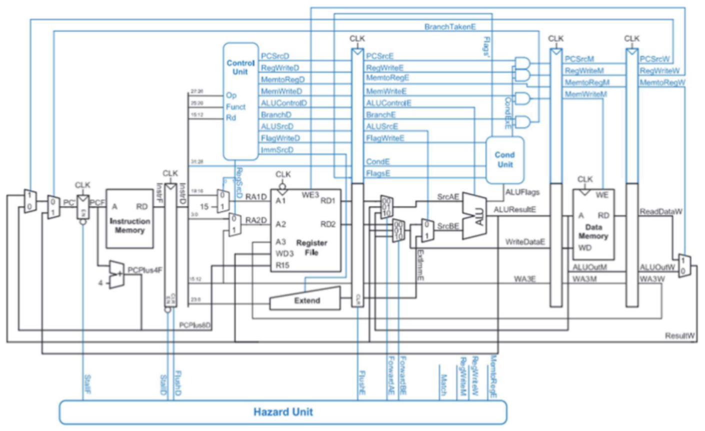
Chenhan Dai, Junchao Zhou
EE469
May 5, 2023
Lab 3 Report

Procedure
This lab involved one main task: design and simulate a simple pipelined processor. We need to add pipelining to our lab2 processor and resolve the issues that come with the performance enhancement (Figure 1). After building the single processor, we then load a test program to confirm the system works.



`Figure 1: ARM 5 Stage Pipelined Processor`

Figure 1: ARM 5 Stage Pipelined Processor

Datapath
We firstly added four pipeline registers to separate the datapath into five stages (Figure 2).

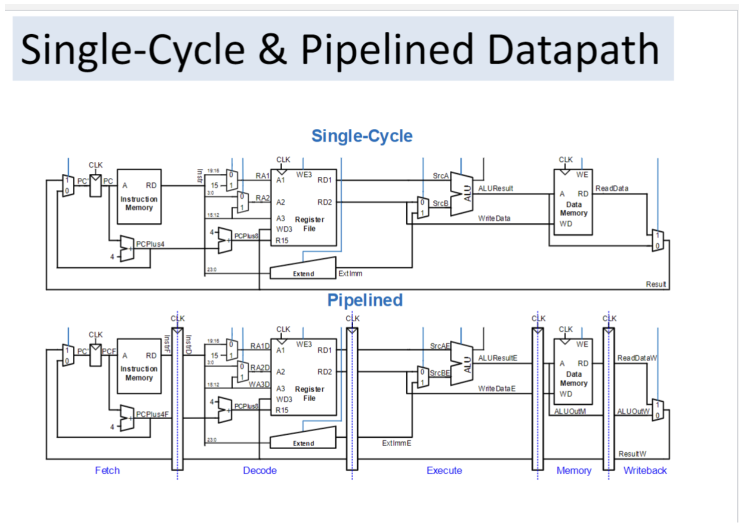
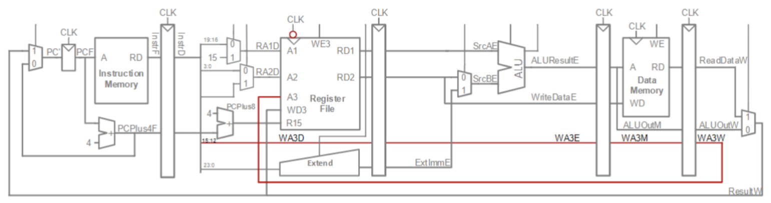


Figure 2: Datapaths of Single-cycle and Pipelined

After that, we corrected the datapath by making the WA3 signal pipelined along through the Execution, Memory and Writeback stages (Figure 3).

Corrected Pipelined Datapath

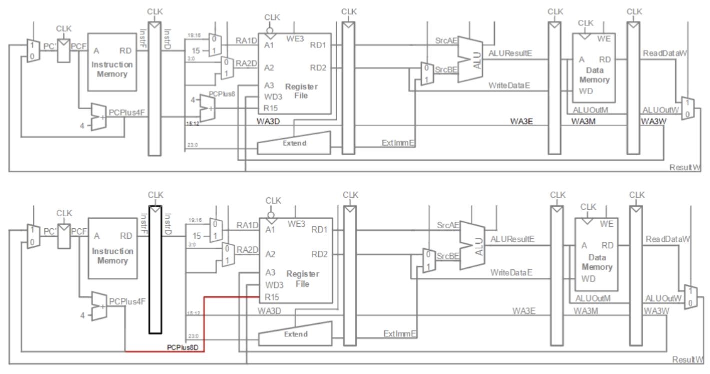


- ***WA3* must arrive at same time as *Result***
- **Register file written on falling edge of *CLK***

Figure 3: Corrected pipelined datapath

Then, we optimized the PC logic by eliminating a register and adder (Figure 4).

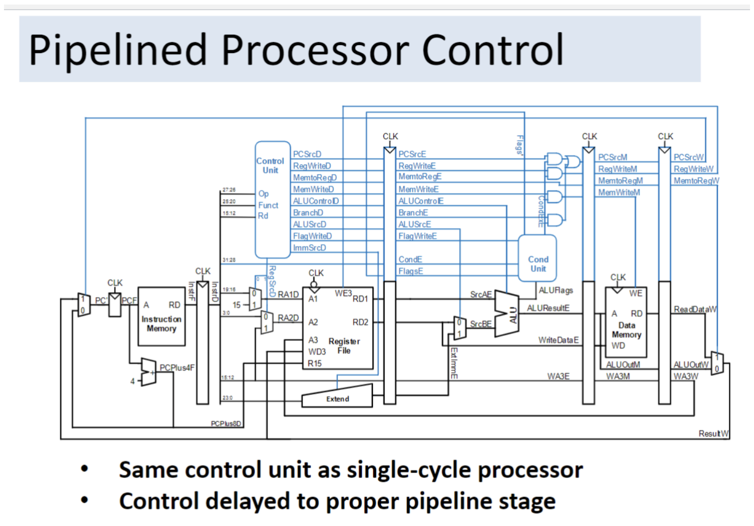
Optimized Pipelined Datapath



Remove adder by using *PCPlus4F* after *PC* has been updated to *PC*+4

Figure 4: Optimized PC logic eliminating a register and adder

Control
Then we make the control signals pipelined along with the data in order to make them remain synchronized with the instructions (Figure 5)..

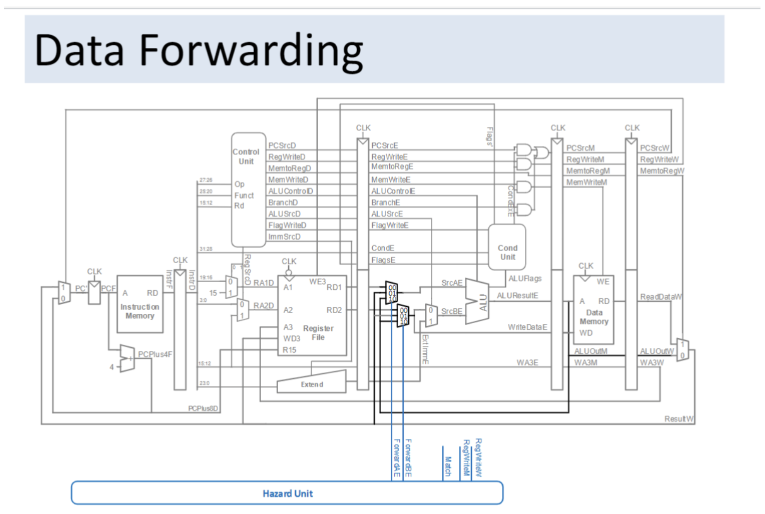


Pipelined Processor Control

- Same control unit as single-cycle processor
- Control delayed to proper pipeline stage

Figure 5: **Pipelined processor with control**

Because forwarding is necessary when an instruction in the Execute stage has a source register matching the destination register of an instruction in the Memory or Writeback stage, we modified the pipelined processor to support forwarding (Figure 6).



Data Forwarding

Figure 6:Pipelined processor with forwarding to solve hazards

And we added the stalls in order to let LDR work properly if it is an LDR and its destination register matches either source operand of the instruction in the Decode Stage (Figure 7).

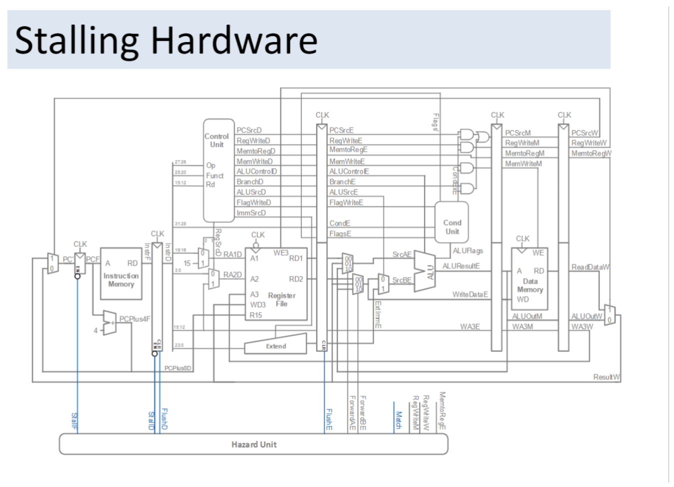


Figure 7: Pipelined processor with stalls to solve LDR data hazard

In the end, we modified the pipelined processor to make sure moving the branch decision earlier and handle control hazards (Figure 8).

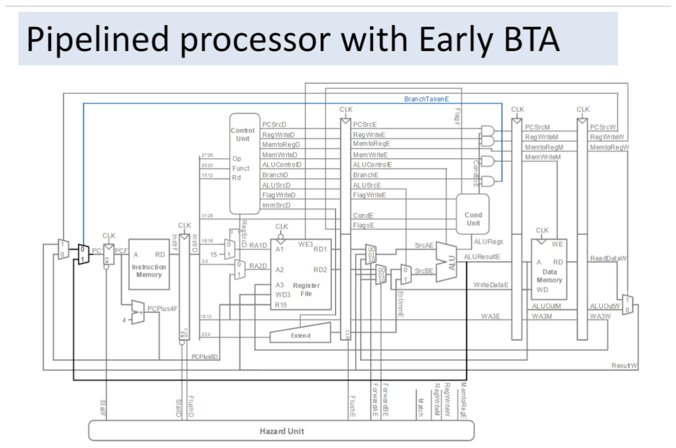


Figure 8:Pipelined processor handling branch control hazard

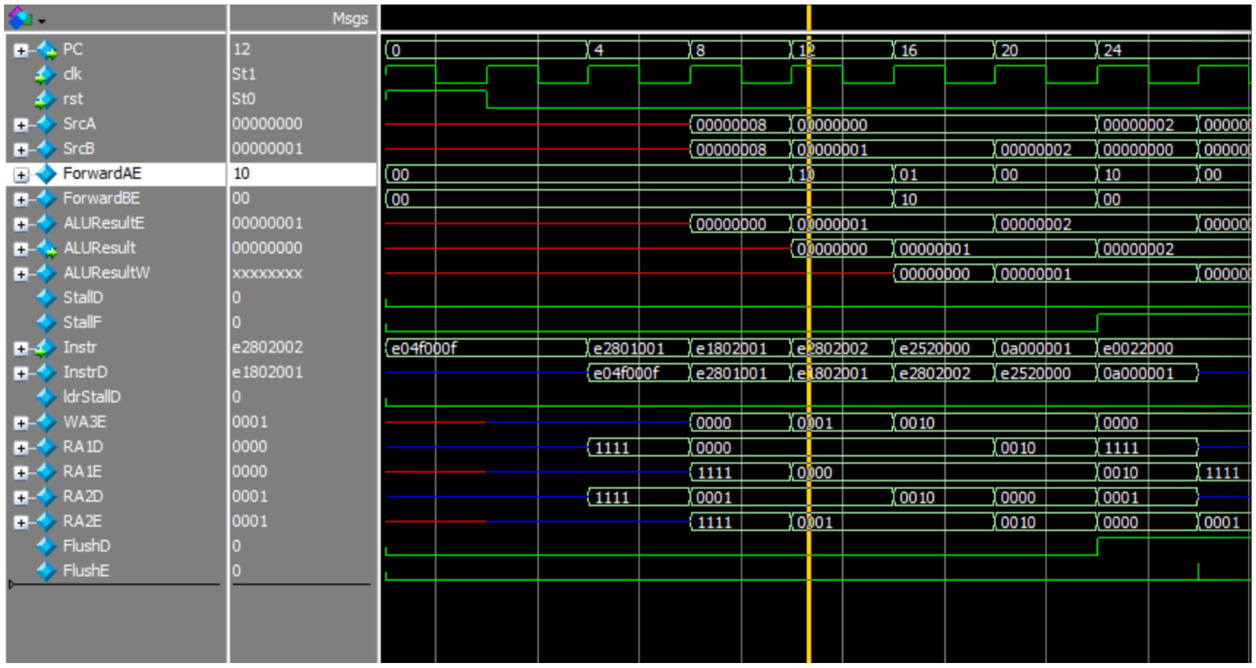
Task:
After completing the design, we use the following program to verify that our system resolved all the potential hazards: data hazard and control hazard.

```
Main      SUB  R0 R15 R15
          ADD  R1 R0 #1
          ORR  R2 R0 R1
          ADD  R2 R0 #2
          SUBS R0 R2 #0
          BEQ  TAG1
          AND  R2 R2 R0
          AND  R1 R2 R0
TAG1      ADD  R9 R1 R0
          STR  R9 [R0, #9]
          LDR  R3 [R0, #9]
          AND  R2 R3 R2
```

