به نام خدا

گزارش کار آزمایشگاه معماری کامپیوتر

محمدرضا عظیمی ۸۱۰۱۹۶۵۱۰

علی جعفرزاده ۸۱۰۱۹۵۳۷۳

فهرست مطالب

مقدمه

در این گزارش قصد داریم روند پیاده سازی پردازنده‌ی ARM با استفاده از شبیه ساز Modelsim و به زبان Verilog را شرح دهیم.

بنابراین ابتدا مراحل و قسمت های مختلف پیاده سازی این پردازنده را شرح داده و سپس در یک TestBench و با ورودی های محک، پردازنده خود را مورد آزمایش قرار می دهیم.

معرفی پردازنده ARM

پردازنده ARM شامل ۵ مرحله و ۴ رجیستر در میان این مراحل می باشد. پردازنده ای که ما در این آزمایش پیاده سازی می کنیم، یک نمونه ی ساده از پردازنده ی ARM است که دارای ۱۳ دستورالعمل اصلی می باشد.

این پردازنده که از ۵ مرحله خط لوله تشکیل می شود، خط آدرسی به پهنای ۳۲ بیت بهمراه ۱۶ ثبات همه منظوره است که ثبات ۱۵ به منظور PC استفاده می شود و ثبات ۱۴ نیز به عنوان Link Register .

آدرس دهی در این پردازنده بر حسب بایت بوده و فضای آدرس داده و دستورات مجزا می باشد.

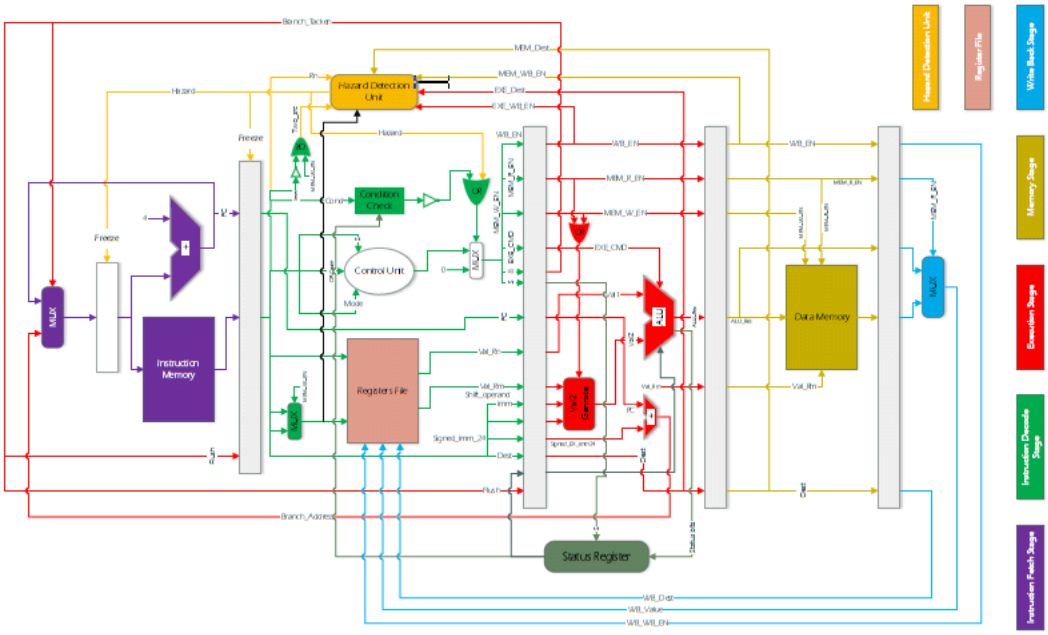
در این پردازنده تمامی پرش ها از نوع محلی تعریف شده است و پس از پرش مقدار رجیستر شمارنده دستور به شکل زیر خواهد بود.

PC=PC+ (signed_immed_24<<2) +4

این پردازنده همچنین مجهز به واحد تشخیص مخاطره است که در بخش مقتضی به آن خواهیم پرداخت.

لازم به ذکر است که واحد ارسال به جلو در این پردازنده پیاده سازی نمی شود.

تصویر مربوط به معماری این پردازنده را در زیر مشاهده می کنیم:



این پردازنده در لبه بالارونده هر clock، بر اساس مقدار PC در مرحله واکشی (IF) دستوری از حافظه دستورات خوانده و از طریق رجیستر میانی به مرحله کد گشایی یعنی ID ارسال می کند.

در این پردازنده در هر لحظه حداکثر ۵ دستور بصورت همزمان در حال اجرا می باشند که در نتیجه‌ی استفاده از قابلیت خط لوله در پیاده سازی این پردازنده است.

مرحله واکشی (Instruction Fetch)

اولین مرحله از معماری پردازنده ARM، مرحله IF می باشد. وظیفه این مرحله این است که ابتدا مقدار PC را محاسبه کرده و سپس دستور متناظر با آن را از حافظه دستورات استخراج کند.

پس از انجام این کار، دستور را از طریق رجیستر میانی یعنی IF2ID به مرحله بعد ارسال می کند.

در این بخش یک Register، Mux، Adder و یک Instruction Memory وجود دارد. تمامی دستورات برنامه در Instruction Memory قرار دارند.

IFSTAGE > IFstage.v

```
1   module IFSTAGE (input clk ,rst ,freeze ,branch_track ,input[31:0] branch_addr,
2                   output[31 :0] instruction, pc );
3     wire[31 :0] nextPc ,npc;
4
5     Adder adder(4 ,pc ,nextPc);
6
7     Multiplexer_2inputs#(.N(32)) mux (branch_addr ,nextPc ,branch_track ,npc);
8
9     PC Pc(clk ,rst ,~freeze ,npc ,pc);
10
11    InstrcutionSet instructionSet(pc ,instruction);
12  endmodule // IFSTAGE
13
```

ماژول Adder مقدار PC را با عدد ۴ جمع می کند.

