [1:0]**: Forwarding control signal for ALU source B.
- **PCSrcE [1:0]**: Program counter source select signal for the execute stage.
- **ALUControlD [2:0]**: ALU control signal for the decode stage.
- **ImmSrcD [2:0]**: Immediate source select signal for the decode stage.

Outputs

- **BranchE**: Branch signal for the execute stage.
- **JumpE**: Jump signal for the execute stage.
- **JalrE**: JALR signal for the execute stage.

- **ZeroE**: Zero flag from the ALU in the execute stage.
- **ALUResult0**: ALU result (not used).
- **RegWriteM**: Register write enable signal for the memory stage.
- **RegWriteW**: Register write enable signal for the write-back stage.
- **ResultSrcE [1:0]**: Result source select signal for the execute stage.
- **func3D [2:0]**: Function field from the decode stage.
- **func3E [2:0]**: Function field from the execute stage.
- **func7 [6:0]**: Function field from the decode stage.
- **op [6:0]**: Opcode from the decode stage.
- **Rs1D [4:0]**: Source register 1 from the decode stage.
- **Rs2D [4:0]**: Source register 2 from the decode stage.
- **Rs1E [4:0]**: Source register 1 from the execute stage.
- **Rs2E [4:0]**: Source register 2 from the execute stage.
- **RdE [4:0]**: Destination register from the execute stage.
- **RdM [4:0]**: Destination register from the memory stage.
- **RdW [4:0]**: Destination register from the write-back stage.
- **ResultSrcM [1:0]**: Result source select signal for the memory stage.
- **ResultSrcW [1:0]**: Result source select signal for the write-back stage.

Operation

The `DataPath` module coordinates the flow of data through the CPU, ensuring that instructions are fetched, decoded, executed, and their results are written back to the appropriate registers. It uses several submodules to achieve this:

- **Mux3to1**: Multiplexes three inputs based on a select signal.
- **Register**: Stores the current value of the program counter.
- **InstMemory**: Fetches instructions from memory.
- **Adder**: Calculates the next program counter value.
- **IF_ID_Regs**: Holds values between the fetch and decode stages.
- **RegisterFile**: Manages the CPU's registers.
- **ImmExtend**: Extends immediate values to the correct width.
- **ID_EX_Regs**: Holds values between the decode and execute stages.
- **Mux4to1**: Multiplexes four inputs based on a select signal.
- **ALU**: Performs arithmetic and logic operations.
- **EX_Mem_Regs**: Holds values between the execute and memory stages.
- **DataMem**: Manages data memory access.
- **Mem_WB_Regs**: Holds values between the memory and write-back stages.
- **ResultSrcMux**: Selects the final result to be written back to the registers.

DFlipFlop:

```
module DFlipFlop(input clk, en, sclr, d, output reg q = 1'b0);
    always @(posedge clk) begin
        if (sclr)
            q <= 1'b0;
        else if (en)
```

```
            q <= d;
    end
endmodule
```

DFlipFlop Module for D Flip-Flop with Enable and Synchronous Clear

Overview

The `DFlipFlop` module implements a D flip-flop with enable and synchronous clear functionality.

Module Interface

Inputs

- **clk**: Clock signal.
- **en**: Enable signal. When high, the D flip-flop updates its output.
- **sclr**: Synchronous clear signal. When high, the output is reset to 0 on the clock edge.
- **d**: Data input. The value to be stored in the flip-flop when `en` is high and `sclr` is low.

Outputs

- **q**: Output of the D flip-flop. It holds the stored value.

Operation

The `DFlipFlop` module updates its output `q` based on the following conditions:

- **Synchronous Clear (sclr)**: If `sclr` is high on the rising edge of the clock, `q` is set to 0.
- **Enable (en)**: If `en` is high on the rising edge of the clock and `sclr` is low, `q` is updated to the value of `d`.

HazardUnit:

```
module HazardUnit(
    input RegWriteM, RegWriteW,
    input [1:0] ResultSrcE, PCSrcE, ResultSrcM, ResultSrcW,
    input [4:0] Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW,
    output StallF, StallD, FlushD, FlushE,
    output [1:0] ForwardAE, ForwardBE
);
    wire loadStall;

    assign ForwardAE = ((Rs1E == RdM && RegWriteM) && Rs1E != 0) ? 2'b10 :
                       (((Rs1E == RdW && RegWriteW) && Rs1E != 0) || (Rs1E == RdW && ResultSrcW == 2'b11 &&
                       (Rs1E == RdM && ResultSrcM == 2'b11 && Rs1E != 0) ? 2'b11 :
                       2'b00;

    assign ForwardBE = ((Rs2E == RdM && RegWriteM) && Rs2E != 0) ? 2'b10 :
                       (((Rs2E == RdW && RegWriteW) && Rs2E != 0) || (Rs2E == RdW && ResultSrcW == 2'b11 &&
                       (Rs2E == RdM && ResultSrcM == 2'b11 && Rs2E != 0) ? 2'b11 :
                       2'b00;

    assign loadStall = ((Rs1D == RdE) || (Rs2D == RdE)) && ResultSrcE == 2'b01;
    assign StallF = loadStall;
    assign StallD = loadStall;
    assign FlushE = loadStall || (PCSrcE == 2'b01) || (PCSrcE == 2'b10);
```

```
    assign FlushD = (PCSrcE == 2'b01) ? 1 : 0;
endmodule
```

HazardUnit Module for Handling Pipeline Hazards

Overview

The `HazardUnit` module detects and handles pipeline hazards in a CPU to ensure correct instruction execution.

Module Interface

Inputs

- **RegWriteM**: Register write enable signal for the memory stage.
- **RegWriteW**: Register write enable signal for the write-back stage.
- **ResultSrcE [1:0]**: Result source select signal for the execute stage.
- **PCSrcE [1:0]**: Program counter source select signal for the execute stage.
- **ResultSrcM [1:0]**: Result source select signal for the memory stage.
- **ResultSrcW [1:0]**: Result source select signal for the write-back stage.
- **Rs1D [4:0]**: Source register 1 from the decode stage.
- **Rs2D [4:0]**: Source register 2 from the decode stage.
- **Rs1E [4:0]**: Source register 1 from the execute stage.
- **Rs2E [4:0]**: Source register 2 from the execute stage.
- **RdE [4:0]**: Destination register from the execute stage.
- **RdM [4:0]**: Destination register from the memory stage.
- **RdW [4:0]**: Destination register from the write-back stage.

Outputs

- **StallF**: Stall signal for the fetch stage.
- **StallD**: Stall signal for the decode stage.
- **FlushD**: Flush signal for the decode stage.
- **FlushE**: Flush signal for the execute stage.
- **ForwardAE [1:0]**: Forwarding control signal for ALU source A.
- **ForwardBE [1:0]**: Forwarding control signal for ALU source B.

Operation

The `HazardUnit` module handles data hazards and control hazards by generating control signals for stalling and flushing the pipeline, as well as forwarding signals for resolving data dependencies.

Forwarding Logic

- **ForwardAE**: Selects the source for ALU operand A.
  - `2'b10`: Forward from memory stage (`RegWriteM` is set and `Rs1E == RdM`).
  - `2'b01`: Forward from write-back stage (`RegWriteW` is set and `Rs1E == RdW` or `ResultSrcW == 2'b11`).
  - `2'b11`: Forward from memory stage with special handling (`ResultSrcM == 2'b11`).
  - `2'b00`: No forwarding.
- **ForwardBE**: Selects the source for ALU operand B.
  - `2'b10`: Forward from memory stage (`RegWriteM` is set and `Rs2E == RdM`).
  - `2'b01`: Forward from write-back stage (`RegWriteW` is set and `Rs2E == RdW` or `ResultSrcW == 2'b11`).

- `2'b11`: Forward from memory stage with special handling (`ResultSrcM == 2'b11`).
- `2'b00`: No forwarding.

Stalling and Flushing Logic

- **loadStall**: Detects load-use hazards when the result source of the execute stage is a load instruction.
  - `loadStall = ((Rs1D == RdE) || (Rs2D == RdE)) && ResultSrcE == 2'b01`.
- **StallF**: Stalls the fetch stage if there is a load-use hazard.
  - `StallF = loadStall`.
- **StallD**: Stalls the decode stage if there is a load-use hazard.
  - `StallD = loadStall`.
- **FlushE**: Flushes the execute stage on load-use hazards or branch/jump instructions.
  - `FlushE = loadStall || (PCSrcE == 2'b01) || (PCSrcE == 2'b10)`.
- **FlushD**: Flushes the decode stage on branch instructions.
  - `FlushD = (PCSrcE == 2'b01) ? 1 : 0`.

HazardUnit:

```
`define I_TYPE 3'b000
`define S_TYPE 3'b001
`define J_TYPE 3'b010
`define B_TYPE 3'b011
`define U_TYPE 3'b100

module ImmediateExtend(input [2:0] immSelect, input [31:7] instructionPart, output reg [31:0] extendedImmed
    always @(immSelect, instructionPart) begin
        case(immSelect)
            `I_TYPE: extendedImmediate = {{20{instructionPart[31]}}, instructionPart[31:20]};
            `S_TYPE: extendedImmediate = {{20{instructionPart[31]}}, instructionPart[31:25], instructionPar
            `J_TYPE: extendedImmediate = {{12{instructionPart[31]}}, instructionPart[31], instructionPart[1
            `B_TYPE: extendedImmediate = {{19{instructionPart[31]}}, instructionPart[31], instructionPart[7
            `U_TYPE: extendedImmediate = {instructionPart[31:12], {12{1'b0}}};
            default: extendedImmediate = {32{1'b0}};
        endcase
    end
endmodule
```

HazardUnit Module for Handling Pipeline Hazards

Overview

The `HazardUnit` module detects and handles pipeline hazards in a CPU to ensure correct instruction execution.

Module Interface

Inputs

- **RegWriteM**: Register write enable signal for the memory stage.

