Aaron Hong (ahong02)
Stephen Macris (smacris)
EE 469
9 May 2023

Lab 3 Report

Procedure

In this lab, the single cycle processor was upgraded to a pipelined processor. Stalling, flushing, and forwarding logic were added to handle data and control hazards.

Results

```
11100000010011110000000000001111    // 0  MAIN  SUB  R0 R15 R15    R0 = 0
11100010100000000001000000000001    // 4        ADD  R1 R0 #1     R1 = 1
11100001100000000010000000000001    // 8        ORR  R2 R0 R1     R2 = 1
11100010100000000010000000000010    // 12       ADD  R2 R0 #2     R2 = 2
11100010010100100000000000000000    // 16       SUBS R0 R2 #0     R0 = 2
00001010000000000000000000000001    // 20       BEQ  TAG1
11100000000000100010000000000000    // 24       AND  R2 R2 R0     R2 = 2
11100000000000100001000000000000    // 28       AND  R1 R2 R0     R1 = 2
11100000100000011001000000000000    // 32 TAG1  ADD  R9 R1 R0     R9 = 4
11100101100000000100100000001001    // 36       STR  R9 [R0, #9]
11100101100100000011000000001001    // 40       LDR  R3 [R0, #9]  R3 = 4
11100000000000011001000000000010    // 44       AND  R2 R3 R2     R2 = 0
```

Figure 1: Test assembly instructions with expected register outputs

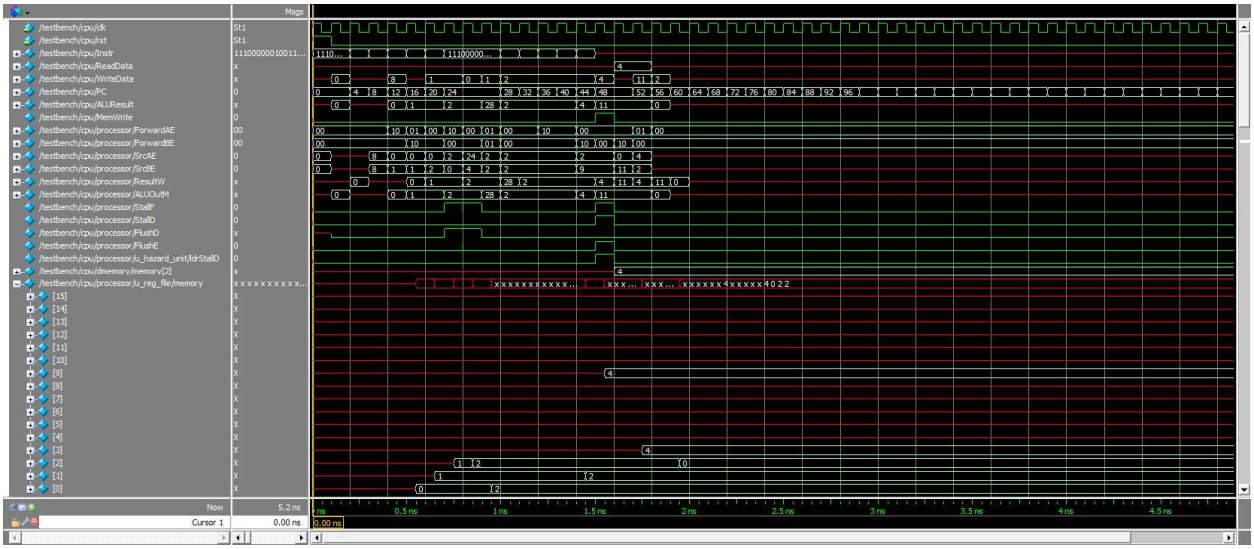

Figure 2: Simulation waveform

As shown in Figure 1 and 2, the pipelined processor is executing the instructions correctly. Stalling, flushing, and forwarding processes are also demonstrated. Figure 2 will be investigated more thoroughly in the following sections. Figures 3, 4, and 5 are magnified portions of Figure 2.

