به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

آزمایشگاه معماری کامپیوتر

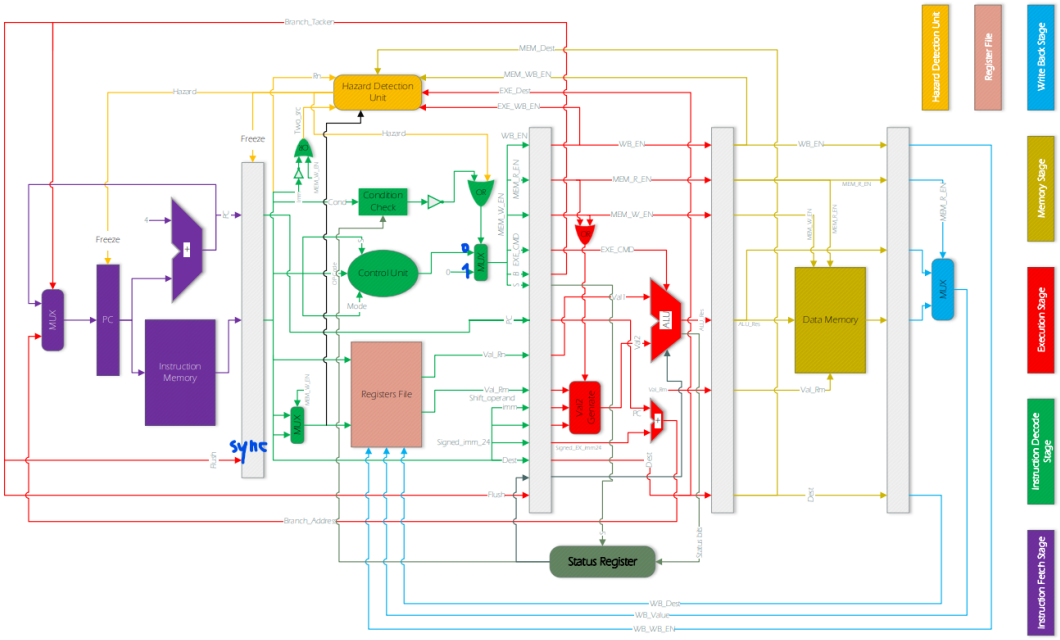
استاد رستگار

نام و نام خانوادگی	سیده دیبا روانشید شیرازی، ثمر نیک فرجاد
شماره دانشجویی	810199431 ، 810199508
تاریخ ارسال گزارش	2 تیرماه 1403

فهرست گزارش سوالات

مقدمه

توضیحات کلی پردازنده



در این آزمایشگاه ما باید یک پردازنده ARM ساده که دارای 13 دستور العمل اصلی است، پیاده سازی کنیم. ARMنوعی از معماری پردازنده های کامپیوتری است. طبق شکل بالا دارای چندین بخش است که در هر جلسه یک بخش از این پردازنده را پیاده سازی کردیم.

مشخصات پردازنده

نام پردازنده ی ما : ARM968E-S میباشد که دارای مشخصات زیر میباشد:

1. پهنای خط داده : 32 بیت
2. تعداد مراحل خط لوله : 5
3. تعداد دستورات : 13 دستور
4. میزان تاخیر انشعاب : 2 مرحله
5. 16 ثبات همه منظوره
6. آدرس دهی بر حسب بایت و فضای آدرس دستورات و داده تفکیک شده می باشد.
7. تمامی پرش ها از نوع مجلی تعریف شده است.

8. قابلیت تشخیص هازارد داده ای و واحد ارسال به جلو نداریم ولی به صورت دستی به آن اضافه میکنیم.

در تمامی بخش ها پیاده سازی در زبان وریلاگ میباشد و در نهایت پس از شبیه سازی در نرم افزار ModelSim، با استفاده از نرم افزار Quartus سنتز کردیم و روی FPGA بردیم. با سیگنال تب هم نتایج خود را چک کردیم.

جلسه اول : ایجاد ماژول ها

در ابتدا باید 5 مرحله خط لوله پردازنده را به همراه مرحله واکشی به صورت کامل پیاده سازی کنیم.

هر بخش شامل یک ماژول برای عملیات ها و یک ماژول رجیستر بعد از آن میباشد.

ورودی ها و خروجی ها فعلا در حد کلاک و ریست و PC_in و PC_out هستند.

```
(
    input clk,
    input rst,
    input [31:0] PC_in,
    output[31:0] PC,
);
```

```
module IF_Stage (
    input clk,rst,freeze,Branch_taken ,
    input[31:0] BranchAddr,
    output [31:0] PC,Instruction
);
```

```
module IF_Stage_Reg (
    input clk,rst,freeze,flush,
    input[31:0]PC_in, Instruction_in,
    output reg [31:0] PC,Instruction
);
```

رجیستر PC مانند یک شمارنده عمل می کند، که از صفر شروع به شمارش می کند، تا به بیشینه مقدار خود برسد. اگر کلید Reset زده شود، مقدار PC صفر میشود. همچنین در صورت 1 بودن freeze مقدار ورودی در رجیستر بارگذاری نمی شود.

در هر ماژول صرفا باید مینوشتیم assign PC=PC_in و در ماژول رجیستر باید با دیدن هر کلاک ورودی را به خروجی بدهیم.

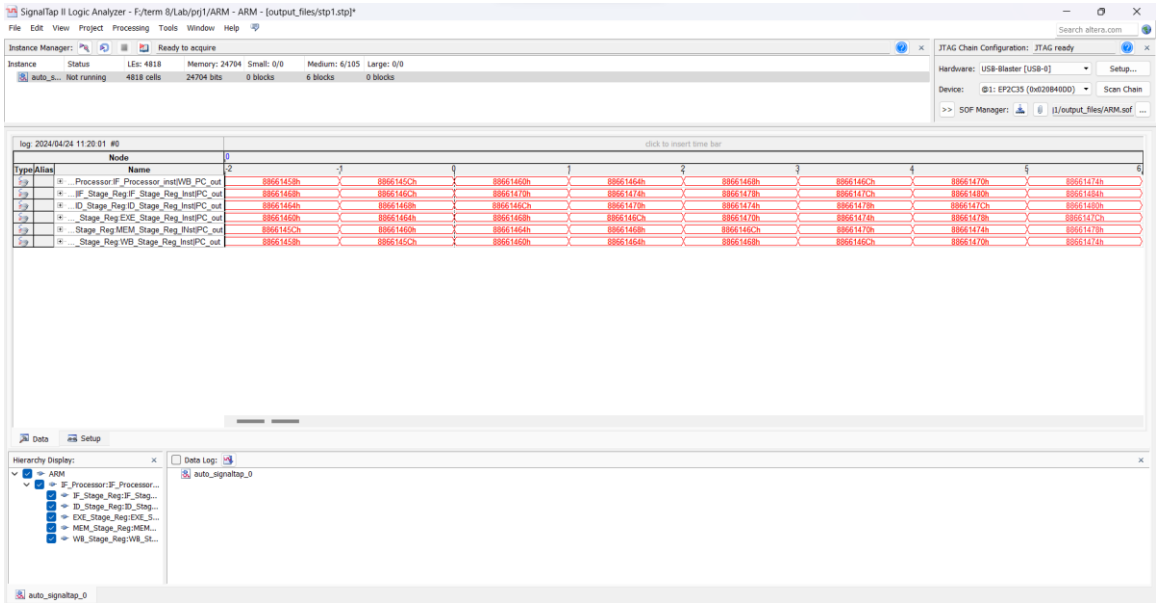
در نهایت ماژول IF به صورت زیر در می آید که از آن به عنوان تاپ لول نمونه گیری میکنیم.

F:/term 8/Lab/final_project_with forwarding/IF_Processor.v - Default

```
Ln#
 1  module IF_Processor(
 2    input clk, rst,
 3    output[31:0] WB_PC_out
 4  );
 5
 6  wire freeze, Branch_taken, flush;
 7  wire[31:0] BranchAddr, PC, Instruction, IF_PC_out, IF_Instruction_out;
 8  wire[31:0] ID_PC, ID_PC_out, EXE_PC, EXE_PC_out, MEM_PC, MEM_PC_out, WB_PC;
 9
10  assign {freeze, Branch_taken, BranchAddr, flush} = 35'd0;
11
12  IF_Stage IF_Stage_Inst(
13    clk, rst, freeze, Branch_taken,
14    BranchAddr,
15    PC, Instruction
16  );
17
18  IF_Stage_Reg IF_Stage_Reg_Inst(
19    clk, rst, freeze, flush,
20    PC, Instruction,
21    IF_PC_out, IF_Instruction_out
22  );
23
24  ID_Stage ID_Stage_Inst (
25    clk,
26    rst,
27    IF_PC_out,
28    ID_PC
29  );
30
31  ID_Stage_Reg ID_Stage_Reg_Inst(
32    clk, rst,
33    ID_PC,
34    ID_PC_out
35  );
36
37  EXE_Stage EXE_Stage_Inst (
38    clk,
39    rst,
40    ID_PC_out,
41    EXE_PC
42  );
43
44  EXE_Stage_Reg EXE_Stage_Reg_Inst(
45    clk, rst,
46    EXE_PC,
47    EXE_PC_out
48  );
49  MEM_Stage MEM_Stage_Inst (
50    clk,
51    rst,
52    EXE_PC_out,
53    MEM_PC
54  );
55  MEM_Stage_Reg MEM_Stage_Reg_INst(
56    clk, rst,
57    MEM_PC,
58    MEM_PC_out
59  );
60  WB_Stage WB_Stage_Inst (
61    clk,
62    rst,
63    MEM_PC_out,
64    WB_PC
65  );
66  WB_Stage_Reg WB_Stage_Reg_Inst(
67    clk, rst,
68    WB_PC,
69    WB_PC_out
70  );
71  endmodule
```