Result:

i. An example of forwarding from the memory stage to the execute stage.
In the first instruction, the result of the subtraction operation (R15 - R15) is stored in R0 in the memory stage. In the second instruction, R0 is used as an operand in the execution stage to perform subtraction (R0 + #1). In this case, forwarding from the memory stage to the execute stage is used to provide the value of R0 to the execute stage without waiting for it to be written back to the register file.

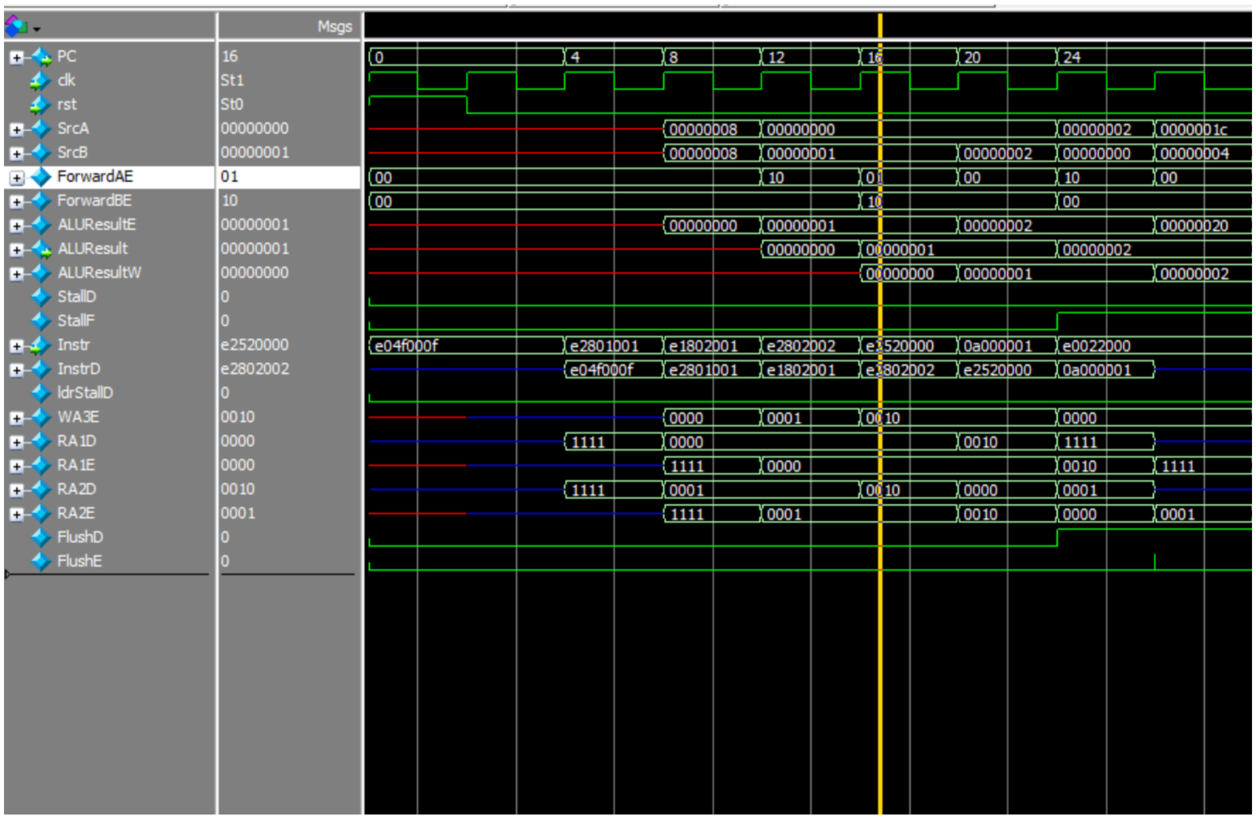
From the following waveform, we could see that at fifth cycle, the ALUResult is been stored in the R0 in memory stage for the first instruction, and as the source R0 is been called as one source in the execution stage, the forward AE becomes 10 to let the sourceA read the ALUResult directly from the R0 in the memory stage.



ii. An example of forwarding from the writeback stage to the execute stage.

In the first instruction, the result of the subtraction operation (R15 - R15) is stored in R0 in the writeback stage. In the third instruction, R0 is used as an operand in the execution stage to perform ORR (R0, R1). In this case, forwarding from the writeback stage to the execute stage is used to provide the value of R0 to the execute stage without waiting for it to be read from the register file.

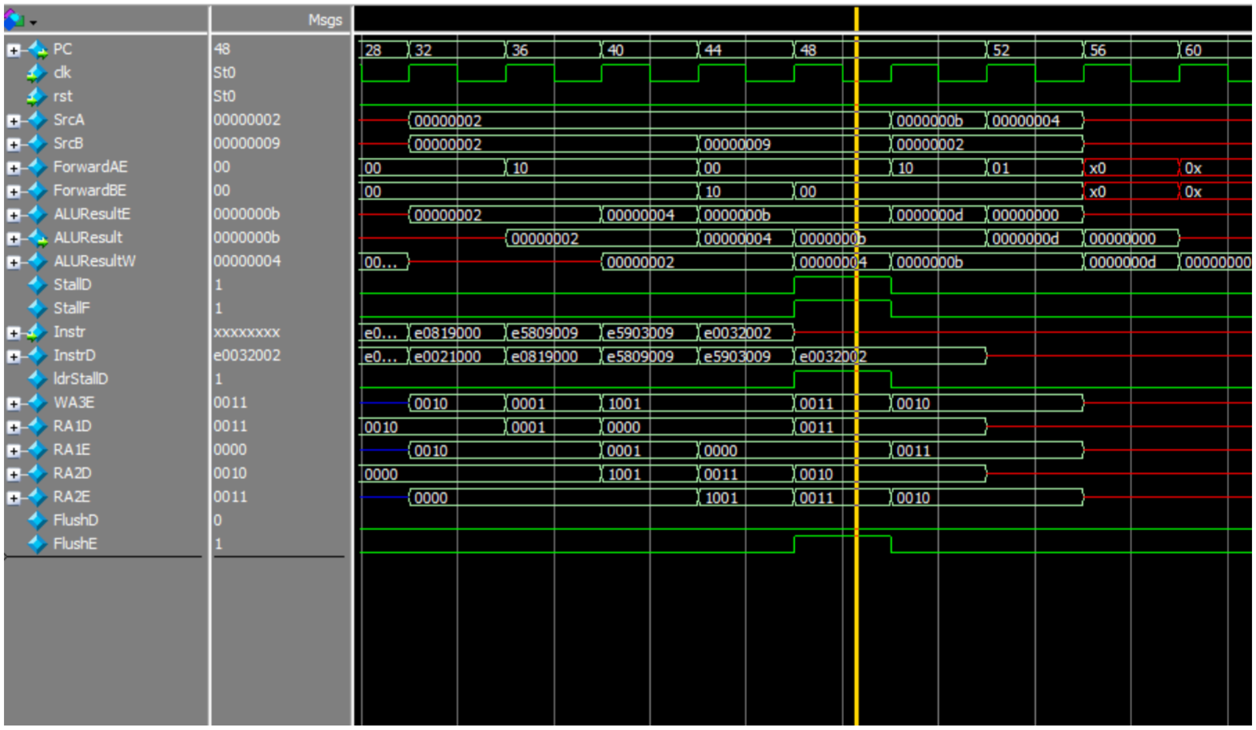
From the following waveform, we could see that at sixth cycle, the ALUResultW has been given and is going to write back to the register in the writeback stage for the first instruction, and as R0 is being called as source A in the execution stage in the third instruction. The ForwardAE becomes 01 let the source A read the ALUResultW directly from the R0 in the writeback stage.



iii. An example of stalling for a memory instruction.

In the eleventh instruction, the data from memory location [R0, #9] is loaded into R3 in the memory stage. In the next instruction, R3 is used as an operand in the execution stage to perform addition (R3 + R2). However, since the data in R3 is not yet available (it is still being loaded from memory), the processor has to wait for the memory operation to complete before executing the second instruction. This causes a pipeline stall.

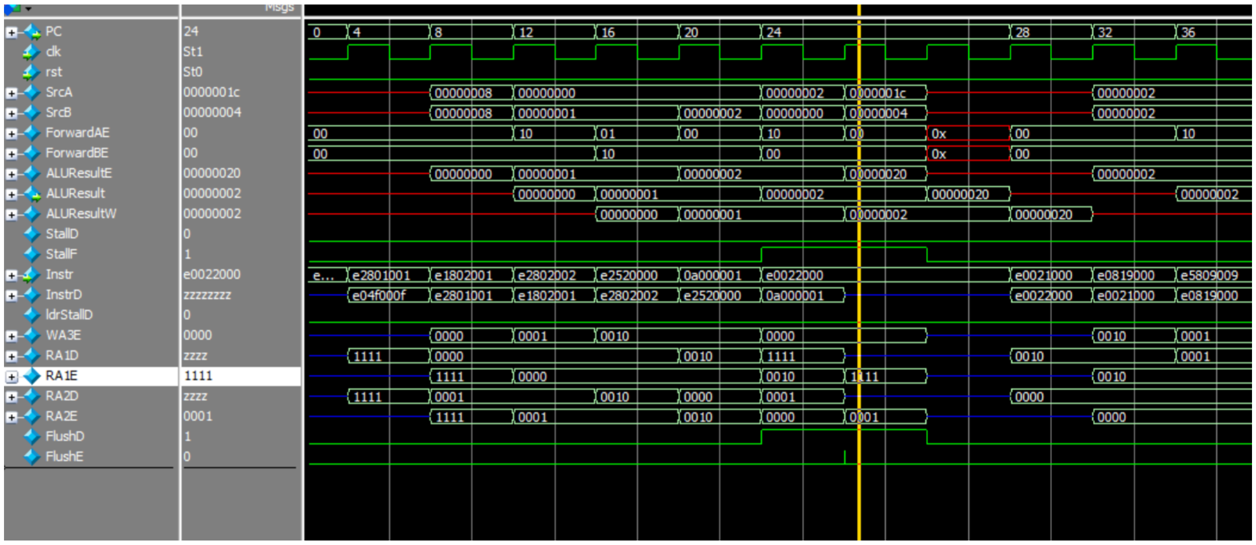
From the following waveform, we could see that when RA1D = WA3E, the stalling is executed. The StallD and StallF are asserted to force the Decode and Fetch stage registers to hold their old value.



iv. An example of flushing for a branch instruction.

