آزمایش هشتم: ALU اعداد مختلط

برای طراحی این مدار، ماژول هایی مانند جمع/ تفریق کننده، ضرب کننده، حافظه داده و دستور و کنترلر داشته باشیم. حافظه داده این مدار، حافظه ای به اندازه 64 کلمه 8 بیتی است که قابلیت خواندن، نوشتن و ریست شدن دارد. بنابراین سیگنال های clk، write_addr، read_addr، write و read و کلمه 8 بیتی write_data، ورودی های این مدار و کلمه 8 بیتی read_data، خروجی آن است. سیگنال ریست آسنکرون و active low است.

اگر reset = 0 باشد، تمام خانه های حافظه برابر صفر و در غیر اینصورت اگر سیگنال write فعال باشد داده مورد نظر در آدرس مشخص شده حافظه نوشته و اگر سیگنال read فعال باشد، از آن خوانده می شود.

```
21  module data_mem #(parameter DATA_WIDTH = 8, MEMORY_SIZE = 64)
22                   (input clk,
23                    input [$clog2(MEMORY_SIZE)-1:0] write_addr, read_addr,
24                    input [DATA_WIDTH-1:0] write_data,
25                    input write,
26                    input read,
27                    input reset,
28                    output [DATA_WIDTH-1:0] read_data);
29
30      reg [DATA_WIDTH-1:0] mem [MEMORY_SIZE-1:0];
31      integer i;
32
33      always @(posedge clk or negedge reset) begin
34        if (!reset) begin
35           for (i = 0 ; i < MEMORY_SIZE; i=i+1)
36              mem[i] = 0;
37        end else begin
38           if (write) mem[write_addr] = write_data;
39           if (read) read_data = mem[read_addr];
40        end
41      end
42  endmodule
```

حافظه دستور هم مانند حافظه داده پیاده سازی می شود. این حافظه گنجایش 32 دستورالعمل 20 بیتی را دارد.

```
21  module instruction_mem #(parameter INSTRUCTION_WIDTH = 20, MEMORY_SIZE = 32)
22                          (input [$clog2(MEMORY_SIZE)-1:0] addr,
23                           input [INSTRUCTION_WIDTH-1:0] data_in,
24                           input write_en, reset, clk,
25                           output reg [INSTRUCTION_WIDTH-1:0] data_out);
26
27     reg [INSTRUCTION_WIDTH-1:0] memory [MEMORY_SIZE-1:0];
28
29     integer i;
30
31     always @(posedge clk or negedge reset) begin
32        if (!reset) begin
33           for (i = 0 ; i < MEMORY_SIZE; i=i+1)
34              memory[i] = 0;
35        end else if (write_en) memory[addr] = data_in;
36        else data_out = memory[addr];
37
38     end
39
40  endmodule
```

برای پیاده سازی ماژول های ضرب کننده و جمع/ تفریق کننده اعداد مختلط، از ماژول های ضرب و جمع/تفریق عادی به نوعی استفاده می کنیم که عملیات جمع/تفریق در 2 کلاک (2 جمع/تفریق) و ضرب در 5 کلاک (4 ضرب، 1 جمع و 1 تفریق) انجام

شود. برای این کار از دو رجیستر و یک سیم (src1, src2, dst) به عنوان ورودی adder استفاده می کنیم. یک رجیستر به نام counter برای مشخص کردن اینکه مقدار src1 و src2 چه باشد، تعریف می کنیم. اگر enable = 0 باشد، counter = 0 می شود. بعد از فعال شدن enable، اگر counter = 00 بود، src1 و src2 به ترتیب برابر 4 بیت پرارزش a و b می شوند و counter = 1 می شود. در حالت بعدی، ابتدا 4 بیت پرارزش result برابر حاصل جمع/ تفریق دو ورودی قبلی می شود و سپس مقدار src1 و src2، آپدیت می شود و برابر 4 بیت کم ارزش a و b می شود و counter = 2 می شود. در حالت نهایی هم 4 بیت کم ارزش result با مقدار جدید dst پر می شود و counter=0 می شود.

