Eugene Ngo

EE 469

May 3, 2022

Lab 3 Report

Procedure:

This lab was comprised of one task:

1. Implementing the pipelined arm CPU module by changing the original single-cycle CPU from lab 2.

After the pipelined CPU was implemented, it was thoroughly tested in Modelsim, ensuring to test forwarding from memory to execute, forwarding from writeback stage to execute, an example of stalling for a memory instruction and flushing for a branch instruction.

Task #1:

This was the main task, transforming the original single cycle CPU into a pipelined CPU using multiple cycles per instruction and skipping and forwarding between instructions as required. This was done by following the mapped-out diagram of the pipelined CPU from class and creating more logic signals as required, to forward and skip as required. At each stage of the cycle a flip-flop was added, to ensure that each cycle executed once per clock cycle (technically, each stage is executed near instantaneously, just the input and output of each stage is on a clock cycle to ensure proper pipelining). The schematic I implemented is shown in the diagram below.

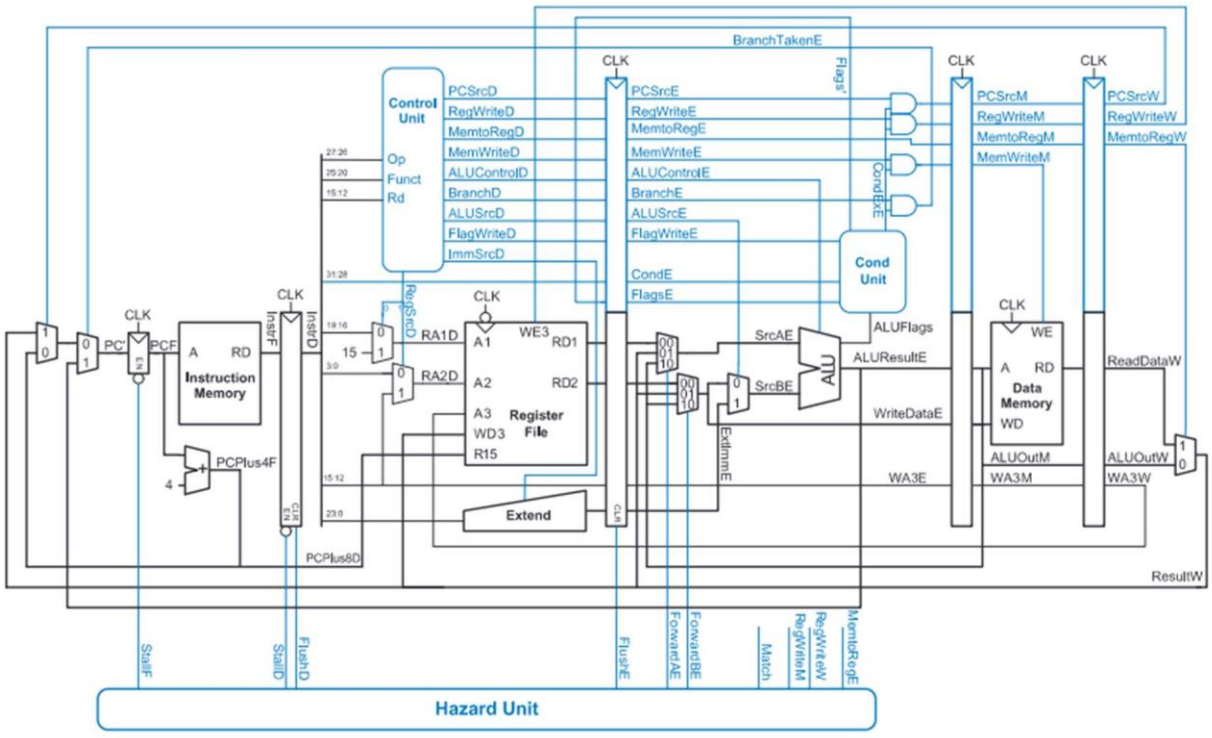

Figure 1: Schematic for a pipelined ARM CPU

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

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, memfiled2.dat and memfile3.dat.

Results

Task #1:

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

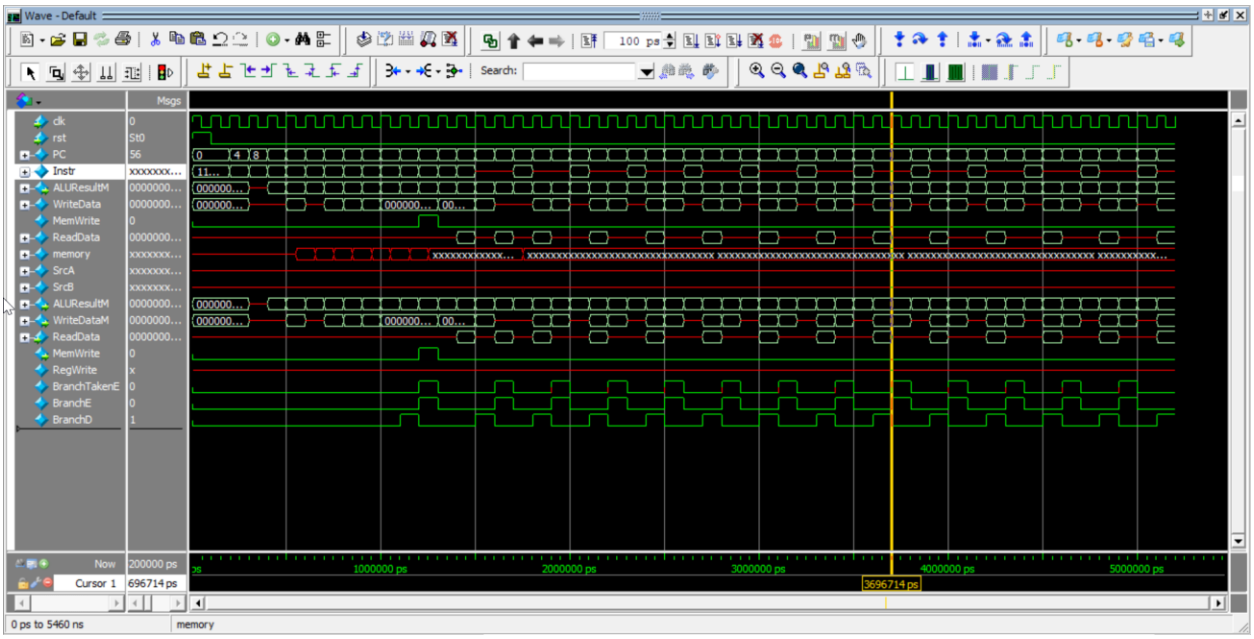

Figure 2: The waveform generated for task1, memfile.dat

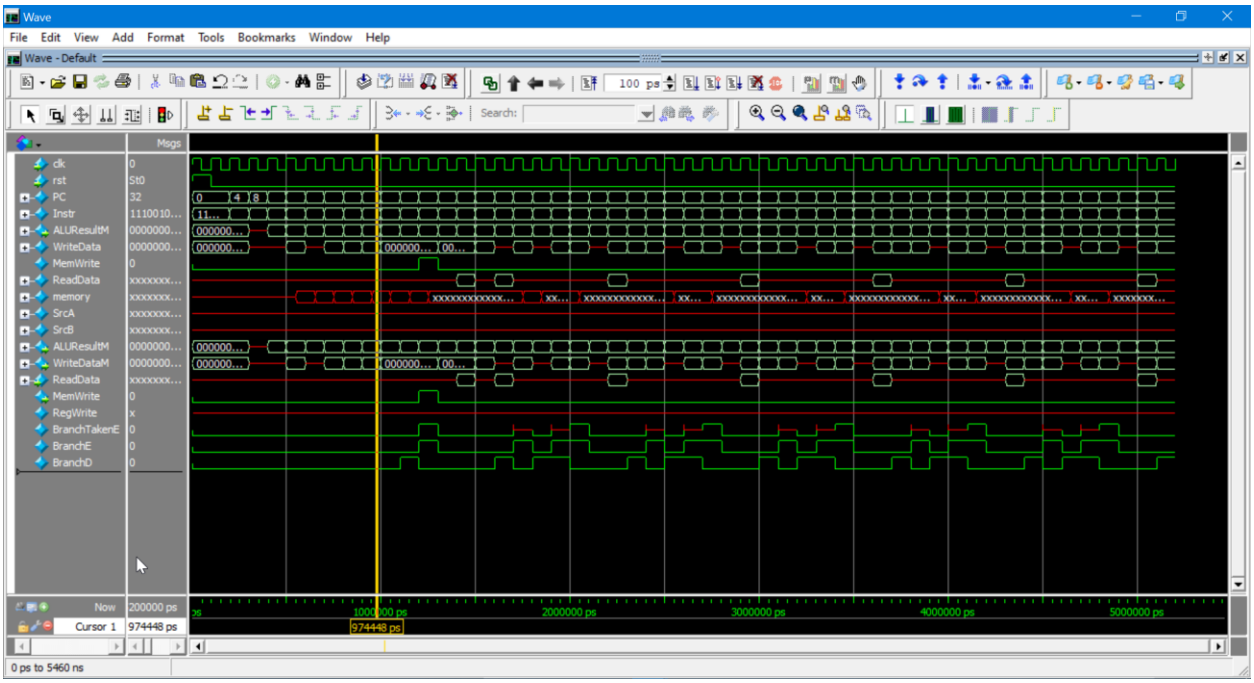

Figure 3: The waveform generated for task1, memfile2.dat

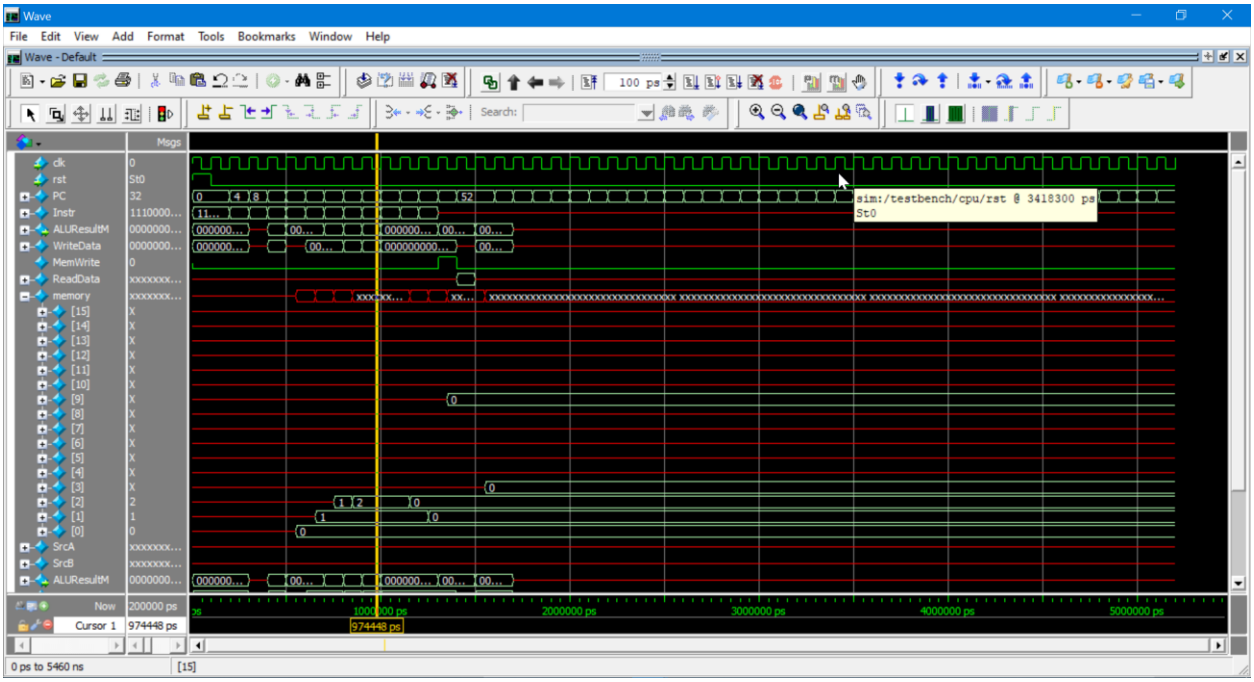

Figure 4: The waveform generated for task1, memfile3.dat

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

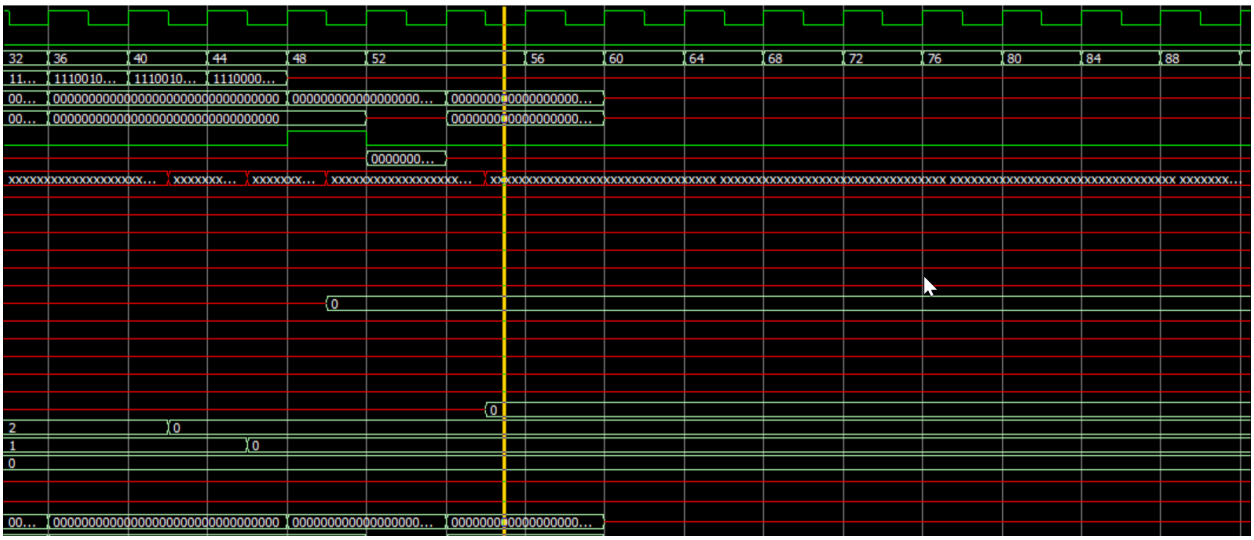

Figure 5: Memfile3.dat

As shown in Figure 5, Memfile3.dat stalls at PC = 52 as the previous two instructions (44 and 48) are store and load respectively, with consecutive