

```

1  // RISCVMulti.sv
2  // risc-v multicycle processor
3  // Christian Johnson
4  // chrjohnson@hmc.edu
5  // 12/3/2024
6
7  // top module
8  module top(input logic      clk, reset,
9             output logic [31:0] writedata, dataadr,
10             output logic      memwrite);
11
12     // Declaring internal logic for the top module
13     logic [31:0] readdata;
14
15     // instantiate processor and memory
16     RISCVMulti    RISCVMulti(clk, reset, memwrite, writedata, dataadr, readdata);
17     unimem        unimem(clk, memwrite, dataadr, writedata, readdata);
18
19 endmodule
20
21 // unified memory module
22 // combines both the dmem and imem module from the RISCVsingl processor
23 module unimem(input logic      clk, WE,
24               input logic [31:0] A, WD,
25               output logic [31:0] RD);
26
27     // Declaring and initalizing RAM
28     logic [31:0] RAM[63:0];
29
30     // Instruction memory logic
31     initial
32         $readmemh("memfile.dat", RAM);
33
34     always_ff @(posedge clk)
35         if (WE) RAM[A[31:2]] <= WD;
36
37     assign RD = RAM[A[31:2]]; // word aligned
38 endmodule
39
40 // RISCVMulti module containing calls to the controller and datapath modules
41 module RISCVMulti(input      clk, reset,
42                   output      memwrite,
43                   output [31:0] writedata, dataadr,
44                   input  [31:0] readdata);
45
46     // Internal logic for RISCVMulti
47     logic      zero, adrsrc, irwrite, pcwrite, regwrite;
48     logic [1:0] resultsrc, alusrcb, alusrca, immsrc;
49     logic [2:0] alucontrol;
50     logic [31:0] instr;
51
52     // instantiate the controller and datapath
53     multicycle_controller c(clk, reset, instr[6:0],
54                             instr[14:12], instr[30], zero,
55                             immsrc, alusrca, alusrcb, resultsrc,
56                             adrsrc, alucontrol, irwrite,
57                             pcwrite, regwrite, memwrite);
58
59     datapath dp(clk, reset, regwrite,
60                 adrsrc, pcwrite, irwrite,
61                 resultsrc, immsrc, alucontrol,
62                 alusrca, alusrcb,
63                 readdata, instr, zero, dataadr, writedata);
64 endmodule
65
66 // Datapath module
67 module datapath(input logic      clk, reset,
68                input logic      regwrite,
69                input logic      adrsrc,
70                input logic      pcwrite,

```

```

71         input logic          irwrite,
72         input logic [1:0]    resultsrc, immsrc,
73         input logic [2:0]    alucontrol,
74         input logic [1:0]    alusrca, alusrcb,
75         input logic [31:0]   readdata,
76         output logic [31:0]  instr,
77         output logic          zero,
78         output logic [31:0]  adr, writedata);
79
80 // Defining all internal logic for datapath
81 logic [31:0]    pcnext, pc, oldpc;
82 logic [31:0]    data;
83 logic [31:0]    a, srca, srcb;
84 logic [31:0]    immext;
85 logic [31:0]    alurestult, aluout;
86 logic [31:0]    result;
87 logic [31:0]    regfileout1, regfileout2;
88
89 // pc logic
90 flopenr #(32) pcreg(clk, reset, pcwrite, pcnext, pc);
91
92 // adr logic
93 mux2 #(32)    adrmux(pc, result, adrsrc, adr);
94
95 // oldpc/instr logic
96 flopenr #(32) oldpcreg(clk, reset, irwrite, pc, oldpc);
97 flopenr #(32) instrreg(clk, reset, irwrite, readdata, instr);
98
99 // data logic
100 flopr #(32)    datareg(clk, reset, readdata, data);
101
102 // register file logic
103 RegFile        rf(clk, regwrite, instr[19:15], instr[24:20],