در ماژول adder، برای مشخص کردن جمع یا تفریق، از ورودی add_en استفاده می شود.

```
21  module cadder #(parameter DATA_WIDTH = 8)
22                 (input [DATA_WIDTH-1:0] A, B,
23                  input add_en,
24                  input enable, clk,
25                  output reg ready,
26                  output reg [DATA_WIDTH-1:0] result);
27
28     reg [1:0] counter;
29
30     reg [DATA_WIDTH/2 - 1: 0] src1;
31     reg [DATA_WIDTH/2 - 1: 0] src2;
32     wire [DATA_WIDTH/2 - 1: 0] dst;
33
34     adder #(DATA_WIDTH/2) adder (src1, src2, add_en, dst);
35
36    always @(posedge clk) begin
37       if (enable) begin
38          case (counter)
39             2'd2: begin
40                result[DATA_WIDTH/2 - 1:0] = dst;
41                counter = 2'd0;
42                ready = 1'b1;
43             end
44             2'd1: begin
45                result[DATA_WIDTH-1: DATA_WIDTH/2] = dst;
46                src1 = A[DATA_WIDTH/2 - 1:0];
47                src2 = B[DATA_WIDTH/2 - 1:0];
48                counter = 2'd2;
49             end
50             2'd0: begin
51                src1 = A[DATA_WIDTH-1: DATA_WIDTH/2];
52                src2 = B[DATA_WIDTH-1: DATA_WIDTH/2];
53                ready = 1'b0;
54                counter = 2'd1;
55             end
56          endcase
57       end else begin
58          counter = 2'd0;
59          ready = 1'b0;
60       end
61    end
62
63 endmodule
```

```
21  module adder #(parameter DATA_WIDTH = 8)
22                (input [DATA_WIDTH-1:0] A, B,
23                 input add_en,
24                 output reg [DATA_WIDTH-1:0] result);
25
26    always @(A or B) begin
27       result = (add_en)? (A+B): (A-B);
28    end
29
30 endmodule
```

برای پیاده سازی ماژول ضرب کننده هم از همین روش استفاده می کنیم. بدین صورت که سه مقدار adder_src1، addr_src2 و adder_dst را به ماژول adder و سه ورودی mult_src1، mult_src2 و mult_dst را به ماژول multiplier می دهیم. دوباره از counter استفاده می کنیم. بدین صورت که اگر counter = 0 بود، حاصلضرب 4 بیت پرارزش هر دو ورودی را بدست آورده و counter را یکی اضافه می کنیم. در مرحله بعد هم حاصلضرب 4 بیت کم ارزش هر دو را

