به نام خدا

پروژه: ***طراحی مدار مالتی سایکل (MultiCycle) پردازنده MIPS***

طراحان:

آریا ابریشم دار **میلاد بارونی**

در این پروژه یک مدار مالتی سایکل پردازنده MIPS با استفاده از زبان توصیف سخت افزار verilog طراحی شده است. که دارای سه پورت ورودی مجزا از پردازنده با نام ها INM، INMD و NMI هستند که توضیحات هرکدام در شرح پروژه موجود است. مداری که باید به صورت ماژولار طراحی می شد به شکل زیر است:

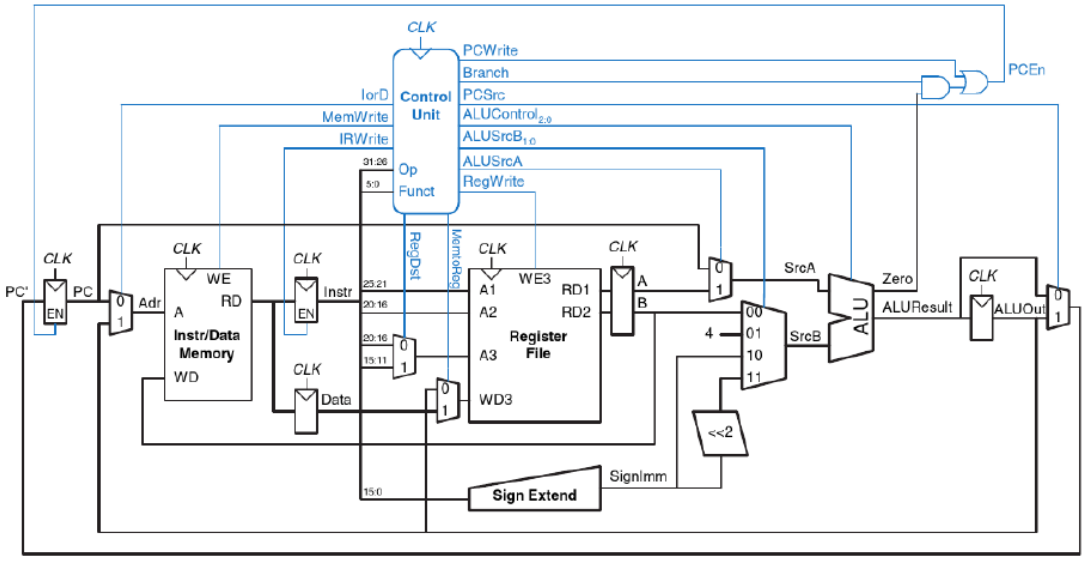


Figure 1: MIPS data-path

که هرکدام از ماژول ها در یک فایل جدای وریلاگ با پسوند v. موجود است. ماژول هایی که برای این مدار در نظر گرفته شده اند عبارتند از :

pc, Memmux, Memory, IR, WriteDate, Dregmux, MemtoRegMux, Regfile, SignEx, RegFileOut, Controller,

SrcAmux, SrcBmux, ALU, ALUcontroller, ALUout, ALUoutMux, Initializer

که در ادامه به توضیح در مورد هر ماژول و ماژول کلی یعنی DataPath پرداخته شده است.

در ماژول DataPath سرهم بندی تک تک ماژول ها به صورت جدا جدا صورت گرفته است. قابل ذکر است که ماژول ها در این ماژول اصلی به صورت سنکرون و یا همزمان اجرا میشوند و اجرای آنها در مرحله های بعد وابسته به خروجی مرحله های قبل ماژول های قبل است.