104                  instr[11:7], result, regfileout1, regfileout2);
105
106 // extend logic
107 Extend         ext(instr[31:7], immsrc, immext);
108
109 // a/writedata logic
110 flopr #(32)    areg(clk, reset, regfileout1, a);
111 flopr #(32)    writedatareg(clk, reset, regfileout2, writedata);
112
113 // srca logic
114 mux3 #(32)    srcamux(pc, oldpc, a, alusrca, srca);
115
116 // srcb logic
117 mux3 #(32)    srcbmux(writedata, immext, 32'd4, alusrcb, srcb);
118
119 // alu logic
120 alu           alu(srca, srcb, alucontrol, alurestult, zero);
121
122 // aluout logic
123 flopr #(32)    aluoutreg(clk, reset, alurestult, aluout);
124
125 // result logic
126 mux3 #(32)    resultmux(aluout, data, alurestult, resultsrc, result);
127 assign        pcnext = result;
128 endmodule
129
130 // Structural Verilog Code for the ALU controller (Adapted from lab10_CJ)
131 module multicycle_controller(input logic clk,
132                             input logic reset,
133                             input logic [6:0] op,
134                             input logic [2:0] funct3,
135                             input logic funct7b5,
136                             input logic zero,
137                             output logic [1:0] immsrc,
138                             output logic [1:0] alusrca, alusrcb,
139                             output logic [1:0] resultsrc,
140                             output logic adrsrc,

```

```

141         output logic [2:0] alucontrol,
142         output logic irwrite, pcwrite,
143         output logic regwrite, memwrite);
144
145     // Defining internal logic for the multicycle controller
146     logic pcupdate, branch;
147     logic [1:0] aluop;
148
149     // Calling mainFSM module
150     mainFSM fsm(clk, reset, op, pcupdate, branch, regwrite, memwrite, irwrite, resultsrc,
    alusrca, alusrcb, adrsrc, aluop);
151
152
153     /*
154     Calling alu_decoder (from lab 2 implementation)
155     aluop[1]: a
156     aluop[0]: b
157
158     funct3[2]: c
159     funct3[1]: d
160     funct3[0]: e
161
162     op[5]: f
163     funct7b5: g
164     */
165     alu_decoder aludec(aluop[1], aluop[0], funct3[2], funct3[1], funct3[0], op[5],
    funct7b5, alucontrol[2], alucontrol[1], alucontrol[0]);
166
167     // Calling instr_decoder module
168     instr_decoder id(op, immsrc);
169
170
171     // output logic
172     assign pcwrite = branch & zero | pcupdate;
173 endmodule
174
175 module mainFSM(input logic clk,
176               input logic reset,
177               input logic [6:0] op,
178               output logic pcupdate,
179               output logic branch,
180               output logic regwrite, memwrite,
181               output logic irwrite,
182               output logic [1:0] resultsrc,
183               output logic [1:0] alusrca, alusrcb,
184               output logic adrsrc,
185               output logic [1:0] aluop);
186
187     // Internal logic for states
188     typedef enum logic [3:0] {FETCH, DECODE, MEMADR, MEMREAD, MEMWB, MEMWRITE, EXECUTER,
    ALUWB, EXECUTEI, JAL, BEQ} statetype;
189     statetype state, nextstate;
190     logic [13:0] controls;
191
192     // state register
193     always_ff @(posedge clk, posedge reset)
194         if (reset) state <= FETCH;
195         else state <= nextstate;
196
197
198     // next state logic
199     always_comb
200         casez (state)
201             FETCH:      nextstate = DECODE;
202
203             DECODE:     casez (op)
204                 7'b0?00011:      nextstate = MEMADR;
