

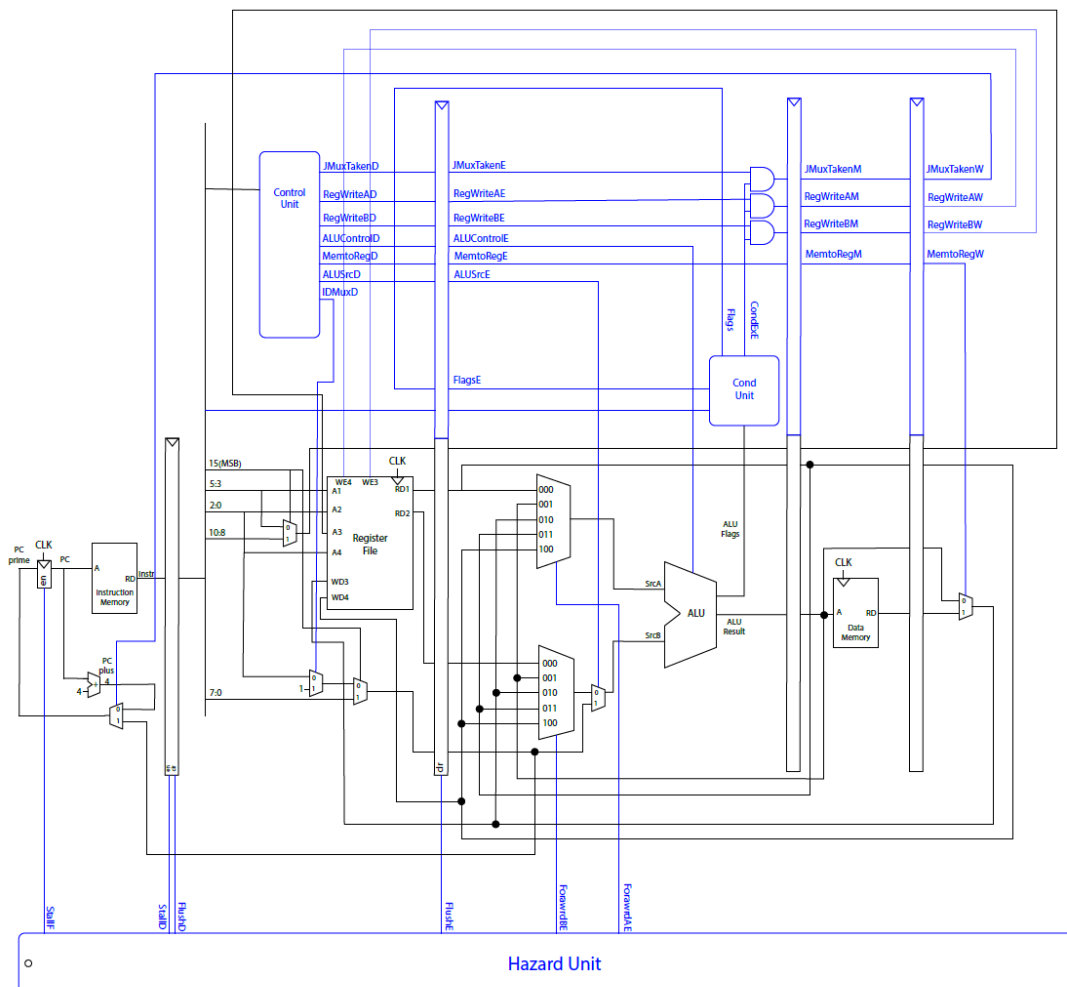
## دورنما

در آزمایش قبلی طراحی یک پردازنده تک‌چرخه‌ای را با موفقیت به اتمام رساندیم. حال برای استفاده بهینه از تمامی اجزای مدار با انجام یک سری تغییرات آن را به پردازنده‌ی خط‌لوله تبدیل می‌کنیم. با انجام این کار عملکرد و توان پردازنده بهبود می‌یابد ولی باید به یک سری نکات در حیطه انواع هازارد دقت کرد و از خراب‌شدن و نامعتبرشدن آن‌ها جلوگیری نمود.

به این منظور پردازنده قبلی را به پنج بخش Fetch, Decode, Execute, Memory, WriteBack تقسیم می‌کنیم.

عموم کدها مانند سری قبل می‌باشد و در این گزارش فقط کدهای تغییر یافته آورده شده‌است.

## شمای کلی مدار



## ماژول CPU

```

module cpu( input logic          clk, reset,
            output logic [7:0]   PCF,
            input logic  [15:0]  InstrF,
            output logic [7:0]   ALUResultM,
            input logic  [7:0]   ReadDataM,
            output logic [1:0]   ShowE,
            output logic [7:0]   ShowDataE);

logic [3:0] ALUFlags, ALUControlD, FlagsE, Flags_prim;
logic RegWriteAD, RegWriteBD,
      RegWriteAM, RegWriteBM,
      RegWriteAW, RegWriteBW,
      ImmSrcD, MemtoRegD,
      IDmuxD, JMuxD, JMuxTakenE,
      FlagWriteD, FlagWriteE,
      CondExE,
      MemtoRegE,
      FlushE, FlushD,
      StallD, StallF;
logic [15:0] InstrD, InstrE;
logic [1:0] ShowD;
logic [7:0] Match;
logic [2:0] ForwardAE, ForwardBE;

controller c(  clk, reset, InstrD,
               RegWriteAD, RegWriteBD, ImmSrcD,
               IDmuxD, JMuxD,
               MemtoRegD, ShowD,
               FlagWriteD ,ALUControlD); // ✓

condunit   cu( clk, reset, InstrE,
               FlagWriteE, FlagsE, ALUFlags,
               Flags_prim, CondExE); // ✓

hazardunit hu(clk, reset, MemtoRegE, JMuxTakenE,
               RegWriteAW, RegWriteAM,
               RegWriteBW, RegWriteBM,
               Match, ForwardAE, ForwardBE,
               FlushE, FlushD,
               StallD, StallF); // ✓

datapath dp(  clk, reset,
               InstrF, InstrD, InstrE,
               FlagsE, ALUFlags, Flags_prim, CondExE,