محاسبه می کنیم و ++counter. وقتی counter=2 بود، حاصلضرب 4 بیت پرارزش a در 4 بیت کم ارزش b را بدست آورده و عمل تفریق را بین دو حاصلضرب قبلی انجام می دهیم. وقتی counter = 3 شد، عمل ضرب 4 بیت کم ارزش a در 4 بیت پرارزش b را انجام می دهیم و در مرحله آخر حاصل جمع آن ها را بدست می آوریم.

```
21  module cmult #(parameter WIDTH = 8)(
22      input [WIDTH-1:0] A,
23      input [WIDTH-1:0] B,
24      input clk,
25      input enable,
26      output reg ready,
27      output reg[WIDTH-1: 0] mult
28      );
29
30      reg [2:0] counter;
31
32      reg [WIDTH/2 - 1:0] adder_src1;
33      reg [WIDTH/2 - 1:0] adder_src2;
34      wire [WIDTH/2 - 1:0] adder_dst;
35
36      reg add_en;
37
38      reg [WIDTH/2 - 1:0] mult_src1;
39      reg [WIDTH/2 - 1:0] mult_src2;
40      wire [WIDTH/2 - 1:0] mult_dst;
41
42      reg [WIDTH/2 - 1:0] temp1;
43      reg [WIDTH/2 - 1:0] temp2;
44
45      adder #(WIDTH/2) adder (adder_src1, adder_src2, add_en, adder_dst);
46      multiplier #(WIDTH/2) multer (mult_src1, mult_src2, mult_dst);
47
48      always @(posedge clk) begin
49         if(enable) begin
50            case (counter)
51              3'd5: begin
52                 mult[WIDTH/2 - 1:0] = adder_dst;
53                 ready = 1'b1;
54                 counter = 3'd0;
55              end
56              3'd4: begin
57                 temp2 = mult_dst;
58                 adder_src1 = temp1;
59                 adder_src2 = temp2;
60                 add_en = 1'b1;
61                 counter = 3'd5;
62              end
63              3'd3: begin
64                 mult[WIDTH-1: WIDTH/2] = adder_dst;
65                 temp1 = mult_dst;
66                 mult_src1 = A[WIDTH/2 - 1:0];
67                 mult_src2 = B[WIDTH-1: WIDTH/2];
68                 counter = 3'd4;
69              end
70              3'd2: begin
71                 temp2 = mult_dst;
72                 mult_src1 = A[WIDTH-1: WIDTH/2];
73                 mult_src2 = B[WIDTH/2 - 1:0];
74                 adder_src1 = temp1;
75                 adder_src2 = temp2;
76                 add_en = 1'b0;
77                 counter = 3'd3;
78              end
79              3'd1: begin
80                 temp1 = mult_dst;
81                 mult_src1 = A[WIDTH/2 - 1:0];
82                 mult_src2 = B[WIDTH/2 - 1:0];
83                 counter = 3'd2;
84              end
```

```
            3'd0: begin
               mult_src1 = A[WIDTH-1: WIDTH/2];
               mult_src2 = B[WIDTH-1: WIDTH/2];
               ready = 1'b0;
               counter = 3'd1;
            end
          endcase
        end else begin
          counter = 3'd0;
          ready = 1'b0;
        end
      end
endmodule
```

```
module multiplier #(parameter DATA_WIDTH = 8)
                   (input [DATA_WIDTH-1:0] A, B,
                    output reg [DATA_WIDTH-1:0] result);

   always @(A or B) begin
      result = A * B;
   end

endmodule

```

در این مرحله بعد از پیاده سازی جمع کننده، ضرب کننده و حافظه ها، برای هر یک از عملیات fetch، load، execute و save یک کنترلر طراحی کرده و همه آن ها را به کنترلر مرکزی متصل می کنیم.

با fetch_controller آغاز می کنیم. این ماژول، رجیستر pc را به عنوان ورودی دریافت کرده و دستورالمعل مربوط به آن را به صورت دیکود شده (opcode, src1, src2, dst) خروجی می دهد و تمام شدن مرحله fetch را به سیگنال finish و ready نشان می دهد. در این ماژول مقدار pc را در mar نگهداری می کنیم و دستورالعمل موجود در خانه به آدرس mar در حافظه دستور را در mir می ریزیم. حالا با کمک counter، شروع به دیکود دستور می کنیم به این صورت که اگر counter=0 بود، 2 بیت اول mir مربوط به opcode و 18 بیت باقی مانده به طور مساوی بین src1، src2 و dst تقسیم می شوند. بعد از این مرحله است که کار این ماژول به اتمام رسیده و وارد مرحله load می شویم.

```
module fetch_cont #(parameter INS_MEMORY_SIZE = 32, DATA_MEMORY_SIZE = 64, WIDTH = 3*$clog2(DATA_MEMORY_SIZE) + 2)
                   (input [$clog2(INS_MEMORY_SIZE)-1:0] pc, input clk,
                    output reg [1:0] opcode,
                    output reg [$clog2(DATA_MEMORY_SIZE)-1:0] src1, src2, dst,
                    output reg ready = 1'b0,
                    output reg finished);
    wire[$clog2(INS_MEMORY_SIZE)-1:0] mar;
    wire[WIDTH-1:0] mir;
    assign mar = pc;
    reg counter;
    instruction_mem #(.INSTRUCTION_WIDTH(WIDTH), .MEMORY_SIZE(INS_MEMORY_SIZE)) im
    (.addr(mar), .data_in({WIDTH{1'b0}}), .write_en(1'b0), .clk(clk),
    .reset(reset), .data_out(mir));

    always @(posedge clk) begin
    if (!finished) begin
         case (counter)
            1'b1: begin
               ready = 1'b1;
               counter = 1'b0;
               if (opcode == 2'b00) finished = 1'b1;
            end
            1'b0: begin
               src1 = mir[(WIDTH-2)/3 - 1:0];
               src2 = mir[2*(WIDTH-2)/3 - 1: (WIDTH-2)/3];
               dst = mir[WIDTH-3: 2*(WIDTH-2)/3];
               opcode = mir[WIDTH-1: WIDTH-2];
               counter = 1'b1;
               ready = 1'b0;
            end
         endcase
       end
    end
endmodule
```

