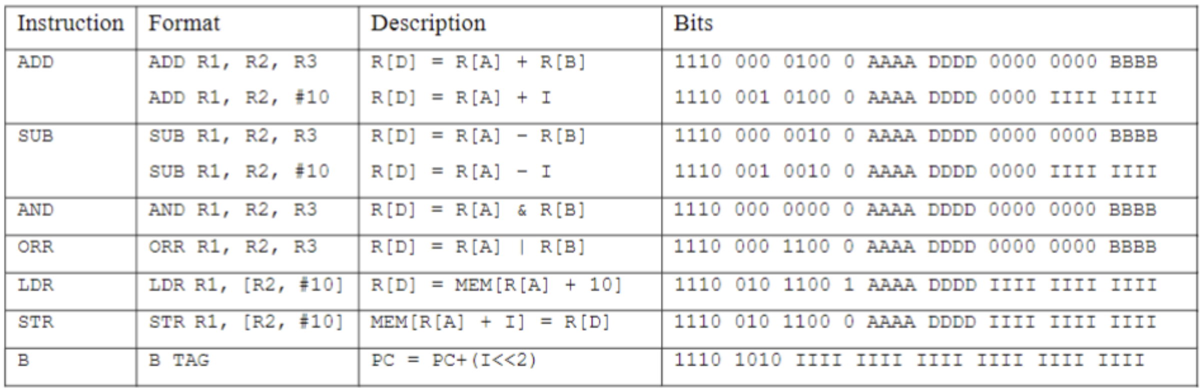
**Junchao Zhou, Chenhan Dai**
**EE469**
**April 20, 2023**
**Lab 2 Report**

## Procedure

The lab exercise involved two main tasks: integrating the register file and ALU into the datapath of an ARM processor and simulating the updated design, as well as implementing the CMP/SUBS instruction and conditional branching. The simulation step was crucial in ensuring the correctness of the design, while the addition of CMP/SUBS and conditional branching allowed for more complex operations and decision-making within the processor.

## Task 1:

The first step is to integrate the register file and ALU modules into the datapath of the ARM processor. Before doing this, it is important to fully understand the function of each signal and how they are represented (Table 1). And then we finished the codes of inserting the reg_file and ALU based on the given schematic (Figure 1 and 2).



| Instruction | Format | Description | Bits |
|---|---|---|---|
| ADD | ADD R1, R2, R3 | R[D] = R[A] + R[B] | 1110 000 0100 0 AAAA DDDD 0000 0000 BBBB |
| | ADD R1, R2, #10 | R[D] = R[A] + I | 1110 001 0100 0 AAAA DDDD 0000 IIII IIII |
| SUB | SUB R1, R2, R3 | R[D] = R[A] - R[B] | 1110 000 0010 0 AAAA DDDD 0000 0000 BBBB |
| | SUB R1, R2, #10 | R[D] = R[A] - I | 1110 001 0010 0 AAAA DDDD 0000 IIII IIII |
| AND | AND R1, R2, R3 | R[D] = R[A] & R[B] | 1110 000 0000 0 AAAA DDDD 0000 0000 BBBB |
| ORR | ORR R1, R2, R3 | R[D] = R[A] \| R[B] | 1110 000 1100 0 AAAA DDDD 0000 0000 BBBB |
| LDR | LDR R1, [R2, #10] | R[D] = MEM[R[A] + 10] | 1110 010 1100 1 AAAA DDDD IIII IIII IIII |
| STR | STR R1, [R2, #10] | MEM[R[A] + I] = R[D] | 1110 010 1100 0 AAAA DDDD IIII IIII IIII |
| B | B TAG | PC = PC+(I<<2) | 1110 1010 IIII IIII IIII IIII IIII IIII |

**Table 1: Processor Base Instruction Set**

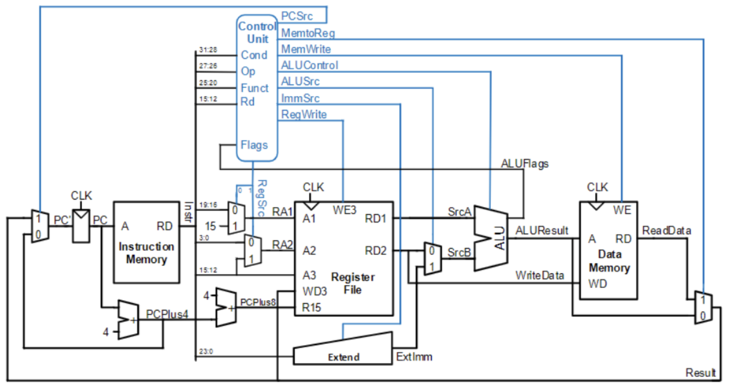
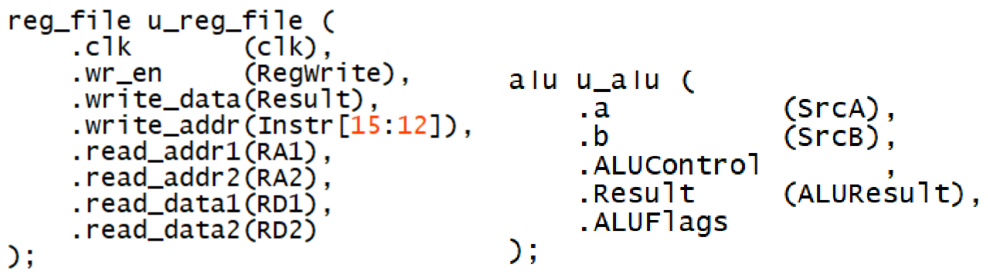


**Figure 1: Single-cycle ARM processor**



```
reg_file u_reg_file (
    .clk        (clk),
    .wr_en      (RegWrite),
    .write_data(Result),
    .write_addr(Instr[15:12]),
    .read_addr1(RA1),
    .read_addr2(RA2),
    .read_data1(RD1),
    .read_data2(RD2)
);
```

```
alu u_alu (
    .a          (SrcA),
    .b          (SrcB),
    .ALUControl
                ,
    .Result     (ALUResult),
    .ALUFlags
);
```

**Figure 2: The code for inserting reg_file and alu into arm**

**Task 2**:

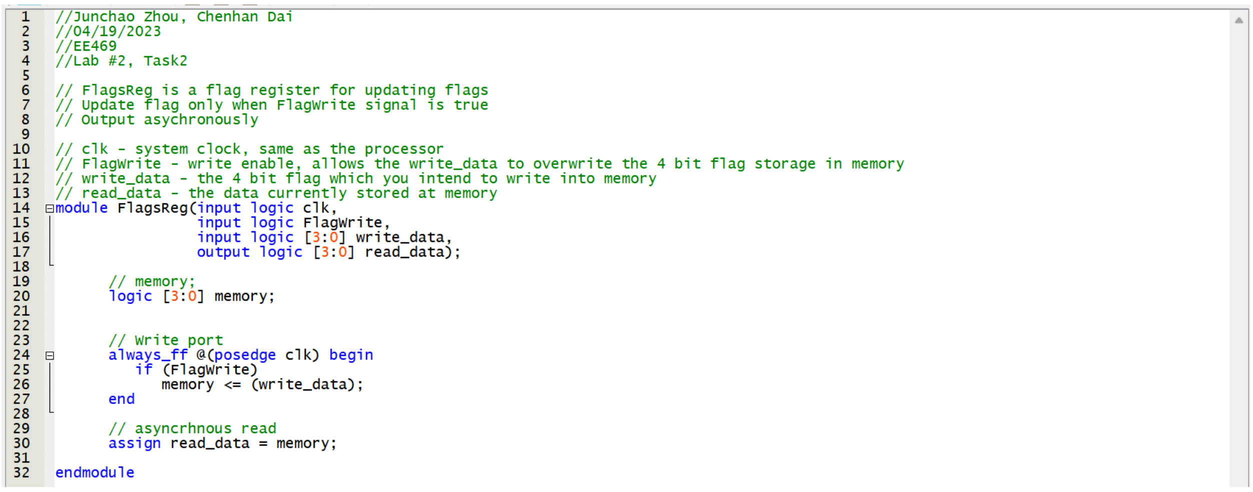
The second task requires us to introduce a new instruction called SUBS/CMP, which performs a subtraction and also saves the resulting flags in a special register called FlagsReg(Table 2). These flags can be used to evaluate condition logic for branching.

| Instruction | Format | Description | Bits |
|---|---|---|---|
| CMP | CMP R1, R2, R3 | R[D] = R[A] - R[B], Flag | 1110 000 0010 1 AAAA DDDD 0000 0000 BBBB |
| | CMP R1, R2, #10 | R[D] = R[A] - I, Flag | 1110 001 0010 1 AAAA DDDD 0000 IIII IIII |
| BXX | BXX TAG | PC = PC+(I<<2) if COND | COND 1010 IIII IIII IIII IIII IIII IIII |
| | B | Unconditional | 1110 |
| | BEQ | Equal | 0000 |
| | BNE | Not Equal | 0001 |
| | BGE | Greater or Equal | 1010 |
| | BGT | Greater | 1100 |
| | BLE | Less or Equal | 1101 |
| | BLT | Less | 1011 |

**Table 2: Added command and branching conditions**

To implement this, we added a new register called FlagsReg into our arm code. This register is used to store the flags produced by the recent CMP command, and the  always_ff block is being implemented to modify the register when a new control signal called FlagWrite is asserted(See Figure 3).

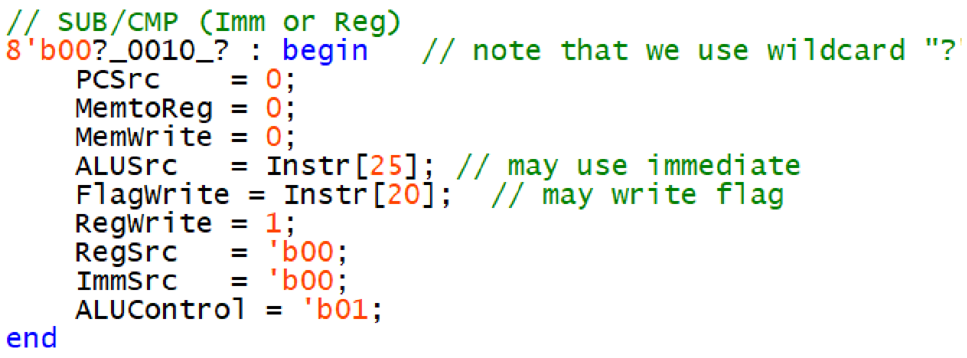
```
FlagsReg u_flags_reg (.clk, .FlagWrite(FlagWrite & CondEx), .write_data(ALUFlags), .read_data(StatusFlag));
```



```
//Junchao Zhou, Chenhan Dai
//04/19/2023
//EE469
//Lab #2, Task2

// FlagsReg is a flag register for updating flags
// Update flag only when FlagWrite signal is true
// Output asychronously

// clk - system clock, same as the processor
// FlagWrite - write enable, allows the write_data to overwrite the 4 bit flag storage in memory
// write_data - the 4 bit flag which you intend to write into memory
// read_data - the data currently stored at memory
module FlagsReg(input logic clk,
                input logic FlagWrite,
                input logic [3:0] write_data,
                output logic [3:0] read_data);

      // memory;
      logic [3:0] memory;


      // Write port
      always_ff @(posedge clk) begin
         if (FlagWrite)
            memory <= (write_data);
      end

      // asyncrhnous read
      assign read_data = memory;

endmodule
```

