```
1.  `timescale 1ns / 1ps
2.
3.  ///////////fields of IR
4.  `define oper_type IR[31:27]
5.  `define rdst      IR[26:22]
6.  `define rsrc1     IR[21:17]
7.  `define imm_mode  IR[16]
8.  `define rsrc2     IR[15:11]
9.  `define isrc      IR[15:0]
10.
11.
12. ////////////////arithmetic operation
13. `define movsgpr        5'b00000
14. `define mov            5'b00001
15. `define add            5'b00010
16. `define sub            5'b00011
17. `define mul            5'b00100
18.
19. ////////////////logical operations : and or xor xnor nand nor not
20.
21. `define ror            5'b00101
22. `define rand           5'b00110
23. `define rxor           5'b00111
24. `define rxnor          5'b01000
25. `define rnand          5'b01001
26. `define rnor           5'b01010
27. `define rnot           5'b01011
28.
29. ////////////////////// load & store instructions
30.
31. `define storereg       5'b01101   //////store content of register in data memory
32. `define storedin       5'b01110   ////// store content of din bus in data memory
33. `define senddout       5'b01111   /////send data from DM to dout bus
34. `define sendreg        5'b10001   ////// send data from DM to register
35.
36. ////////////////////////////// Jump and branch instructions
37. `define jump           5'b10010  ////jump to address
38. `define jcarry         5'b10011  ////jump if carry
39. `define jnocarry       5'b10100
40. `define jsign          5'b10101  ////jump if sign
41. `define jnosign        5'b10110
42. `define jzero          5'b10111  //// jump if zero
43. `define jnozero        5'b11000
44. `define joverflow      5'b11001 ////jump if overflow
45. `define jnooverflow    5'b11010
46.
47. ////////////////////////////halt
48. `define halt           5'b11011
49.
50.
51.
52. module top(
53. input clk,sys_rst,
54. input [15:0] din,
55. output reg [15:0] dout
56. );
57.
58. ////////////////adding program and data memory
59. reg [31:0] inst_mem [15:0]; ////program memory
60. reg [15:0] data_mem [15:0]; ////data memory
61.
```

```
62.
63.
64.
65.
66. reg [31:0] IR;            ////// instruction register  <--ir[31:27]--><--ir[26:22]--><--
    ir[21:17]--><--ir[16]--><--ir[15:11]--><--ir[10:0]-->
67.                          //////fields                 <---  oper  --><--   rdest --><--
    rsrc1 --><--modesel--><--  rsrc2 --><--unused  -->
68.                          //////fields                 <---  oper  --><--   rdest --><--
    rsrc1 --><--modesel--><--  immediate_date      -->
69.
70. reg [15:0] GPR [31:0] ;   ///////general purpose register gpr[0] ..... gpr[31]
71.
72.
73.
74. reg [15:0] SGPR ;      ///// msb of multiplication --> special register
75.
76. reg [31:0] mul_res;
77.
78.
79. reg sign = 0, zero = 0, overflow = 0, carry = 0; ///condition flag
80. reg [16:0] temp_sum;
81.
82. reg jmp_flag = 0;
83. reg stop = 0;
84.
85. task decode_inst();
86.  begin
87.
88. jmp_flag = 1'b0;
89. stop     = 1'b0;
90.
91. case(`oper_type)
92. ////////////////////////////////
93. `movsgpr: begin
94.
95.    GPR[`rdst] = SGPR;
96.
97. end
98.
99. ////////////////////////////////
100.       `mov : begin
101.         if(`imm_mode)
102.              GPR[`rdst]  = `isrc;
103.         else
104.             GPR[`rdst]   = GPR[`rsrc1];
105.       end
106.
107.       ////////////////////////////////////////////////////
108.
109.       `add : begin
110.            if(`imm_mode)
111.              GPR[`rdst]   = GPR[`rsrc1] + `isrc;
112.           else
113.              GPR[`rdst]   = GPR[`rsrc1] + GPR[`rsrc2];
114.       end
115.
