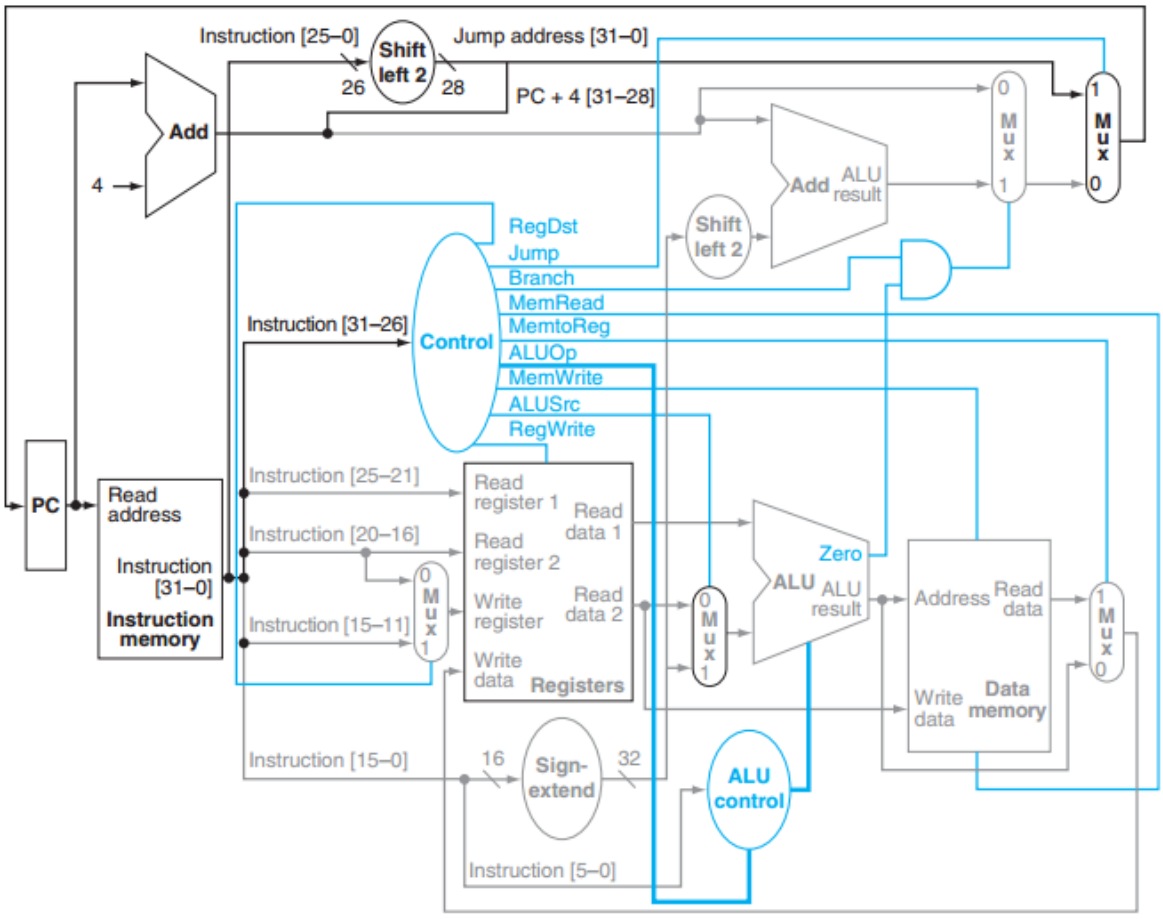
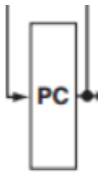
طراحی و پیاده سازی یک پردازنده سینگل سایکل :

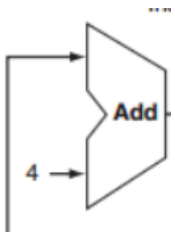
در این گزارش قصد داریم تا یک پردازنده مشابه MIPS طراحی کنیم که تقریبا مشابه MIPS است اما پردازنده بجای 32 بیت ، محاسبات 16 بیتی را انجام می دهد و همچنین طول حافظه 8 بیتی می باشد ، بنابراین داده ها و دستور العمل های ما در دو خونه از حافظه جای میگیرند. شکل کلی بخش های مختلف آن را در زیر می توانید ببینید :



