```
1   // Arjun Simha and Jarod Marshel
2   // 5/5/2023
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   // clk - system clock
8   // rst - system reset
9   // Instr - incoming 32 bit instruction from imem, contains opcode, condition, addresses and or immediates
10  // ReadData - data read out of the dmem
11  // WriteData - data to be written to the dmem
12  // MemWrite - write enable to allowed WriteData to overwrite an existing dmem word
13  // PC - the current program count value, goes to imem to fetch instruciton
14  // ALUResult - result of the ALU operation, sent as address to the dmem
15  // Input bitwidth: 1 + 1 + 32 + 32 = 66 bits
16  // Output bitwidth: 32 + 32 + 1 = 65 bits
17  // Total bitwidth: 131 bits
18  module arm (
19      input  logic        clk, rst,
20      input  logic [31:0] Instr_Fetch,
21      input  logic [31:0] ReadData_Mem,
22      output logic [31:0] WriteData_Mem,
23      output logic [31:0] PC_Fetch, ALUResult_Mem,
24      output logic        MemWrite_Mem
25  );
26      // Logic for different stages
27      logic MemWrite_Dec, MemWrite_Exe;
28      logic [31:0] ALUOut_Mem;
29      logic [31:0] ALUResult_Exe, ALUOut_Wr, WriteData_Exe;
30      logic [31:0] Instr_Dec, ReadData_Wr;
31
32      // Logic for datapath buses and datapath signals
33      logic [31:0] PCPrime, PCPlus4_Fetch, PCPlus8_Dec, PCPlus8_Exe; // PC signals
34      logic [3:0] RA1_Dec, RA2_Dec, RA1_Exe, RA2_Exe; // Regfile input addresses
35      logic [31:0] RD1_Dec, RD2_Dec, RD1_Exe, RD2_Exe; // Raw regfile outputs
36      logic [3:0] ALUFlags; // ALU flags
37      logic [31:0] ExtImm_Dec, ExtImm_Exe, SrcA_Exe, SrcB_Exe; // Immediate/ALU inputs
38      logic [31:0] Result_Wr; // Computed value written to regfile
39
40      // Logic control signals
41      logic PCSrc_Dec, MemtoReg_Dec, ALUSrc_Dec, RegWrite_Dec, Branch_Dec;
42      logic [1:0] RegSrc_Dec, ImmSrc_Dec, ALUControl_Dec;
43      //More control logic that represents the control logic in different stages.
44      logic PCSrc_Exe, MemtoReg_Exe, ALUSrc_Exe, RegWrite_Exe, Branch_Exe;
45      logic [1:0] RegSrc_Exe, ALUControl_Exe;
46      logic PCSrc_Mem, MemtoReg_Mem, RegWrite_Mem;
47      logic PCSrc_Wr, MemtoReg_Wr, RegWrite_Wr;
48      //Logic that is for the writing address
49      logic [3:0] WA3_Dec, WA3_Exe, WA3_Mem, WA3_Wr;
50      assign WA3_Dec = rst? 4'h0 : Instr_Dec[15:12];
51      //Logic that is for the conditional execution
52      logic [3:0] Cond_Dec, Cond_Exe;
```

```
53      //The D is the instruction and the E is pipelined.
54      assign Cond_Dec = rst? 4'h0 : Instr_Dec[31:28];
55
56      // Pipelining and hazard control signals
57      logic Stall_Fetch, Stall_Dec, Flush_Dec, Flush_Exe, PCWrPending_Fetch, Match_12D_Exe, ldrstall;
58      logic Match_1E_Mem, Match_2E_Mem, Match_1E_Wr, Match_2E_Wr;
59      logic [1:0] ForwardA_Exe, ForwardB_Exe;
60      logic BranchTaken_Exe, BranchTaken_Fetch;
61
62      // Flag register vars
63      logic [3:0] FlagsReg;
64      logic FlagWrite_Dec, FlagWrite_Exe;
65      logic path_en;
66
67
68      /* 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
69      ** 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
70      ** ports of the register file and alu take some time to study and understand the logic and flow of the datapath.