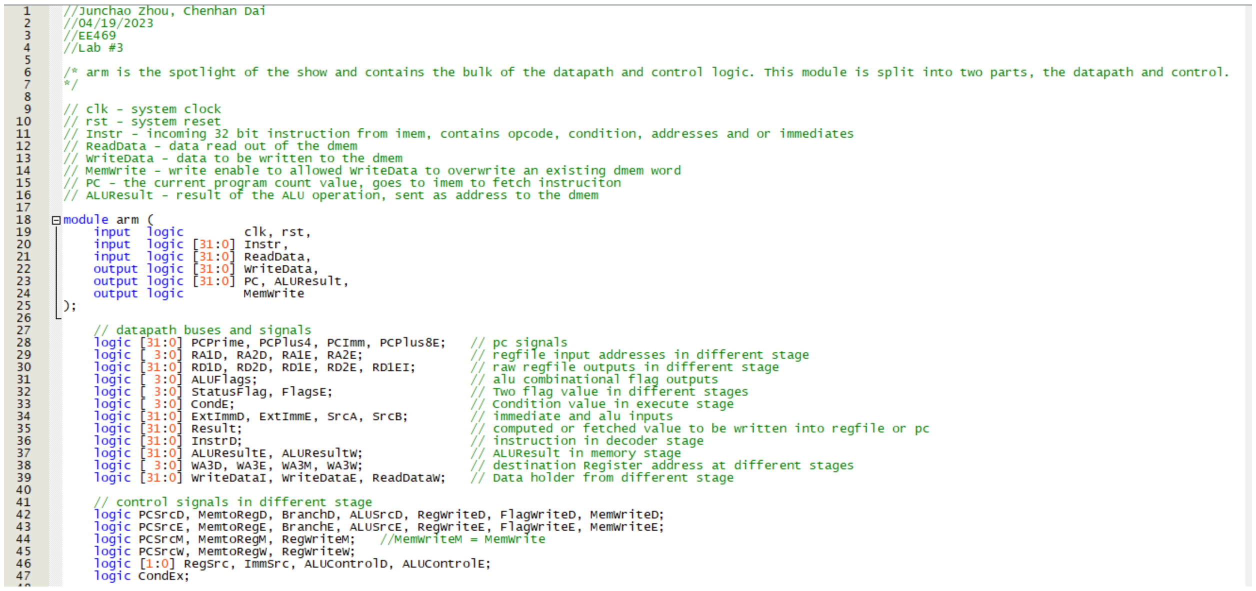
In the sixth instruction, a branch instruction is executed that checks whether the result of the previous subtraction operation (R2 - #0) is zero. If it is zero, the program jumps to TAG1. Since the processor cannot determine whether the branch is taken or not until the execution stage of the next instruction, the instructions in the pipeline after the sixth instruction have to be flushed (i.e., discarded) if the branch is taken. This is done to ensure that the program counter points to the correct instruction after the branch is taken.

From the following waveform, we could see that when branch is taken, flushD and flushE are asserted to force the registers of Decode and Execute stages flushed.

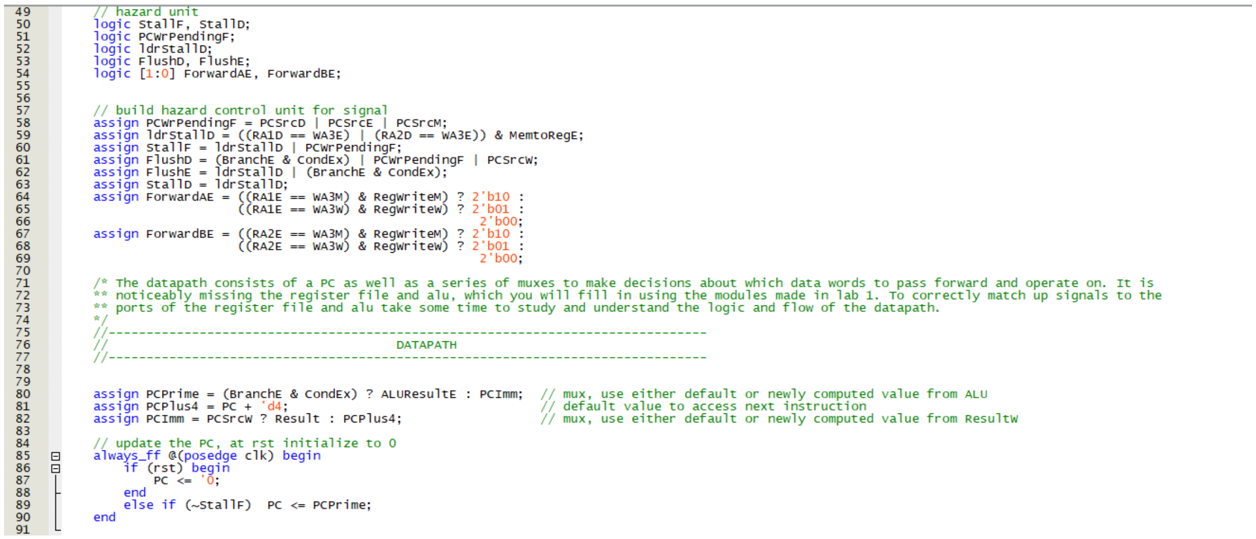


Appendix: SystemVerilog code

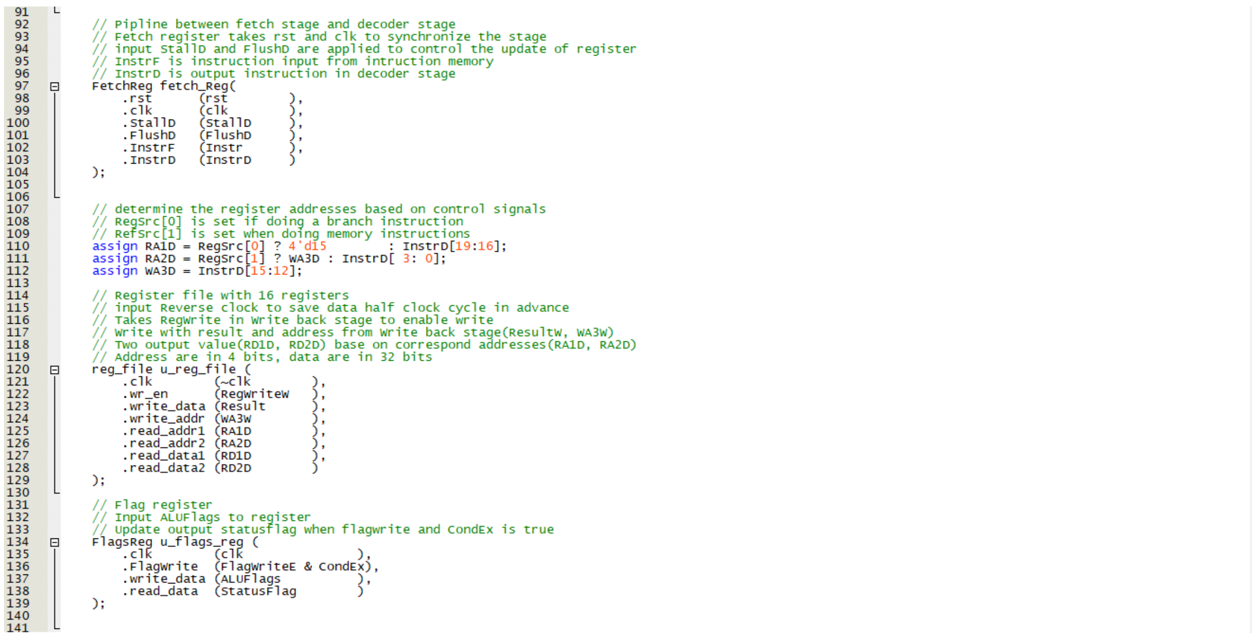
1) **arm.sv**



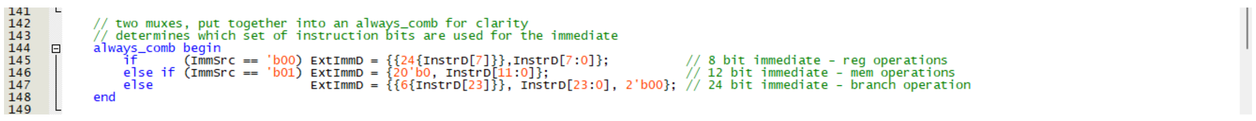
```
//Junchao Zhou, Chenhan Dai
//04/19/2023
//EE469
//Lab #3

/* arm is the spotlight of the show and contains the bulk of the datapath and control logic. This module is split into two parts, the datapath and control.
*/

// clk - system clock
// rst - system reset
// Instr - incoming 32 bit instruction from imem, contains opcode, condition, addresses and or immediates
// ReadData - data read out of the dmem
// WriteData - data to be written to the dmem
// MemWrite - write enable to allowed WriteData to overwrite an existing dmem word
// PC - the current program count value, goes to imem to fetch instruciton
// ALUResult - result of the ALU operation, sent as address to the dmem

module arm (
    input  logic        clk, rst,
    input  logic [31:0] Instr,
    input  logic [31:0] ReadData,
    output logic [31:0] WriteData,
    output logic [31:0] PC, ALUResult,
    output logic        MemWrite
);

    // datapath buses and signals
    logic [31:0] PCPrime, PCPlus4, PCImm, PCPlus8E;   // pc signals
    logic [ 3:0] RA1D, RA2D, RA1E, RA2E;              // regfile input addresses in different stage
    logic [31:0] RD1D, RD2D, RD1E, RD2E, RD1EI;       // raw regfile outputs in different stage
    logic [ 3:0] ALUFlags;                            // alu combinational flag outputs
    logic [ 3:0] StatusFlag, FlagsE;                  // Two flag value in different stages
    logic [ 3:0] CondE;                               // Condition value in execute stage
    logic [31:0] ExtImmD, ExtImmE, SrcA, SrcB;        // immediate and alu inputs
    logic [31:0] Result;                              // computed or fetched value to be written into regfile or pc
    logic [31:0] InstrD;                              // instruction in decoder stage
    logic [31:0] ALUResultE, ALUResultW;              // ALUResult in memory stage
    logic [ 3:0] WA3D, WA3E, WA3M, WA3W;              // destination Register address at different stages
    logic [31:0] WriteDataI, WriteDataE, ReadDataW;   // Data holder from different stage

    // control signals in different stage
    logic PCSrcD, MemtoRegD, BranchD, ALUSrcD, RegWriteD, FlagWriteD, MemWriteD;
    logic PCSrcE, MemtoRegE, BranchE, ALUSrcE, RegWriteE, FlagWriteE, MemWriteE;
    logic PCSrcM, MemtoRegM, RegWriteM;    //MemWriteM = MemWrite
    logic PCSrcW, MemtoRegW, RegWriteW;
    logic [1:0] RegSrc, ImmSrc, ALUControlD, ALUControlE;
    logic CondEx;
```