کلاک را به مقدار 50 هرتز و یک سوییچ را به عنوان ریست انتخاب کردیم. Sw[0]

حرکت موج گونه ی PC را در سیگنال تب مشاهده میکنیم:

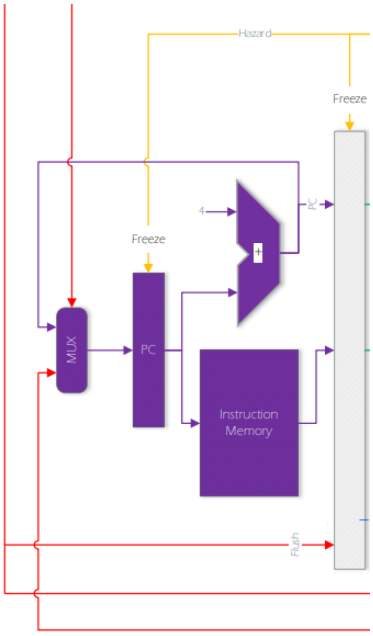


ارور ها

در این بخش در هنگام کار کردن با سیگنال تب به مشکلاتی خوردیم. دلیل آن این بود که اولین بار در اجرای مراحل سیگنال تب اشتباه کردیم و وقتی مجدد یک سیگنال تب جدید ایجاد کردیم به دلیل تشابه های اسمی با سیگنال تب قبلی مشکلاتی بوجود آمد. مجبور شدیم یک پروژه جدید در کوارتس ایجاد کنیم و همه مراحل را از اول انجام دهیم.

بخش اول :IF Stage

در این بخش هدف اصلی ما واکشی دستور العمل ها میباشد.



با توجه به شکل بالا ما نیاز به یک ثبات برای نگه داری شماره ی برنامه یا همان PC داریم تا بتوانیم دستورات را دانه به دانه اجرا کنیم.

همچنین به یک Instruction Memory برای نگه داری دستور العمل ها نیاز داریم.

کد ها

نیاز به یک اددر داشتیم که به صورت جمع ورودی(PC) با عدد 4 آن را میسازیم.

خروجی این ماژول به عنوان PC هم از کل استیج خارج میشود و هم مجدد به مالتیپلکسر وارد میشود.

مالتیپلکسر با توجه به مقدار Branch taken آدرس PC+4 یا آدرس پرش را خروجی میدهد.

PC خودش یک رجیستر است که میتوانیم مقدار آن را فریز و یا ریست کنیم. خروجی مالتیپلکسر به آن وارد میشود.

در نهایت آدرس PC وارد موری میشود و در موری با توجه به PC یک سری دستور العمل تعبیه شده است که توسط خود شما به ما داده شده است. دستور العمل مد نظر خوانده شده و به صورت خروجی کل استیج به بیرون داده میشود.

در نهایت ترکیب همه ی این ماژول ها استیج IF را میسازد :

F:/term 8/Lab/final_project_with forwarding/IF_Stage.v - Default *

```
Ln#
 1  module IF_Stage (
 2    input clk, rst, freeze, Branch_taken,
 3    input[31:0] BranchAddr,
 4    output[31:0] PC, Instruction
 5  );
 6
 7  wire[31:0] IF_MUX_out;
 8  wire[31:0] IF_PCReg_out;
 9
10  IF_MUX IF_MUX_inst(
11    Branch_taken,
12    BranchAddr, PC,
13    IF_MUX_out
14  );
15
16  IF_PCReg IF_PCReg_inst(
17    clk, rst, freeze,
18    IF_MUX_out,
19    IF_PCReg_out
20  );
21
22  IF_InstMem IF_InstMem_inst(
23    IF_PCReg_out,
24    Instruction
25  );
26
27  IF_Adder IF_Adder_inst(
28    IF_PCReg_out,
29    PC
30  );
31
32  endmodule
```

(ماژول ها به صورت جدا تست نشدند.)

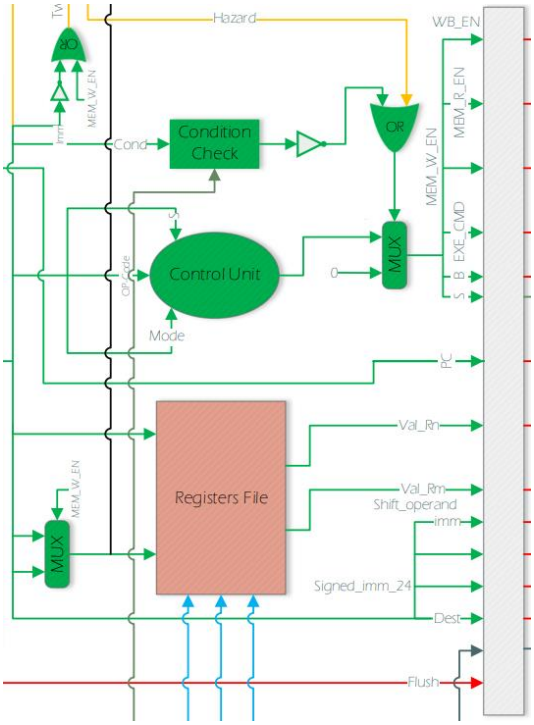
ارور ها

مشکلی در این بخش نداشتیم. فقط اینکه در ابتدا Instruction Memory ساده تر بود برای اینکه بتوانیم کارکرد ماژول ها را تست کنیم.

باید در نهایت آن را با دستورات کامل پر میکردیم که تا مدت خوبی فراموش کردیم این کار را انجام دهیم.

بخش دوم: ID Stage

در این بخش هدف ما کدگشایی دستورالعمل ها بود.



با توجه به شکل بالا این بخش از رجیستر فایل تشکیل شده است که ماژول ثبات عمومی است.

ورودی های آن آدرس هایی است که مقادیر داخل آن را نیاز داریم یا میخواهیم تغییر دهیم.

یک ماژول کنترل یونیت داریم که تمام سیگنال های کنترلی کل پردازنده را تولید میکند.

ماژول کاندیشن چک ها برای بررسی یک سری شرط است که در خود دستور ها وجود دارد.

کد ها

ابتدا یک مالتیپلکسر پشت ثبات عمومی داریم که با توجه به اینکه میخواهیم مقداری را بنویسیم یا نه ورودی را تغییر میدهد.

ثبات عمومی :

این ماژول دارای 16 ثبات 32 بیتی میباشد.

این ثبات هم قابلیت خواندن همزمان از 2 ثبات را به صورت ناهمگام دارد و هم نوشتن به صورت همگام در آن انجام میشود.

پس چندین آدرس از روی دستورالعمل به این ماژول وارد میشود و مقادیر مورد نظر هر آدرس را به ما میدهد.

یا اگر مقداری محاسبه شده که باید در آدرسی نوشته شود، آن مقدار و آدرس مورد نظر به صورت ورودی وارد میشوند و مقدار در آدرس مورد نظر نوشته میشود.

کد این بخش در ادامه آمده است:

F:/term 8/Lab/final_project_with forwarding/RegisterFile.v - Default

```
Ln#
1   module RegisterFile (
2     input clk, rst,
3     input[3:0] src1, src2, Dest_wb,
4     input[31:0] Result_WB,
5     input writeBackEn,
6     output[31:0] reg1, reg2
7   );
8   reg [31:0] regfile[0:14];
9
10  integer j;
11  assign reg1 = regfile[src1];
12  assign reg2 = regfile[src2];
13  always @(negedge clk, posedge rst) begin
14    if(rst) begin
15      for (j=0; j<15; j=j+1)
16        begin
17          regfile[j] <= j;
18        end
19    end
20    else if(writeBackEn) begin
21      regfile[Dest_wb] <= Result_WB;
22    end
23  end
24  endmodule
25
26
```

کنترل یونیت

این ماژول سه ورودی از بیت های مختلف دستور العمل را میگیرد و با توجه به معنی آن ها سیگنال های کنترلی مورد نظر را تولید میکند.

این بیت ها شامل دو بیت mode و 4 بیت op-code و بیت s هستند. که هر کدام را به صورت جداگانه بررسی میکنیم:

mode:

این دو بیت، بیت های 26 و 27 دستورالعمل هستند که سه حالت دارند :

1. 00 به معنی این است که عملیات ما دستور محاسباتی یا منطقی است.
2. 01 به معنی این است که دستور کار با حافظه است.
3. 10 دستور branch را مشخص میکند.

S:

بیت 20 دستور العمل است که نشان میدهد که آیا دستور ما مقدار status register را تغییر میدهد یا نه. همانطور از این بیت میتوان سیگنال های کنترلی نوشتن و خواندن را مشخص کرد.

Opcode:

بیت 21 تا 24 دستور العمل را شامل میشود که زیر دستور ها را مشخص میکند. مثلا اینکه دستور جمع است یا منها یا شیفت دادن. این بخش کارکرد ALU را مشخص میکند. پس به صورت کامندی که وارد بخش exe میشود آن را نام گذاری میکنیم.