memory accesses to the same location. To ensure correct data being read in, the CPU stalls for memory instruction store being completed.

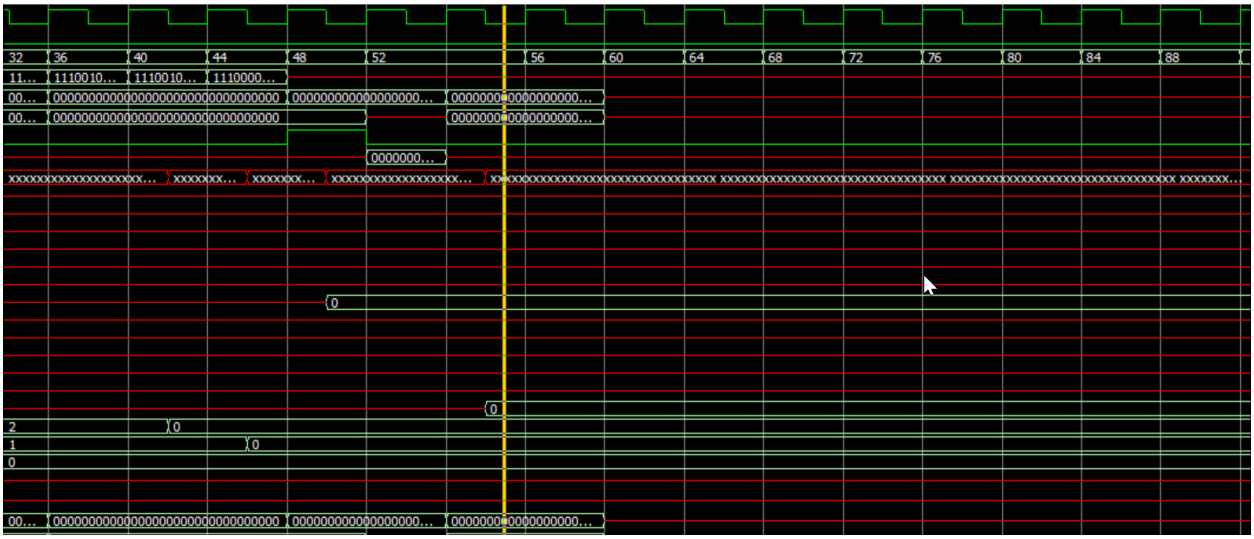

Figure 6: Memfile.dat

As shown in Figure 6, Memfile.dat forwards data at PC = 52 as the instruction before loads a value into R3 which is used in PC = 52, thus the computed value is forwarded to PC = 52's execute stage, ensuring the integrity of the CPU architecture.

Appendix

See the following list for the order:

top.sv
testbench.sv
reg_file.sv
reg_file_testbench.sv
dmem.sv
imem.sv
fullAdder.sv
arm.sv
alu.sv
alu_testbench.sv