116.       ////////////////////////////////////////////////////////
117.
118.       `sub : begin
119.            if(`imm_mode)
```

```
120.               GPR[`rdst]  = GPR[`rsrc1] - `isrc;
121.           else
122.             GPR[`rdst]   = GPR[`rsrc1] - GPR[`rsrc2];
123.       end
124.
125.       ////////////////////////////////////////////////////////
126.
127.       `mul : begin
128.            if(`imm_mode)
129.              mul_res   = GPR[`rsrc1] * `isrc;
130.           else
131.              mul_res   = GPR[`rsrc1] * GPR[`rsrc2];
132.
133.           GPR[`rdst]   =  mul_res[15:0];
134.           SGPR         =  mul_res[31:16];
135.       end
136.
137.       ////////////////////////////////////////////////////////// bitwise or
138.
139.       `ror : begin
140.            if(`imm_mode)
141.              GPR[`rdst]  = GPR[`rsrc1] | `isrc;
142.           else
143.             GPR[`rdst]   = GPR[`rsrc1] | GPR[`rsrc2];
144.       end
145.
146.       ////////////////////////////////////////////////////////////bitwise and
147.
148.       `rand : begin
149.            if(`imm_mode)
150.              GPR[`rdst]  = GPR[`rsrc1] & `isrc;
151.           else
152.             GPR[`rdst]   = GPR[`rsrc1] & GPR[`rsrc2];
153.       end
154.
155.       ////////////////////////////////////////////////////////// bitwise xor
156.
157.       `rxor : begin
158.            if(`imm_mode)
159.              GPR[`rdst]  = GPR[`rsrc1] ^ `isrc;
160.           else
161.             GPR[`rdst]   = GPR[`rsrc1] ^ GPR[`rsrc2];
162.       end
163.
164.       ////////////////////////////////////////////////////////// bitwise xnor
165.
166.       `rxnor : begin
167.            if(`imm_mode)
168.              GPR[`rdst]  = GPR[`rsrc1] ~^ `isrc;
169.           else
170.              GPR[`rdst]   = GPR[`rsrc1] ~^ GPR[`rsrc2];
171.       end
172.
173.       ////////////////////////////////////////////////////////// bitwisw nand
174.
175.       `rnand : begin
176.            if(`imm_mode)
177.              GPR[`rdst]  = ~(GPR[`rsrc1] & `isrc);
178.           else
179.             GPR[`rdst]   = ~(GPR[`rsrc1] & GPR[`rsrc2]);
180.       end
```

```
181.
182.       ////////////////////////////////////////////////////////////bitwise nor
183.
184.       `rnor : begin
185.            if(`imm_mode)
186.              GPR[`rdst]  = ~(GPR[`rsrc1] | `isrc);
187.           else
188.             GPR[`rdst]   = ~(GPR[`rsrc1] | GPR[`rsrc2]);
189.       end
190.
191.       ////////////////////////////////////////////////////////////not
192.
193.       `rnot : begin
194.            if(`imm_mode)
195.              GPR[`rdst]  = ~(`isrc);
196.           else
197.              GPR[`rdst]   = ~(GPR[`rsrc1]);
198.       end
199.
200.       ////////////////////////////////////////////////////////////
201.
202.       `storedin: begin
203.          data_mem[`isrc] = din;
204.       end
205.
206.       ////////////////////////////////////////////////////////////
207.
208.       `storereg: begin
209.          data_mem[`isrc] = GPR[`rsrc1];
210.       end
211.
212.       ////////////////////////////////////////////////////////////
213.
214.
215.       `senddout: begin
216.          dout  = data_mem[`isrc];
217.       end
218.
219.       ////////////////////////////////////////////////////////////
220.
221.       `sendreg: begin
222.         GPR[`rdst] =  data_mem[`isrc];
223.       end
224.
225.       ////////////////////////////////////////////////////////////
226.
227.       `jump: begin
228.        jmp_flag = 1'b1;
229.       end
230.