```

```
RegWriteAD, RegWriteBD,  
RegWriteAW, RegWriteAM,  
RegWriteBW, RegWriteBM,  
ImmSrcD,  
IDmxD, JMxD, JMuxTakenE,  
MemtoRegD, MemtoRegE,  
FlagWriteD, FlagWriteE,  
ShowD, ShowE, ShowDataE,  
ReadDataM, PCF, ALUResultM,  
ALUControlD,  
Match, ForwardAE, ForwardBE,  
FlushE, FlushD,  
StallD, StallF);
```

endmodule

### عملکرد:

منطق اصلی کد همانند تک‌چرخه‌ایست با این تفاوت که بیت‌های کنترلی برای هر یک از ۵ بخش بالا به طور جداگانه در نظر گرفته شده‌اند و همچنین یک سری بیت برای مدیریت هازاردها به آن اضافه می‌شود.

لازم به ذکر است ماژول Controller و CondUnit هم به علت همین تقسیم‌بندی‌ها باید از دیگر جدا شوند و یک ماژول Hazard برای مدیریت سیگنال‌های مربوطه ایجاد شود.

## Datapath ماژول

```

module datapath(    input logic clk, reset,
                    input logic [15:0] InstrF,
                    output logic [15:0] InstrD, InstrE,
                    output logic [3:0]  FlagsE, ALUFlags,
                    input logic [3:0]  Flags_prim,
                    input logic        CondExE,
                                RegWriteAD, RegWriteBD,
                    output logic        RegWriteAW, RegWriteAM,
                                RegWriteBW, RegWriteBM,
                    input logic        ImmSrcD,
                                IDmxD, JMuxD,
                    output logic        JMuxTakenE,
                    input logic        MemtoRegD,
                    output logic        MemtoRegE,
                    input logic        FlagWriteD,
                    output logic        FlagWriteE,
                    input logic [1:0]  ShowD,
                    output logic [1:0]  ShowE,
                    output logic [7:0]  ShowDataE,
                    input logic [7:0]  ReadDataM,
                    output logic [7:0]  PCF,
                    output logic [7:0]  ALUResultM,
                    input logic [3:0]  ALUControlD,
                    output logic [7:0]  Match,
                    input logic [2:0]  ForwardAE, ForwardBE,
                    input logic        FlushE, FlushD,
                                StallD, StallF);

    // Fetch Logics
    logic [7:0] PCprime, PCplus4;
    // Decode Logics
    logic [2:0] RA1, RA2, WA3, WA4;
    logic [7:0] RD1, RD2, idtemp, ImmD;
    //Execute Logics
    logic [7:0] RD2temp, ImmE, SrcA, SrcB, ALUResultE, RD1E, RD2E;
    logic        ImmSrcE, RegWriteAE, RegWriteBE, RegWriteAE_temp, RegWriteBE_temp
;
    logic [3:0] ALUControlE;
    logic [2:0] RA1E, RA2E, WA3E, WA4E;
    // Memory Logics
    logic [7:0] RD1M;
    logic        MemtoRegM;
    logic [2:0] WA3M, WA4M;

```

```

// Writeback Logics
logic [7:0] ALUResultW, ResultW, ReadDataW, RD1W;
logic [2:0] WA3W, WA4W;
logic      MemtoRegW;
// Match Logics
logic Match_1E_M_3, Match_1E_W_3, Match_2E_M_3, Match_2E_W_3,
      Match_1E_M_4, Match_1E_W_4, Match_2E_M_4, Match_2E_W_4;

// Fetch Stage
mux2 #(8)      pcmux( PCPlus4, ImmE, JMuxTakenE, PCprime);
flopnr #(8)     PCreg( clk, reset, ~StallF, PCprime, PCF);
adder  #(8)      pcadd( PCF, 8'b100, PCPlus4);

// Decode Stage
assign RA1 = InstrD[5:3];
assign RA2 = InstrD[2:0];
assign WA4 = InstrD[2:0];
PipeD piped(    clk, StallD, FlushD, InstrF, InstrD);
mux2 #(3)      wa3mux(InstrD[5:3], InstrD[10:8], InstrD[15], WA3);
regfile        rf(clk, RegWriteAW, RegWriteBW,
                  RA1, RA2,
                  WA3W, WA4W,
                  ResultW, RD1W,
                  RD1, RD2);
mux2 #(8)      idmux({5'b0, InstrD[2:0]}, 8'b1, IDmuxD, idtemp);
mux2 #(8)      immmux(idtemp, InstrD[7:0], InstrD[15], ImmD);

// Execute Stage
PipeE pipee(    clk, FlushE,
                RA1, RA1E,
                RA2, RA2E,
                RD1, RD1E,
                RD2, RD2E,
                WA3, WA3E,
                WA4, WA4E,
                ImmD, ImmE,
                ImmSrcD, ImmSrcE,
                ALUControlD, ALUControlE,
                Flags_prim, FlagsE,
                FlagWriteD, FlagWriteE,
                RegWriteAD, RegWriteAE,
                RegWriteBD, RegWriteBE,
                MemtoRegD, MemtoRegE,
                ShowD, ShowE,
                InstrD, InstrE);