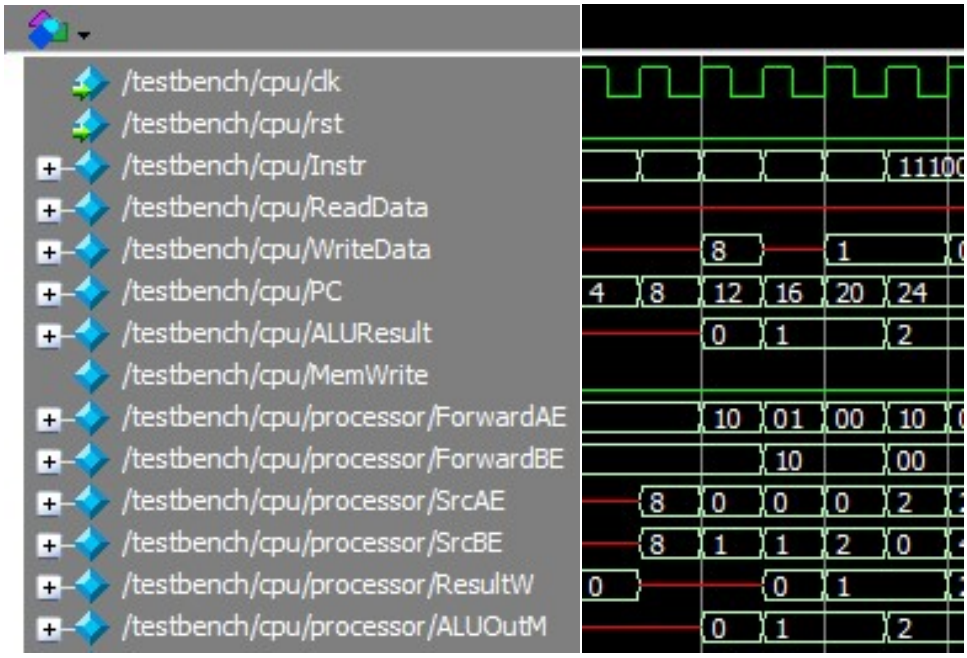

Figure 3: Forwarding signals

When PC=12, instruction 4 is in the execution stage and instruction 0 is in the memory stage. Since source register A of instruction 4 is the same as the destination register of instruction 0, there is a match between execute and memory stages and ForwardAE is set to 10. Hence, the data from the memory stage is forwarded to the execution stage: SrcAE = ALUOutM.

When PC=16, instruction 8 is in the execution stage and instruction 0 is in the writeback stage. Since source register A of instruction 8 is the same as the destination register of instruction 0, there is a match between execute and writeback stages and ForwardAE is set to 01. Hence, the data from the writeback stage is forwarded to the execution stage: SrcAE = ResultW.

