Eugene Ngo
CSE 469
April 20, 2023
Lab 2 Report

## Procedure:

This lab was comprised of two tasks:

1. Implementing the given arm CPU module with the ALU and reg_file that was created in Lab 1.

2. Add extra control path logic to implement CMP, B EQ, B NE, B GE, B GT, B LE, B LT instructions

Once the designs for both tasks were implemented, they were thoroughly tested in ModelSim using the given testbenches and the DAT files, to demonstrate their functionality.

## Task #1:

The first task was to implement the given ARM CPU with the ALU and reg_file that were created in Lab 1. This was done by looking through the mapped-out diagram for the single-cycle CPU in the Lab 2 document and then aligning the proper signals that were already implemented in the ARM module with the inputs and outputs for the reg_file. The outputs from the reg_file were then properly aligned with the inputs of the ALU and then processed by the ALU and outputted for the memory or writeback sections. The schematic I implemented is shown in the diagram below.

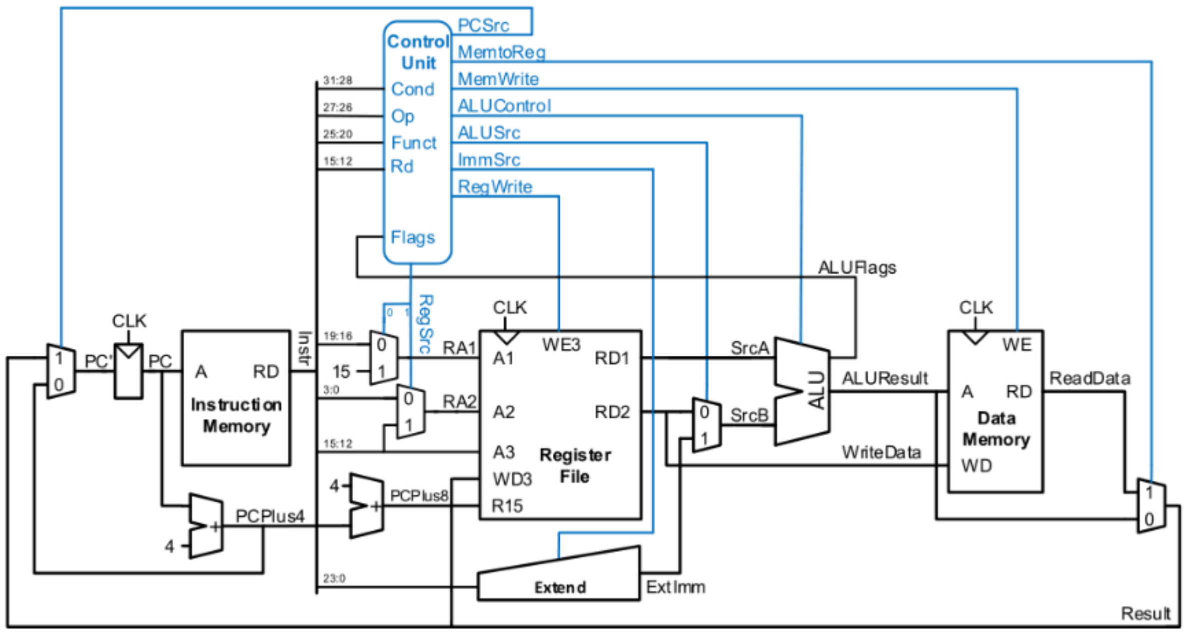
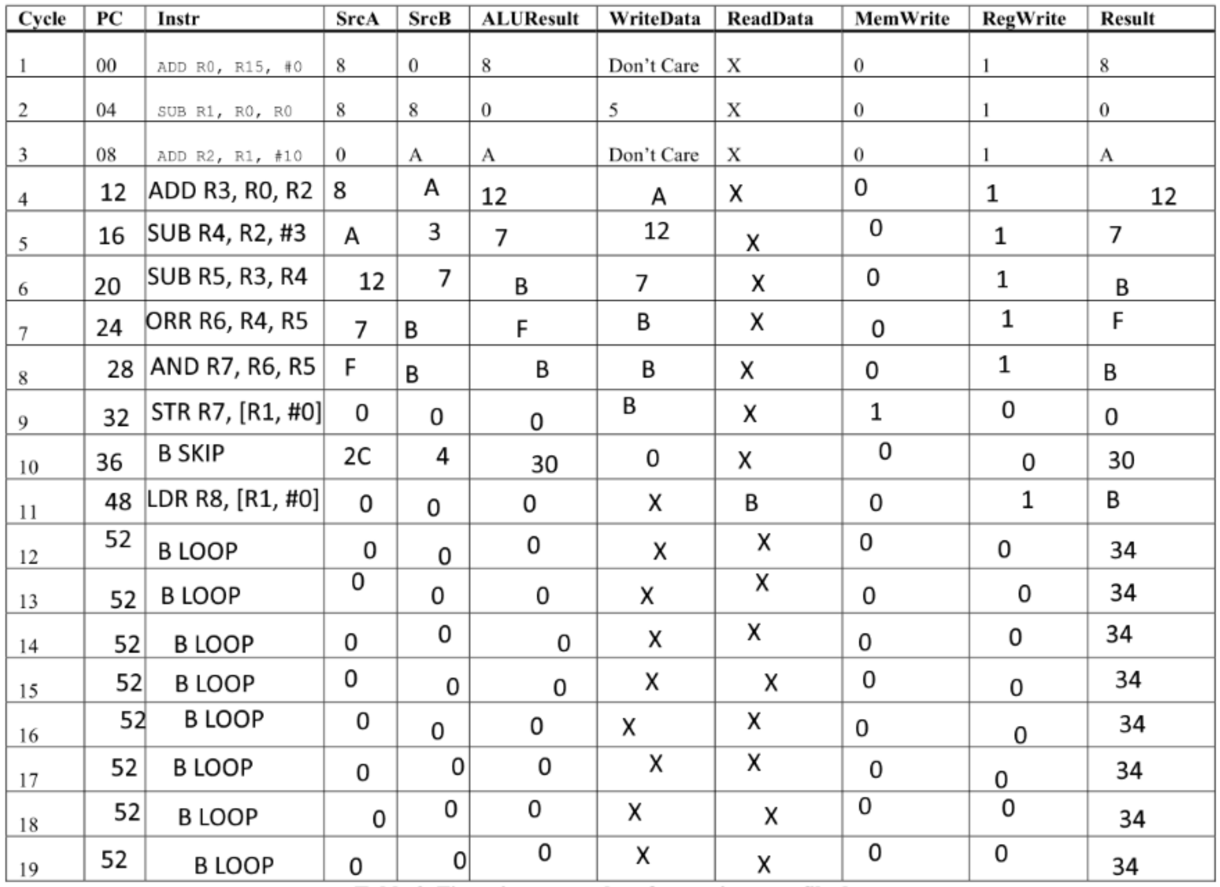