```

```

assign ShowDataE = RD1E;
mux5 #(8)   srcamux(RD1E, ALUResultM, ResultW, RD1M, RD1W, ForwardAE, SrcA);
mux5 #(8)   rd2mux(RD2E, ALUResultM, ResultW, RD1M, RD1W, ForwardBE, RD2temp)
;

mux2 #(8)   srcbmux(RD2temp, ImmE, ImmSrcE, SrcB);
alu         alu(SrcA, SrcB, ALUControlE, ALUResultE, ALUFlags);
assign RegWriteAE_temp = RegWriteAE & CondExE;
assign RegWriteBE_temp = RegWriteBE & CondExE;
assign JMuxTakenE = JMuxD & CondExE;

// Memory Stage
PipeM   pipem(clk, RD1E, RD1M,
              ALUResultE, ALUResultM,
              WA3E, WA3M,
              WA4E, WA4M,
              RegWriteAE_temp, RegWriteAM,
              RegWriteBE_temp, RegWriteBM,
              MemtoRegE, MemtoRegM);

// Writeback Stage
PipeW   pipew(clk, RD1M, RD1W,
              ALUResultM, ALUResultW,
              WA3M, WA3W,
              WA4M, WA4W,
              RegWriteAM, RegWriteAW,
              RegWriteBM, RegWriteBW,
              MemtoRegM, MemtoRegW,
              ReadDataM, ReadDataW);

mux2 #(8)   resmux(ALUResultW, ReadDataW, MemtoRegW, ResultW);

// Match Hazardunit Logic
always_comb begin
    Match_1E_M_3 = (RA1E == WA3M);
    Match_1E_W_3 = (RA1E == WA3W);
    Match_2E_M_3 = (RA2E == WA3M);
    Match_2E_W_3 = (RA2E == WA3W);
    Match_1E_M_4 = (RA1E == WA4M);
    Match_1E_W_4 = (RA1E == WA4W);
    Match_2E_M_4 = (RA2E == WA3M);
    Match_2E_W_4 = (RA2E == WA3W);
end
assign Match = {    Match_1E_M_3, Match_1E_W_3, Match_2E_M_3, Match_2E_W_3,
                    Match_1E_M_4, Match_1E_W_4, Match_2E_M_4, Match_2E_W_
4};

```

endmodule

### عملکرد:

منطق کد همچنان تغییری نمی‌کند و صرفاً سیگنال‌های قبلی به زیربخش‌های تقسیم می‌شوند و مانند قبل، به آخر اسم آن سیگنال‌ها اسم استیت مربوطه اضافه می‌شود. همچنین پایپ‌ها که جدا کننده استیج‌های مختلف هستند در این بخش تعریف می‌شود.

## ماژول Controller

```

module controller(  input logic          clk, reset,
                    input logic [15:0]  InstrD,
                    output logic         RegWriteAD, RegWriteBD, ImmSrcD,
                    output logic         IDMuxD, JMuxD,
                    output logic         MemtoRegD,
                    output logic [1:0]   ShowD,
                    output logic         FlagWriteD,
                    output logic [3:0]   ALUControlD);

    logic [7:0] controls;

    // Main Decoder
    always_comb
        casex(InstrD[15:6])
            10'b000000001: controls = 8'b10000000;
            10'b000000010: controls = 8'b10000000;
            10'b000000011: controls = 8'b10000000;
            10'b000000100: controls = 8'b10000000;
            10'b000000101: controls = 8'b10000000;
            10'b000000110: controls = 8'b10000000;
            10'b000000111: controls = 8'b11000000;
            10'b000001000: controls = 8'b10000000;
            10'b000001001: controls = 8'b10100000;
            10'b000001010: controls = 8'b10100000;
            10'b000001011: controls = 8'b10100000;
            10'b000001100: controls = 8'b10100000;
            10'b000001101: controls = 8'b10100000;
            10'b000001110: controls = 8'b10100000;
            10'b000001111: controls = 8'b10110000;
            10'b0000010000: controls = 8'b10110000;
            10'b0000000000: controls = 8'b00000000;
            10'b0000010010: controls = 8'b00000001;
            10'b0000010011: controls = 8'b00000010;
            10'b0000010100: controls = 8'b00000000;
            10'b10000?????: controls = 8'b00100100;
            10'b10001?????: controls = 8'b00100100;
            10'b10010?????: controls = 8'b00100100;
            10'b10011?????: controls = 8'b00100100;
            10'b10100?????: controls = 8'b00100100;
            10'b10101?????: controls = 8'b00100100;
            10'b10110?????: controls = 8'b10100000;
            10'b10111?????: controls = 8'b10101000;