همچنین برای مشخص کردن نوشتن هم کاربرد دارد.

کد این بخش به صورت زیر میباشد :

F:/term 8/Lab/final_project_with forwarding/ID_CntrlUnt.v - Default *

```
Ln#
 1  module CntrlUnt (
 2    input sIn,  input[1:0] Mode,   input[3:0] OpCode,  output reg[8:0] CntrlUnt_out)
 3  reg[3:0] EXE_CMD;
 4  reg MEM_W_EN, MEM_R_EN, WB_EN, Branch, sOut;
 5  always @(OpCode, Mode, sIn) begin
 6     EXE_CMD = 4'b0;
 7    {MEM_W_EN, MEM_R_EN, WB_EN, Branch, sOut} = 5'b0;
 8    case (OpCode)
 9      4'b1101: EXE_CMD = 4'b0001; // MOV
10      4'b1111: EXE_CMD = 4'b1001; // MVN
11      4'b0100: EXE_CMD = 4'b0010; // ADD, LDR, STR
12      4'b0101: EXE_CMD = 4'b0011; // ADC
13      4'b0010: EXE_CMD = 4'b0100; // SUB
14      4'b0110: EXE_CMD = 4'b0101; // SBC
15      4'b0000: EXE_CMD = 4'b0110; // AND
16      4'b1100: EXE_CMD = 4'b0111; // ORR
17      4'b0001: EXE_CMD = 4'b1000; // EOR
18      4'b1010: EXE_CMD = 4'b0100; // CMP
19      4'b1000: EXE_CMD = 4'b0110; // TST
20      default: EXE_CMD = 4'b0001;
21    endcase
22    case (Mode)
23      2'b00: begin
24        sOut = sIn;
25        WB_EN = (OpCode == 4'b1010 || OpCode == 4'b1000) ? 1'b0 : 1'b1;
26        end
27      2'b01: begin
28          WB_EN = sIn;
29          MEM_R_EN = sIn;
30          MEM_W_EN = ~sIn;
31      end
32      2'b10: Branch = 1'b1;
33    endcase
34    CntrlUnt_out = {EXE_CMD, MEM_W_EN, MEM_R_EN, WB_EN, Branch, sOut};
35    end
36  endmodule
```

کاندیشن چک

بیت های 28 تا 31 هم به عنوان شروط در دستورالعمل قرار دارند. 4 بیت n,z,c,v را داریم که برقراری شرط ها در جدول 3 دستور کار اورده شده بود و دیگر تکرار نمیکنیم. این ماژول باعث میشود که اگر شرط ها بر قرار نبودند تمامی خروجی های کنترل یونیت صفر شوند. پس بعد از آن نیاز به یک مالتیپلکسر داریم که این کار را انجام دهد.

کد این بخش به صورت زیر میباشد :

F:/term 8/Lab/final_project_with forwarding/ID_CondChck.v - Default *

```
Ln#
 1  module ID_CondChck (
 2    input[3:0] cond,
 3    input[3:0] status,
 4    output reg result
 5  );
 6  wire n, z, c, v;
 7  assign {n, z, c, v} = status;
 8   always @(cond, n, z, c, v) begin
 9          result = 1'b0;
10          case (cond)
11              4'b0000: result = z;
12              4'b0001: result = ~z;
13              4'b0010: result = c;
14              4'b0011: result = ~c;
15              4'b0100: result = n;
16              4'b0101: result = ~n;
17              4'b0110: result = v;
18              4'b0111: result = ~v;
19              4'b1000: result = c & ~z;
20              4'b1001: result = ~c | z;
21              4'b1010: result = (n == v);
22              4'b1011: result = (n != v);
23              4'b1100: result = ~z & (n == v);
24              4'b1101: result = z & (n != v);
25              4'b1110: result = 1'b1;
26              default: result = 1'b0;
27          endcase
28    end
29  endmodule
```

ماژول نهایی ID Stage

این ماژول را با عکس زیر توضیح میدهیم:

```
module ID_Stage (
  input clk,
  input rst,
  input hazard,
  input[3:0] Dest_wb, status,
  input[31:0] Result_WB,
  input writeBackEn,
  input[31:0] IF_PC_out,
  input [31:0] Instruction,
  output [8:0] CntrlUnt_MUX_out,
  output [31:0] Rn, Rm,
  output  imm,
  output [11:0] shiftOp,
  output [23:0] Signed_imm_24,
  output [3:0] Dest, Rmd,
  output Two_src
);

wire cond_out, ID_MEM_W_EN;
wire [8:0] CntrlUnt_out;
wire [3:0] ID_RegFile_MUX_out, cond;
wire [31:0] ID_PC;
assign cond = Instruction[31:28];
assign ID_MEM_W_EN = CntrlUnt_out[4];
assign ID_PC = IF_PC_out;
assign imm = Instruction[25];
assign shiftOp = Instruction[11:0];
assign Signed_imm_24 = Instruction[23:0];
assign Dest = Instruction[15:12];
assign Two_src = (~imm) | ID_MEM_W_EN;
assign Rmd = ID_RegFile_MUX_out;
```

در نهایت ورودی های ما شامل هازارد و دستورالعمل وpc و ورودی های دیگری از استیج های بعدی میباشد. در این بخش توانستیم سیگنال های کنترلی را تولید کنیم که به صورت cntrlUnt_MUX_out میباشد و هدف اصلی این بخش حساب میشد. خروجی های رجیستر فایل هم که دو مقدار Rn و Rm

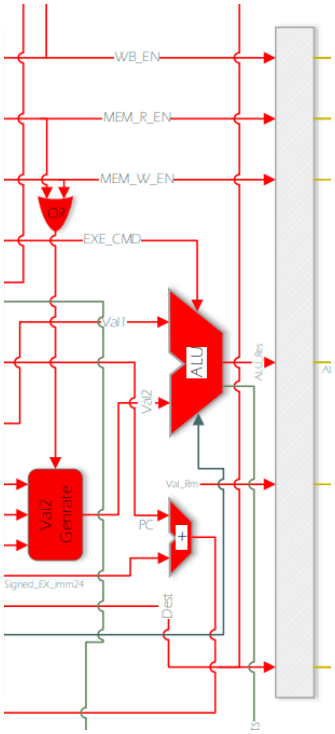
بودند را به مرحله بعدی میدهیم. بخش های خاص دیگر مانند shift operandو imm24 را نیز از دستورالعمل جدا کرده و به مرحله بعد میدهیم.

چند خروجی دیگر مربوط به هازارد یونیت میشوند که بعدا بررسی میکنیم.

همانطور که در قبل اشاره کردم هر ماژول به طور جداگانه تست نداشت و در نهایت همه بخش ها با هم تست شدند. در نهایت کارکرد کل پردازنده توسط دستیاران دیده شده و تایید شده میباشد.

بخش سوم: EXE Stage

در این بخش هدف ما انجام دادن عملیات ها روی مقادیر میباشد.



ماژول اصلی این بخش ALU میباشد که مسئول تولید کردن مقادیر و انجام عملیات ها میباشد.

از آنجایی که مقدار اول همیشه ثابت است اما مقدار دوم میتواند یک عدد یا همان شیفت و یا امیدیت باشد ما برای ایجاد کردن مقدار دوم نیاز به یک Val2Generator داریم.

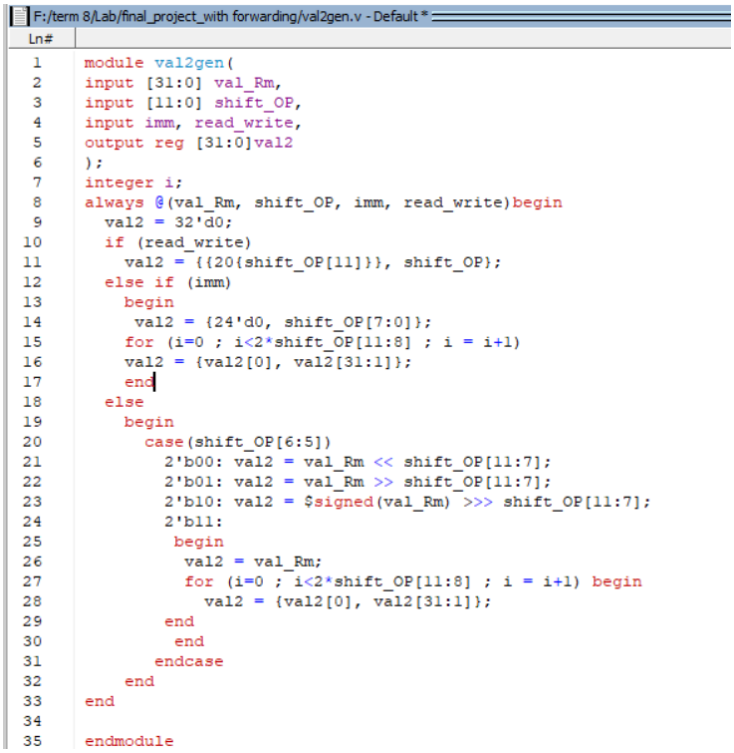
یک جمع کننده هم داریم که در اصل برای محاسبه ی آدرس پرش استفاده میشود.

کد ها

Val2Generator

در این بخش میخواهیم مقدار دوم ورودی ALU را بدست بیاوریم.