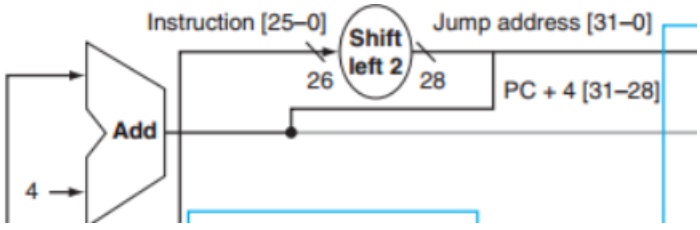
حال به تعریف اجمالی از بخش های مختلف این CPU می پردازیم :



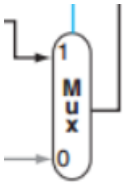
رجیستر PC به آدرس دستور بعدی اشاره می کند و بعد از اجرای هر دستور 2 واحد به آن اضافه می شود (هر دستور 16 بیتی هستش و در دو خونه حافظه جا میشه بنابراین باید هر بار دو واحد به آدرسی که PC بهش اشاره می کنه اضافه کنیم)



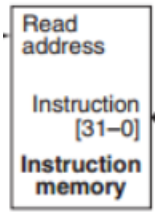
عمل جمع توسط یک جمع کننده انجام می شود ، البته در تصویر جمع کننده با 4 رو میبینم ولی ما قصد داریم با جمع کنند عدد 2 پیاده سازی رو انجام بدیم .



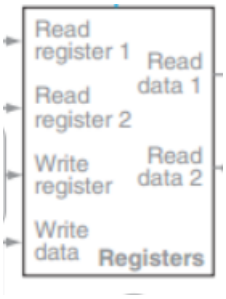
واحد شیفت برای دستور های پرشی و جامپ کاربرد دارد که نیاز است عدد مورد در 4 ضرب شود که این کار با دوبار شیفت به چپ انجام می شود .



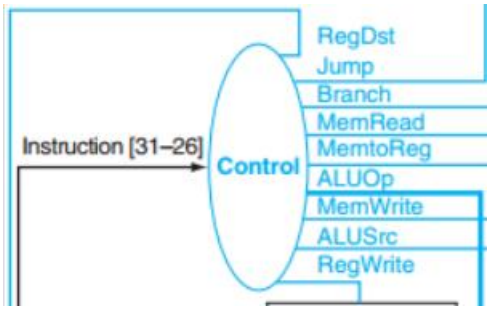
یک مالتی پلکسر که توسط واحد کنترل ، کنترل می شود و تعیین می کند آدرس PC به عنوان دستور بعدی اجرا شود و یا آدرس دستور های پرشی و جامپ .



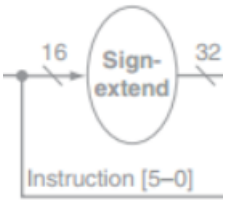
یک رجیستر 16 بیتی که دستورالعمل Fetch شده از حافظه را در خود نگه میدارد تا بخش های مختلف دستورالعمل به واحد های مختلف ارسال شود و دستور بعد از دیکُد شدن اجرا شود .



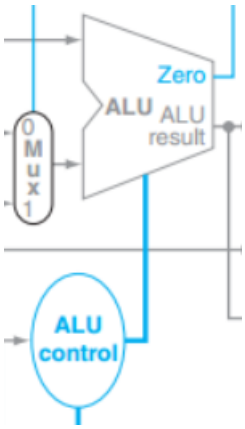
یک رجیستر فایل که رجیستر های مختلفی در آن قرارداده شده تا در انجام محاسبات و اجرای دستورات از آنها استفاده کنیم .