```



```
        default:          controls = 8'bx;
    endcase

    assign {    RegWriteAD, RegWriteBD, ImmSrcD, IDMuxD, JMuxD,
                MemtoRegD, ShowD} = controls;

    always_comb
        if(InstrD[15]) begin
            ALUControlD = 4'b0101;
        end else begin
            casex(InstrD[10:6])
                5'b00000: ALUControlD = 4'b0000;
                5'b00001: ALUControlD = 4'b0000;
                5'b00010: ALUControlD = 4'b0001;
                5'b00011: ALUControlD = 4'b0010;
                5'b00100: ALUControlD = 4'b0011;
                5'b00101: ALUControlD = 4'b0100;
                5'b00110: ALUControlD = 4'b0101;
                5'b00111: ALUControlD = 4'b0110;
                5'b01000: ALUControlD = 4'b0111;
                5'b01001: ALUControlD = 4'b1000;
                5'b01010: ALUControlD = 4'b1001;
                5'b01011: ALUControlD = 4'b1010;
                5'b01100: ALUControlD = 4'b1011;
                5'b01101: ALUControlD = 4'b1100;
                5'b01110: ALUControlD = 4'b1101;
                5'b01111: ALUControlD = 4'b0000;
                5'b10000: ALUControlD = 4'b0010;
                5'b10100: ALUControlD = 4'b0010;
            default:      ALUControlD = 4'b0000;
        endcase
    end

    always_comb
        casex(InstrD[15:6])
            10'b0000000001: FlagWriteD = 1'b1;
            10'b0000000010: FlagWriteD = 1'b1;
            10'b0000000011: FlagWriteD = 1'b1;
            10'b0000000100: FlagWriteD = 1'b1;
            10'b0000000101: FlagWriteD = 1'b1;
            10'b0000000110: FlagWriteD = 1'b0;
            10'b0000000111: FlagWriteD = 1'b0;
            10'b0000001000: FlagWriteD = 1'b0;
            10'b0000001001: FlagWriteD = 1'b1;
            10'b0000001010: FlagWriteD = 1'b1;
```

```
10'b0000001011: FlagWriteD = 1'b1;
10'b0000001100: FlagWriteD = 1'b1;
10'b0000001101: FlagWriteD = 1'b1;
10'b0000001110: FlagWriteD = 1'b1;
10'b0000001111: FlagWriteD = 1'b1;
10'b0000010000: FlagWriteD = 1'b1;
10'b0000000000: FlagWriteD = 1'b0;
10'b0000010010: FlagWriteD = 1'b0;
10'b0000010011: FlagWriteD = 1'b0;
10'b0000010100: FlagWriteD = 1'b1;
default:          FlagWriteD = 1'bx;
endcase
endmodule
```

### عملکرد:

در این بخش دستورات ورودی برنامه مستقیماً decode می‌شوند. (برخلاف تک‌چرخه‌ای) و فقط سیگنال‌های استیج Decode در آن بررسی می‌شود.

## ماژول CondUnit

```

module condunit(    input logic          clk, reset,
                    input logic [15:0] InstrE,
                    input logic          FlagWriteE,
                    input logic [3:0]    FlagsE, ALUFlags,
                    output logic [3:0]    Flags_prim,
                    output logic          CondExE);

    logic CF, ZF, SF, OF;

    assign {CF, ZF, SF, OF} = FlagsE;

    always_comb
        if(FlagWriteE) Flags_prim = ALUFlags;
        else Flags_prim = FlagsE;

    always_comb
        casex(InstrE[15:11])
            5'b0????: CondExE = 1'b1;
            5'b10000: CondExE = ZF;
            5'b10001: CondExE = CF;
            5'b10010: CondExE = ~(CF | ZF);
            5'b10011: CondExE = (SF !== OF);
            5'b10100: CondExE = (SF === OF) & ~ZF;
            5'b10101: CondExE = 1'b1;
            5'b10110: CondExE = 1'b1;
            5'b10111: CondExE = 1'b1;
            default:    CondExE = 1'bx;
        endcase
endmodule

```

## عملکرد:

این بخش وظیفه تصمیم‌گیری اجرای دستورات و بررسی برآورده شدن دستورات را برعهده دارد. بدیهی است که این بخش باید در استیج Execute پردازنده قرار بگیرد.

## ماژول HazardUnit

```

module hazardunit(  input logic      clk, reset,
                    MemtoRegE, JMuxTakenE,
                    RegWriteAW, RegWriteAM,
                    RegWriteBW, RegWriteBM,
                    input logic [7:0] Match,
                    output logic[2:0] ForwardAE, ForwardBE,
                    output logic      FlushE, FlushD,
                    StallD, StallF);

  logic LDRstall;
  assign LDRstall = (!Match) & MemtoRegE;
  assign StallF = LDRstall;
  assign StallD = LDRstall;
  assign FlushE = LDRstall | JMuxTakenE;
  assign FlushD = JMuxTakenE;

  logic Match_1E_M_3, Match_1E_W_3, Match_1E_M_4, Match_1E_W_4,
        Match_2E_M_3, Match_2E_W_3, Match_2E_M_4, Match_2E_W_4;
  assign { Match_1E_M_3, Match_1E_W_3, Match_1E_M_4, Match_1E_W_4,
          Match_2E_M_3, Match_2E_W_3, Match_2E_M_4, Match_2E_W_4 } = Match;

  always_comb begin
    if(Match_1E_M_3 & RegWriteAM) begin
      ForwardAE = 3'b001; // SrcAE = ALUOutM
    end else if(Match_1E_W_3 & RegWriteAW) begin
      ForwardAE = 3'b010; // SrcAE = ResultW
    end else if(Match_1E_M_4 & RegWriteBM) begin
      ForwardAE = 3'b011; // SrcAE = RD1_M
    end else if(Match_1E_W_4 & RegWriteBW) begin
      ForwardAE = 3'b100; // SrcAE = RD1_W
    end else begin
      ForwardAE = 3'b000; //SrcAE form regfile
    end

    if(Match_2E_M_3 & RegWriteAM) begin
      ForwardBE = 3'b001; // SrcBE = ALUOutM
    end else if(Match_2E_W_3 & RegWriteAW) begin
      ForwardBE = 3'b010; // SrcBE = ResultW
    end else if(Match_2E_M_4 & RegWriteBM) begin
      ForwardBE = 3'b011; // SrcBE = RD1_M
    end else if(Match_2E_W_4 & RegWriteBW) begin
      ForwardBE = 3'b100; // SrcBE = RD1_W
    end else begin
      ForwardBE = 3'b000; //SrcBE form regfile
    end
  end

```

```
end  
end  
endmodule
```

### عملکرد:

عملیات جلوگیری از رخداد هازارد در این بخش صورت می‌گیرد.