**Figure 3. Single-cycle ARM processor**

*Figure 1: Schematic for a single-cycle ARM CPU*

This schematic was then written and compiled in Quartus and then tested in ModelSim by varying the Instruction input based on the memfile.dat file.

The values of instructions and the resulting signals and outputs were tabulated to determine the proper expected values from the CPU as it goes through memfile.dat



| Cycle | PC | Instr | SrcA | SrcB | ALUResult | WriteData | ReadData | MemWrite | 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 | 5 | 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 | 2C | 4 | 30 | 0 | X | 0 | 0 | 30 |
| 11 | 48 | LDR R8, [R1, #0] | 0 | 0 | 0 | X | B | 0 | 1 | B |
| 12 | 52 | B LOOP | 0 | 0 | 0 | X | X | 0 | 0 | 34 |
| 13 | 52 | B LOOP | 0 | 0 | 0 | X | X | 0 | 0 | 34 |
| 14 | 52 | B LOOP | 0 | 0 | 0 | X | X | 0 | 0 | 34 |
| 15 | 52 | B LOOP | 0 | 0 | 0 | X | X | 0 | 0 | 34 |
| 16 | 52 | B LOOP | 0 | 0 | 0 | X | X | 0 | 0 | 34 |
| 17 | 52 | B LOOP | 0 | 0 | 0 | X | X | 0 | 0 | 34 |
| 18 | 52 | B LOOP | 0 | 0 | 0 | X | X | 0 | 0 | 34 |
| 19 | 52 | B LOOP | 0 | 0 | 0 | X | X | 0 | 0 | 34 |

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

*Figure 2: Signals and output when running memfile.dat*

## Task #2:

The second task was to implement CMP, B EQ, B NE, B GE, B GT, B LE, and B LT instructions. This required modifying the control logic and data path to first store the flags from CMP instructions, then to modify the control logic sections to account for the conditional bits within the instruction to implement the new instructions.

Following the implementation of the new instructions, the modules were compiled in Quartus and then tested in ModelSim by varying the Instruction input based on the memfile2.dat file.