71      */
72      //-------------------------------------------------------------------------------
73      //                                      DATAPATH
74      //-------------------------------------------------------------------------------
75
76
77      // MUX
78      logic [31:0] temp;
79      assign temp = PCSrc_Wr ? Result_Wr : PCPlus4_Fetch; // mux
80      assign PCPrime = BranchTaken_Exe ? ALUResult_Exe : temp; // mux
81      assign PCPlus4_Fetch = PC_Fetch + 'd4;
82      assign PCPlus8_Dec = PCPlus4_Fetch;
83
84      // Update the PC
85      always_ff @(posedge clk) begin
86          if (rst) PC_Fetch <= '0;
87          else if (Stall_Fetch) PC_Fetch <= PC_Fetch;
88          else PC_Fetch <= PCPrime;
89      end
90
91      // The control signals determine the register addresses. If a branch instruction is being executed, then RegSrc[0] is
    set.
92      // If a memory instruction is being executed, then RegSrc[1] is set.
93      assign RA1_Dec = RegSrc_Dec[0] ? 4'd15 : Instr_Dec[19:16];
94      assign RA2_Dec = RegSrc_Dec[1] ? Instr_Dec[15:12] : Instr_Dec[3:0];
95
96      // Create an instance of reg_file with the following assignments: clk is connected to the clock signal,
97      // wr_en is connected to RegWrite_Wr, write_data is connected to Result_Wr, write_addr is connected to WA3_Wr,
98      // read_addr1 is connected to RA1_Dec, read_addr2 is connected to RA2_Dec, read_data1 is connected to rd_temp1, and
    read_data2 is connected to rd_temp2.
99      logic [31:0] rd_temp1, rd_temp2, int_R1, int_R2;
100     reg_file u_reg_file (
101         .clk       (clk),
```

```
102             .wr_en     (RegWrite_Wr),
103             .write_data(Result_Wr),
104             .write_addr(WA3_Wr),
105             .read_addr1(RA1_Dec),
106             .read_addr2(RA2_Dec),
107             .read_data1(rd_temp1),
108             .read_data2(rd_temp2)
109         );
110
111         // Reg file logic
112         always begin
113           if (rst) begin
114               RD1_Dec = 32'h00000000;
115               RD2_Dec = 32'h00000000;
116           end else begin
117               if (RA1_Dec == 4'hf) begin
118                   RD1_Dec = PCPlus8_Dec;
119               end else if (rd_temp1[31] == 1 | rd_temp1[31] == 0) begin
120                   RD1_Dec = rd_temp1;
121               end else begin
122                   RD1_Dec = 32'h00000000;
123               end
124
125               if (RA2_Dec == 4'hf) begin
126                   RD2_Dec = PCPlus8_Dec;
127               end else if (rd_temp2[31] == 1 | rd_temp2[31] == 0) begin
128                   RD1_Dec = rd_temp2;
129               end else begin
130                   RD2_Dec = 32'h00000000;
131               end
132           end
133         end
134
135         // Two multiplexers are combined into an always_comb block to determine what the immediate value is.
136         always_comb begin
137             if (ImmSrc_Dec == 'b00) begin
138                 ExtImm_Dec = {{24{Instr_Dec[7]}},Instr_Dec[7:0]}; // 8 bit immediate - register
139             end
140             else if (ImmSrc_Dec == 'b01) begin
141                 ExtImm_Dec = {20'b0, Instr_Dec[11:0]}; // 12 bit immediate - memory
142             end
143             else begin
144                 ExtImm_Dec = {{6{Instr_Dec[23]}}, Instr_Dec[23:0], 2'b00}; // 24 bit immediate - branch
145             end
146         end
147
148         // While WriteData and SrcA are outputs of the register file that are directly accessible, the choice for the second
    operand of an
149         // arithmetic or logical operation, SrcB, can be made between using the output of the register file or an immediate
    value.
150         logic [31:0] tempA;
151         logic [31:0] tempB;
152         // The value of temp2 to WriteDataE. The values of SrcA and SrcB are determined based on a two-input multiplexer used
```

```
    by the
153     // hazard unit. Since the value that goes into SrcB goes through a second multiplexer, a second temporary variable is
    used to store
154     // this value. Additionally, a temporary variable is used for SrcA since it also goes through a decision process. The
    commented-out
155     // line assigns PCPlus8E to WriteDataE if the 15th register in the register file is being accessed.