- **RegWriteW**: Register write enable signal for the write-back stage.
- **ResultSrcE [1:0]**: Result source select signal for the execute stage.
- **PCSrcE [1:0]**: Program counter source select signal for the execute stage.
- **ResultSrcM [1:0]**: Result source select signal for the memory stage.
- **ResultSrcW [1:0]**: Result source select signal for the write-back stage.
- **Rs1D [4:0]**: Source register 1 from the decode stage.
- **Rs2D [4:0]**: Source register 2 from the decode stage.
- **Rs1E [4:0]**: Source register 1 from the execute stage.
- **Rs2E [4:0]**: Source register 2 from the execute stage.
- **RdE [4:0]**: Destination register from the execute stage.
- **RdM [4:0]**: Destination register from the memory stage.
- **RdW [4:0]**: Destination register from the write-back stage.

Outputs

- **StallF**: Stall signal for the fetch stage.
- **StallD**: Stall signal for the decode stage.
- **FlushD**: Flush signal for the decode stage.
- **FlushE**: Flush signal for the execute stage.
- **ForwardAE [1:0]**: Forwarding control signal for ALU source A.
- **ForwardBE [1:0]**: Forwarding control signal for ALU source B.

Operation

The `HazardUnit` module handles data hazards and control hazards by generating control signals for stalling and flushing the pipeline, as well as forwarding signals for resolving data dependencies.

Forwarding Logic

- **ForwardAE**: Selects the source for ALU operand A.
  - `2'b10`: Forward from memory stage (`RegWriteM` is set and `Rs1E == RdM`).
  - `2'b01`: Forward from write-back stage (`RegWriteW` is set and `Rs1E == RdW` or `ResultSrcW == 2'b11`).
  - `2'b11`: Forward from memory stage with special handling (`ResultSrcM == 2'b11`).
  - `2'b00`: No forwarding.
- **ForwardBE**: Selects the source for ALU operand B.
  - `2'b10`: Forward from memory stage (`RegWriteM` is set and `Rs2E == RdM`).
  - `2'b01`: Forward from write-back stage (`RegWriteW` is set and `Rs2E == RdW` or `ResultSrcW == 2'b11`).
  - `2'b11`: Forward from memory stage with special handling (`ResultSrcM == 2'b11`).
  - `2'b00`: No forwarding.

Stalling and Flushing Logic

- **loadStall**: Detects load-use hazards when the result source of the execute stage is a load instruction.
  - `loadStall = ((Rs1D == RdE) || (Rs2D == RdE)) && ResultSrcE == 2'b01`.
- **StallF**: Stalls the fetch stage if there is a load-use hazard.
  - `StallF = loadStall`.
- **StallD**: Stalls the decode stage if there is a load-use hazard.
  - `StallD = loadStall`.
- **FlushE**: Flushes the execute stage on load-use hazards or branch/jump instructions.

- `FlushE = loadStall || (PCSrcE == 2'b01) || (PCSrcE == 2'b10)`.

- **FlushD**: Flushes the decode stage on branch instructions.
  - `FlushD = (PCSrcE == 2'b01) ? 1 : 0`.

InstructionMemory:

```
module InstructionMemory(input [31:0] address, output [31:0] readData);
    reg [7:0] memory [0:$pow(2, 16)-1];

    wire [31:0] alignedAddress;
    assign alignedAddress = {address[31:2], 2'b00};

    initial $readmemh("instructions.mem", memory);
    assign readData = {memory[alignedAddress + 3], memory[alignedAddress + 2], memory[alignedAddress + 1],
endmodule
```

InstructionMemory Module for Instruction Fetch

Overview

The `InstructionMemory` module implements a memory that stores instructions and allows fetching instructions based on an address.

Module Interface

Inputs

- **address [31:0]**: The 32-bit address for fetching the instruction.

Outputs

- **readData [31:0]**: The 32-bit instruction fetched from memory.

Operation

The `InstructionMemory` module reads a 32-bit instruction from memory based on the provided address.

Memory Initialization

- The memory is initialized using the `$readmemh` function to load instructions from the `instructions.mem` file.

Address Alignment

- The `alignedAddress` is calculated by aligning the 32-bit address to a 4-byte boundary.

Instruction Fetch

- The instruction fetch operation is performed by combining four 8-bit memory locations into a 32-bit `readData` output.

MemoryUnit:

```
module MemoryUnit (input clk, we, input [31:0] address, writeData, output reg [31:0] readData);
    reg [7:0] memory [0:$pow(2, 16)-1];
    wire [31:0] alignedAddress;

    assign alignedAddress = {address[31:2], 2'b00};
```

```
    initial $readmemb("data.mem", memory);

    always @(address or alignedAddress) begin
        readData = {memory[alignedAddress + 3], memory[alignedAddress + 2], memory[alignedAddress + 1], mem
    end

    always @(posedge clk) begin
        if (we)
            {memory[alignedAddress + 3], memory[alignedAddress + 2], memory[alignedAddress + 1], memory[ali
        else
            memory[alignedAddress] <= memory[alignedAddress];
    end
endmodule
```

MemoryUnit Module for 32-bit Data Memory

Overview

The `MemoryUnit` module implements a 32-bit data memory with read and write capabilities, using a byte-addressable memory array.

Module Interface

Inputs

- **clk**: Clock signal.
- **we**: Write enable signal.
- **address [31:0]**: The 32-bit address for memory access.
- **writeData [31:0]**: The 32-bit data to be written to memory.

Outputs

- **readData [31:0]**: The 32-bit data read from memory.

Operation

The `MemoryUnit` module performs memory read and write operations based on the input signals.

Memory Initialization

- The memory is initialized using the `$readmemb` function to load data from the `data.mem` file.

Memory Read

- The memory read operation is performed by combining four 8-bit memory locations into a 32-bit `readData` output.
- The `alignedAddress` is calculated by aligning the 32-bit address to a 4-byte boundary.

Memory Write

- The memory write operation occurs on the positive edge of the clock (`posedge clk`) when the write enable (`we`) signal is asserted.
- Four 8-bit memory locations are combined to store the 32-bit `writeData` input.

Example Usage

Below is an example of how the `MemoryUnit` module can be instantiated and used in a higher-level Verilog module.

```
module TopModule;
    reg clk;
    reg we;
    reg [31:0] address;
    reg [31:0] writeData;
```

```
    wire [31:0] readData;

    // Instantiate the MemoryUnit
    MemoryUnit memory_instance(.clk(clk), .we(we), .address(address), .writeData(writeData), .readData(r
eadData));

    initial begin
        // Initialize signals
        clk = 0;
        we = 0;
        address = 32'h00000000;
        writeData = 32'h12345678;

        // Toggle clock and perform write operation
        #10 clk = 1; we = 1;
        #10 clk = 0; we = 0;

        // Toggle clock and perform read operation
        #10 clk = 1;
        #10 clk = 0;

        $display("Read Data: %h", readData);
    end
endmodule
```

```
module Mux2to1(input sel, input [31:0] a, b, output[31:0] out);
    assign out = (sel == 0) ? a : b;
endmodule
```

```
module Mux3to1(input [1:0] sel, input[31:0] a, b, c, output [31:0] out);
    assign out= (sel == 2'b00) ? a:
            (sel == 2'b01) ? b:
            (sel == 2'b10) ? c:
            {32{1'b0}};
endmodule
```

```
module Mux4to1(input [1:0] sel, input[31:0] a, b, c, d, output [31:0] out);
    assign out= (sel == 2'b00) ? a:
            (sel == 2'b01) ? b:
            (sel == 2'b10) ? c:
            (sel == 2'b11) ? d:
            {32{1'b0}};
endmodule
```

Register:

```
module Register #(parameter N = 32) (
    input clk, en, sclr,
    input [N-1:0] d,
    output reg [N-1:0] q = {N{1'b0}}
);
    always @(posedge clk) begin
        if (sclr)
            q <= {N{1'b0}};
        else if (en)
            q <= d;
    end
endmodule
```

Register Module for Parameterized Register with Enable and Synchronous Clear

Overview

The `Register` module implements a parameterized register with enable and synchronous clear functionality.

Module Interface

Parameters

- **N**: Width of the register. Default is 32 bits.

Inputs

- **clk**: Clock signal.
- **en**: Enable signal. When high, the register updates its output.
- **sclr**: Synchronous clear signal. When high, the output is reset to 0 on the clock edge.
- **d [N-1:0]**: Data input. The value to be stored in the register when `en` is high and `sclr` is low.

Outputs

- **q [N-1:0]**: Output of the register. It holds the stored value.

Operation

The `Register` module updates its output `q` based on the following conditions:

- **Synchronous Clear (sclr)**: If `sclr` is high on the rising edge of the clock, `q` is set to 0.
- **Enable (en)**: If `en` is high on the rising edge of the clock and `sclr` is low, `q` is updated to the value of `d`.

```
module IF_ID_Regs(
    input clk, sclr, en,
    input [31:0] InstrF, PCF, PCPlus4F,
    output [31:0] InstrD, PCD, PCPlus4D
);
    Register R1(clk, en, sclr, InstrF, InstrD);
    Register R2(clk, en, sclr, PCF, PCD);
    Register R3(clk, en, sclr, PCPlus4F, PCPlus4D);
```

```
endmodule

module ID_EX_Regs(
    input clk, sclr, RegWriteD, MemWriteD, ALUSrcD, JumpD, BranchD, JalrD,
    input [1:0] ResultSrcD,
    input [2:0] ALUControlD, func3D,
    input [4:0] Rs1D, Rs2D, RdD,
    input [31:0] PCD, ExtImmD, PCPlus4D, RD1D, RD2D,
    output RegWriteE, MemWriteE, ALUSrcE, JumpE, BranchE, JalrE,
    output [1:0] ResultSrcE,
    output [2:0] ALUControlE, func3E,
    output [4:0] Rs1E, Rs2E, RdE,
    output [31:0] PCE, ExtImmE, PCPlus4E, RD1E, RD2E
);
    DFlipFlop D1(clk, 1'b1, sclr, RegWriteD, RegWriteE);
    DFlipFlop D2(clk, 1'b1, sclr, MemWriteD, MemWriteE);
    DFlipFlop D3(clk, 1'b1, sclr, JumpD, JumpE);
    DFlipFlop D4(clk, 1'b1, sclr, ALUSrcD, ALUSrcE);
    DFlipFlop D5(clk, 1'b1, sclr, BranchD, BranchE);
    DFlipFlop D6(clk, 1'b1, sclr, JalrD, JalrE);
    Register #(2) R4 (clk, 1'b1, sclr, ResultSrcD, ResultSrcE);
    Register #(3) R5 (clk, 1'b1, sclr, ALUControlD, ALUControlE);
    Register #(3) R6 (clk, 1'b1, sclr, func3D, func3E);
    Register #(5) R7 (clk, 1'b1, sclr, Rs1D, Rs1E);
    Register #(5) R8 (clk, 1'b1, sclr, Rs2D, Rs2E);
    Register #(5) R9 (clk, 1'b1, sclr, RdD, RdE);
    Register R10 (clk, 1'b1, sclr, PCD, PCE);
    Register R11(clk, 1'b1, sclr, ExtImmD, ExtImmE);
    Register R12(clk, 1'b1, sclr, PCPlus4D, PCPlus4E);
    Register R13(clk, 1'b1, sclr, RD1D, RD1E);
    Register R14(clk, 1'b1, sclr, RD2D, RD2E);
endmodule

module EX_Mem_Regs(
    input clk, RegWriteE, MemWriteE,
    input [1:0] ResultSrcE,
    input [4:0] RdE,
    input [31:0] ALUResultE, WriteDataE, ExtImmE, PCPlus4E,
    output RegWriteM, MemWriteM,
    output [1:0] ResultSrcM,
    output [4:0] RdM,
    output [31:0] ALUResultM, WriteDataM, ExtImmM, PCPlus4M
);
    DFlipFlop D7(clk, 1'b1, 1'b0, RegWriteE, RegWriteM);
    DFlipFlop D8(clk, 1'b1, 1'b0, MemWriteE, MemWriteM);
    Register #(2) R15 (clk, 1'b1, 1'b0, ResultSrcE, ResultSrcM);
    Register #(5) R16 (clk, 1'b1, 1'b0, RdE, RdM);
    Register R17 (clk, 1'b1, 1'b0, ALUResultE, ALUResultM);
    Register R18 (clk, 1'b1, 1'b0, WriteDataE, WriteDataM);
    Register R19 (clk, 1'b1, 1'b0, ExtImmE, ExtImmM);
    Register R20 (clk, 1'b1, 1'b0, PCPlus4E, PCPlus4M);
endmodule

module Mem_WB_Regs(
    input clk, RegWriteM,
    input [1:0] ResultSrcM,
    input [4:0] RdM,
    input [31:0] ALUResultM, ReadDataM, ExtImmM, PCPlus4M,
    output RegWriteW,
    output [1:0] ResultSrcW,
```

```
    output [4:0] RdW,
    output [31:0] ALUResultW, ReadDataW, ExtImmW, PCPlus4W
);
    DFlipFlop D9(clk, 1'b1, 1'b0, RegWriteM, RegWriteW);
    Register #(2) R21 (clk, 1'b1, 1'b0, ResultSrcM, ResultSrcW);
    Register #(5) R22 (clk, 1'b1, 1'b0, RdM, RdW);
    Register R23 (clk, 1'b1, 1'b0, ALUResultM, ALUResultW);
    Register R24 (clk, 1'b1, 1'b0, ReadDataM, ReadDataW);
    Register R25 (clk, 1'b1, 1'b0, ExtImmM, ExtImmW);
    Register R26 (clk, 1'b1, 1'b0, PCPlus4M, PCPlus4W);
endmodule
```

Pipeline Register Modules for CPU Data Path

Overview

The following modules implement pipeline registers for different stages of a CPU data path: `IF_ID_Regs`, `ID_EX_Regs`, `EX_Mem_Regs`, and `Mem_WB_Regs`.

IF_ID_Regs Module

Inputs

- **clk**: Clock signal.
- **sclr**: Synchronous clear signal.
- **en**: Enable signal.
- **InstrF [31:0]**: Instruction fetched.
- **PCF [31:0]**: Program counter value.
- **PCPlus4F [31:0]**: Program counter plus 4 value.

Outputs

- **InstrD [31:0]**: Instruction decoded.
- **PCD [31:0]**: Program counter value.
- **PCPlus4D [31:0]**: Program counter plus 4 value.

Functionality

This module transfers values from the instruction fetch stage to the instruction decode stage.

ID_EX_Regs Module

Inputs

- **clk**: Clock signal.
- **sclr**: Synchronous clear signal.
- **RegWriteD**: Register write enable signal.
- **MemWriteD**: Memory write enable signal.
- **ALUSrcD**: ALU source select signal.
- **JumpD**: Jump signal.
- **BranchD**: Branch signal.
- **JalrD**: JALR signal.
- **ResultSrcD [1:0]**: Result source select signal.
- **ALUControlD [2:0]**: ALU control signal.

- **func3D [2:0]**: Function code.
- **Rs1D [4:0]**: Source register 1.
- **Rs2D [4:0]**: Source register 2.
- **RdD [4:0]**: Destination register.
- **PCD [31:0]**: Program counter value.
- **ExtImmD [31:0]**: Extended immediate value.
- **PCPlus4D [31:0]**: Program counter plus 4 value.
- **RD1D [31:0]**: Register data 1.
- **RD2D [31:0]**: Register data 2.

Outputs

- **RegWriteE**: Register write enable signal.
- **MemWriteE**: Memory write enable signal.
- **ALUSrcE**: ALU source select signal.
- **JumpE**: Jump signal.
- **BranchE**: Branch signal.
- **JalrE**: JALR signal.
- **ResultSrcE [1:0]**: Result source select signal.
- **ALUControlE [2:0]**: ALU control signal.
- **func3E [2:0]**: Function code.
- **Rs1E [4:0]**: Source register 1.
- **Rs2E [4:0]**: Source register 2.
- **RdE [4:0]**: Destination register.
- **PCE [31:0]**: Program counter value.
- **ExtImmE [31:0]**: Extended immediate value.
- **PCPlus4E [31:0]**: Program counter plus 4 value.
- **RD1E [31:0]**: Register data 1.
- **RD2E [31:0]**: Register data 2.

Functionality

This module transfers values from the instruction decode stage to the execution stage.

EX_Mem_Regs Module

Inputs

- **clk**: Clock signal.
- **RegWriteE**: Register write enable signal.
- **MemWriteE**: Memory write enable signal.
- **ResultSrcE [1:0]**: Result source select signal.
- **RdE [4:0]**: Destination register.
- **ALUResultE [31:0]**: ALU result.
- **WriteDataE [31:0]**: Write data.
- **ExtImmE [31:0]**: Extended immediate value.
- **PCPlus4E [31:0]**: Program counter plus 4 value.

Outputs

- **RegWriteM**: Register write enable signal.
- **MemWriteM**: Memory write enable signal.
- **ResultSrcM [1:0]**: Result source select signal.
- **RdM [4:0]**: Destination register.
- **ALUResultM [31:0]**: ALU result.
- **WriteDataM [31:0]**: Write data.
- **ExtImmM [31:0]**: Extended immediate value.
- **PCPlus4M [31:0]**: Program counter plus 4 value.

Functionality

This module transfers values from the execution stage to the memory stage.

Mem_WB_Regs Module

Inputs

- **clk**: Clock signal.
- **RegWriteM**: Register write enable signal.
- **ResultSrcM [1:0]**: Result source select signal.
- **RdM [4:0]**: Destination register.
- **ALUResultM [31:0]**: ALU result.
- **ReadDataM [31:0]**: Read data.
- **ExtImmM [31:0]**: Extended immediate value.
- **PCPlus4M [31:0]**: Program counter plus 4 value.

Outputs

- **RegWriteW**: Register write enable signal.
- **ResultSrcW [1:0]**: Result source select signal.
- **RdW [4:0]**: Destination register.
- **ALUResultW [31:0]**: ALU result.
- **ReadDataW [31:0]**: Read data.
- **ExtImmW [31:0]**: Extended immediate value.
- **PCPlus4W [31:0]**: Program counter plus 4 value.

Functionality

This module transfers values from the memory stage to the write-back stage.

Register File:

```
module RegisterFile(input clk, we, input [4:0] A1, A2, A3, input[31:0] WD3, output[31:0] RD1, RD2);
    reg[31:0] regFile [0:31];
    initial begin
        regFile[0] = 32'b0;
    end
    assign RD1 = regFile[A1];
    assign RD2 = regFile[A2];
    always@(negedge clk)begin
```

```
        if(we == 1 && A3 != 32'b0)
            regFile[A3] <= WD3;
        else
            regFile[A3] <= regFile[A3];
    end
endmodule
```

Register File Module

Overview

The `RegisterFile` module is a multi-port memory structure used to store and access register values in a RISC-V processor. It supports simultaneous reads and a single write per clock cycle.

Module Interface

Inputs

- **clk**: The clock signal.
- **regWrite**: The register write enable signal.
- **readRegister1 [`BITS(WordCount)-1:0]**: The address of the first register to read.
- **readRegister2 [`BITS(WordCount)-1:0]**: The address of the second register to read.
- **writeRegister [`BITS(WordCount)-1:0]**: The address of the register to write.
- **writeData [WordLen-1:0]**: The data to write to the register.

Outputs

- **readData1 [WordLen-1:0]**: The data read from the first register.
- **readData2 [WordLen-1:0]**: The data read from the second register.

Parameters

- **WordLen**: The bit-width of each register. Default is 32.
- **WordCount**: The number of registers. Default is 32.

Operation

The `RegisterFile` module reads data from two registers and writes data to one register. Register 0 is hardwired to 0. Writes occur on the negative edge of the clock if the `regWrite` signal is enabled. Reads also occur on the negative edge of the clock.

```
module SumModule(input [31:0] num1, num2, output signed [31:0] result);
    assign result = num1 + num2;
endmodule
```

Adder Module for 32-bit Addition

Overview

The `SumModule` performs a 32-bit addition operation on two input operands and produces a 32-bit result.

Module Interface

Inputs

- **num1 [31:0]**: The first 32-bit operand for the addition.
- **num2 [31:0]**: The second 32-bit operand for the addition.

Outputs

- **result [31:0]**: The 32-bit signed result of the addition operation.

Operation

The `SumModule` performs a simple addition operation on the two input operands, `num1` and `num2`, and assigns the sum to the output `result`.

