```
1   // Nattapon Oonlamom
2   // 04/21/2023
3   // EE 469
4   // Lab 2: ARM Single-Cycle Processor
5
6   /* ALU - Artihmatic Logic Unit: (32-bit)
7    * The ALU is able to Add, Subtract, Compare (AND, OR)
8    * The ALU also verify Negative, Zero, Carry, oVerflow
9    *
10   * Overall Inputs/Outputs listed below:
11   *    Inputs:  32-bit a, b
12   *             2-bit  ALUControl
13   *    Outputs: 32-bit Result
14   *             4-bit  ALUFlags
15   */
16  module alu (input  logic [31:0] a, b,
17              input  logic [1:0]  ALUControl,
18              output logic [31:0] Result,
19              output logic [3:0]  ALUFlags);
20
21     // Logic for ALU
22     logic [32:0] sum;   // overall sum
23     logic [31:0] temp;  // updated b
24     logic c_out, temp1, temp2;
25
26     // Assign addition with updated b and a carry
27     assign temp  = temp1 ? ~b:b; // store b
28     assign temp1 = (ALUControl == 2'b01) ? 1'b1:1'b0;  // pad for new b
29     assign sum = ({1'b0, a} + {1'b0, temp} + ALUControl[0]); // adder
30     assign temp2 = (~ALUControl[1]) ? sum[32]:1'b0; // Declare variables for c_out
31     assign c_out = temp1 ? temp[31]:temp2;
32
33     // Combinational logics for ALU
34     always_comb begin
35        case (ALUControl)
36           2'b00: Result = sum;   // Add
37           2'b01: Result = sum;   // Sub
38           2'b10: Result = a & b; // AND
39           2'b11: Result = a | b; // OR
40        endcase
41     end
42
43     assign ALUFlags[3] = Result[31];                                  // Negative Flag   (N)
44     assign ALUFlags[2] = (Result == 32'b0);                           // Zero Flag       (Z)
45     assign ALUFlags[1] = c_out;                                       // Carry Flag      (C)
46     assign ALUFlags[0] = ((~ALUControl[1]) &&                         // oVerflow flag   (V)
47                           (~(ALUControl[0] ^ a[31] ^ b[31])) &&
48                           (a[31] ^ Result[31]));
49  endmodule
```

```
1   // Nattapon Oonlamom
2   // 04/21/2023
3   // EE 469
4   // Lab 2: ARM Single-Cycle Processor
5
6   /*
7    * Testbench for the alu.sv to test for correctly functioning ALUControl
8    * and displaying the correct result as expected
9    */
10  module alu_testbench();
11     // logic to simulate
12     logic clk;
13     logic [31:0]  a, b;
14     logic [1:0]   ALUControl;
15     logic [31:0]  Result;
16     logic [3:0]   ALUFlags;
17     logic [103:0] testvectors [1000:0];
18
19     // device under test
20     alu dut(.*);
21
22     // clock setup
23     parameter clock_period = 100;
24
25     initial clk = 1;
26     always begin
27        #(clock_period /2);
28        clk <= ~clk;
29     end
30
31     // initial simulation
32     initial begin
33        $readmemh("alu.tv", testvectors);
34
35        for(int i = 0; i < 20; i = i + 1) begin
36           {ALUControl, a, b, Result, ALUFlags} = testvectors[i];   @(posedge clk);
37        end
38     end
39  endmodule
```

```
1   // Nattapon Oonlamom
2   // 04/21/2023
3   // EE 469
4   // Lab 2: ARM Single-Cycle Processor
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] Instr,
21      input  logic [31:0] ReadData,
22      output logic [31:0] WriteData,
23      output logic [31:0] PC, ALUResult,
24      output logic        MemWrite
25  );
26
27      // datapath buses and signals
28      logic [31:0] PCPrime, PCPlus4, PCPlus8; // pc signals
29      logic [ 3:0] RA1, RA2;                  // regfile input addresses
30      logic [31:0] RD1, RD2;                  // raw regfile outputs
31      logic [ 3:0] ALUFlags;                  // alu combinational flag outputs
32      logic [31:0] ExtImm, SrcA, SrcB;        // immediate and alu inputs
33      logic [31:0] Result;                    // computed or fetched value to be written into
    regfile or pc
34
35      // control signals
36      logic PCSrc, MemtoReg, ALUSrc, RegWrite;
37      logic [1:0] RegSrc, ImmSrc, ALUControl;
38
39
40      /* 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
41      ** 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
42      ** ports of the register file and alu take some time to study and understand the logic
    and flow of the datapath.