در ماژول load، باید از روی آدرس داده ها، آنها را از حافظه داده به src1، src2 و dst منتقل کنیم. در ابتدا src1 و سپس src2 مقداردهی می شوند و خانه به آدرس dst برای ذخیره داده جدید آماده می شود.

```
module load_handler #(parameter DATA_WIDTH = 8, DATA_MEMORY_SIZE = 64)
                     (input [1:0] opcode_in,
                      input [$clog2(DATA_MEMORY_SIZE)-1:0] src1_addr, src2_addr, dst_addr,
                      input [DATA_WIDTH-1:0] data_in,
                      input enable, clk,
                      output reg [1:0] opcode_out,
                      output reg [DATA_WIDTH-1:0] src1, src2,
                      output reg [$clog2(DATA_MEMORY_SIZE)-1:0] addr_out, dst_out,
                      output reg ready = 1'b0);

   reg [1:0] counter;

   reg [$clog2(DATA_MEMORY_SIZE)-1:0] src1_adr, src2_adr, dst_adr;

   always @(posedge clk) begin

      if (enable) begin
         case (counter)
            2'd2: begin
               src2 = data_in;
               opcode_out = opcode;
               dst_out = dst_adr;
               ready = 1'b1;
               counter = 2'd0;
            end
            2'd1: begin
               src1 = data_in;
               addr_out = src2_adr;
               counter = 2'd2;
            end
            2'd0: begin
               src1_adr = src1_addr;
               src2_adr = src2_addr;
               dst_adr = dst_addr;
               addr_out = src1_adr;
               ready = 1'b0;
               counter = 2'd1;
            end
         endcase
      end else begin
         counter = 2'd0;
      end

   end

endmodule
```