Code

```
module SumModule(input [31:0] num1, num2, output signed [31:0] result);
    assign result = num1 + num2;
endmodule
```

TestBench :

```
`timescale 1ns/1ns
module TBPipe();
    reg clk = 1'b0;
    TopLevelPipe P(clk);

    always #20 clk =~ clk;

    initial begin
        #20000
        $stop;
    end
endmodule
```

TopLevelPipeline:

```
module TopLevelPipe(input clk);
    wire MemWriteD, ALUSrcD, RegWriteD, JumpD, BranchD, JalrD, RegWriteM, RegWriteW, JalrE, JumpE, BranchE,
    wire[1:0] ResultSrcD, ResultSrcE, PCSrcE, ForwardAE, ForwardBE, ResultSrcM, ResultSrcW;
    wire[2:0] func3D, func3E, ALUControlD, ImmSrcD;
    wire[4:0] Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW;
    wire[6:0] func7, op;
    ControlUnit CTRL(func3D, func7, op, MemWriteD, ALUSrcD, RegWriteD, JumpD, BranchD, JalrD, ResultSrcD, A
    HazardUnit  HU(RegWriteM, RegWriteW, ResultSrcE, PCSrcE, ResultSrcM, ResultSrcW, Rs1D, Rs2D, Rs1E, Rs2E
    ControlDecoder      DC(JalrE, JumpE, BranchE, ZeroE, ALUResult0, func3E, PCSrcE);
    DataPath    DP(clk, MemWriteD, ALUSrcD, RegWriteD, JumpD, BranchD, JalrD, StallF, StallD, FlushD, Flush
        ResultSrcD, ForwardAE, ForwardBE, PCSrcE, ALUControlD, ImmSrcD, BranchE, JumpE, JalrE, ZeroE, ALURe
        ResultSrcE, func3D, func3E, func7, op, Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW, ResultSrcM, ResultSrc
endmodule
```

TopLevelPipeline Module for CPU Pipeline Integration

Overview

The `TopLevelPipeline` module integrates various control, hazard detection, and data path modules to implement a complete CPU pipeline.

Module Interface

Inputs

- **clk**: Clock signal.

Internal Signals

- **Control Signals**: Signals for memory write, ALU source, register write, jump, branch, and JALR operations.
  - `memWriteD`, `aluSrcD`, `regWriteD`, `jumpD`, `branchD`, `jalrD`
  - `regWriteM`, `regWriteW`, `jalrE`, `jumpE`, `branchE`, `zeroE`, `aluResult0`
- **Hazard Signals**: Signals for stalling, flushing, and forwarding.
  - `stallF`, `stallD`, `flushD`, `flushE`
  - `forwardAE`, `forwardBE`
- **Result Source Signals**: Signals for selecting the result source.
  - `resultSrcD`, `resultSrcE`, `pcSrcE`
  - `resultSrcM`, `resultSrcW`
- **Function and Opcode Signals**: Signals for function codes and opcode.
  - `func3D`, `func3E`, `aluControlD`, `immSrcD`
  - `rs1D`, `rs2D`, `rs1E`, `rs2E`, `rdE`, `rdM`, `rdW`
  - `func7`, `opcode`

Components

Control Unit

The `ControlUnit` module generates control signals based on the instruction's function codes and opcode.

Hazard Unit

The `HazardUnit` module detects and handles pipeline hazards by generating control signals for stalling, flushing, and forwarding.

Control Decoder

The `ControlDecoder` module determines the source for the program counter (`pcSrcE`) based on various control signals and the function code.

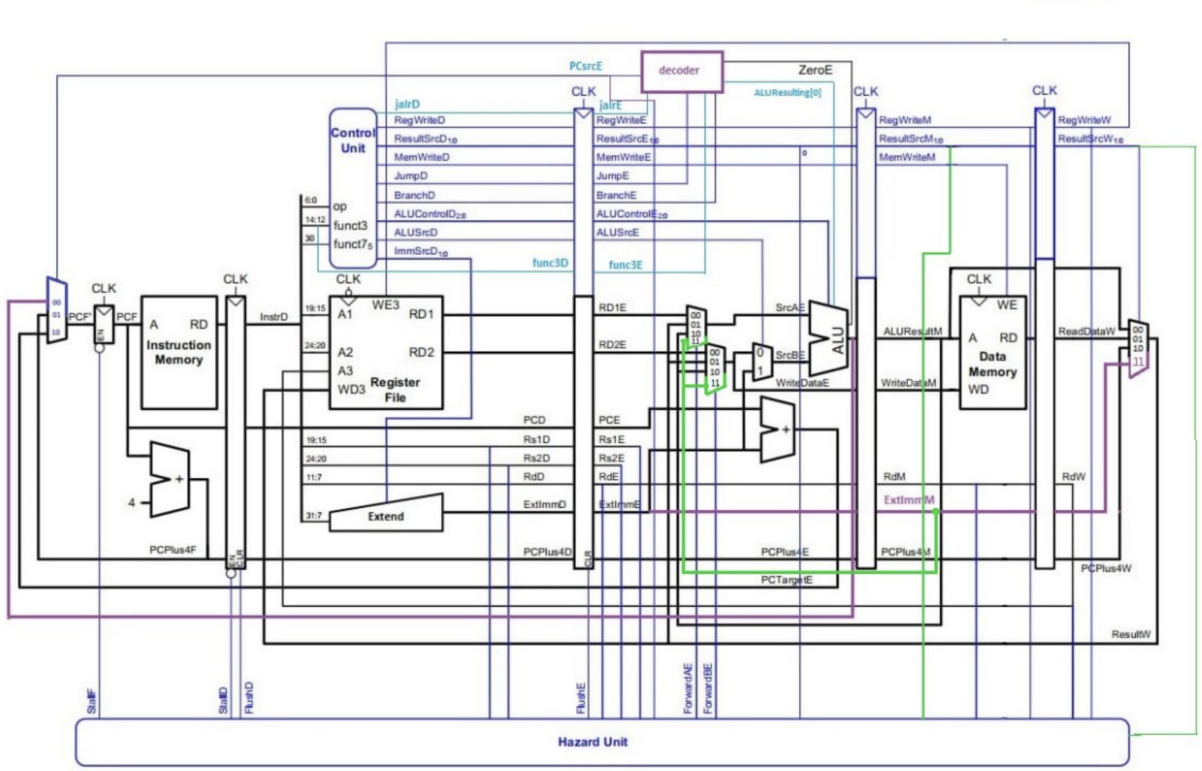
Data Path

The `DataPath` module handles the flow of data through the CPU pipeline stages: instruction fetch, decode, execute, memory access, and write-back.

Controler Unit Signall :

		PCSrc	ResultSrc	MemWrite	ALUControl	ALUSrc	ImmSrc	RegWrite
R-Type	add	00	00	0	010	0	XXX	1
	sub	00	00	0	110	0	XXX	1
	and	00	00	0	000	0	XXX	1
	or	00	00	0	001	0	XXX	1
	slt	00	00	0	111	0	XXX	1
I-Type	lw	00	01	0	010	1	000	1
	addi	00	00	0	010	1	000	1
	xori	00	00	0	011	1	000	1
	ori	00	00	0	001	1	000	1
	slti	00	00	0	111	1	000	1
	jalr	10	10	0	010	1	000	1
S-Type	sw	00	XX	1	010	1	001	0
J-Type	jal	01	10	0	XXX	X	010	1
B-Type	beq	zero ? 01 : 00	XX	0	110	0	011	0
	bne	zero ? 00 : 01	XX	0	110	0	011	0
	blt	aluresult[0] ? 01 : 00	XX	0	111	0	011	0
	bge	aluresult[0] ? 00 : 01	XX	0	111	0	011	0
U-Type	lui	00	11	0	XXX	X	100	1

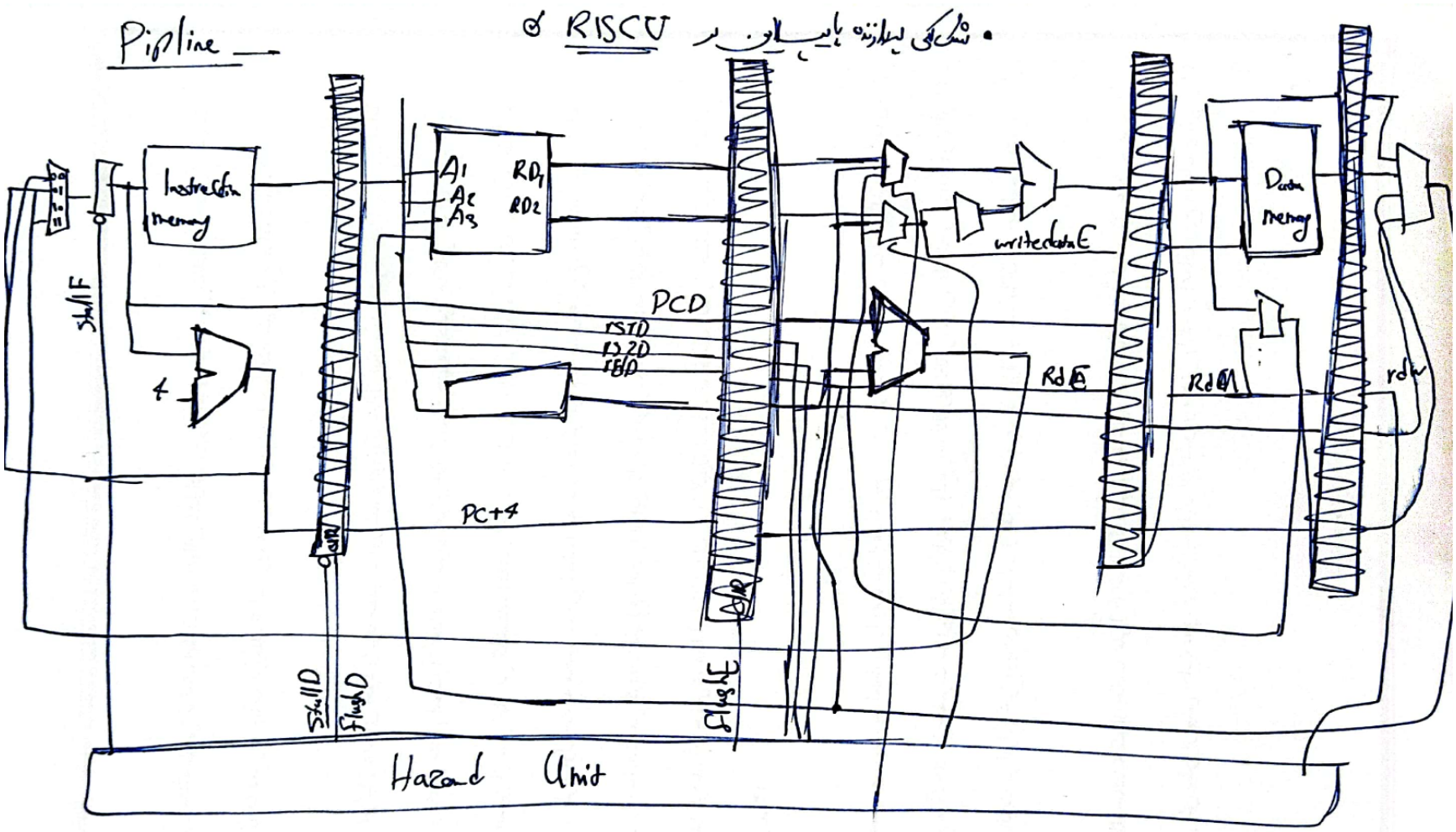
DataPath :