این ورودی سه حالت دارد که از روی کد راحتتر قابل توضیح دادن است:



اول از همه چک میشود که دستور مربوط به نوشتن یا خواندن است یا خیر. اگر بود مقدار خروجی ما به صورت sign extend شده ی shift_op میباشد.

اگر در حالت immediate بودیم نیاز داریم که ابتدا مقدار shift_op را از بیت 0 تا 7 برداریم و سپس به دو برابر مقداری که از بیت های 8 تا 11 گفته شده آن را در 32 بیت به سمت راست شیفت بدهیم. این کار در لوپ انجام میشود.

حالت بعدی حالتی است که به یک میزان مشخصی باید دیتای ورودی (VAL_RM) را شیفت بدهیم و سپس مقدار خروجی را تولید کنیم. در اینجا به توجه به بیت ها 6 و 5 shift_op نوع شیفت مشخص میشود.

ALU

این ماژول از آنجایی که باید عملیات را انجام بدهد، دو اپرند 32 بیتی، یک ورودی 4 بیتی (همان EXE_cmd) که نشان میدهد کدام عملیات باید انجام شود، و یک بیت کری به عنوان ورودی میگیرد. خروجی های آن یک 32 بیتی و یک 4 بیتی برای status register.

با توجه به چهار بیت EXE_cmd و جدول 5 دستور کار عملیات را انجام میدهیم و مقدار خروجی را تولید میکنیم.

بیت های C و V در عملیات جمع و تفریق تولید می شوند و در بقیه دستورات صفر خواهند بود. بیت Nبه معنای منفی بودن نتیجه خروجی ALUاست و همیشه برابر بیت 31ام نتیجه خواهد بود. همچنین بیت Z در صورت صفر بودن نتیجه ALU یک خواهد بود

در ورودی هم باید حواسمان باشد که مقدار carry_in را نیز در جمع و منها اضافه کنیم. در جمع که فقط مقدار صفر را در بیت های پر ارزش آن قرار میدهیم. برای منها کردن اما باید حواسمان باشد که قرینه ی آن را بگذاریم و سپس مقادیر صفر را در بیت های پر ارزش آن قرار دهیم.

F:/term 8/Lab/final_project_with forwarding/ALU.v - Default *

```
Ln#
 1  module ALU(
 2    input[31:0] val1, val2,
 3    input[3:0] EXE_cmd,
 4    input[3:0] status,
 5    output[3:0] status_bits,
 6    output reg[31:0] ALU_Res);
 7  reg c, v;
 8  wire z, n;
 9  wire [31:0] carry_pos, carry_neg;
10  assign carry_pos = {{(31){1'b0}}, status[1]};
11  assign carry_neg = {{(31){1'b0}}, ~status[1]};
12  assign status_bits = {n ,z ,c ,v};
13  assign z = ~|ALU_Res;
14  assign n = ALU_Res[31];
15  always @(val1, val2, EXE_cmd, carry_pos, carry_neg) begin
16  ALU_Res = 32'd0;
17  c = 1'b0;
18    case(EXE_cmd)
19      4'b0001: ALU_Res = val2;
20      4'b1001: ALU_Res = ~val2;
21      4'b0010: {c, ALU_Res} = val1 + val2;
22      4'b0011: {c, ALU_Res} = val1 + val2 + carry_pos;
23      4'b0100: {c, ALU_Res} = val1 - val2;
24      4'b0101: {c, ALU_Res} = val1 - val2 - carry_neg;
25      4'b0110: ALU_Res = val1 & val2;
26      4'b0111: ALU_Res = val1 | val2;
27      4'b1000: ALU_Res = val1 ^ val2;
28      default: ALU_Res = {32{1'b0}};
29    endcase
30    v = 1'b0;
31    if (EXE_cmd[3:1] == 3'b001) begin
32    v = (val1[31] == val2[31])&&(val1[31] != ALU_Res[31]);
33    end
34    else if (EXE_cmd[3:1] == 3'b010) begin
35    v = (val1[31] != val2[31])&&(val1[31] != ALU_Res[31]);
36    end
```

جمع کننده

همانطور که قبلا هم اشاره شد این ماژول برای محاسبه ی آدرس پرش استفاده میشود. به این صورت که pc+4 و مقدار sign extend شده 24 بیت سمت راست دستور branch را با هم جمع میکند و این میشود مکان دستوری که به آن پرش کرده ایم.

ارور ها

قسمت بعد از ALU case را به دو روش مختلف نوشتیم. روش اول اینگونه بود:

```
/*if (ALU_Res == 4'b0100 || ALU_Res == 4'b0101) begin
  v = (val1[31] == val2[31])&&(val1[31] != ALU_Res[31]);
end
else if (ALU_Res == 4'b0010 || ALU_Res == 4'b0110) begin
  v = (val1[31] != val2[31])&&(val1[31] != ALU_Res[31]);
end*/
```

روشی که در کد اصلی پیش گرفته ایم بهتر و خوانا تر میباشد.

بخش چهارم: MEM Stage

در این بخش حافظه ی کل پردازنده را طراحی میکنیم. این استیج دو ورودی 32 بیتی ، یکی برای آدرس و یکی مقداری که قرار است در آدرس قرار بگیرد، دریافت میکند. همچنین سیگنال های خواندن و نوشتن در اینجا مشخص میکنند که مقداری را میخواهیم بخوانیم یا بنویسیم.

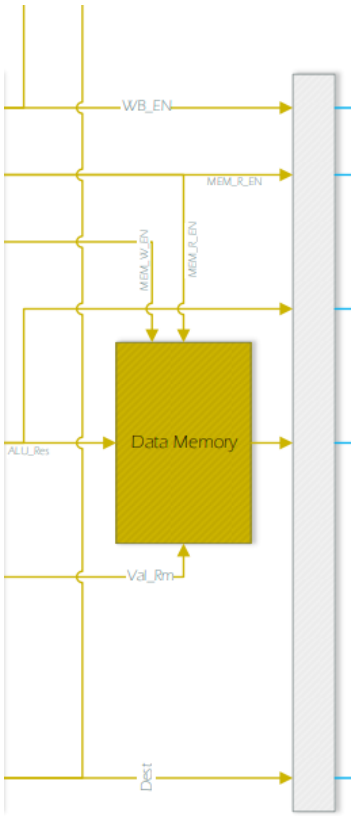
خواندن به صورت ترکیبی و نوشتن در حافظه به صورت ترتیبی انجام میشود. نوشتن در لبه بالا رونده رونده ی کالک صورت میگیرد.

این حافظه تنها یک خط آدرس دارد که هم برای نوشتن و هم برای خواندن استفاده می شود. آدرس در حقیقت داده ی محاسبه شده توسط ALU در مرحله ی قبل میباشد.

حافظه را به صورت آرایه ای از رجیستر ها تعریف میکنیم.

خواندن و نوشتن فقط از آدرسهای مضرب 4 انجام میشود و اگر عددی غیر این بدهیم از ابتدای خط میخواند.

فضای ما از داده 1024 شروع میشود.



کد ها :

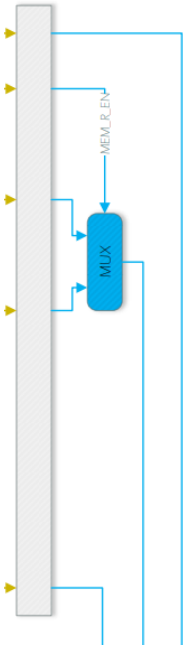
در اینجا هم کد این بخش را مشاهده میکنید:

F:/term 8/Lab/final_project_with forwarding/DataMem.v - Default

```
Ln#
 1  module DataMem (
 2    input clk, rst, MEM_W_EN, MEM_R_EN,
 3    input [31:0] ALU_Res, Rm_Val,
 4    output reg[31:0] memData
 5  );
 6    wire [31:0] aligned_add, add;
 7    reg [31:0] memory [0:63];
 8
 9    assign add = ALU_Res - 32'd1024;
10    assign aligned_add = {2'b00, add[31:2]};
11
12    integer i;
13    always @(negedge clk, posedge rst) begin
14      if (rst)
15        for (i = 0; i < 64; i = i + 1) begin
16          memory[i] = 32'd0;
17        end
18      else if (MEM_W_EN)
19        memory[aligned_add] <= Rm_Val;
20    end
21
22    //assign memData = memory[aligned_add];
23    always @(MEM_R_EN, aligned_add) begin
24      if (MEM_R_EN)
25        memData = memory[aligned_add];
26    end
27
28  endmodule
```

بخش پنجم :WB Stage

در این بخش فقط به یک مالتیپلکسر نیاز داریم.



این ماژول داده خوانده شده از حافظه و داده محاسبه شده توسط ALU را به همراه شماره رجیستر مقصد از طریق رجیسترهای پایپ لاین به عنوان ورودی دریافت میکند و با توجه به سیگنال های کنترلی محتوای رجیسترفایل را تغییر میدهد.

همچنین هرگاه WB_EN برابر یک باشد، با توجه به MEM_R_EN داده ی خوانده شده از حافظه یا داده ی محاسبه شده توسط ALU در رجیستر فایل نوشته می شود.

