به نام خدا

آزمایشگاه معماری کامپیوتر (CALab)

گزارش جلسه سوم

ایمان رسولی پرتو 810199425 & پارسا حداد منفرد 810198380

EXE Stage, WB Stage

ALU ←

ماژول ALU رو به شکل behavioral طراحی میکنیم؛ همچنین این ماژول بیت های Z, N, V, C رو برای Status register تولید میکنه. با توجه به جدول command های ALU توصیف زیر رو خواهیم داشت:

```
module ALU(in1, in2, cin, opcode, result, neg, zer, cout, ov);
  input [31:0] in1, in2;
  input cin;
  input [3:0] opcode;
  output reg [31:0] result;
  output neg, zer, ov;
  output reg cout;

  wire is_add_sub, ov_val;

  always @(in1, in2, opcode)begin
    result = 32'b0;
    case(opcode)
      4'b0001: result = in2;
      4'b1001: result = ~in2;
      4'b0010: {cout, result} = in1 + in2;
      4'b0011: {cout, result} = in1 + in2 + cin;
      4'b0100: {cout, result} = in1 - in2;
      4'b0101: {cout, result} = in1 - in2 - ~cin;
      4'b0110: result = in1 & in2;
      4'b0111: result = in1 | in2;
      4'b1000: result = in1 ^ in2;
      default: result = 32'bz;
    endcase
  end

  assign neg = result[31];
  assign zer = ~(| result);

  assign is_add_sub = opcode == 4'b0010 | opcode == 4'b0011 | opcode == 4'b0100 |opcode == 4'b0101;
  assign ov_val = (in1[31] & in2[31] & ~result[31]) | (~in1[31] & ~in2[31] & result[31]);
  assign ov = (is_add_sub) ? ov_val : 1'b0;

endmodule
```

Fig1. ALU Verilog description

Val2 Generator ←

ماژول val2 generator به منظور تولید ورودی دوم ALU به کار برده میشود بنا بر ساختار دستورات پردازنده ARM مقدار ورودی دوم ALU می تواند یکی از مقادیر 32 بیت عدد فوری (32 bit immediate) یا شیفت فوری (immediate shift) باشد با توجه به دستور کار نحوه تولید این مقادیر توسط val2 generator در کد وریلاگ زیر نوشته شده است

این ماژول با دستور shift_operand و سیگنال های کنترلی imm و control_input و دریافت مقدار ValRm خروجی Val2 را تولید می کند

```verilog
module Val2Gen (shift_operand, imm, mem_read, mem_write, Val_Rm, val2);
  input[11:0] shift_operand;
  input imm, mem_read, mem_write;
  input [31:0] Val_Rm;
  output [31:0] val2;

  wire [5:0] rotate_imm2;
  wire [7:0] immed_8;
  wire [31:0] container32;
  wire [31:0] shifted_immed;
  wire [31:0] shifted_reg2;
  wire signed [31:0] arith_right_shifted_reg2;
  wire [4:0] shift_imm;
  wire [1:0] shift;

  assign val2 = (mem_read | mem_write) ? shift_operand : // STR, LDR -> 12bit offset
                (imm) ? shifted_immed : shifted_reg2; // 32-bit immediate(container rotation) vs immeditate shift(shifted Rm)


  // 32-bit immediate(container rotation)
  assign rotate_imm2 = shift_operand[11:8] << 1;
  assign immed_8 = shift_operand[7:0];
  assign container32 = {24'b0, immed_8};
  assign shifted_immed = (container32 >> rotate_imm2) | (container32 << 32 - rotate_imm2);

  //immeditate shift(shifted Rm)
  assign shift_imm = shift_operand[11:7];
  assign shift = shift_operand[6:5];
  //this id done here, becuase >>> not work in conditional assignment
  assign arith_right_shifted_reg2 = $signed(Val_Rm) >>> shift_imm;
  assign shifted_reg2 = (shift == 2'b00) ? Val_Rm << shift_imm: //logical shift left
                        (shift == 2'b01) ? Val_Rm >> shift_imm: //logical shift right
                        (shift == 2'b10) ? arith_right_shifted_reg2://arithmatic shift right
                        (shift == 2'b11) ? (Val_Rm >> shift_imm) | (Val_Rm << (32 - shift_imm)): 31'bx;

endmodule
```