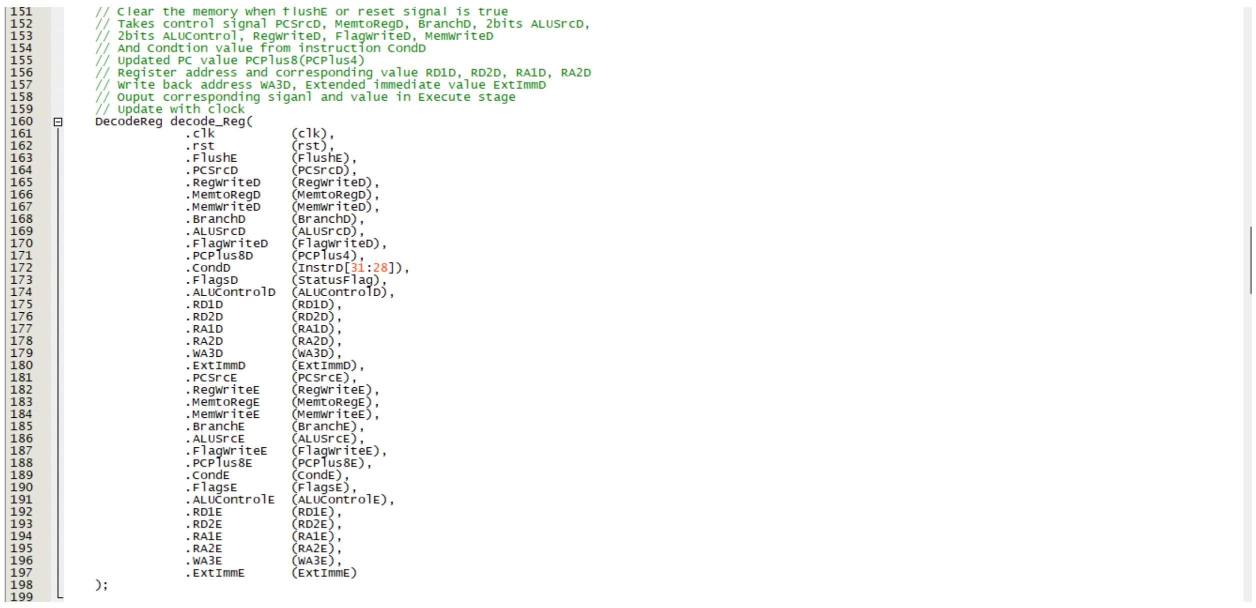
```
    // hazard unit
    logic StallF, StallD;
    logic PCWrPendingF;
    logic ldrStallD;
    logic FlushD, FlushE;
    logic [1:0] ForwardAE, ForwardBE;


    // build hazard control unit for signal
    assign PCWrPendingF = PCSrcD | PCSrcE | PCSrcM;
    assign ldrStallD = ((RA1D == WA3E) | (RA2D == WA3E)) & MemtoRegE;
    assign StallF = ldrStallD | PCWrPendingF;
    assign FlushD = (BranchE & CondEx) | PCWrPendingF | PCSrcW;
    assign FlushE = ldrStallD | (BranchE & CondEx);
    assign StallD = ldrStallD;
    assign ForwardAE = ((RA1E == WA3M) & RegWriteM) ? 2'b10 :
                       ((RA1E == WA3W) & RegWriteW) ? 2'b01 :
                                                       2'b00;
    assign ForwardBE = ((RA2E == WA3M) & RegWriteM) ? 2'b10 :
                       ((RA2E == WA3W) & RegWriteW) ? 2'b01 :
                                                       2'b00;

    /* The datapath consists of a PC as well as a series of muxes to make decisions about which data words to pass forward and operate on. It is
    ** noticeably missing the register file and alu, which you will fill in using the modules made in lab 1. To correctly match up signals to the
    ** ports of the register file and alu take some time to study and understand the logic and flow of the datapath.
    */
    //-------------------------------------------------------------------------------
    //                                      DATAPATH
    //-------------------------------------------------------------------------------


    assign PCPrime = (BranchE & CondEx) ? ALUResultE : PCImm;   // mux, use either default or newly computed value from ALU
    assign PCPlus4 = PC + 'd4;                                  // default value to access next instruction
    assign PCImm = PCSrcW ? Result : PCPlus4;                   // mux, use either default or newly computed value from ResultW

    // update the PC, at rst initialize to 0
    always_ff @(posedge clk) begin
        if (rst) begin
            PC <= '0;
        end
        else if (~StallF)  PC <= PCPrime;
    end
```



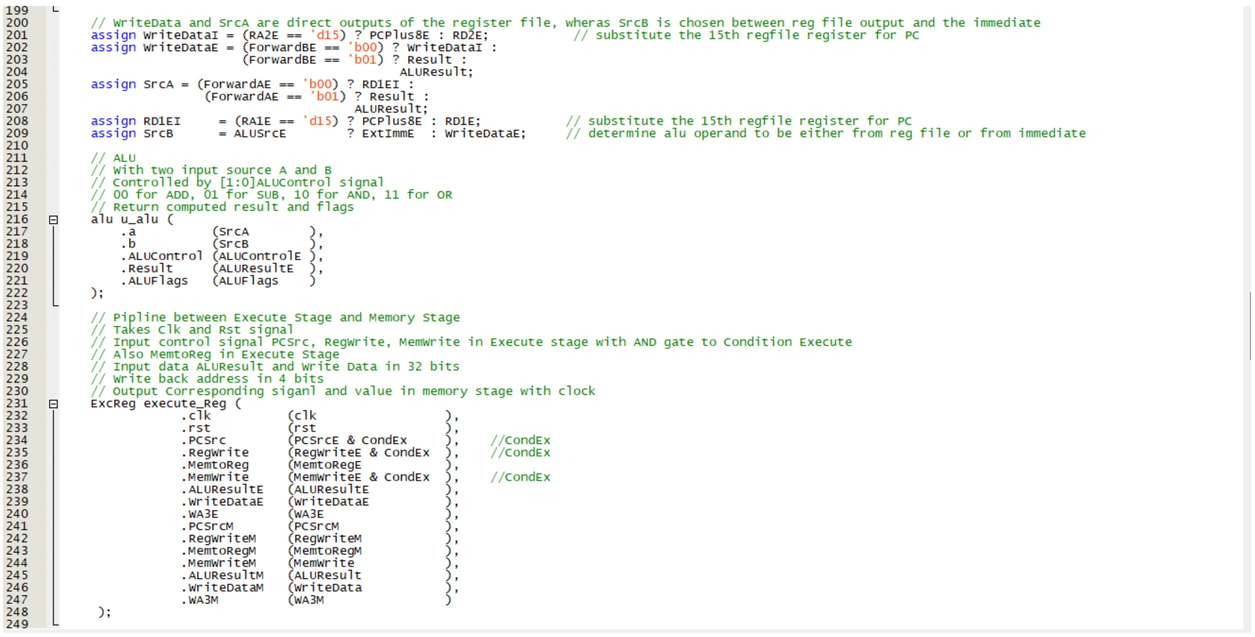
```
    // Pipline between fetch stage and decoder stage
    // Fetch register takes rst and clk to synchronize the stage
    // input StallD and FlushD are applied to control the update of register
    // InstrF is instruction input from intruction memory
    // InstrD is output instruction in decoder stage
    FetchReg fetch_Reg(
        .rst      (rst        ),
        .clk      (clk        ),
        .StallD   (StallD     ),
        .FlushD   (FlushD     ),
        .InstrF   (Instr      ),
        .InstrD   (InstrD     )
    );


    // determine the register addresses based on control signals
    // RegSrc[0] is set if doing a branch instruction
    // RefSrc[1] is set when doing memory instructions
    assign RA1D = RegSrc[0] ? 4'd15        : InstrD[19:16];
    assign RA2D = RegSrc[1] ? WA3D : InstrD[ 3: 0];
    assign WA3D = InstrD[15:12];

    // Register file with 16 registers
    // input Reverse clock to save data half clock cycle in advance
    // Takes RegWrite in Write back stage to enable write
    // Write with result and address from Write back stage(ResultW, WA3W)
    // Two output value(RD1D, RD2D) base on correspond addresses(RA1D, RA2D)
    // Address are in 4 bits, data are in 32 bits
    reg_file u_reg_file (
        .clk        (~clk        ),
        .wr_en      (RegWriteW   ),
        .write_data (Result      ),
        .write_addr (WA3W        ),
        .read_addr1 (RA1D        ),
        .read_addr2 (RA2D        ),
        .read_data1 (RD1D        ),
        .read_data2 (RD2D        )
    );

    // Flag register
    // Input ALUFlags to register
    // Update output statusflag when flagwrite and CondEx is true
    FlagsReg u_flags_reg (
        .clk        (clk                ),
        .FlagWrite  (FlagWriteE & CondEx),
        .write_data (ALUFlags           ),
        .read_data  (StatusFlag         )
    );

```



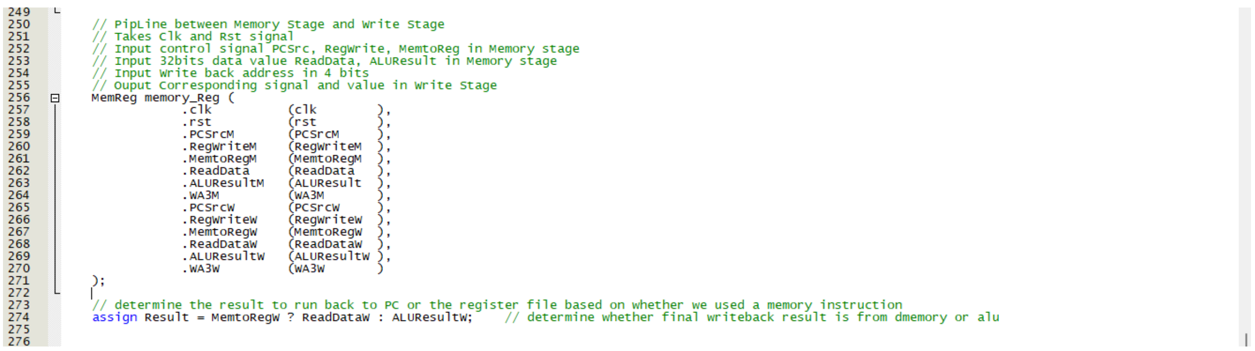
```
    // two muxes, put together into an always_comb for clarity
    // determines which set of instruction bits are used for the immediate
    always_comb begin
        if      (ImmSrc == 'b00) ExtImmD = {{24{InstrD[7]}},InstrD[7:0]};          // 8 bit immediate - reg operations
        else if (ImmSrc == 'b01) ExtImmD = {20'b0, InstrD[11:0]};                  // 12 bit immediate - mem operations
        else                     ExtImmD = {{6{InstrD[23]}}, InstrD[23:0], 2'b00}; // 24 bit immediate - branch operation
    end
```