Before testing, the PC sequence was written out to determine what set of instructions should have been taken based on the conditional branching.

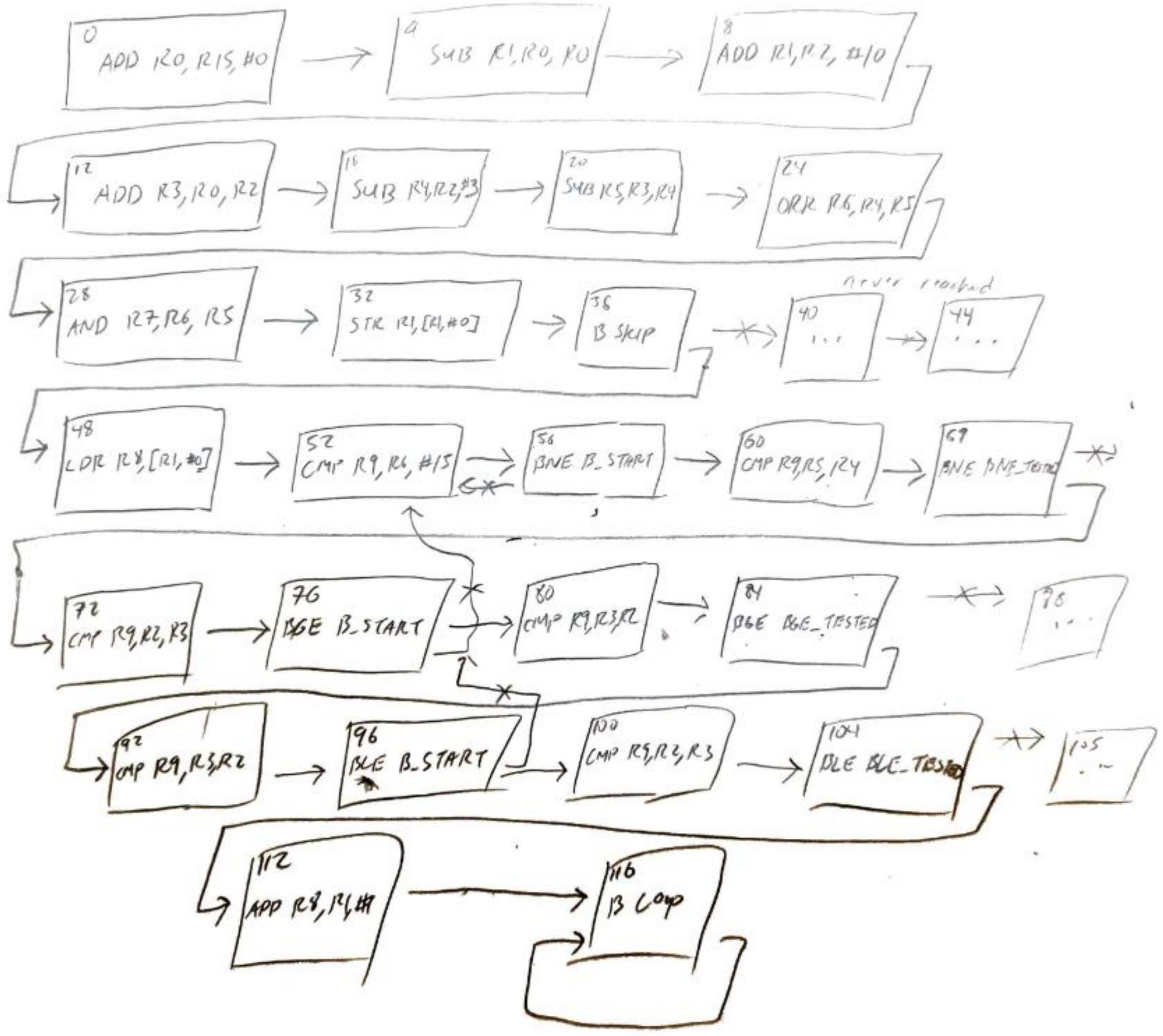


*Figure 3: This is the PC sequence for memfile2.dat*

The PC sequence displayed above in Figure 3 was compared with the simulation results to determine if the instructions were implemented correctly.

## Results

### Task #1:

After implementing the ARM module, I ran Modelsim to test it.