**Figure 3: The code for FlagsReg**

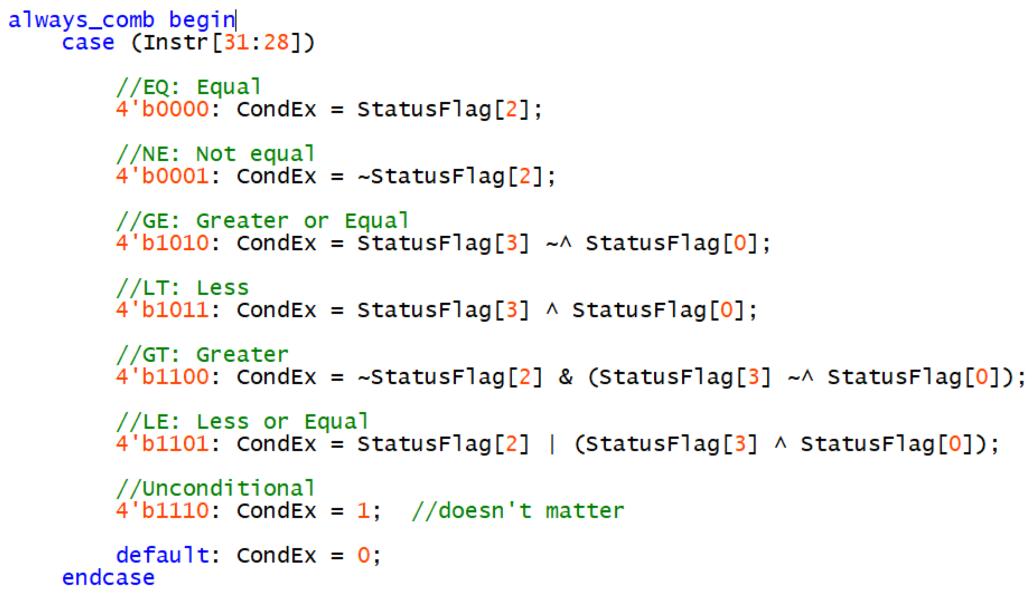
At the same time, we also improved the control by updating the SUB instruction(See Figure 4).



```
// SUB/CMP (Imm or Reg)
8'b00?_0010_? : begin   // note that we use wildcard "?
    PCSrc     = 0;
    MemtoReg = 0;
    MemWrite = 0;
    ALUSrc    = Instr[25]; // may use immediate
    FlagWrite = Instr[20];  // may write flag
    RegWrite = 1;
    RegSrc    = 'b00;
    ImmSrc    = 'b00;
    ALUControl = 'b01;
end
```

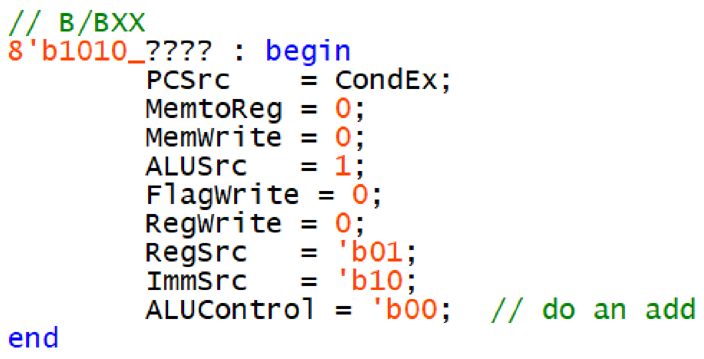
**Figure 4: the updated SUB/CMP instruction**

We also added logic to compute the conditions defined by EQ, NE, GE, GT, LE, and LT based on the saved flag bits. Hence, we updated the control for the branch instruction to check if the condition is valid before executing the branch or just ignoring the instruction completely (See Figure 5 and 6).



```
always_comb begin
    case (Instr[31:28])

        //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;  //doesn't matter

        default: CondEx = 0;
    endcase
```

**Figure 5: the logic to compute the conditions defined by EQ, NE, GE, LE and LT**



```
// B/BXX
8'b1010_???? : begin
          PCSrc     = CondEx;
          MemtoReg = 0;
          MemWrite = 0;
          ALUSrc    = 1;
          FlagWrite = 0;
          RegWrite = 0;
          RegSrc    = 'b01;
          ImmSrc    = 'b10;
          ALUControl = 'b00;   // do an add
end
```

**Figure 6: the updated B/BXX instruction**

**Result**

**Task 1:**

The process of adding the register file and ALU to the ARM code has been completed, and a simulation of the processor has been run using ModelSim. The instructions for the simulation are read from the memfile.dat file(Figure 7).

```
MAIN      ADD R0, R15, #0
          SUB R1, R0, R0
          ADD R2, R1, #10
          ADD R3, R0, R2
          SUB R4, R2, #3
          SUB R5, R3, R4
          ORR R6, R4, R5
          AND R7, R6, R5
          STR R7, [R1, #0]
          B SKIP
          STR R1, [R1, #0]
          B LOOP
SKIP      LDR R8, [R1, #0]
LOOP      B LOOP
```

**Figure 7: Base Processor Testing Program "memfile.dat"**

The simulation has generated a waveform, which shows the behavior of various signals in the system. The waveform contains the following elements in order from top to bottom: clock signal (clk), reset signal (rst), program counter (PC), instruction (Instr), ALU result (ALUResult), data to be written (WriteData), memory write signal (MemWrite), and data read from memory (ReadData). For better illustration, we set PC as decimal to compare the steps in .dat file, memWrite and RegWrite are set to binay, and others are set to hexdecimal.  The screen shot of the waveform can be seen below in Figure 8.

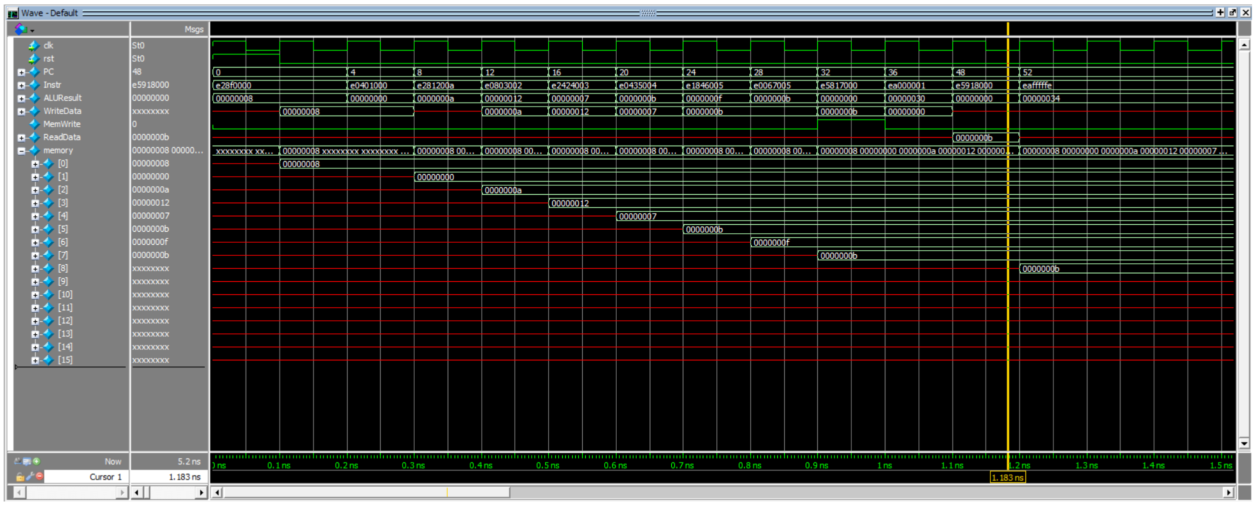


**Figure 8: The waveform generated by the Processor.**

The following is the completed version of Table 3.

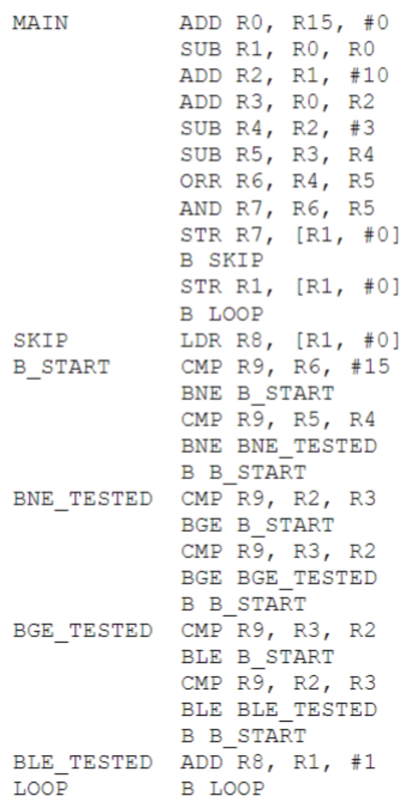
| Cycle | PC | Instr | SrcA | SrcB | ALUResult | WriteData | Read Data | Mem Write | RegWrite | Result |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 00 | ADD R0, R15, #0 | 8 | 0 | 8 | Don't Care | X | 0 | 1 | 8 |
| 2 | 04 | SUB R1, R0, R0 | 8 | 8 | 0 | 8 | X | 0 | 1 | 0 |
| 3 | 08 | ADD R2, R1, #10 | 0 | A | A | Don't Care | X | 0 | 1 | A |
| 4 | 12 | ADD R3, R0, R2 | 8 | A | 12 | A | X | 0 | 1 | 12 |
| 5 | 16 | SUB R4, R2, #3 | A | 3 | 7 | 12 | X | 0 | 1 | 7 |
| 6 | 20 | SUB R5, R3, R4 | 12 | 7 | B | 7 | X | 0 | 1 | B |
| 7 | 24 | ORR R6, R4, R5 | 7 | B | F | B | X | 0 | 1 | F |
| 8 | 28 | AND R7, R6, R5 | F | B | B | B | X | 0 | 1 | B |
| 9 | 32 | STR R7, [R1, #0] | 0 | 0 | 0 | B | X | 1 | 0 | 0 |
| 10 | 36 | B SKIP(B # 1) | 2C | 4 | 30 | 0 | X | 0 | 0 | 30 |
| 11 | 48 | LDR R8, [R1, #0] | 0 | 0 | B | Don't Care | B | 0 | 1 | B |
| 12 | 52 | B LOOP(B # -1) | 3C | fffffff8 | 34 | Don't Care | X | 0 | 0 | 34 |
| 13 | 52 | B LOOP # -1 | 3C | fffffff8 | 34 | Don't Care | X | 0 | 0 | 34 |
| 14 | 52 | B LOOP # -1 | 3C | fffffff8 | 34 | Don't Care | X | 0 | 0 | 34 |
| 15 | 52 | B LOOP # -1 | 3C | fffffff8 | 34 | Don't Care | X | 0 | 0 | 34 |
| 16 | 52 | B LOOP # -1 | 3C | fffffff8 | 34 | Don't Care | X | 0 | 0 | 34 |
| 17 | 52 | B LOOP # -1 | 3C | fffffff8 | 34 | Don't Care | X | 0 | 0 | 34 |
| 18 | 52 | B LOOP # -1 | 3C | fffffff8 | 34 | Don't Care | X | 0 | 0 | 34 |
| 19 | 52 | B LOOP # -1 | 3C | fffffff8 | 34 | Don't Care | X | 0 | 0 | 34 |