See the attached documents for the code:

```
1   // Eugene Ngo
2   // 5/3/23
3   // EE 469
4   // Lab 3
5
6   /* top is a structurally made toplevel module. It consists of 3 instantiations, as well as
    the signals that link them.
7   ** It is almost totally self-contained, with no outputs and two system inputs: clk and rst.
    clk represents the clock
8   ** the system runs on, with one instruction being read and executed every cycle. rst is the
    system reset and should
9   ** be run for at least a cycle when simulating the system.
10  */
11
12  // clk - system clock
13  // rst - system reset. Technically unnecessary
14  `timescale 1ns/10ps
15  module top(
16      input logic clk, rst
17  );
18
19      // processor io signals
20      logic [31:0] Instr;
21      logic [31:0] ReadData;
22      logic [31:0] WriteData;
23      logic [31:0] PC, ALUResult;
24      logic        MemWrite;
25
26      // our single cycle arm processor
27      arm processor (
28          .clk        (clk        ),
29          .rst        (rst        ),
30          .InstrF      (Instr      ),
31          .ReadData   (ReadData   ),
32          .WriteDataM  (WriteData  ),
33          .PC         (PC         ),
34          .ALUResultM  (ALUResult  ),
35          .MemWrite   (MemWrite   )
36      );
37
38      // instruction memory
39      // contained machine code instructions which instruct processor on which operations to
    make
40      // effectively a rom because our processor cannot write to it
41      imem imemory (
42          .addr   (PC     ),
43          .instr  (Instr  )
44      );
45
46      // data memory
47      // containes data accessible by the processor through ldr and str commands
48      dmem dmemory (
49          .clk     (clk       ),
50          .wr_en   (MemWrite  ),
51          .addr    (ALUResult ),
52          .wr_data (WriteData ),
53          .rd_data (ReadData  )
54      );
55
56
57  endmodule
58
59  /* testbench is a simulation module which simply instantiates the processor system and runs
    50 cycles
60  ** of instructions before terminating. At termination, specific register file values are
    checked to
61  ** verify the processors' ability to execute the implemented instructions.
62  */
63  module testbench();
64
65      // system signals
66      logic clk, rst;
67
```

```
68        // generate clock with 100ps clk period
69        initial begin
70            clk = '1;
71            forever #50 clk = ~clk;
72        end
73
74        // processor instantion. Within is the processor as well as imem and dmem
75        top cpu (.clk(clk), .rst(rst));
76
77        initial begin
78            // start with a basic reset
79            rst = 1; @(posedge clk);
80            rst <= 0; @(posedge clk);
81
82            // repeat for 50 cycles. Not all 50 are necessary, however a loop at the end of the
      program will keep anything weird from happening
83            repeat(50) @(posedge clk);
84
85            // basic checking to ensure the right final answer is achieved. These DO NOT prove
      your system works. A more careful look at your
86            // simulation and code will be made.
87
88            // task 1:
89            assert(cpu.processor.u_reg_file.memory[8] == 32'd11) $display("Task 1 Passed");
90            else                                                 $display("Task 1 Failed");
91
92            // task 2:
93    //        assert(cpu.processor.u_reg_file.memory[8] == 32'd1)  $display("Task 2 Passed");
94    //        else                                                 $display("Task 2 Failed");
95
96            $stop;
97        end
98
99    endmodule
100
```

```
1   // Eugene Ngo
2   // 5/3/23
3   // EE 469
4   // Lab 3
5
6   /* testbench is a simulation module which simply instantiates the processor system and runs
    50 cycles
7   ** of instructions before terminating. At termination, specific register file values are
    checked to
8   ** verify the processors' ability to execute the implemented instructions.
9   */
10  `timescale 1ns/10ps
11  module testbench();
12
13      // system signals
14      logic clk, rst;
15
16      // generate clock with 100ps clk period
17      initial begin
18          clk = '1;
19          forever #50 clk = ~clk;
20      end
21
22      // processor instantion. Within is the processor as well as imem and dmem
23      top cpu (.clk(clk), .rst(rst));
24
25      initial begin
26          // start with a basic reset
27          rst = 1; @(posedge clk);
28          rst <= 0; @(posedge clk);
29
30          // repeat for 50 cycles. Not all 50 are necessary, however a loop at the end of the
    program will keep anything weird from happening
31          repeat(50) @(posedge clk);
32
33          // basic checking to ensure the right final answer is achieved. These DO NOT prove
    your system works. A more careful look at your
34          // simulation and code will be made.
35
36          // task 1:
37          assert(cpu.processor.u_reg_file.memory[8] == 32'd11) $display("Task 1 Passed");
38          else                                                 $display("Task 1 Failed");
39
40          // task 2:
41          //assert(cpu.processor.u_reg_file.memory[8] == 32'd1)  $display("Task 2 Passed");
42          //else                                                 $display("Task 2 Failed");
43
44          $stop;
45      end
46
47  endmodule
```

```
1   // Eugene Ngo
2   // 5/3/23
3   // EE 469
4   // Lab 3
5
6   // reg file is a 16x32 register file that has 1
7   // write port and 2 asynchronous read ports.
8   // When wr_en is true, we write the write data into the
9   // write address.
10  module reg_file(
11     input  logic        clk, wr_en,
12     input  logic [31:0] write_data,
13     input  logic [3:0]  write_addr,
14     input  logic [3:0]  read_addr1, read_addr2,
15     output logic [31:0] read_data1, read_data2);
16
17    logic [15:0][31:0] memory;
18
19    always_ff @(posedge clk) begin
20      if (wr_en) begin
21        memory[write_addr] <= write_data;
22       end
23
24
25    end
26    assign read_data1 = memory[read_addr1];
27    assign read_data2 = memory[read_addr2];
28  endmodule
29
30
31  // reg_file_testbench is a testing fiel for reg_file
32  // that tests  that write data is written
33  // into register file a cycle after wr_en is true,
34  // checks if read data updates the register data
35  // the same cycle as the address asserted,
36  // and checks if read data is updated to write data
37  // the cycle after address is provided if
38  // write address is the same and wr_en is true
39  module reg_file_testbench();
40     logic CLOCK_50;
41     logic clk;
42     logic wr_en;
43     logic [31:0] write_data;
44     logic [3:0]  write_addr;
45     logic [3:0]  read_addr1, read_addr2;
46     logic [31:0] read_data1, read_data2;
47     assign CLOCK_50 = clk;
48
49     reg_file dut (.clk, .wr_en, .write_data, .write_addr, .read_addr1,. read_addr2, .
    read_data1, .read_data2);
50     parameter CLOCK_PERIOD=100;
51     initial begin
52        clk <= 0;
53        forever #(CLOCK_PERIOD/2) clk <= ~clk;
54     end
55
56
57     initial begin
58        wr_en <= 0; write_data = 5; write_addr = 3; read_addr1 = 2; read_addr2 = 3; @(posedge
    clk);
59        wr_en <= 1;  repeat(1) @(posedge clk);
60        write_data = 10; write_addr = 4;@(posedge clk);
61        write_data = 11; write_addr = 5;@(posedge clk);
62        read_addr1 = 4; @(posedge clk);
63        read_addr2 = 5; @(posedge clk);
64        write_data = 12; write_addr = 4; @(posedge clk);
65        write_data = 13; write_addr = 5; @(posedge clk);
66
67     end
68  endmodule
```

```
1   // Eugene Ngo
2   // 5/3/23
3   // EE 469
4   // Lab 3
5
6   // reg_file_testbench is a testing fiel for reg_file
7   // that tests  that write data is written
8   // into register file a cycle after wr_en is true,
