



Bilkent University
CS 224 - Section 1 - Preliminary Design Report - Lab 5

Mehmet Akif Şahin - 22203673

15 April 2024

Part B - Potential Hazards

1. Compute use - data hazard : can be solved using forwarding
2. Load use - data hazard : can be solved using forwarding and stalling depending on case
3. Branch - control hazard : can be solved using early branch detection with branch prediction and stalling depending on case

Part C - Hazard Unit Signals

```

StallF = StallD = FlushE = lwstall | branchstall
lwstall = MemToRegE & ( rtE == rsD | rtE == rtD )
branchstall = branchD & RegWriteE & ( WriteRegE == rsD | WriteRegE == rtD )
|
branchD & MemToRegM & (WriteRegM == rsD | WriteRegM == rtD)
ForwardAD = RegWriteM & ( rsD != 0 & rsD == WriteRegM )
ForwardBD = RegWriteM & ( rtD != 0 & rtD == WriteRegM )

IF rsE != 0 & rsE == WriteRegM & RegWriteM
THEN ForwardAE = 10
ELSE IF rsE != 0 & rsE == WriteRegW & RegWriteW
THEN ForwardAE = 01
ELSE
THEN ForwardAE = 00

IF rtE != 0 & rtE == WriteRegM & RegWriteM
THEN ForwardBE = 10
ELSE IF rtE != 0 & rtE == WriteRegW & RegWriteW
THEN ForwardBE = 01
ELSE
THEN ForwardBE = 00

```

Part D - Code

```

1  module PipeFtoD(input logic[31:0] instr, PcPlus4F,
2      input logic EN, clk, clear, reset,
3      output logic[31:0] instrD, PcPlus4D);
4
5      always_ff @(posedge clk or posedge reset)
6          if (reset | clear)
7              begin
8                  instrD <= 0;
9                  PcPlus4D <= 0;
10                 end
11             else
12                 if(EN)
13                     begin
14                         instrD<=instr;
15                         PcPlus4D<=PcPlus4F;
16                     end
17
18 endmodule
19
20 module PipeWtoF(input logic[31:0] PC,
21     input logic EN, clk, reset,      // StallF will be connected as this
22     output logic[31:0] PCF);
23
24     always_ff @(posedge clk, posedge reset) begin

```

```

25         if (reset) begin
26             PCF <= 0;
27         end else if (EN) begin
28             PCF <= PC;
29         end
30     end
31 endmodule
32
33 module PipeDtoE(input logic clk, clear, reset, // connect clear to FlushE
34                 input logic RegWriteD, MemtoRegD, MemWriteD,
35                 input logic [2:0] ALUControlD,
36                 input logic ALUSrcD, RegDstD,
37                 input logic [31:0] Read1D, Read2D,
38                 input logic [4:0] RsD, RtD, RdD,
39                 input logic [31:0] SignImmD,
40                 output logic RegWriteE, MemtoRegE, MemWriteE,
41                 output logic [2:0] ALUControlE,
42                 output logic ALUSrcE, RegDstE,
43                 output logic [31:0] Read1E, Read2E,
44                 output logic [4:0] RsE, RtE, RdE,
45                 output logic [31:0] SignImmE
46             );
47
48     always_ff @(posedge clk or posedge reset)
49     begin
50         if ( reset )
51         begin
52             RegWriteE <= 0;
53             MemtoRegE <= 0;
54             MemWriteE <= 0;
55             ALUControlE <= 0;
56             ALUSrcE <= 0;
57             RegDstE <= 0;
58             Read1E <= 0;
59             Read2E <= 0;
60             RsE <= 0;
61             RtE <= 0;
62             RdE <= 0;
63             SignImmE <= 0;
64         end
65         else
66         begin
67             if ( clear )
68             begin
69                 RegWriteE <= 0;
70                 MemtoRegE <= 0;
71                 MemWriteE <= 0;
72                 ALUControlE <= 0;
73                 ALUSrcE <= 0;
74                 RegDstE <= 0;
75                 Read1E <= 0;
76                 Read2E <= 0;
77                 RsE <= 0;
78                 RtE <= 0;
79                 RdE <= 0;
80                 SignImmE <= 0;
81             end
82             else
83             begin
84                 RegWriteE <= RegWriteD;
85                 MemtoRegE <= MemtoRegD;
86                 MemWriteE <= MemWriteD;