**Table 3. First nineteen cycles of executing memfile.dat**

**Task 2**:

The process of adding the CMP/SUBS and Conditional Branching has been completed, and a simulation of the processor has been run using ModelSim. The instructions for the simulation are read from the memfile2.dat file(Figure 9).  And we wrote down the expected PC sequence on the left hand side.

| | |
|---|---|
| 0 | ADD R0, R15, #0 |
| 4 | SUB R1, R0, R0 |
| 8 | ADD R2, R1, #10 |
| 12 | ADD R3, R0, R2 |
| 16 | SUB R4, R2, #3 |
| 20 | SUB R5, R3, R4 |
| 24 | ORR R6, R4, R5 |
| 28 | AND R7, R6, R5 |
| 32 | STR R7, [R1, #0] |
| 36 | B SKIP |
| 48 | ***SKIP*** LDR R8, [R1, #0] |
| 52 | ***B_START*** CMP R9, R6, #15 |
| 56 | BNE B_START |
| 60 | CMP R9, R5, R4 |
| 64 | BNE BNE_TESTED |
| 72 | ***BNE_TESTED*** CMP R9, R2, R3 |
| 76 | BGE B_START |
| 80 | CMP R9, R3, R2 |
| 84 | BGE BGE_TESTED |
| 92 | ***BGE_TESTED*** CMP R9, R3, R2 |
| 96 | BLE B_START |
| 100 | CMP R9, R2, R3 |
| 104 | BLE BLE_TESTED |
| 112 | ***BLE_TESTED*** ADD R8, R1, #1 |
| 116 | ***LOOP*** B LOOP |



```
MAIN        ADD R0, R15, #0
            SUB R1, R0, R0
            ADD R2, R1, #10
            ADD R3, R0, R2
            SUB R4, R2, #3
            SUB R5, R3, R4
            ORR R6, R4, R5
            AND R7, R6, R5
            STR R7, [R1, #0]
            B SKIP
            STR R1, [R1, #0]
            B LOOP
SKIP        LDR R8, [R1, #0]
B_START     CMP R9, R6, #15
            BNE B_START
            CMP R9, R5, R4
            BNE BNE_TESTED
            B B_START
BNE_TESTED  CMP R9, R2, R3
            BGE B_START
            CMP R9, R3, R2
            BGE BGE_TESTED
            B B_START
BGE_TESTED  CMP R9, R3, R2
            BLE B_START
            CMP R9, R2, R3
            BLE BLE_TESTED
            B B_START
BLE_TESTED  ADD R8, R1, #1
LOOP        B LOOP
```

**Figure 9: Updated Processor Testing Program "memfile2.dat"(On right)**

The simulation has generated a waveform, which shows the behavior of various signals in the system. The waveform contains the following elements in order from top to bottom: clock signal (clk), reset signal (rst), program counter (PC), instruction (Instr), ALU result (ALUResult), data to be written (WriteData), memory write signal (MemWrite), and data read from memory (ReadData). The screen shot of the waveform can be seen below in Figure 10. Due to the large dimensions of the screenshot, it has been divided into two sections. The upper portion depicts information from PC0 to PC92, while the lower section displays data from PC48 to PC116.

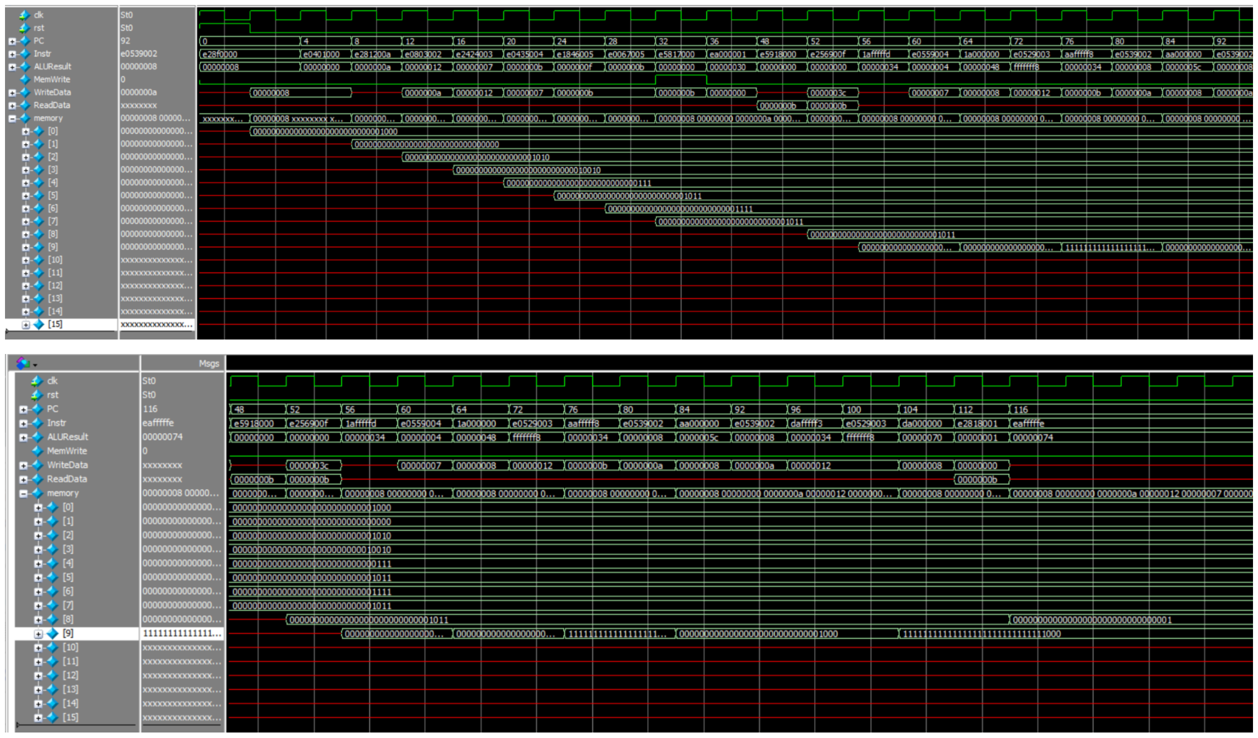
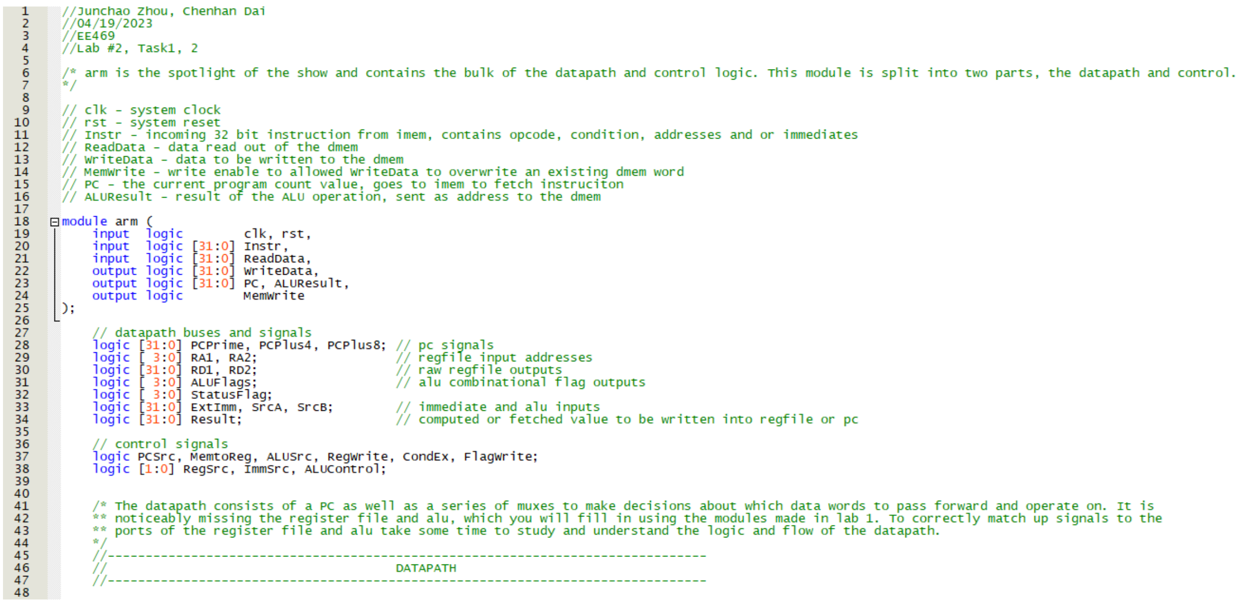


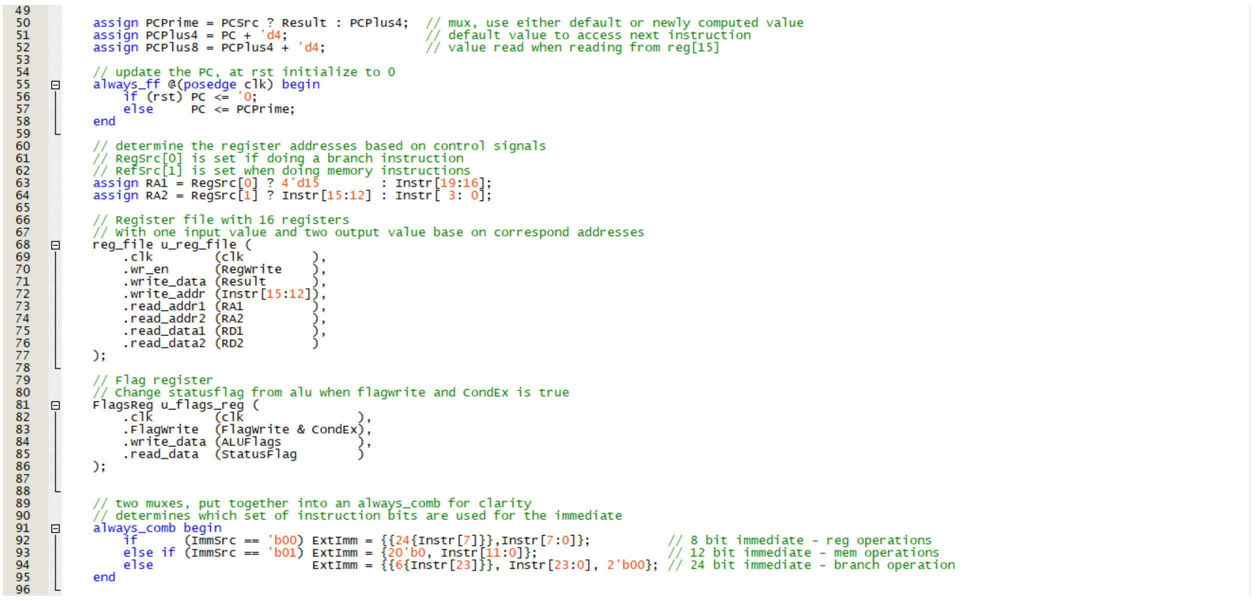
**Figure 10: The waveform generated by the Processor**