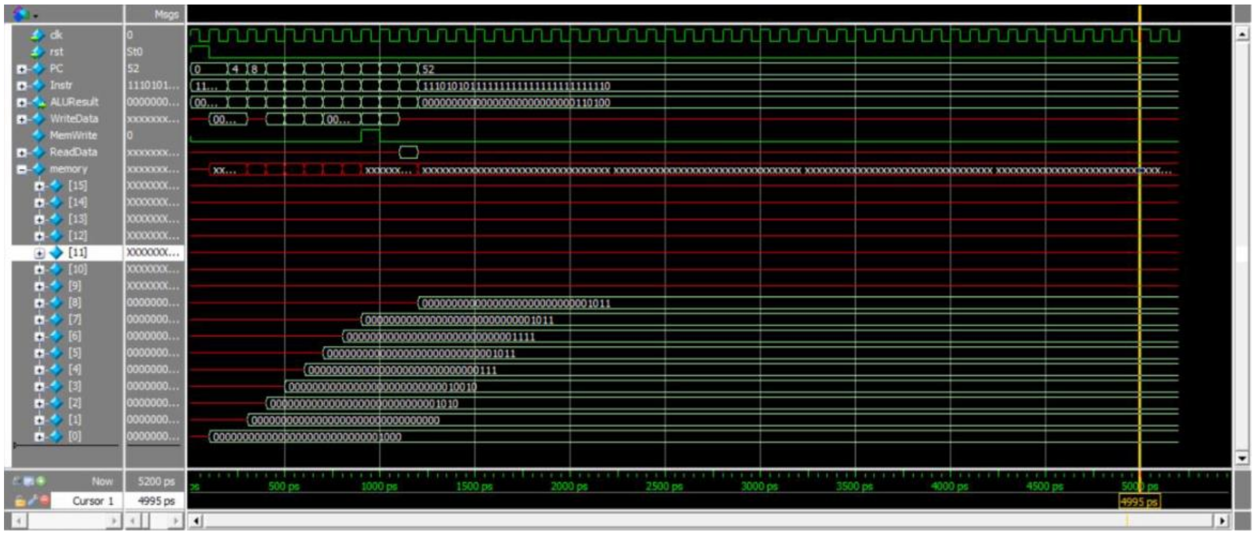


*Figure 4: The waveform generated for the task 1 ARM module testbench (memfile.dat)*

As seen in the waveforms above, the ARM CPU varies the values of the regfile based on the instructions executed, as a CPU should.

## Task #2:

After implementing the new instructions in Quartus, I ran Modelsim to test it.

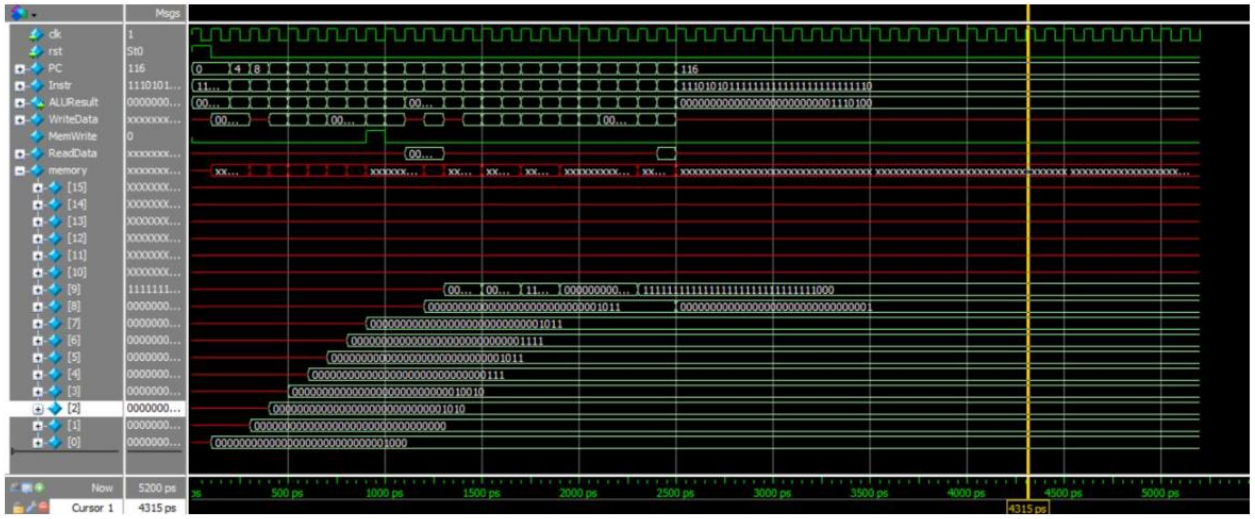


*Figure 5: Modelsim waves for testing CMP, B EQ, B NE, B GE, B GT, B LE, B LT when implemented (memefile2.dat)*

As seen in the waveforms above, the ARM CPU varies the values of the regfile based and PC based on the branches and instructions executed, as a CPU should.