205                 7'b0110011:      nextstate = EXECUTER;
206                 7'b0010011:      nextstate = EXECUTEI;
207                 7'b1101111:      nextstate = JAL;

```

```

208             7'b1100011:      nextstate = BEQ;
209             default:        nextstate = state;
210         endcase
211
212     MEMADR:      casez(op)
213                 7'b0000011:      nextstate = MEMREAD;
214                 7'b0100011:      nextstate = MEMWRITE;
215                 default:        nextstate = state;
216             endcase
217
218     MEMREAD:     nextstate = MEMWB;
219
220     MEMWB:       nextstate = FETCH;
221
222     MEMWRITE:    nextstate = FETCH;
223
224     EXECUTER:    nextstate = ALUWB;
225
226     ALUWB:       nextstate = FETCH;
227
228     EXECUTEI:    nextstate = ALUWB;
229
230     JAL:         nextstate = ALUWB;
231
232     BEQ:         nextstate = FETCH;
233
234     default:     nextstate = state;
235 endcase
236
237
238
239 // setting current state signals
240 always_comb
241     case (state)
242         //
243 pcupdate__branch__regwrite__memwrite__irwrite__resultsrc__alusrcb__alusrca__adrsrc__aluop
244     FETCH:       controls = 14'b1_0_0_0_1_10_10_00_0_00;
245
246     DECODE:      controls = 14'b0_0_0_0_0_00_01_01_0_00;
247
248     MEMADR:      controls = 14'b0_0_0_0_0_00_01_10_0_00;
249
250     MEMREAD:     controls = 14'b0_0_0_0_0_00_00_00_1_00;
251
252     MEMWB:       controls = 14'b0_0_1_0_0_01_00_00_0_00;
253
254     MEMWRITE:    controls = 14'b0_0_0_1_0_00_00_00_1_00;
255
256     EXECUTER:    controls = 14'b0_0_0_0_0_00_00_10_0_10;
257
258     ALUWB:       controls = 14'b0_0_1_0_0_00_00_00_0_00;
259
260     EXECUTEI:    controls = 14'b0_0_0_0_0_00_01_10_0_10;
261
262     JAL:         controls = 14'b1_0_0_0_0_00_10_01_0_00;
263
264     BEQ:         controls = 14'b0_1_0_0_0_00_00_10_0_01;
265
266     default:     controls = 14'b0_0_0_0_0_00_00_00_0_00;
267 endcase
268
269     assign {pcupdate, branch, regwrite, memwrite, irwrite, resultsrc, alusrcb, alusrca,
adrsrc, aluop} = controls;
270 endmodule
271
272 // Structural Verilog Code for the ALU Decoder adapted from lab 2
273 module alu_decoder(input logic a, b, c, d, e, f, g,
274                   output logic y2, y1, y0);
275 // Declaring the internal logic signals or local variables
276 // which can only be used inside of this module

```

```

276     logic n1, n2, n3, na, nb, nc, nd, ne;
277
278     // Getting negations of each input
279     not g1(na, a);
280     not g2(nb, b);
281     not g3(nc, c);
282     not g4(nd, d);
283     not g5(ne, e);
284
285     // y2 output logic
286     and a1(y2, a, nb, nc, d, ne);
287
288     // y1 output logic
289     and a2(y1, a, nb, c, d);
290
291     // y0 output logic
292     and a3(n1, na, b);
293     and a4(n2, a, nb, nc, nd, ne, f, g);
294     and a5(n3, a, nb, d, ne);
295     or o1(y0, n1, n2, n3);
296 endmodule
297
298 // Structural Verilog Code for the instr decoder
299 module instr_decoder(input logic [6:0] op,
300                     output logic [1:0] immsrc);
301
302     // Logic for choosing the immsrc
303     always_comb
304         casez (op)
305             7'b0110011: immsrc <= 2'b00; // R-type data processing (dont care but setting
to 0's)
306             7'b0010011: immsrc <= 2'b00; // I-type data processing
307             7'b0000011: immsrc <= 2'b00; // LW
308             7'b0100011: immsrc <= 2'b01; // SW
309             7'b1100011: immsrc <= 2'b10; // BEQ
310             7'b1101111: immsrc <= 2'b11; // JAL
311             default: immsrc <= 2'b00; // ???