## Appendix: SystemVerilog code

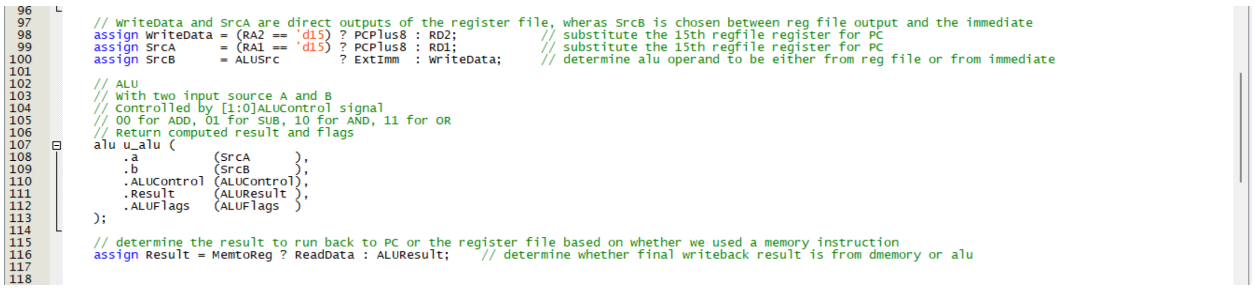
### 1) arm.sv



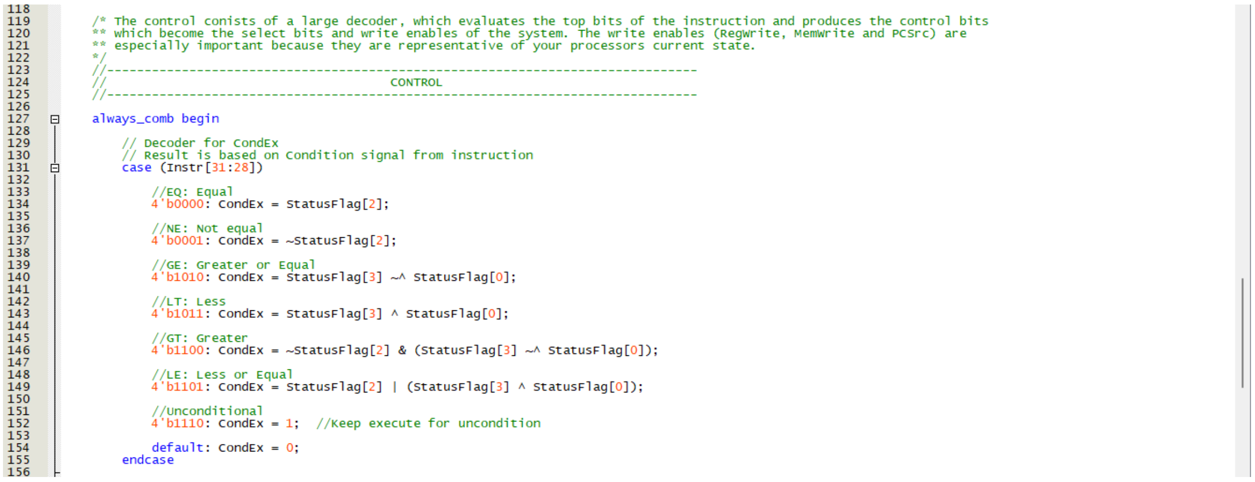
```
//Junchao Zhou, Chenhan Dai
//04/19/2023
//EE469
//Lab #2, Task1, 2

/* 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, PCPlus8; // pc signals
    logic [ 3:0] RA1, RA2;                  // regfile input addresses
    logic [31:0] RD1, RD2;                  // raw regfile outputs
    logic [ 3:0] ALUFlags;                  // alu combinational flag outputs
    logic [ 3:0] StatusFlag;
    logic [31:0] ExtImm, SrcA, SrcB;        // immediate and alu inputs
    logic [31:0] Result;                    // computed or fetched value to be written into regfile or pc

    // control signals
    logic PCSrc, MemtoReg, ALUSrc, RegWrite, CondEx, FlagWrite;
    logic [1:0] RegSrc, ImmSrc, ALUControl;


    /* 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
    //-------------------------------------------------------------------------------
```



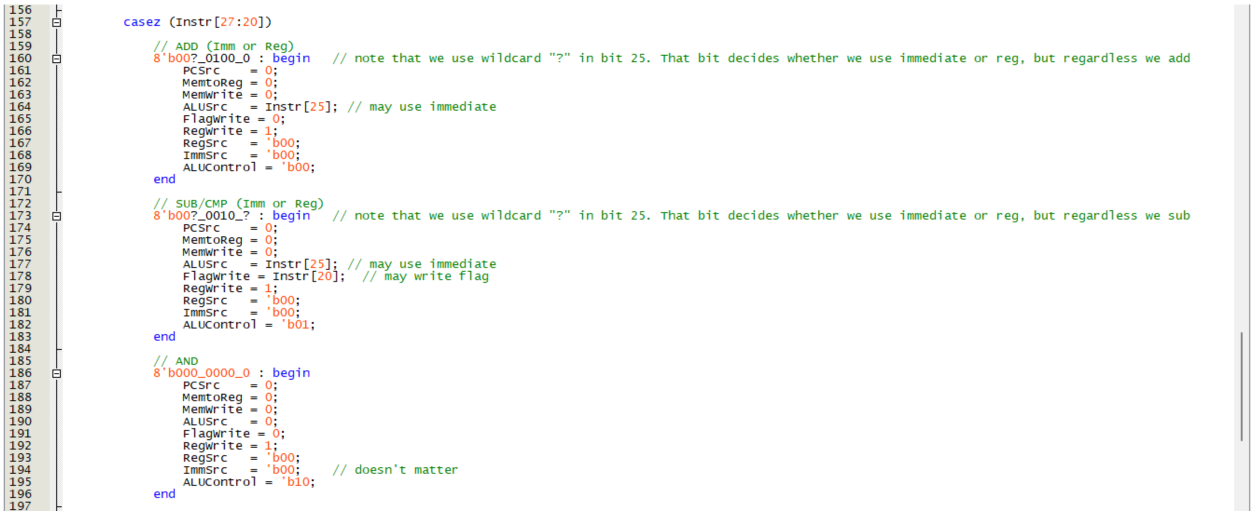
```

    assign PCPrime = PCSrc ? Result : PCPlus4;  // mux, use either default or newly computed value
    assign PCPlus4 = PC + 'd4;                  // default value to access next instruction
    assign PCPlus8 = PCPlus4 + 'd4;             // value read when reading from reg[15]

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

    // 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 RA1 = RegSrc[0] ? 4'd15         : Instr[19:16];
    assign RA2 = RegSrc[1] ? Instr[15:12] : Instr[ 3: 0];

    // Register file with 16 registers
    // With one input value and two output value base on correspond addresses
    reg_file u_reg_file (
        .clk        (clk         ),
        .wr_en      (RegWrite    ),
        .write_data (Result      ),
        .write_addr (Instr[15:12]),
        .read_addr1 (RA1         ),
        .read_addr2 (RA2         ),
        .read_data1 (RD1         ),
        .read_data2 (RD2         )
    );

    // Flag register
    // Change statusflag from alu when flagwrite and CondEx is true
    FlagsReg u_flags_reg (
        .clk        (clk                ),
        .FlagWrite  (FlagWrite & 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) ExtImm = {{24{Instr[7]}},Instr[7:0]};          // 8 bit immediate - reg operations
        else if (ImmSrc == 'b01) ExtImm = {20'b0, Instr[11:0]};                 // 12 bit immediate - mem operations
        else                     ExtImm = {{6{Instr[23]}}, Instr[23:0], 2'b00}; // 24 bit immediate - branch operation
    end
```



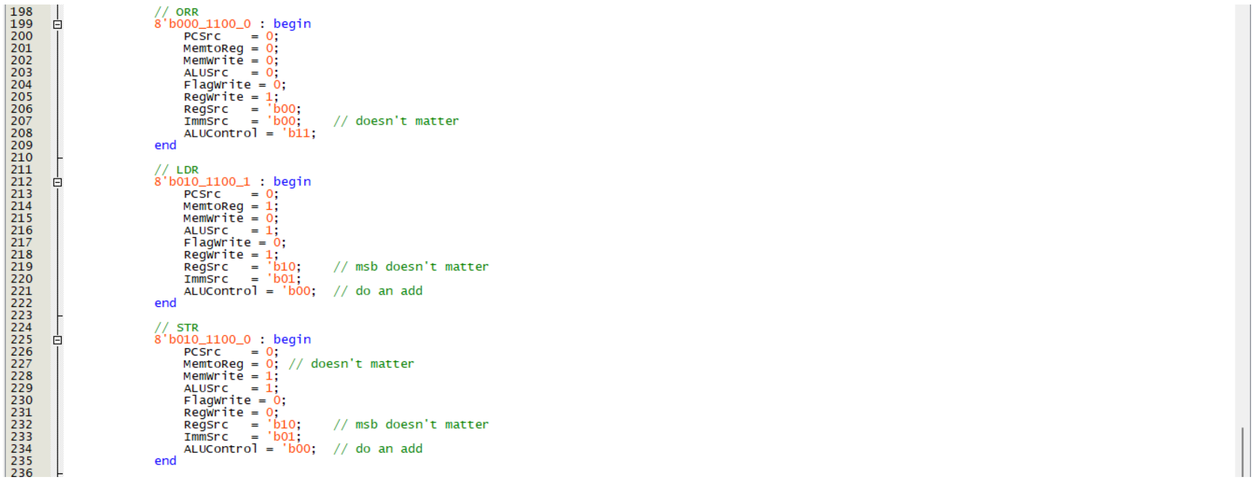
```

    // WriteData and SrcA are direct outputs of the register file, wheras SrcB is chosen between reg file output and the immediate
    assign WriteData = (RA2 == 'd15) ? PCPlus8 : RD2;           // substitute the 15th regfile register for PC
    assign SrcA      = (RA1 == 'd15) ? PCPlus8 : RD1;           // substitute the 15th regfile register for PC
    assign SrcB      = ALUSrc        ? ExtImm  : WriteData;     // 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 (ALUControl),
        .Result     (ALUResult ),
        .ALUFlags   (ALUFlags  )
    );

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