```

```

87         ALUControlE <= ALUControlD;
88         ALUSrcE <= ALUSrcD;
89         RegDstE <= RegDstD;
90         Read1E <= Read1D;
91         Read2E <= Read2D;
92         RsE <= RsD;
93         RtE <= RtD;
94         RdE <= RdD;
95         SignImmE <= SignImmD;
96     end
97 end
98
99
100 endmodule
101
102 module PipeEtoM(input logic clk, reset,
103     input logic RegWriteE, MemtoRegE, MemWriteE,
104     input logic [31:0] ALUOutE, WriteDataE,
105     input logic [4:0] WriteRegE,
106     output logic RegWriteM, MemtoRegM, MemWriteM,
107     output logic [31:0] ALUOutM, WriteDataM,
108     output logic [4:0] WriteRegM
109 );
110
111     always_ff @(posedge clk or posedge reset)
112     begin
113         if ( reset )
114         begin
115             RegWriteM <= 0;
116             MemtoRegM <= 0;
117             MemWriteM <= 0;
118             ALUOutM <= 0;
119             WriteDataM <= 0;
120             WriteRegM <= 0;
121         end
122         else
123         begin
124             RegWriteM <= RegWriteE;
125             MemtoRegM <= MemtoRegE;
126             MemWriteM <= MemWriteE;
127             ALUOutM <= ALUOutE;
128             WriteDataM <= WriteDataE;
129             WriteRegM <= WriteRegE;
130         end
131     end
132 endmodule
133
134 module PipeMtoW(input logic clk, reset,
135     input logic RegWriteM, MemtoRegM,
136     input logic [31:0] ReadDataM, ALUOutM,
137     input logic [4:0] WriteRegM,
138     output logic RegWriteW, MemtoRegW,
139     output logic [31:0] ReadDataW, ALUOutW,
140     output logic [4:0] WriteRegW
141 );
142
143     always_ff @(posedge clk or posedge reset)
144     begin
145         if (reset)
146         begin
147             RegWriteW <= 0;
148             MemtoRegW <= 0;

```

```

149         ReadDataW <= 0;
150         ALUOutW <= 0;
151         WriteRegW <= 0;
152     end
153     else
154     begin
155         RegWriteW <= RegWriteM;
156         MemtoRegW <= MemtoRegM;
157         ReadDataW <= ReadDataM;
158         ALUOutW <= ALUOutM;
159         WriteRegW <= WriteRegM;
160     end
161 end
162
163 endmodule
164
165 module datapath (input  logic clk, reset,
166                 input  logic RegWriteD, MemtoRegD, MemWriteD,
167                 input  logic [2:0] ALUControlD,
168                 input  logic ALUSrcD, RegDstD, BranchD, jump,
169                 input  logic stallF, stallD, ForwardAD, ForwardBD, FlushE,
170                 input  logic [1:0] ForwardAE, ForwardBE,
171
172                 output logic [4:0] RsD, RtD, RsE, RtE,
173                 output logic [4:0] WriteRegE, WriteRegM, WriteRegW,
174                 output logic [5:0] opcode, func,
175                 output logic RegWriteW, RegWriteM, RegWriteE, MemtoRegE, MemtoRegM,
176
177                 output logic MemWriteE,
178                 output logic [31:0] ALUOutE, WriteDataE, pc, PC_prime,
179                 output logic [4:0] writereg
180             );
181
182     logic EqualD, MemWriteM, ftodclear;
183     logic PcSrcD, MemtoRegW;
184     logic [31:0] PC, PCF, instrF, instrD, PcSrcA, PcSrcB, PcPlus4F, PcPlus4D,
185                 EqualD1, EqualD2;
186     logic [31:0] PcBranchD, ALUOutW, ReadDataW, ResultW, RD1, RD2;
187     logic [4:0] RdD;
188
189     logic [31:0] PCbranch, SignImmD, SignImmShifted, SrcAE, SrcBE, SrcBEwImm,
190                 ALUOutM, WriteDataM, ReadDataM;
191
192     logic [2:0] ALUControlE;
193     logic ALUSrcE, RegDstE;
194     logic [31:0] Read1E, Read2E;
195     logic [4:0] RdE;
196     logic [31:0] SignImmE;
197
198     mux2 #(32) result_mux(ALUOutW, ReadDataW, MemtoRegW, ResultW);
199
200     PipeWtoF pWtoF(PC, ~stallF, clk, reset, PCF);
201     Writeback stage pipe
202
203     assign pc = PCF;
204     assign PC_prime = PC;
205
206     assign PcPlus4F = PCF + 4;
207     from fetch stage
208     mux2 #(32) pc_mux(PcPlus4F, PcBranchD, PcSrcD, PCbranch);
209     PcBranchD is from decode stage