۱- ماژول SignEx:

میدانیم که برای برخی از دستورات معماری MIPS ما نیازمند یک تبدیل ۱۶ به ۳۲ بیتی هستیم که بتوانیم جمع و یا تفریق و یا هر عمل دیگر ریاضیاتی و یا محاسباتی را روی آن انجام دهیم بنابراین این ماژول با Syntax خاص زبان وریلاگ به شکل زیر تعریف شده است :

```
module SignEx(
        input [15:0]wirein,
        output [31:0]wireout
        );
assign
wireout={{16{wirein[15]}},wirein};
endmodule
```

که در اینصورت مشاهده میشود که یک ورودی ۱۶ بیتی به یکی خروجی ۳۲ تبدیل شده است.

۲- ماژول PC :

به این صورت که جز ماژول های کلاک دار است. یک ورودی دارد که یک کلاک بعد همان ورودی را در خروجی ظاهر میکند. تمامی پنج ماژول بعدی از همین نوع هستند:

```
module PC(
   input PCEn,
   input [31:0] PC_in,
   input clk,
   output reg [31:0] PC_out
);

initial begin
PC_out <= 128;
end

always @(posedge clk)begin

  if (PCEn )begin
   PC_out <= PC_in ;
  end

end
```

این ماژول علاوه بر حساسیت به کلاک به PCEn هم حساس است که توسط Controller تعیین میشود.

ماژول Controller در ادامه توضیح داده شده است.

۳- ماژول IR :

باز هم یک ورودی را یک کلاک بعد به خروجی میدهد و باز هم یک پورت ورودی دیگر به نمام IRWrite دارد که توسط controller مشخص میشود:

```
module IR(
   input IRWrite,
   input clk,
   input [31:0] Inst,
   output reg [31:0] Out
);
always @(posedge clk)begin
  if (IRWrite == 1'b1)begin
    Out <= Inst;
  end
 end
endmodule
```

۴- ماژول WriteData:

این ماژول نیازی به ورودی از Cotroller ندارد و صرفا توسط clock که با پورت clk مشخص شده کنترل میشود:

```
module WriteData(
        input clk,
        input [31:0] read_data,
        output reg [31:0] data

    );

        always @(posedge clk)
        begin
        data <= read_data;
        end

endmodule
```

۵- ماژول ALUout:

مثل ماژول قبلی صرفا یکی ورودی را بر حسب کلاک به خروجی منتقل میکند.

```
module ALUout(
   input clk,
   input [31:0]ALUResult,
   output reg [31:0] aluout
);
always @(posedge clk)begin
  aluout <= ALUResult;
end

endmodule
```

۶- ماژول RegfileOut :

این ماژول برخلاف ماژول های کلاکی قبل بر اساس دو ورودی و دو خروجی کار میکند. به این صورت که دو ورودی آن خروجی های RegFile به عنوان یک ماژول است و خروجی آن نیز همون ورودی ها پس از یک کلاک است

```
module RegfileOut(
    input [31:0] A,
    input [31:0] B,
    input clk,
    output reg [31:0] A_out,
    output reg [31:0] B_out
);
always @(clk)begin
    A_out <= A;
    B_out <= B;
end

endmodule
```

در ادامه به توضیح تمامی ماژول های انتخاب گر و یا مالتی پلکسر میپردازیم. تعداد این ماژول ها شش ماژول است که همگی نحوه‌ی کارکرد یکسانی دارند متنهی به این شکل که نحوه انتخاب آنها و یا عمل انتخاب یا عدم انتخاب آنها در هرکدام متفاوت است.

۷- ماژول Memmux :

این ماژول ورودی ماژول Memory را مشخص میکند که میتواند pc و یا Aluout باشد.

```
module MemMux(
    input [31:0] PC_out,
    input [31:0] ALUOut,
    input IorD,
    output reg [31:0] out
);

always @(PC_out, ALUOut, IorD)begin
  if(IorD)begin
    out <= ALUOut;
  end
  else
    out <= PC_out;
  end
endmodule
```

۸- ماژول Dregmux :