واحد کنترل مغز متفکر سیستم ماست که قرار است به کمک دستورالعملی که بصورت ورودی دریافت میکند سیگنال های خروجی کنترلی را صادر کند . این سیگنال ها به بخش های مختلفی ورود پیدا کرده و مشخص میکنند هر قسمت به چه صورت و با چه ترتیبی کار کند .

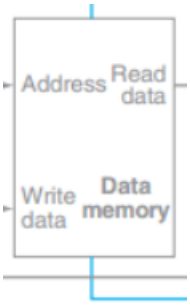


واحد توسعه کمک میکند یک عدد را با حفظ ارزش اش ، به بیت های بیشتری توسعه دهیم ، که در پروژه ما یک عدد هشت بیتی را با حفظ ارزش به یک عدد 16 بیتی تبدیل میکند.



واحد ALU Control مشخص میکند که چه عملی روی داده های ورودی ALU باید صورت گیرد (جمع ؟ تفریق ؟ ...) . خود واحد ALU نیز دو ورودی گرفته و طبق مقدار سیگنالی که از ALU Contorl می آید ، پردازش موردنظر را روی دو داده ورودی انجام می دهد .

نکته : خود واحد ALU Contorl نیز توسط یک ورودی که از سمت Control Unit می آید کنترل می شود .



واحد حافظه که می تواند تعدادی رجیستر در کنار پردازنده باشد ، برنامه های اسمبلی (کد ماشین) در این حافظه نوشته می شود ، البته کاربرد دیگه حافظه نوشتن و خواندن در آن است که می توان اعدادی رو در آن نوشت و از آن خواند.

با بررسی بخش های مختلف سخت افزاری پردازنده ، در قسمت بعدی گزارشکار به بررسی کد های وریلاگ میپردازیم .

```
1  `timescale 1ns / 1ps
2
3  module Sign_Ex(
4      input  [5:0] in6,
5      output reg [15:0] out16
6      );
7        always @(*)
8        begin
9        if (in6[5]==1)
10          out16={16'b1111111111,in6};
11          else
12          out16={16'b0000000000,in6};
13       end
14
15 endmodule
16
```

ماژول بخش توسعه که عدد شش بیتی مورد نیاز را گرفته و به عددی 16 بیتی تبدیل می کند .

```
13       always @(*) begin
14          if (inp1 > inp2) begin
15          Zero=0;
16          Gt=1;
17          Lt=0;
18          end
19          else if (inp2 > inp1)begin
20          Zero=0;
21          Gt=0;
22          Lt=1;
23          end
24          else begin
25          Zero=1;
26          Gt=0;
27          Lt=0;
28          end
29
30
31          case(ControlALU)
32          4'b0000 : Result=inp1+inp2;
33          4'b0001 : Result=inp1-inp2;
34          4'b0010 : Result=inp1&inp2;
35          4'b0011 : Result=inp1|inp2;
36          4'b0100 : Result=inp1^inp2;
37        //4'b0111 : Result= set on less than;
38          4'b0101 : Result=~(inp1|inp2);
39          endcase
40
41
42       end
```

ماژول بالا ، مربوط به واحد ALU می باشد که با توجه به سیگنال کنترلی اش یکی از اعمال جمع تفریق ، اند و اور یا ایکسور و یا نور را انجام می دهد .

```
3  module ControlUnit(
4      input [15:0] Instr,
5      input Gt,
6      input Lt,
7      input Zero,
8      output reg Branch,
9      output reg RegDest,
10     output reg MemRd,
11     output reg MemWr,
12     output reg [3:0] ALUOp,
13     output reg ALUSrc,
14     output reg MemReg,
15     output reg Jum,
16     output reg RegWr
17     );
```