43      */
44      //-------------------------------------------------------------------------------
45      //                                    DATAPATH
46      //-------------------------------------------------------------------------------
47
48
49      assign PCPrime = PCSrc ? Result : PCPlus4;  // mux, use either default or newly
    computed value
50      assign PCPlus4 = PC + 'd4;                  // default value to access next instruction
51      assign PCPlus8 = PCPlus4 + 'd4;             // value read when reading from reg[15]
52
53      // update the PC, at rst initialize to 0
54      always_ff @(posedge clk) begin
55          if (rst) PC <= '0;
56          else     PC <= PCPrime;
57      end
58
59      // determine the register addresses based on control signals
60      // RegSrc[0] is set if doing a branch instruction
61      // RefSrc[1] is set when doing memory instructions
62      assign RA1 = RegSrc[0] ? 4'd15        : Instr[19:16];
63      assign RA2 = RegSrc[1] ? Instr[15:12] : Instr[ 3: 0];
64
65      // TODO: insert your reg file here
66      // TODO: instantiates 16x32-bit register file
67      //       with two asynchronous read ports
68      //       to hold values for computation of the processor
69      reg_file u_reg_file (
```

```
70            .clk       (clk),
71            .wr_en     (RegWrite),
72            .write_data(Result),
73            .write_addr(Instr[15:12]),
74            .read_addr1(RA1),
75            .read_addr2(RA2),
76            .read_data1(RD1),
77            .read_data2(RD2)
78        );
79
80        // Logic for the new register
81        logic [3:0] FlagsReg;
82        logic FlagWrite;
83
84        // Store new flags into the registor
85        always_ff @(posedge clk) begin
86          if (FlagWrite) FlagsReg <= ALUFlags;
87          else FlagsReg <= FlagsReg;
88        end
89
90        // Declare values for possible conditions
91        logic EQ, NE, GE, GT, LE, LT;
92        always_comb begin
93          EQ = (FlagsReg[2]);
94          NE = (~FlagsReg[2]);
95          GE = (~(FlagsReg[3] ^ FlagsReg[0]));
96          GT = ((~FlagsReg[2]) & GE);
97          LE = (FlagsReg[3] ^ FlagsReg[0]);
98          LT = (FlagsReg[2] | LE);
99        end
100
101       // two muxes, put together into an always_comb for clarity
102       // determines which set of instruction bits are used for the immediate
103       always_comb begin
104           if      (ImmSrc == 'b00) ExtImm = {{24{Instr[7]}},Instr[7:0]};          // 8 bit
      immediate - reg operations
105           else if (ImmSrc == 'b01) ExtImm = {20'b0, Instr[11:0]};                 // 12 bit
      immediate - mem operations
106           else                     ExtImm = {{6{Instr[23]}}, Instr[23:0], 2'b00}; // 24 bit
      immediate - branch operation
107       end
108
109       // WriteData and SrcA are direct outputs of the register file, wheras SrcB is chosen
      between reg file output and the immediate
110       assign WriteData = (RA2 == 'd15) ? PCPlus8 : RD2;           // substitute the 15th
      regfile register for PC
111       assign SrcA      = (RA1 == 'd15) ? PCPlus8 : RD1;           // substitute the 15th
      regfile register for PC
112       assign SrcB      = ALUSrc        ? ExtImm  : WriteData;     // determine alu operand to
      be either from reg file or from immediate
113
114       // TODO: insert your alu here
115       // TODO: instantiates ALU file,
116       //       processes AND, OR, ADD, or SUB,
117       //       outputs depending on ALUControl,
118       //       computes flags for Zero(Z), Negative(N), Carry(C), oVerflow(V)
119       alu u_alu (
120           .a          (SrcA),
121           .b          (SrcB),
122           .ALUControl (ALUControl),
123           .Result     (ALUResult),
124           .ALUFlags   (ALUFlags)
125       );
126
127       // determine the result to run back to PC or the register file based on whether we used
      a memory instruction
128       assign Result = MemtoReg ? ReadData : ALUResult;    // determine whether final
      writeback result is from dmemory or alu
129
130
131       /* The control conists of a large decoder, which evaluates the top bits of the
      instruction and produces the control bits
132       ** which become the select bits and write enables of the system. The write enables
      (RegWrite, MemWrite and PCSrc) are
133       ** especially important because they are representative of your processors current
      state.