231.       `jcarry: begin
232.         if(carry == 1'b1)
233.            jmp_flag = 1'b1;
234.          else
235.            jmp_flag = 1'b0;
236.       end
237.
238.       `jsign: begin
239.         if(sign == 1'b1)
240.            jmp_flag = 1'b1;
241.          else
```

```
242.            jmp_flag = 1'b0;
243.        end
244.
245.        `jzero: begin
246.          if(zero == 1'b1)
247.            jmp_flag = 1'b1;
248.           else
249.            jmp_flag = 1'b0;
250.        end
251.
252.
253.        `joverflow: begin
254.          if(overflow == 1'b1)
255.            jmp_flag = 1'b1;
256.           else
257.            jmp_flag = 1'b0;
258.        end
259.
260.        `jnocarry: begin
261.          if(carry == 1'b0)
262.            jmp_flag = 1'b1;
263.           else
264.            jmp_flag = 1'b0;
265.        end
266.
267.        `jnosign: begin
268.          if(sign == 1'b0)
269.            jmp_flag = 1'b1;
270.           else
271.            jmp_flag = 1'b0;
272.        end
273.
274.        `jnozero: begin
275.          if(zero == 1'b0)
276.            jmp_flag = 1'b1;
277.           else
278.            jmp_flag = 1'b0;
279.        end
280.
281.
282.        `jnooverflow: begin
283.          if(overflow == 1'b0)
284.            jmp_flag = 1'b1;
285.           else
286.            jmp_flag = 1'b0;
287.        end
288.
289.        ////////////////////////////////////////////////////////
290.        `halt : begin
291.        stop = 1'b1;
292.        end
293.
294.        endcase
295.
296.        end
297.        endtask
298.
299.
300.
301.        ///////////////////////logic for condition flag
302.
```

```
303.
304.       task decode_condflag();
305.       begin
306.
307.       /////////////////sign bit
308.       if(`oper_type == `mul)
309.         sign = SGPR[15];
310.       else
311.         sign = GPR[`rdst][15];
312.
313.       ////////////////carry bit
314.
315.       if(`oper_type == `add)
316.          begin
317.             if(`imm_mode)
318.                begin
319.                temp_sum = GPR[`rsrc1] + `isrc;
320.                carry    = temp_sum[16];
321.                end
322.             else
323.                begin
324.                temp_sum = GPR[`rsrc1] + GPR[`rsrc2];
325.                carry    = temp_sum[16];
326.                end   end
327.          else
328.           begin
329.               carry  = 1'b0;
330.           end
331.
332.
333.       ////////////////////// zero bit
334.
335.        zero =   ( ~(|GPR[`rdst]) ~(|SGPR[15:0]) );
336.
337.
338.       ////////////////////////overflow bit
339.
340.       if(`oper_type == `add)
341.            begin
342.              if(`imm_mode)
343.                overflow = ( (~GPR[`rsrc1][15] & ~IR[15] & GPR[`rdst][15] ) | (GPR[`rsrc1][15]
    & IR[15] & ~GPR[`rdst][15]) );
344.              else
345.                overflow = ( (~GPR[`rsrc1][15] & ~GPR[`rsrc2][15] & GPR[`rdst][15]) |
    (GPR[`rsrc1][15] & GPR[`rsrc2][15] & ~GPR[`rdst][15]));
346.            end
347.         else if(`oper_type == `sub)
348.           begin
349.              if(`imm_mode)
350.                overflow = ( (~GPR[`rsrc1][15] & IR[15] & GPR[`rdst][15] ) | (GPR[`rsrc1][15]
    & ~IR[15] & ~GPR[`rdst][15]) );
351.              else
352.                overflow = ( (~GPR[`rsrc1][15] & GPR[`rsrc2][15] & GPR[`rdst][15]) |
    (GPR[`rsrc1][15] & ~GPR[`rsrc2][15] & ~GPR[`rdst][15]));
353.           end
354.         else
355.            begin
356.            overflow = 1'b0;
357.            end
358.
359.       end
```

```
360.       endtask
361.
362.
363.
364.
365.
