به نام خدا

آزمایشگاه معماری کامپیوتر (CALab)

گزارش جلسه چهارم

ایمان رسولی پرتو 810199425 & پارسا حداد منفرد 810198380

MEM stage, Hazard Unit

MEM stage ←

بخش Memory شامل یک حافظه 32 بیتی با 64 خانه هست که آدرس رو از ALU میگیره و براساس اینکه خوندن یا نوشتن رو انجام بده، عملیات مورد نظر رو پیاده سازی میکنه. با توجه به اینکه آدرس حافظه از 1024 شروع میشه، برای synchronize کردن این موضوع آدرس ورودی مموری رو از 1024 کسر میکنیم. برای حل موضوع align که فقط آدرس های مضرب 4 خونده میشن، آدرس رو تقسیم به 4 میکنیم (دو بیت شیفت به چپ).

```
22  module MEM_stage(clk, MEM_read, MEM_write, address, data, MEM_result);
23    input clk, MEM_read, MEM_write;
24    input [31:0] address, data;
25    output [31:0] MEM_result;
26
27    reg [31:0] memory [0:64];
28
29    always@(posedge clk)begin
30      if(MEM_write)
31        memory[(address - 32'd1024) >> 2] = data;
32    end
33
34    assign MEM_result = (MEM_read)? memory[(address - 32'd1024) >> 2] : 32'b0;
35
36  endmodule
```

Fig1. Memory Verilog description

Hazard Unit ←

واحد تشخیص hazard، وابستگی های داده ای رو شناسایی میکنه و در صورت وجود، pipeline رو stall میکنه (بر اساس Destination ای که hazard اتفاق میفته تعداد سیکل های stall متفاوته). این ماژول به این شکل کار میکنه که اگر مثلا در بخش ID یکی از رجیسترهای رجیستر فایل (مثلا R3) بخواهد مقدار آپدیت شده یک رجیستر فایل دیگه (مثلا R2) رو بگیره و مقدار R2 هنوز آپدیت نشده باشه (در مرحله EXE یا MEM باشه) hazard unit هازارد رو تشخیص میده و سیگنال flush رو active میکنه.

اگر دستور move باشه، src1 همواره صفره و hazard unit به اشتباه هازارد تشخیص میده. برای حل این موضوع کنترلر بررسی میکنه که اگر opcode دستور move بود به این ماژول اطلاع میده.

```
module hazard_Detection_Unit(src1, src2, Exe_Dest, Exe_WB_EN, move, Mem_Dest,
                             Mem_WB_EN, Two_src, hazard_Detected);
  input [3:0] src1, src2;
  input [3:0] Exe_Dest;
  input Exe_WB_EN, move;
  input [3:0] Mem_Dest;
  input Mem_WB_EN;
  input Two_src;
  output reg hazard_Detected;

  always@(src1, src2, Exe_Dest, Exe_WB_EN, Mem_Dest, Mem_WB_EN, Two_src)begin
    hazard_Detected = 1'b0;
    if(~move)begin
      if(Two_src)
        if((Exe_Dest ==src1 && Exe_WB_EN) || (Mem_Dest == src2 && Mem_WB_EN == 1'b1))
      hazard_Detected = 1'b1;
      else
        if((Exe_Dest == src1 && Exe_WB_EN) || (Mem_Dest == src1 && Mem_WB_EN))
      hazard_Detected = 1'b1;
    end
  end
endmodule
```

Fig2. Hazard unit Verilog description

Status Register ←

** این ماژول در جلسه پیاده سازی ID به اشتباه پیاده سازی نشد و مطالبش اینجا مطرح میشه.

این ماژول status خروجی های وضعیتی ALU رو در صورت اینکه دستور اعلام به روز رسانی وضعیت کنه، در خودش ذخیره میکنه و به condition check میده؛ همچنین این رجیستر Cin ماژول ALU رو هم به روز میکنه و بهش میده.

```
module status_register(clk, rst, S_in, neg_in, zer_in, cin, ov_in, SR);
  input clk, rst;
  input S_in, neg_in, zer_in, cin, ov_in;
  output reg [3:0] SR;

  reg neg, zer, ov, cout;
  always@(negedge clk)begin
    if(rst)begin
      neg <= 1'b0;
      zer <= 1'b0;
      ov <= 1'b0;
      cout <= 1'b0;
    end
    else if(S_in)begin
      neg <= neg_in;
      zer <= zer_in;
      ov <= ov_in;
      cout <= cin;
    end
  end

  assign SR = {neg, zer, cout, ov};

endmodule
```