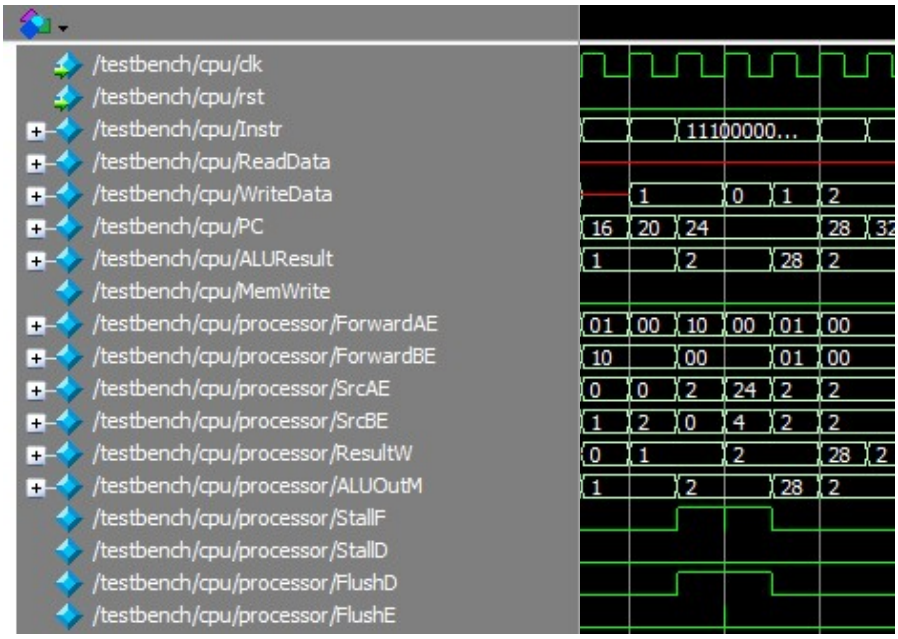

Figure 4: Branch flushing

Figure 4 shows that the two cycles of instruction are being flushed when the branch instruction (instruction 20) enters the decode stage.

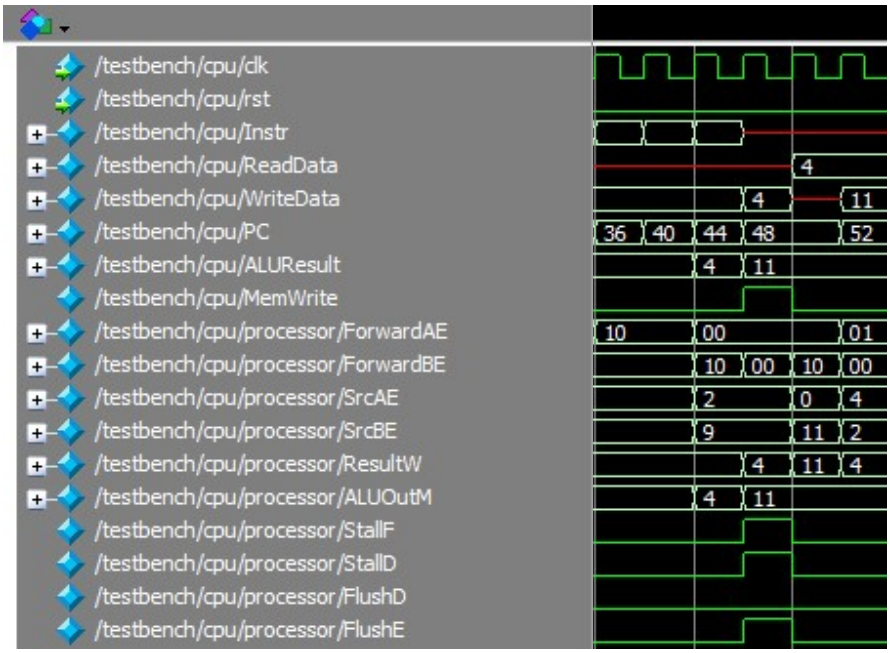

Figure 5: Memory stalling

Figure 5 shows that the processor stalled for the LDR instruction (instruction 40) to finish before executing instruction 44, which accessed the destination register of instruction 40.