Fig2. Val2 generator Verilog description

← وصل کردن module های ساخته شده و طراحی بخش EXE stage

ماژول های ساخته شده رو در بخش EXE توصیف میکنیم؛ همچنین آدرس Branch رو در این بخش محاسبه میکنیم؛ برای محاسبه این آدرس 24 بیت signed_imm24 رو ابتدا sign extend کرده و سپس 2 بیت به چپ شیفت میدیم.

```verilog
module EXE_stage(clk, EXE_CMD, MEM_R_EN, MEM_W_EN, PC, Val_Rn, Val_Rm, imm,
                 Shift_operand, Signed_EX_imm24, SR, ALU_Res, Branch_Address,
                 status);
  input clk;
  input [3:0] EXE_CMD;
  input MEM_R_EN, MEM_W_EN;
  input [31:0] PC;
  input [31:0] Val_Rn, Val_Rm;
  input imm;
  input [11:0] Shift_operand;
  input [23:0] Signed_EX_imm24;
  input [3:0] SR;

  output [31:0] ALU_Res, Branch_Address;
  output [3:0] status;

  wire [31:0] Val1, Val2;
  wire cin, neg, zer, cout, ov;
  wire [3:0] opcode;

  assign Val1 = Val_Rn;
  assign cin = SR[1];
  assign opcode = EXE_CMD;

  wire control_input;
  assign control_input = MEM_R_EN || MEM_W_EN;

  Val2_Gen V2G(Shift_operand, imm, Val_Rm, control_input, Val2);
  ALU alu(Val1, Val2, cin, opcode, ALU_Res, neg, zer, cout, ov);

  assign status = {neg, zer, cout, ov};
  assign Branch_Address = PC + {{6{Signed_EX_imm24[23]}}, Signed_EX_imm24, 2'b00};

endmodule
```

Fig3. EXE stage

← طراحی EXE stage register

مشابه ID stage register سیگنال هایی که به این بخش میان رو رجیستر میکنیم تا در دسترس MEM stage قرار بگیرن.

```
113  module EXE_Reg(clk, rst, WB_EN_IN, MEM_R_EN_IN, MEM_W_EN_IN, ALU_result_in,
114                 ST_val_in, Dest_in, WB_EN, MEM_R_EN, MEM_W_EN, ALU_result,
115                    ST_val, Dest);
116    input clk, rst, WB_EN_IN, MEM_R_EN_IN, MEM_W_EN_IN;
117    input [31:0] ALU_result_in, ST_val_in;
118    input [3:0] Dest_in;
119
120    output reg WB_EN, MEM_R_EN, MEM_W_EN;
121    output reg [31:0] ALU_result, ST_val;
122    output reg [3:0] Dest;
123
124    always@(posedge clk, posedge rst)begin
125      if(rst) begin
126        WB_EN <= 1'b0;
127        MEM_R_EN <= 1'b0;
128        MEM_W_EN <= 1'b0;
129        ALU_result <= 32'b0;
130        ST_val <= 32'b0;
131        Dest <= 4'b0;
132      end
133      else begin
134        WB_EN <= WB_EN_IN;
135        MEM_R_EN <= MEM_R_EN_IN;
136        MEM_W_EN <= MEM_W_EN_IN;
137        ALU_result <= ALU_result_in;
138        ST_val <= ST_val_in;
139        Dest <= Dest_in;
140      end
141    end
142
143  endmodule
```

Fig4. EXE stage register

← طراحی WB stage

این بخش شامل یک Multiplexer هست که ورودی Register file write رو مشخص میکنه (بر اساس سیگنال mem write). همچنین این بخش نیازی به register نداره چون آخرین pipeline stage خواهد بود.

```
69   module WB_stage(ALU_result, MEM_result, MEM_R_en, WB_Value);
70     input [31:0] ALU_result, MEM_result;
71     input MEM_R_en;
72     output [31:0] WB_Value;
73
74     assign WB_Value = (MEM_R_en) ? ALU_result : MEM_result;
75
```