واحد کنترل به دلیل طولانی بودن حجم کد آن ، فقط ورودی و خروجی های آن را اینجا میبینیم ، تمام سیگنال های خروجی دارای اهمیت ویژه ای هستند که باید بصورت صحیح در این ماژول مقداردهی شوند .

```
39 reg [15:0] PC=0;
40 reg [15:0] K=0;
41 reg [15:0] Instr;
42 wire [15:0] OutSign;
43 wire Branch,RegDest,MemRd,MemWr,ALUSrc,MemReg,RegWr,Jum,Gt,Lt;
44 wire [3:0] ALUOp;
45 wire Zero;
46 wire [15:0] Result;
47
48 reg [2:0] R1;
49 reg [2:0] R2;
50 reg [2:0] WR; // RD
51 reg [15:0] WrightData; //writer reg >>> write data
52 reg [15:0] Data1;
53 reg [15:0] Data2;
54 reg [15:0] Data2ALU;
55
```

حال نوبت به ماژول اصلی پردازنده می رسد ، تمام رجیستر های مختلفی که به آنها نیاز داریم تعریف شدند (توجه : کد پردازنده را بصورت رفتاری نوشته ایم) .

```
60
61      Sign_Ex U1 (         //Instanc SignExt (ok)
62       .in6(Instr[5:0]),
63       .out16(OutSign)
64       );
65
66
67       ControlUnit U2(    //Instanc Control
68       .Instr(Instr),
69       .Zero(Zero),
70       .Gt(Gt),
71       .Lt(Lt),
72       .Branch(Branch),
73       .RegDest(RegDest),
74       .MemRd(MemRd),
75       .MemWr(MemWr),
76       .ALUOp(ALUOp),
77       .ALUSrc(ALUSrc),
78       .MemReg(MemReg),
79       .Jum(Jum),
80       .RegWr(RegWr)
81       );
82
83       ALU U3 (     //Instanc ALU
84       .inp1(Data1),
85       .inp2(Data2ALU),
86       .Result(Result),
87       .ControlALU(ALUOp),
88       .Gt(Gt),
89       .Lt(Lt),
90       .Zero(Zero)
91       );
92
```

هر سه واحدی که در صفحه قبل معرفی شدند به صورت ماژول مجزا ، درون ماژول اصلی پردازنده فراخوانی می شوند و سیگنال های مرتبط با هر ماژول به سیگنال های درونی ماژول مذکور وصل می شود تا سه واحد ALU و SignEx و Contorl را در اختیار داشته باشیم و بتوانیم با آنها ارتباط برقرار کنیم .

```
116
117     if (MemRd==1 && MemReg==0) //Load
118           WrightData=Result;
119           else if (MemRd==1 && MemReg==1)
120              WrightData={MEM[Result+1],MEM[Result]};
121
122
123
124     if (RegWr==1 && MemReg==0) //R-Typ
125           WrightData=Result;
126
127
128     if (MemWr==1 && MemReg==0) //Store
129           MEM[Result]=Data2;
130
131     if (RegWr==1)
132     begin
133     File[WR]=WrightData;
134     File[0]=0;
135     end
136
```

در این بخش دستور های مختلف اجرا می شوند البته همگی با شروطی قابل اجرا هستند که تمام شروط از سوی واحد کنترل تعیین می شوند .

```
142   always @(posedge Clk)
143   begin
144
145   if (Branch) //BUN
146   begin
147         K=(OutSign)*2;
148         PC=PC+2+K;
149   end
150
151   if(Jum) //JUMP
152   begin
153         PC=PC+2;
154         PC={PC[15:13],Instr[11:0]*2};
155   end
156
157   if (Branch==0 && Jum==0)
158      begin
159      PC=PC+2;
160   end
161
162   end
163
```

واحد PC نیز با توجه به اینکه دستور جامپ در پیش داریم و یا پرش و یا اینکه در حالت عادی خودش قرار دارد ، مقدار خود را با هر لبه کلاک آپدیت میکند . (نکته هر دستور در یک لبه کلاک انجام می شود)

