# گزارش پروژه دوم درس اصول سیستمهای کامپیوتری


سید علی عمادی - ۹۵۱۰۰۱۹۱

امین جلالی - ۹۵۱۰۰۰۹٤

گام ۱: طراحی واحد محاسبه و منطق

به هرکدام از مقادیر operation یک عمل خاص نسبت داده شده است که در جدول زیر آمده است.

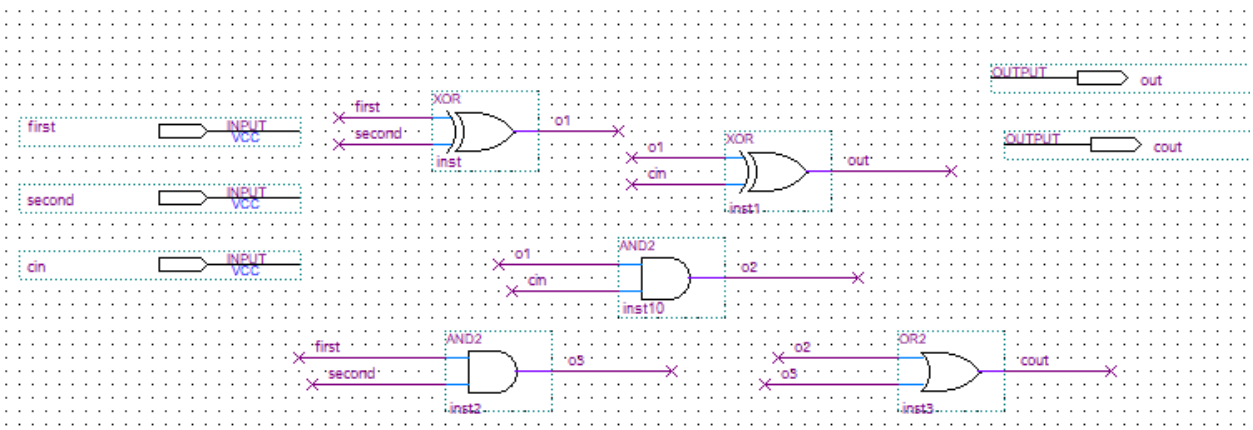
| Operation | عملیات |
|---|---|
| 0000 | Or |
| 0001 | And |
| 0010 | Xor |
| 0011 | Nor |
| 0100 | Add |
| 0101 | Sub |
| 0110 | Min(A,B) |
| 0111 | div |
| 1000 | Mul |
| 1001 | Sra |
| 1010 | Srl |
| 1011 | Sll |

طرز کار alu به اینصورت است که ابتدا تمام این ۱۲ عملیات محاسبه می شوند،سپس با استفاده از multiplexer که سیگنال selector آن همان operation است،مقدار درست به C و سایر خروجی‌ها نسبت داده می‌شود.شبیه‌سازی تمام عملیات alu در آخر بخش alu آورده شده است.

حال طرز پیاده‌سازی عملیات غیربدیهی را به‌ترتیب توضیح می‌دهیم:

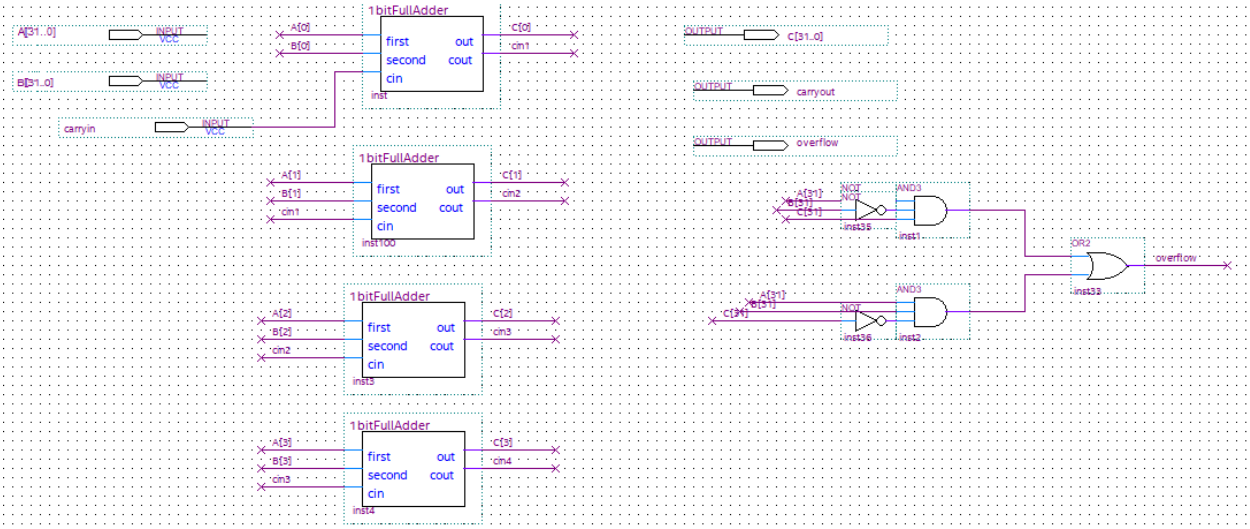
- Add :