366.       ////////////////////////////////////////////////////////////////////////////////////////
   ////////////////////
367.
368.       ////////////////////////////////////////////
369.       ///////////reading program
370.
371.       initial begin
372.       $readmemb("inst_data.mem",inst_mem);
373.       end
374.
375.       ////////////////////////////////////////////////
376.       //////////reading instructions one after another
377.       reg [2:0] count = 0;
378.       integer PC = 0;
379.       /*
380.       always@(posedge clk)
381.       begin
382.         if(sys_rst)
383.          begin
384.            count <= 0;
385.            PC    <= 0;
386.          end
387.          else
388.          begin
389.            if(count < 4)
390.            begin
391.            count <= count + 1;
392.            end
393.            else
394.            begin
395.            count <= 0;
396.            PC    <= PC + 1;
397.            end
398.        end
399.       end
400.       */
401.       ////////////////////////////////////////////////
402.       ////////reading instructions
403.       /*
404.       always@(*)
405.       begin
406.       if(sys_rst == 1'b1)
407.       IR = 0;
408.       else
409.       begin
410.       IR = inst_mem[PC];
411.       decode_inst();
412.       decode_condflag();
413.       end
414.       end
415.       */
416.       ////////////////////////////////////////////////
417.       ////////////////////////////////// fsm states
418.       parameter idle = 0, fetch_inst = 1, dec_exec_inst = 2, next_inst = 3, sense_halt = 4,
   delay_next_inst = 5;
```

```
419.       //////idle : check reset state
420.       ///// fetch_inst : load instrcution from Program memory
421.       ///// dec_exec_inst : execute instruction + update condition flag
422.       ///// next_inst : next instruction to be fetched
423.       reg [2:0] state = idle, next_state = idle;
424.       //////////////////////////////// fsm states
425.
426.       ////////////////////reset decoder
427.       always@(posedge clk)
428.       begin
429.        if(sys_rst)
430.          state <= idle;
431.        else
432.          state <= next_state;
433.       end
434.
435.
436.       ////////////////////next state decoder + output decoder
437.
438.       always@(*)
439.       begin
440.         case(state)
441.          idle: begin
442.            IR         = 32'h0;
443.            PC         = 0;
444.            next_state = fetch_inst;
445.          end
446.
447.         fetch_inst: begin
448.           IR          =  inst_mem[PC];
449.           next_state  = dec_exec_inst;
450.         end
451.
452.         dec_exec_inst: begin
453.           decode_inst();
454.           decode_condflag();
455.           next_state  = delay_next_inst;
456.         end
457.
458.
459.         delay_next_inst:begin
460.         if(count < 4)
461.              next_state  = delay_next_inst;
462.            else
463.              next_state  = next_inst;
464.         end
465.
466.         next_inst: begin
467.             next_state = sense_halt;
468.             if(jmp_flag == 1'b1)
469.               PC = `isrc;
470.             else
471.               PC = PC + 1;
472.          end
473.
474.
475.        sense_halt: begin
476.           if(stop == 1'b0)
477.             next_state = fetch_inst;
478.           else if(sys_rst == 1'b1)
479.             next_state = idle;
```

```
480.           else
481.             next_state = sense_halt;
482.        end
483.
484.         default : next_state = idle;
485.
486.         endcase
487.
488.       end
489.
490.
491.       ////////////////////////////////// count update
492.
493.       always@(posedge clk)
494.       begin
495.       case(state)
496.
497.        idle : begin
498.           count <= 0;
499.        end
500.
501.        fetch_inst: begin
502.          count <= 0;
503.        end
504.
505.        dec_exec_inst : begin
506.          count <= 0;
507.        end
508.
509.        delay_next_inst: begin
510.          count  <= count + 1;
511.        end
512.
513.         next_inst : begin
514.           count <= 0;
515.        end
516.
517.         sense_halt : begin
518.           count <= 0;
519.        end
520.
521.        default : count <= 0;
522.
523.
524.       endcase
525.       end
526.
527.
528.
529.       endmodule
```