```



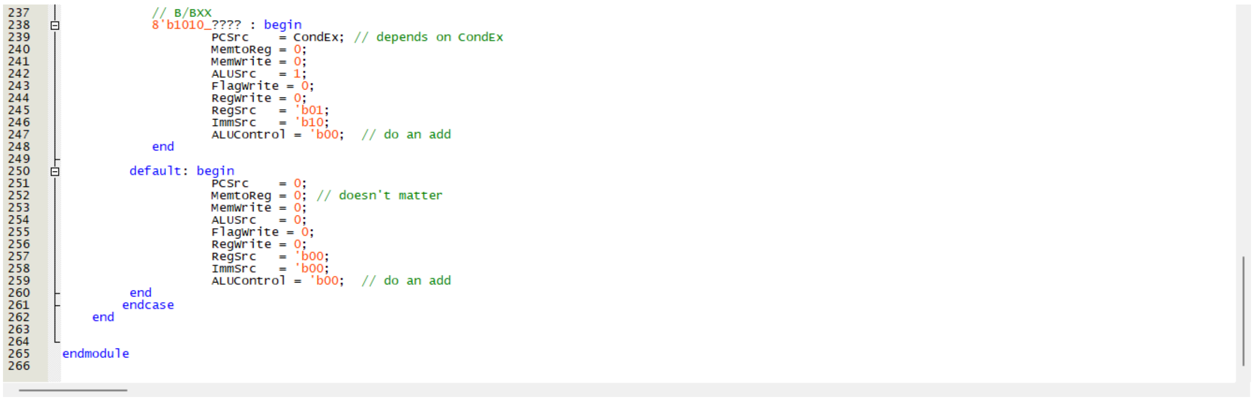
```
/* The control consists of a large decoder, which evaluates the top bits of the instruction and produces the control bits
** which become the select bits and write enables of the system. The write enables (RegWrite, MemWrite and PCSrc) are
** especially important because they are representative of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

always_comb begin

    // Decoder for CondEx
    // Result is based on Condition signal from instruction
    case (Instr[31:28])

        //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 (Instr[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
            PCSrc    = 0;
            MemtoReg = 0;
            MemWrite = 0;
            ALUSrc   = Instr[25]; // may use immediate
            FlagWrite = 0;
            RegWrite = 1;
            RegSrc   = 'b00;
            ImmSrc   = 'b00;
            ALUControl = '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
            PCSrc    = 0;
            MemtoReg = 0;
            MemWrite = 0;
            ALUSrc   = Instr[25]; // may use immediate
            FlagWrite = Instr[20];  // may write flag
            RegWrite = 1;
            RegSrc   = 'b00;
            ImmSrc   = 'b00;
            ALUControl = 'b01;
        end

        // AND
        8'b000_0000_0 : begin
            PCSrc    = 0;
            MemtoReg = 0;
            MemWrite = 0;
            ALUSrc   = 0;
            FlagWrite = 0;
            RegWrite = 1;
            RegSrc   = 'b00;
            ImmSrc   = 'b00;    // doesn't matter
            ALUControl = 'b10;
        end
```

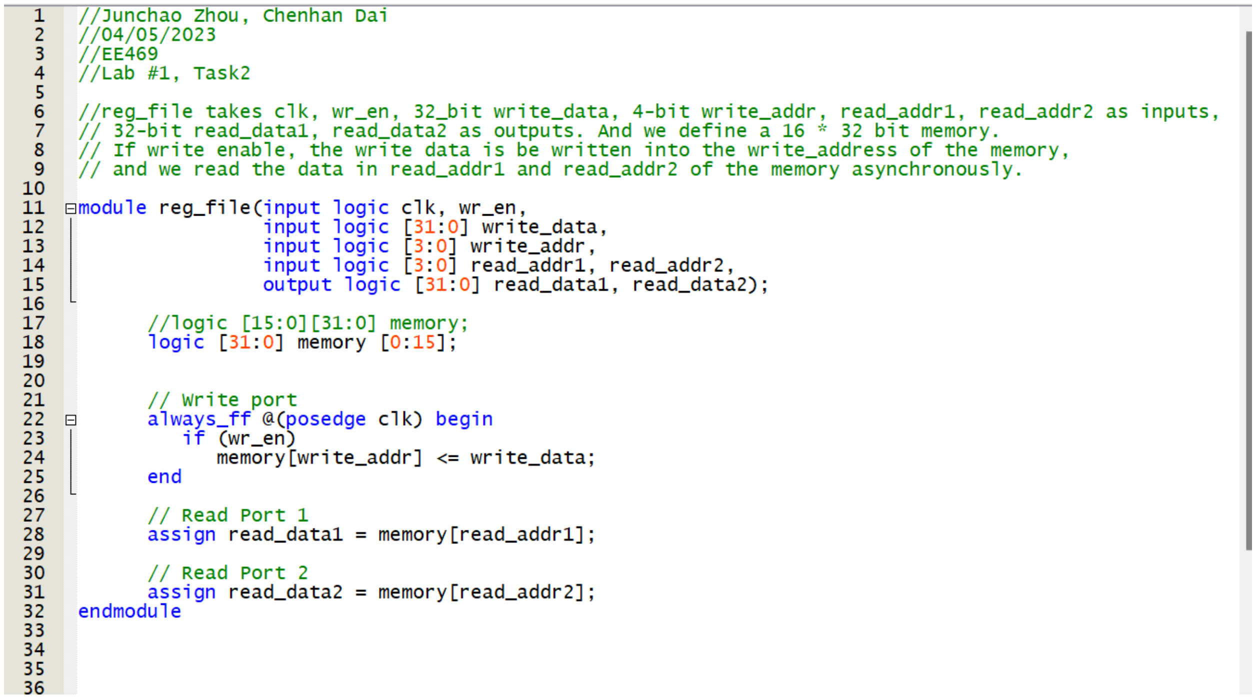


```
            // ORR
            8'b000_1100_0 : begin
                PCSrc    = 0;
                MemtoReg = 0;
                MemWrite = 0;
                ALUSrc   = 0;
                FlagWrite = 0;
                RegWrite = 1;
                RegSrc   = 'b00;
                ImmSrc   = 'b00;    // doesn't matter
                ALUControl = 'b11;
            end

            // LDR
            8'b010_1100_1 : begin
                PCSrc    = 0;
                MemtoReg = 1;
                MemWrite = 0;
                ALUSrc   = 1;
                FlagWrite = 0;
                RegWrite = 1;
                RegSrc   = 'b10;    // msb doesn't matter
                ImmSrc   = 'b01;
                ALUControl = 'b00;  // do an add
            end

            // STR
            8'b010_1100_0 : begin
                PCSrc    = 0;
                MemtoReg = 0; // doesn't matter
                MemWrite = 1;
                ALUSrc   = 1;
                FlagWrite = 0;
                RegWrite = 0;
                RegSrc   = 'b10;    // msb doesn't matter
                ImmSrc   = 'b01;
                ALUControl = 'b00;  // do an add
            end
```

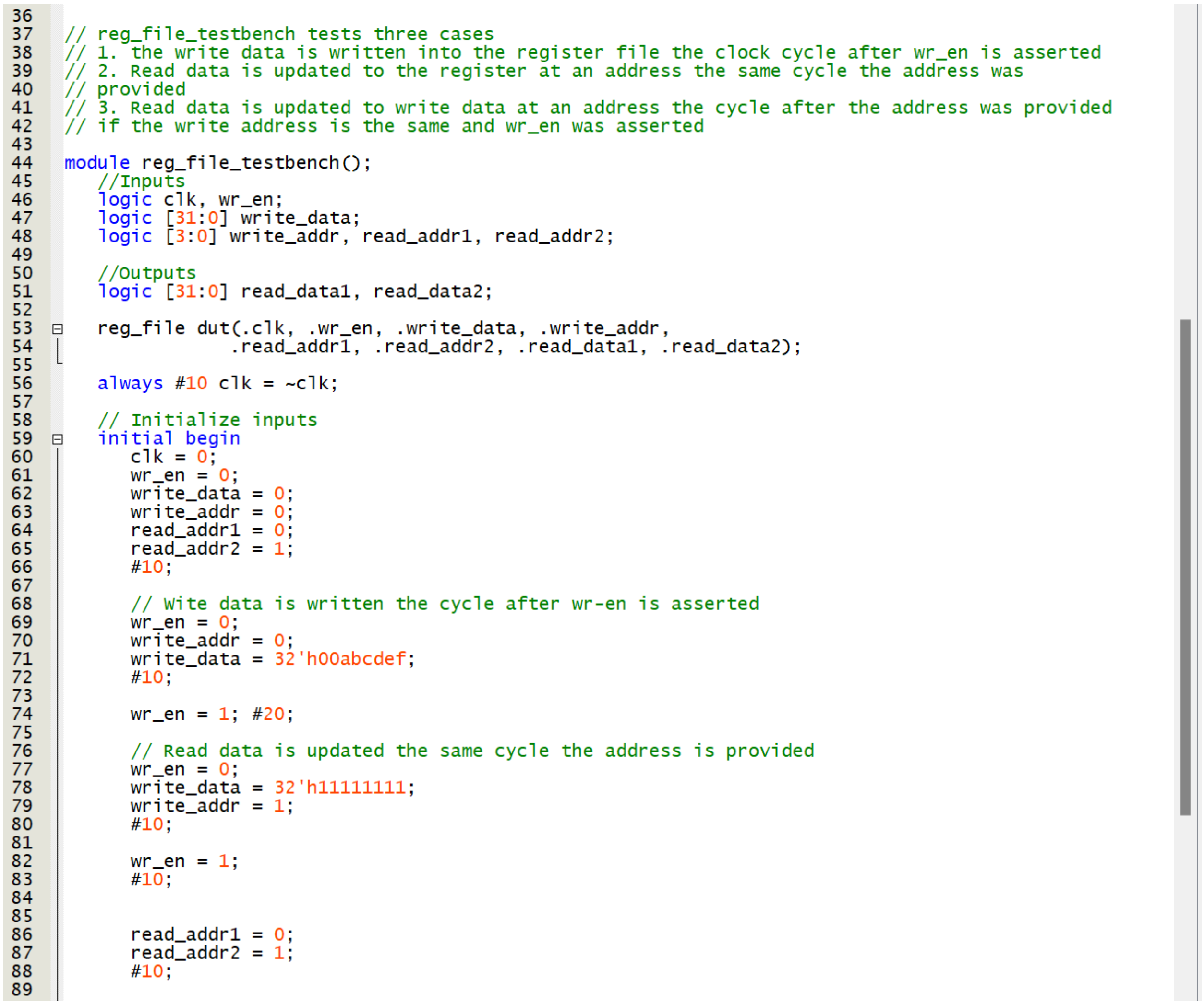


```
            // B/BXX
            8'b1010_???? : begin
                    PCSrc    = CondEx; // depends on CondEx
                    MemtoReg = 0;
                    MemWrite = 0;
                    ALUSrc   = 1;
                    FlagWrite = 0;
                    RegWrite = 0;
                    RegSrc   = 'b01;
                    ImmSrc   = 'b10;
                    ALUControl = 'b00;  // do an add
            end

         default: begin
                    PCSrc    = 0;
                    MemtoReg = 0; // doesn't matter
                    MemWrite = 0;
                    ALUSrc   = 0;
                    FlagWrite = 0;
                    RegWrite = 0;
                    RegSrc   = 'b00;
                    ImmSrc   = 'b00;
                    ALUControl = 'b00;  // do an add
         end
        endcase
    end

endmodule
```