156     always_comb begin
157        if (ForwardA_Exe[0]&(~ForwardA_Exe[1])) begin
158          tempA = Result_Wr;
159        end else if ((~ForwardA_Exe[0])&(ForwardA_Exe[1])) begin
160          tempA = ALUOut_Mem;
161        end else begin
162          tempA = RD1_Exe;
163        end
164     end
165
166     always_comb begin
167        if (ForwardB_Exe[0]&(~ForwardB_Exe[1])) begin
168          tempB = Result_Wr;
169        end else if ((~ForwardB_Exe[0])&(ForwardB_Exe[1])) begin
170          tempB = ALUOut_Mem;
171        end else begin
172          tempB = RD2_Exe;
173        end
174     end
175
176     // assign WriteData, SrcA and SrcB.
177     assign WriteData = tempB;
178     assign SrcA_Exe = rst? 32'h00000000 : tempA;
179     assign SrcB = rst? 32'h00000000 : (ALUSrc_Exe ? ExtImm_Exe : tempB);
180
181     // Create an instance of ALU with A being assigned to SrcA, B is assigned to SrcB, control is assigned to ALUControl.
182     // Result is assigned to ALUResult, and flags is assigned to ALUFlags.
183     logic [31:0] ALUResult_temp;
184     alu u_alu (
185         .A          (SrcA_Exe),
186         .B          (SrcB_Exe),
187         .control (ALUControl_Exe),
188         .result     (ALUResult_temp),
189         .flags   (ALUFlags)
190     );
191
192     assign ALUResult_Exe = rst? 32'h00000000 : ALUResult_temp;
193
194     // Decide whether to direct the result to the program counter (PC) or the register file, depending on whether a
    memory instruction was used.
195     assign Result_Wr = MemtoReg_Wr ? ReadData_Wr : ALUOut_Wr;    // Determine whether the final writeback result is from
    data memory or the arithmetic logic unit (ALU).
196
197
198     /* The hazard control stage determines and assigns the flush and stall logics, which will be used in the subsequent
    pipelining stage.
199     *  These logic values are based on the lecture.
```

```
200       */
201       //-------------------------------------------------------------------------------
202       //                                      HAZARD CONTROL
203       //-------------------------------------------------------------------------------
204
205
206       // Assign vars
207       assign BranchTaken_Exe = Branch_Exe & path_en;
208       assign BranchTaken_Fetch = BranchTaken_Exe;
209       assign Match_1E_Mem = rst? 1'b0 : (RA1_Exe == WA3_Mem);
210       assign Match_2E_Mem = rst? 1'b0 : (RA2_Exe == WA3_Mem);
211       assign Match_1E_Wr = rst? 1'b0 : (RA1_Exe == WA3_Wr);
212       assign Match_2E_Wr = rst? 1'b0 : (RA2_Exe == WA3_Wr);
213
214       // Assigns the forward A values
215       always_comb begin
216          if (Match_1E_Mem & RegWrite_Mem)
217            ForwardA_Exe = 2'b10;
218          else if (Match_1E_Wr & RegWrite_Wr)
219            ForwardA_Exe = 2'b01;
220          else
221            ForwardA_Exe = 2'b00;
222       end
223
224       // Assigns the forward B values
225       always_comb begin
226          if (Match_2E_Mem & RegWrite_Mem)
227            ForwardB_Exe = 2'b10;
228          else if (Match_2E_Wr & RegWrite_Wr)
229            ForwardB_Exe = 2'b01;
230          else
231            ForwardB_Exe = 2'b00;
232       end
233
234       assign Match_12D_Exe = rst? 1'b0 : ((RA1_Dec == WA3_Exe) + (RA2_Dec == WA3_Exe));
235       assign ldrstall = rst? 1'b0 : (Match_12D_Exe & MemtoReg_Exe);
236       assign Stall_Fetch = rst? 1'b0 : (ldrstall + PCWrPending_Fetch);
237       assign Stall_Dec = rst? 1'b0 : ldrstall;
238       assign Flush_Exe = rst? 1'b0 : (ldrstall + BranchTaken_Fetch);
239       assign Flush_Dec = rst? 1'b0 : (PCWrPending_Fetch + PCSrc_Wr + BranchTaken_Fetch);
240       assign PCWrPending_Fetch = rst? 1'b0 : (PCSrc_Dec + PCSrc_Exe + PCSrc_Mem);
241
242
243       /* In the pipelining stage, four large register columns are created using one-bit instantiations of the
      pipeline_register module.