Appendix

```
1   /* 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.
2   */
3
4   // clk - system clock
5   // rst - system reset
6   // Instr - incoming 32 bit instruction from imem, contains opcode, condition, addresses and
    or immediates
7   // ReadData - data read out of the dmem
8   // WriteData - data to be written to the dmem
9   // MemW - write enable to allowed WriteData to overwrite an existing dmem word
10  // PC - the current program count value, goes to imem to fetch instruciton
11  // ALUResult - result of the ALU operation, sent as address to the dmem
12
13  module arm (
14      input  logic        clk, rst,
15      input  logic [31:0] Instr,
16      input  logic [31:0] ReadData,
17      output logic [31:0] WriteData,
18      output logic [31:0] PC, ALUResult,
19      output logic        MemWrite
20  );
21
22      // datapath buses and signals
23      logic [31:0] PCF, PCPrime, PCPlus4F, PCPlus8D; // pc signals
24      logic [ 3:0] RA1D, RA2D, RA1E, RA2E;                    // regfile input addresses
25      logic [31:0] RD1D, RD2D, RD1E, RD2E;                    // raw regfile outputs
26      logic [ 3:0] ALUFlags, FlagsD, FlagsE;                  // alu combinational flag outputs
27      logic [31:0] ExtImmD, ExtImmE, SrcAE, SrcBE;          // immediate and alu inputs
28      logic [31:0] ResultW;                       // computed or fetched value to be written
    into regfile or pc
29
30      // control signals
31      logic PCSrcD, PCSrcE, PCSrcM, PCSrcW, PCWrPendingF;
32      logic RegWriteD, RegWriteE, RegWriteM, RegWriteW;
33      logic MemtoRegD, MemtoRegE, MemtoRegM, MemtoRegW;
34      logic MemWriteD, MemWriteE, MemWriteM;
35      logic [1:0] ALUControlD, ALUControlE;
36      logic BranchD, BranchE;
37      logic ALUSrcD, ALUSrcE;
38      logic [1:0] ImmSrcD;
39      logic Match_1E_M, Match_2E_M;
40      logic Match_1E_W, Match_2E_W;
41      logic Match_12D_E;
42      logic BranchTakenE, StallF, StallD, FlushD, FlushE, ldrStallD, CondEx, CondExE;
43      logic [1:0] FlagWriteD, FlagWriteE, RegSrcD;
44      logic [1:0] ForwardAE, ForwardBE;
45      logic [3:0] CondE, WA3E, WA3M, WA3W;
46      logic [31:0] ALUOutW, ALUOutM;
47      logic [31:0] InstrF, InstrD;
48      logic [31:0] ReadDataM, ReadDataW;
49      logic [31:0] WriteDataE, WriteDataM;
50      logic [31:0] ALUResultE;
51
52
53      /* 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
54      ** 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
55      ** ports of the register file and alu take some time to study and understand the logic
    and flow of the datapath.
56      */
57      //------------------------------------------------------------------------------
58      //                                    DATAPATH
59      //------------------------------------------------------------------------------
60
61      // connect module inputs and outputs to datapath
62      assign InstrF = Instr;
63      assign ReadDataM = ReadData;
64      assign WriteData = WriteDataM;
65      assign MemWrite = MemWriteM;
66      assign PC = PCF;
67      assign ALUResult = ALUOutM;
68
69      assign PCPrime = BranchTakenE ? ALUResultE : (PCSrcW ? ResultW : PCPlus4F);  // mux,
    use either default or newly computed value
```

```
70        assign PCPlus4F = PCF + 'd4;                    // default value to access next instruction
71        assign PCPlus8D = PCPlus4F;              // value read when reading from reg[15]
72
73        // update the PC, at rst initialize to 0
74        always_ff @(posedge clk) begin
75            if (rst)          PCF <= '0;
76            else if (!StallF) PCF <= PCPrime;
77            else              PCF <= PCF;
78        end
79
80        // determine the register addresses based on control signals
81        // RegSrcD[0] is set if doing a branch instruction
82        // RefSrc[1] is set when doing memory instructions
83        assign RA1D = RegSrcD[0] ? 4'd15         : InstrD[19:16];