IFSTAGE > adder.v

```
1   module Adder (input [31 :0] a , b ,output [31: 0] result);
2     assign result = a + b;
3   endmodule // Adder
4
```

همچنین ماژول Multiplexer_2inputs ورودی را از بین branch_addr و nextPc انتخاب می کند.

Multiplexer_2inputs.v

```v
module Multiplexer_2inputs #(parameter N)(a ,b ,mode ,out);
      input [N-1 : 0] a ,b;
      input mode;
      output [N-1 :0] out;

      assign out = mode ? a : b;

endmodule // Multiplexer
```

در ماژول IF2ID، مقادیر به مرحله بعد منتقل می شوند.

در صورت فعال بودن سیگنال flush، تمامی مقادیر صفر شده و در غیر اینصورت، مقادیر از ورودی به خروجی منتقل می شوند.

تمام این کارها به شرطی رخ می دهد که سیگنال freeze غیر فعال باشد. در غیر اینصورت مقادیر در رجیستر باقی مانده و منتقل نمی شوند.

Register > IF2IDReg.v

```v
module IF2ID (input clk, rst, flush, freeze, input[31: 0] pc_in, instruction_in,
              output reg [31: 0] pc, instruction);

  always @ (posedge clk, posedge rst) begin
    if (rst) begin
      pc <= 0;
      instruction <= 0;
    end
    else begin
      if (~freeze) begin
        if (flush) begin
          instruction <= 0;
          pc <= 0;
        end
        else begin
          instruction <= instruction_in;
          pc <= pc_in;
        end
      end
    end
  end

endmodule // IF2ID
```

در ماژول InstructionSet، دستورات از فایلی که بعنوان ورودی داده شده است خوانده می شوند و سپس در رجیستر ذخیره خواهند شد.

IFSTAGE > instructionSet.v

```
1  module InstrcutionSet (input[31 :0] address ,output [31 :0] instructionOut);
2      reg [7 :0] instruction[0 : 4095];
3      initial begin $readmemb("instructions.txt", instruction); end
4      assign instructionOut =
5          {instruction[address] ,instruction[address + 1] ,instruction[address + 2] ,instruction[address + 3]};
6  endmodule // InstrcutionSet
7
```

این حافظه شامل ۴۰۹۶ خط داده‌ی ۸ بیتی بوده که هر ۴ خط از آن (یعنی ۳۲ بیت) یک دستور را تشکیل می دهند.

مرحله کدگشایی (Instruction Decode)

در این مرحله که دومین مرحله از مراحل معماری خط لوله پردازنده ARM می باشد، دستوری که از مرحله واکشی رسیده با توجه به نوع آن تفکیک شده و سیگنال ها و داده های مورد نیاز از درون آن استخراج می شود.

در این مرحله غیر از رجیستر میانی ID2EXE، ماژول های ConditionCheck، ControlUnit و RegisterFile نیز قرار دارند.