9   // checks if read data updates the register data
10  // the same cycle as the address asserted,
11  // and checks if read data is updated to write data
12  // the cycle after address is provided if
13  // write address is the same and wr_en is true
14  module reg_file_testbench();
15     logic CLOCK_50;
16     logic clk;
17     logic wr_en;
18     logic [31:0] write_data;
19     logic [3:0]  write_addr;
20     logic [3:0]  read_addr1, read_addr2;
21     logic [31:0] read_data1, read_data2;
22     assign CLOCK_50 = clk;
23
24     reg_file dut (.clk, .wr_en, .write_data, .write_addr, .read_addr1,. read_addr2, .
       read_data1, .read_data2);
25     parameter CLOCK_PERIOD=100;
26     initial begin
27        clk <= 0;
28        forever #(CLOCK_PERIOD/2) clk <= ~clk;
29     end
30
31
32     initial begin
33        wr_en <= 0; write_data = 5; write_addr = 3; read_addr1 = 2; read_addr2 = 3; @(posedge
       clk);
34        wr_en <= 1;  repeat(1) @(posedge clk);
35        write_data = 10; write_addr = 4;@(posedge clk);
36        write_data = 11; write_addr = 5;@(posedge clk);
37        read_addr1 = 4; @(posedge clk);
38        read_addr2 = 5; @(posedge clk);
39        write_data = 12; write_addr = 4; @(posedge clk);
40        write_data = 13; write_addr = 5; @(posedge clk);
41
42     end
43  endmodule
```

```
1   // Eugene Ngo
2   // 5/3/23
3   // EE 469
4   // Lab 3
5
6   /* dmem is a more traditional, albeit very uninteresting, random access 64 word x 32 bit
    per word memory.
7   ** This module is also written in RTL, and likely strongly resembles your own register file
    except for a
8   ** few minor differences. The first is that there is only a single read port, compared to
    the register
9   ** file's two read ports. The other difference is that the dmem is also byte aligned, and
    therefore
10  ** discards the bottom two bits of the address when doing a read or write.
11  */
12
13  // clk - system clock, same as the processor
14  // wr_en - write enable, allows the wr_data to overwrite the 32 bit word stored in
    memory[addr]
15  // addr - the location to which you intend to read or write from
16  // wr_data - the 32 bit data word which you intend to write into memory
17  // rd_data - the data currently stored at memory[addr]
18  module dmem (
19      input  logic        clk, wr_en,
20      input  logic [31:0] addr,
21      input  logic [31:0] wr_data,
22      output logic [31:0] rd_data
23  );
24
25      logic [31:0] memory [63:0];
26
27      // asyncrhnous read
28      assign rd_data = memory[addr[31:2]]; // word aligned, drop bottom 2 bits
29
30      // syncrhonous gated write
31      always_ff @(posedge clk) begin
32          if (wr_en) memory[addr[31:2]] <= wr_data; // word aligned, drop bottom 2 bits
33      end
34
35  endmodule
```

```
1   // Eugene Ngo
2   // 5/3/23
3   // EE 469
4   // Lab 3
5
6   /* imem is the read only, 64 word x 32 bit per word instruction memory for our processor.
7   ** Its module is written in RTL, and it strongly resembles a ROM (read only memory) or LUT
8   ** (look up table). This memory has no clock, and cannot be written to, but rather it
9   ** asynchronously reads out the word stored in its memory as soon as an address is given.
10  ** The address and memory are byte aligned, meaning that the bottom two bits are discarded
11  ** when looking for the word. One important line to note is the
12  **      Initial $readmemb("memfile.dat", memory);
13  ** which determines the contents of the memory when the system is initialized. You will alte
14  ** this line to use programs given to you as a part of this lab.
15  */
16
17  // addr - 32 bit address to determine the instruction to return. Note not all 32 bits are
    used since this
18  //        memory only has 64 words
19  // instr - 32 bit instruction to be sent to the processor
20  module imem(
21      input  logic [31:0] addr,
22      output logic [31:0] instr
23  );
24      logic [31:0] memory [63:0];
25
26      // modify the name and potentially directory prefix of the file within to load the
    correct program and preprocessing
27      initial $readmemb("C:\\Users\\egeen\\Desktop\\School\\EE 469\\Lab\\Lab 3\\memfile.dat",
    memory);
28
29
30      assign instr = memory[addr[31:2]]; // word aligned, drops bottom 2 bits
31
32  endmodule
```

```
1   // Eugene Ngo
2   // 5/3/23
3   // EE 469
4   // Lab 3
5
6   // fullAdder takes in 2 bits, a and b, as well as a possible carry in bit and adds them
    together
7   // if there is a carry out, we output that bit in cout.
8   module fullAdder (A,B,cin,sum,cout);
9      input logic A,B,cin;
10     output logic sum,cout;
11
12     assign sum = A ^ B ^ cin;
13     assign cout = A & B | cin & (A^B);
14  endmodule
15
16  module fullAdder_testbench();
17     logic A,B,cin,sum,cout;
18     fullAdder dut (A,B,cin,sum,cout);
19
20     integer i;
21     initial begin
22        for (i = 0; i < 2**3; i++) begin
23           {A,B,cin} = i; #10;
24        end
25     end
26
27  endmodule
28
```

```
1   // Eugene Ngo
2   // 5/3/23
3   // EE 469
4   // Lab 3
5
6   /* 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.
7   */
8
9   // clk - system clock
10  // rst - system reset
11  // Instr - incoming 32 bit instruction from imem, contains opcode, condition, addresses and
    or immediates
12  // ReadData - data read out of the dmem
13  // WriteData - data to be written to the dmem
14  // MemWrite - write enable to allowed WriteData to overwrite an existing dmem word
15  // PC - the current program count value, goes to imem to fetch instruciton
16  // ALUResult - Result of the ALU operation, sent as address to the dmem
17
18  module arm (
19      input  logic        clk, rst,
20      input  logic [31:0] InstrF,
21      input  logic [31:0] ReadData,
22      output logic [31:0] WriteDataM,
23      output logic [31:0] PC, ALUResultM,
24      output logic        MemWrite
25  );
26
27      // datapath buses and signals
28      logic [31:0] PCPrime, PCPlus4, PCPlus8, PCF; // pc signals
29      logic [ 3:0] RA1, RA2;                   // regfile input addresses
30      logic [31:0] RD1E, RD1D, RD2E, RD2D;                    // raw regfile outputs
31      logic [ 3:0] ALUFlags;                   // alu combinational flag outputs
32      logic [ 3:0] FlagsReg;                   // Flag output from recent CMP
33      logic [31:0] ExtImm, SrcA, SrcB, ExtImmE;         // immediate and alu inputs
34      logic [31:0] ResultW;                     // computed or fetched value to be written
    into regfile or pc
35      logic [31:0] RA1E, RA2E, SrcAE, SrcBE, InstrD, ALUResultE, ALUOutW, WriteDataE,
    ReadDataW;
36
37      // control signals
38      logic PCSrc, MemToReg, ALUSrc, RegWrite, FlagWrite;
39      logic [1:0] RegSrc, ImmSrc, ALUControl;
40      logic PCSrcD, PCSrcE, PCSrcM, PCSrcW;
41      logic RegWriteD, RegWriteE, RegWriteM, RegWriteW;
42      logic MemToRegD, MemToRegE, MemToRegM, MemToRegW;
43      logic MemWriteD, MemWriteE, MemWriteM;
44      logic CondExE;
45      logic [1:0] ALUControlD, ALUControlE;
46      logic BranchD, BranchE;
47      logic ALUSrcD, ALUSrcE;
48      logic StallF, StallD;
49      logic FlushD, FlushE;
50      logic ldrstall;
51      logic PCWrPendingF;
52      logic FlagWriteD, FlagWriteE;
53      logic BranchTakenE;
54      logic [3:0] FlagsE, Flags;
55      logic [3:0] CondE;
56      logic [1:0] ForwardAE, ForwardBE;
57      logic [3:0] WA3E, WA3M, WA3W;
58
59
60      /* 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
61      ** 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
62      ** ports of the register file and alu take some time to study and understand the logic
    and flow of the datapath.
63      */
64      //-------------------------------------------------------------------------------
65      //                                      DATAPATH