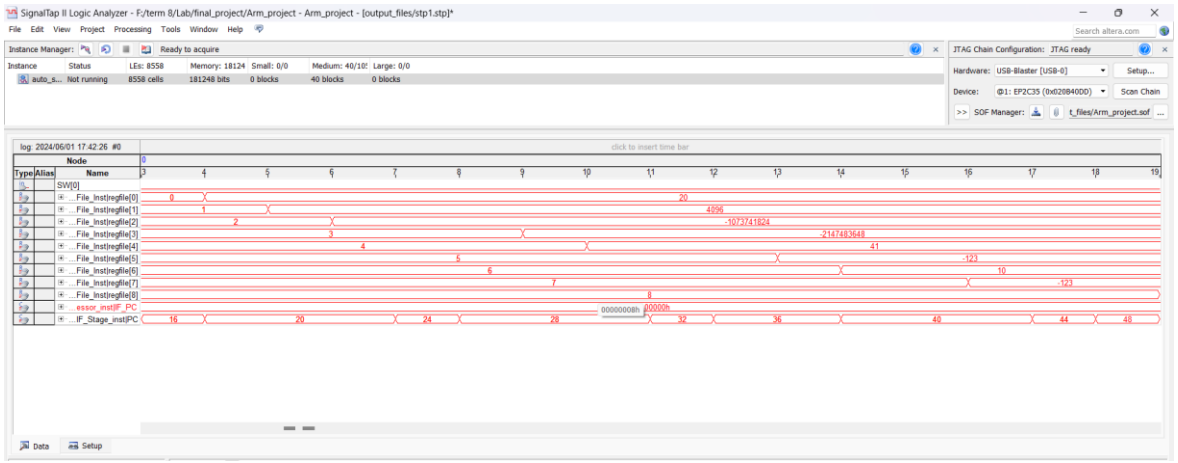
تست کردن پردازنده

همه بخش های پردازنده را به هم متصل میکنیم.

تعداد المان های مورد نیاز برای تولید این برنامه و تعداد رجیستر ها و پین ها و بیت های مموری، همه در شکل زیر آمده است:

Flow Summary	
Flow Status	Successful - Sat Jun 01 17:42:05 2024
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	Arm_project
Top-level Entity Name	ARM
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	11,332 / 33,216 (34 %)
Total combinational functions	5,814 / 33,216 (18 %)
Dedicated logic registers	9,237 / 33,216 (28 %)
Total registers	9237
Total pins	418 / 475 (88 %)
Total virtual pins	0
Total memory bits	181,248 / 483,840 (37 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

در بخش سیگنال تب مشاهده میکنیم که مقدار نهایی هر رجیستر درست میباشد و کل پردازنده به درستی کار خود را انجام داده است.



بخش ششم Hazard Detection :

اول ببینیم که در پردازنده ها چه مخاطره هایی رخ میدهد:

مخاطره ی ساختاری:

این مخاطره در خود خط لوله و به دلیل ساختار آن رخ میدهد. بین مرحله ی ID و WB به دلیل همزمانی خواندن و نوشتن از ثبات های عمومی ممکن است مخاطره رخ بدهد. اما ما در نوشتن از لبه پایین رونده و برای خواندن از لبه بالا رونده استفاده کردیم، به همین دلیل این مشکل را نخواهیم داشت.

مخاطره ی کنترلی:

بدلیل وجود دستورات پرش ممکن است محاسبه آدرس پرش با تاخیر تشخیص داده شود و دستور اشتباهی وارد خط لوله بشود. اما ما FLUSH را داریم که از این موضوع جلوگیری میکند

دلیل اضافه شدن این یونیت در اصل **مخاطره ی داده ای** میباشد

مخاطره ی داده ای:

این مخاطره خودش سه دسته میشود:

خواندن پس از نوشتن: وقتی هنوز مقدار جدید یک رجیستر را ننوشته ایم، میخواهیم مقدارش را بخوانیم.

نوشتن پس از خواندن: میخواهیم مقداری را بخوانیم که ممکن است مقدار آن تغییر کند.

نوشتن پس از نوشتن: ترتیب نوشتن در رجیستر مقصد ممکن است تغییر کند. این نوع مخاطره در پردازنده ی ما رخ نمیدهد.

راه رفع آن:

باید در یک جایی از مدار Src1 و Src2 را با مقصد های مرحله آخر(EXE و MEM) مقایسه کنیم. اینکار در مرحله ID رخ میدهد.

و اگر یکی بودند باید یک سیگنال کنترلی را یک کنیم که باعث شود خط لوله متوقف شود . حبابی را به خط لوله وارد کند.

همچنین باید همه حالت هارا در نظر بگیریم:

مثلا 4 حالت زیر :

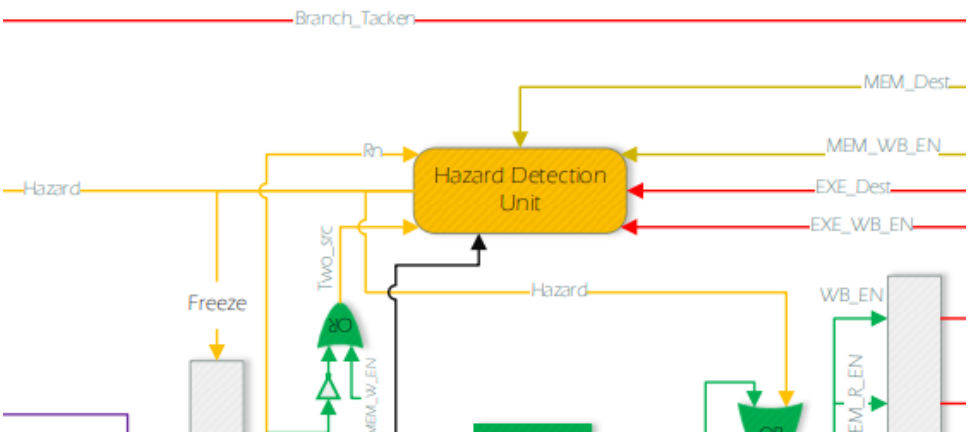
1. برابری Src1 با مقصد EXE و در صورت یک بودن wb_en در مرحله EXE
2. برابری Src1 با مقصد MEM و در صورت یک بودن wb_en در مرحله MEM
3. برابری Src2 با مقصد EXE و در صورت یک بودن wb_en در مرحله EXE و دو منبعی بودن دستور
4. برابری Src2 با مقصد MEM و در صورت یک بودن wb_en در مرحله MEM و دو منبعی بودن دستور

F:/term 8/Lab/final_project_with Sram/HzrdDtctr.v - Default *

```
Ln#
 1  module HzrdDtctr (
 2    input MEM_WB_EN, EXE_WB_EN, Two_src,
 3    input [3:0] EXE_Dest, MEM_Dest, src1, src2,
 4    output Hazard
 5  );
 6
 7      assign Hazard = ((EXE_WB_EN == 1'b1)&&(src2 == EXE_Dest)) ||
 8       ((MEM_WB_EN == 1'b1)&&(src2 == MEM_Dest)) ||
 9       ((EXE_WB_EN == 1'b1)&&(Two_src && (src1 == EXE_Dest)))||
10       ((MEM_WB_EN == 1'b1)&&(Two_src && (src1 == MEM_Dest)));
11
12
13  endmodule
```

همچنین حواسمان هست که باید قابلیت فریز شدن را به رجیستر PC و رجیستر های بعد آن اضافه کنیم.

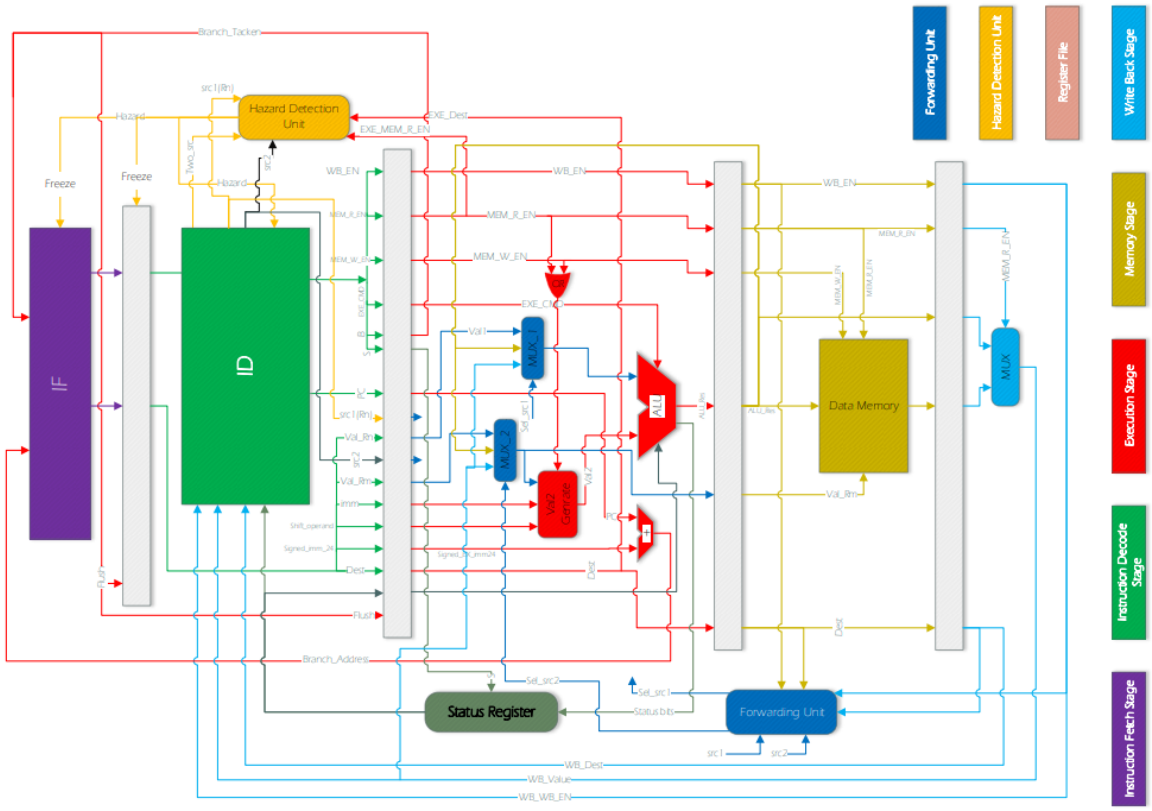
بسته به تمام این نکات ورودی های این یونیت به صورت زیر میبشد و خروجی آن همان فریز یا هازارد میباشد



بخش هفتم Forwarding Unit :

طبق بخش قبل متوجه شدیم که برای بعضی از دستور ها مجبور هستیم بخشی از پایپ لاین را متوقف کنیم تا مقادیر مورد نظر تولید و نوشته شوند.