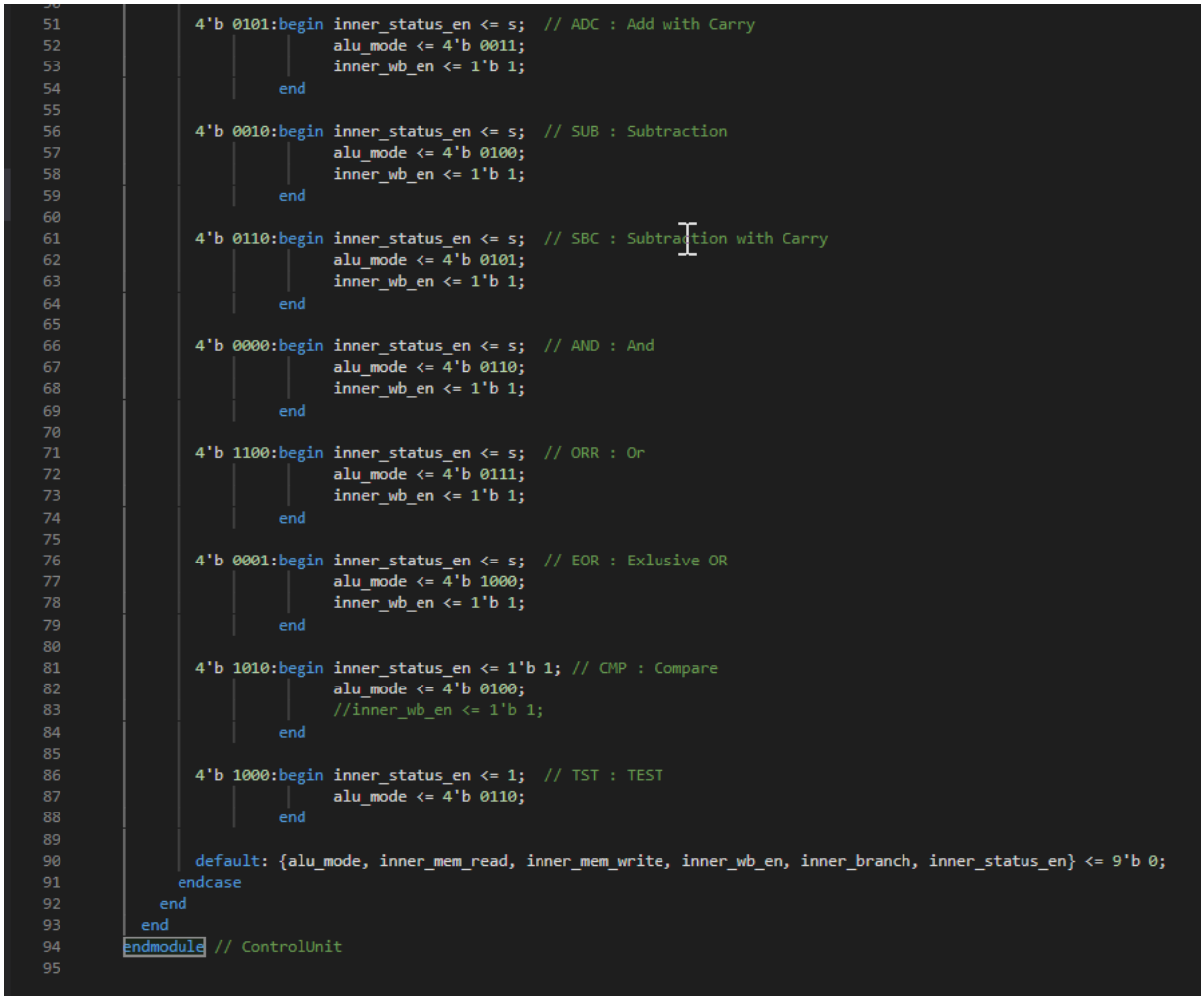
IDSTAGE > IDstage.v

```
module IDSTAGE (input clk, rst, write_back_en, hazard, input[31: 0] pc_in, instruction,
                reg_data_wb, input[3 :0] dest_wb, status, output[31: 0] pc, reg1, reg2,
                output[3: 0] aluCommand, dest, src1, src2, output status_en, mem_read, mem_write,
                wb_en, branch, I, two_src, output[23: 0] b_signed_imm, output[11: 0] shifter_operand);

  assign pc = pc_in;

  assign two_src = ~I | mem_write;

  // Immidate
  assign I = instruction[25];
  assign b_signed_imm = instruction[23: 0];
  assign shifter_operand = instruction[11 :0];


  wire is_ok, cond_result;
  assign is_ok = (~cond_result | hazard) & (|instruction);

  wire c_mem_read, c_mem_write, c_branch, c_wb_en, c_status_en;
  wire[3: 0] c_aluCommand;

  //MUX
  assign wb_en      = is_ok ? 0: c_wb_en;
  assign mem_read   = is_ok ? 0: c_mem_read;
  assign mem_write  = is_ok ? 0: c_mem_write;
  assign branch     = is_ok ? 0: c_branch;
  assign status_en  = is_ok ? 0: c_status_en;
  assign aluCommand = is_ok ? 0: c_aluCommand;

  assign dest = instruction[15: 12];


  ConditionCheck conditionchecker(.condition(instruction[31: 28]),
                                  .status(status),
                                  .out_result(cond_result));

  ControlUnit controler(.mode(instruction[27 :26]), .op_code(instruction[24: 21]), .s(instruction[20]),
                        .alu_command(c_aluCommand), .mem_read(c_mem_read),
                        .mem_write(c_mem_write), .wb_en(c_wb_en), .branch(c_branch),
                        .status_en(c_status_en));


  wire[3: 0] reg_src2;
  assign reg_src2 = c_mem_write? instruction[15: 12]: instruction[3: 0];  // c_mem_write? Rd(STR): Rm;
  RegisterFile registerfile(.clk(clk), .rst(rst), .write_back_en(write_back_en),
                            .src1(instruction[19: 16]), .src2(reg_src2), .dest_wb(dest_wb),
                            .result_wb(reg_data_wb), .reg1(reg1), .reg2(reg2));

  assign src1 = instruction[19: 16];
  assign src2 = reg_src2;

endmodule // IDSTAGE
```

ابندا به توضیح ماژول واحد کنترل می پردازیم. در این بخش سیگنال های کنترلی برای استفاده در قسمت های بعدی پردازنده تولید می شوند. در این ماژول بر اساس mode، op_code و بیت s، بر اساس روند عملیاتی هر دستور، خروجی های alu_command, output mem_read, mem_write, wb_en, branch, status_en تولید می شوند.

IDSTAGE > Controler.v

```v
module ControlUnit (input[1: 0] mode, input[3: 0] op_code, input s,
                    output [3: 0] alu_command, output mem_read, mem_write,
                    wb_en, branch, status_en);
  reg [3: 0] alu_mode;
  assign alu_command = alu_mode;

  reg inner_mem_read, inner_mem_write, inner_wb_en, inner_branch, inner_status_en;

  assign mem_read = inner_mem_read;
  assign mem_write = inner_mem_write;
  assign wb_en = inner_wb_en;
  assign branch = inner_branch;
  assign status_en = inner_status_en;

  always @ ( mode, op_code, s ) begin
    {alu_mode, inner_mem_read, inner_mem_write, inner_wb_en, inner_branch, inner_status_en} <= 9'b 0;
    if (mode == 2'b 10) //B : Branch
      inner_branch <= 1'b 1;
    else if (mode == 2'b 01) begin
      case (s)
          1'b 1:begin inner_status_en <= s; // LDR : Load Register
                      inner_mem_read <= 1'b 1;
                      alu_mode <= 4'b 0010;
                      inner_wb_en <= 1;
                end
          1'b 0:begin inner_status_en <= s; // STR : Store Register
                      inner_mem_write <= 1'b 1;
                      alu_mode <= 4'b 0010;
                end
      endcase
    end
    else begin
      case (op_code)
      // 4'b 0000: {alu_mode, inner_mem_read, inner_mem_write, inner_wb_en, inner_branch, inner_status_en} <= 9'b 0;

        4'b 1101:begin inner_status_en <= s;  // MOV : Move
                       alu_mode <= 4'b 0001;
                       inner_wb_en <= 1;
                end

        4'b 1111:begin inner_status_en <= s;  // MVN : BitWise Not and Move
                       alu_mode <= 4'b 1001;
                       inner_wb_en <= 1;
                end

        4'b 0100:begin inner_status_en <= s;  // ADD : Add
                      alu_mode <= 4'b 0010;
                      inner_wb_en <= 1;
                end