66      //-------------------------------------------------------------------------------
```

```
67
68      // Checks if we are branching and change PC accordingly
69       always_comb begin
70         if(BranchTakenE) begin
71            PCPrime = ALUResultE - 8;
72         end else if(PCSrcW) begin
73            PCPrime = ResultW;
74         end else begin
75            PCPrime = PCPlus4;
76         end
77       end
78
79       assign PCPlus4 = PCF + 'd4;                  // default value to access next instruction
80       assign PCPlus8 = PCPlus4 + 'd4;             // value read when reading from reg[15]
81
82       // update the PC, at rst initialize to 0
83       always_ff @(posedge clk) begin
84           if (rst) PC <= '0;
85           else     PC <= PCPrime;
86       end
87
88       // determine the register addresses based on control signals
89       // RegSrc[0] is set if doing a branch instruction
90       // RefSrc[1] is set when doing memory instructions
91       assign RA1 = RegSrc[0] ? 4'd15        : InstrD[19:16];
92       assign RA2 = RegSrc[1] ? InstrD[15:12] : InstrD[ 3: 0];
93
94       // register memory
95       // when instructed, we write into the registers the value we want
96       // also read out any data that we request via our RA1 and RA2
97       reg_file u_reg_file (
98           .clk       (!clk),
99           .wr_en     (RegWriteW),
100          .write_data(ResultW),
101          .write_addr(WA3W),
102          .read_addr1(RA1),
103          .read_addr2(RA2),
104          .read_data1(RD1D),
105          .read_data2(RD2D)
106      );
107
108      // two muxes, put together into an always_comb for clarity
109      // determines which set of instruction bits are used for the immediate
110      always_comb begin
111          if      (ImmSrc == 'b00) ExtImm = {{24{InstrD[7]}},InstrD[7:0]};          // 8 bit
     immediate - reg operations
112          else if (ImmSrc == 'b01) ExtImm = {20'b0, InstrD[11:0]};                 // 12 bit
     immediate - mem operations
113          else                     ExtImm = {{6{InstrD[23]}}, InstrD[23:0], 2'b00}; // 24 bit
     immediate - branch operation
114      end
115
116      assign SrcBE = (ALUSrcE) ? ExtImmE : WriteDataE;
117
118
119
120      //data forwarding
121      // Changes ForwardAE output depending on whether or not
122      // execute stage register matches memory or writeback registers
123      logic MATCH_1E_M, MATCH_2E_M, MATCH_1E_W, MATCH_2E_W;
124      always_comb begin
125        MATCH_1E_M = RA1E == WA3M;
126        MATCH_2E_M = RA2E == WA3M;
127        MATCH_1E_W = RA1E == WA3W;
128        MATCH_2E_W = RA2E == WA3W;
129        if(MATCH_1E_M & RegWriteM) begin
130           ForwardAE = 2'b10;
131        end else if (MATCH_1E_W & RegWriteW) begin
132           ForwardAE = 2'b01;
133        end else begin
134           ForwardAE = 2'b00;
135        end
136        if(MATCH_2E_M & RegWriteM) begin
```

```
137             ForwardBE = 2'b10;
138           end else if (MATCH_2E_W & RegWriteW) begin
139             ForwardBE = 2'b01;
140           end else begin
141             ForwardBE = 2'b00;
142           end
143         end
144
145       //stalling and flushing
146
147         assign PCSrcD = (RegWriteD & (InstrD[15:12] == 4'b1111));
148         assign ldrstall = (MemToRegE) & ((RA1 == WA3E) | (RA2 == WA3E));
149         assign PCWrPendingF = PCSrcD | PCSrcE | PCSrcM;
150         assign StallF = ldrstall | PCWrPendingF;
151         assign FlushD = PCWrPendingF | PCSrcW | BranchTakenE;
152         assign FlushE = ldrstall | BranchTakenE;
153         assign StallD = ldrstall;
154
155         // Forward multiplexer
156         always_comb begin
157           case(ForwardAE)
158           2'b00 : begin
159             SrcAE = (RA1E == 'd15) ? PCPlus8 : RD1E;
160           end
161           2'b01 : begin
162             SrcAE = ResultW; //data forward to end
163           end
164           2'b10 : begin
165             SrcAE = ALUResultM; //data forward to previous alu instruction
166           end
167           default : begin
168             SrcAE = 0;
169           end
170           endcase
171           case(ForwardBE)
172           2'b00 : begin
173             WriteDataE = (RA2E == 'd15) ? PCPlus8 : RD2E;
174           end
175           2'b01 : begin
176             WriteDataE = ResultW; //data forward to end
177           end
178           2'b10 : begin
179             WriteDataE = ALUResultM; //data forward to previous alu instruction
180           end
181           default : begin
182             WriteDataE = 0;
183           end
184           endcase
185         end
186
187         // instruction memory
188         // contained machine code instructions which instruct processor on which operations to
    make
189         // effectively a rom because our processor cannot write to it
190         alu u_alu (
191             .a          (SrcAE),
192             .b          (SrcBE),
193             .ALUControl (ALUControlE),
194             .Result     (ALUResultE),
195             .ALUFlags   (ALUFlags)
196         );
197
198         assign MemWrite = MemWriteM;
199
200         // register 1
201         always_ff@(posedge clk) begin
202             if(rst | FlushD) begin
203                 InstrD <= 0;
204             end else if (StallD) begin
205                 InstrD <= InstrD;
206             end else begin
207                 InstrD <= InstrF;
208             end
```

```
209
210             if(rst) begin
211                PCF <= 0;
212             end else if (StallF) begin
213                PCF <= PCF;
214             end else begin
215                PCF <= PCPrime;
216             end
217         end
218
219         // register 2
220         always_ff@(posedge clk) begin
221             if(rst | FlushE) begin
222                PCSrcE <= 0;
223                RegWriteE <= 0;
224                ALUControlE <= 0;
225                MemToRegE <= 0;
226                MemWriteE <= 0;
227                RD1E <= 0;
228                RD2E <= 0;
229                RA1E <= 0;
230                RA2E <= 0;
231                FlagWriteE <= 0;
232                CondE <= 0;
233                BranchE <= 0;
234                ALUSrcE <= 0;
235                FlagsE <= 0;
236                WA3E <= 0;
237                ExtImmE <= 0;
238             end else begin
239                PCSrcE <= PCSrcD;
240                RegWriteE <= RegWriteD;
241                ALUControlE <= ALUControlD;
242                MemToRegE <= MemToRegD;
243                MemWriteE <= MemWriteD;
244                RD1E <= RD1D;
245                RD2E <= RD2D;
246                RA1E <= RA1;
247                RA2E <= RA2;
248                FlagWriteE <= FlagWriteD;
249                ExtImmE <= ExtImm;
250                BranchE <= BranchD;
251                ALUSrcE <= ALUSrcD;
252                WA3E <=InstrD[15:12];
253                CondE <= InstrD[31:28];
254                if(FlagWriteE) begin
255                   FlagsE <= ALUFlags;
256                end
257             end
258         end
259
260         // register 3
261         always_ff@(posedge clk) begin
262             if(rst) begin
263                PCSrcM <= 0;
264                WA3M <= 0;
265                ALUResultM <= 0;
266                WriteDataM <= 0;
267                RegWriteM <= 0;
268                MemToRegM <= 0;
269                MemWriteM <= 0;
270             end else begin
271                PCSrcM <= PCSrcE & CondExE;
272                WA3M <= WA3E;
273                ALUResultM <= ALUResultE;
274                WriteDataM <= WriteDataE;
275                RegWriteM <= RegWriteE & CondExE;
276                MemToRegM <= MemToRegE;
277                MemWriteM <= MemWriteE & CondExE;
278             end
279         end
280
281         assign BranchTakenE = (BranchE & CondExE);
```

```
282
283         // register 4