## Appendix

See the following list for the order:

top.sv

testbench.sv

dmem.sv

imem.sv

alu.sv

alu_testbench.sv

arm.sv

fullAdder.sv

reg_file.sv

reg_file_testbench.sv

singleALU.sv

See the attached documents for the code:

```
// Eugene Ngo
// 4/20/2023
// CSE 469
// Lab 2, Task 1 and 2

/* 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


// testbench tests the behaviors of the ALU by running through an instance of the
// top module. The top module instantiates the imeme module which calls on
// the memfile.dat and memfile2.dat files that send in the instruction inputs
// for the testbench module to use. The results are compared to actual expected
// values to determine if the functionality of the overall CPU is correct.
module testbench();

    // system signals
    logic clk, rst;

    // generate clock with 100ps clk period
```

```systemverilog
    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 happening
        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 your
        // 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
```

```
// Eugene Ngo
// 4/20/2023
// CSE 469
// Lab 2, Task 1 and 2

/* 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 happening
        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 your
        // 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
```

```systemverilog
// Eugene Ngo
// 4/20/2023
// CSE 469
// Lab 2, Task 1 and 2

/* 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
```

```
// Eugene Ngo
// 4/20/2023
// CSE 469
// Lab 2, Task 1 and 2

/* 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(
    "C:\\Users\\egeen\\Desktop\\School\\EE 469\\Lab\\Lab 2\\memfile2.dat",
    memory);


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

endmodule
```

```
// Eugene Ngo
// 4/20/2023
// CSE 469
// Lab 1, Task 3

// A module to implement a 32 bit ARM-based ALU.
// This contains the central logic for calling on submodules
// that make up the ALU.

module alu(input logic [31:0] a, b,
           input logic [1:0] ALUControl,
           output logic [31:0] Result,
           output logic [3:0] ALUFlags);
   // 00 = add
   // 01 = subtract
   // 10 = AND
   // 11 = OR

   logic [31:0] carries;

   // This initial call to the singleALU is to set the carry input for the
   // genvar statements.
   singleALU setCarries (.a(a[0]), .b(b[0]), .carryIn(ALUControl[0]),
                         .ALUControl(ALUControl), .Result(Result[0]),
                         .carryOut(carries[0]));

   // The genvar statements break down the 32 bit inputs and sends them to
   // 1 bit ALUs that add and subtract them based on the input control signal.
   // The carry outs are processed and sent to the next bit that is covered,
   // thus linking them and creating the actual 32 bit output value.
   genvar i;
   generate
      for (i = 1; i < 32; i++) begin: ALUPipeline
         singleALU results (.a(a[i]), .b(b[i]), .carryIn(carries[i - 1]),
                            .ALUControl(ALUControl), .Result(Result[i]),
                            .carryOut(carries[i]));
      end // end loop
   endgenerate // end generate

   // Setting flags:
   // Overflow is xor of carry[30], carry[31]
   xor overFlowCheck (ALUFlags[0], carries[31], carries[30]);
   // Carryout is the last carry.
   assign ALUFlags[1] = carries[31];

   // Inefficient and bad style. RTL would be better.
   // Checks if any one of the bits within the 32-bit outputted value
   // contains any 1s. If there is a single 1, the output zero flag is not
   // raised.
   nor zeroChecker
      (ALUFlags[2], Result[31], Result[30], Result[29], Result[28],
       Result[27], Result[26], Result[25], Result[24], Result[23],
       Result[22], Result[21], Result[20], Result[19], Result[18],
       Result[17], Result[16], Result[15], Result[14], Result[13],
       Result[12], Result[11], Result[10], Result[9], Result[8],
       Result[7], Result[6], Result[5], Result[4], Result[3],
       Result[2], Result[1], Result[0]);

   assign ALUFlags[3] = Result[31];

endmodule

// alu_testbench tests the behaviors of the ALU by running a .tv file
// through it. The results are compared to actual expected values to
// determine if the functionality of the ALU is correct.

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];

   // Calls on the ALU module to test it.
   alu dut (.a(a), .b(b), .ALUControl(ALUControl), .Result(Result),
            .ALUFlags(ALUFlags));

   parameter CLOCK_PERIOD = 100;

   initial clk = 1;

   // Generates a clock signal
   always begin
         #(CLOCK_PERIOD/2);
         clk = ~clk;
   end

   // Reads through the .tv file and individually and test if the
   // vector file values are the same as the values generated
   // by the alu files.
   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 loop
   end // end initial
endmodule