```



```
51          4'b 0101:begin inner_status_en <= s;  // ADC : Add with Carry
52                         alu_mode <= 4'b 0011;
53                         inner_wb_en <= 1'b 1;
54                     end
55
56          4'b 0010:begin inner_status_en <= s;  // SUB : Subtraction
57                         alu_mode <= 4'b 0100;
58                         inner_wb_en <= 1'b 1;
59                     end
60
61          4'b 0110:begin inner_status_en <= s;  // SBC : Subtraction with Carry
62                         alu_mode <= 4'b 0101;
63                         inner_wb_en <= 1'b 1;
64                     end
65
66          4'b 0000:begin inner_status_en <= s;  // AND : And
67                         alu_mode <= 4'b 0110;
68                         inner_wb_en <= 1'b 1;
69                     end
70
71          4'b 1100:begin inner_status_en <= s;  // ORR : Or
72                         alu_mode <= 4'b 0111;
73                         inner_wb_en <= 1'b 1;
74                     end
75
76          4'b 0001:begin inner_status_en <= s;  // EOR : Exlusive OR
77                         alu_mode <= 4'b 1000;
78                         inner_wb_en <= 1'b 1;
79                     end
80
81          4'b 1010:begin inner_status_en <= 1'b 1; // CMP : Compare
82                         alu_mode <= 4'b 0100;
83                         //inner_wb_en <= 1'b 1;
84                     end
85
86          4'b 1000:begin inner_status_en <= 1;  // TST : TEST
87                         alu_mode <= 4'b 0110;
88                     end
89
90          default: {alu_mode, inner_mem_read, inner_mem_write, inner_wb_en, inner_branch, inner_status_en} <= 9'b 0;
91        endcase
92      end
93    end
94  endmodule // ControlUnit
95
```

این پردازنده دارای ۱۶ رجیستر ۳۲ بیتی می باشد که در RegisterFile تعریف شده اند.

IDSTAGE > RegisterFile.v

```
1   module RegisterFile (input clk, rst, write_back_en, input[3: 0] src1, src2, dest_wb,
2                        input[31: 0] result_wb, output[31: 0] reg1, reg2);
3
4     reg [31:0] registers [0: 14];
5
6     assign reg1 = registers[src1];
7     assign reg2 = registers[src2];
8
9     integer i;
10    always @ (negedge clk, posedge rst) begin
11      if (rst)
12        for(i = 0 ; i < 15 ; i = i+1) registers[i] = i;
13
14      if (write_back_en)
15        registers[dest_wb] <= result_wb;
16
17    end
18
19  endmodule // RegisterFile
20
```

هر کدام از این رجیسترها با شماره شان مقدار اولیه دهی شده اند.

برای بررسی شرایط مربوط به condition، ماژول ConditionCheck را بر اساس جدول موجود در دستور کار پیاده سازی می کنیم.

IDSTAGE > ConditionCheck.v

```v
module ConditionCheck (input[3: 0] condition, status, output out_result);

  wire n_flag;
  assign n_flag = status[3];
  wire z_flag;
  assign z_flag = status[2];
  wire c_flag;
  assign c_flag = status[1];
  wire v_flag;
  assign v_flag = status[0];
  reg result;
  assign out_result = result;

  always @ ( condition, status ) begin
  case (condition)
    4'b 0000: result = z_flag ;   // EQ : Equal

    4'b 0001: result = ~z_flag ;  // NE : Not Equal

    4'b 0010: result = c_flag ;   // CS/HS : Carry Set/ Unsigned higher or same

    4'b 0011: result = ~c_flag ;  // CC/LO : Carry clear/ Unsigned lower

    4'b 0100: result = n_flag ;   // MI : Minus/negetive

    4'b 0101: result = ~n_flag ;  // PL: Plus/Posotive or Zero

    4'b 0110: result = v_flag ;   // VS : Overflow

    4'b 0111: result = ~v_flag ;  // VC : no Overflow

    4'b 1000: result = (c_flag & ~z_flag) ;   // HI : Unsigned higher

    4'b 1001: result = (~c_flag & z_flag) ;   // LS : Unsigned Lower or Same

    4'b 1010: result = (n_flag == v_flag) ;   // GE : Signed Grater than or Equal

    4'b 1011: result = (n_flag != v_flag) ;   // LT : Signed Less than

    4'b 1100: result = (~z_flag & (n_flag == v_flag)) ;   // GT : Signed Greater than

    4'b 1101: result = (z_flag & (n_flag != v_flag)) ;   // LE : Signed Less than or Equal

    4'b 1110 , 4'b 1111: result = 1'b 1 ;   // AL : Always

    default: result = 1'b 0;
  endcase
  end

endmodule // ConditionCheck

```

ثبات وضعیت نگهدارنده ی مقادیر n، z، c، v می باشد و مقدار result_out بر آن اساس تعیین می شود.

از آنجاییکه ممکن است مخاطراتی در حین اجرای دستورات رخ دهد، واحد تشخیص مخاطره را جهت یافتن برخی مخاطرات داده ای که بواسطه خواندن پس از نوشتن رخ می دهد را پیاده سازی می کنیم.

این ماژول درصورت یافتن مخاطره، سیگنال hazard_detected را فعال می کند.