ماژول کنترلر execute، مسئول اجرای دستورالعمل است. بدین صورت که اگر مشغول اجرای دستورالعملی نباشد، مقادیر src1، src2 و dst را لود کرده و با توجه به opcode موجود تصمیم می گیرد چه عملی (جمع، تفریق یا ضرب) انجام شود. این ماژول اتمام کار خود را با سیگنال ready نمایش می دهد که اگر کار هر یک از ماژول های جمع کننده یا ضرب کننده تمام شود، این سیگنال 1 می شود.

```
21  module execont #(parameter WIDTH = 8, SIZE = 64)
22                  (input [$clog2(SIZE)-1:0] dst_addr, input [WIDTH-1:0] src1, src2,
23                   input[1:0] opcode, input enable, clk,
24                   output ready,
25                   output [WIDTH-1:0] dst,
26                   output [$clog2(SIZE)-1:0] dst_out
27      );
28      reg working;
29      reg [$clog2(SIZE)-1:0] dst_adr;
30      reg [WIDTH-1:0] src1_real, src2_real;
31      reg [1:0] opcode_real;
32      reg add_en, mul_en, add_sub;
33      assign ready = mul_ready | add_ready;
34      cadder cadd(.A(src1_real), .B(src2_real), .result(dst), .clk(clk), .enable(add_en),
35      .add_en(add_sub), .ready(add_ready));
36      cmult cmult(.A(src1_real), .B(src2_real), .mult(dst), .clk(clk), .enable(mul_en),
37      .ready(mul_ready));
38      always @(posedge clk) begin
39        if (enable) begin
40           if (ready) working = 1'b0;
41           else working = 1'b1;
42           case(opcode_real)
43               2'b10: begin //add
44               add_en = 1;
45               mul_en = 0;
46               add_sub = 1;
47               end
48               2'b11: begin //sub
49               add_en = 1;
50               mul_en = 0;
51               add_sub = 0;
52               end
53               2'b01: begin //mul
54               add_en = 0;
55               mul_en = 1;
56               add_sub = 0;
57               end
58           endcase
59           if (!working) begin
60              dst_adr = dst_addr;
61              src1_real = src1;
62              src2_real = src2;
63              opcode_real = opcode;
64           end
65
66        end else begin
67           working = 1'b0;
68           add_en = 0;
69           mul_en = 0;
70        end
71
72      end
73
74  endmodule
```

در آخر هم ماژول save مسئول ذخیره داده جدید در آدرس dst است. طرز کار این ماژول مانند load است.

```
21  module save_handler #(parameter DATA_WIDTH = 8, DATA_MEMORY_SIZE = 64)
22                       (input [$clog2(DATA_MEMORY_SIZE)-1:0] dst_addr,
23                        input [DATA_WIDTH-1:0] data_in,
24                        input enable, clk,
25                        output reg [$clog2(DATA_MEMORY_SIZE)-1:0] addr_out,
26                        output reg [DATA_WIDTH-1:0] data_out,
27                        output reg ready = 1'b0);
28
29     reg counter;
30
31     always @(posedge clk) begin
32
33        if (enable) begin
34           case (counter)
35              1'b1: begin
36                 ready = 1'b1;
37                 counter = 1'b0;
38              end
39              1'b0: begin
40                 addr_out = dst_addr;
41                 data_out = data_in;
42                 ready = 1'b0;
43                 counter = 1'b1;
44              end
45           endcase
46        end else begin
47           counter = 1'b0;
48        end
49
50     end
51
52  endmodule
```