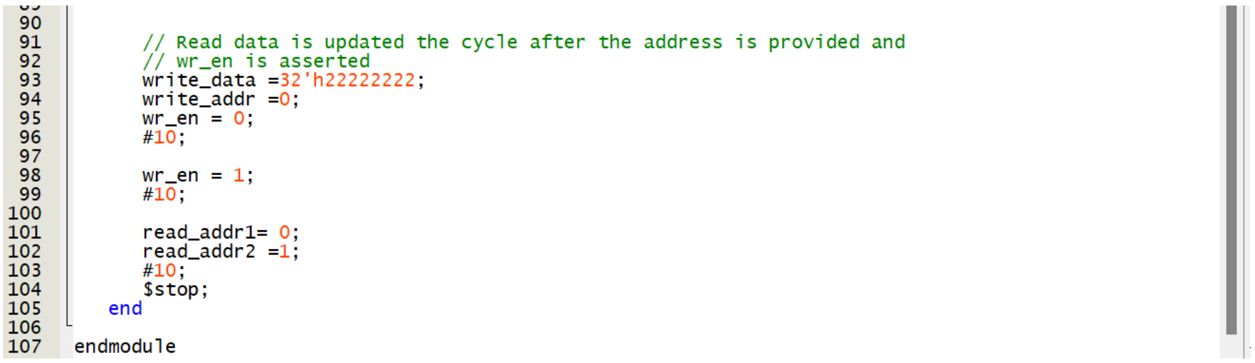
## 2) reg_file.sv



```
//Junchao Zhou, Chenhan Dai
//04/05/2023
//EE469
//Lab #1, Task2

//reg_file takes clk, wr_en, 32_bit write_data, 4-bit write_addr, read_addr1, read_addr2 as inputs,
// 32-bit read_data1, read_data2 as outputs. And we define a 16 * 32 bit memory.
// If write enable, the write data is be written into the write_address of the memory,
// and we read the data in read_addr1 and read_addr2 of the memory asynchronously.

module reg_file(input logic clk, wr_en,
                input logic [31:0] write_data,
                input logic [3:0] write_addr,
                input logic [3:0] read_addr1, read_addr2,
                output logic [31:0] read_data1, read_data2);

      //logic [15:0][31:0] memory;
      logic [31:0] memory [0:15];


      // Write port
      always_ff @(posedge clk) begin
         if (wr_en)
            memory[write_addr] <= write_data;
      end

      // Read Port 1
      assign read_data1 = memory[read_addr1];

      // Read Port 2
      assign read_data2 = memory[read_addr2];
endmodule
```

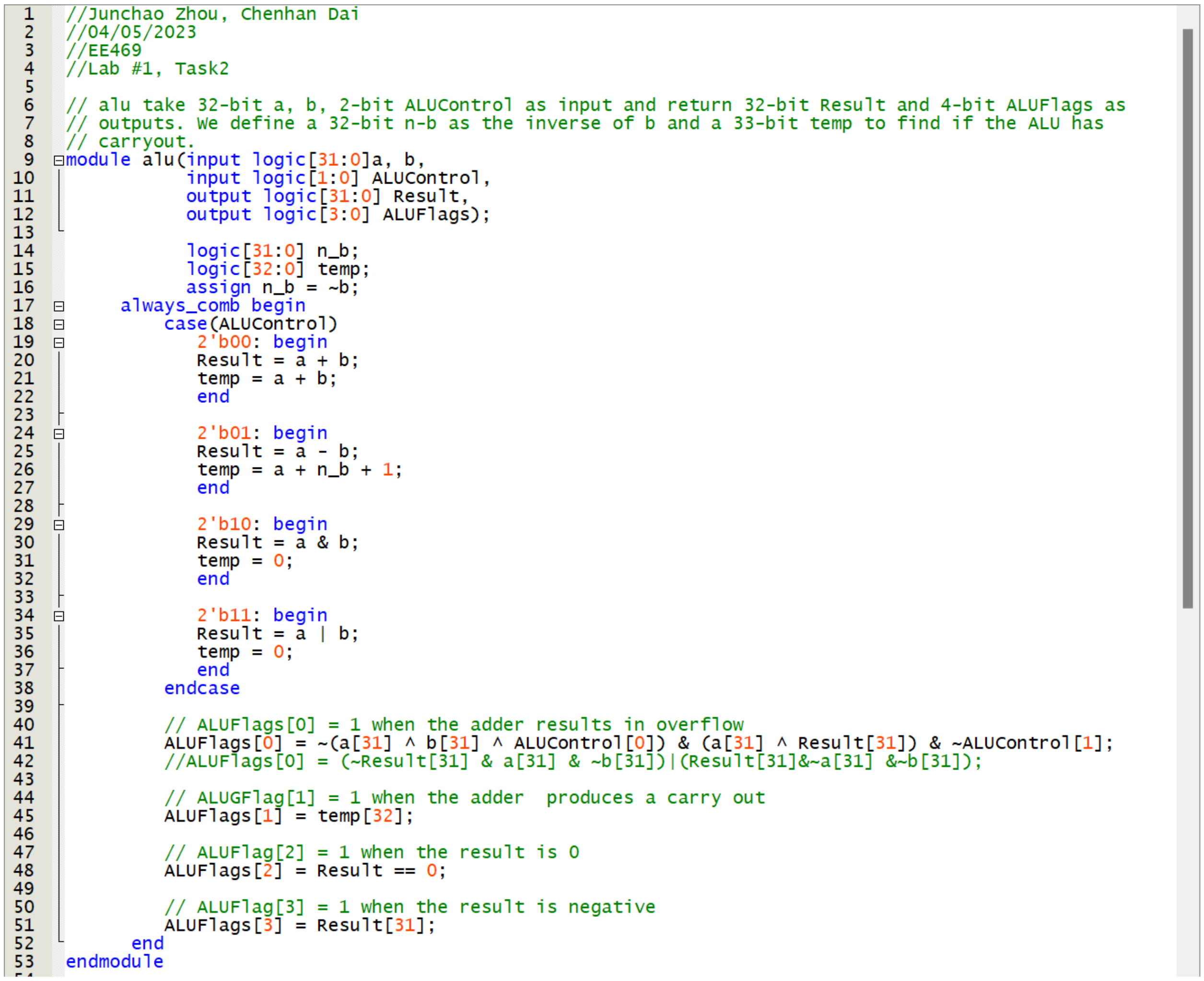


```
// reg_file_testbench tests three cases
// 1. the write data is written into the register file the clock cycle after wr_en is asserted
// 2. Read data is updated to the register at an address the same cycle the address was
// provided
// 3. Read data is updated to write data at an address the cycle after the address was provided
// if the write address is the same and wr_en was asserted

module reg_file_testbench();
   //Inputs
   logic clk, wr_en;
   logic [31:0] write_data;
   logic [3:0] write_addr, read_addr1, read_addr2;

   //Outputs
   logic [31:0] read_data1, read_data2;

   reg_file dut(.clk, .wr_en, .write_data, .write_addr,
                .read_addr1, .read_addr2, .read_data1, .read_data2);

   always #10 clk = ~clk;

   // Initialize inputs
   initial begin
      clk = 0;
      wr_en = 0;
      write_data = 0;
      write_addr = 0;
      read_addr1 = 0;
      read_addr2 = 1;
      #10;

      // Wite data is written the cycle after wr-en is asserted
      wr_en = 0;
      write_addr = 0;
      write_data = 32'h00abcdef;
      #10;

      wr_en = 1; #20;

      // Read data is updated the same cycle the address is provided
      wr_en = 0;
      write_data = 32'h11111111;
      write_addr = 1;
      #10;

      wr_en = 1;
      #10;


      read_addr1 = 0;
      read_addr2 = 1;
      #10;

```

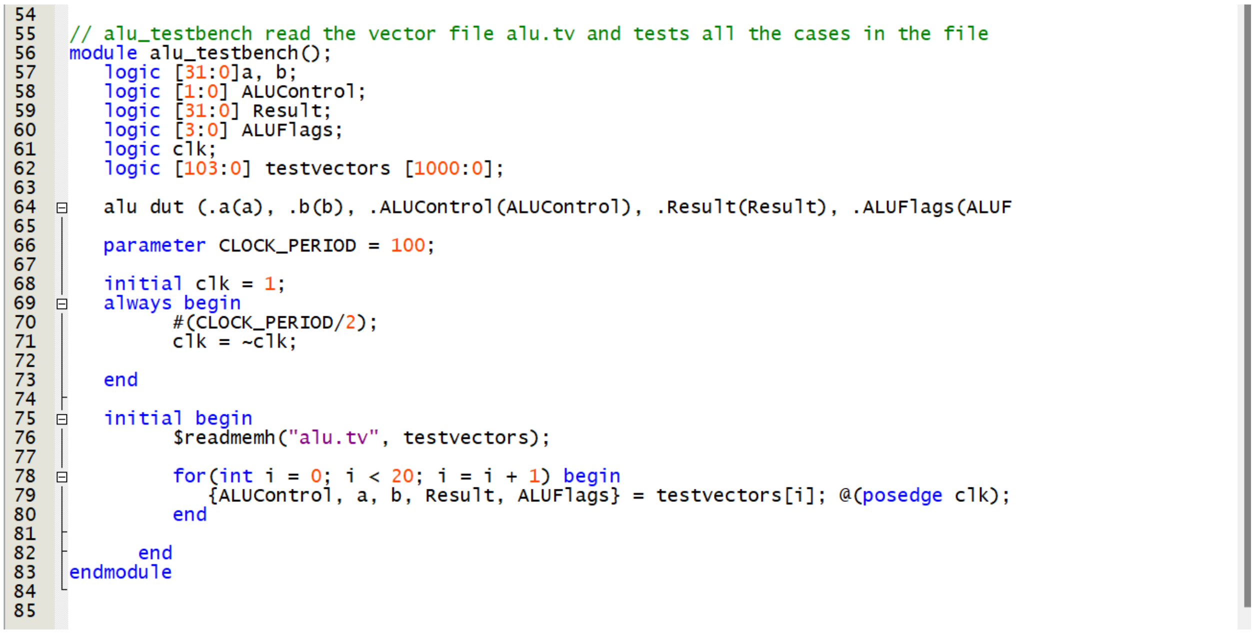


```

      // Read data is updated the cycle after the address is provided and
      // wr_en is asserted
      write_data =32'h22222222;
      write_addr =0;
      wr_en = 0;
      #10;

      wr_en = 1;
      #10;

      read_addr1= 0;
      read_addr2 =1;
      #10;
      $stop;
   end

endmodule
```