اما برای ما سرعت اهمیت دارد و بعضی وقت ها مقادیری که هنوز نوشته نشده اند مقادیرشان آماده است و فقط کافی است که زودتر آن را به جلو انتقال بدهیم.



همانطور که مشاهده میکنید چندین بخش به کل پردازنده اضافه شده است که شامل یک یونیت فوروادینگ، دو مالتی پلکسر و تغییراتی در هازارد یونیت میباشد که در ادامه به کارکرد هر یک میپردازیم:

فورواردینگ یونیت

این ماژول به این صورت کار میکند که مشخص میکند نیاز به فوروادینگ داریم یا خیر.

همچنین یک سیگنال به نام forwardEn داریم که به ما کمک میکند که مشخص کنیم که میخواهیم از فوروادینگ استفاده کنیم یا خیر.

در بخش فوروادینگ چندین شرط را باید چک کنیم:

src1 و src2 را با dest های بخش mem و wb مقایسه میکنم. دلیل این کار این است که میخواهیم ببینیم اگر در بخش های قبلی به مقادیری نیاز داریم که در بخش های بعدی محاسبه شده اند آن مقادیر را برگردانیم. در عین حال باید چک کنیم که wb اصلا enable میباشد یا خیر. پس این مقایسه ها ابتدا شرطenable بودن را نیز میخواهند.

حال خروجی ها به صورت سلکت هستند که مشخص میکند در مرحله EXE ورودی های ALU چه باشند. مثلا اگر فوروادینگ نیازی نباشد همان مقادیر قبلی فرستاده میشوند. اگر فوروادینگ نیاز باشد باید سیم بکشیم و مقادیر جدید را به EXE وارد کنیم و سلکت را هم جوری انتخاب کنیم که این مقادیر را بفرستد.

پس سلکت در کل سه حالت دارد:

یا مقادیر دیفالت 00 را میفرستیم. یا مقادیر از MEM آمده : 01 ، یا از مقادیر داخل بخش WB آمده است.:10 . برای Src1 و Src2 جداگانه این موضوعات را چک میکنیم و سیگنال های سلکت را تولید میکنیم.

یک نکته ی قابل توجه این است که اولویت فورواد کردن مقادیر با بخش MEM میباشد

کد این بخش به صورت زیر میباشد:

F:/term 8/Lab/final_project_with Sram/Forward_Unit.v - Default

```
Ln#
 1  module Forward_Unit(
 2      input forward_En,
 3      input [3:0] src1, src2,
 4      input wbEnMem, wbEnWb,
 5      input [3:0] destMem, destWb,
 6      output reg [1:0] sel_Src1, sel_Src2
 7      );
 8      always @(forward_En, src1, wbEnMem, wbEnWb, destMem, destWb) begin
 9          sel_Src1 = 2'b00;
10          if (forward_En) begin
11              if (wbEnMem && (destMem == src1)) begin
12                  sel_Src1 = 2'b01;
13              end
14              else if (wbEnWb && (destWb == src1)) begin
15                  sel_Src1 = 2'b10;
16              end
17          end
18      end
19
20      always @(forward_En, src2, wbEnMem, wbEnWb, destMem, destWb) begin
21          sel_Src2 = 2'b00;
22          if (forward_En) begin
23              if (wbEnMem && (destMem == src2)) begin
24                  sel_Src2 = 2'b01;
25              end
26              else if (wbEnWb && (destWb == src2)) begin
27                  sel_Src2 = 2'b10;
28              end
29          end
30        end
31  endmodule
```

هازارد یونیت

باید حواسمان باشد که هازارد یونیت را نیز تغییر بدهیم.

F:/term 8/Lab/final_project_with Sram/HzrdDtctr_2.v - Default

```
Ln#
 1  module HzrdDtctr_2 (
 2    input MEM_WB_EN, EXE_WB_EN, Two_src, EXE_MEM_R_EN,
 3    input [3:0] EXE_Dest, MEM_Dest, src1, src2,
 4    input Forward_EN,
 5    output reg Hazard
 6  );
 7  always @(MEM_WB_EN, EXE_WB_EN, Two_src,EXE_Dest, MEM_Dest, src1, src2, EXE_MEM_R_EN, Forward_EN) begin
 8     Hazard = 1'b0;
 9     if (Forward_EN) begin
10       if (EXE_MEM_R_EN)
11         begin
12           if (src2 == EXE_Dest || (Two_src && (src1 == EXE_Dest))) begin
13             Hazard = 1'b1;
14           end
15         end
16     end
17     else begin
18        if (EXE_WB_EN) begin
19          if (src1 == EXE_Dest || (Two_src && (src2 == EXE_Dest))) begin
20            Hazard = 1'b1;
21          end
22        end
23        if (MEM_WB_EN) begin
24          if (src1 == MEM_Dest || (Two_src && (src2 == MEM_Dest))) begin
25            Hazard = 1'b1;
26          end
27        end
28      end
29  end
30
31  endmodule
```

در اینجا مشاهده میکنید که تغییری که حاصل شده است این است که وقتی فوروادینگ enable است دیگر نیاز به توقف نداریم و باید فقط حالاتی که انتقال به جلو برای آن ها ممکن نیست را متوقف کنیم. و دستور LDR این شرایط را دارا است. این موضوع را به بخش با فوروادینگ اضافه میکنیم و یک else برای بخش قبلی که بدون فوروادینگ بود میگذاریم.

اضافه کردن دو مالتی پلکسر :

در این بخش همان سلکت هایی که برای دو سورس مقدار دهی کردیم وارد دو مالتیپلکسر میشوند که مشخص میکند ورودی اول و دوم ALU چیست. هر ماکس سه ورودی دارد که طبق سلکت ها قبلا توضیح داده شده اند. این نکته که اگر فوروادینگ Enable نبود همیشه ورودی اول وارد شود هم در نظر میگیریم.

در نهایت forwardEn را یک میکنیم و در کوارتس مدل را شبیه سازی میکنیم.

تست کردن پردازنده با فوروادینگ

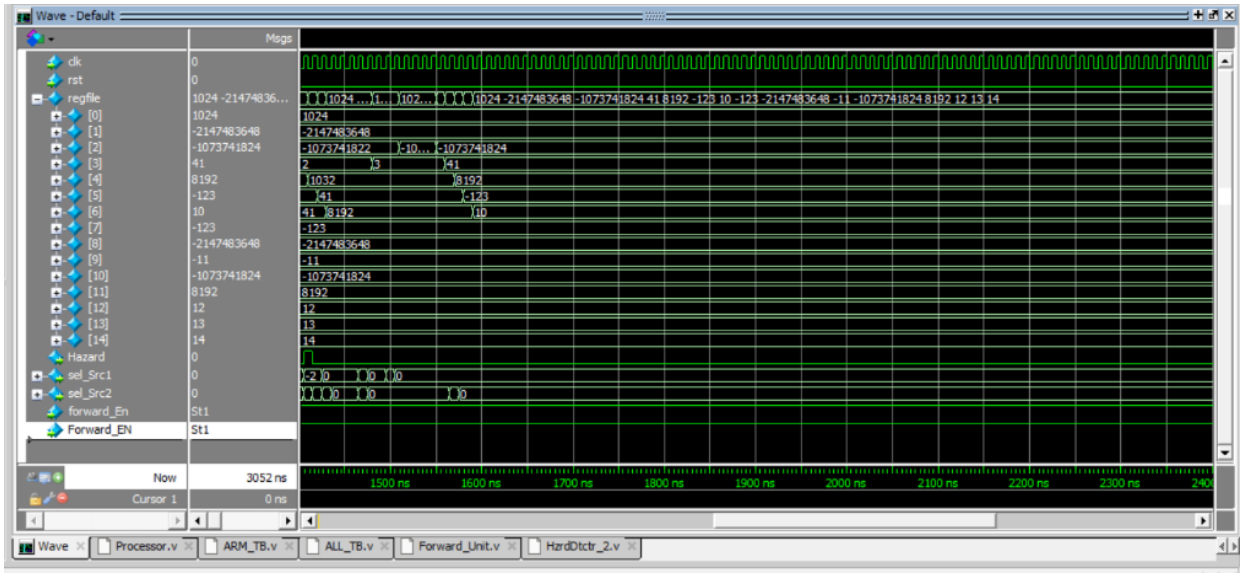
در این بخش ابتدا در مدلسیم و سپس در کوارتس مدل را شبیه سازی میکنیم. مشاهده میکنیم که به درستی کار میکند. نکته ی حائز اهمیت این است که سرعت سیستم به مقدار زیادی افزایش یافته و تعداد کلاک های کمتری نیاز داشته است، چرا که همانطور که توضیح داده شد نیازی نبود که برای رسیدن داده ها صبر کنیم، آن ها را زودتر به جلو میفرستادیم.