حال که بخش های مختلف پردازنده تعیین و ارتباط ها برقرار شد ، زمان آن رسیده است تا حافظه و رجیستر فایل ها را تعریف کرده و با مقادیر مناسب آنها را پر کنیم .

```
 7  reg [7:0 ] MEM  [0:33];
 8  reg [15:0] File [0:7];
 9  initial begin
10  File[0]=16'b0000000000000000;
11  File[1]=16'b0000000000000000;
12  File[2]=16'b0000000000000101;
13  File[3]=16'b0000000000010000;
14  File[4]=16'b0000000000000000;
15  File[5]=16'b0000000000000100;
16  File[6]=16'b0000000000000000;
17  File[7]=16'b0000000000001000;
18
19  MEM[1 ]=8'b00000111; MEM[0 ]=8'b01110000;  // F(6) = F(3) + F(5) = 16 + 4 = 20
20  MEM[3 ]=8'b11110000; MEM[2 ]=8'b00000100;  // Jump To Address 8
21  MEM[5 ]=8'b00000000; MEM[4 ]=8'b00000000;
22  MEM[7 ]=8'b00000000; MEM[6 ]=8'b10000000;
23  MEM[9 ]=8'b00001110; MEM[8 ]=8'b10001001;  // F(1) = F(7) - F(2) = 8 - 5 = 3
24  MEM[11]=8'b01110111; MEM[10]=8'b00010000;  // Load F(4)=MEM[16+F(3)] = MEM[32] = 2;
25  MEM[13]=8'b00000000; MEM[12]=8'b00000000;
26  MEM[15]=8'b00000000; MEM[14]=8'b00000000;
27  MEM[17]=8'b00000000; MEM[16]=8'b00000000;
28  MEM[19]=8'b00000000; MEM[18]=8'b00000000;
29  MEM[21]=8'b00000000; MEM[20]=8'b00000000;
30  MEM[23]=8'b00000000; MEM[22]=8'b00000000;
31  MEM[25]=8'b00000000; MEM[24]=8'b00000000;
32  MEM[27]=8'b00000000; MEM[26]=8'b00000000;
33  MEM[29]=8'b00000000; MEM[28]=8'b00000000;
34  MEM[31]=8'b00000000; MEM[30]=8'b00000000;
35  MEM[33]=8'b00000000; MEM[32]=8'b00000010;
36
37  end
38
```

تعداد هشت عدد رجیستر فایل تعریف شده و با مقادیر دلخواه مقداردهی اولیه شده اند . سپس 34 بایت حافظه دلخواه نیز تعریف شد که درون خونه های ابتدایی حافظه برنامه اسمبلی فوق نوشته است .

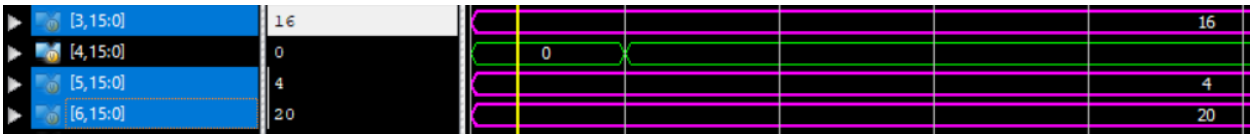
در دستور اول محتوای رجیستر فایل 5 و 3 باهم جمع شده و در رجیستر فایل 6 ریخته می شود ، یعنی جمع عدد 16 و 4 که 20 خواهد بود .

در دستور بعدی به آدرس 8 حافظه جامپ می کنیم تا دستور موجود در 8 را اجرا نماییم . که در این خونه تفریق رجیستر فایل هفتم و دوم قرار دارد یعنی مقدار 8 منهای 5 که 3 می شود و در رجیستر فایل یک ریخته می شود .

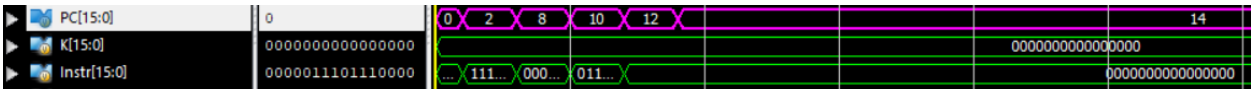
در دستور آخر عمل Load را انجام داده ایم ، محتوای خونه 32 حافظه که مقدار آن 2 می باشد در رجیستر فایل چهارم قرارداده می شود .