به عبارت دیگر عملیات و زمان و نحوه اجرا دستورات stalling, forwarding بر عهده این بخش می‌باشد.

لازم به یادآوری است forwarding برای زمانی است که می‌خواهیم از مقدار یک رجیستر استفاده کنیم در صورتی که هنوز این مقدار در رجیستر فایل موجود نیست اما می‌توان مقدار آن را زودتر در مراحل Memory و Execute با این عملیات بدست آورد.

عملیات stall هم برای زمانی است که می‌خواهیم از دیتا مموری مقداری بخوانیم اما در این حالت forwarding جواب‌گو نیست و مجبوریم که دستورهای قبلی را متوقف کنیم تا مقدار درست محاسبه شود و ادامه کار انجام شود.

## ماژول PipeD

```
module PipeD(    input logic clk, StallD, FlushD,
                input logic [15:0] InstrF,
                output logic [15:0] InstrD);
    always_ff @(posedge clk)
        if (FlushD) InstrD <= 16'b0;    //NOP
        else if (~StallD) InstrD <= InstrF;
endmodule
```

### عملکرد:

پایپ حذف‌فاصل استیج Fetch و Decode  
اگر بیت Flush روشن باشد مقدار Instruction را برابر صفر قرار داده (NOP) که باعث ایجاد حباب و کاری نکردن مدار می‌شود.  
اگر بیت Flush خاموش باشد و از طرفی Stall هم نداشته باشیم داده می‌تواند به طور ایمن از Fetch به Decode برود.

## ماژول PipeE

```

module PipeE(    input logic clk, FlushE,
                input logic [2:0]  RA1,
                output logic [2:0]  RA1E,
                input logic [2:0]  RA2,
                output logic [2:0]  RA2E,
                input logic [7:0]  RD1,
                output logic [7:0]  RD1E,
                input logic [7:0]  RD2,
                output logic [7:0]  RD2E,
                input logic [2:0]  WA3,
                output logic [2:0]  WA3E,
                input logic [2:0]  WA4,
                output logic [2:0]  WA4E,
                input logic [7:0]  Imm,
                output logic [7:0]  ImmE,
                input logic        ImmSrcD,
                output logic        ImmSrcE,
                input logic [3:0]  ALUControlD,
                output logic [3:0]  ALUControlE,
                input logic [3:0]  Flags_prim,
                output logic [3:0]  FlagsE,
                input logic        FlagWriteD,
                output logic        FlagWriteE,
                input logic        RegWriteAD,
                output logic        RegWriteAE,
                input logic        RegWriteBD,
                output logic        RegWriteBE,
                input logic        MemtoRegD,
                output logic        MemtoRegE,
                input logic [1:0]  ShowD,
                output logic [1:0]  ShowE,
                input logic [15:0] InstrD,
                output logic [15:0] InstrE);

    logic[50:0] inputs, outputs;
    assign inputs = {RA1, RA2, RD1, RD2, WA3, WA4, Imm, ImmSrcD, ALUControlD,
Flags_prim, FlagWriteD, RegWriteAD, RegWriteBD, MemtoRegD, ShowD};
    assign {RA1E, RA2E, RD1E, RD2E, WA3E, WA4E, ImmE, ImmSrcE, ALUControlE, Flags
E, FlagWriteE, RegWriteAE, RegWriteBE, MemtoRegE, ShowE} = outputs;
    always_ff @(posedge clk) begin
        if (FlushE) {outputs, InstrE} <= {inputs, 16'b0};    //NOP
        else        {outputs, InstrE} <= {inputs, InstrD};
    end
endmodule

```

آزمایشگاه معماری و مدارهای منطقی - پروژه پنجم

متین زیودار - محمد خدام

### عملکرد:

پایپ حذف‌افصل استیج Decode و Execute

عملکرد مانند مرحله قبل است با این تفاوت که بیت‌های بیشتری به استیج بعدی می‌روند و دیگر بیت Stallی نداریم.



## ماژول PipeM

```
module PipeM(    input logic clk,
                input logic [7:0] RD1E,
                output logic [7:0] RD1M,
                input logic [7:0] ALUResultE,
                output logic [7:0] ALUResultM,
                input logic [2:0] WA3E,
                output logic [2:0] WA3M,
                input logic [2:0] WA4E,
                output logic [2:0] WA4M,
                input logic RegWriteAE_temp,
                output logic RegWriteAM,
                input logic RegWriteBE_temp,
                output logic RegWriteBM,
                input logic MemtoRegE,
                output logic MemtoRegM);

    logic [24:0] inputs, outputs;
    assign inputs = {RD1E, ALUResultE, WA3E, WA4E, RegWriteAE_temp, RegWriteBE_temp, MemtoRegE};
    assign {RD1M, ALUResultM, WA3M, WA4M, RegWriteAM, RegWriteBM, MemtoRegM} = outputs;
    always_ff @(posedge clk)
        outputs <= inputs;
endmodule
```

## عملکرد:

پایپ حذف‌افصل استیج Fetch و Memory

همانند مرحله قبلیست فقط دیگر بیت Flush را هم ندارد و داده‌ها به طور ایمن به مرحله بعد می‌روند.