## 3) alu.sv

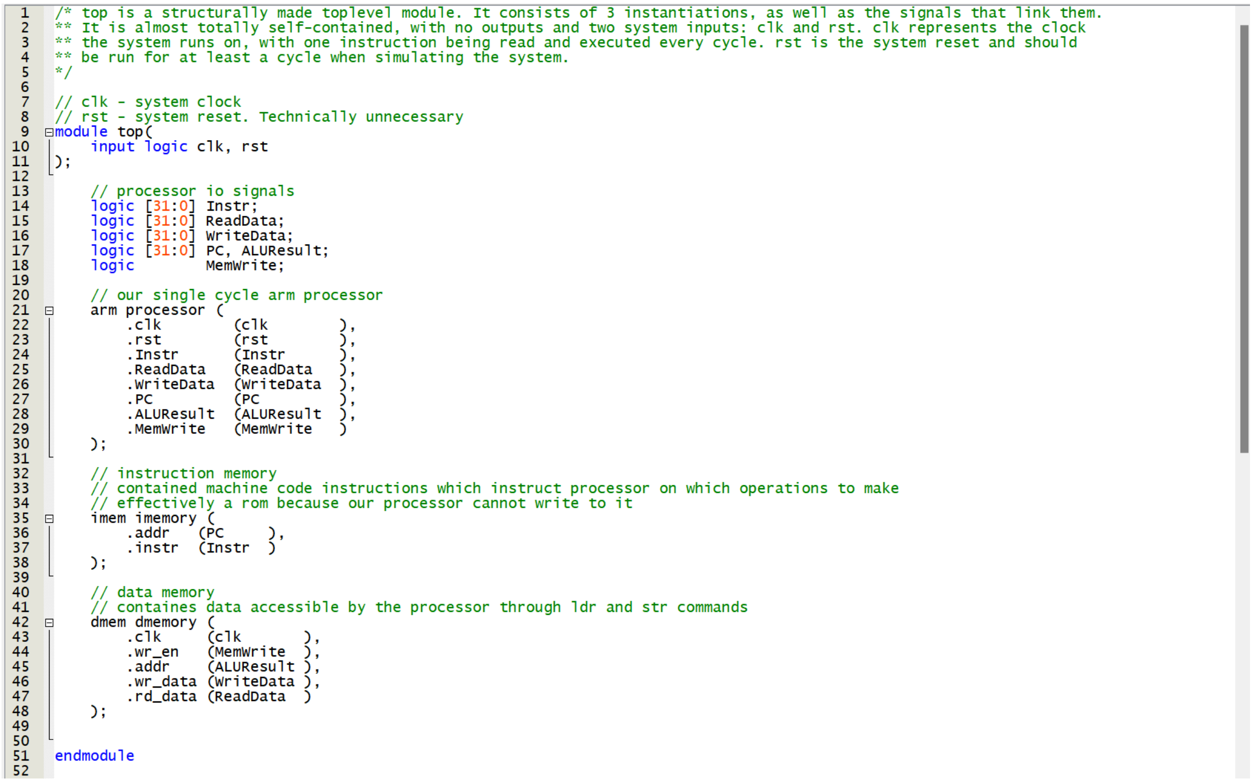


```
//Junchao Zhou, Chenhan Dai
//04/05/2023
//EE469
//Lab #1, Task2

// alu take 32-bit a, b, 2-bit ALUControl as input and return 32-bit Result and 4-bit ALUFlags as
// outputs. We define a 32-bit n-b as the inverse of b and a 33-bit temp to find if the ALU has
// carryout.
module alu(input logic[31:0]a, b,
           input logic[1:0] ALUControl,
           output logic[31:0] Result,
           output logic[3:0] ALUFlags);

           logic[31:0] n_b;
           logic[32:0] temp;
           assign n_b = ~b;
     always_comb begin
         case(ALUControl)
            2'b00: begin
            Result = a + b;
            temp = a + b;
            end

            2'b01: begin
            Result = a - b;
            temp = a + n_b + 1;
            end

            2'b10: begin
            Result = a & b;
            temp = 0;
            end

            2'b11: begin
            Result = a | b;
            temp = 0;
            end
         endcase

         // ALUFlags[0] = 1 when the adder results in overflow
         ALUFlags[0] = ~(a[31] ^ b[31] ^ ALUControl[0]) & (a[31] ^ Result[31]) & ~ALUControl[1];
         //ALUFlags[0] = (~Result[31] & a[31] & ~b[31])|(Result[31]&~a[31] &~b[31]);

         // ALUGFlag[1] = 1 when the adder  produces a carry out
         ALUFlags[1] = temp[32];

         // ALUFlag[2] = 1 when the result is 0
         ALUFlags[2] = Result == 0;

         // ALUFlag[3] = 1 when the result is negative
         ALUFlags[3] = Result[31];
      end
endmodule
```

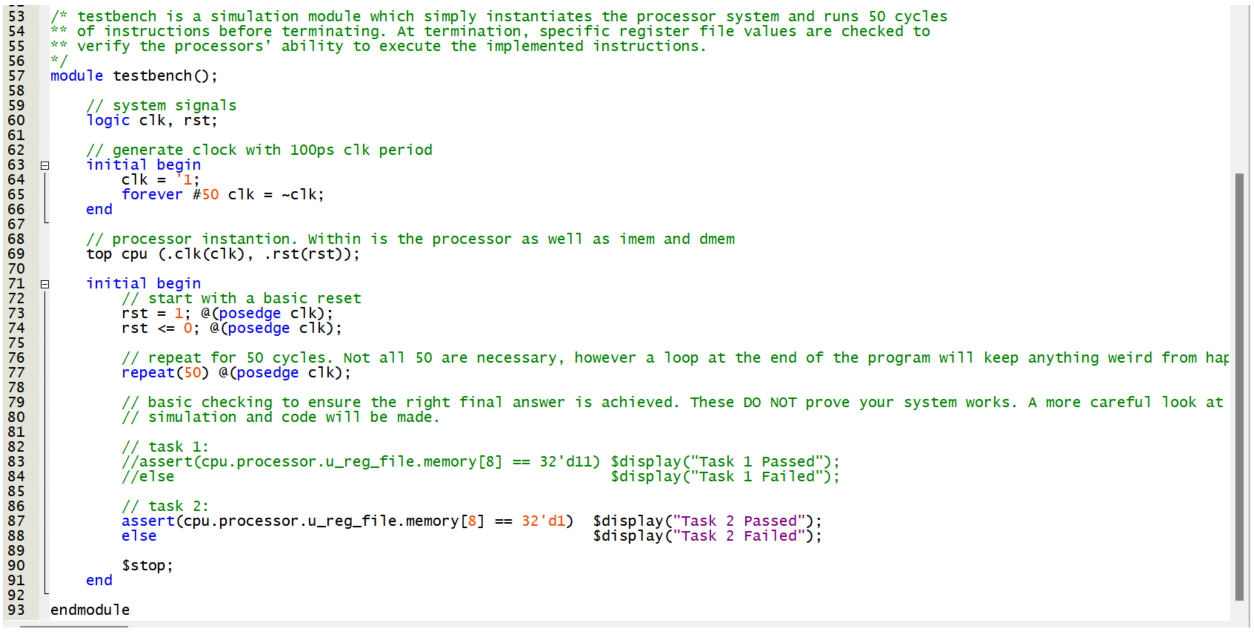


```

// alu_testbench read the vector file alu.tv and tests all the cases in the file
module alu_testbench();
   logic [31:0]a, b;
   logic [1:0] ALUControl;
   logic [31:0] Result;
   logic [3:0] ALUFlags;
   logic clk;
   logic [103:0] testvectors [1000:0];

   alu dut (.a(a), .b(b), .ALUControl(ALUControl), .Result(Result), .ALUFlags(ALUF

   parameter CLOCK_PERIOD = 100;

   initial clk = 1;
   always begin
         #(CLOCK_PERIOD/2);
         clk = ~clk;

   end

   initial begin
         $readmemh("alu.tv", testvectors);

         for(int i = 0; i < 20; i = i + 1) begin
            {ALUControl, a, b, Result, ALUFlags} = testvectors[i]; @(posedge clk);
         end

      end
endmodule

```

## 4) top.sv

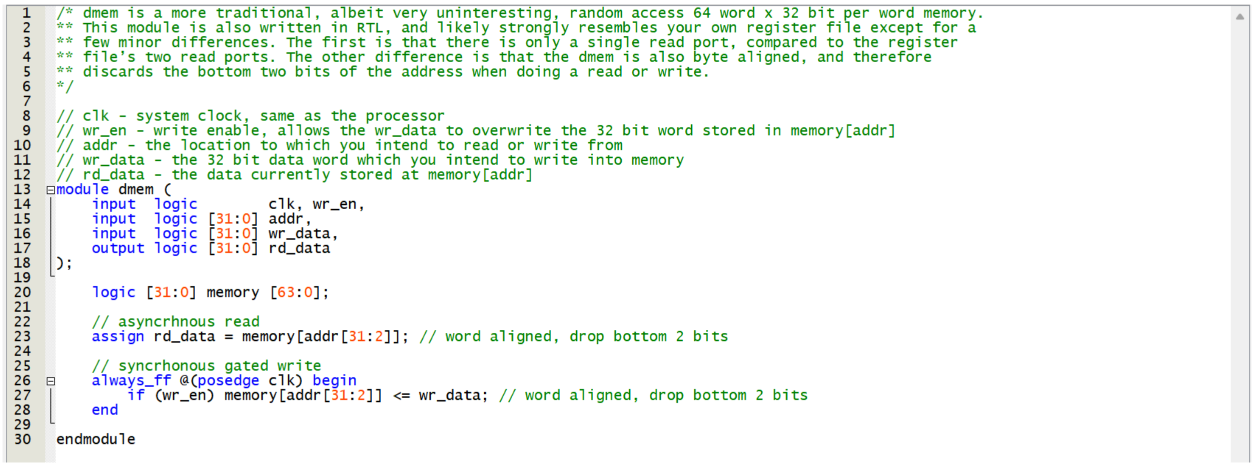


```systemverilog
/* top is a structurally made toplevel module. It consists of 3 instantiations, as well as the signals that link them.
** It is almost totally self-contained, with no outputs and two system inputs: clk and rst. clk represents the clock
** the system runs on, with one instruction being read and executed every cycle. rst is the system reset and should
** be run for at least a cycle when simulating the system.
*/

// clk - system clock
// rst - system reset. Technically unnecessary
module top(
    input logic clk, rst
);

    // processor io signals
    logic [31:0] Instr;
    logic [31:0] ReadData;
    logic [31:0] WriteData;
    logic [31:0] PC, ALUResult;
    logic        MemWrite;

    // our single cycle arm processor
    arm processor (
        .clk        (clk        ),
        .rst        (rst        ),
        .Instr      (Instr      ),
        .ReadData   (ReadData   ),
        .WriteData  (WriteData  ),
        .PC         (PC         ),
        .ALUResult  (ALUResult  ),
        .MemWrite   (MemWrite   )
    );

    // instruction memory
    // contained machine code instructions which instruct processor on which operations to make
    // effectively a rom because our processor cannot write to it
    imem imemory (
        .addr   (PC     ),
        .instr  (Instr  )
    );

    // data memory
    // containes data accessible by the processor through ldr and str commands
    dmem dmemory (
        .clk     (clk       ),
        .wr_en   (MemWrite  ),
        .addr    (ALUResult ),
        .wr_data (WriteData ),
        .rd_data (ReadData  )
    );


endmodule