Fig3. Status register Verilog description

← وصل کردن تمام بخش ها و تست Benchmark روی پردازنده

با اتمام ساخت ماژول ها، در Top module تمام قسمت ها رو بهم متصل میکنیم و برنامه رو روی پردازنده تست میکنیم.

```
module ARM(clk, rst);
  input clk, rst;
  wire [31:0] WB_Value;
  wire WB_WB_EN;
  wire [3:0] SR;
  wire [31:0] ALU_result, Val_Rm_EXE;
  wire [3:0] WB_DEST;

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

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

  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;

  wire [31:0] ALU_Res;
  wire [3:0] status;
  wire cin;

  wire [31:0] address;
  assign address = ALU_result;

  wire WB_EN_MEM, MEM_R_EN_MEM;
  wire [31:0] ALU_res_MEM, Mem_read_val_MEM;
  wire [3:0] Dest_MEM;

  wire [31:0] MEM_result;

  wire S_reg_out;
  wire Exe_WB_EN, Mem_WB_EN;
  wire [3:0] Exe_Dest, Mem_Dest;
  assign Exe_Dest = IDreg_Dest;
  assign Exe_WB_EN = IDreg_WB_EN_out;
  assign Mem_Dest = WB_DEST;
  assign Mem_WB_EN = WB_WB_EN;
  wire hazard_Detected;
  wire [31:0] PC;
  wire move;

  assign flush = B;

  IF_stage IF(clk, rst, hazard_Detected, B, BranchAddr, IF_PC, IF_Instruction);

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

  ID_stage ID(clk, rst, IF_Reg_Instruction, WB_Value, WB_EN_MEM, Dest_MEM, hazard_Detected, 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, move);

  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, SR[1], 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, cin, 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, cin,
        Shift_operand, Signed_imm_24, 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_WB_EN, MEM_R_EN, MEM_W_EN, ALU_result,
                    Val_Rm_EXE, WB_DEST);

  MEM_stage MEM(clk, MEM_R_EN, MEM_W_EN, ALU_result, Val_Rm_EXE, MEM_result);

  MEM_stage_reg MEM_Reg(clk, rst, WB_WB_EN, MEM_R_EN, ALU_result, MEM_result, WB_DEST, WB_EN_MEM,
              MEM_R_EN_MEM, ALU_res_MEM, Mem_read_val_MEM, Dest_MEM);

  WB_stage WB(ALU_res_MEM, Mem_read_val_MEM, MEM_R_EN_MEM, WB_Value);

  status_register status_reg(clk, rst, S, status[3], status[2], status[1], status[0], S_reg_out, SR);

  hazard_Detection_Unit hazard_unit(src1, src2, Exe_Dest, Exe_WB_EN, move, Mem_Dest,
                        Mem_WB_EN, Two_src, hazard_Detected);

endmodule
```

Fig4. ARM processor top module

با تست کردن برنامه روی پردازنده خواهیم داشت:

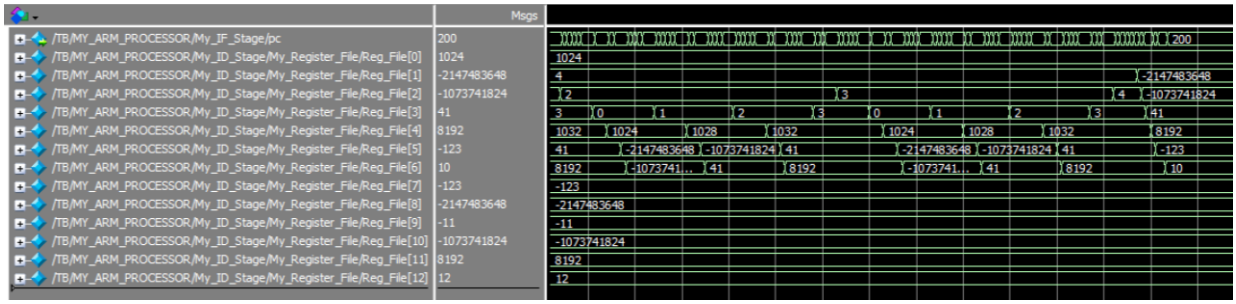

Fig5. Test benchmark on ARM

با توجه به شکل بالا میبینیم که اعداد sort شدن و همچنین مقدار رجیستر R6 برابر 10 شده.

در نهایت با سنتز مجموعه و دیدن سیگنال های Signal Tab خواهیم داشت:

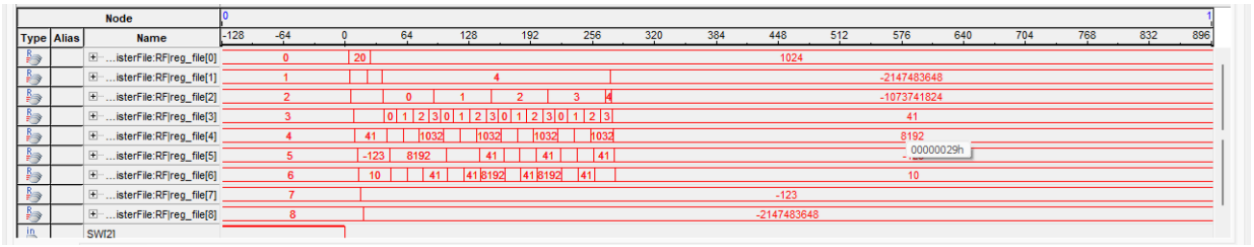

Fig6. ARM Synthesis result – Signal Tab

مشاهده میکنیم که مقادیر رجیسترها صحیحه و اعداد sort شدن.