## ماژول PipeW

```
module PipeW ( input logic clk,
               input logic [7:0] RD1M,
               output logic [7:0] RD1W,
               input logic [7:0] ALUResultM,
               output logic [7:0] ALUResultW,
               input logic [2:0] WA3M,
               output logic [2:0] WA3W,
               input logic [2:0] WA4M,
               output logic [2:0] WA4W,
               input logic RegWriteAM,
               output logic RegWriteAW,
               input logic RegWriteBM,
               output logic RegWriteBW,
               input logic MemtoRegM,
               output logic MemtoRegW,
               input logic [7:0] ReadDataM,
               output logic [7:0] ReadDataW);
    logic [32:0] inputs, outputs;
    assign inputs = {RD1M, ALUResultM, WA3M, WA4M, RegWriteAM, RegWriteBM, MemtoRegM, ReadDataM};
    assign {RD1W, ALUResultW, WA3W, WA4W, RegWriteAW, RegWriteBW, MemtoRegW, ReadDataW} = outputs;
    always_ff @(posedge clk)
        outputs <= inputs;
endmodule
```

## عملکرد:

پایپ حذف فاصل استیج Memory و WriteBack

مثل مرحله قبل

## تست و راستی‌آزمایی

### کد اسمبلی دستورات

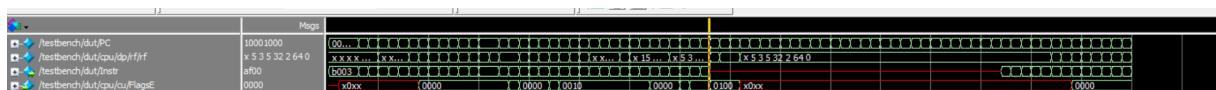
```
LI R0 #3 //R0 = 3
LI R1 #2 //R1 = 2
LI R2 #1 //R2 = 1
LI R3 #4 //R3 = 4
OR R1 R0 //R1 = 3
XOR R2 R3 //R2 = 5
MOV R5 R2 //R5 = 5
ADD R1 R2 //R1 = 8
SUB R2 R1 //R2 = -3
XCHG R5 R0 // R0 = 5 R5 = 3
NOT R2 // R2 = 11111101 R2 = 00000010 = 2
JL to start //not accept
SAR R0 #2 //R0 = 1
SLR R3 #1 //R3 = 2
SAL R3 #4 //R3 = 32
SLL R1 #3 //R1 = 64
DEC R0 #1 // R0 = 0
DEC R0 #1 // R0 = -1
INC R0 #1 // R0 = 0
NOP
CMP R0 R1 //R0-R1 = 0-64=-64
LI R4 #5 // R4 = 5
LI R6 #15 // R6 = 15
ShowR R4
JA to start //not accept
ShowR R6
AND R6 R4 // R6 = 5
ShowRseg R4
ShowRseg R6
CMP R6 R4
ROL R4 #3 //Change
ROR R4 #3 //Revert Change
CMP R6 R4
JE to JMP0 //accept so R7 doesn't affect
LI R7 #7
JMP to 0
```

### کد زبان ماشین دستورات (هگزادسیمال)

```
B003
B102
B201
B304
0108
0113
01AA
004A
```

00D1  
01E8  
0217  
9D00  
0242  
0299  
02DC  
030B  
0405  
0403  
03C7  
0033  
0501  
B405  
B60F  
9700  
04A0  
04B0  
00B4  
04E0  
04F7  
0534  
0363  
03A3  
0534  
858C  
B707  
AF00

## یایپ لاین:



با بررسی تک تک اینستراکشن‌ها با خروجی مربوطه به این نتیجه می‌رسیم که پردازنده به درستی کار می‌کند.

## تک چرخه‌ای:



باز هم با مقایسه حالت‌های نهایی این دو پردازنده مشاهده می‌شود که نمودارها برابرند که این حاکی از درست کار کردن پردازنده‌هاست.

## سنتز پردازنده

Flow Summary	
<<Filter>>	
Flow Status	Successful - Fri Jun 19 23:34:04 2020
Quartus Prime Version	19.1.0 Build 670 09/22/2019 SJ Lite Edition
Revision Name	Pipeline
Top-level Entity Name	top
Family	Cyclone IV E
Total logic elements	556 / 6,272 ( 9 % )
Total registers	182
Total pins	18 / 92 ( 20 % )
Total virtual pins	0
Total memory bits	0 / 276,480 ( 0 % )
Embedded Multiplier 9-bit elements	0 / 30 ( 0 % )
Total PLLs	0 / 2 ( 0 % )
Device	EP4CE6E22C6
Timing Models	Final

Post-Synthesis Netlist Statistics for Top Partition		
<<Filter>>		
	Type	Count
1	boundary_port	18
2	▼ cycloneiii_ff	182
1	ENA	77
2	ENA CLR SLD	6
3	ENA SLD	1
4	SCLR	2
5	SLD	9
6	plain	87
3	▼ cycloneiii_lcell_comb	549
1	▼ arith	13
1	2 data inputs	6
2	3 data inputs	7
2	▼ normal	536
1	0 data inputs	1
2	1 data inputs	4
3	2 data inputs	44
4	3 data inputs	98
5	4 data inputs	389
4		
5	Max LUT depth	15.00
6	Average LUT depth	7.35


تعداد کلیه عناصر پردازنده ۶۲۷۲ تا است که حدود ۹ درصد آن یعنی ۵۵۶ تا عنصر منطقی دارد. همچنین المان‌های اصلی مدار شامل ۱ صفر ورودی و ۴ یک ورودی و ۵۰ دو ورودی و ۱۰۵ سه ورودی و ۳۸۹ چهار ورودی می‌باشد.