84        assign RA2D = RegSrcD[1] ? InstrD[15:12] : InstrD[ 3: 0];
85
86        // Instantiates a register file to hold values.
87        reg_file u_reg_file (
88            .clk        (!clk),
89            .wr_en      (RegWriteW),
90            .write_data (ResultW),
91            .write_addr (WA3W),
92            .read_addr1 (RA1D),
93            .read_addr2 (RA2D),
94            .read_data1 (RD1D),
95            .read_data2 (RD2D)
96        );
97
98        // two muxes, put together into an always_comb for clarity
99        // determines which set of instruction bits are used for the immediate
100       always_comb begin
101           if      (ImmSrcD == 'b00) ExtImmD = {{24{InstrD[7]}},InstrD[7:0]};          // 8
      bit immediate - reg operations
102           else if (ImmSrcD == 'b01) ExtImmD = {20'b0, InstrD[11:0]};                 // 12
      bit immediate - mem operations
103           else                      ExtImmD = {{6{InstrD[23]}}, InstrD[23:0], 2'b00}; // 24
      bit immediate - branch operation
104       end
105
106       // WriteData and SrcA are direct outputs of the register file, wheras SrcB is chosen
      between reg file output and the immediate
107       assign SrcBE = ALUSrcE ? ExtImmE : WriteDataE;     // determine alu operand to be
      either from reg file or from immediate
108       // Depending on forwarding control signal from hazard unit, choose appropriate ALU
      operand
109       always_comb begin
110         case (ForwardAE)
111           2'b00: SrcAE = (RA1E == 'd15) ? (BranchTakenE ? PCPlus8D : PCF) : RD1E; //
      substitute the 15th regfile register for PC
112           2'b01: SrcAE = ResultW;
113           2'b10: SrcAE = ALUOutM;
114           default: SrcAE = RD1E;
115         endcase
116
117         case (ForwardBE)
118           2'b00: WriteDataE = (RA2E == 'd15) ? (BranchTakenE ? PCPlus8D : PCF) : RD2E; //
      substitute the 15th regfile register for PC
119           2'b01: WriteDataE = ResultW;
120           2'b10: WriteDataE = ALUOutM;
121           default: WriteDataE = RD2E;
122         endcase
123       end
124
125
126       // Instantiates an alu module to do arithmetic operations.
127       alu u_alu (
128           .a          (SrcAE),
129           .b          (SrcBE),
130           .ALUControl (ALUControlE),
131           .Result     (ALUResultE),
132           .ALUFlags   (ALUFlags)
133       );
134
135       // determine the result to run back to PC or the register file based on whether we used
      a memory instruction
136       assign ResultW = MemtoRegW ? ReadDataW : ALUOutW;    // determine whether final
```

```
    writeback result is from dmemory or alu
137
138         // input signals for the hazard unit
139         assign Match_1E_M = (RA1E == WA3M);
140         assign Match_2E_M = (RA2E == WA3M);
141         assign Match_1E_W = (RA1E == WA3W);
142         assign Match_2E_W = (RA2E == WA3W);
143         assign Match_12D_E = ((RA1D == WA3E) + (RA2D == WA3E));
144         assign PCWrPendingF = (PCSrcD + PCSrcE + PCSrcM) & !BranchTakenE;
145         assign BranchTakenE = BranchE & CondExE;
146
147         // Instantiate a conditional unit to manage flags register and determine instruction
    execution
148         cond_unit u_cond_unit (
149               .cond       (CondE),
150               .flags      (FlagsE),
151               .ALUFlags   (ALUFlags),
152               .flag_write (FlagWriteE),
153               .flags_out  (FlagsD),
154               .cond_ex    (CondExE)
155         );
156
157         // Instantiate a hazard unit to provide stalling, flushing, and forwarding control
    signals
158         hazard_unit u_hazard_unit (
159               .Match_1E_M (Match_1E_M),
160               .Match_2E_M (Match_2E_M),
161               .Match_1E_W (Match_1E_W),
162               .Match_2E_W (Match_2E_W),
163               .Match_12D_E(Match_12D_E),
164               .RegWriteM  (RegWriteM),
165               .RegWriteW  (RegWriteW),