```

```
// Eugene Ngo
// 4/7/2023
// CSE 469
// Lab 1 Task 3

// alu_testbench tests all expected, unexpected, and edgecase behaviors
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(ALUFlags));

   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 loop
   end // end initial
endmodule
```

```systemverilog
// Eugene Ngo
// 4/20/2023
// CSE 469
// Lab 2, Task 1 and 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] FlagsReg;                  // register for storing flag outputs
    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, 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];

    // Takes the control signals from the control module, and combines them with the
    // different input signals RA1, RA2, Instr[15:12], Result. It outputs the read data
    // signals: RD1, RD2 and writes in and saves data into its own memory if the
    // wr_en signal is enabled.
    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)
    );

    // 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

    // Forwards the outputs from the register files and the selection logic above,
    // to the ALU. The ALU computes a mathematical operation on the incoming values
    // based on the inputs and outputs the result and the Flags that are triggered.
    alu u_alu (
        .a          (SrcA),
        .b          (SrcB),
        .ALUControl (ALUControl),
        .Result     (ALUResult),
        .ALUFlags   (ALUFlags)
    );

    // FlagsReg setting logic
    // sets the flags if the FlagWrite control signal is enabled
    always_ff @ (posedge clk) begin
      if (FlagWrite) begin
         FlagsReg <= ALUFlags;
      end
   end

    // 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 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
        casez (Instr[31:20])

```

```
            // ADD (Imm or Reg)
            12'b111000?01000 : 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
                RegWrite = 1;
                RegSrc   = 'b00;
                ImmSrc   = 'b00;
                ALUControl = 'b00;
                FlagWrite = 0;
            end

            // SUB (Imm or Reg)
            12'b111000?00100 : 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
                RegWrite = 1;
                RegSrc   = 'b00;
                ImmSrc   = 'b00;
                ALUControl = 'b01;
                FlagWrite = 0;
            end

            // AND
            12'b111000000000 : begin
                PCSrc    = 0;
                MemtoReg = 0;
                MemWrite = 0;
                ALUSrc   = 0;
                RegWrite = 1;
                RegSrc   = 'b00;
                ImmSrc   = 'b00;    // doesn't matter
                ALUControl = 'b10;
                FlagWrite = 0;
            end

            // ORR
            12'b111000011000 : begin
                PCSrc    = 0;
                MemtoReg = 0;
                MemWrite = 0;
                ALUSrc   = 0;
                RegWrite = 1;
                RegSrc   = 'b00;
                ImmSrc   = 'b00;    // doesn't matter
                ALUControl = 'b11;
                FlagWrite = 0;
            end

            // LDR
            12'b111001011001 : begin
                PCSrc    = 0;
                MemtoReg = 1;
                MemWrite = 0;
                ALUSrc   = 1;
                RegWrite = 1;
                RegSrc   = 'b10;    // msb doesn't matter
                ImmSrc   = 'b01;
                ALUControl = 'b00;  // do an add
                FlagWrite = 0;
            end

            // STR
            12'b111001011000 : begin
                PCSrc    = 0;
                MemtoReg = 0; // doesn't matter
                MemWrite = 1;
                ALUSrc   = 1;
```

```
                    RegWrite = 0;
                    RegSrc   = 'b10;    // msb doesn't matter
                    ImmSrc   = 'b01;
                    ALUControl = 'b00;  // do an add
                    FlagWrite = 0;
                end

                // B
                12'b11101010???? : begin
                    PCSrc    = 1;
                    MemtoReg = 0;
                    MemWrite = 0;
                    ALUSrc   = 1;
                    RegWrite = 0;
                    RegSrc   = 'b01;
                    ImmSrc   = 'b10;
                    ALUControl = 'b00;  // do an add
                    FlagWrite = 0;
                end

                // CMP
                12'b111000?00101 : begin
                   PCSrc    = 0;
                    MemtoReg = 0;
                    MemWrite = 0;
                    ALUSrc   = Instr[25]; // may use immediate
                    RegWrite = 1;
                    RegSrc   = 'b00;
                    ImmSrc   = 'b00;
                    ALUControl = 'b01;
                    FlagWrite = 1;
                end

                // B EQ
                12'b00001010???? : begin
                    PCSrc    = FlagsReg[2];
                    MemtoReg = 0;
                    MemWrite = 0;
                    ALUSrc   = 1;
                    RegWrite = 0;
                    RegSrc   = 'b01;
                    ImmSrc   = 'b10;
                    ALUControl = 'b00;  // do an add
                    FlagWrite = 0;
                end

                // B NE
                12'b00011010???? : begin
                    PCSrc    = ~ FlagsReg[2];
                    MemtoReg = 0;
                    MemWrite = 0;
                    ALUSrc   = 1;
                    RegWrite = 0;
                    RegSrc   = 'b01;
                    ImmSrc   = 'b10;
                    ALUControl = 'b00;  // do an add
                    FlagWrite = 0;
                end

                // B GE
                12'b10101010???? : begin
                    PCSrc    = ~ FlagsReg[3] | FlagsReg[2];
                    MemtoReg = 0;
                    MemWrite = 0;
                    ALUSrc   = 1;
                    RegWrite = 0;
                    RegSrc   = 'b01;
                    ImmSrc   = 'b10;
                    ALUControl = 'b00;  // do an add
                    FlagWrite = 0;
                end

                // B GT
                12'b11001010???? : begin
                    PCSrc    = ~ FlagsReg[3];
                    MemtoReg = 0;
                    MemWrite = 0;
                    ALUSrc   = 1;
                    RegWrite = 0;
                    RegSrc   = 'b01;
                    ImmSrc   = 'b10;
                    ALUControl = 'b00;  // do an add
                    FlagWrite = 0;
                end

                // B LE
                12'b11011010???? : begin
                    PCSrc    = FlagsReg[3] | FlagsReg[2];
                    MemtoReg = 0;
                    MemWrite = 0;
                    ALUSrc   = 1;
                    RegWrite = 0;
                    RegSrc   = 'b01;
                    ImmSrc   = 'b10;
                    ALUControl = 'b00;  // do an add
                    FlagWrite = 0;
                end

                // B LT
                12'b10111010???? : begin
                    PCSrc    = FlagsReg[3];
                    MemtoReg = 0;
                    MemWrite = 0;
                    ALUSrc   = 1;
                    RegWrite = 0;
                    RegSrc   = 'b01;
                    ImmSrc   = 'b10;
                    ALUControl = 'b00;  // do an add
                    FlagWrite = 0;
                end

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