حالات مختلف این مخاطره بصورت زیر است:

- برابری src1 با مقصد EXE در صورت یک بودن WB_EN در مرحله اجرا
- برابری src1 با مقصد MEM در صورت یک بودن WB_EN در مرحله حافظه
- برابری src2 با مقصد EXE در صورت یک بودن WB_EN در مرحله اجرا و دو منبعی بودن دستور
- برابری src2 با مقصد MEM در صورت یک بودن WB_EN در مرحله حافظه و دو منبعی بودن دستور

حال ماژول ID2EXE را مشاهده می کنیم:

Register > ID2EXE.v

```
module ID2EXE (input clk, rst, flush, status_en_in, mem_read_in, mem_write_in,
               wb_en_in, branch_in, I_in, input[31: 0] pc_in, reg1_in, reg2_in,
               input[3: 0] aluCommand_in, dest_in, status_in, input[23: 0] b_signed_imm_in,
               input[11: 0] shifter_operand_in, output reg status_en_out, mem_read_out,
               mem_write_out, wb_en_out, branch_out, I_out, output reg[31: 0] pc_out,
               reg1_out, reg2_out, output reg[3: 0] aluCommand_out, dest_out, status_out,
               output reg[23: 0] b_signed_imm_out, output reg[11: 0] shifter_operand_out);

  always @ (posedge clk, posedge rst) begin
    if (rst | flush) begin
      pc_out <= 32'b 0;
      status_en_out <= 0;
      mem_read_out <= 0;
      mem_write_out <= 0;
      wb_en_out <= 0;
      branch_out <= 0;
      I_out <= 0;
      reg1_out <= 32'b 0;
      reg2_out <= 32'b 0;
      aluCommand_out <= 4'b 0;
      dest_out <= 4'b 0;
      status_out <= 4'b 0;
      b_signed_imm_out <= 24'b 0;
      shifter_operand_out <= 12'b 0;
    end
    else begin
      pc_out <= pc_in;
      status_en_out <= status_en_in;
      mem_read_out <= mem_read_in;
      mem_write_out <= mem_write_in;
      wb_en_out <= wb_en_in;
      branch_out <= branch_in;
      I_out <= I_in;
      reg1_out <= reg1_in;
      reg2_out <= reg2_in;
      aluCommand_out <= aluCommand_in;
      dest_out <= dest_in;
      status_out <= status_in;
      b_signed_imm_out <= b_signed_imm_in;
      shifter_operand_out <= shifter_operand_in;
    end
  end
```

مرحله اجرا (Execution)

در این مرحله خروجی دستورات و یا آدرس های مورد نیاز در مراحل دیگر تولید می شود. این ماژول دارای یک alu، Val2Generator و یک رجیستر در مرز انتهایی اش است.

EXESTAGE > EXEstage.v

```
1   module EXEstage (input I, mem_read, mem_write, input[3: 0] aluCommand, status_in,
2                    input[31: 0] pc_in, reg1, reg2, input[11: 0] shifter_operand,
3                    input[23: 0] b_signed_imm, output[31: 0] branch_address, result,
4                    output [3: 0] status_out);
5
6     assign pc = pc_in;
7     wire[31: 0] val2, b_address;
8
9     Val2Generator val2Generator(.I(I), .mem_en(mem_read|mem_write), .shifter(shifter_operand),
10                                .register(reg2), .result(val2));
11
12
13    ALU alu(.command(aluCommand), .status_in(status_in), .operand1(reg1),
14            .operand2(val2), .status(status_out), .result(result));
15
16    SignExtend#(.N(24)) signExtend(.in(b_signed_imm), .out(b_address));
17
18    assign branch_address = pc_in + (b_address << 2) + 4; // Adder in the map!!
19
20  endmodule // EXEstage
21
```

ماژول alu محاسبات مورد نیاز ما را انجام می دهد. این ماژول با توجه به مقدار command، عملیات مربوطه را روی operand ها انجام داده و وضعیت z, c, n, v را در status ذخیره می کند.

EXESTAGE > ALU.v

```v
module ALU (input[3: 0] command, status_in, input signed[31: 0] operand1, operand2,
            output[3: 0] status, output reg[31: 0] result);

  reg c_flag, v_flag;
  wire z_flag, n_flag;
  assign z_flag = ~|result;
  assign n_flag = result[31];

  assign status = {n_flag, z_flag, c_flag, v_flag};

  always @ ( command, operand1, operand2 ) begin
    {result, c_flag, v_flag} = 0;
    case (command)

      4'b 0001: result = operand2 ; // MOV : Move

      4'b 1001: result = ~operand2; // MVN : BitWise Not and Move

      4'b 0010: begin   // ADD
                  {c_flag, result} = operand1 + operand2 ;
                  v_flag = (operand1[31] & operand2[31] & ~result[31]) |
                           (~operand1[31] & ~operand2[31] & result[31]);
                end

      4'b 0011: begin   // ADC : Add with Carry
                  {c_flag, result} = operand1 + operand2 + status_in[1] ;
                  v_flag = (operand1[31] & operand2[31] & ~result[31]) |
                           (~operand1[31] & ~operand2[31] & result[31]);
                 end

      4'b 0100: begin   // SUB : Subtraction
                  result = operand1 - operand2;
                  v_flag = (operand1[31] & operand2[31] & ~result[31]) |
                           (~operand1[31] & ~operand2[31] & result[31]);
                end

      4'b 0101: begin   // SBC : Subtraction with Carry
                  result = operand1 - operand2 - 1;
                  v_flag = (operand1[31] & operand2[31] & ~result[31]) |
                           (~operand1[31] & ~operand2[31] & result[31]);
                end

      4'b 0110: result = operand1 & operand2;     // AND : And
      4'b 0111: result = operand1 | operand2;     // ORR : Or
      4'b 1000: result = operand1 ^ operand2;     // EOR : Exlusive OR

      default: ;
    endcase
  end

endmodule // ALU
```

برای محاسبه مقدار operand دوم از ماژول Val2Generator استفاده می کنیم.