Description :

Pipline —

• شکل کلی پیاده‌سازی پایپ‌لاین در RISCV



- به پایه لوله ای :

- ابتدا توضیحاتی راجع به این پایه لوله ای و نحوه کارکردنش بدهم:

این پایه لوله ای به این گونه کار میکند که به جای اجرای دستورات با پایه لوله ای Single Cycle مانند multi cycle

یعنی هر دستور به چند مرحله تقسیم میشود.

- چون کنترل دقیقا شبیه به پایه لوله ای Single Cycle یا multi cycle می باشد:

- به ترتیب اینتر هایی که وجود دارند به صورت زیر هستند:

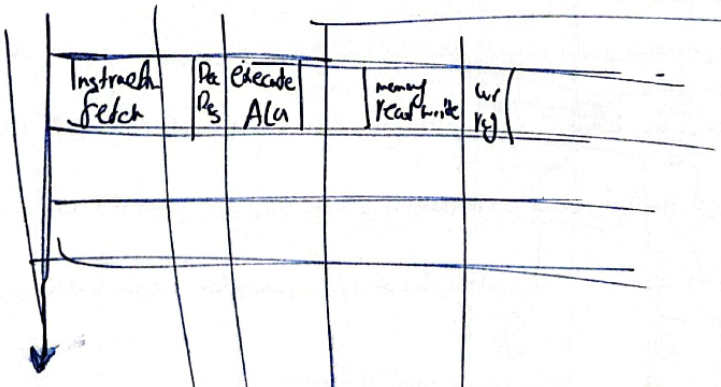
-fetch
-Decode
-Execution
-memory
-writeback

- حالا میخواهم تک تک این مراحل را توضیح دهم:

که حال مطابق با نیاز دستور از دستورات پایه لوله ای استفاده میکند.

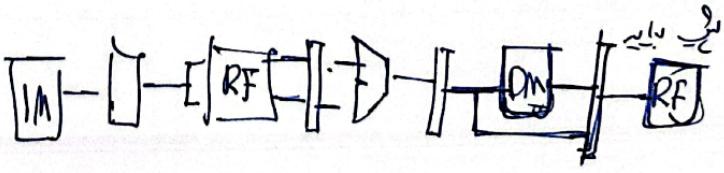
از این دستورات را استفاده میکند و تا جایی که لازم است عمل میکند.

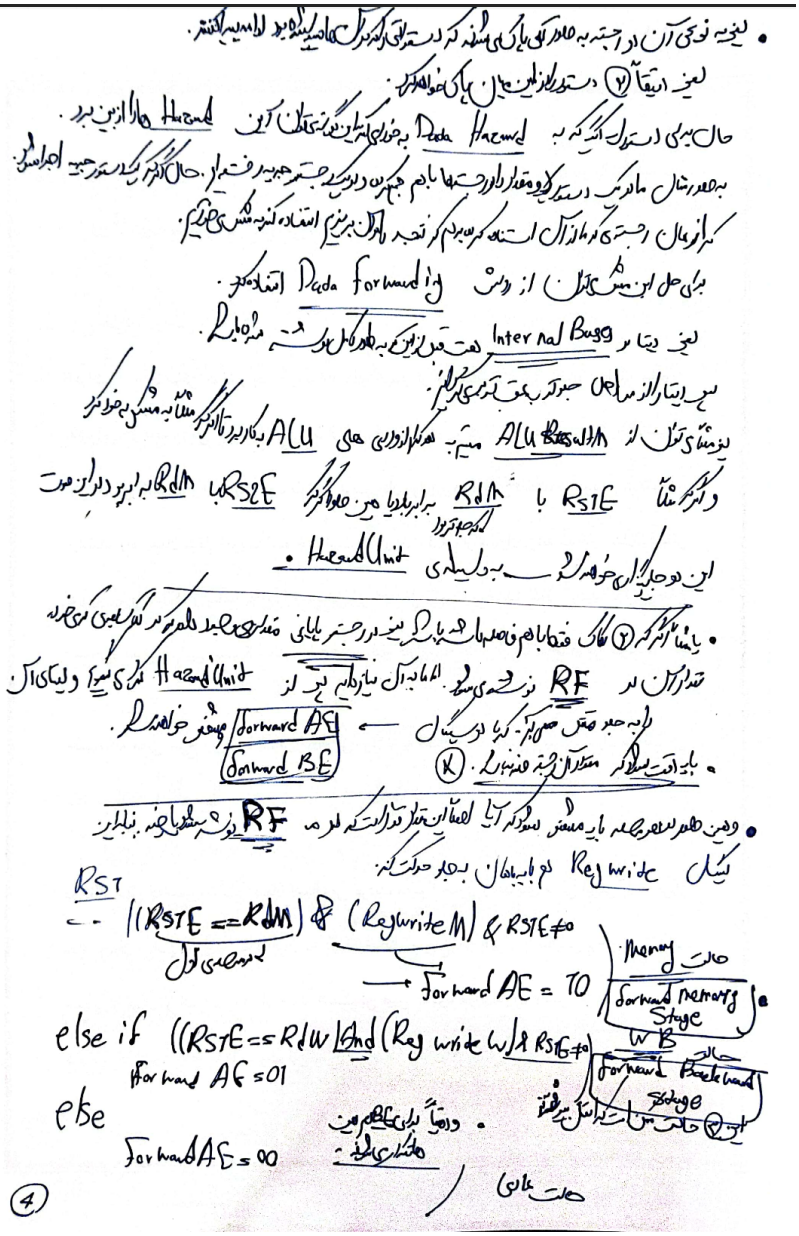
- از این نوع نوشتن هم نقل به نقل میکنند:



در این پایه لوله ای Register file آن به گونه ای طراحی شده است که در نیمه اول کلاک باید بتوانیم دستورات را بنویسیم و در نیمه دوم کلاک باید بتوانیم دستورات را بخوانیم.

اگر این ترتیب را رعایت نکنیم از Hazard ها جلوگیری خواهیم کرد که بعدا در مورد آن صحبت میکنیم.





• حال اگر مسئله Data Hazard را به همین صورت که برای حافظه دیتا یا از حافظه Data memory لود کنیم
مثلی این کار باید حتماً یک بار یک دستور متوقف کنیم چطور یک دستور نمان عقب خواهد کرد.

• به نوعی مقادیر موجود در PC و رجیستر لوکال enable نشده و تغییری نکنند این کار دوباره باعث مقادیر
سابق شده و در چرخه بعدی قرار گیرند. سپس آن دستوری که وارد memory خالی در ابتدای مسیر لوله
و در سیکل بعدی در آن RF نوشته خواهد شد.
و نام متغیرهای را به مانند بقیه سیگنال‌ها می‌آوریم سپس کدهای زیر به این صورت می‌باشد:
حتماً بایستی Result SrcE = 1 باشد.

lwStall = ((Rs1D == RdE) or (Rs2D == RdE)) AND ResultSrcE0

و توان در مدل نمای فیزیکی shiftE را فعال کرد که مقادیر موجود در آن یک سیکل و به مسیر بعدی برود.

StallF = StallD = FlushE = lwStall ← مقادیر باقیمانده را داریم

این را تا بسیار فعال کنید.

• Control Hazard :

• همان طور که قبلاً توضیح دادیم این نوع از Hazard را به سه دسته Beq و نوع Branch و JAL و JALR تقسیم کردیم
که در پیش‌بینی‌های ما بخش EXE بررسی خواهد شد.

• Branch misprediction Penalty : ← ② اسم دیگر آن را به این صورت نیز می‌نامند.
• به تعداد دستوراتی که باید خالی شوند زمانی که
برنچ اتفاق می‌افتد گفته می‌شود.