```

```

205     mux2 #(32) jump_mux(PCbranch, { PcPlus4D[31:28], instrD[25:0], 2'b00}, jump, PC
206     );
207     // Note that normally whole PCF should be driven to
208     // instruction memory. However for our instruction
209     // memory this is not necessary
210     imem im1(PCF[7:2], instrF); // Instantiated
211     instruction memory
212
213     assign ftodclear = PcSrcD | jump;
214
215     PipeFtoD pFtoD(instrF, PcPlus4F, ~stallD, clk, ftodclear, reset, instrD,
216     PcPlus4D); // Fetch stage pipe
217
218     regfile rf(clk, RegWriteW, instrD[25:21], instrD[20:16],
219     WriteRegW, ResultW, RD1, RD2);
220     // Add the rest.
221
222     signext immsignext (instrD[15:0], SignImmD);
223     sl2 shiftimm (SignImmD, SignImmShifted);
224     adder branchadder (SignImmShifted, PcPlus4D, PcBranchD);
225
226     mux2 #(32) RD1mux (RD1, ALUOutM, ForwardAD, EqualD1);
227     mux2 #(32) RD2mux (RD2, ALUOutM, ForwardBD, EqualD2);
228     assign EqualD = EqualD1 == EqualD2;
229     assign PcSrcD = BranchD && EqualD;
230
231     assign opcode = instrD[31:26];
232     assign func = instrD[5:0];
233
234     assign RsD = instrD[25:21];
235     assign RtD = instrD[20:16];
236     assign RdD = instrD[15:11];
237
238     PipeDtoE pipedtoe (clk, FlushE, reset,
239     RegWriteD, MemtoRegD, MemWriteD,
240     ALUControlD,
241     ALUSrcD, RegDstD,
242     RD1, RD2,
243     RsD, RtD, RdD,
244     SignImmD,
245     RegWriteE, MemtoRegE, MemWriteE,
246     ALUControlE,
247     ALUSrcE, RegDstE,
248     Read1E, Read2E,
249     RsE, RtE, RdE,
250     SignImmE
251     );
252
253     assign writereg = RtE;
254
255     mux2 #(5) writeregEmux (RtE, RdE, RegDstE, WriteRegE);
256
257     mux4 #(32) SrcAEmux (Read1E, ResultW, ALUOutM, 0, ForwardAE, SrcAE);
258     mux4 #(32) SrcBEmux (Read2E, ResultW, ALUOutM, 0, ForwardBE, SrcBE);
259
260     mux2 #(32) immmux (SrcBE, SignImmE, ALUSrcE, SrcBEWImm);
261
262     alu alu (SrcAE, SrcBEWImm,
263     ALUControlE,
264     ALUOutE);
265
266     assign WriteDataE = SrcBE;