از یک fullAdder یک بیتی که شماتیک آن در زیر آمده است استفاده شده و سپس خروجی carryout بیت iام به عنوان carryin بیت i+1 ام در نظر گرفته شده است.



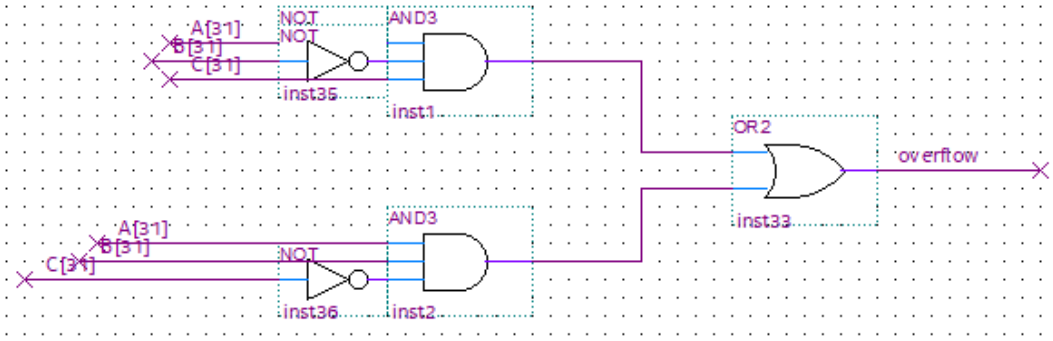

شکل ۱-Full adder یک بیتی

در شکل زیر قسمتی از سخت افزار جمع آمده است:




شکل ۲–قمستی از سخت‌افزار جمع ۳۲بیتی

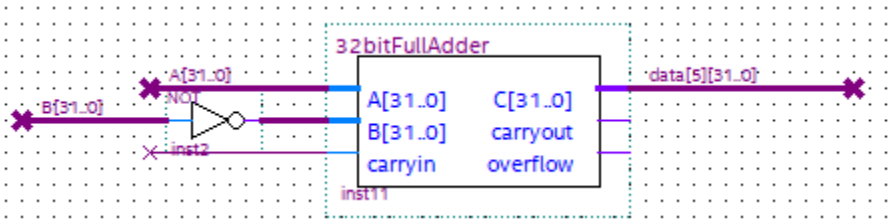
برای تشخیص وقوع overflow هم از مداری به شکل زیر استفاده می‌شود.به طور کلی در جمع اگر حاصل جمع دو عدد مثبت عددی منفی شود و یا حاصل جمع دو عدد منفی یک عدد مثبت شود overflow رخ داده است.




شکل ۳– مدار محاسبه overflow

- Sub:

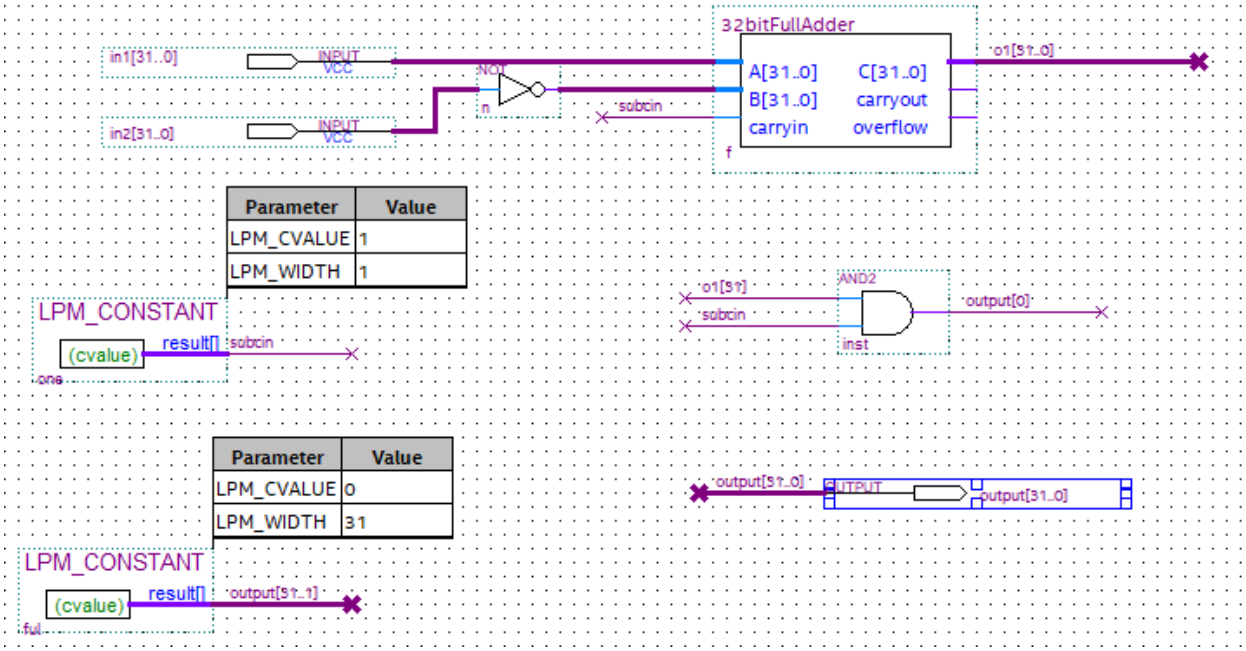
می‌دانیم که $a - b = a + \bar{b} + 1$ بنابراین برای محاسبه تفریق دو عدد هم از همان سخت‌افزار جمع استفاده می‌کنیم با این تفاوت که ابتدا b را not می‌کنیم و carryIn اولیه را هم برابر با ۱ قرار می‌دهیم.




شکل ۴–سخت‌افزار تفریق

- Min(A,B):

A – B محاسبه می‌شود اگر سمت چپ ترین بیت برابر با ۱ بود ، خروجی برابر با ۱ می‌شود و اگر سمت‌چپ‌ترین بین برابر با ۰ بود خروجی برابر با صفر می‌شود.




شکل ۵- سخت افزار min

- Div :

برای تقسیم از الگوریتم موجود در کتاب استفاده شده است.ابتدا دو رجیستر ۶۴بیتی با نامهای remainder و divisorدرنظر گرفته شده‌اند.که ۳۲ بیت سمت راست رجیستر remainder برابر با مقسوم و ۳۲ بیت سمت چپ divisor برابر با مقسوم علیه است. همچنین رجیستر دیگری با نام quotient وجود دارد که ۳۲ بیتی است و مقدار اولیه آن صفر است.مراحل زیر را۳۳ بار انجام می‌دهیم:

1.remainder = remainder – divisor
2a. if remainder >= 0 → remainder = remainder + divisor,sll quotient, quotient[0] = 0
2b. if remainder < 0 → sll quotient, quotient[0] = 1
3.shift divisor right

پس از ۳۳ بار تکرار quotient برابر با خارج قسمت تقسیم و remainder برابر با باقیمانده تقسیم خواهدشد.

کد وریلاگ عملیات تقسیم:

```
module div(A,B,C);
    input [31:0] A;
    input [31:0] B;
    output [31:0] C;
    reg [31:0] quotient,C,first,second;
    reg [63:0] remainder,divisor;
    integer i;
    reg flag0,flag1,sign;
    always@(*)begin
        flag0 = 0;
        flag1 = 0;
        first[31:0] = A;
        second[31:0] = B;
        sign = 0;
        if (first[31] == 1)begin
            first = ~first;
            first = first + 1;
            flag0 = 1;
        end
        if (second[31] == 1)begin
            second = ~second;
            second = second + 1;
            flag1 = 1;
        end
        if ((flag0 == 0 && flag1 == 1) || (flag0 == 1 && flag1 == 0))begin
            sign = 1;
        end
        quotient[31:0] = 0;
        divisor[31:0] = 0;
        divisor[63:32] = second;
        remainder[31:0] = first;
        remainder[63:32] = 0;
        for(i = 0;i < 33 ; i = i + 1) begin
            remainder = remainder - divisor;
            if (remainder[63])begin
                remainder = remainder + divisor;
                quotient = (quotient << 1);
                //quotient[0] = 1'b0;
                divisor = (divisor >> 1);
            end
            else begin
                quotient = (quotient << 1);
                quotient[0] = 1'b1;
                divisor = (divisor >> 1);
            end
        end
        C = quotient;
        if (sign == 1) begin
            C = ~C;
            C = C + 1;
        end
    end
endmodule
```