166               .MemtoRegE  (MemtoRegE),
167               .BranchTakenE (BranchTakenE),
168               .PCWrPendingF (PCWrPendingF),
169               .PCSrcW (PCSrcW),
170               .ForwardAE  (ForwardAE),
171               .ForwardBE  (ForwardBE),
172               .StallF (StallF),
173               .StallD (StallD),
174               .FlushD (FlushD),
175               .FlushE (FlushE)
176         );
177
178         // Synchronously move signals along the datapath
179         always_ff @(posedge clk) begin
180               //Fetch to Decode
181               if (FlushD) InstrD <= 32'b0;
182               else if (!rst && !StallD) InstrD <= InstrF;
183
184               //Decode to Execute
185               if (!FlushE) begin
186                  PCSrcE <= PCSrcD;
187                  RegWriteE <= RegWriteD;
188                  MemtoRegE <= MemtoRegD;
189                  MemWriteE <= MemWriteD;
190                  ALUControlE <= ALUControlD;
191                  BranchE <= BranchD;
192                  ALUSrcE <= ALUSrcD;
193                  FlagWriteE <= FlagWriteD;
194                  CondE <= InstrD[31:28];
195                  FlagsE <= FlagsD;
196                  RD1E <= RD1D;
197                  RD2E <= RD2D;
198                  RA1E <= RA1D;
199                  RA2E <= RA2D;
200                  WA3E <= InstrD[15:12];
201                  ExtImmE <= ExtImmD;
202               end else begin
203                  PCSrcE <= 0;
204                  RegWriteE <= 0;
205                  MemtoRegE <= 0;
206                  MemWriteE <= 0;
207                  ALUControlE <= 0;
208                  BranchE <= 0;
209                  ALUSrcE <= 0;
```

```
210                 FlagWriteE <= 0;
211                 CondE <= 4'b1111;
212                 FlagsE <= 0;
213                 RD1E <= 0;
214                 RD2E <= 0;
215                 WA3E <= 0;
216                 ExtImmE <= 0;
217             end
218 
219             //Execute to Memory
220             if (!BranchTakenE) begin
221                 PCSrcM <= PCSrcE & CondExE;
222                 RegWriteM <= RegWriteE & CondExE;
223                 MemtoRegM <= MemtoRegE;
224                 MemWriteM <= MemWriteE & CondExE;
225                 WriteDataM <= WriteDataE;
226                 ALUOutM <= ALUResultE;
227                 WA3M <= WA3E;
228             end
229 
230             //Memory to Writeback
231             ReadDataW <= ReadDataM;
232             ALUOutW <= ALUOutM;
233             WA3W <= WA3M;
234             PCSrcW <= PCSrcM;
235             RegWriteW <= RegWriteM;
236             MemtoRegW <= MemtoRegM;
237         end
238 
239 
240         /* The control conists of a large decoder, which evaluates the top bits of the instruction and produces the control bits
241         ** which become the select bits and write enables of the system. The write enables (RegW, MemW and PCS) are
242         ** especially important because they are representative of your processors current state.
243         */
244         //------------------------------------------------------------------------------
245         //                                      CONTROL
246         //------------------------------------------------------------------------------
247         always_comb begin
248             casez (InstrD[27:20])
249 
250                 // ADD (Imm or Reg)
251                 8'b00?_0100_? : begin   // note that we use wildcard "?" in bit 25. That bit decides whether we use immediate or reg, but regardless we add
252                     PCSrcD    = 0;
253                     MemtoRegD = 0;
254                     MemWriteD = 0;
255                     ALUSrcD   = InstrD[25]; // may use immediate
256                     RegWriteD = 1;
257                     RegSrcD   = 'b00;
258                     ImmSrcD   = 'b00;
259                     ALUControlD = 'b00;
260                     FlagWriteD = {InstrD[20], InstrD[20]};