284         assign ResultW = (MemToRegW) ? ReadDataW : ALUOutW;
285         always_ff@(posedge clk) begin
286           if(rst) begin
287              PCSrcW <= 0;
288              RegWriteW <= 0;
289              MemToRegW <= 0;
290              ALUOutW <= 0;
291              ReadDataW <= 0;
292              WA3W <= 0;
293           end else begin
294              PCSrcW <= PCSrcM;
295              RegWriteW <= RegWriteM;
296              MemToRegW <= MemToRegM;
297              ALUOutW <= ALUResultM;
298              ReadDataW <= ReadData;
299              WA3W <= WA3M;
300           end
301         end
302
303         logic V = FlagsE[0];
304         logic C = FlagsE[1];
305         logic Z = FlagsE[2];
306         logic N = FlagsE[3];
307
308         always_comb begin
309         case(CondE)
310            4'b0000: begin
311              CondExE = Z;
312            end
313            4'b0001: begin
314              CondExE = !Z;
315            end
316            4'b0010: begin
317              CondExE = C;
318            end
319            4'b0011: begin
320              CondExE = !C;
321            end
322            4'b0100: begin
323              CondExE = N;
324            end
325            4'b0101: begin
326              CondExE = !N;
327            end
328            4'b0110: begin
329              CondExE = V;
330            end
331            4'b0111: begin
332              CondExE = !V;
333            end
334            4'b1000: begin
335              CondExE = !Z&C;
336            end
337            4'b1001: begin
338              CondExE = Z | !C;
339            end
340            4'b1010: begin
341              CondExE = !(N ^ V);
342            end
343            4'b1011: begin
344              CondExE = N ^ V;
345            end
346            4'b1100: begin
347              CondExE = !Z & !(N ^ V);
348            end
349            4'b1101: begin
350              CondExE = Z | (N ^ V);
351            end
352            4'b1110: begin
353              CondExE = 1;
354            end
```

```
355         4'b1111: begin
356           CondExE = 1;
357         end
358         default : begin
359           CondExE = 1;
360         end
361         endcase
362       end
363
364       /* The control conists of a large decoder, which evaluates the top bits of the instruction and produces the control bits
365       ** which become the select bits and write enables of the system. The write enables (RegWrite, MemWrite and PCSrc) are
366       ** especially important because they are representative of your processors current state.
367       */
368       //------------------------------------------------------------------------------
369       //                                      CONTROL
370       //------------------------------------------------------------------------------
371
372
373       always_comb begin
374           casez (InstrD[31:20])
375
376               // ADD (Imm or Reg)
377               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
378                   //PCSrcD    = 0;
379                   MemToRegD = 0;
380                   MemWriteD = 0;
381                   ALUSrcD   = InstrD[25]; // may use immediate
382                   RegWriteD = 1;
383                   RegSrc    = 'b00;
384                   ImmSrc    = 'b00;
385                   ALUControlD = 'b00;
386                   FlagWriteD = 0;
387                   BranchD = 0;
388               end
389
390               // SUB (Imm or Reg)
391               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
392                   //PCSrcD    = 0;
393                   MemToRegD = 0;
394                   MemWriteD = 0;
395                   ALUSrcD   = InstrD[25]; // may use immediate
396                   RegWriteD = 1;
397                   RegSrc    = 'b00;
398                   ImmSrc    = 'b00;
399                   ALUControlD = 'b01;
400                   FlagWriteD = 0;
401                   BranchD = 0;
402               end
403
404               // CMP (Imm or Reg)
405               12'B111000?00101 : begin   // note that we use wildcard "?" in bit 25. That bit decides whether we use immediate or reg, but regardless we sub
406                   //PCSrcD    = 0;
407                   MemToRegD = 0;
408                   MemWriteD = 0;
409                   ALUSrcD   = InstrD[25]; // may use immediate
410                   RegWriteD = 1;
411                   RegSrc    = 'b00;
412                   ImmSrc    = 'b00;
413                   ALUControlD = 'b01;
414                   FlagWriteD = 1;
415                   BranchD = 0;
416               end
417
418               // AND
419               12'b111000000000 : begin
420                   //PCSrcD    = 0;
421                   MemToRegD = 0;
```

```
422                     MemWriteD = 0;
423                     ALUSrcD   = 0;
424                     RegWriteD = 1;
425                     RegSrc    = 'b00;
426                     ImmSrc    = 'b00;    // doesn't matter
427                     ALUControlD = 'b10;
428                     FlagWriteD = 0;
429                     BranchD = 0;
430                 end
431
432                 // ORR
433                 12'b111000011000 : begin
434                     //PCSrcD    = 0;
435                     MemToRegD = 0;
436                     MemWriteD = 0;
437                     ALUSrcD   = 0;
438                     RegWriteD = 1;
439                     RegSrc    = 'b00;
440                     ImmSrc    = 'b00;    // doesn't matter
441                     ALUControlD = 'b11;
442                     FlagWriteD = 0;
443                     BranchD = 0;
444                 end
445
446                 // LDR
447                 12'b111001011001 : begin
448                     //PCSrcD    = 0;
449                     MemToRegD = 1;
450                     MemWriteD = 0;
451                     ALUSrcD   = 1;
452                     RegWriteD = 1;
453                     RegSrc    = 'b10;    // msb doesn't matter
454                     ImmSrc    = 'b01;
455                     ALUControlD = 'b00; // do an add
456                     FlagWriteD = 0;
457                     BranchD = 0;
458                 end
459
460                 // STR
461                 12'b111001011000 : begin
462                     //PCSrcD    = 0;
463                     MemToRegD = 0; // doesn't matter
464                     MemWriteD = 1;
465                     ALUSrcD   = 1;
466                     RegWriteD = 0;
467                     RegSrc    = 'b10;    // msb doesn't matter
468                     ImmSrc    = 'b01;
469                     ALUControlD = 'b00;  // do an add
470                     FlagWriteD = 0;
471                     BranchD = 0;
472                 end
473
474                 // B
475                 12'b????1010???? : begin
476                         case (InstrD[31:28])
477                             4'b1110 : begin
478                                 //PCSrcD    = 1;
479                                 MemToRegD = 0;
480                                 MemWriteD = 0;
481                                 ALUSrcD   = 1;
482                                 RegWriteD = 0;
483                                 RegSrc    = 'b01;
484                                 ImmSrc    = 'b10;
485                                 ALUControlD = 'b00;  // do an add
486                                 FlagWriteD = 0;
487                                 BranchD = 1;
488                             end
489
490                             // equal
491                             4'b0000 : begin
492                                //PCSrcD    = 1;
493                                MemToRegD = 0;
494                                MemWriteD = 0;
```

```
495                             ALUSrcD   = 1;
496                             RegWriteD = 0;
497                             RegSrc    = 'b01;