در انتها برای برقراری ارتباط بین این 4 ماژول کنترل کننده، به یک کنترلر مرکزی نیاز داریم. در این کنترلر زمانی یک دستور کامل اجرا می شود که سیگنال finished فعال شده باشد و داده ها در حال تغییر نباشند. زمانی که سیگنال save_ready فعال شود، سیگنال enable و changing مربوط به آن خانه حافظه غیرفعال می شود. زمانی که سیگنال execution_ready فعال شود، مدار باید آماده ذخیره داده شود و مرحله اجرا به پایان می رسد. زمانی که مرحله لود تمام شود، باید آماده تغییر دادن داده موجود در خانه به آدرس dst حافظه و اجرای دستور شویم. زمانی که مرحله fetch تمام شود، اگر دستورالعملی که درحال اجراست، دارد تغییری در خانه ای از حافظه ایجاد می کند، صبر می کنیم و سپس این دستور را به مرحله اجرا برده و pc را یک واحد افزایش داده و دستور بعدی را fetch می کنیم.

```
21  module controller #(parameter DATA_WIDTH = 8, INS_MEM_SIZE = 32, DATA_MEM_SIZE = 64)
22                     (input clk, reset, output executed);
23
24
25     reg [$clog2(INS_MEM_SIZE)-1:0] pc;
26     reg [DATA_MEM_SIZE-1:0] changing;
27     reg finished;
28     integer i;
29
30     assign executed = finished & (!(|changing));
31
32     data_mem #(.DATA_WIDTH(DATA_WIDTH), .MEMORY_SIZE(DATA_MEM_SIZE)) data_memory
33     (.clk(clk), .write_addr(save_addr), .read_addr(load_addr), .write_data(data_write),
34      .write(load_enable), .read(save_enable), .reset(reset), .read_data(data_read));
35
36     fetch_cont #(.INS_MEMORY_SIZE(INS_MEM_SIZE), .DATA_MEMORY_SIZE(DATA_MEM_SIZE)) fetch_handle
37     (.pc(pc), .clk(clk), .opcode(op), .src1(src1_adr), .src2(src2_adr), .dst(dst_adr),
38      .ready(fetch_ready), .finished(finished));
39
40     load_handler #(.DATA_WIDTH(DATA_WIDTH), .DATA_MEMORY_SIZE(DATA_MEM_SIZE)) load_handle
41     (.opcode_in(op), .src1_addr(src1_adr), .src2_addr(src2_adr), .dst_addr(dst_adr),
42      .data_in(data_read), .enable(load_enable), .clk(clk), .opcode_out(opc),
43      .src1(source1), .src2(source2), .addr_out(load_addr), .dst_out(dst_out), .ready(load_ready));
44
45     execont #(.WIDTH(DATA_WIDTH), .SIZE(DATA_MEM_SIZE)) execution_handle
46     (.dst_addr(dst_out), .src1(source1), .src2(source2), .opcode(opc), .enable(execution_enable),
47      .clk(clk), .ready(execution_ready), .dst(res), .dst_out(dst_address));
48
49     save_handler #(.DATA_WIDTH(DATA_WIDTH), .DATA_MEMORY_SIZE(DATA_MEM_SIZE)) save_handle
50     (.dst_addr(dst_address), .data_in(res), .enable(save_enable), .clk(clk), .addr_out(save_addr),
51      .data_out(data_write), .ready(save_ready));
52
53     always @(negedge reset) begin
54
55           changing = {DATA_MEM_SIZE{1'b0}};
56           pc = 0;
57           load_enable = 1'b0;
58           execution_enable = 1'b0;
59           save_enable = 1'b0;
60
61     end
62
63     always @(posedge save_ready) begin
64        save_enable = 0;
65        changing[save_addr] = 0;
66     end
67
68     always @(posedge execution_ready) begin
69        execution_enable = 0;
70        save_enable = 1'b1;
71     end
72
73     always @(posedge load_ready) begin
74        load_enable = 0;
75        changing[dst_adr] = 1'b1;
76        if (execution_enable) begin
77           @(negedge execution_enable) execution_enable = 1'b1;
78        end else begin
79           execution_enable = 1'b1;
80        end
81     end
82
83     always @(posedge fetch_ready) begin
84        if (load_enable) begin
```

```
85          @(negedge load_enable) begin
86             if (changing[src1_adr]) begin
87                @(negedge changing[src1_adr]) begin
88                   if (changing[src2_adr]) begin
89                      @(negedge changing[src2_adr]) begin
90                         load_enable = 1'b1;
91                         pc = pc + 1'b1;
92                      end
93                   end
94                end
95             end else if (changing[src2_adr]) begin
96                @(negedge changing[src2_adr]) begin
97                   load_enable = 1'b1;
98                   pc = pc + 1'b1;
99                end
100            end else begin
101               load_enable = 1'b1;
102               pc = pc + 1'b1;
103            end
104      end end else begin
105         begin
106            if (changing[src1_adr]) begin
107               @(negedge changing[src1_adr]) begin
108                  if (changing[src2_adr]) begin
109                     @(negedge changing[src2_adr]) begin
110                        load_enable = 1'b1;
111                        pc = pc + 1'b1;
112                     end
113                  end
114               end
115            end else if (changing[src2_adr]) begin
116               @(negedge changing[src2_adr]) begin
117                  load_enable = 1'b1;
118                  pc = pc + 1'b1;
119               end
120            end else begin
121               load_enable = 1'b1;
122               pc = pc + 1'b1;
123            end
124      end
125      end
126   end
127
128
129   endmodule
```