```

```
134         */
135         //------------------------------------------------------------------------------
136         //                                      CONTROL
137         //------------------------------------------------------------------------------
138
139         always_comb begin
140             casez (Instr[27:20])
141
142                 // ADD (Imm or Reg)
143                 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
144                     PCSrc    = 0;
145                     MemtoReg = 0;
146                     MemWrite = 0;
147                     ALUSrc   = Instr[25]; // may use immediate
148                     RegWrite = 1;
149                     RegSrc   = 'b00;
150                     ImmSrc   = 'b00;
151                     ALUControl = 'b00;
152                     FlagWrite  = 0;
153                 end
154
155                 // SUB (Imm or Reg)
156                 8'b00?_0010_0 : begin   // note that we use wildcard "?" in bit 25. That bit decides whether we use immediate or reg, but regardless we sub
157                     PCSrc    = 0;
158                     MemtoReg = 0;
159                     MemWrite = 0;
160                     ALUSrc   = Instr[25]; // may use immediate
161                     RegWrite = 1;
162                     RegSrc   = 'b00;
163                     ImmSrc   = 'b00;
164                     ALUControl = 'b01;
165                     FlagWrite  = 0;
166                 end
167
168                 // AND
169                 8'b000_0000_0 : begin
170                     PCSrc    = 0;
171                     MemtoReg = 0;
172                     MemWrite = 0;
173                     ALUSrc   = 0;
174                     RegWrite = 1;
175                     RegSrc   = 'b00;
176                     ImmSrc   = 'b00;    // doesn't matter
177                     ALUControl = 'b10;
178                     FlagWrite  = 0;
179                 end
180
181                 // ORR
182                 8'b000_1100_0 : begin
183                     PCSrc    = 0;
184                     MemtoReg = 0;
185                     MemWrite = 0;
186                     ALUSrc   = 0;
187                     RegWrite = 1;
188                     RegSrc   = 'b00;
189                     ImmSrc   = 'b00;    // doesn't matter
190                     ALUControl = 'b11;
191                     FlagWrite  = 0;
192                 end
193
194                 // LDR
195                 8'b010_1100_1 : begin
196                     PCSrc    = 0;
197                     MemtoReg = 1;
198                     MemWrite = 0;
199                     ALUSrc   = 1;
200                     RegWrite = 1;
201                     RegSrc   = 'b10;    // msb doesn't matter
202                     ImmSrc   = 'b01;
203                     ALUControl = 'b00;  // do an add
204                     FlagWrite  = 0;
205                 end
206
207                 // STR
```

```
208                 8'b010_1100_0 : begin
209                     PCSrc    = 0;
210                     MemtoReg = 0; // doesn't matter
211                     MemWrite = 1;
212                     ALUSrc   = 1;
213                     RegWrite = 0;
214                     RegSrc   = 'b10;    // msb doesn't matter
215                     ImmSrc   = 'b01;
216                     ALUControl = 'b00;  // do an add
217                     FlagWrite  = 0;
218                 end
219 
220                 // B
221                 8'b1010_???? : begin
222                         if (Instr[31:28] == 4'b0000) begin // equal
223                             if (EQ) PCSrc = 1;
224                             else    PCSrc = 0;
225                         end
226                         else if (Instr[31:28] == 4'b0001) begin // unequal
227                             if (NE) PCSrc = 1;