```

```

263 PipeEtoM pipeetom (clk, reset,
264                     RegWriteE, MemtoRegE, MemWriteE,
265                     ALUOutE, WriteDataE,
266                     WriteRegE,
267                     RegWriteM, MemtoRegM, MemWriteM,
268                     ALUOutM, WriteDataM,
269                     WriteRegM
270                     );
271
272
273 dmem dmem (clk, MemWriteM,
274            ALUOutM, WriteDataM,
275            ReadDataM);
276
277 PipeMtoW pipemtow (clk, reset,
278                   RegWriteM, MemtoRegM,
279                   ReadDataM, ALUOutM,
280                   WriteRegM,
281                   RegWriteW, MemtoRegW,
282                   ReadDataW, ALUOutW,
283                   WriteRegW
284                   );
285
286 endmodule
287
288 // parameterized 2-to-1 MUX
289 module mux4 #(parameter WIDTH = 8)
290             (input  logic[WIDTH-1:0] d0, d1, d2, d3,
291              input  logic[1:0] s,
292              output logic[WIDTH-1:0] y);
293
294     assign y = s[1] ? (s[0] ? d3 : d2) : (s[0] ? d1 : d0);
295 endmodule
296
297
298 module HazardUnit(
299     input logic branchD,
300     input logic [4:0] WriteRegW, WriteRegM, WriteRegE,
301     input logic RegWriteW, RegWriteM, RegWriteE, MemtoRegE, MemtoRegM,
302     input logic [4:0] rsE,rtE,
303     input logic [4:0] rsD,rtD,
304     output logic ForwardAD,ForwardBD,
305     output logic [2:0] ForwardAE,ForwardBE,
306     output logic FlushE,StallD,StallF, lwstall, branchstall
307
308 );
309
310 // logic lwstall, branchstall;
311
312 always_comb begin
313     lwstall = MemtoRegE & ( rtE == rsD | rtE == rtD );
314     branchstall = (branchD & RegWriteE & ( WriteRegE == rsD | WriteRegE == rtD
315                                     |
316                                     (branchD & MemtoRegM & ( WriteRegM == rsD | WriteRegM == rtD
317                                     )));
318     StallF = lwstall | branchstall;
319     StallD = lwstall | branchstall;
320     FlushE = lwstall | branchstall;
321     ForwardAD = RegWriteM & ( rsD != 0 & rsD == WriteRegM );
322     ForwardBD = RegWriteM & ( rtD != 0 & rtD == WriteRegM );

```

```

323         if ( rsE != 0 & rsE == WriteRegM & RegWriteM ) begin
324             ForwardAE = 2'b10;
325         end
326         else if ( rsE != 0 & rsE == WriteRegW & RegWriteW ) begin
327             ForwardAE = 2'b01;
328         end
329         else begin
330             ForwardAE = 2'b00;
331         end
332
333         if ( rtE != 0 & rtE == WriteRegM & RegWriteM ) begin
334             ForwardBE = 2'b10;
335         end
336         else if ( rtE != 0 & rtE == WriteRegW & RegWriteW ) begin
337             ForwardBE = 2'b01;
338         end
339         else begin
340             ForwardBE = 2'b00;
341         end
342     end
343 endmodule
344
345
346 module mips (input  logic      clk, reset,
347              output logic [31:0] writedata, dataaddr,
348              output logic      memwrite, regwrite,
349              output logic [31:0] pc, PC_prime,
350              output logic lwstall, branchstall, branch,
351              output logic [4:0] writereg, rsD, rtD, regdst
352
353              );
354
355     logic      memtoreg, pcsrc, zero, alusrc, regWriteD, jump;
356     logic [2:0] alucontrol;
357     logic [5:0] op, funct;
358
359     logic stallF, stallD, ForwardAD, ForwardBD, FlushE, RegWriteW, RegWriteM,
360           MemtoRegE, MemtoRegM, MemWriteD;
361
362     logic [1:0] ForwardAE, ForwardBE;
363
364     logic [4:0] rsE, rtE, WriteRegE, WriteRegM, WriteRegW;
365
366     datapath dp (clk, reset,
367                 regWriteD, memtoreg, MemWriteD,
368                 alucontrol,
369                 alusrc, regdst, branch, jump,
370                 stallF, stallD, ForwardAD, ForwardBD, FlushE,
371                 ForwardAE, ForwardBE,
372
373                 rsD, rtD, rsE, rtE,
374                 WriteRegE, WriteRegM, WriteRegW,
375                 op, funct,
376                 RegWriteW, RegWriteM, regwrite, MemtoRegE, MemtoRegM,
377
378                 memwrite,
379                 dataaddr, writedata,
380                 pc, PC_prime, writereg
381             );
382
383     controller cont (op, funct,
384                     memtoreg, MemWriteD,