498                             ImmSrc    = 'b10;
499                             ALUControlD = 'b00;
500                             FlagWriteD = 0;
501                             BranchD = 1;
502                         end
503
504                         // not equal
505                         4'b0001 : begin
506                             //PCSrcD    = 1;
507                             MemToRegD = 0;
508                             MemWriteD = 0;
509                             ALUSrcD   = 1;
510                             RegWriteD = 0;
511                             RegSrc    = 'b01;
512                             ImmSrc    = 'b10;
513                             ALUControlD = 'b00;  // do an add
514                             FlagWriteD = 0;
515                             BranchD = 1;
516                         end
517
518                         // Greater or Equal
519                         4'b1010 : begin
520                             //PCSrcD    = 1;
521                             MemToRegD = 0;
522                             MemWriteD = 0;
523                             ALUSrcD   = 1;
524                             RegWriteD = 0;
525                             RegSrc    = 'b01;
526                             ImmSrc    = 'b10;
527                             ALUControlD = 'b00;  // do an add
528                             FlagWriteD = 0;
529                             BranchD = 1;
530                         end
531
532                         // Greater
533                         4'b1100 : begin
534                             //PCSrcD    = 1;
535                             MemToRegD = 0;
536                             MemWriteD = 0;
537                             ALUSrcD   = 1;
538                             RegWriteD = 0;
539                             RegSrc    = 'b01;
540                             ImmSrc    = 'b10;
541                             ALUControlD = 'b00;  // do an add
542                             FlagWriteD = 0;
543                             BranchD = 1;
544                         end
545
546                         // Less or Equal
547                         4'b1101 : begin
548                             //PCSrcD    = 1;
549                             MemToRegD = 0;
550                             MemWriteD = 0;
551                             ALUSrcD   = 1;
552                             RegWriteD = 0;
553                             RegSrc    = 'b01;
554                             ImmSrc    = 'b10;
555                             ALUControlD = 'b00;  // do an add
556                             FlagWriteD = 0;
557                             BranchD = 1;
558                         end
559
560                         // Less
561                         4'b1011 : begin
562                             //PCSrcD    = 1;
563                             MemToRegD = 0;
564                             MemWriteD = 0;
565                             ALUSrcD   = 1;
566                             RegWriteD = 0;
567                             RegSrc    = 'b01;
```

```
568                             ImmSrc   = 'b10;
569                             ALUControlD = 'b00;  // do an add
570                             FlagWriteD = 0;
571                             BranchD = 1;
572                          end
573
574                          default: begin
575                             //PCSrcD    = 0;
576                             MemToRegD = 0; // doesn't matter
577                             MemWriteD = 0;
578                             ALUSrcD   = 0;
579                             RegWriteD = 0;
580                             RegSrc   = 'b00;
581                             ImmSrc   = 'b00;
582                             ALUControlD = 'b00;  // do an add
583                             FlagWriteD = 0;
584                             BranchD = 0;
585                          end
586                       endcase
587
588
589                end
590
591             default: begin
592                   //PCSrcD    = 0;
593                      MemToRegD = 0;
594                      MemWriteD = 0;
595                      ALUSrcD   = 0;
596                      RegWriteD = 0;
597                      RegSrc   = 'b00;
598                      ImmSrc   = 'b00;
599                      ALUControlD = 'b00;
600                      FlagWriteD = 0;
601                      BranchD = 0;
602             end
603            endcase
604       end
605
606
607   endmodule
```

```
1   // Eugene Ngo
2   // 5/3/23
3   // EE 469
4   // Lab 3
5
6   // alu is a module capable of adding, subtracting, anding and oring
7   // depending on what the control input says,
8   // and reports any flags that may appear
9   `timescale 1ns/10ps
10  module alu(input  logic [31:0] a, b,
11             input  logic [1:0]  ALUControl,
12             output logic [31:0] Result,
13             output logic [3:0]  ALUFlags);
14     logic [31:0] ADD, AND, OR, c0, b1;
15     assign AND = a & b;
16     assign OR = a | b;
17
18     // converts to negative if we are subtracting
19     always_comb begin
20        if (ALUControl[0] == 0) begin
21           b1 <= b;
22        end else begin
23           b1 <= ~b;
24        end
25     end
26
27     // Adder for all bits
28     // does our subtraction or addition for every bit in
29     // out A and B
30     fullAdder firstbit (.A(a[0]),.B(b1[0]),.cin(ALUControl[0]),.sum(ADD[0]),.cout(c0[0]));
31     genvar i;
32     generate
33        for (i = 1; i < 32; i++) begin : gen
34           fullAdder otherbits(.A(a[i]),.B(b1[i]),.cin(c0[i-1]),.sum(ADD[i]),.cout(c0[i]));
35        end
36     endgenerate
37
38     // determines output based on our ALUControl
39     always_comb begin
40        if (ALUControl == 2'b11) begin
41           Result <= OR;
42        end else if (ALUControl == 2'b10) begin
43           Result <= AND;
44        end else begin
45           Result <= ADD;
46        end
47     end
48
49     // Sets all our flags for our computation
50     assign ALUFlags[3] = Result[31];
51     assign ALUFlags[2] = (Result == 0);
52     assign ALUFlags[1] = c0[31] & !ALUControl[1];
53     assign ALUFlags[0] = !(a[31] ^ b[31] ^ ALUControl[0]) & (a[31] ^ ADD[31]) & !ALUControl[1
   ];
54  endmodule
55
56  // alu_testbench tests a variety of different inputs and ouputs
57  // pulled from the alu.tv file.
58  module alu_testbench();
59     logic [31:0] a, b;
60     logic [1:0]  ALUControl;
61     logic [31:0] Result;
62     logic [3:0]  ALUFlags;
63     logic clk;
64     logic [103:0] testvectors [1000:0];
65
66     alu dut (.a,.b,.ALUControl,.Result,.ALUFlags);
67
68     parameter CLOCK_PERIOD=100;
69
70     initial clk = 1;
71     always begin
72        #(CLOCK_PERIOD/2);
```

```
73          clk <= ~clk;
74       end
75
76
77       initial begin
78          $readmemh("alu.tv", testvectors);
79          for (int i = 0; i < 20; i = i + 1) begin
80             {ALUControl,a,b,Result,ALUFlags} = testvectors[i]; @(posedge clk);
81          end
82       end
83    endmodule
```

```
1   // Eugene Ngo
2   // 5/3/23
3   // EE 469
4   // Lab 3
5
6   // alu_testbench tests a variety of different inputs and ouputs
7   // pulled from the alu.tv file.
8   module alu_testbench();
9      logic [31:0] a, b;
10     logic [1:0]  ALUControl;
11     logic [31:0] Result;
12     logic [3:0]  ALUFlags;
13     logic clk;
14     logic [103:0] testvectors [1000:0];
15
16     alu dut (.a,.b,.ALUControl,.Result,.ALUFlags);
17
18     parameter CLOCK_PERIOD=100;
19
20     initial clk = 1;
21     always begin
22        #(CLOCK_PERIOD/2);
23        clk <= ~clk;
24     end
25
26
27     initial begin
28        $readmemh("alu.tv", testvectors);
29        for (int i = 0; i < 20; i = i + 1) begin
30           {ALUControl,a,b,Result,ALUFlags} = testvectors[i]; @(posedge clk);
31        end
32     end
33  endmodule
```