که در واقع ورودی ریجستری که داده‌ای باید در آن نوشته شود را مشخص میکند.

```
module DRegmux(
    input [31:0] r1,
    input [31:0] r2,
    input RegDst,
    output reg [31:0] out
);
always @(r1, r2, RegDst)begin
  if (RegDst)begin
    out <= r2;
  end
  else
    out <= r1;
end

endmodule
```

۹- ماژول MemtoRegmux:

که این ماژول داده ای را که باید در یک ریجیستر نوشته شود را مشخص میکند که ممکن است از خود دستور بیاید و یا ALUresult باشد.

```
module MemtoRegMux(
    input [31:0] wb,
    input [31:0] data,
    input MemtoReg,
    output reg [31:0] out
);
always @(wb, data,
MemtoReg)begin
  if (MemtoReg)begin
    out <= data;
  end
  else
    out <= wb;
end
endmodule
```

۱۰- ماژول SrcAmux :

که بین pc و خروجی regfile یکی را بر اساس مقادیر کنترولر مشخص میکند.

```
module SrcAMux(
   input [31:0] A,
   input [31:0] Pc,
   input ALUSrcA,
   output reg [31:0] muxout
);
always @(A, Pc, ALUSrcA)begin
 if (ALUSrcA)begin
   muxout <= A;
 end
 else
   muxout <= Pc;
end
endmodule
```

۱۱- ماژول SrcBmux :

که بین اکستند شده ۳۲ بیتی و یا همان عدد ضرب در چهار و یا عدد چهار و یا خروجی regfile یکی را مشخص میکند. خط انتخاب دوبیتی آن نیز از ماژول کنترلر می آید.

```
module SrcBMux(
   input [31:0] B,
   input [31:0] exsign,
   input [1:0] ALUSrcB,
   output reg  [31:0] muxout
);
always @(B, exsign, ALUSrcB)begin
   if (ALUSrcB == 2'b00)begin
      muxout <= B;
   end
   if (ALUSrcB == 2'b01)begin
      muxout <= 4;
   end
   if (ALUSrcB == 2'b10)begin
      muxout <= exsign;
   end
   if (ALUSrcB == 2'b11)begin
      muxout <= exsign << 2;
   end
end
endmodule
```

۱۲- ماژول ALUoutmux :

که در واقع مقدار پی سی بعد و یا مقداری که باید در memory ذخیره شود را مشخص میکند.

```
module ALUOutMux(
    input [31:0] ALUout,
    input [31:0] ALUResutl,
    input PCSrc,
    output reg [31:0] out
);


always @(ALUout, ALUResutl,
PCSrc)begin
  if (!PCSrc)begin
    out <= ALUResutl;
  end
  else
    out <= ALUout;
end
endmodule
```

اما ماژول های اصلی یعنی regfile, memory, alu, alucontrol, contorller پیاده سازی پیچیده تری نسبت به ماژول های قبلی دارند که در پایین به توضیح کلی در مورد آنها بسنده شده است.

۱۳- ماژول regfile :

این ماژول به مانند یک حافظه است. اما نوع ورودی و خروجی گرفتن آن نسبت به memory متفاوت است. به این صورت که در این ماژول ما دو آدرس میگیریم و دو خروجی را نمایش میدهیم. همزمان میتوانیم با گرفتن یک آدرس ریجستر و یک داده در یک Halfclock مقدار آن داده را در ریجستر ذخیره سازی کنیم(اگر پورت RegWrite این را تایید کند). به این ترتیب ما ماژول را به شکل زیر تعریف کرده‌ایم.

```
module Regfile(
    input clk,
    input RegWrite,
    input [4:0]A1,
    input [4:0]A2,
    input [4:0]A3,
    input [31:0]WD3,
    output reg [31:0] RD1, RD2
);
    // defining the register file;
    reg [31:0] registers[31:0];

            initial begin
           registers[0] = 0;
            end
    always @(posedge clk)
        if(RegWrite)
            registers[A3] <= WD3;
        else
        begin
            RD1 <= registers[A1];
            RD2 <= registers[A2];
        end
endmodule
```