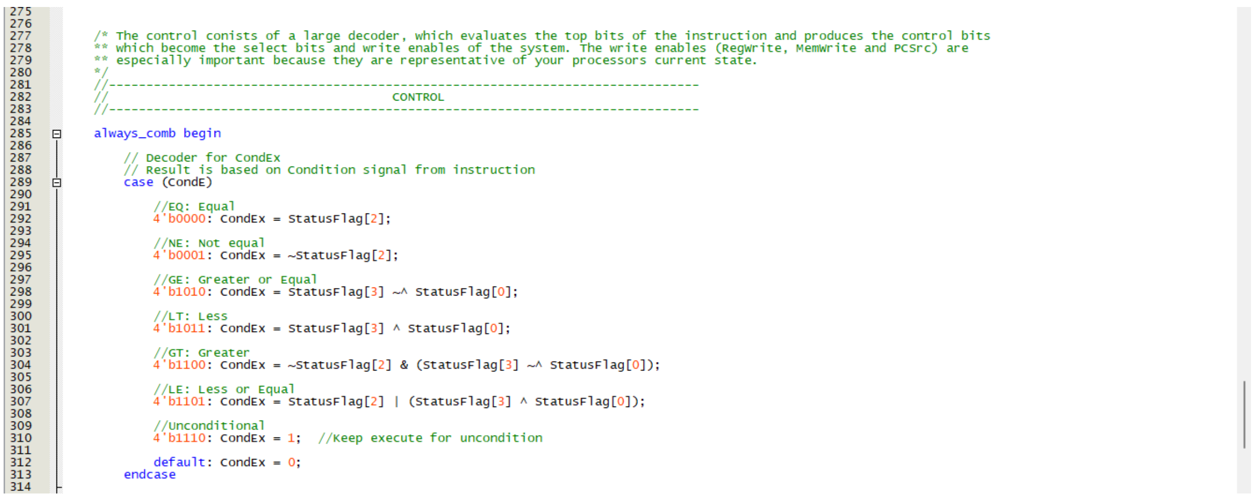
```
    // Clear the memory when flushE or reset signal is true
    // Takes control signal PCSrcD, MemtoRegD, BranchD, 2bits ALUSrcD,
    // 2bits ALUControl, RegWriteD, FlagWriteD, MemWriteD
    // And Condtion value from instruction CondD
    // Updated PC value PCPlus8(PCPlus4)
    // Register address and corresponding value RD1D, RD2D, RA1D, RA2D
    // Write back address WA3D, Extended immediate value ExtImmD
    // Ouput corresponding siganl and value in Execute stage
    // Update with clock
    DecodeReg decode_Reg(
                .clk           (clk),
                .rst           (rst),
                .FlushE        (FlushE),
                .PCSrcD        (PCSrcD),
                .RegWriteD     (RegWriteD),
                .MemtoRegD     (MemtoRegD),
                .MemWriteD     (MemWriteD),
                .BranchD       (BranchD),
                .ALUSrcD       (ALUSrcD),
                .FlagWriteD    (FlagWriteD),
                .PCPlus8D      (PCPlus4),
                .CondD         (InstrD[31:28]),
                .FlagsD        (StatusFlag),
                .ALUControlD   (ALUControlD),
                .RD1D          (RD1D),
                .RD2D          (RD2D),
                .RA1D          (RA1D),
                .RA2D          (RA2D),
                .WA3D          (WA3D),
                .ExtImmD       (ExtImmD),
                .PCSrcE        (PCSrcE),
                .RegWriteE     (RegWriteE),
                .MemtoRegE     (MemtoRegE),
                .MemWriteE     (MemWriteE),
                .BranchE       (BranchE),
                .ALUSrcE       (ALUSrcE),
                .FlagWriteE    (FlagWriteE),
                .PCPlus8E      (PCPlus8E),
                .CondE         (CondE),
                .FlagsE        (FlagsE),
                .ALUControlE   (ALUControlE),
                .RD1E          (RD1E),
                .RD2E          (RD2E),
                .RA1E          (RA1E),
                .RA2E          (RA2E),
                .WA3E          (WA3E),
                .ExtImmE       (ExtImmE)
    );

```



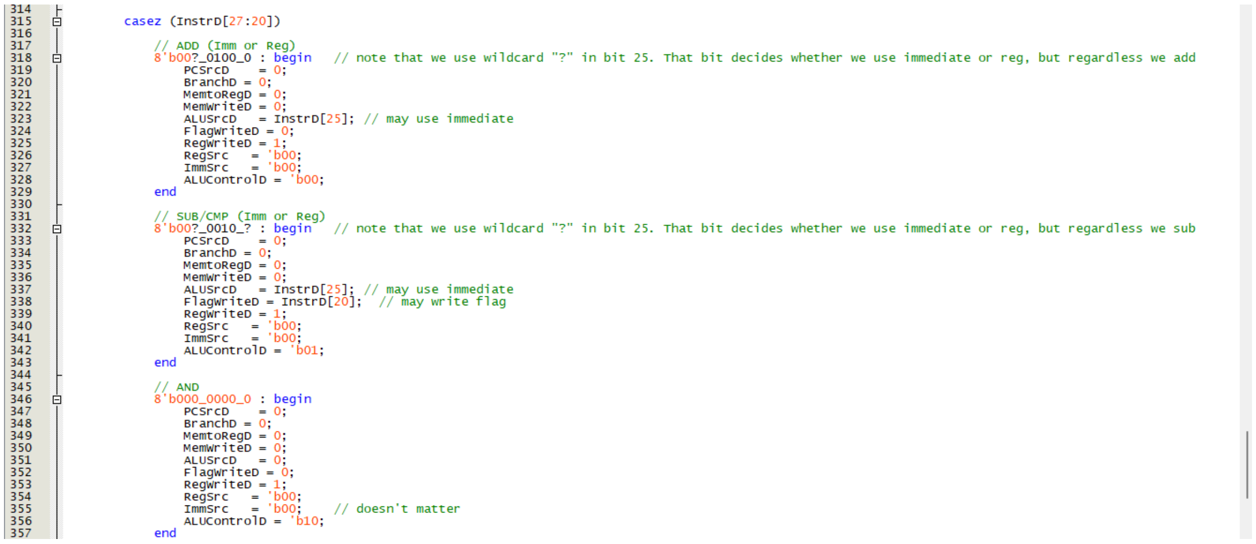
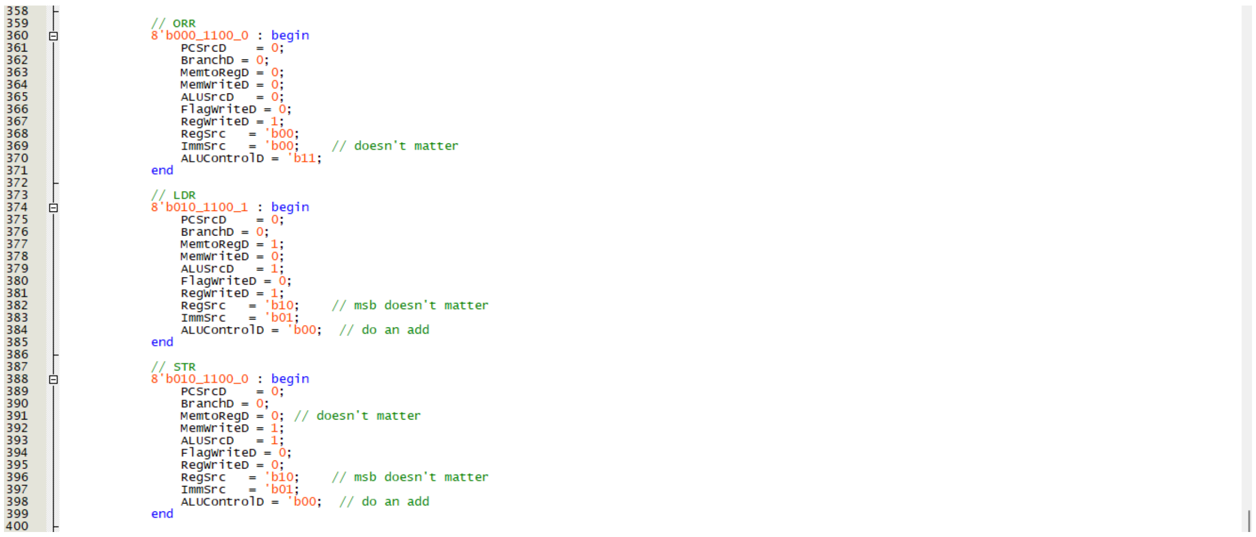
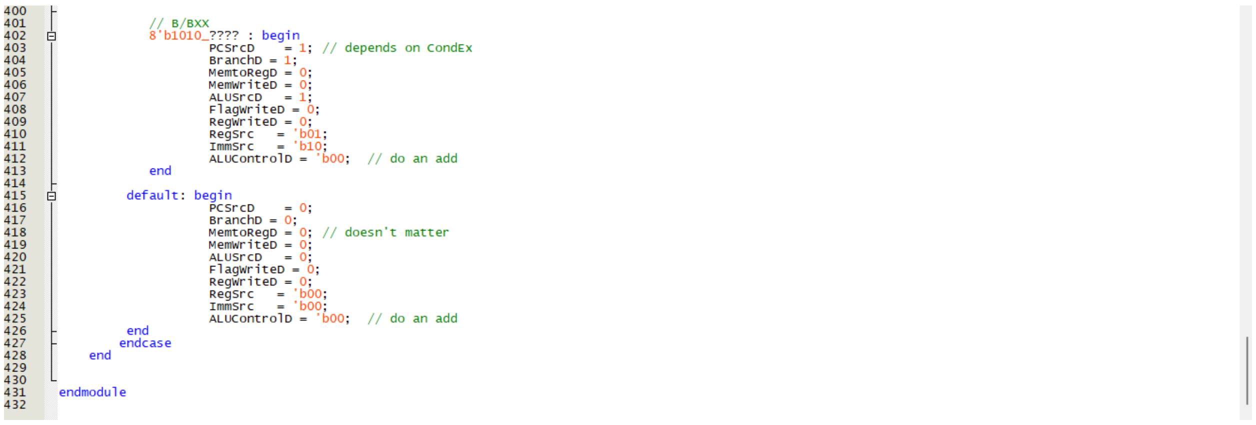
```
    // WriteData and SrcA are direct outputs of the register file, wheras SrcB is chosen between reg file output and the immediate
    assign WriteDataI = (RA2E == 'd15) ? PCPlus8E : RD2E;            // substitute the 15th regfile register for PC
    assign WriteDataE = (ForwardBE == 'b00) ? WriteDataI :
                        (ForwardBE == 'b01) ? Result :
                                              ALUResult;
    assign SrcA = (ForwardAE == 'b00) ? RD1EI :
                  (ForwardAE == 'b01) ? Result :
                                        ALUResult;
    assign RD1EI     = (RA1E == 'd15) ? PCPlus8E : RD1E;            // substitute the 15th regfile register for PC
    assign SrcB      = ALUSrcE        ? ExtImmE  : WriteDataE;      // determine alu operand to be either from reg file or from immediate

    // ALU
    // With two input source A and B
    // Controlled by [1:0]ALUControl signal
    // 00 for ADD, 01 for SUB, 10 for AND, 11 for OR
    // Return computed result and flags
    alu u_alu (
        .a          (SrcA        ),
        .b          (SrcB        ),
        .ALUControl (ALUControlE ),
        .Result     (ALUResultE  ),
        .ALUFlags   (ALUFlags    )
    );

    // Pipline between Execute Stage and Memory Stage
    // Takes Clk and Rst signal
    // Input control signal PCSrc, RegWrite, MemWrite in Execute stage with AND gate to Condition Execute
    // Also MemtoReg in Execute Stage
    // Input data ALUResult and Write Data in 32 bits
    // Write back address in 4 bits
    // Output Corresponding siganl and value in memory stage with clock
    ExcReg execute_Reg (
                .clk          (clk                  ),
                .rst          (rst                  ),
                .PCSrc        (PCSrcE & CondEx      ),     //CondEx
                .RegWrite     (RegWriteE & CondEx   ),     //CondEx
                .MemtoReg     (MemtoRegE            ),
                .MemWrite     (MemWriteE & CondEx   ),     //CondEx
                .ALUResultE   (ALUResultE           ),
                .WriteDataE   (WriteDataE           ),
                .WA3E         (WA3E                 ),
                .PCSrcM       (PCSrcM               ),
                .RegWriteM    (RegWriteM            ),
                .MemtoRegM    (MemtoRegM            ),
                .MemWriteM    (MemWrite             ),
                .ALUResultM   (ALUResult            ),
                .WriteDataM   (WriteData            ),
                .WA3M         (WA3M                 )
     );
```