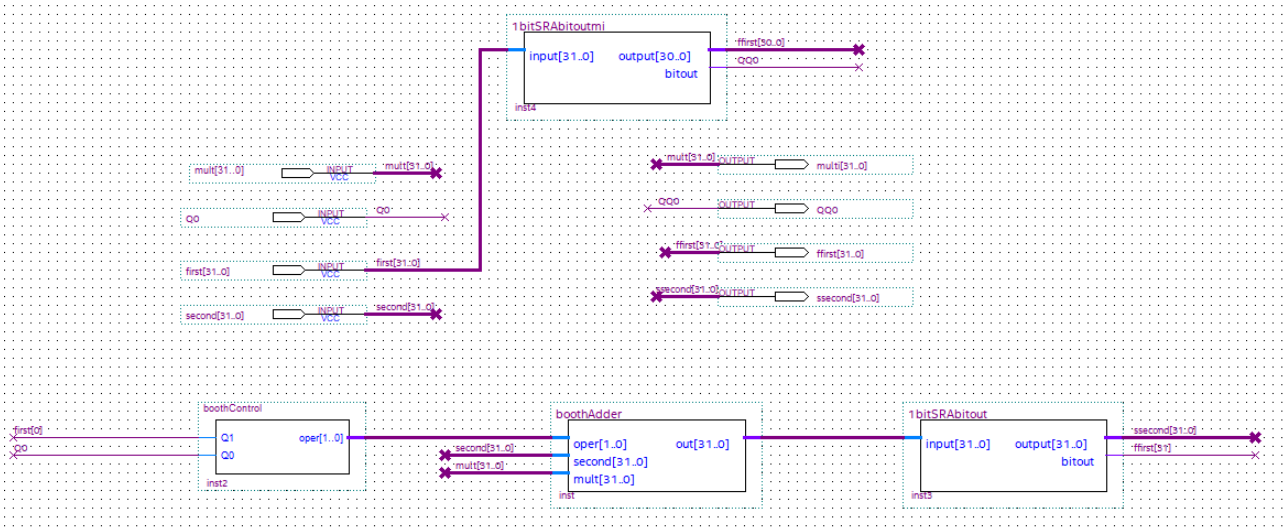
- Mul:

برای ضرب از الگوریتم booth استفاده شده است.در زیر هم کد وریلاگ و هم مدار این الگوریتم آمده است.