یک حافظه ریجستری ۳۲ تایی از ریجستر هایی ۳۲ بیتی

نوشتن در یک ریجیستر

خواندن از یک ریجیستر

۱۴- ماژول Memory :

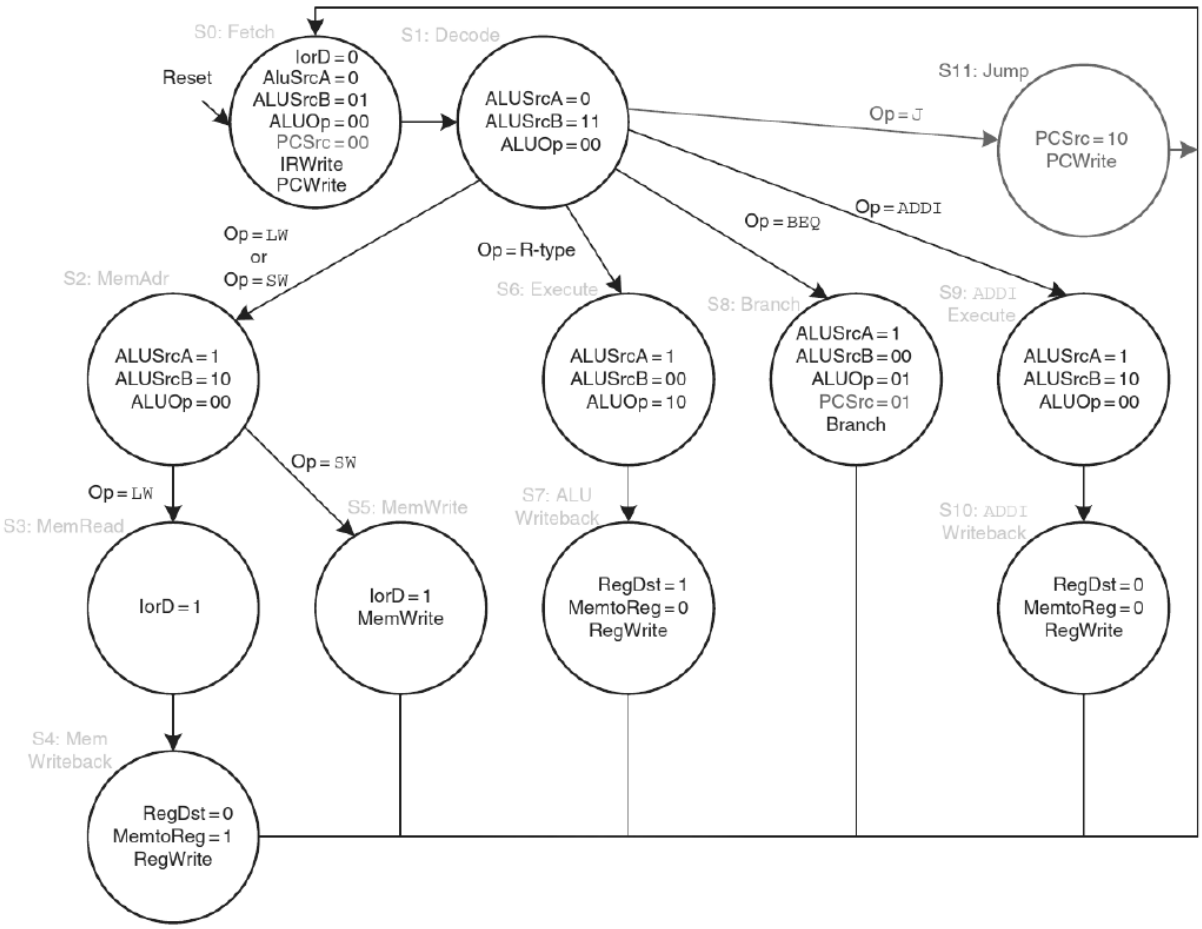
بیشترین شباهت را با ماژول regfile دارد با این تفاوت که در تعریف حافظه با آن متفاوت است و همچنین یکسری مقادیر به صورت پیشفرض در آن قرار گرفته شده‌اند که میتوان از آنها در مقدار دهی و یا اجرای کردن دستورات اسفاده کرد. در اینجا آدرس شروع دستورات ۱۲۸ و آدرس شروع مموری داده ها ۱ در نظر گرفته شده است.

```
module Memory (
input wire [31:0] addr,          // Memory Address
input wire [31:0] write_data,    // Memory Address
Contents
input wire MemWrite,
input wire clk,                  // All synchronous elements,
including memories, should have a clock signal
output reg [31:0] read_data      // Output of Memory
Address Contents
);
// a 512 two word memory
reg [31:0] MEMO[511:0];

initial begin
  MEMO [0] <= 32'd8;
  MEMO [1] <= 32'd1;
  MEMO [2] <= 32'd1;

  MEMO[128] <= 32'h8c030000;
  MEMO[132] <= 32'h8c040001;
  MEMO[136] <= 32'h8c050002;
  MEMO[140] <= 32'h8c010002;
  MEMO[144] <= 32'h10600004;
  MEMO[148] <= 32'h00852020;
  MEMO[152] <= 32'h00852822;
  MEMO[156] <= 32'h00611820;
  MEMO[160] <= 32'h10000002;
  MEMO[164] <= 32'h00852020;
  MEMO[168] <= 32'hac040006;
  MEMO[172] <= 32'h0064402A;
  MEMO[176] <= 32'h00852020;

end

always @(*) begin
  if (MemWrite == 1'b1) begin
    MEMO[addr] <= write_data;
  end
  else begin
    read_data <= MEMO[addr];
  end
end

endmodule
```