261                     BranchD = 0;
262                 end
263 
264                 // SUB/CMP (Imm or Reg)
265                 8'b00?_0010_? : begin   // note that we use wildcard "?" in bit 25. That bit decides whether we use immediate or reg, but regardless we sub
266                     PCSrcD    = 0;
267                     MemtoRegD = 0;
268                     MemWriteD = 0;
269                     ALUSrcD   = InstrD[25]; // may use immediate
270                     RegWriteD = 1;
271                     RegSrcD   = 'b00;
272                     ImmSrcD   = 'b00;
273                     ALUControlD = 'b01;
274                     FlagWriteD = {InstrD[20], InstrD[20]};
275                     BranchD = 0;
276                 end
277 
278                 // AND
279                 8'b000_0000_? : begin
280                     PCSrcD    = 0;
```

```
281                     MemtoRegD = 0;
282                     MemWriteD = 0;
283                     ALUSrcD   = 0;
284                     RegWriteD = 1;
285                     RegSrcD   = 'b00;
286                     ImmSrcD   = 'b00;    // doesn't matter
287                     ALUControlD = 'b10;
288                     FlagWriteD = {InstrD[20], 1'b0};
289                     BranchD = 0;
290                     //FlagWriteE = 00;
291                 end
292 
293                 // ORR
294                 8'b000_1100_? : begin
295                     PCSrcD    = 0;
296                     MemtoRegD = 0;
297                     MemWriteD = 0;
298                     ALUSrcD   = 0;
299                     RegWriteD = 1;
300                     RegSrcD   = 'b00;
301                     ImmSrcD   = 'b00;    // doesn't matter
302                     ALUControlD = 'b11;
303                     FlagWriteD = {InstrD[20], 1'b0};
304                     BranchD = 0;
305                 end
306 
307                 // LDR
308                 8'b010_1100_1 : begin
309                     PCSrcD    = 0;
310                     MemtoRegD = 1;
311                     MemWriteD = 0;
312                     ALUSrcD   = 1;
313                     RegWriteD = 1;
314                     RegSrcD   = 'b10;    // msb doesn't matter
315                     ImmSrcD   = 'b01;
316                     ALUControlD = 'b00;  // do an add
317                     FlagWriteD = 00;
318                     BranchD = 0;
319                 end
320 
321                 // STR
322                 8'b010_1100_0 : begin
323                     PCSrcD    = 0;
324                     MemtoRegD = 0; // doesn't matter
325                     MemWriteD = 1;
326                     ALUSrcD   = 1;
327                     RegWriteD = 0;
328                     RegSrcD   = 'b10;    // msb doesn't matter
329                     ImmSrcD   = 'b01;
330                     ALUControlD = 'b00;  // do an add
331                     FlagWriteD = 00;
332                     BranchD = 0;
333                 end
334 
335                 // B
336                 8'b1010_???? : begin
337                    if(CondExE) begin
338                        PCSrcD    = 1;
339                        MemtoRegD = 0;
340                        MemWriteD = 0;
341                        ALUSrcD   = 1;
342                        RegWriteD = 0;
343                        RegSrcD   = 'b01;
344                        ImmSrcD   = 'b10;
345                        ALUControlD = 'b00;  // do an add
346                        FlagWriteD = 00;
347                        BranchD = 1;
348                     end else begin
349                        PCSrcD    = 0;
350                        MemtoRegD = 0;
351                        MemWriteD = 0;
352                        ALUSrcD   = 0;
353                        RegWriteD = 0;
354                        RegSrcD   = 'b00;
355                        ImmSrcD   = 'b00;    // doesn't matter
356                        ALUControlD = 'b00;
```

```
357                     FlagWriteD = 00;
358                     BranchD = 0;