مشاهده خروجی برنامه :

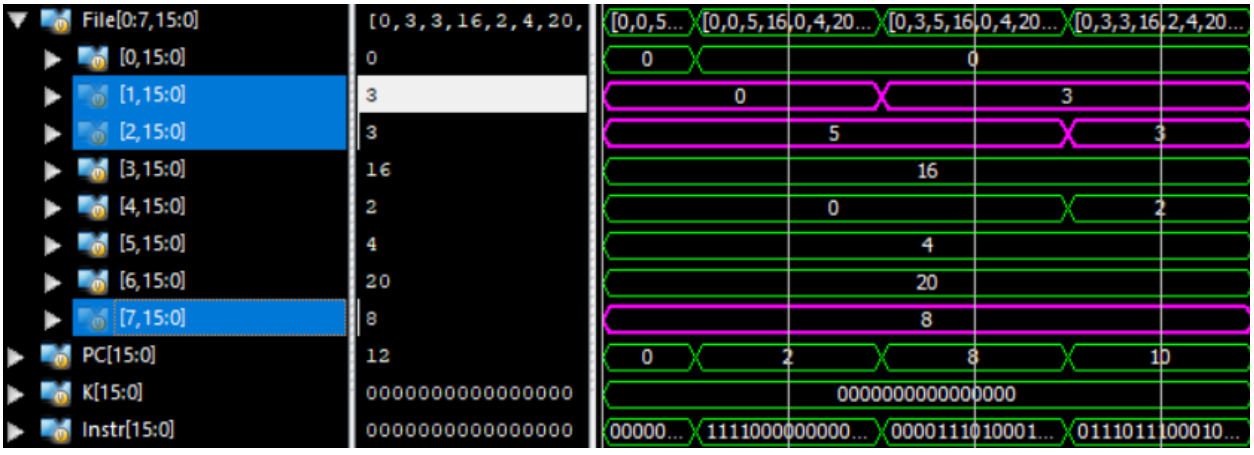
که البته چیز واضحی مشاهده نمی شود بنابراین بصورت مجزا در زیر دستورات اجرا شده را نمایش می دهیم .



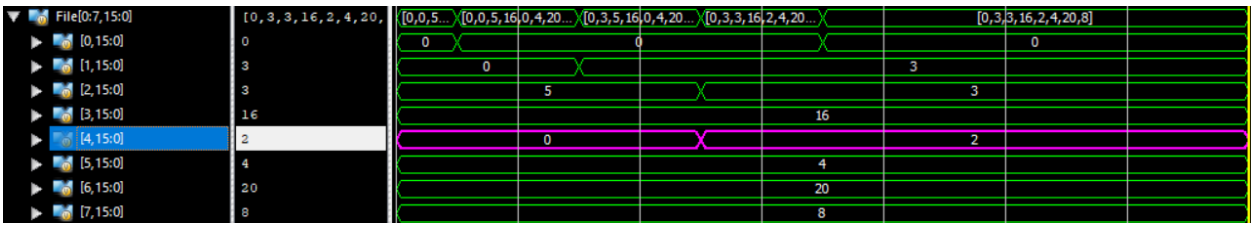
در دستور اول : محتوای دو رجیستر مذکور یعنی 4 و 16 جمع شده و در رجیستر بعدی عدد 20 قرار گرفته است .



در دستور دوم : همانطور که می بینید مقدار PC بعد از اجرای دستور اول ناگهان به عدد 8 پرش می کند .



در دستور سوم محتوای رجیستر هفت و دو از هم تفریق می شود یعنی 8 منهای 5 که حاصل 3 است و در رجیستر فایل اول ریخته شده است .



و در دستور آخر محتوای حافظه 32 که دو است در رجیستر فایل چهارم قرار گرفته است .