```
// Eugene Ngo
// 4/20/2023
// CSE 469
// Lab 1 Task 3

// A module to implement a fullAdder.
// This file was reused from a given file in EE 371
module fullAdder (a, b, carryIn, Result, carryOut);

   input  logic a, b, carryIn;
   output logic Result, carryOut;

   assign  Result = a ^ b ^ carryIn;
   assign carryOut = (a & b) | (carryIn & (a ^ b));

endmodule
```

```
// Eugene Ngo
// 4/20/2023
// CSE 469
// Lab 1, Task 2

// A module to implement a 32 bit, 16 register, reg_file.
// This module takes in data and stores it in flip flops.
// The data can then be read via two read inputs and outputs.
// The data can be written in via 1 write input and 1 write enable.

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);

   // Stores all the values.
   logic [15:0][31:0] memory;

   // Main logic unit. Clocked flip flops.
   always_ff @ (posedge clk) begin
   // if write enabled, write data
      if (wr_en) begin
         memory[write_addr] <= write_data;
      end
   end

   // Read out data the instant the read_data signals are updated.
   always_comb begin
      read_data1 = memory[read_addr1];
      read_data2 = memory[read_addr2];
   end

endmodule

// reg_file_testbench tests the behaviors of the reg_file by inputting
// expected testing values. This tests for cycle delays in writes,
// same-cycle reads, and 1 cycle delays for reads occuring after writes.
// The results are compared to expected values to determine if the
// functionality of the reg_file is correct.