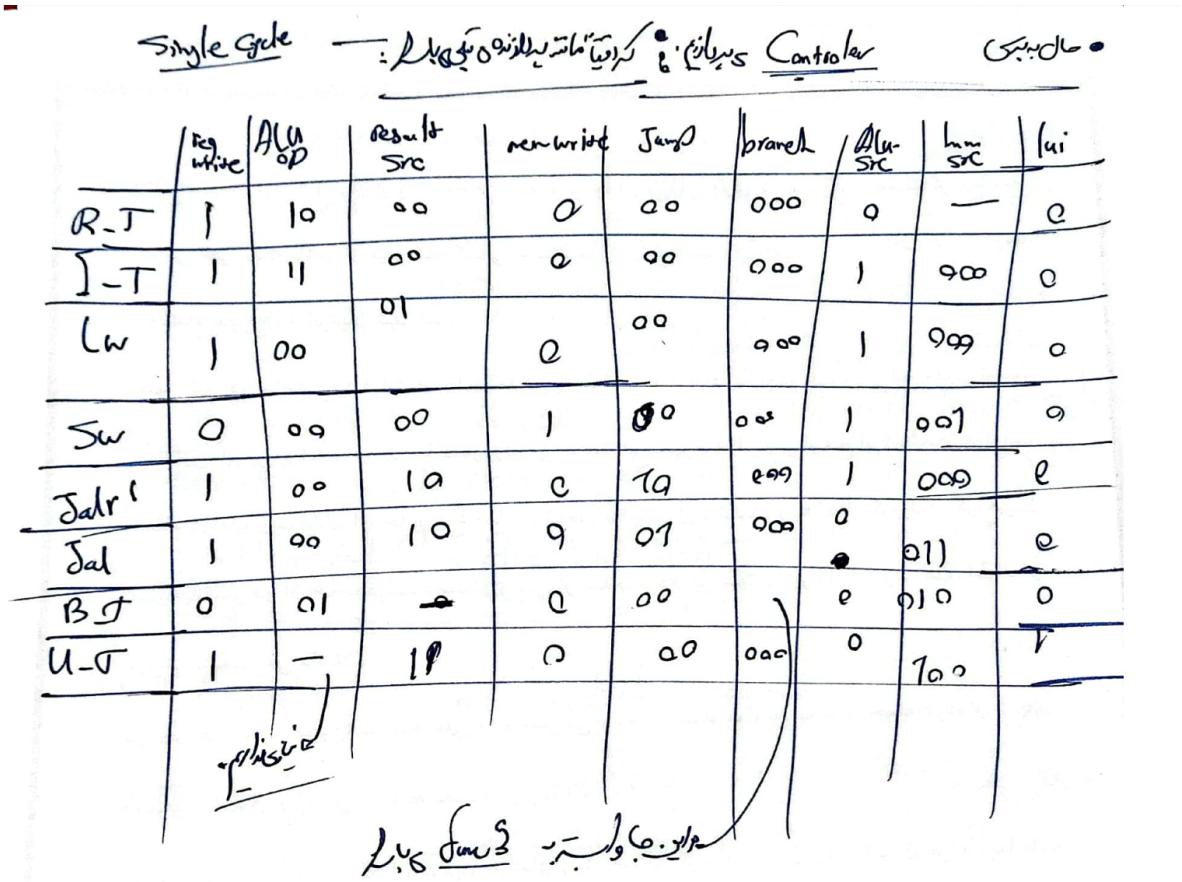
در Execution Stage و Decode Stage باید پاک شوند.

• FlushD = PCSrcE ← مقدار PC جدید بر PC فرستاده شود.

• FlushE = lwStall or PCSrcE

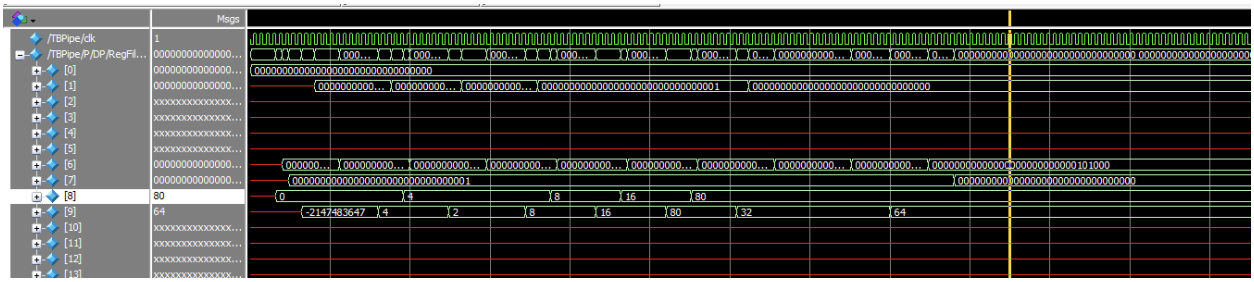
• حال این دو تا تغییر را می‌توانیم ذخیره کنیم.

⑤



Test Bench results:

we can see that in 8 register we put largest value :



New Testbench for get min of that :

this is new testbench : located in inst.mem :

```
03
24
00
00
13
03
40
00
93
23
83
02
63
8e
03
00
83
24
03
00
b3
a0
84
00
63
84
00
00
33
04
90
00
13
03
43
00
6f
f0
5f
fe
37
35
00
00
ef
05
20
00
```

```
03240000
13034000
93238302
```

```
638e0300
83240300
b3a08400
63840000
33049000
13034300
6ff05ffe
37350000
ef052000
```

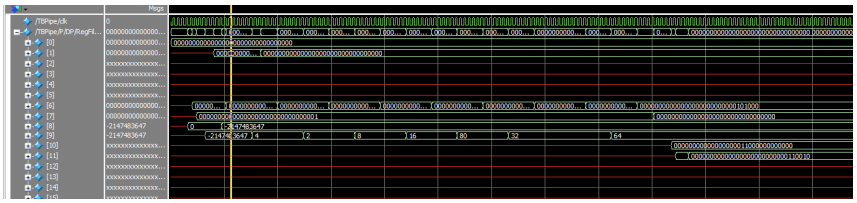
Now, let's convert each one into RISC-V assembly:

1. `03240000` → `lw x8, 0(x0)`
2. `13034000` → `addi x6, x0, 64`
3. `93238302` → `addi x7, x7, 40`
4. `638e0300` → `beq x0, x7, offset` (offset to be determined based on the position of the instruction)
5. `83240300` → `lw x8, 0(x6)`
6. `b3a08400` → `add x9, x9, x8`
7. `63840000` → `beq x8, x0, offset` (offset to be determined based on the position of the instruction)
8. `33049000` → `add x8, x2, x0`
9. `13034300` → `addi x6, x8, 12`
10. `6ff05ffe` → `jal x0, offset` (offset to be determined based on the position of the instruction)
11. `37350000` → `lui x10, 0x35`
12. `ef052000` → `jal x11, 32`

Let's assemble this into RISC-V assembly code:

```
    lw x8, 0(x0)        # 03240000
    addi x6, x0, 64     # 13034000
    addi x7, x7, 40     # 93238302
    beq x0, x7, label1  # 638e0300
    lw x8, 0(x6)        # 83240300
    add x9, x9, x8      # b3a08400
    beq x8, x0, label2  # 63840000
    add x8, x2, x0      # 33049000
    addi x6, x8, 12     # 13034300
    jal x0, label3      # 6ff05ffe
    lui x10, 0x35       # 37350000
    jal x11, 32         # ef052000
```

this programm finds min of that 10 elements in array



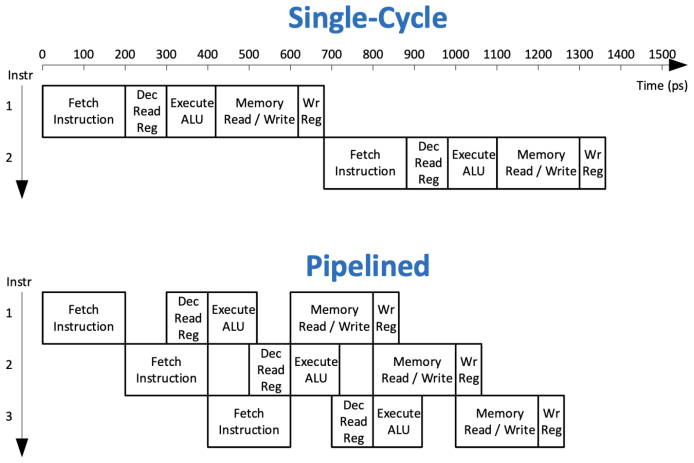
Overview of RISC_V :

Pipelined RISC-V Processor

Pipelined RISC-V Processor

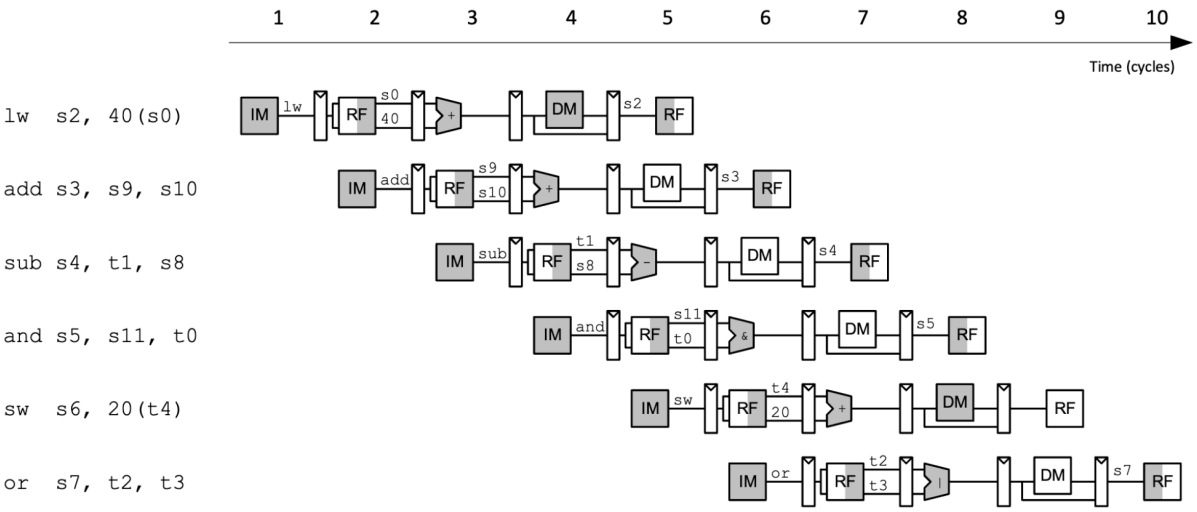
- **Temporal parallelism**
- Divide single-cycle processor into **5 stages**:
  - Fetch
  - Decode
  - Execute
  - Memory
  - Writeback
- Add **pipeline registers** between stages