### بررسی زمان و مسیرهای بحرانی

Slow 1200mV 85C Model			
Command Info		Summary of Paths	
	Delay	From Node	To Node
1	8.397	cpu:cpu datapath:dp PipeE:pipee outputs[23]	cpu:cpu data...e outputs[8]

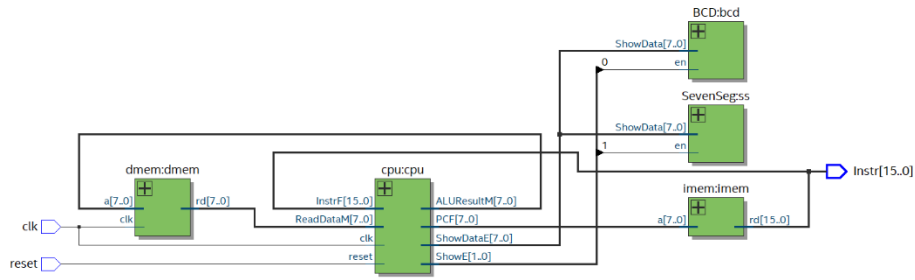
مسیر بحرانی در شکل بالا مشخص شده است و همچنین زمان تاخیر برای آن ۸/۳۹۷ ثانیه می باشد.

### بررسی بیشینه فرکانس پردازنده

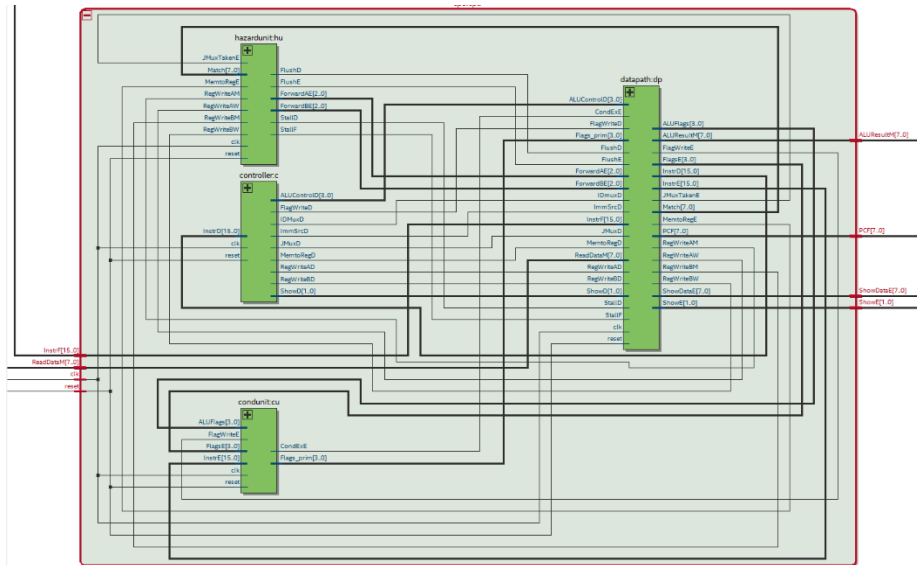
Slow 1200mV 85C Model Fmax Summary				
 <<Filter>>				
	Fmax	Restricted Fmax	Clock Name	Note
1	115.25 MHz	115.25 MHz	clk	

این پردازنده در دمای ۸۵ درجه سانتی گراد و ولتاژ ۱۲۰۰ میلی ولت، دارای فرکانس بیشینه ۱۱۵/۲۵ مگاهرتز می باشد.