244       * The comment also explains that signal renaming will be performed to use appropriate names throughout the code. For
      example, instead
245       * of using SrcA and SrcB, SrcAE and SrcBE will be used. Then, in this stage, the registers will be used to create
      these renamed signals.
246       */
247       //-------------------------------------------------------------------------------
248       //                                      PIPELINING
249       //-------------------------------------------------------------------------------
```

```
250
251
252         // First pipeline register
253         genvar y;
254         generate
255             for (y = 0; y < 32; y++) begin : firstpipe
256                 // Pass in the values to the pipeline_register modules.
257                 pipeline_register p2 (.rst, .q(Instr_Dec[y]), .d(Instr_Fetch[y]), .clk, .f(Flush_Dec), .s(Stall_Dec));
258             end
259         endgenerate
260
261         // Second pipeline register
262         pipeline_register p3 (.rst, .q(PCSrc_Exe), .d(PCSrc_Dec), .clk, .f(Flush_Exe), .s(1'b0));
263         pipeline_register p4 (.rst, .q(RegWrite_Exe), .d(RegWrite_Dec), .clk, .f(Flush_Exe), .s(1'b0));
264         pipeline_register p5 (.rst, .q(MemtoReg_Exe), .d(MemtoReg_Dec), .clk, .f(Flush_Exe), .s(1'b0));
265         pipeline_register p6 (.rst, .q(MemWrite_Exe), .d(MemWrite_Dec), .clk, .f(Flush_Exe), .s(1'b0));
266         pipeline_register p7 (.rst, .q(ALUControl_Exe[0]), .d(ALUControl_Dec[0]), .clk, .f(Flush_Exe), .s(1'b0));
267         pipeline_register p7_2 (.rst, .q(ALUControl_Exe[1]), .d(ALUControl_Dec[1]), .clk, .f(Flush_Exe), .s(1'b0));
268         pipeline_register p8 (.rst, .q(Branch_Exe), .d(Branch_Dec), .clk, .f(Flush_Exe), .s(1'b0));
269         pipeline_register p9 (.rst, .q(ALUSrc_Exe), .d(ALUSrc_Dec), .clk, .f(Flush_Exe), .s(1'b0));
270         pipeline_register p10 (.rst, .q(FlagWrite_Exe), .d(FlagWrite_Dec), .clk, .f(Flush_Exe), .s(1'b0));
271         pipeline_register p11 (.rst, .q(Cond_Exe[0]), .d(Cond_Dec[0]), .clk, .f(Flush_Exe), .s(1'b0));
272         pipeline_register p12 (.rst, .q(Cond_Exe[1]), .d(Cond_Dec[1]), .clk, .f(Flush_Exe), .s(1'b0));
273         pipeline_register p13 (.rst, .q(Cond_Exe[2]), .d(Cond_Dec[2]), .clk, .f(Flush_Exe), .s(1'b0));
274         pipeline_register p14 (.rst, .q(Cond_Exe[3]), .d(Cond_Dec[3]), .clk, .f(Flush_Exe), .s(1'b0));
275
276
277         // Pipeline for datapath
278         pipeline_register p17 (.rst, .q(WA3_Exe[3]), .d(WA3_Dec[3]), .clk, .f(Flush_Exe), .s(1'b0));
279         pipeline_register p18 (.rst, .q(WA3_Exe[2]), .d(WA3_Dec[2]), .clk, .f(Flush_Exe), .s(1'b0));
280         pipeline_register p19 (.rst, .q(WA3_Exe[1]), .d(WA3_Dec[1]), .clk, .f(Flush_Exe), .s(1'b0));