این ماژول بر جسب مقادیر سیگنال های I, mem_en, shifter, register، خروجی result که ۳۲ بیتی است را تولید می کند.

EXESTAGE > Val2Generator.v

```
module Val2Generator (input I, mem_en, input[11: 0] shifter, input[31: 0] register,
                      output[31 :0] result);

  reg [31: 0] im_result, not_im_result;
  wire[31: 0] offset;
  SignExtend#(.N(12)) signExtend(shifter, offset);

  assign result = mem_en ? offset :
                  I ? im_result : not_im_result;

  integer i, rounds;
  always @ (shifter) begin
    im_result = {24'b 0, shifter[7: 0]};
    rounds = shifter[11: 8];
    for( i=0 ; i < rounds; i=i+1)
      im_result = {im_result[1: 0], im_result[31: 2]};
  end

  integer rot_round, j;
  always @ ( shifter, register ) begin
    rot_round = shifter[11: 7];
    not_im_result = register;
    case (shifter[6: 5])
      2'b 00: not_im_result = not_im_result << rot_round; // Logical shift left
      2'b 01: not_im_result = not_im_result >> rot_round; // Logical shift right
      2'b 10: begin // Arithmetic shift right
                //not_im_result[31-rot_round: 0] = not_im_result[31: rot_round];
                for( j=0 ; j < (31-rot_round+1); j=j+1)
                  not_im_result[31-rot_round-j] = not_im_result[31-j];
                for(j=0 ; j < rot_round; j=j+1)
                  not_im_result[31-j] = not_im_result[31];
              end
      2'b 11: begin
                for( j=0 ; j < rot_round; j=j+1)
                  not_im_result = {not_im_result[0], not_im_result[31: 1]};
              end
      default: ;
    endcase

  end

endmodule // ShifterOperand
```

حال معماری رجیستر EXE2MEM را مشاهده می کنیم:

Register > EXE2MEM.v

```
1   module EXE2MEM (input clk, rst, wb_en_in, mem_read_in, mem_write_in,
2                   input[3: 0] dest_in, input[31: 0] result_in, reg2_in,
3                   output reg wb_en_out, mem_read_out, mem_write_out,
4                   output reg[3: 0] dest_out, output reg[31: 0] result_out, reg2_out);
5
6     always @ (posedge clk, posedge rst) begin
7
8       if (rst) begin
9         wb_en_out <= 0;
10        mem_read_out <= 0;
11        mem_write_out <= 0;
12        dest_out <= 0;
13        result_out <= 0;
14        reg2_out <= 0;
15      end
16      else begin
17        wb_en_out <= wb_en_in;
18        mem_read_out <= mem_read_in;
19        mem_write_out <= mem_write_in;
20        dest_out <= dest_in;
21        result_out <= result_in;
22        reg2_out <= reg2_in;
23      end
24
25    end
26
27  endmodule // EXE2MEM
```

مرحله حافظه (Memory)

در این مرحله حافظه ی ما دارای یک خط آدرس است که از آن برای خواندن و نوشتن استفاده می شود. نتیجه حاصل از alu از مرحله قبل به آن وارد می شود.

MEMSTAGE > MEMstage.v

```
1   module MEMstage (input clk, rst, write_en, read_en, input[31: 0] input_data,
2                    address, output[31: 0] data);
3
4     reg [31:0] registers [0: 64*1024]; //2^16 = 64KB
5
6     assign data = read_en ? registers[address] : 0;
7
8     integer i;
9     always @ (posedge clk, posedge rst) begin
10      if (rst)
11        for(i = 0 ; i < 64*1024 ; i = i+1) registers[i] = i;
12      if (write_en)
13        registers[address] <= input_data;
14    end
15
16  endmodule // MEMstage
17
```

این حافظه به حجم ۶۴ کیلوبایت می باشد و داده های آن ۴ بایتی یعنی ۳۲ بیتی هستند.

در این حافظه ابتدا مقادیر با شماره سطر متناظرشان مقدار اولیه دهی شده و سپس در صورت فعال بودن سیگنال نوشتن، داده در آن نوشته شده و در صورت فعال بودن سیگنال خواندن، داده را از حافظه می خوانیم و در خروجی می ریزیم.

حال رجیستر میان این بخش و بخش بازنشانی یا همان WB را مشاهده می کنیم:

Register > MEM2WB.v

```v
module MEM2WB (input clk, rst, wb_en_in, mem_read_in, input[3: 0] dest_in,
               input[31: 0] alu_result_in, mem_data_in, output reg wb_en_out,
               mem_read_out, output reg[3: 0] dest_out, output reg[31: 0] mem_data_out,
               alu_result_out);

  always @ (posedge clk, posedge rst) begin

    if (rst) begin
      wb_en_out <= 0;
      mem_read_out <= 0;
      dest_out <= 0;
      mem_data_out <= 0;
      alu_result_out <= 0;
    end
    else begin
      wb_en_out <= wb_en_in;
      mem_read_out <= mem_read_in;
      dest_out <= dest_in;
      mem_data_out <= mem_data_in;
      alu_result_out <= alu_result_in;
    end

  end

endmodule // MEM2WB

```

مرحله بازنشانی (Write Back)

در این مرحله بر اساس مقدار سیگنال mem_read، خروجی از بین داده تولید شده از خروجی حافظه و خروجی مرحله اجرا انتخاب می شود.