359                   end
360               end
361
362   //            // CMP
363   //            8'b00?00101 : begin
364   //                PCS    = 0;
365   //                MemtoRegD = 0;
366   //                MemW = 0;
367   //                ALUSrcD   = Instr[25];
368   //                RegW = 1;
369   //                RegSrcD   = 'b00;
370   //                ImmSrcD   = 'b00;
371   //                ALUControlD = 'b01;  // subtract
372   //                FlagWriteE = 11;
373   //              end
374
375             default: begin
376                     PCSrcD    = 0;
377                     MemtoRegD = 0; // doesn't matter
378                     MemWriteD = 0;
379                     ALUSrcD   = 0;
380                     RegWriteD = 0;
381                     RegSrcD   = 'b00;
382                     ImmSrcD   = 'b00;
383                     ALUControlD = 'b00;  // do an add
384                     FlagWriteD = 00;
385                     BranchD = 0;
386             end
387           endcase
388       end
389
390
391   endmodule
```

```
1   //Aaron Hong (ahong02)
2   //Stephen Macris (smacris)
3   //5/3/23
4   //EE469 Lab3
5
6   //This module creates an asynchronous conditional unit that manages the flags register
7   //and determines conditional execution.
8
9   module cond_unit (cond, flags, ALUFlags, flag_write, flags_out, cond_ex);
10
11     input  logic [3:0] cond, flags, ALUFlags;
12     input  logic [1:0] flag_write;
13     output logic cond_ex;
14     output logic [3:0] flags_out;
15
16     always_comb begin
17        case (cond)
18           4'b0000: cond_ex = flags[2];
19           4'b0001: cond_ex = !flags[2];
20           4'b0010: cond_ex = flags[1];
21           4'b0011: cond_ex = !flags[1];
22           4'b0100: cond_ex = flags[3];
23           4'b0101: cond_ex = !flags[3];
24           4'b0110: cond_ex = flags[0];
25           4'b0111: cond_ex = !flags[0];
26           4'b1000: cond_ex = !flags[2] && flags[1];
27           4'b1001: cond_ex = flags[2] || !flags[1];
28           4'b1010: cond_ex = !(flags[3] ^ flags[0]);
29           4'b1011: cond_ex = flags[3] ^ flags[0];
30           4'b1100: cond_ex = !flags[2] && !(flags[3] ^ flags[0]);
31           4'b1101: cond_ex = flags[2] || (flags[3] ^ flags[0]);
32           4'b1110: cond_ex = 1;
33           default: cond_ex = 0;
34        endcase
35
36        case (flag_write)
37           2'b11: flags_out = ALUFlags;
38           2'b10: flags_out = {ALUFlags[3:2], 2'b00};
39           2'b01: flags_out = {2'b00, ALUFlags[1:0]};
40           2'b00: flags_out = 4'b0000;
41           default: flags_out = 4'b0000;
42        endcase
43     end
44
45  endmodule
46
47
```

```
1   //Aaron Hong (ahong02)
2   //Stephen Macris (smacris)
3   //5/3/23
4   //EE469 Lab3
5
6   //This module creates an asynchronous hazard unit that outputs stalling, flushing, and
    forwarding control signals.
7
8   module hazard_unit (
9      input logic Match_1E_M, Match_2E_M,
10     input logic Match_1E_W, Match_2E_W,
11     input logic Match_12D_E,
12     input logic RegWriteM, RegWriteW, MemtoRegE, BranchTakenE, PCWrPendingF, PCSrcW,
13     output logic [1:0] ForwardAE, ForwardBE,
14     output logic StallF, StallD, FlushD, FlushE
15     );
16
17     logic ldrStallD;
18
19     assign ldrStallD = Match_12D_E & MemtoRegE;
20
21     assign StallF = ldrStallD + PCWrPendingF;
22     assign FlushD = PCWrPendingF + PCSrcW + BranchTakenE;
23     assign FlushE = ldrStallD + BranchTakenE;
24     assign StallD = ldrStallD;
25
26     always_comb begin
27        if      (Match_1E_M && RegWriteM) ForwardAE = 2'b10;
28        else if (Match_1E_W && RegWriteW) ForwardAE = 2'b01;
29        else                              ForwardAE = 2'b00;
30
31        if      (Match_2E_M && RegWriteM) ForwardBE = 2'b10;
32        else if (Match_2E_W && RegWriteW) ForwardBE = 2'b01;
33        else                              ForwardBE = 2'b00;
34     end
35
36  endmodule
37
38
```