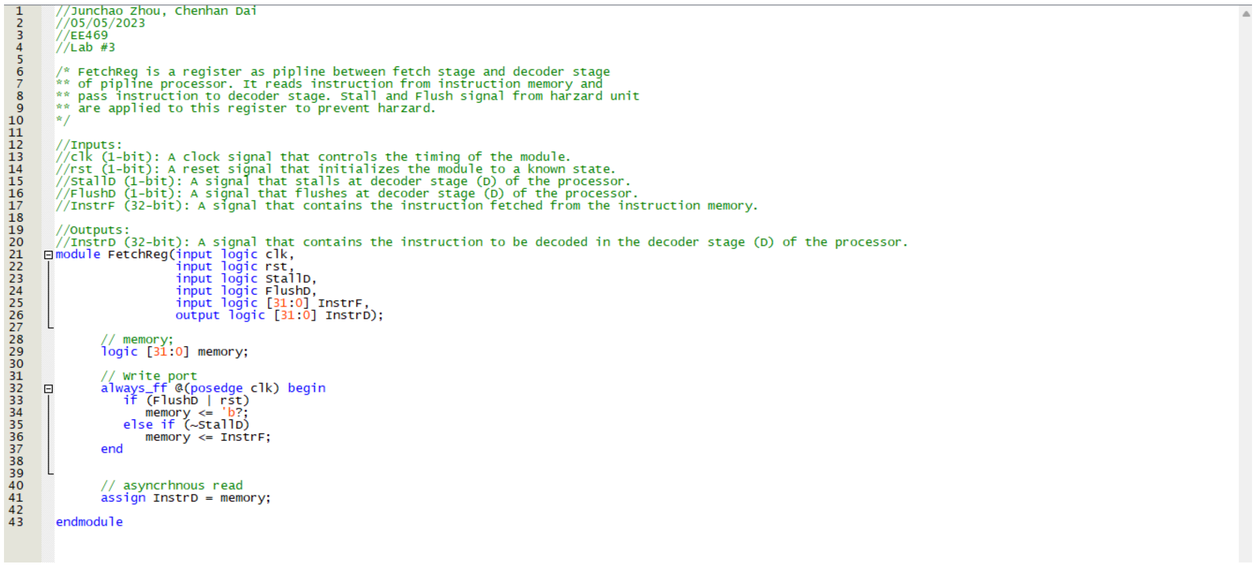
```
    // PipLine between Memory Stage and Write Stage
    // Takes Clk and Rst signal
    // Input control signal PCSrc, RegWrite, MemtoReg in Memory stage
    // Input 32bits data value ReadData, ALUResult in Memory stage
    // Input Write back address in 4 bits
    // Ouput Corresponding signal and value in Write Stage
    MemReg memory_Reg (
                .clk          (clk        ),
                .rst          (rst        ),
                .PCSrcM       (PCSrcM     ),
                .RegWriteM    (RegWriteM  ),
                .MemtoRegM    (MemtoRegM  ),
                .ReadData     (ReadData   ),
                .ALUResultM   (ALUResult  ),
                .WA3M         (WA3M       ),
                .PCSrcW       (PCSrcW     ),
                .RegWriteW    (RegWriteW  ),
                .MemtoRegW    (MemtoRegW  ),
                .ReadDataW    (ReadDataW  ),
                .ALUResultW   (ALUResultW ),
                .WA3W         (WA3W       )
    );

    // determine the result to run back to PC or the register file based on whether we used a memory instruction
    assign Result = MemtoRegW ? ReadDataW : ALUResultW;    // determine whether final writeback result is from dmemory or alu
```



```
    /* The control conists of a large decoder, which evaluates the top bits of the instruction and produces the control bits
    ** which become the select bits and write enables of the system. The write enables (RegWrite, MemWrite and PCSrc) are
    ** especially important because they are representative of your processors current state.
    */
    //-------------------------------------------------------------------------------
    //                                     CONTROL
    //-------------------------------------------------------------------------------

    always_comb begin

        // Decoder for CondEx
        // Result is based on Condition signal from instruction
        case (CondE)

            //EQ: Equal
            4'b0000: CondEx = StatusFlag[2];

            //NE: Not equal
            4'b0001: CondEx = ~StatusFlag[2];

            //GE: Greater or Equal
            4'b1010: CondEx = StatusFlag[3] ~^ StatusFlag[0];

            //LT: Less
            4'b1011: CondEx = StatusFlag[3] ^ StatusFlag[0];

            //GT: Greater
            4'b1100: CondEx = ~StatusFlag[2] & (StatusFlag[3] ~^ StatusFlag[0]);

            //LE: Less or Equal
            4'b1101: CondEx = StatusFlag[2] | (StatusFlag[3] ^ StatusFlag[0]);

            //Unconditional
            4'b1110: CondEx = 1;  //Keep execute for uncondition

            default: CondEx = 0;
        endcase
```

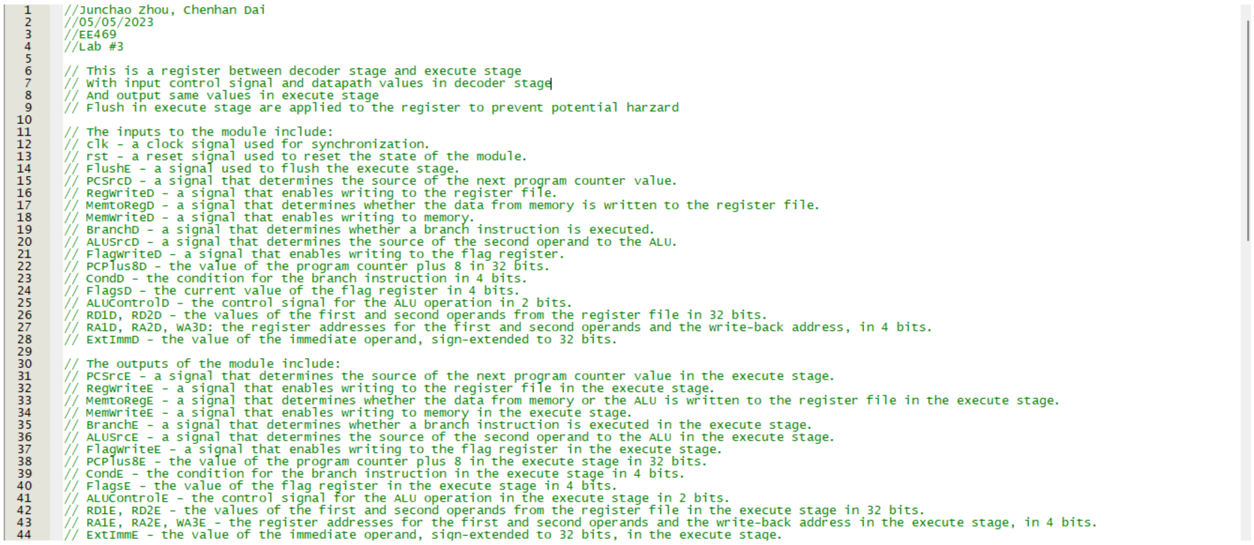

```

        casez (InstrD[27:20])

            // ADD (Imm or Reg)
            8'b00?_0100_0 : begin   // note that we use wildcard "?" in bit 25. That bit decides whether we use immediate or reg, but regardless we add
                PCSrcD    = 0;
                BranchD = 0;
                MemtoRegD = 0;
                MemWriteD = 0;
                ALUSrcD   = InstrD[25]; // may use immediate
                FlagWriteD = 0;
                RegWriteD = 1;
                RegSrc    = 'b00;
                ImmSrc    = 'b00;
                ALUControlD = 'b00;
            end

            // SUB/CMP (Imm or Reg)
            8'b00?_0010_? : begin   // note that we use wildcard "?" in bit 25. That bit decides whether we use immediate or reg, but regardless we sub
                PCSrcD    = 0;
                BranchD = 0;
                MemtoRegD = 0;
                MemWriteD = 0;
                ALUSrcD   = InstrD[25]; // may use immediate
                FlagWriteD = InstrD[20];  // may write flag
                RegWriteD = 1;
                RegSrc    = 'b00;
                ImmSrc    = 'b00;
                ALUControlD = 'b01;
            end

            // AND
            8'b000_0000_0 : begin
                PCSrcD    = 0;
                BranchD = 0;
                MemtoRegD = 0;
                MemWriteD = 0;
                ALUSrcD   = 0;
                FlagWriteD = 0;
                RegWriteD = 1;
                RegSrc    = 'b00;
                ImmSrc    = 'b00;     // doesn't matter
                ALUControlD = 'b10;
            end
```

```

            // ORR
            8'b000_1100_0 : begin
                PCSrcD    = 0;
                BranchD = 0;
                MemtoRegD = 0;
                MemWriteD = 0;
                ALUSrcD   = 0;
                FlagWriteD = 0;
                RegWriteD = 1;
                RegSrc    = 'b00;
                ImmSrc    = 'b00;     // doesn't matter
                ALUControlD = 'b11;
            end

            // LDR
            8'b010_1100_1 : begin
                PCSrcD    = 0;
                BranchD = 0;
                MemtoRegD = 1;
                MemWriteD = 0;
                ALUSrcD   = 1;
                FlagWriteD = 0;
                RegWriteD = 1;
                RegSrc    = 'b10;     // msb doesn't matter
                ImmSrc    = 'b01;
                ALUControlD = 'b00;   // do an add
            end

            // STR
            8'b010_1100_0 : begin
                PCSrcD    = 0;
                BranchD = 0;
                MemtoRegD = 0; // doesn't matter
                MemWriteD = 1;
                ALUSrcD   = 1;
                FlagWriteD = 0;
                RegWriteD = 0;
                RegSrc    = 'b10;     // msb doesn't matter
                ImmSrc    = 'b01;
                ALUControlD = 'b00;   // do an add
            end

```

```

            // B/BXX
            8'b1010_???? : begin
                    PCSrcD   = 1; // depends on CondEx
                    BranchD = 1;
                    MemtoRegD = 0;
                    MemWriteD = 0;
                    ALUSrcD   = 1;
                    FlagWriteD = 0;
                    RegWriteD = 0;
                    RegSrc   = 'b01;
                    ImmSrc   = 'b10;
                    ALUControlD = 'b00;   // do an add
            end

        default: begin
                    PCSrcD   = 0;
                    BranchD = 0;
                    MemtoRegD = 0; // doesn't matter
                    MemWriteD = 0;
                    ALUSrcD   = 0;
                    FlagWriteD = 0;
                    RegWriteD = 0;
                    RegSrc   = 'b00;
                    ImmSrc   = 'b00;
                    ALUControlD = 'b00;   // do an add
        end
        endcase
    end

endmodule

```

## 2) FetchReg.sv

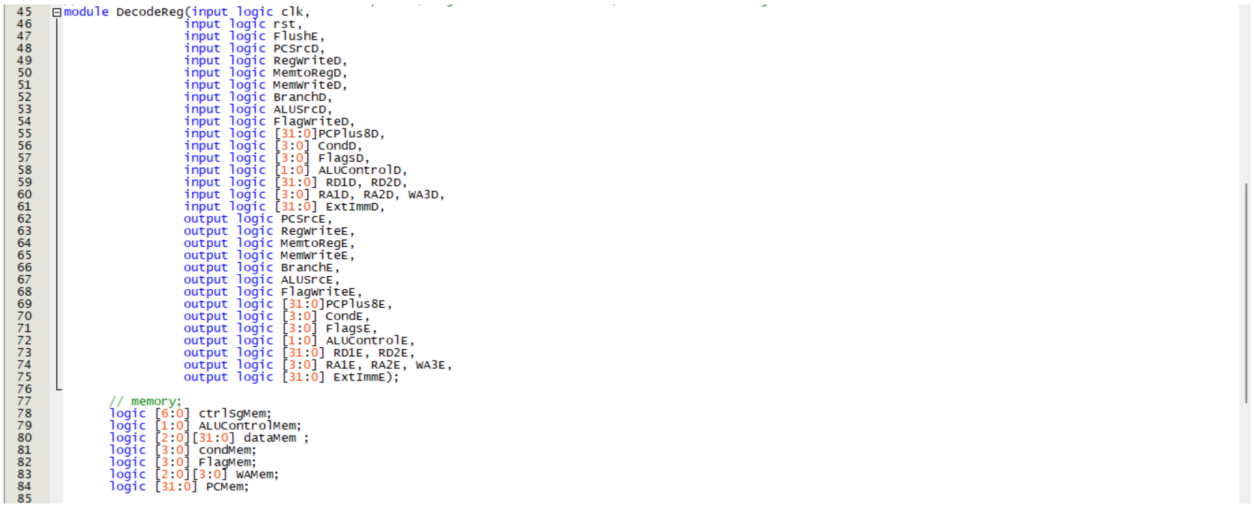


```
//Junchao Zhou, Chenhan Dai
//05/05/2023
//EE469
//Lab #3

/* FetchReg is a register as pipline between fetch stage and decoder stage
** of pipline processor. It reads instruction from instruction memory and
** pass instruction to decoder stage. Stall and Flush signal from harzard unit
** are applied to this register to prevent harzard.
*/

//Inputs:
//clk (1-bit): A clock signal that controls the timing of the module.
//rst (1-bit): A reset signal that initializes the module to a known state.
//StallD (1-bit): A signal that stalls at decoder stage (D) of the processor.
//FlushD (1-bit): A signal that flushes at decoder stage (D) of the processor.
//InstrF (32-bit): A signal that contains the instruction fetched from the instruction memory.

//Outputs:
//InstrD (32-bit): A signal that contains the instruction to be decoded in the decoder stage (D) of the processor.
module FetchReg(input logic clk,
                input logic rst,
                input logic StallD,
                input logic FlushD,
                input logic [31:0] InstrF,
                output logic [31:0] InstrD);

      // memory;
      logic [31:0] memory;

      // Write port
      always_ff @(posedge clk) begin
         if (FlushD | rst)
            memory <= 'b?;
         else if (~StallD)
            memory <= InstrF;
      end

      // asyncrhnous read
      assign InstrD = memory;

endmodule
```