281         pipeline_register p20 (.rst, .q(WA3_Exe[0]), .d(WA3_Dec[0]), .clk, .f(Flush_Exe), .s(1'b0));
282         // Flop RA1D and RA2D
283         pipeline_register p47 (.rst, .q(RA1_Exe[3]), .d(RA1_Dec[3]), .clk, .f(Flush_Exe), .s(1'b0));
284         pipeline_register p48 (.rst, .q(RA1_Exe[2]), .d(RA1_Dec[2]), .clk, .f(Flush_Exe), .s(1'b0));
285         pipeline_register p49 (.rst, .q(RA1_Exe[1]), .d(RA1_Dec[1]), .clk, .f(Flush_Exe), .s(1'b0));
286         pipeline_register p50 (.rst, .q(RA1_Exe[0]), .d(RA1_Dec[0]), .clk, .f(Flush_Exe), .s(1'b0));
287         pipeline_register p57 (.rst, .q(RA2_Exe[3]), .d(RA2_Dec[3]), .clk, .f(Flush_Exe), .s(1'b0));
288         pipeline_register p58 (.rst, .q(RA2_Exe[2]), .d(RA2_Dec[2]), .clk, .f(Flush_Exe), .s(1'b0));
289         pipeline_register p59 (.rst, .q(RA2_Exe[1]), .d(RA2_Dec[1]), .clk, .f(Flush_Exe), .s(1'b0));
290         pipeline_register p60 (.rst, .q(RA2_Exe[0]), .d(RA2_Dec[0]), .clk, .f(Flush_Exe), .s(1'b0));
291         // Flop RD1, RD2, ExtImm, and PCPlus8
292         genvar z;
293         generate
294             for (z = 0; z < 32; z++) begin : secondpipe
295                 // Pass in values to pipeline registers
296                 pipeline_register p21 (.rst, .q(RD1_Exe[z]), .d(RD1_Dec[z]), .clk, .f(Flush_Exe), .s(1'b0));
297                 pipeline_register p22 (.rst, .q(RD2_Exe[z]), .d(RD2_Dec[z]), .clk, .f(Flush_Exe), .s(1'b0));
298                 pipeline_register p23 (.rst, .q(ExtImm_Exe[z]), .d(ExtImm_Dec[z]), .clk, .f(Flush_Exe), .s(1'b0));
299                 pipeline_register p23_1 (.rst, .q(PCPlus8_Exe[z]), .d(PCPlus8_Dec[z]), .clk, .f(Flush_Exe), .s(1'b0));
300             end
301         endgenerate
302
```

```
303
304       // Third pipeline register
305       pipeline_register p25 (.rst, .q(PCSrcM), .d(PCSrcE&enable_path), .clk, .f(1'b0), .s(1'b0));
306       pipeline_register p26 (.rst, .q(RegWriteM), .d(RegWriteE&enable_path), .clk, .f(1'b0), .s(1'b0));
307       pipeline_register p27 (.rst, .q(MemtoRegM), .d(MemtoReg_Exe), .clk, .f(1'b0), .s(1'b0));
308       pipeline_register p28 (.rst, .q(MemWriteM), .d(MemWriteE&enable_path), .clk, .f(1'b0), .s(1'b0));
309       pipeline_register p29 (.rst, .q(WA3_Mem[3]), .d(WA3_Exe[3]), .clk, .f(1'b0), .s(1'b0));
310       pipeline_register p29_2 (.rst, .q(WA3_Mem[2]), .d(WA3_Exe[2]), .clk, .f(1'b0), .s(1'b0));
311       pipeline_register p29_3 (.rst, .q(WA3_Mem[1]), .d(WA3_Exe[1]), .clk, .f(1'b0), .s(1'b0));
312       pipeline_register p29_4 (.rst, .q(WA3_Mem[0]), .d(WA3_Exe[0]), .clk, .f(1'b0), .s(1'b0));
313       // Flop ALUResult and WriteData
314       genvar m;
315       generate
316       for (m = 0; m < 32; m++) begin : thirdpipe
317            // Pass in the values to the pipeline_register modules.