Fig5. WB stage

← متصل کردن بخش های IF, ID, EXE و تست بخشی از benchmark

سه بخش IF, ID, EXE رو به همراه رجیسترهاشون به هم متصل میکنیم.

```
module ARM_EXE(clk, rst, WB_Value, WB_WB_EN, WB_DEST, SR,WB_EN, MEM_R_EN, MEM_W_EN,
        ALU_result, ST_val, Dest);
    input clk, rst;
    input [31:0] WB_Value;
    input WB_WB_EN;
    input [3:0] WB_DEST;
    input [3:0] SR;
    output reg WB_EN, MEM_R_EN, MEM_W_EN;
    output reg [31:0] ALU_result, ST_val;
    output reg [3:0] Dest;

    //IF
    wire freeze, Branch_taken;
    wire [31:0] BranchAddr, IF_PC, IF_Instruction;

    assign freeze = 1'b0;
    assign Brach_taken = 1'b0;
    assign BranchAddr = 32'b0;

    wire flush;
    wire hazard;
    wire [31:0] IF_Reg_PC, IF_Reg_Instruction;

    assign flush = 1'b0;
    assign hazard = 1'b0;
    //ID
    wire ID_WB_EN_out, ID_MEM_R_EN_out, ID_MEM_W_EN_out, ID_B_out, ID_S_out;
    wire [3:0] ID_EXE_CMD_out;
    wire [31:0] ID_Val_Rn_out, ID_Val_Rm_out;
    wire ID_imm_out;
    wire [11:0] ID_Shift_operand_out;
    wire [23:0] ID_Signed_imm_24_out;
    wire [3:0] ID_Dest_out;
    wire [3:0] src1, src2;
    wire Two_src;
    //ID Reg
    wire IDreg_WB_EN_out, IDreg_MEM_R_EN_out, IDreg_MEM_W_EN_out, B, S;
    wire [3:0] EXE_CMD;
    wire [31:0] Val_Rn, Val_Rm;
    wire imm;
    wire [11:0] Shift_operand;
    wire [23:0] Signed_imm_24;
    wire [3:0] IDreg_Dest;
```

```
    //EXE Reg
    //wire [31:0] ALU_Result;
    //wire [31:0] ST_val;

    IF_stage IF(clk, rst, freeze, Branch_taken, BranchAddr, IF_PC, IF_Instruction);

    IF_stage_reg IF_Reg(clk, rst, freeze, flush, IF_PC, IF_Instruction, IF_Reg_PC, IF_Reg_Instruction);

    ID_stage ID(clk, rst, IF_Reg_Instruction, WB_Value, WB_WB_EN, WB_DEST, hazard, SR,
            ID_WB_EN_out, ID_MEM_R_EN_out, ID_MEM_W_EN_out, ID_B_out, ID_S_out,
            ID_EXE_CMD_out, ID_Val_Rn_out, ID_Val_Rm_out, ID_imm_out, ID_Shift_operand_out,
            ID_Signed_imm_24_out, ID_Dest_out, src1, src2, Two_src);

    ID_stage_reg ID_Reg(clk, rst, flush, ID_WB_EN_out, ID_MEM_R_EN_out, ID_MEM_W_EN_out,
                ID_B_out, ID_S_out, ID_EXE_CMD_out, IF_Reg_PC, ID_Val_Rn_out,
                ID_Val_Rm_out, ID_imm_out, ID_Shift_operand_out, ID_Signed_imm_24_out,
                ID_Dest_out, IDreg_WB_EN_out, IDreg_MEM_R_EN_out, IDreg_MEM_W_EN_out,
                B, S, EXE_CMD, PC, Val_Rn, Val_Rm, imm, Shift_operand, Signed_imm_24, IDreg_Dest);

    EXE_stage EXE(clk, EXE_CMD, IDreg_MEM_R_EN_out, IDreg_MEM_W_EN_out, PC, Val_Rn, Val_Rm, imm,
          Shift_operand, Signed_imm24, SR, ALU_Res, BranchAddr, status);

    EXE_stage_reg EXE_Reg(clk, rst, IDreg_WB_EN_out, IDreg_MEM_R_EN_out, IDreg_MEM_W_EN_out, ALU_Res,
                  Val_Rm, IDreg_Dest, WB_EN, MEM_R_EN, MEM_W_EN, ALU_result,
                      ST_val, Dest);
endmodule
```