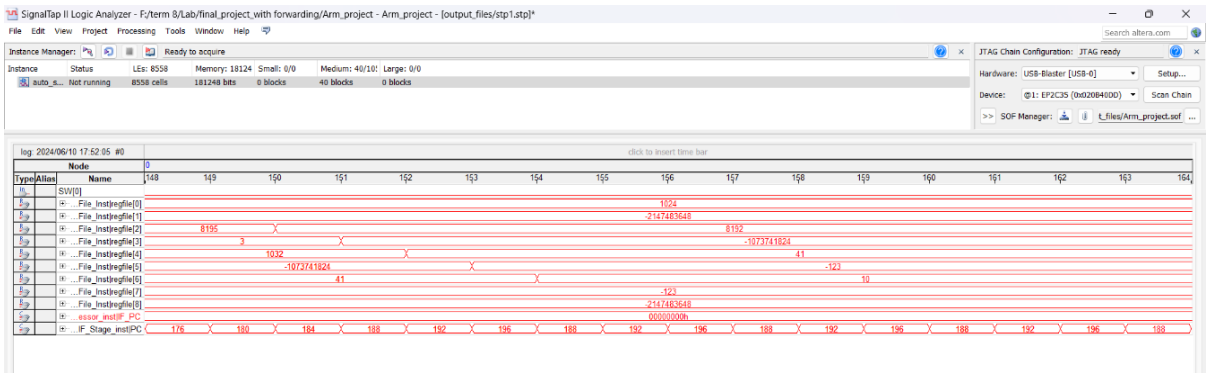
قبل از اضافه کردن فورواردینگ : 2380 نانو ثانیه : 476 کلاک

بعد از اضافه کردن فورواردینگ: 1780 نانو ثانیه: 356 کلاک

افزایش سرعت:

$$\frac{476 - 356}{476} = 25\ \%$$





میزان هزینه بر کارایی:

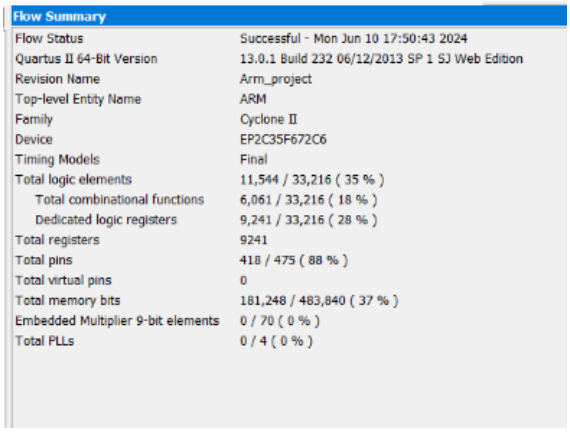
نسبت به دفعه قبل چند المان بیشتر داشته ایم؟

در این حالت المان های منطقی بیشتری نسبت به دفعه قبل نیاز داریم. این هزینه ای است که برای سرعت بیشتر میدهیم.

در بخش بدون فورواردینگ 11332 المان منطقی داشتیم اما اینجا طبق شکل زیر 11544 المان داریم:

$$\frac{11544 - 11332}{11332} = 2\ \%$$

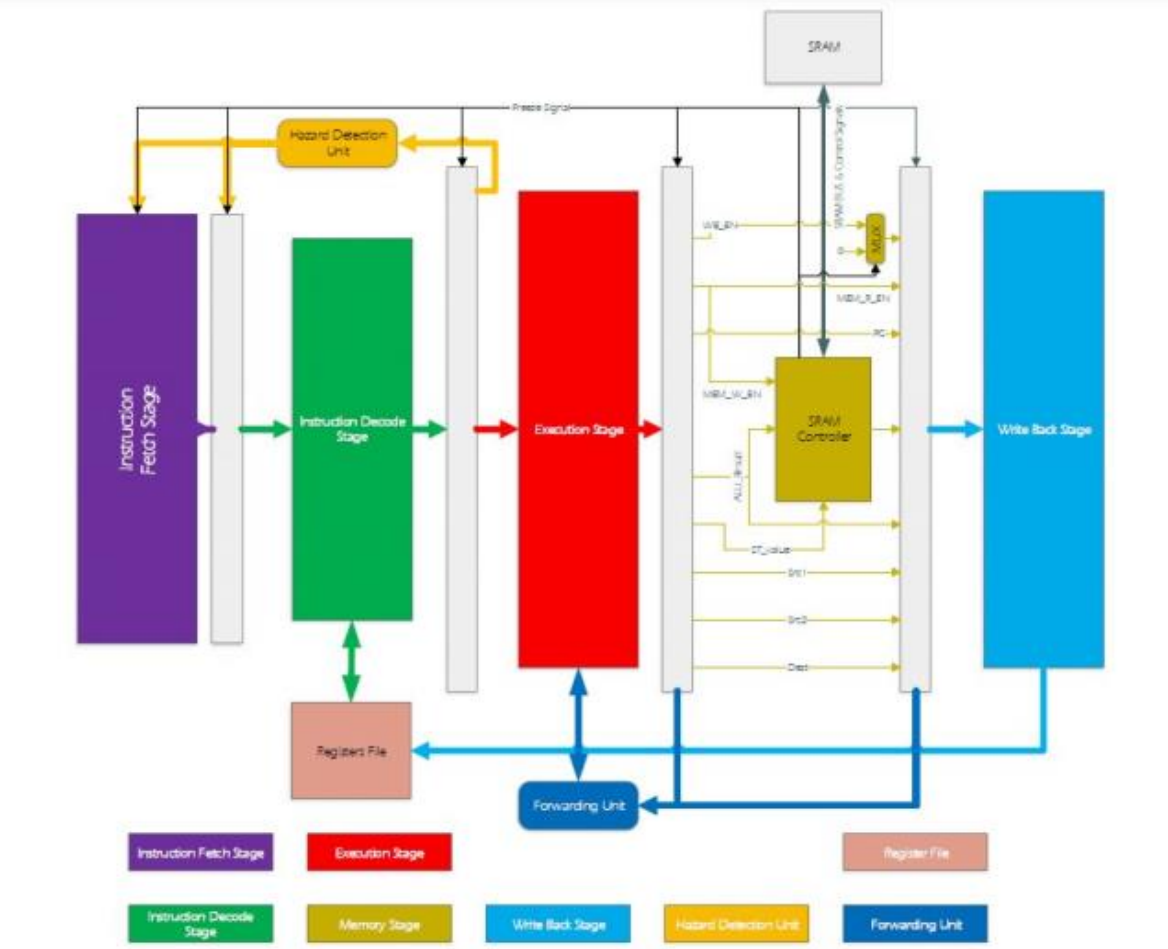
هزینه ای که دادیم نسب به سرعتی که گرفتیم معقول و به صرفه میباشد.



Flow Summary	
Flow Status	Successful - Mon Jun 10 17:50:43 2024
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	Arm_project
Top-level Entity Name	ARM
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	11,544 / 33,216 (35 %)
Total combinational functions	6,061 / 33,216 (18 %)
Dedicated logic registers	9,241 / 33,216 (28 %)
Total registers	9241
Total pins	418 / 475 (88 %)
Total virtual pins	0
Total memory bits	181,248 / 483,840 (37 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

بخش هشتم SRAM :

با توجه به محدود بودن حافظه‌ی پردازنده، نیاز به یک حافظه‌ی بزرگ‌تر خارجی به جای حافظه‌ی کوچک داخلی می‌باشد که ما از نوع SRAM آن استفاده کردیم.



کدها

برای پیاده‌سازی این بخش ما تلاش کردیم تا ۳ ماژول را پیاده‌سازی کنیم.

ماژول SRAM

```
module SRAM(
    input clk, rst,
    input SRAM_WE_N,
    input [17:0] SRAM_ADDR,
    inout [15:0] SRAM_DQ
);
    reg [15:0] memory [0:511];
    assign SRAM_DQ = (SRAM_WE_N == 1'b1) ? memory[SRAM_ADDR] : 16'dz;

    always @(posedge clk) begin
        if (SRAM_WE_N == 1'b0) begin
            memory[SRAM_ADDR] = SRAM_DQ;
        end
    end
endmodule
```

این ماژول جزو پردازنده نبود و در واقع قرار بود برای تست آن استفاده شود. این ماژول به نوعی حافظه روی برد را در کد ما شبیه‌سازی می‌کرد.