Single-Cycle vs. Pipelined Processor



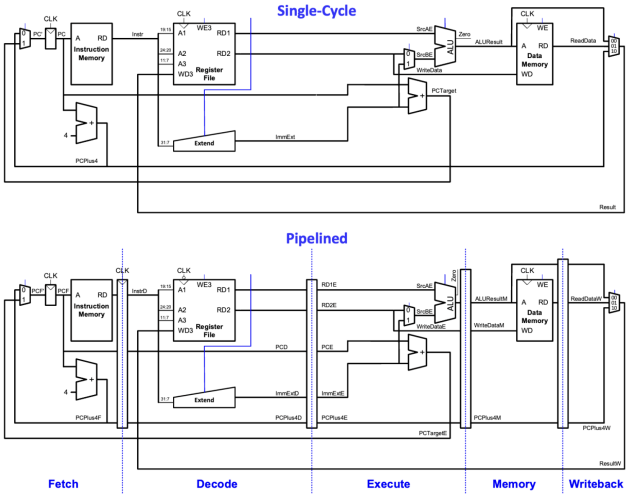
3

Pipelined Processor Abstraction



4

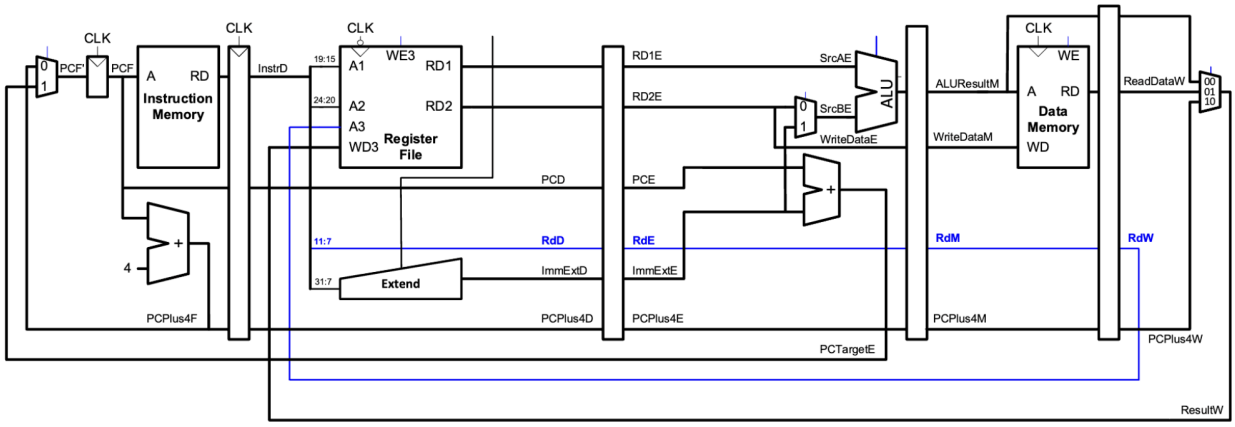
Single-Cycle & Pipelined Datapaths



Signals in Pipelined Processor are appended with first letter of stage (i.e., PCF, PCD, PCE).

5

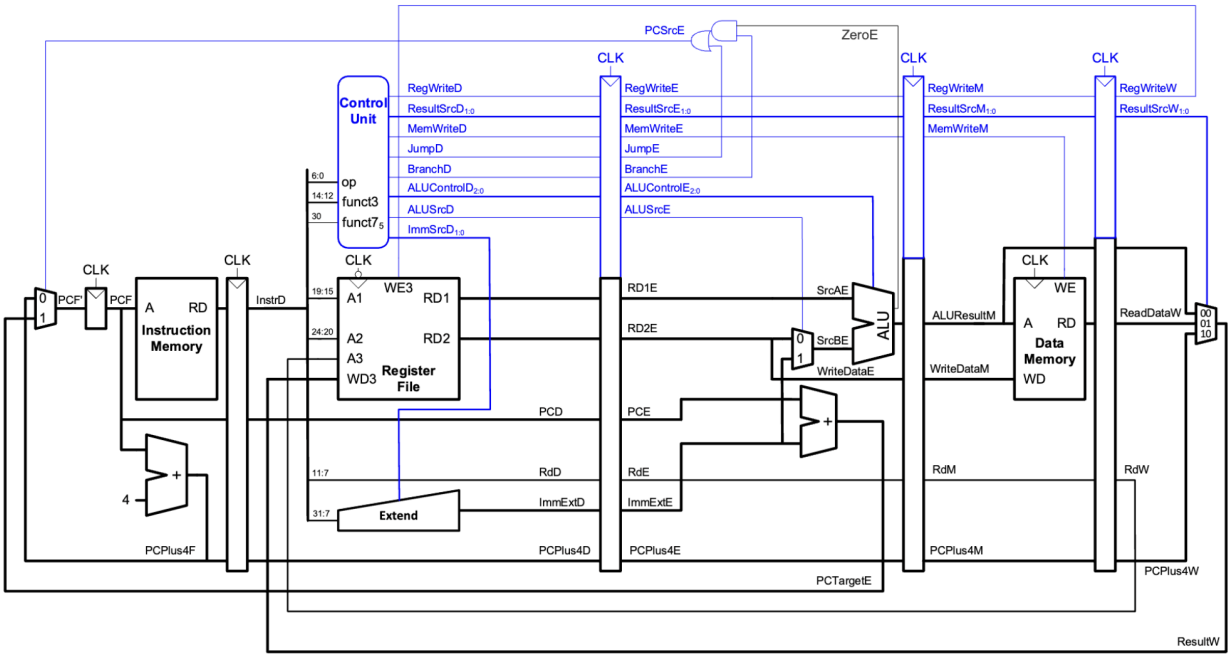
Corrected Pipelined Datapath



- ***Rd*** must arrive at same time as ***Result***
- Register file written on **falling edge** of *CLK*

6

Pipelined Processor with Control



- **Same control unit** as single-cycle processor
- **Control signals travel with** the instruction (drop off when used)

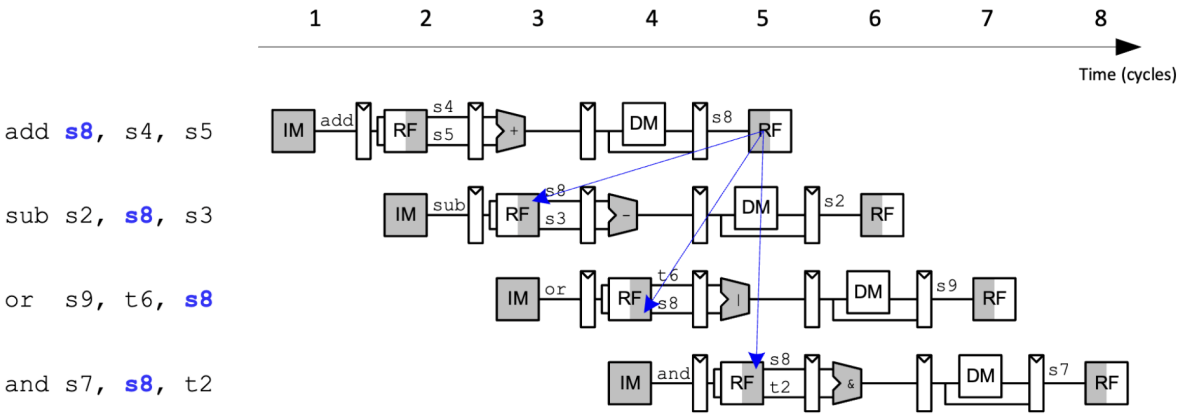
7

Pipelined Processor Hazards

Pipelined Hazards

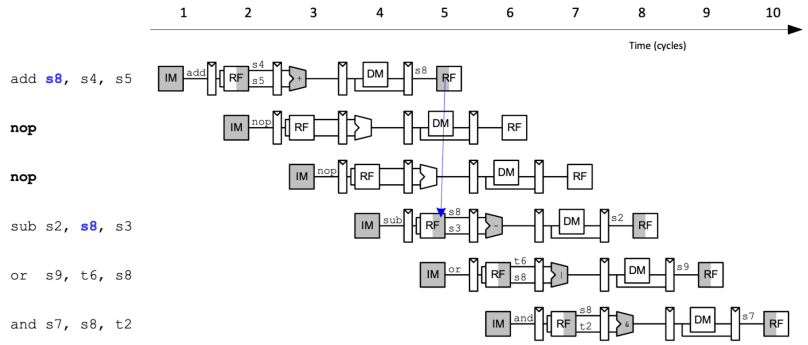
- When an instruction depends on result from instruction that hasn't completed
- Types:
  - **Data hazard:** register value not yet written back to register file
  - **Control hazard:** next instruction not decided yet (caused by branch)

Data Hazard



Handling Data Hazards

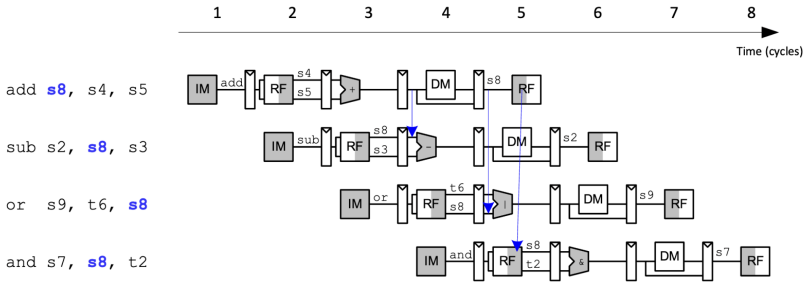
- **Insert** enough **nops** for result to be ready
- Or move independent useful instructions forward



12

Data Forwarding

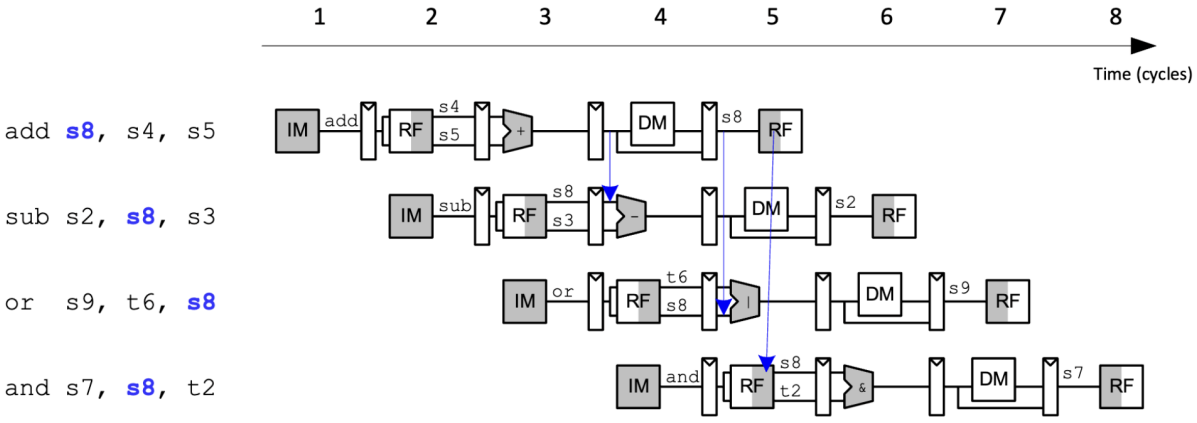
- Data is **available on internal busses** before it is written back to the register file (RF).
- **Forward data** from internal busses **to Execute stage**.