```



```

384         alusrc,
385         regdst, regWriteD,
386         jump,
387         alucontrol,
388         branch);
389
390 HazardUnit hu (
391     branch,
392     WriteRegW, WriteRegM, WriteRegE,
393     RegWriteW, RegWriteM, regwrite, MemtoRegE, MemtoRegM,
394     rsE,rtE,
395     rsD,rtD,
396     ForwardAD,ForwardBD,
397     ForwardAE,ForwardBE,
398     FlushE,stallD,stallF,
399
400     lwstall, branchstall
401 );
402
403 endmodule
404
405
406 // External instruction memory used by MIPS single-cycle
407 // processor. It models instruction memory as a stored-program
408 // ROM, with address as input, and instruction as output
409 // Modify it to test your own programs.
410
411 module imem ( input logic [5:0] addr, output logic [31:0] instr);
412
413 // imem is modeled as a lookup table, a stored-program byte-addressable ROM
414     always_comb
415         case ({addr,2'b00}) // word-aligned fetch
416 //
417 // *****
418 // Here, you can paste your own test cases that you prepared for the part 1-g.
419 // Below is a program from the single-cycle lab.
420 // *****
421 //
422 //         address      instruction
423 //         -----      -
424             8'h00: instr = 32'h20080001;
425             8'h04: instr = 32'h20090002;
426             8'h08: instr = 32'h0109502a;
427             8'h0c: instr = 32'h1140ffff;
428             8'h10: instr = 32'h0128502a;
429             8'h14: instr = 32'h11400001;
430             8'h18: instr = 32'h200a0008;
431             8'h1c: instr = 32'h08000000;
432             default: instr = {32{1'bx}}; // unknown address
433         endcase
434 endmodule
435
436
437 // *****
438 // Below are the modules that you shouldn't need to modify at all..
439 // *****
440
441 module controller(input logic[5:0] op, funct,
442                 output logic memtoreg, memwrite,
443                 output logic alusrc,
444                 output logic regdst, regwrite,
445                 output logic jump,

```

```

446         output logic[2:0] alucontrol,
447         output logic branch);
448
449     logic [1:0] aluop;
450
451     maindec md (op, memtoreg, memwrite, branch, alusrc, regdst, regwrite,
452               jump, aluop);
453
454     aludec ad (funct, aluop, alucontrol);
455
456 endmodule
457
458 // External data memory used by MIPS single-cycle processor
459
460 module dmem (input logic clk, we,
461             input logic[31:0] a, wd,
462             output logic[31:0] rd);
463
464     logic [31:0] RAM[63:0];
465
466     assign rd = RAM[a[31:2]]; // word-aligned read (for lw)
467
468     always_ff @(posedge clk)
469         if (we)
470             RAM[a[31:2]] <= wd; // word-aligned write (for sw)
471
472 endmodule
473
474 module maindec (input logic[5:0] op,
475               output logic memtoreg, memwrite, branch,
476               output logic alusrc, regdst, regwrite, jump,
477               output logic[1:0] aluop );
478     logic [8:0] controls;
479
480     assign {regwrite, regdst, alusrc, branch, memwrite,
481           memtoreg, aluop, jump} = controls;
482
483     always_comb
484         case(op)
485             6'b000000: controls <= 9'b110000100; // R-type
486             6'b100011: controls <= 9'b101001000; // LW
487             6'b101011: controls <= 9'b001010000; // SW
488             6'b000100: controls <= 9'b000100010; // BEQ
489             6'b001000: controls <= 9'b101000000; // ADDI
490             6'b000010: controls <= 9'b000000001; // J
491             default: controls <= 9'bxxxxxxx; // illegal op
492         endcase
493 endmodule
494
495 module aludec (input logic[5:0] funct,
496             input logic[1:0] aluop,
497             output logic[2:0] alucontrol);
498     always_comb
499         case(aluop)
500             2'b00: alucontrol = 3'b010; // add (for lw/sw/addi)
501             2'b01: alucontrol = 3'b110; // sub (for beq)
502             default: case(funct) // R-TYPE instructions
503                 6'b100000: alucontrol = 3'b010; // ADD
504                 6'b100010: alucontrol = 3'b110; // SUB
505                 6'b100100: alucontrol = 3'b000; // AND
506                 6'b100101: alucontrol = 3'b001; // OR
507                 6'b101010: alucontrol = 3'b111; // SLT