ماژول SRAM MUX

```
module SRAM_MUX (
   input sel,
     input MEM_WB_EN,
     output SRAM_MUX_out
);

assign SRAM_MUX_out = sel ? 1'd0: MEM_WB_EN;

endmodule

```

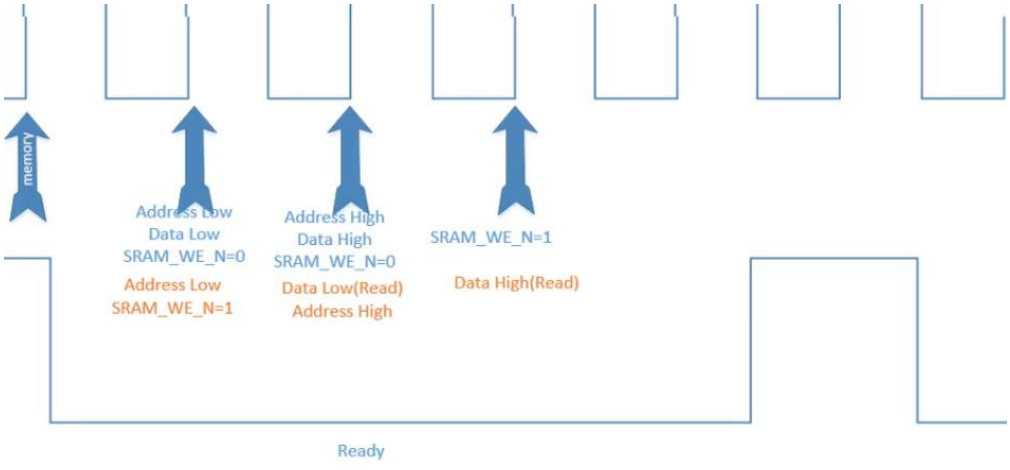
این ماژول مشخص می‌کند که WB فعال باشد یا خیر. در واقع در صورتی که هنوز SRAM Controller در حال گذراندن استیت های خود است، چیزی نوشته نمی‌شود و برای همین هم سیگنال خروجی این ماژول صفر می ماند تا کار SRAM Controller تمام شود.

```
module SRAM_Controller(
    input clk, rst,
    input wrEn, rdEn,
    input [31:0] address,
    input [31:0] writeData,
    output reg [31:0] readData,
    output reg ready,              // to freeze other stages

    inout [15:0] SRAM_DQ,          // SRAM Data bus 16 bits
    output reg [17:0] SRAM_ADDR,   // SRAM Address bus 18 bits
    output SRAM_UB_N,              // SRAM High-byte data mask
    output SRAM_LB_N,              // SRAM Low-byte data mask
    output reg SRAM_WE_N,          // SRAM Write enable
    output SRAM_CE_N,              // SRAM Chip enable
    output SRAM_OE_N               // SRAM Output enable
);
    assign {SRAM_UB_N, SRAM_LB_N, SRAM_CE_N, SRAM_OE_N} = 4'b0000;
    wire [31:0] memAddr;
    assign memAddr = address - 32'd1024;
    wire [17:0] sramLowAddr, sramHighAddr;
    assign sramLowAddr = {memAddr[18:2], 1'b0};
    assign sramHighAddr = sramLowAddr + 18'd1;
    reg [15:0] DQ;
    assign SRAM_DQ = wrEn ? DQ : 16'bz;
    localparam Idle = 3'd0, 2 = 3'd1, 3 = 3'd2, 4 = 3'd3, 5 = 3'd4, Done = 3'd5;
    reg [2:0] ps, ns;

    always @(ps or wrEn or rdEn) begin
        case (ps)
            Idle: ns = (wrEn == 1'b1 || rdEn == 1'b1) ? 2 : Idle;
            2: ns = 3;
            3: ns = 4;
            4: ns = 5;
            5: ns = Done;
            Done: ns = Idle;
        endcase
    end
    always @(posedge clk or posedge rst) begin
        if (rst) ps <= Idle;
        else ps <= ns;
    end

    always @(ps, wrEn, rdEn, sramLowAddr, SRAM_DQ, writeData, sramHighAddr) begin
        SRAM_ADDR = 18'b0;
        SRAM_WE_N = 1'b1;
        ready = 1'b0;

        case (ps)
            Idle: ready = ~(wrEn | rdEn);
            2: begin
                SRAM_WE_N = ~wrEn;
                if (rdEn) begin
                    SRAM_ADDR = sramLowAddr;
                end
                else if (wrEn) begin
                    SRAM_ADDR = sramLowAddr;
                    DQ = writeData[15:0];
                end
            end
            3: begin
                SRAM_WE_N = ~wrEn;
                if (rdEn) begin
                    SRAM_ADDR = sramHighAddr;
                    readData[15:0] <= SRAM_DQ;
                end
                else if (wrEn) begin
                    SRAM_ADDR = sramHighAddr;
                    DQ = writeData[31:16];
                end
            end
            4: begin
                SRAM_WE_N = 1'b1;
                readData[31:16] <= SRAM_DQ;
            end
            5: begin
                SRAM_WE_N = 1'b1;
            end
            Done: ready = 1'b1;
        endcase
    end
endmodule
```

این ماژول در واقع اصلی‌ترین بخش کار ما بود. در ادامه به توضیح ورودی‌خروجی‌های با پیشوند SRAM_ پرداخته می‌شود :

- ADDR : این ورودی ۱۸ بیتی، آدرس خواندن یا نوشتن ما را مشخص می‌کند. (حافظه ۵۱۲ کیلوبایتی ما نیاز به این تعداد خط آدرس دارد.)
- DQ : این پورت هم ورودی و هم خروجی تعریف شده و برای رد و بدل کردن داده‌ی خواندنی یا نوشتنی استفاده می‌شود. در مواقع خواندن، نیاز از تا زد شود تا داده به درستی خوانده شود. این پورت ۱۶ بیتی است و لذا برای انتقال داده‌های ما که جند بیت دارند، باید طی چند کلاک از این پورت استفاده کرد.
- WE_N : این ورودی که با صفر فعال می‌شود، هنگام نوشتن را مشخص می‌کند.
- RE_N : این ورودی که با صفر فعال می‌شود، هنگام خواندن را مشخص می‌کند.
- LB_N , UB_N , CE_N , OE_N به ترتیب برای فعال کردن خروجی حافظه، چیپ حافظه، بایت پرارزش و بایت کم‌ارزش به کار می‌روند و با صفر فعال می‌شوند. این چهار عدد در این قسمت همواره به صفر متصل هستند.

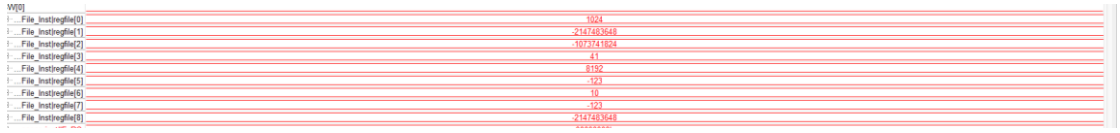
حالات این کنترلر به شکل زیر در ویدیو به تعداد ۶ حالت (۴ حالت برای نوشتن و ۳ حالت برای خواندن و ۶ برای حافظه نهان) تعریف شدند.



در حین خواندن/نوشتن پردازنده متوقف می شود تا در آخر آماده شدن رخ دهد و پردازنده به ادامه‌ی کار خود بپردازد. حین کار با دستورات مموری، باقی رجیسترها در حین انجام این عملیات فریز می‌شوند.

طبیعی است که این چند سیکل شدن فعالیت بخش مموری، زمان انجام عملیات طولانی‌تر می شود و پرفرمنس کاهش می‌یابد. اما به دلیل استفاده از حافظه خارجی، تعداد المان‌های استفاده‌شده کم می‌شوند.

در صورت کارکرد درست، همانند بخش‌های قبل به شکل زیر برای سیگنال تپ خواهیم رسید :



در نتیجه سنتز این امر مشخص می‌شود :

Flow Summary	
Flow Status	Successful - Tue May 23 18:12:55
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	asdf
Top-level Entity Name	arm
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	5,652 / 33,216 (17 %)
Total combinational functions	3,680 / 33,216 (11 %)
Dedicated logic registers	3,549 / 33,216 (11 %)
Total registers	3549
Total pins	418 / 475 (88 %)
Total virtual pins	0
Total memory bits	199,680 / 483,840 (41 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

ارور های کلی

در نوشتن تاپ ماژول کلی یک سری از سیم ها در بالا تعریف نشده بودند اما modelsim از این موضوع ایراد نمیگیرد.

یک بخش بود که به دلیل کوچک بزرگ بودن یک حرف در نام یک سیم، سیم ها به درستی به هم وصل نشده بودند و این موضوع بخش فورواردینگ را دچار اختلال کرده بود.

در حالتی که داشتیم فایل ها را از مادلسیم به کوارتس میبردیم. باید حواسمان باشد که تست بنچ هارا نبریم. در عین حال یک سری از ماژول ها درست سنتز نمیشدند. در نهایت فهمیدیم که در جایی که i را برای چرخش در لوپ ها تعریف کرده بودیم نوشته بودیم initial. ولی بخش هایی که initial دارند در کوارتس سنتز نمیشوند.

همچنین از نظر استراکچر کد ها، در بخش رجیستر یک جا از if و else به اشتباهی استفاده کرده بودیم. یعنی شرطی که در else بود باید در هر صورت اجرا میشد نه اینکه else شود.

در برخی از بخش ها سیگنال ها درست به هم متصل نشده بودند.

یک جا اشتباهی Rn و Rm را جابجا وصل کرده بودیم.

در بخش Sram به مشکل ایکس شدن سیگنال ها میخوردیم که به دلیل ریختن همزمان دو مقدار در آن سیم بود.