13

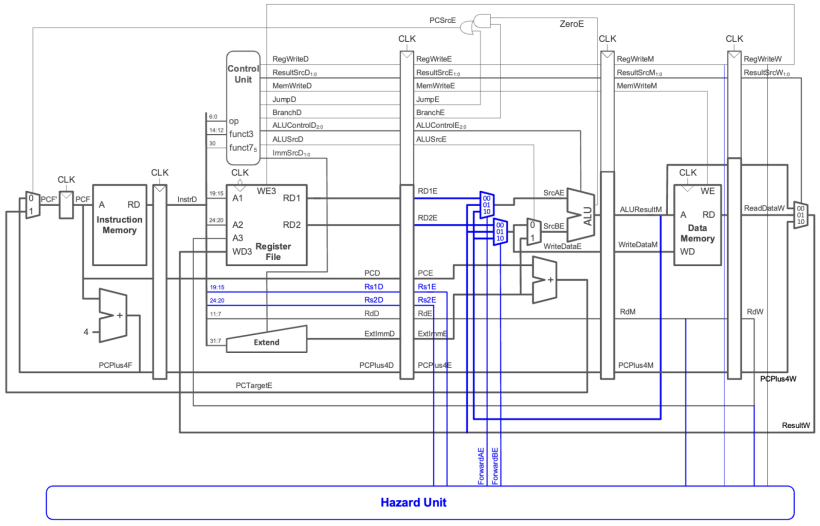
Data Forwarding

- Check if source register **in Execute stage** **matches** destination register of instruction **in Memory or Writeback stage**.
- If so, forward result.



14

Data Forwarding: Hazard Unit



15

Data Forwarding

- **Case 1:** **Execute** stage *Rs1* or *Rs2* matches **Memory** stage *Rd*?
  Forward from Memory stage
- **Case 2:** **Execute** stage *Rs1* or *Rs2* matches **Writeback** stage *Rd*?
  Forward from Writeback stage
- **Case 3:** Otherwise use value read from register file (as usual)

Equations for *Rs1*:

```
if      ((Rs1E == RdM) AND RegWriteM)          // Case 1
            ForwardAE = 10
else if ((Rs1E == RdW) AND RegWriteW)          // Case 2
            ForwardAE = 01
else        ForwardAE = 00                     // Case 3
```

***ForwardBE** equations are similar (replace **Rs1E** with **Rs2E**)*

16

Data Forwarding

- **Case 1:** **Execute** stage *Rs1* or *Rs2* matches **Memory** stage *Rd*?
  Forward from Memory stage
- **Case 2:** **Execute** stage *Rs1* or *Rs2* matches **Writeback** stage *Rd*?
  Forward from Writeback stage
- **Case 3:** Otherwise use value read from register file (as usual)

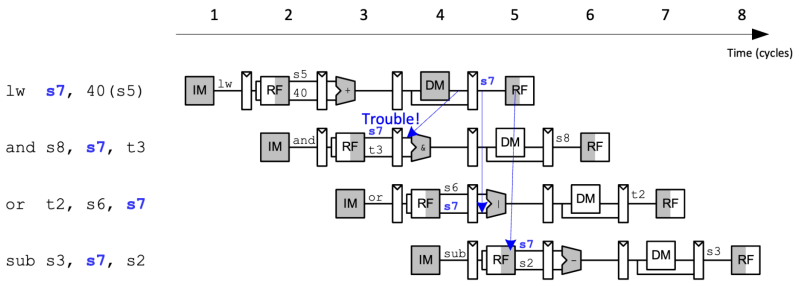
Equations for *Rs1*:

```
if      ((Rs1E == RdM) AND RegWriteM) AND (Rs1E != 0) // Case 1
            ForwardAE = 10
else if ((Rs1E == RdW) AND RegWriteW) AND (Rs1E != 0) // Case 2
            ForwardAE = 01
else        ForwardAE = 00                                  // Case 3
```

***ForwardBE** equations are similar (replace **Rs1E** with **Rs2E**)*

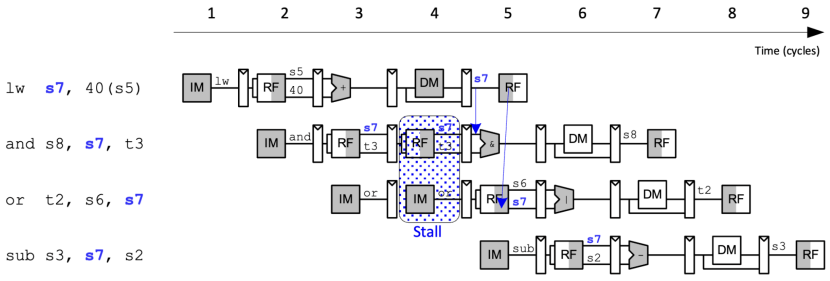
17

Data Hazard due to `lw` Dependency



18

Stalling to solve `lw` Data Dependency



19

Stalling Logic

- Is either **source register in the Decode stage** the same as the **destination register in the Execute stage**?

  AND

- Is the instruction in the **Execute stage a `lw`**?

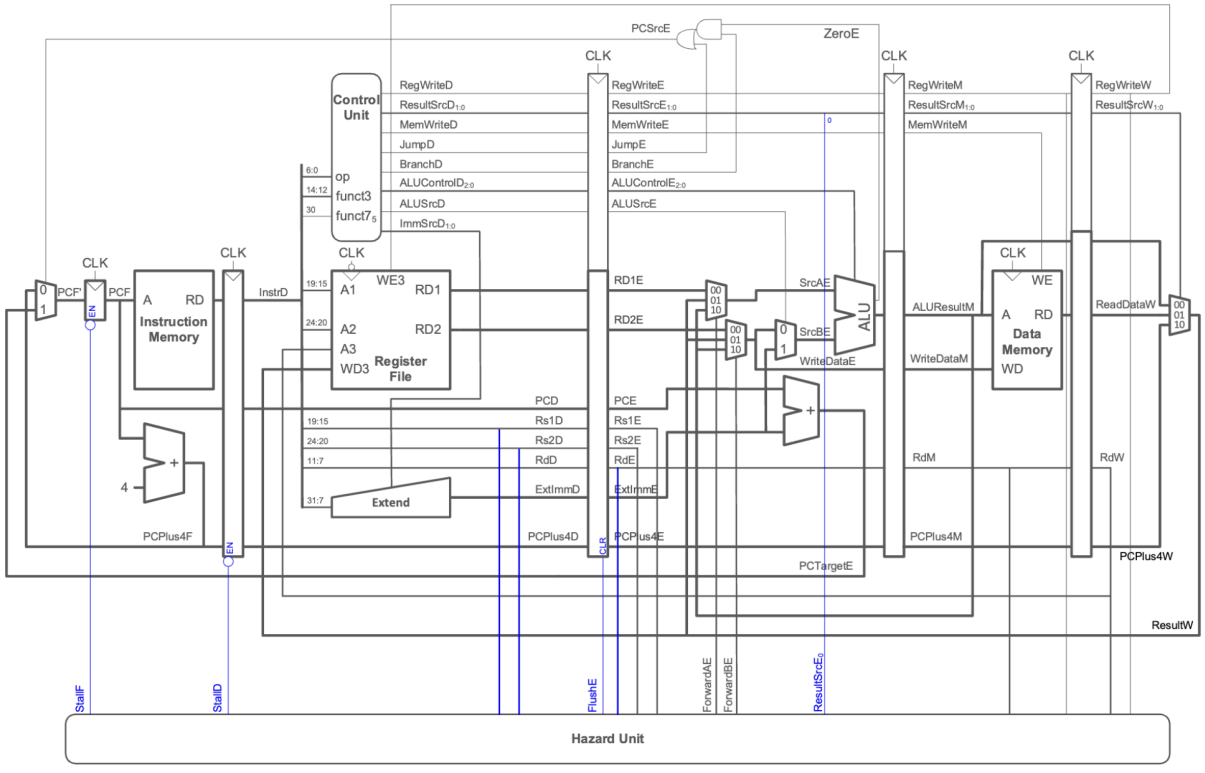
lwStall = ((***Rs1D*** == ***RdE***) OR (***Rs2D*** == ***RdE***)) **AND** $ResultSrcE_0$

StallF = ***StallD*** = ***FlushE*** = ***lwStall***

(Stall the Fetch and Decode stages, and flush the Execute stage.)

20

Stalling Hardware



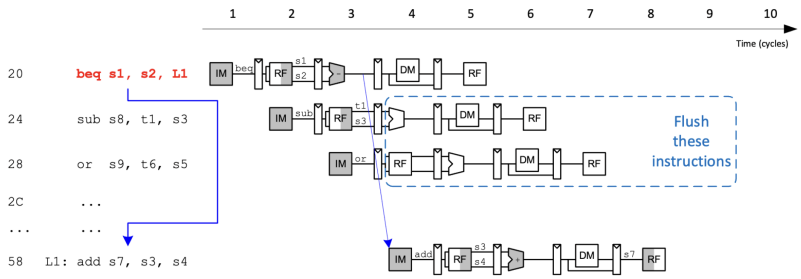
21

Pipelined Processor Control Hazards

Control Hazards

- **`beq`:**
  - Branch **not determined until the Execute stage** of pipeline
  - **Instructions** after branch **fetched** before branch occurs
  - These **2 instructions must be flushed** if branch happens

Control Hazards



Branch misprediction penalty:

The number of instructions flushed when a branch is taken (in this case, 2 instructions)

Control Hazards: Flushing Logic

- If branch is taken in execute stage, need to flush the instructions in the Fetch and Decode stages
  - Do this by clearing Decode and Execute Pipeline registers using *FlushD* and *FlushE*
- **Equations:**

  FlushD = PCSrcE

  FlushE = lwStall OR *PCSrcE*