318            pipeline_register p29_5 (.rst, .q(ALUResult_Mem[m]), .d(ALUResult_Exe[m]), .clk, .f(1'b0), .s(1'b0));
319            pipeline_register p29_6 (.rst, .q(WriteData_Mem[m]), .d(WriteData_Exe[m]), .clk, .f(1'b0), .s(1'b0));
320        end
321       endgenerate
322
323       // Fourth pipeline register
324       pipeline_register p30 (.rst, .q(PCSrcW), .d(PCSrcM), .clk, .f(1'b0), .s(1'b0));
325       pipeline_register p31 (.rst, .q(RegWriteW), .d(RegWriteM), .clk, .f(1'b0), .s(1'b0));
326       pipeline_register p32 (.rst, .q(MemtoRegW), .d(MemtoRegM), .clk, .f(1'b0), .s(1'b0));
327       pipeline_register p33 (.rst, .q(WA3_Wr[3]), .d(WA3_Mem[3]), .clk, .f(1'b0), .s(1'b0));
328       pipeline_register p34 (.rst, .q(WA3_Wr[2]), .d(WA3_Mem[2]), .clk, .f(1'b0), .s(1'b0));
329       pipeline_register p35 (.rst, .q(WA3_Wr[1]), .d(WA3_Mem[1]), .clk, .f(1'b0), .s(1'b0));
330       pipeline_register p36 (.rst, .q(WA3_Wr[0]), .d(WA3_Mem[0]), .clk, .f(1'b0), .s(1'b0));
331       // Flop ALUOut and ReadData
332       assign ALUOut_Mem = ALUResult_Mem;
333       genvar n;
334       generate
335       for (n = 0; n < 32; n++) begin : fourpipe
336            // Pass in the values to the pipeline_register modules.
337            pipeline_register p37 (.rst, .q(ALUOut_Wr[n]), .d(ALUOut_Mem[n]), .clk, .f(1'b0), .s(1'b0));
338            pipeline_register p38 (.rst, .q(ReadData_Wr[n]), .d(ReadData_Mem[n]), .clk, .f(1'b0), .s(1'b0));
339        end
340       endgenerate
341
342
343       /* A significant component of the system is the extensive decoder that assesses the upper portion of the instruction
      and generates control bits.
344       * These control bits serve as the select bits and write enables for the system. Among these, the write enables
      (RegWrite, MemWrite, and PCSrc)
345       * are of utmost importance as they reflect the current state of the processor.
346       */
347       //-------------------------------------------------------------------------------
348       //                                      CONTROL
349       //-------------------------------------------------------------------------------
350
351
352       // Flag register logic
353       assign FlagWrite_Dec = rst? 1'b0 : Instr_Dec[20];
```

```
354       always_ff @(FlagWrite_Exe) begin
355         if (rst) begin
356            FlagsReg <= 4'b0000;
357         end else begin
358            FlagsReg <= ALUFlags;
359         end
360       end
361
362       // Table 3: Flag logic
363       always_comb begin
364           case (Cond_Exe)
365               4'b1110: begin
366                  // Unconditional: set enable to 1
367                  path_en = 1;
368               end
369               4'b0000: begin
370                  // Equal to: zero flag (true)
371                  path_en = ((FlagsReg[2]));
372               end
373               4'b0001: begin
374                  // Not equal to: zero flag (false)
375                  path_en = (~(FlagsReg[2]));
376               end
377               4'b1010: begin
378                  // Greater than or equal to: negative flag (false)
379                  path_en = (~(FlagsReg[3]));
380               end
381               4'b1100: begin
382                  // Greater than: negative and zero flag (false)
383                  path_en = (~(FlagsReg[3]) & (~(FlagsReg[2])));
384               end
385               4'b1101: begin
386                  // Less than or equal to: negative flag or zero flag is (true)
387                  path_en = ((FlagsReg[3]) | ((FlagsReg[2])));
388               end
389               4'b1011: begin
390                  // Less than: negative (true), zero (false)
391                  path_en = ((FlagsReg[3]) & (~(FlagsReg[2])));
392               end
393                  // DefaultL: set enable to 0
394               default: begin
395                  path_en = 0;
396               end
397           endcase
398       end
399
400
401       always_comb begin
402           casez (Instr_Dec[27:20])