228                             else    PCSrc = 0;
229                         end
230                         else if (Instr[31:28] == 4'b1010) begin // greater than || equal
231                             if (GE) PCSrc = 1;
232                             else    PCSrc = 0;
233                         end
234                         else if (Instr[31:28] == 4'b1100) begin // greater than
235                             if (GT) PCSrc = 1;
236                             else    PCSrc = 0;
237                         end
238                         else if (Instr[31:28] == 4'b1101) begin // less than || equal
239                             if (LE) PCSrc = 1;
240                             else    PCSrc = 0;
241                         end
242                         else if (Instr[31:28] == 4'b1011) begin // less than
243                             if (LT) PCSrc = 1;
244                             else    PCSrc = 0;
245                         end
246                         else   // default case
247                            PCSrc    = 1;
248                            MemtoReg = 0;
249                            MemWrite = 0;
250                            ALUSrc   = 1;
251                            RegWrite = 0;
252                            RegSrc   = 'b01;
253                            ImmSrc   = 'b10;
254                            ALUControl = 'b00;  // do an add
255                            FlagWrite  = 0;
256                 end
257 
258                 // SUBS/CMP for immediate or reg
259                 8'b00?_0010_1 : begin   // "?" decides between immediate or reg
260                         PCSrc    = 0;
261                         MemtoReg = 0;
262                         MemWrite = 0;
263                         ALUSrc   = Instr[25]; // immediate
264                         RegWrite = 1;
265                         RegSrc   = 'b00;
266                         ImmSrc   = 'b00;
267                         ALUControl = 'b01;
268                         FlagWrite  = 1;
269                 end
270 
271                 default: begin
272                         PCSrc    = 0;
273                         MemtoReg = 0; // doesn't matter
274                         MemWrite = 0;
275                         ALUSrc   = 0;
276                         RegWrite = 0;
277                         RegSrc   = 'b00;
278                         ImmSrc   = 'b00;
279                         ALUControl = 'b00;  // do an add
280                         FlagWrite = 0;
281                 end
282             endcase
283         end
```

```
284
285     endmodule
286
```

```
1   // Nattapon Oonlamom
2   // 04/21/2023
3   // EE 469
4   // Lab 2: ARM Single-Cycle Processor
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   // Nattapon Oonlamom
2   // 04/21/2023
3   // EE 469
4   // Lab 2: ARM Single-Cycle Processor
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
    alter
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("memfile.dat", memory);    // Task 1
28      initial $readmemb("memfile2.dat", memory); // Task 2
29
30
31      assign instr = memory[addr[31:2]]; // word aligned, drops bottom 2 bits
32
33  endmodule
```

```
1   // ADD R - 1110000010000AAAADDDD00000000BBBB
2   // ADD I - 1110001010000AAAADDDD0000IIIIIIII
3   // SUB R - 1110000001000AAAADDDD00000000BBBB
4   // SUB I - 1110001001000AAAADDDD0000IIIIIIII
5   // AND   - 1110000000000AAAADDDD00000000BBBB
6   // ORR   - 1110000110000AAAADDDD00000000BBBB
7   // LDR   - 1110010110001AAAADDDDIIIIIIIIIIII
8   // STR   - 1110010110000AAAADDDDIIIIIIIIIIII
9   // B     - 11101010IIIIIIIIIIIIIIIIIIIIIIII
10
11
12  11100010100011110000000000000000 // MAIN       ADD R0, R15, #0          0
13  11100000010000000001000000000000 //            SUB R1, R0, R0           4
14  11100010100000010010000000001010 //            ADD R2, R1, #10          8