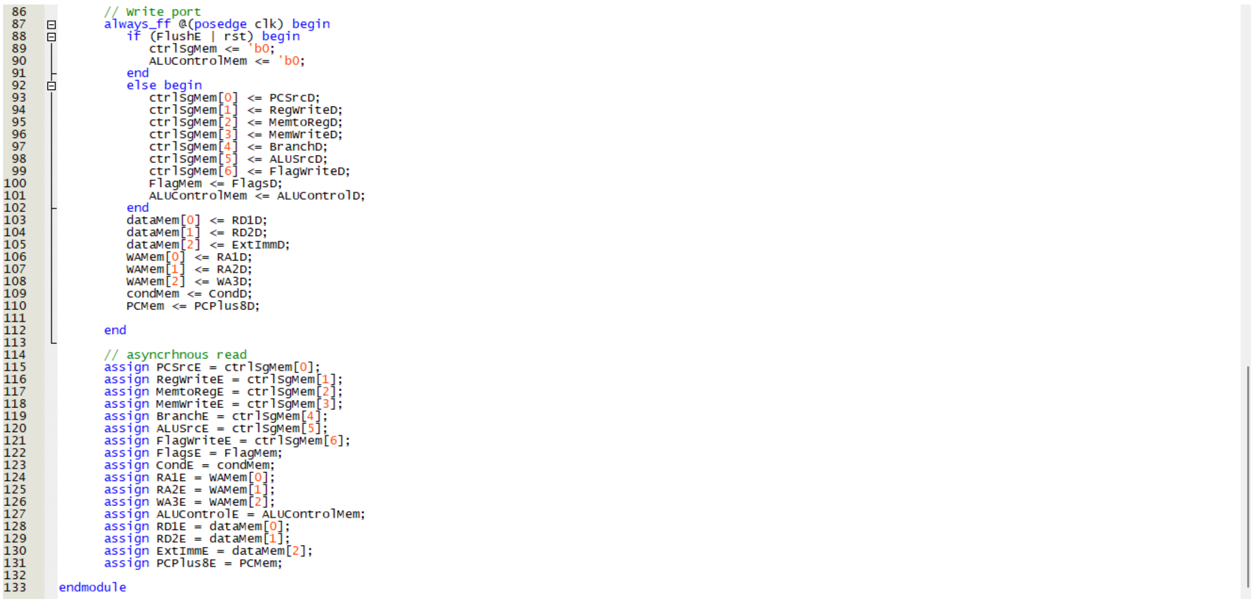
3) DecodeReg.sv



```
//Junchao Zhou, Chenhan Dai
//05/05/2023
//EE469
//Lab #3

// This is a register between decoder stage and execute stage
// With input control signal and datapath values in decoder stage
// And output same values in execute stage
// Flush in execute stage are applied to the register to prevent potential harzard

// The inputs to the module include:
// clk - a clock signal used for synchronization.
// rst - a reset signal used to reset the state of the module.
// FlushE - a signal used to flush the execute stage.
// PCSrcD - a signal that determines the source of the next program counter value.
// RegWriteD - a signal that enables writing to the register file.
// MemtoRegD - a signal that determines whether the data from memory is written to the register file.
// MemWriteD - a signal that enables writing to memory.
// BranchD - a signal that determines whether a branch instruction is executed.
// ALUSrcD - a signal that determines the source of the second operand to the ALU.
// FlagWriteD - a signal that enables writing to the flag register.
// PCPlus8D - the value of the program counter plus 8 in 32 bits.
// CondD - the condition for the branch instruction in 4 bits.
// FlagsD - the current value of the flag register in 4 bits.
// ALUControlD - the control signal for the ALU operation in 2 bits.
// RD1D, RD2D - the values of the first and second operands from the register file in 32 bits.
// RA1D, RA2D, WA3D: the register addresses for the first and second operands and the write-back address, in 4 bits.
// ExtImmD - the value of the immediate operand, sign-extended to 32 bits.

// The outputs of the module include:
// PCSrcE - a signal that determines the source of the next program counter value in the execute stage.
// RegWriteE - a signal that enables writing to the register file in the execute stage.
// MemtoRegE - a signal that determines whether the data from memory or the ALU is written to the register file in the execute stage.
// MemWriteE - a signal that enables writing to memory in the execute stage.
// BranchE - a signal that determines whether a branch instruction is executed in the execute stage.
// ALUSrcE - a signal that determines the source of the second operand to the ALU in the execute stage.
// FlagWriteE - a signal that enables writing to the flag register in the execute stage.
// PCPlus8E - the value of the program counter plus 8 in the execute stage in 32 bits.
// CondE - the condition for the branch instruction in the execute stage in 4 bits.
// FlagsE - the value of the flag register in the execute stage in 4 bits.
// ALUControlE - the control signal for the ALU operation in the execute stage in 2 bits.
// RD1E, RD2E - the values of the first and second operands from the register file in the execute stage in 32 bits.
// RA1E, RA2E, WA3E - the register addresses for the first and second operands and the write-back address in the execute stage, in 4 bits.
// ExtImmE - the value of the immediate operand, sign-extended to 32 bits, in the execute stage.
```

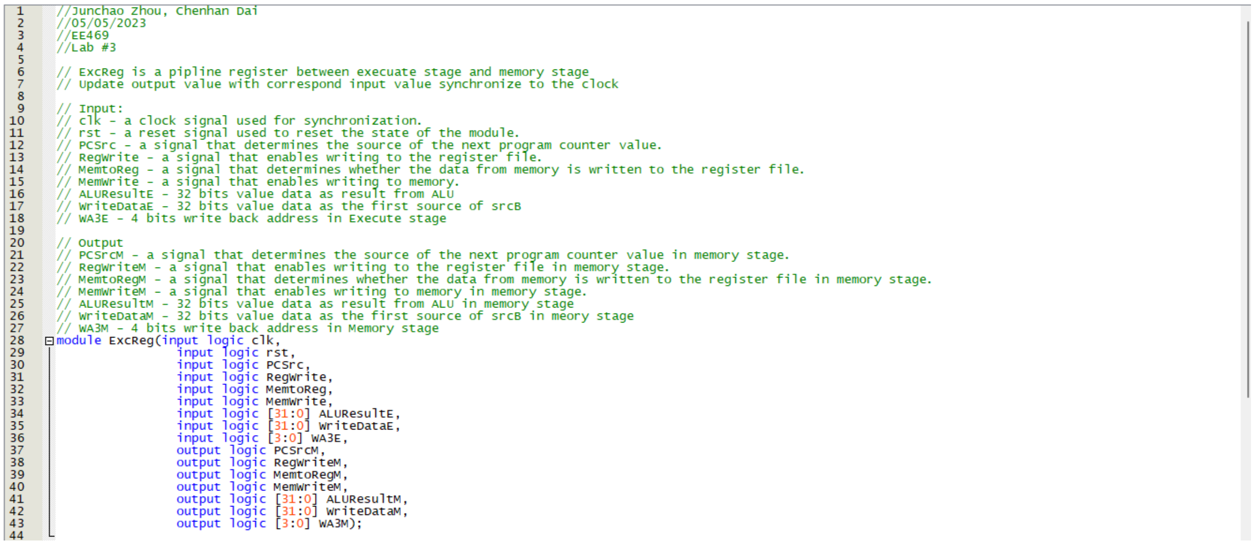


```
module DecodeReg(input logic clk,
                input logic rst,
                input logic FlushE,
                input logic PCSrcD,
                input logic RegWriteD,
                input logic MemtoRegD,
                input logic MemWriteD,
                input logic BranchD,
                input logic ALUSrcD,
                input logic FlagWriteD,
                input logic [31:0]PCPlus8D,
                input logic [3:0] CondD,
                input logic [3:0] FlagsD,
                input logic [1:0] ALUControlD,
                input logic [31:0] RD1D, RD2D,
                input logic [3:0] RA1D, RA2D, WA3D,
                input logic [31:0] ExtImmD,
                output logic PCSrcE,
                output logic RegWriteE,
                output logic MemtoRegE,
                output logic MemWriteE,
                output logic BranchE,
                output logic ALUSrcE,
                output logic FlagWriteE,
                output logic [31:0]PCPlus8E,
                output logic [3:0] CondE,
                output logic [3:0] FlagsE,
                output logic [1:0] ALUControlE,
                output logic [31:0] RD1E, RD2E,
                output logic [3:0] RA1E, RA2E, WA3E,
                output logic [31:0] ExtImmE);

      // memory;
      logic [6:0] ctrlSgMem;
      logic [1:0] ALUControlMem;
      logic [2:0][31:0] dataMem ;
      logic [3:0] condMem;
      logic [3:0] FlagMem;
      logic [2:0][3:0] WAMem;
      logic [31:0] PCMem;
```

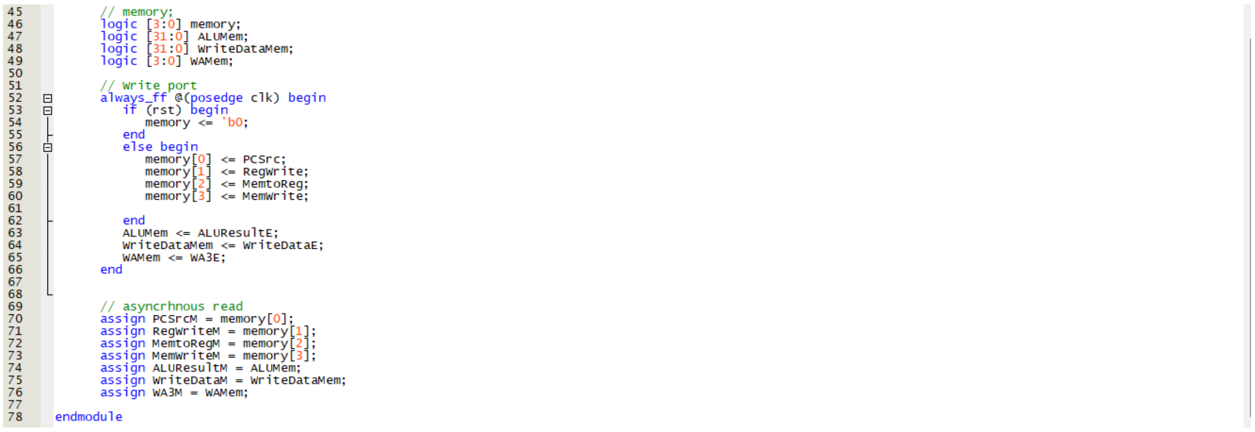


```
      // write port
      always_ff @(posedge clk) begin
         if (FlushE | rst) begin
            ctrlSgMem <= 'b0;
            ALUControlMem <= 'b0;
         end
         else begin
            ctrlSgMem[0] <= PCSrcD;
            ctrlSgMem[1] <= RegWriteD;
            ctrlSgMem[2] <= MemtoRegD;
            ctrlSgMem[3] <= MemWriteD;
            ctrlSgMem[4] <= BranchD;
            ctrlSgMem[5] <= ALUSrcD;
            ctrlSgMem[6] <= FlagWriteD;
            FlagMem <= FlagsD;
            ALUControlMem <= ALUControlD;
         end
         dataMem[0] <= RD1D;
         dataMem[1] <= RD2D;
         dataMem[2] <= ExtImmD;
         WAMem[0] <= RA1D;
         WAMem[1] <= RA2D;
         WAMem[2] <= WA3D;
         condMem <= CondD;
         PCMem <= PCPlus8D;

      end

      // asyncrhnous read
      assign PCSrcE = ctrlSgMem[0];
      assign RegWriteE = ctrlSgMem[1];
      assign MemtoRegE = ctrlSgMem[2];
      assign MemWriteE = ctrlSgMem[3];
      assign BranchE = ctrlSgMem[4];
      assign ALUSrcE = ctrlSgMem[5];
      assign FlagWriteE = ctrlSgMem[6];
      assign FlagsE = FlagMem;
      assign CondE = condMem;
      assign RA1E = WAMem[0];
      assign RA2E = WAMem[1];
      assign WA3E = WAMem[2];
      assign ALUControlE = ALUControlMem;
      assign RD1E = dataMem[0];
      assign RD2E = dataMem[1];
      assign ExtImmE = dataMem[2];
      assign PCPlus8E = PCMem;

endmodule
```