مقادیر پیشفرض

۱۵- ماژول Controller :

برای طراحی ماژول Controller **از یک ماشین با تعداد حالت های محدود و یا** FSM **استفاده کردیم که در زیر آورده شده است:**



که در ماژول کنترولری که نوشته شده است از تمامی دستورات کار با مموری (lw, sw) و دستورات r-type **که شامل** less-than **ها نیز هست پشتیبانی میکند.**

این FSM **در واقع دارد یکسری** state **هایی را مشخص میکند که در آنها یکسری مقادیر در مدار باید تغییر کند که وظیفه‌ی کنترولر را بیان میکند.**

کد وریلاگ آن که در پروژه وجود دارد به این صورت است که ورودی های آن علاوه بر clk تمام اینتراپت هایی هستند که باید اعمال شوند. فرآیند به این شکل است که تمامی interrupt ها در فاز fetch بررسی میشوند اگر وجود داشتند ما به یک فاز دیگر میبرند که بستگی به نوع اینتراپت باید متفاوت باشد. این به مانند این است که به این FSM یک سری state های جدید اضافه شده است. این ماژول در کل یک سری خطوط کنترلی برای مدار تولید میکند . این کنترلر یک opcode را برای ما تولید میکند که آن را به ماژول ALUControl میدهیم.

۱۶- ماژول ALUControl :

در این ماژول ما funct و opcode را به تابع میدهیم و این تابع یک operation برای ما تولید میکند که آن را به ماژول ALU میدهیم. میتوانستیم این کار را در خود ماژول ALU هم انجام دهیم اما برای اینکه قسمت کنترلی آن را از ماژول جدا کنیم این کار را انجام دادیم. چون در واقع در معماری mips این دو از هم جدا هستند. برای پیدا کردن operation ها از کتاب MIPS رفرنس استفاده شده است. با توجه به opcode تولید شده از کنترلر و همچنین funct مربوط به instruction.

۱۷- ماژول ALU :

این ماژول operation ها را از ALUControl میگیرد و روی SrcA و SrcB یک سری اعمالی انجام میدهد. با توجه به اینکه operation چه چیزی تعریف شده باشد.