```

```

508         default:    alucontrol  = 3'bxxx; // ???
509     endcase
510 endcase
511 endmodule
512
513 module regfile (input    logic clk, we3,
514                 input    logic[4:0]  ra1, ra2, wa3,
515                 input    logic[31:0] wd3,
516                 output   logic[31:0] rd1, rd2);
517
518     logic [31:0] rf [31:0];
519
520     // three ported register file: read two ports combinationaly
521     // write third port on rising edge of clock. Register0 hardwired to 0.
522
523     always_ff @(negedge clk)
524         if (we3)
525             rf [wa3] <= wd3;
526
527     assign rd1 = (ra1 != 0) ? rf [ra1] : 0;
528     assign rd2 = (ra2 != 0) ? rf[ ra2] : 0;
529
530 endmodule
531
532 module alu(input  logic [31:0] a, b,
533            input  logic [2:0]  alucont,
534            output logic [31:0] result,
535            output logic zero);
536
537     always_comb
538         case(alucont)
539             3'b010: result = a + b;
540             3'b110: result = a - b;
541             3'b000: result = a & b;
542             3'b001: result = a | b;
543             3'b111: result = (a < b) ? 1 : 0;
544             default: result = {32{1'bx}};
545         endcase
546
547     assign zero = (result == 0) ? 1'b1 : 1'b0;
548
549 endmodule
550
551 module adder (input  logic[31:0] a, b,
552               output logic[31:0] y);
553
554     assign y = a + b;
555 endmodule
556
557 module sl2 (input  logic[31:0] a,
558             output logic[31:0] y);
559
560     assign y = {a[29:0], 2'b00}; // shifts left by 2
561 endmodule
562
563 module signext (input  logic[15:0] a,
564                 output logic[31:0] y);
565
566     assign y = {{16{a[15]}}, a}; // sign-extends 16-bit a
567 endmodule
568
569 // parameterized register

```

```

570 module flopr #(parameter WIDTH = 8)
571     (input logic clk, reset,
572      input logic[WIDTH-1:0] d,
573      output logic[WIDTH-1:0] q);
574
575     always_ff@(posedge clk, posedge reset)
576         if (reset) q <= 0;
577         else      q <= d;
578 endmodule
579
580
581 // paramaterized 2-to-1 MUX
582 module mux2 #(parameter WIDTH = 8)
583     (input  logic[WIDTH-1:0] d0, d1,
584      input  logic s,
585      output logic[WIDTH-1:0] y);
586
587     assign y = s ? d1 : d0;
588 endmodule

```

my_cpu.sv

Part E - Hazard Tests

```

1 // no hazard
2 addi $s0, $zero, 1
3 addi $s1, $zero, 2
4 addi $s2, $zero, 3
5 addi $s3, $zero, 4
6 sw $s0, 0($zero)
7 and $t0, $s0, $s1
8 add $t1, $s1, $s2
9 slt $t2, $s2, $s3
10 sub $t3, $s3, $s0
11 lw $t4, 0($zero)
12
13 // compute use hazard
14 addi $t0, $zero, 9
15 sw $t0, 0($zero)
16 or $t1, $0, $t0
17 and $t2, $0, $t0
18 lw $t3, 0($t2)
19 addi $t4, $zero, $t3
20
21 // load use
22 addi $t0, $zero, 7
23 addi $t1, $zero, 9
24 lw $t0, 0($zero)
25 lw $t1, 4($zero)
26 and $t2, $t1, $t0
27
28 // branch hazard
29 addi $t0, $zero, 1
30 addi $t1, $zero, 2
31 slt $t2, $t0, $t1
32 beq $t2, $zero, -1
33 slt $t2, $t1, $t0
34 beq $t2, $zero, 1
35 addi $t2, $zero, 8
36 j 0

```

test_codes.asm