15  11100000100000000011000000000010 //            ADD R3, R0, R2           12
16  11100010010000100100000000000011 //            SUB R4, R2, #3           16
17  11100000010000110101000000000100 //            SUB R5, R3, R4           20
18  11100001100001000110000000000101 //            ORR R6, R4, R5           24
19  11100000000001100111000000000101 //            AND R7, R6, R5           28
20  11100101100000010111000000000000 //            STR R7, [R1, #0]         32
21  11101010000000000000000000000001 //            B SKIP                   36
22  11100101100000010001000000000000 //            STR R1, [R1, #0]         40
23  11101010000000000000000000000000 //            B LOOP                   44
24  11100101100100011000000000000000 // SKIP       LDR R8, [R1, #0]         48
25  11101010111111111111111111111110 // LOOP       B LOOP                   52
```

```
// ADD R - 1110000010000AAAADDDD00000000BBBB
// ADD I - 1110001010000AAAADDDD0000IIIIIIII
// SUB R - 1110000001000AAAADDDD00000000BBBB
// SUB I - 1110001001000AAAADDDD0000IIIIIIII
// CMP R - 1110000001010AAAADDDD00000000BBBB
// CMP I - 1110001001010AAAADDDD0000IIIIIIII
// AND   - 1110000000000AAAADDDD00000000BBBB
// ORR   - 1110000110000AAAADDDD00000000BBBB
// LDR   - 1110010110010AAAADDDDIIIIIIIIIIII
// STR   - 1110010110000AAAADDDDIIIIIIIIIIII
//COND1010IIIIIIIIIIIIIIIIIIIIIIII

// 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
11100000010000000011000000000010 //              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
```

```
1   // Nattapon Oonlamom
2   // 04/21/2023
3   // EE 469
4   // Lab 2: ARM Single-Cycle Processor
5
6   /* 16x32 Register file module for specified data and address bus widths.
7    * 2 Asynchronous read port (read_addr1 -> read_data1, read_addr2 -> read_data2)
8    * and synchronous write port (write_data -> write_addr if wr_en)
9    *
10   * Overall Inputs/Outputs listed below:
11   *    Inputs:  32-bit write_data
12   *             4-bit  write_addr,read_addr1, read_addr2
13   *             1-bit  clk
14   *    Outputs: 32-bit read_data1, read_data2
15   */
16  module reg_file(input  logic clk, wr_en,
17                      input  logic [31:0] write_data,
18                      input  logic [3:0]  write_addr,
19                      input  logic [3:0]  read_addr1, read_addr2,
20                      output logic [31:0] read_data1, read_data2);
21
22     // array declaration (registers)
23     logic [15:0] memory [31:0];
24
25     // write operation (synchronous)
26     always_ff @(posedge clk) begin
27        if (wr_en) begin
28           memory[write_addr] <= write_data;
29        end
30     end
31
32     // read operation (asynchronous)
33     assign read_data1 = memory[read_addr1];
34     assign read_data2 = memory[read_addr2];
35
36  endmodule // reg_file
37
38
```

```
1   // Nattapon Oonlamom
2   // 04/21/2023
3   // EE 469
4   // Lab 2: ARM Single-Cycle Processor
5
6   /*
7    * Testbench for the reg_file.sv to test for correctly functions read and write
8    */
9   module reg_file_testbench();
10     // logic to simulate
11     logic clk, wr_en;
12     logic [31:0] write_data;
13     logic [3:0]  write_addr;
14     logic [3:0]  read_addr1, read_addr2;
15     logic [31:0] read_data1, read_data2;
16
17     // device under test
18     reg_file dut(.*);
19
20     // clock setup
21     parameter clock_period = 100;
22
23     initial begin
24        clk <= 0;
25        forever #(clock_period /2) clk <= ~clk;
26     end
27
28     // initial simulation
29     initial begin
30        write_data <= 4'b0001; wr_en <= 0; read_addr1 <= 2'b00; read_addr2 <= 2'b00;
   write_addr <= 2'b00;     @(posedge clk);
31        write_data <= 4'b0001; wr_en <= 0; read_addr1 <= 2'b00; read_addr2 <= 2'b00;
   write_addr <= 2'b00;     @(posedge clk);
32        write_data <= 4'b0001; wr_en <= 1; read_addr1 <= 2'b00; read_addr2 <= 2'b00;