```



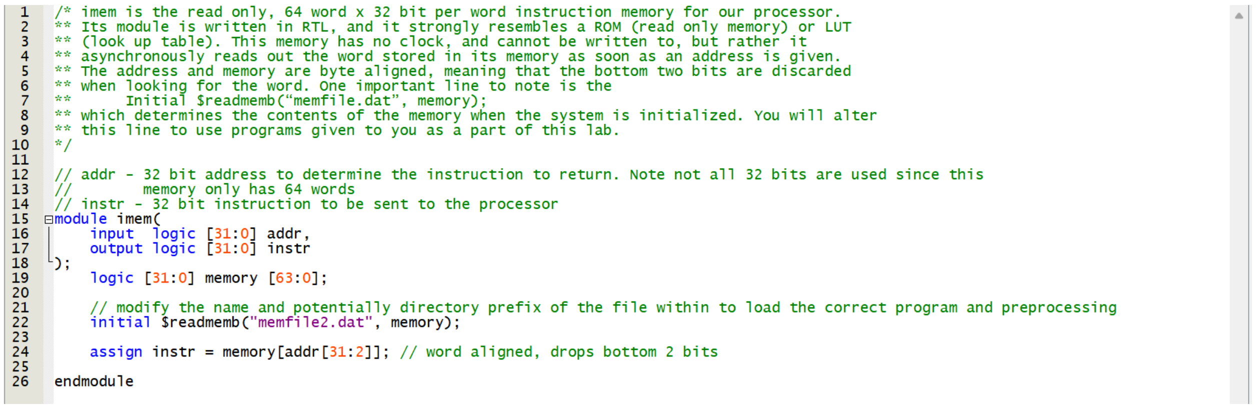
```systemverilog
/* testbench is a simulation module which simply instantiates the processor system and runs 50 cycles
** of instructions before terminating. At termination, specific register file values are checked to
** verify the processors' ability to execute the implemented instructions.
*/
module testbench();

    // system signals
    logic clk, rst;

    // generate clock with 100ps clk period
    initial begin
        clk = '1;
        forever #50 clk = ~clk;
    end

    // processor instantion. Within is the processor as well as imem and dmem
    top cpu (.clk(clk), .rst(rst));

    initial begin
        // start with a basic reset
        rst = 1; @(posedge clk);
        rst <= 0; @(posedge clk);

        // repeat for 50 cycles. Not all 50 are necessary, however a loop at the end of the program will keep anything weird from hap
        repeat(50) @(posedge clk);

        // basic checking to ensure the right final answer is achieved. These DO NOT prove your system works. A more careful look at
        // simulation and code will be made.

        // task 1:
        //assert(cpu.processor.u_reg_file.memory[8] == 32'd11) $display("Task 1 Passed");
        //else                                                  $display("Task 1 Failed");

        // task 2:
        assert(cpu.processor.u_reg_file.memory[8] == 32'd1)  $display("Task 2 Passed");
        else                                                 $display("Task 2 Failed");

        $stop;
    end

endmodule
```

## 5)dmem.sv



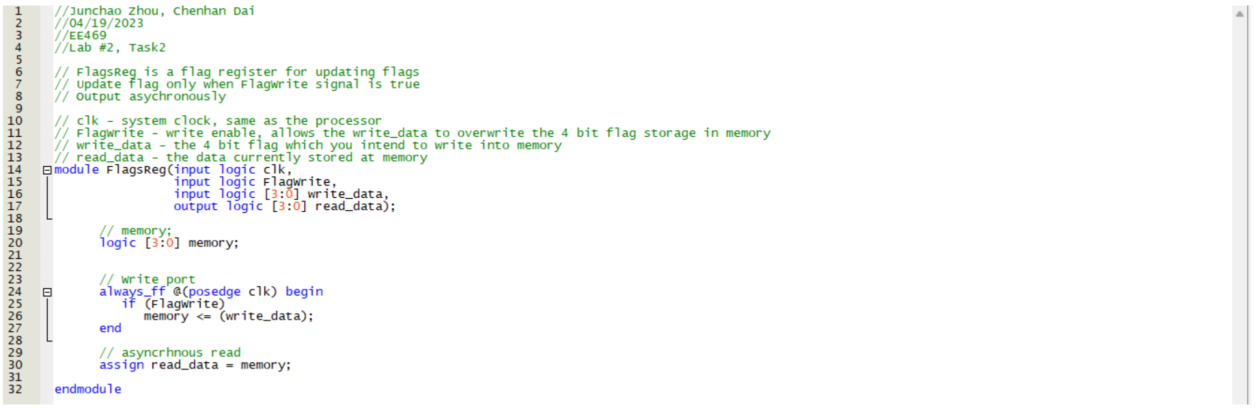
```
/* dmem is a more traditional, albeit very uninteresting, random access 64 word x 32 bit per word memory.
** This module is also written in RTL, and likely strongly resembles your own register file except for a
** few minor differences. The first is that there is only a single read port, compared to the register
** file's two read ports. The other difference is that the dmem is also byte aligned, and therefore
** discards the bottom two bits of the address when doing a read or write.
*/

// clk - system clock, same as the processor
// wr_en - write enable, allows the wr_data to overwrite the 32 bit word stored in memory[addr]
// addr - the location to which you intend to read or write from
// wr_data - the 32 bit data word which you intend to write into memory
// rd_data - the data currently stored at memory[addr]
module dmem (
    input  logic        clk, wr_en,
    input  logic [31:0] addr,
    input  logic [31:0] wr_data,
    output logic [31:0] rd_data
);

    logic [31:0] memory [63:0];

    // asyncrhnous read
    assign rd_data = memory[addr[31:2]]; // word aligned, drop bottom 2 bits

    // syncrhonous gated write
    always_ff @(posedge clk) begin
        if (wr_en) memory[addr[31:2]] <= wr_data; // word aligned, drop bottom 2 bits
    end

endmodule
```

## 6) imem.sv



```
/* imem is the read only, 64 word x 32 bit per word instruction memory for our processor.
** Its module is written in RTL, and it strongly resembles a ROM (read only memory) or LUT
** (look up table). This memory has no clock, and cannot be written to, but rather it
** asynchronously reads out the word stored in its memory as soon as an address is given.
** The address and memory are byte aligned, meaning that the bottom two bits are discarded
** when looking for the word. One important line to note is the
**        Initial $readmemb(“memfile.dat”, memory);
** which determines the contents of the memory when the system is initialized. You will alter
** this line to use programs given to you as a part of this lab.
*/

// addr - 32 bit address to determine the instruction to return. Note not all 32 bits are used since this
//         memory only has 64 words
// instr - 32 bit instruction to be sent to the processor
module imem(
    input  logic [31:0] addr,
    output logic [31:0] instr
);
    logic [31:0] memory [63:0];

    // modify the name and potentially directory prefix of the file within to load the correct program and preprocessing
    initial $readmemb("memfile2.dat", memory);

    assign instr = memory[addr[31:2]]; // word aligned, drops bottom 2 bits

endmodule
```

## 7) FlagsReg.sv



```
//Junchao Zhou, Chenhan Dai
//04/19/2023
//EE469
//Lab #2, Task2

// FlagsReg is a flag register for updating flags
// Update flag only when Flagwrite signal is true
// Output asychronously

// clk - system clock, same as the processor
// Flagwrite - write enable, allows the write_data to overwrite the 4 bit flag storage in memory
// write_data - the 4 bit flag which you intend to write into memory
// read_data - the data currently stored at memory
module FlagsReg(input logic clk,
                input logic Flagwrite,
                input logic [3:0] write_data,
                output logic [3:0] read_data);

      // memory;
      logic [3:0] memory;


      // write port
      always_ff @(posedge clk) begin
         if (Flagwrite)
            memory <= (write_data);
      end

      // asyncrhnous read
      assign read_data = memory;

endmodule
```

## 8) memfile.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
// 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
// B     - 1110_1010_IIII_IIII_IIII_IIII_IIII_IIII


11100010100011110000000000000000 // MAIN           ADD R0, R15, #0                0
11100000010000000001000000000000 //                   SUB R1, R0, R0                 4
11100010100000010010000000001010 //                   ADD R2, R1, #10        8
11100000100000000011000000000010 //                   ADD R3, R0, R2                 12
11100010010000100100000000000011 //                   SUB R4, R2, #3                 16
11100000010000110101000000000100 //                   SUB R5, R3, R4                 20
11100001100001000110000000000101 //                   ORR R6, R4, R5                 24
11100000000001100111000000000101 //                   AND R7, R6, R5                 28
11100101100000010111000000000000 //                   STR R7, [R1, #0]          32
11101010000000000000000000000001  //                  B SKIP               36
11100101100000010001000000000000 //                   STR R1, [R1, #0]       40
11101010000000000000000000000000  //                  B LOOP                    44
11100101100100011000000000000000 // SKIP           LDR R8, [R1, #0]            48
11101010111111111111111111111110  // LOOP       B LOOP                 52
```

**9) memfile2.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


11100010100011110000000000000000 // MAIN            ADD R0, R15, #0                    0
11100000010000000001000000000000 //                    SUB R1, R0, R0                     4
11100010100000010010000000001010 //                    ADD R2, R1, #10        8
11100000100000000011000000000010 //                    ADD R3, R0, R2                     12
11100010010000100100000000000011 //                    SUB R4, R2, #3                     16
11100000010000110101000000000100 //                    SUB R5, R3, R4                     20
11100001100001000110000000000101 //                    ORR R6, R4, R5                     24
11100000000001100111000000000101 //                    AND R7, R6, R5                     28
11100101100000010111000000000000 //                    STR R7, [R1, #0]          32
11101010000000000000000000000001  //                   B SKIP              36
11100101100000010001000000000000 //                    STR R1, [R1, #0]       40
11101010000000000000000000000000  //                   B LOOP                  44
11100101100100011000000000000000 // SKIP             LDR R8, [R1, #0]            48
11100010010101101001000000001111 // B_START     CMP R9, R6, #15         52
00011010111111111111111111111101  //           BNE B_START            56
11100000010101011001000000000100 //            CMP R9, R5, R4          60
00011010000000000000000000000000  //            BNE BNE_TESTED         64
11101010111111111111111111111010  //           B B_START              68
11100000010100101001000000000011 // BNE_TESTED  CMP R9, R2, R3          72
10101010111111111111111111111000  //           BGE B_START            76
11100000010100111001000000000010 //            CMP R9, R3, R2          80
10101010000000000000000000000000  //            BGE BGE_TESTED         84
11101010111111111111111111110101  //           B B_START              88
11100000010100111001000000000010 // BGE_TESTED  CMP R9, R3, R2          92
11011010111111111111111111110011  //           BLE B_START            96
11100000010100101001000000000011 //            CMP R9, R2, R3          100
11011010000000000000000000000000  //            BLE BLE_TESTED         104
11101010111111111111111111110000  //           B B_START              108
11100010100000011000000000000001 // BLE_TESTED  ADD R8, R1, #1         112
11101010111111111111111111111110  // LOOP       B LOOP                 116
```