403
404               // ADD (Imm or Reg)
405               8'b00?_0100_0 : begin
406                   Branch_Dec = 0;
```

```
407                 PCSrc_Dec = 0;
408                 MemtoReg_Dec = 0;
409                 MemWrite_Dec = 0;
410                 ALUSrc_Dec = Instr_Dec[25];
411                 RegWrite_Dec = 1;
412                 RegSrc_Dec = 'b00;
413                 ImmSrc_Dec = 'b00;
414                 ALUControl_Dec = 'b00;
415             end
416
417             // SUB (Imm or Reg)
418             8'b00?_0010_0 : begin
419                 Branch_Dec = 0;
420                 PCSrc_Dec = 0;
421                 MemtoReg_Dec = 0;
422                 MemWrite_Dec = 0;
423                 ALUSrc_Dec = Instr_Dec[25];
424                 RegWrite_Dec = 1;
425                 RegSrc_Dec = 'b00;
426                 ImmSrc_Dec = 'b00;
427                 ALUControl_Dec = 'b01;
428             end
429
430
431             // CMP (Imm or Reg)
432             8'b00?_0010_1 : begin
433                 Branch_Dec = 0;
434                 PCSrc_Dec = 0;
435                 MemtoReg_Dec = 0;
436                 MemWrite_Dec = 0;
437                 ALUSrc_Dec = Instr_Dec[25];
438                 RegWrite_Dec = 1;
439                 RegSrc_Dec = 'b00;
440                 ImmSrc_Dec = 'b00;
441                 ALUControl_Dec = 'b01;
442             end
443
444             // AND
445             8'b000_0000_0 : begin
446                 Branch_Dec = 0;
447                 PCSrc_Dec = 0;
448                 MemtoReg_Dec = 0;
449                 MemWrite_Dec = 0;
450                 ALUSrc_Dec = 0;
451                 RegWrite_Dec = 1;
452                 RegSrc_Dec = 'b00;
453                 ImmSrc_Dec = 'b00;
454                 ALUControl_Dec = 'b10;
455             end
456
457             // OR
458             8'b000_1100_0 : begin
459                 Branch_Dec = 0;
```

```
460                 PCSrc_Dec = 0;
461                 MemtoReg_Dec = 0;
462                 MemWrite_Dec = 0;
463                 ALUSrc_Dec = 0;
464                 RegWrite_Dec = 1;
465                 RegSrc_Dec = 'b00;
466                 ImmSrc_Dec = 'b00;
467                 ALUControl_Dec = 'b11;
468             end
469 
470             // LDR
471             8'b010_1100_1 : begin
472                 Branch_Dec = 0;
473                 PCSrc_Dec = 0;
474                 MemtoReg_Dec = 1;
475                 MemWrite_Dec = 0;
476                 ALUSrc_Dec = 1;
477                 RegWrite_Dec = 1;
478                 RegSrc_Dec = 'b10;
479                 ImmSrc_Dec = 'b01;
480                 ALUControl_Dec = 'b00;
481             end
482 
483             // STR
484             8'b010_1100_0 : begin
485                 Branch_Dec = 0;
486                 PCSrc_Dec = 0;
487                 MemtoReg_Dec = 0;
488                 MemWrite_Dec = 1;
489                 ALUSrc_Dec = 1;
490                 RegWrite_Dec = 0;
491                 RegSrc_Dec = 'b10;
492                 ImmSrc_Dec = 'b01;
493                 ALUControl_Dec = 'b00;
494             end
495 
496             // B
497             8'b1010_???? : begin
498                 Branch_Dec = 1;
499                 PCSrc_Dec = 1;
500                 MemtoReg_Dec = 0;
501                 MemWrite_Dec = 0;
502                 ALUSrc_Dec = 1;
503                 RegWrite_Dec = 0;
504                 RegSrc_Dec = 'b01;
505                 ImmSrc_Dec = 'b01;
506                 ALUControl_Dec = 'b00;
507             end
508 
509             default: begin
510                 Branch_Dec = 0;
511                 PCSrc_Dec = 0;
512                 MemtoReg_Dec = 0;
```

```
513                     MemWrite_Dec = 0;
514                     ALUSrc_Dec = 0;
515                     RegWrite_Dec = 0;
516                     RegSrc_Dec = 'b00;
517                     ImmSrc_Dec = 'b00;
518                     ALUControl_Dec = 'b00;
519                 end
520             endcase
521         end
522
523     endmodule
524
```