WBSTAGE > WBstage.v

```
1   module WBstage (input mem_read, input[31: 0] alu_result, mem_data,
2                   output[31: 0] result);
3
4   assign result = mem_read ? mem_data : alu_result;
5
6   endmodule // WBstage
7
```

مسیر داده (Data Path)

با کنار هم قرار دادن ماژول های طراحی شده، ماژول اصلی که سازنده پردازنده ARM است را پیاده سازی می کنیم:

dataPath.v

```
module ARM_CPU(input clk ,rst);

  wire status_enable, flush;
  wire [3: 0] status_exe_out, status_reg;
  wire hazard_detected;
  StatusRegister statusRegister(.clk(clk),
                                .rst(rst),
                                .enable(status_enable),
                                .status_in(status_exe_out),
                                .status_out(status_reg));

  wire branch_if_in;
  assign flush = branch_if_in;

  wire [31: 0] instruction_if_out, pc_out_if, branch_address_exe_out;
  IFSTAGE ifStage(.clk(clk),
                  .rst(rst),
                  .freeze(hazard_detected),
                  .branch_track(branch_if_in),
                  .branch_addr(branch_address_exe_out),
                  .instruction(instruction_if_out),
                  .pc(pc_out_if));

  wire [31: 0] pc_in_id, pc_id_out, instruction_id_in;
  IF2ID if2id(.clk(clk),
              .rst(rst),
              .freeze(hazard_detected),
              .flush(flush),
              .pc_in(pc_out_if),
              .instruction_in(instruction_if_out),
              .pc(pc_in_id),
              .instruction(instruction_id_in));

```

```
34      wire wb_en_id_in, status_en_id_out, mem_read_id_out, mem_write_id_out,
35          branch_id_out, I_id_out, two_src;
36      wire [3: 0] dest_id_in, alu_command_id_out, dest_id_out, src1, src2;
37      wire [11: 0] shifter_operand_id_out;
38      wire [23: 0] b_signed_imm_id_out;
39      wire [31: 0] wb_data_id_in, reg1_id_out, reg2_id_out;
40
41      IDSTAGE idSTAGE(.clk(clk),
42                      .rst(rst),
43                      .write_back_en(wb_en_id_in),
44                      .hazard(hazard_detected),
45                      .pc_in(pc_in_id),
46                      .instruction(instruction_id_in),
47                      .reg_data_wb(wb_data_id_in),
48                      .dest_wb(dest_id_in),
49                      .status(status_reg),
50                      .pc(pc_id_out),
51                      .reg1(reg1_id_out),
52                      .reg2(reg2_id_out),
53                      .aluCommand(alu_command_id_out),
54                      .dest(dest_id_out),
55                      .src1(src1),
56                      .src2(src2),
57                      .status_en(status_en_id_out),
58                      .mem_read(mem_read_id_out),
59                      .mem_write(mem_write_id_out),
60                      .wb_en(wb_en_id_out),
61                      .branch(branch_id_out),
62                      .I(I_id_out),
63                      .two_src(two_src),
64                      .b_signed_imm(b_signed_imm_id_out),
65                      .shifter_operand(shifter_operand_id_out));
66
```

dataPath.v

```
67    wire mem_read_exe_in, mem_write_exe_in, wb_en_id_mem;
68    wire [3: 0] alu_command_exe_in, dest_id_mem, status_exe_in;
69    wire [11: 0] shifter_operand_exe_in;
70    wire [23: 0] b_signed_imm_exe_in;
71    wire[31: 0] pc_exe_in, pc_out_exe, reg1_exe_in, reg2_exe_in;
72    ID2EXE id2exe(.clk(clk),
73                  .rst(rst),
74                  .flush(flush),
75                  .status_en_in(status_en_id_out),
76                  .mem_read_in(mem_read_id_out),
77                  .mem_write_in(mem_write_id_out),
78                  .wb_en_in(wb_en_id_out),
79                  .branch_in(branch_id_out),
80                  .I_in(I_id_out),
81                  .pc_in(pc_id_out),
82                  .reg1_in(reg1_id_out),
83                  .reg2_in(reg2_id_out),
84                  .aluCommand_in(alu_command_id_out),
85                  .dest_in(dest_id_out),
86                  .status_in(status_reg),
87                  .b_signed_imm_in(b_signed_imm_id_out),
88                  .shifter_operand_in(shifter_operand_id_out),
89                  .status_en_out(status_enable),
90                  .mem_read_out(mem_read_exe_in),
91                  .mem_write_out(mem_write_exe_in),
92                  .wb_en_out(wb_en_id_mem),
93                  .branch_out(branch_if_in),
94                  .I_out(I_exe_in),
95                  .pc_out(pc_exe_in),
96                  .reg1_out(reg1_exe_in),
97                  .reg2_out(reg2_exe_in),
98                  .aluCommand_out(alu_command_exe_in),
99                  .dest_out(dest_id_mem),
100                 .status_out(status_exe_in),
101                 .b_signed_imm_out(b_signed_imm_exe_in),
102                 .shifter_operand_out(shifter_operand_exe_in));
```

```
106     wire [31: 0] result_exe_out;
107     EXEstage exeSTAGE(.I(I_exe_in),
108                       .mem_read(mem_read_exe_in),
109                       .mem_write(mem_write_exe_in),
110                       .aluCommand(alu_command_exe_in),
111                       .status_in(status_exe_in),
112                       .pc_in(pc_exe_in),
113                       .reg1(reg1_exe_in),
114                       .reg2(reg2_exe_in),
115                       .shifter_operand(shifter_operand_exe_in),
116                       .b_signed_imm(b_signed_imm_exe_in),
117                       .branch_address(branch_address_exe_out),
118                       .result(result_exe_out),
119                       .status_out(status_exe_out));
120
121
122     wire wb_en_mem_wb, mem_read_mem_in, mem_write_mem_in;
123     wire [3: 0] dest_mem_wb;
124     wire[31: 0] result_mem_in, reg2_mem_in;
125     EXE2MEM exe2mem(.clk(clk),
126                     .rst(rst),
127                     .wb_en_in(wb_en_id_mem),
128                     .mem_read_in(mem_read_exe_in),
129                     .mem_write_in(mem_write_exe_in),
130                     .dest_in(dest_id_mem),
131                     .result_in(result_exe_out),
132                     .reg2_in(reg2_exe_in),
133                     .wb_en_out(wb_en_mem_wb),
134                     .mem_read_out(mem_read_mem_in),
135                     .mem_write_out(mem_write_mem_in),
136                     .dest_out(dest_mem_wb),
137                     .result_out(result_mem_in),
138                     .reg2_out(reg2_mem_in));
139
```

```
140     wire [31: 0] mem_data_out;
141     MEMstage memSTAGE(.clk(clk),
142                       .rst(rst),
143                       .write_en(mem_write_mem_in),
144                       .read_en(mem_read_mem_in),
145                       .input_data(reg2_mem_in),
146                       .address(result_mem_in),
147                       .data(mem_data_out));
148
149
150     wire mem_read_wb_in;
151     wire[31: 0] mem_data_wb_in, alu_result_wb_in;
152     MEM2WB mem2wb(.clk(clk),
153                   .rst(rst),
154                   .wb_en_in(wb_en_mem_wb),
155                   .mem_read_in(mem_read_mem_in),
156                   .dest_in(dest_mem_wb),
157                   .alu_result_in(result_mem_in),
158                   .mem_data_in(mem_data_out),
159                   .wb_en_out(wb_en_id_in),
160                   .mem_read_out(mem_read_wb_in),
161                   .dest_out(dest_id_in),
162                   .mem_data_out(mem_data_wb_in),
163                   .alu_result_out(alu_result_wb_in));
164
165     WBstage wbSTAGE(.mem_read(mem_read_wb_in),
166                     .alu_result(alu_result_wb_in),
167                     .mem_data(mem_data_wb_in),
168                     .result(wb_data_id_in));
169
```

```
169
170     HazardDetectionUnit hazard_detection_unit(.src1(src1),
171                                               .src2(src2),
172                                               .Exe_Dest(dest_id_mem),
173                                               .Mem_Dest(dest_mem_wb),
174                                               .Mem_WB_EN(wb_en_mem_wb),
175                                               .Exe_WB_EN(wb_en_id_mem),
176                                               .two_src(two_src),
177                                               .hazard_detected(hazard_detected));
178     endmodule
179
```