Fig6. Connect IF, ID, EXE with registers

مشابه بخش ID، سیگنال‌هایی که هنوز پیاده‌سازی نشدن رو inactive میکنیم یا به عنوان input در نظر میگیریم.

18 دستور ابتدایی benchmark رو در حافظه دستورات قرار میدیم و با نوشتن یک Testbench عملکرد مدار رو verify میکنیم.

```
218  module ARM_Tb();
219    reg clk, rst;
220    reg [3:0] SR;
221
222    wire [31:0] ALU_result, ST_val;
223
224    ARM_EXE ARM(clk, rst, SR, ALU_result, ST_val);
225
226    initial begin
227      clk = 1'b0;
228      rst = 1'b1;
229      SR = 4'b0;
230    end
231
232    always #5 clk = ~clk;
233
234    initial begin
235      #12 rst = 1'b0;
236      //#27 rst = 1'b0;
237      #500 $stop;
238    end
239  endmodule
```

Fig7. ARM testbench

چون ماژول Register status هنوز پیاده سازی نشده، سیگنال SR رو به شکل input به مدار میدیم. با شبیه سازی مدار داریم:

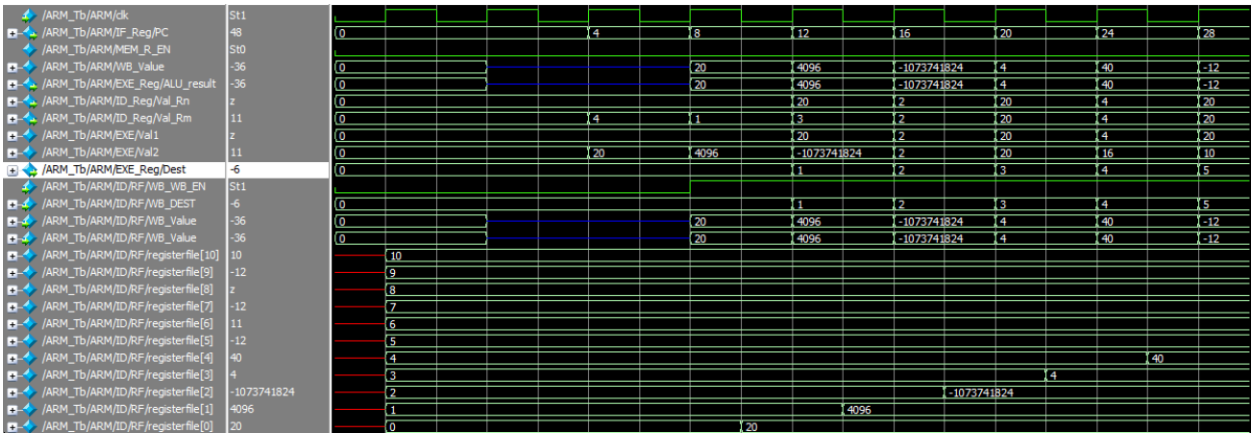

Fig7. Test benchmark on ARM prosseccor

دستور اول در benchmark مقدار R0 رو 0 قرار میده. دستور بعدی R1 رو برابر 4096 قرار میده که به درستی انجام شده. مقدار R2 درسته اما میبینیم که مقدار R3 صحیح نیست. دلیل این امر نبود Data forwarding هست که مقدار R2 هنوز به روز نشده و R3 مقدار قبلی R2 که برابر با عدد رجیستر یعنی 2 هست رو گرفته.