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

   // Calls on the reg_file module to test it.
   reg_file dut (.clk, .wr_en, .write_data, .write_addr, .read_addr1, .read_addr2, .
read_data1, .read_data2);

   parameter clock_period = 10000;

   integer i;

   initial begin // Set up the clock
      clk <= 0;
      for (i=0; i<1000; i++) begin: clockCount
         forever #(clock_period /2) clk <= ~clk;
      end

   end

   initial begin
      $display("%t Behavior check", $time);

            // Testing functionality of
            // 1 cycle delay of writes.
      wr_en = 1'b1;                 @(posedge clk);
      write_data = 32'b0;           @(posedge clk);
      write_addr = 4'b0010;         @(posedge clk);
                                    @(posedge clk);

      write_data = 32'b1;           @(posedge clk);
      write_addr = 4'b0011;         @(posedge clk);
```

```
                                        @(posedge clk);
              // Testing functionality of
              // same cycle reads.
        read_addr1 = 4'b0010;           @(posedge clk);
        read_addr2 = 4'b0011;           @(posedge clk);

              // Testing the functionality of
              // 1 cycle delayed reads
              // after an updated write
        write_addr = 4'b0010;           @(posedge clk);
        read_addr1 = 4'b0010;           @(posedge clk);
                                        @(posedge clk);
        // read_data1 should update to 1 now.

        write_data = 32'b0;             @(posedge clk);
        write_addr = 4'b0011;           @(posedge clk);

        read_addr2 = 4'b0011;           @(posedge clk);
                                        @(posedge clk);
        // read_data2 should update to 0 now.

    end

endmodule
```

```
// Eugene Ngo
// 4/7/2023
// CSE 469
// Lab 1 Task 2

// reg_file__testbench tests all expected, unexpected, and edgecase behaviors
module reg_file_testbench();
   logic clk, wr_en;
   logic [31:0] write_data, read_data1, read_data2;
   logic [3:0] write_addr, read_addr1, read_addr2;

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

   parameter clock_period = 10000;

   integer i;

   initial begin // Set up the clock
      clk <= 0;
      for (i=0; i<1000; i++) begin: clockCount
         forever #(clock_period /2) clk <= ~clk;
      end

   end

   initial begin
      $display("%t Behavior check", $time);

            // Testing functionality of
            // 1 cycle delay of writes.
      wr_en = 1'b1;                 @(posedge clk);
      write_data = 32'b0;           @(posedge clk);
      write_addr = 4'b0010;         @(posedge clk);
                                    @(posedge clk);

      write_data = 32'b1;           @(posedge clk);
      write_addr = 4'b0011;         @(posedge clk);
                                    @(posedge clk);
            // Testing functionality of
            // same cycle reads.
      read_addr1 = 4'b0010;         @(posedge clk);
      read_addr2 = 4'b0011;         @(posedge clk);

            // Testing the functionality of
            // 1 cycle delayed reads
            // after an updated write
      write_addr = 4'b0010;         @(posedge clk);
      read_addr1 = 4'b0010;         @(posedge clk);
                                    @(posedge clk);
      // read_data1 should update to 1 now.

      write_data = 32'b0;           @(posedge clk);
      write_addr = 4'b0011;         @(posedge clk);

      read_addr2 = 4'b0011;         @(posedge clk);
                                    @(posedge clk);
      // read_data2 should update to 0 now.




   end




endmodule

```

```
// Eugene Ngo
// 4/20/2023
// CSE 469
// Lab 2, Task 1 and 2

// A module to implement ALU logic for single bits.
// It takes in single bit values and adds, subtracts, performs OR or AND
// based on the ALUControl signals. The functions are all performed at once
// and then the outputted value is selected.
// Combine single bit ALUs to make up large ALUs.
module singleALU (a, b, carryIn, ALUControl, Result, carryOut);
   input logic a, b, carryIn;
   input logic [1:0] ALUControl;
   output logic Result, carryOut;


   logic [2:0] outputs;

   and andValue (outputs[1], b, a);
   or  orValue  (outputs[2], b, a);

   // Selecting B (Addition = A+B+0, Subtraction = A+(~B)+1)
   // logic subtractSelector = ALUControl[0];

   // MUX to select B (ALUControl[0] == 1 = select ~B)
   wire middleValues[1:0];
   and selectB (middleValues[0], ~ALUControl[0], b);
   and selectNotB (middleValues[1], ALUControl[0], ~b);

   logic selectedB;
   or selectedBValue (selectedB, middleValues[0], middleValues[1]);

   fullAdder fullAddedValue (.a(a), .b(selectedB), .carryIn(carryIn),
                             .Result(outputs[0]), .carryOut(carryOut));

   // MUX to select between computed values (add, sub, and, or)

   always_comb begin
      case (ALUControl)
         2'b00: Result = outputs[0];
         2'b01: Result = outputs[0];
         2'b10: Result = outputs[1];
         2'b11: Result = outputs[2];
         default: Result = 1'bX;
      endcase // end case statements
   end // end comb block

endmodule   // End of module
```