312         endcase
313 endmodule
314
315 // Extend module
316 module Extend(input logic [31:7] Instr,
317              input logic [1:0] ImmSrc,
318              output logic [31:0] ImmExt);
319
320     always_comb
321         case(ImmSrc)
322             // I-type
323             2'b00: ImmExt = {{20{Instr[31]}}, Instr[31:20]};
324             // S-type (Stores)
325             2'b01: ImmExt = {{20{Instr[31]}}, Instr[31:25], Instr[11:7]};
326             // B-type (Branches)
327             2'b10: ImmExt = {{20{Instr[31]}}, Instr[7], Instr[30:25], Instr[11:8], 1'b0};
328             // J-type (Jumps)
329             2'b11: ImmExt = {{12{Instr[31]}}, Instr[19:12], Instr[20], Instr[30:21], 1'b0};
330             default: ImmExt = 32'bx; // undefined
331         endcase
332 endmodule
333
334 // Register File module
335 module RegFile(input logic clk,
336               input logic WE3,
337               input logic [4:0] A1, A2, A3,
338               input logic [31:0] WD3,
339               output logic [31:0] RD1, RD2);
340
341     logic [31:0] rf[31:0];
342
343     // three ported register file
344     // read two ports combinationally (A1/RD1, A2/RD2)

```

```

345 // write third port on rising edge of clock (A3/WD3/WE3)
346 // register 0 hardwired to 0
347
348 always_ff @(posedge clk)
349     if (WE3) rf[A3] <= WD3;
350
351 assign RD1 = (A1 != 0) ? rf[A1] : 0;
352 assign RD2 = (A2 != 0) ? rf[A2] : 0;
353 endmodule
354
355 // flopr module (resettable flip-flop)
356 module flopr #(parameter WIDTH = 8)
357     (input logic clk, reset,
358      input logic [WIDTH-1:0] d,
359      output logic [WIDTH-1:0] q);
360
361     always_ff @(posedge clk, posedge reset)
362         if (reset) q <= 0;
363         else q <= d;
364 endmodule
365
366 // flopenr module (resetable flip-flop with enable)
367 /*module flopenr #(parameter WIDTH = 8)
368     (input logic clk, reset, en,
369      input logic [WIDTH-1:0] d,
370      output logic [WIDTH-1:0] q);
371
372     always_ff @(posedge clk, posedge reset)
373         if (reset) q <= 0;
374         else if (en) q <= d;
375 endmodule
376 */
377
378 module flopenr #(parameter WIDTH = 8)
379     (input logic clk, reset,
380      input logic en,
381      input logic [WIDTH-1:0] d,
382      output logic [WIDTH-1:0] q);
383
384     always_ff @(posedge clk, posedge reset)
385         if (reset) q <= 0;
386         else if (en) q <= d;
387 endmodule
388
389 // 2x1 Mux module
390 module mux2 #(parameter WIDTH = 8)
391     (input logic [WIDTH-1:0] d0, d1,
392      input logic s,
393      output logic [WIDTH-1:0] y);
394
395     assign y = s ? d1 : d0;
396 endmodule
397
398 // 3x1 Mux module
399 module mux3 #(parameter WIDTH = 8)
400     (input logic [WIDTH-1:0] d0, d1, d2,
401      input logic [1:0] s,
402      output logic [WIDTH-1:0] y);
403
404     assign y = s[1] ? d2 : (s[0] ? d1 : d0);
405 endmodule
406
407 // ALU module
408 module alu(input logic [31:0] a, b,
409           input logic [2:0] alucontrol,
410           output logic [31:0] result,
411           output logic zero);
412
413     logic [31:0] condinvb, sum;
414     logic

```

```
415
416     assign sub = (alucontrol[1:0] == 2'b01);
417     assign condinvb = sub ? ~b : b; // for subtraction or slt
418     assign sum = a + condinvb + sub;
419
420     always_comb
421     case (alucontrol)
422         3'b000: result = sum;           // addition
423         3'b001: result = sum;           // subtraction
424         3'b010: result = a & b;         // and
425         3'b011: result = a | b;         // or
426         3'b101: result = sum[31];       // slt
427         default: result = 0;
428     endcase
429
430     assign zero = (result == 32'b0);
431 endmodule
432
```