نتایج

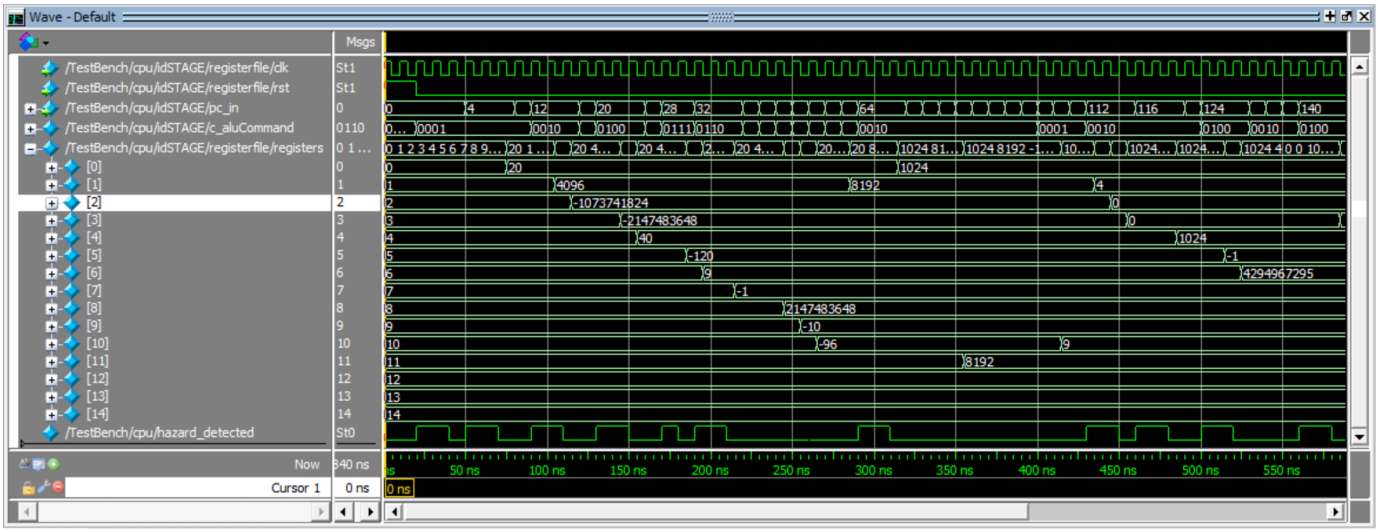
علاوه بر آن که در طول پیاده سازی، هرکدام از مراحل معماری را جداگانه تست کردیم، حال یک TestBench کلی جهت تست کامل پردازنده خود می نویسیم:

testBech.v

```
1   module TestBench ();
2     reg clk, rst;
3     ARM_CPU cpu(clk, rst);
4     always #5 clk=~clk;
5     always begin
6       rst = 1; clk=1;#20
7       rst= 0; #820
8       $stop;
9     end
10  endmodule // TestBench
11
```

در تست در هر ۵ نانو ثانیه کلاک زده و بعد از ۲۰ نانو ثانیه reset کردن پردازنده شروع به فعالیت می کند.

خروجی به صورت زیر خواهد بود:



در ۶۵۰ نانو ثانیه اجرای پردازنده، ۱۳۰ کلاک زده شده و ۴۷ دستور اجرا شدند که مقدار CPI برابر

$$CPI = \frac{130}{47} = 2.76$$

می باشد.