4) ExcReg.sv

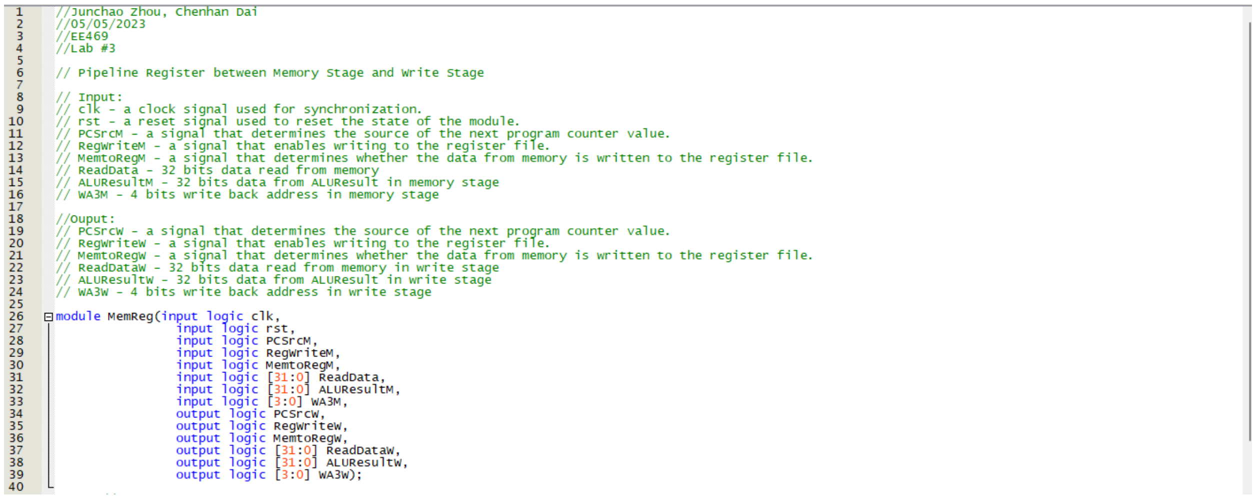


```
//Junchao Zhou, Chenhan Dai
//05/05/2023
//EE469
//Lab #3

// ExcReg is a pipline register between execuate stage and memory stage
// Update output value with correspond input value synchronize to the clock

// Input:
// clk - a clock signal used for synchronization.
// rst - a reset signal used to reset the state of the module.
// PCSrc - a signal that determines the source of the next program counter value.
// RegWrite - a signal that enables writing to the register file.
// MemtoReg - a signal that determines whether the data from memory is written to the register file.
// MemWrite - a signal that enables writing to memory.
// ALUResultE - 32 bits value data as result from ALU
// WriteDataE - 32 bits value data as the first source of srcB
// WA3E - 4 bits write back address in Execute stage

// Output
// PCSrcM - a signal that determines the source of the next program counter value in memory stage.
// RegWriteM - a signal that enables writing to the register file in memory stage.
// MemtoRegM - a signal that determines whether the data from memory is written to the register file in memory stage.
// MemWriteM - a signal that enables writing to memory in memory stage.
// ALUResultM - 32 bits value data as result from ALU in memory stage
// WriteDataM - 32 bits value data as the first source of srcB in meory stage
// WA3M - 4 bits write back address in Memory stage
module ExcReg(input logic clk,
                input logic rst,
                input logic PCSrc,
                input logic RegWrite,
                input logic MemtoReg,
                input logic MemWrite,
                input logic [31:0] ALUResultE,
                input logic [31:0] WriteDataE,
                input logic [3:0] WA3E,
                output logic PCSrcM,
                output logic RegWriteM,
                output logic MemtoRegM,
                output logic MemWriteM,
                output logic [31:0] ALUResultM,
                output logic [31:0] WriteDataM,
                output logic [3:0] WA3M);

```

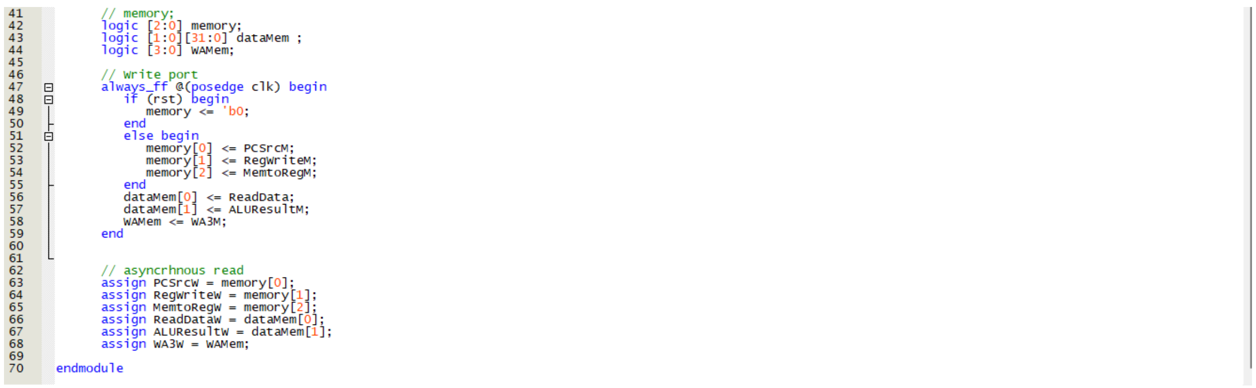


```
      // memory;
      logic [3:0] memory;
      logic [31:0] ALUMem;
      logic [31:0] WriteDataMem;
      logic [3:0] WAMem;

      // write port
      always_ff @(posedge clk) begin
         if (rst) begin
            memory <= 'b0;
         end
         else begin
            memory[0] <= PCSrc;
            memory[1] <= RegWrite;
            memory[2] <= MemtoReg;
            memory[3] <= MemWrite;

         end
         ALUMem <= ALUResultE;
         WriteDataMem <= WriteDataE;
         WAMem <= WA3E;
      end

      // asyncrhnous read
      assign PCSrcM = memory[0];
      assign RegWriteM = memory[1];
      assign MemtoRegM = memory[2];
      assign MemWriteM = memory[3];
      assign ALUResultM = ALUMem;
      assign WriteDataM = WriteDataMem;
      assign WA3M = WAMem;

endmodule
```

5) MemReg.sv



```
//Junchao Zhou, Chenhan Dai
//05/05/2023
//EE469
//Lab #3

// Pipeline Register between Memory Stage and Write Stage

// Input:
// clk - a clock signal used for synchronization.
// rst - a reset signal used to reset the state of the module.
// PCSrcM - a signal that determines the source of the next program counter value.
// RegWriteM - a signal that enables writing to the register file.
// MemtoRegM - a signal that determines whether the data from memory is written to the register file.
// ReadData - 32 bits data read from memory
// ALUResultM - 32 bits data from ALUResult in memory stage
// WA3M - 4 bits write back address in memory stage

//Ouput:
// PCSrcW - a signal that determines the source of the next program counter value.
// RegWriteW - a signal that enables writing to the register file.
// MemtoRegW - a signal that determines whether the data from memory is written to the register file.
// ReadDataW - 32 bits data read from memory in write stage
// ALUResultW - 32 bits data from ALUResult in write stage
// WA3W - 4 bits write back address in write stage

module MemReg(input logic clk,
                input logic rst,
                input logic PCSrcM,
                input logic RegWriteM,
                input logic MemtoRegM,
                input logic [31:0] ReadData,
                input logic [31:0] ALUResultM,
                input logic [3:0] WA3M,
                output logic PCSrcW,
                output logic RegWriteW,
                output logic MemtoRegW,
                output logic [31:0] ReadDataW,
                output logic [31:0] ALUResultW,
                output logic [3:0] WA3W);
```



```
      // memory;
      logic [2:0] memory;
      logic [1:0][31:0] dataMem ;
      logic [3:0] WAMem;

      // Write port
      always_ff @(posedge clk) begin
         if (rst) begin
            memory <= 'b0;
         end
         else begin
            memory[0] <= PCSrcM;
            memory[1] <= RegWriteM;
            memory[2] <= MemtoRegM;
         end
         dataMem[0] <= ReadData;
         dataMem[1] <= ALUResultM;
         WAMem <= WA3M;
      end


      // asyncrhnous read
      assign PCSrcW = memory[0];
      assign RegWriteW = memory[1];
      assign MemtoRegW = memory[2];
      assign ReadDataW = dataMem[0];
      assign ALUResultW = dataMem[1];
      assign WA3W = WAMem;

endmodule
```

6) memfile3.dat

// ADD R - 1110_000_0100_0_AAAA_DDDD_0000_0000_BBBB
// ADD I - 1110_001_0100_0_AAAA_DDDD_0000_IIII_IIII
// SUB R - 1110_000_0010_0_AAAA_DDDD_0000_0000_BBBB
// SUB I - 1110_001_0010_0_AAAA_DDDD_0000_IIII_IIII
// CMP R - 1110_000_0010_1_AAAA_DDDD_0000_0000_BBBB
// CMP I - 1110_001_0010_1_AAAA_DDDD_0000_IIII_IIII
// AND - 1110_000_0000_0_AAAA_DDDD_0000_0000_BBBB
// ORR - 1110_000_1100_0_AAAA_DDDD_0000_0000_BBBB
// LDR - 1110_010_1100_1_AAAA_DDDD_IIII_IIII_IIII
// STR - 1110_010_1100_0_AAAA_DDDD_IIII_IIII_IIII
// COND_1010_IIII_IIII_IIII_IIII_IIII_IIII

// Equal - COND = 0000
// Not Equal - COND = 0001
// Greater or Equal - COND = 1010
// Greater - COND = 1100
// Less or Equal - COND = 1101
// Less - COND = 1011

```
11100000010011110000000000001111   //  MAIN        SUB  R0 R15 R15
11100010100000000001000000000001  //               ADD  R1 R0 #1
11100001100000000001000000000001  //               ORR  R2 R0 R1
11100010100000000010000000000010  //               ADD  R2 R0 #2
11100010010100100000000000000000  //               SUBS R0 R2 #0
00001010000000000000000000000001             //               BEQ  TAG1
11100000000000100010000000000000  //               AND  R2 R2 R0
11100000000000100001000000000000  //               AND  R1 R2 R0
11100000100000011001000000000000  //  TAG1         ADD  R9 R1 R0
11100101100000001001000000001001  //               STR  R9 [R0, #9]
11100101100100000011000000001001  //               LDR  R3 [R0, #9]
11100000000000110010000000000010  //               AND  R2 R3 R2
```