   write_addr <= 2'b00;     @(posedge clk);
33        write_data <= 4'b0010; wr_en <= 1; read_addr1 <= 2'b00; read_addr2 <= 2'b00;
   write_addr <= 2'b00;     @(posedge clk);
34        write_data <= 4'b0010; wr_en <= 0; read_addr1 <= 2'b00; read_addr2 <= 2'b00;
   write_addr <= 2'b00;     @(posedge clk);
35        write_data <= 4'b0010; wr_en <= 0; read_addr1 <= 2'b00; read_addr2 <= 2'b00;
   write_addr <= 2'b00;     @(posedge clk);
36        write_data <= 4'b0011; wr_en <= 1; read_addr1 <= 2'b00; read_addr2 <= 2'b01;
   write_addr <= 2'b01;     @(posedge clk);
37        write_data <= 4'b0011; wr_en <= 0; read_addr1 <= 2'b00; read_addr2 <= 2'b01;
   write_addr <= 2'b01;     @(posedge clk);
38        write_data <= 4'b0011; wr_en <= 0; read_addr1 <= 2'b00; read_addr2 <= 2'b01;
   write_addr <= 2'b01;     @(posedge clk);
39        $stop;
40     end
41
42  endmodule
```

```
1   // Nattapon Oonlamom
2   // 04/21/2023
3   // EE 469
4   // Lab 2: ARM Single-Cycle Processor
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  module testbench();
11
12      // system signals
13      logic clk, rst;
14
15      // generate clock with 100ps clk period
16      initial begin
17          clk = 1;
18          forever #50 clk = ~clk;
19      end
20
21      // processor instantion. within is the processor as well as imem and dmem
22      top cpu (.clk(clk), .rst(rst));
23
24      initial begin
25          // start with a basic reset
26          rst = 1; @(posedge clk);
27          rst <= 0; @(posedge clk);
28
29          // repeat for 50 cycles. Not all 50 are necessary, however a loop at the end of the
    program will keep anything weird from happening
30          repeat(50) @(posedge clk);
31
32          // basic checking to ensure the right final answer is achieved. These DO NOT prove
    your system works. A more careful look at your
33          // simulation and code will be made.
34
35          // task 1:
36          // assert(cpu.processor.u_reg_file.memory[8] == 32'd11) $display("Task 1 Passed");
37          // else                                                 $display("Task 1 Failed");
38
39          // task 2:
40          assert(cpu.processor.u_reg_file.memory[8] == 32'd1)  $display("Task 2 Passed");
41          else                                                 $display("Task 2 Failed");
42
43          $stop;
44      end
45
46  endmodule
```

```
1   // Nattapon Oonlamom
2   // 04/21/2023
3   // EE 469
4   // Lab 2: ARM Single-Cycle Processor
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  module top(
15      input logic clk, rst
16  );
17
18      // processor io signals
19      logic [31:0] Instr;
20      logic [31:0] ReadData;
21      logic [31:0] WriteData;
22      logic [31:0] PC, ALUResult;
23      logic        MemWrite;
24
25      // our single cycle arm processor
26      arm processor (
27          .clk        (clk        ),
28          .rst        (rst        ),
29          .Instr      (Instr      ),
30          .ReadData   (ReadData   ),
31          .WriteData  (WriteData  ),
32          .PC         (PC         ),
33          .ALUResult  (ALUResult  ),
34          .MemWrite   (MemWrite   )
35      );
36
37      // instruction memory
38      // contained machine code instructions which instruct processor on which operations to
    make
39      // effectively a rom because our processor cannot write to it
40      imem imemory (
41          .addr   (PC     ),
42          .instr  (Instr  )
43      );
44
45      // data memory
46      // containes data accessible by the processor through ldr and str commands
47      dmem dmemory (
48          .clk     (clk       ),
49          .wr_en   (MemWrite  ),
50          .addr    (ALUResult ),
51          .wr_data (WriteData ),
52          .rd_data (ReadData  )
53      );
54
55  endmodule
56
```