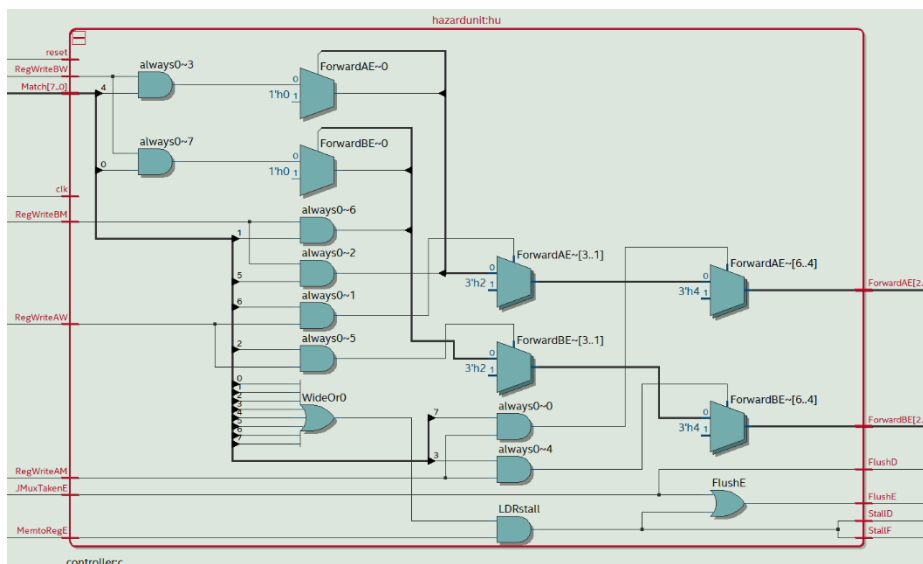
## RTL VIEW



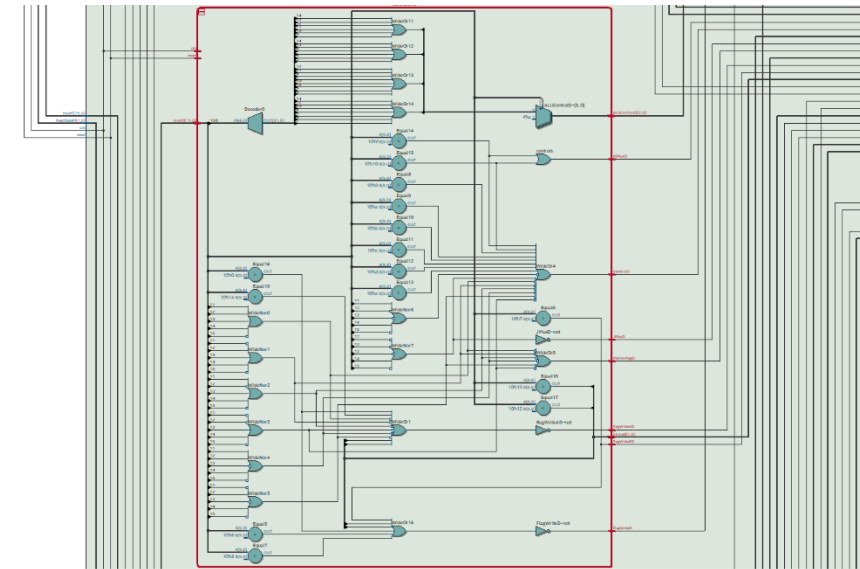
A.Top



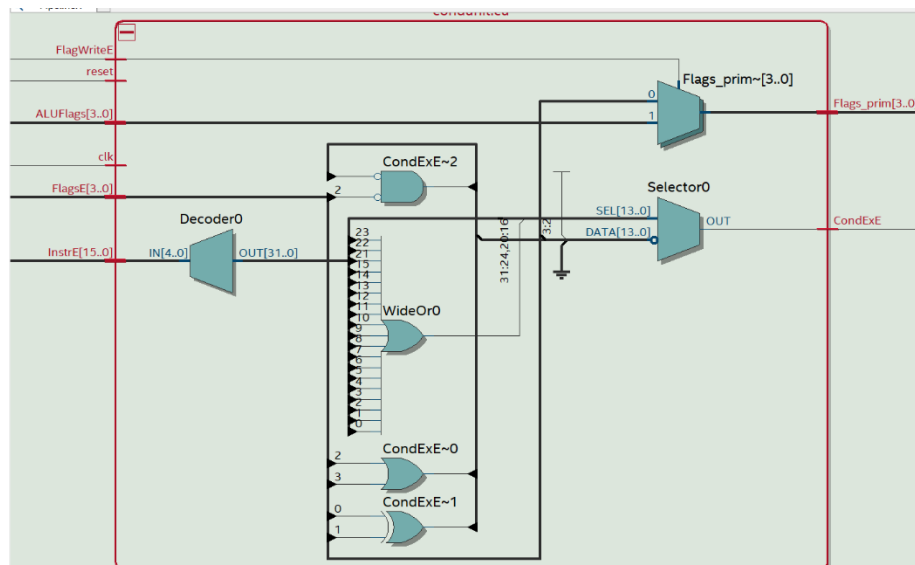
B.CPU



C.HazardUnit

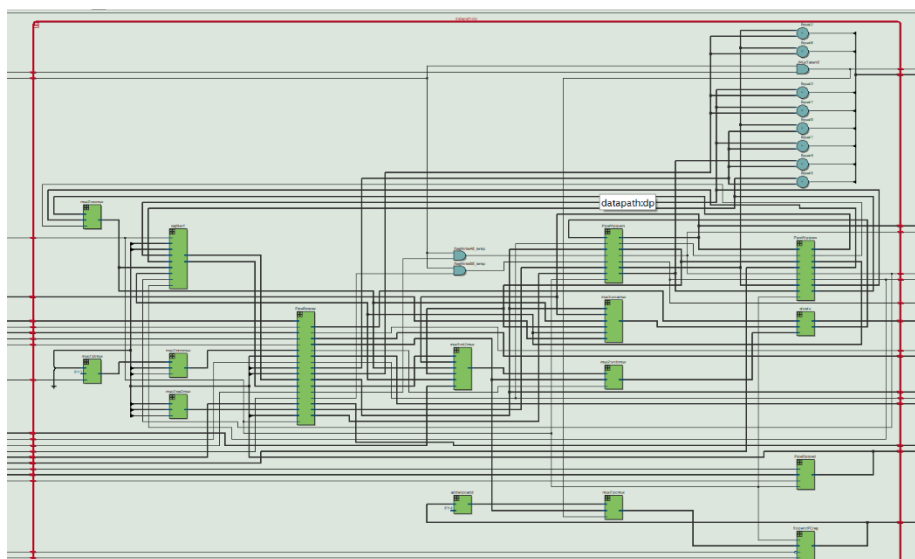


D.Controller



E.CondUnit





*F.Datapath*