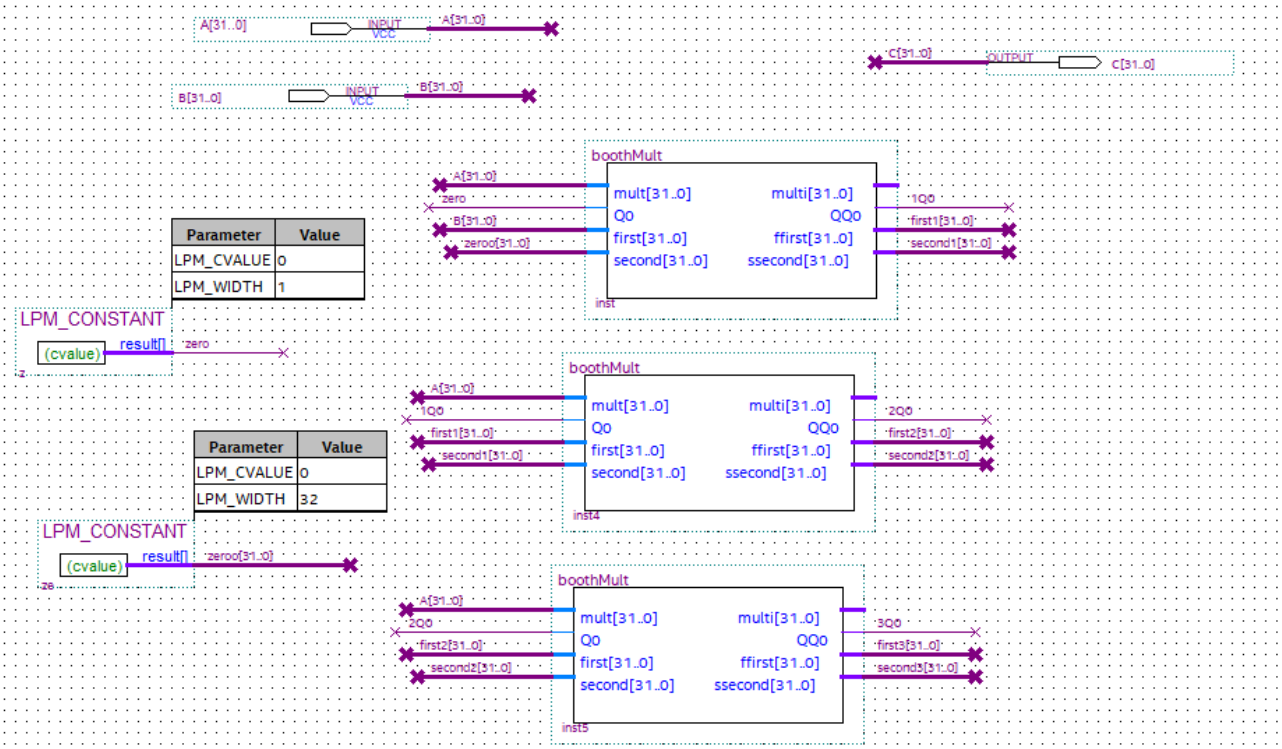
کد وریلاگ :

```
module boothMul(A,B,C,overflow);
   input [31:0] A,B;
   output [31:0] C;
   output overflow;
   reg [64:0] product;
   reg [31:0] C;
   reg overflow;
   integer i;
   always@(*)
   begin
      product[0] = 0;
      product[32:1] = B;
      product[64:33] = 0;
      for (i = 0; i < 32;i = i +1) begin
         if (!product[0] && product[1]) begin// Q = 0 and LSB of prouct = 1
            product[64:33] = product[64:33] - A;
         end
         if (product[0] && !product[1]) begin
            product[64:33] = product[64:33] + A;
         end
         begin
            product = (product >>> 1);
         end
      end
       C = product[32:1];
       overflow = !(product[64:33] == 0);
   end
endmodule
```

شماتیک :



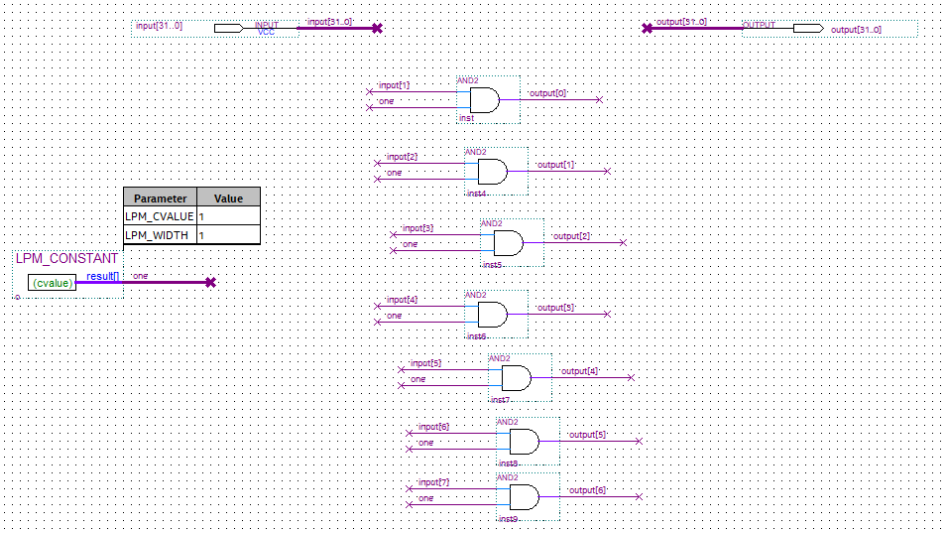

ابتدا Q0 و Q1 به عنوان ورودی به boothcontrol داده می‌شود سپس با توجه به این دو مقدار oper[1..0] تولید می شود و به boothAdder می‌رود.در boothAdder طبق مقدار oper تصمیم گرفته می شود که باید عمل جمع یا تفریق انجام شود و یا کاری انجام نشود.سپس رجیستر ها یک واحد شیفت داده می‌شوند.



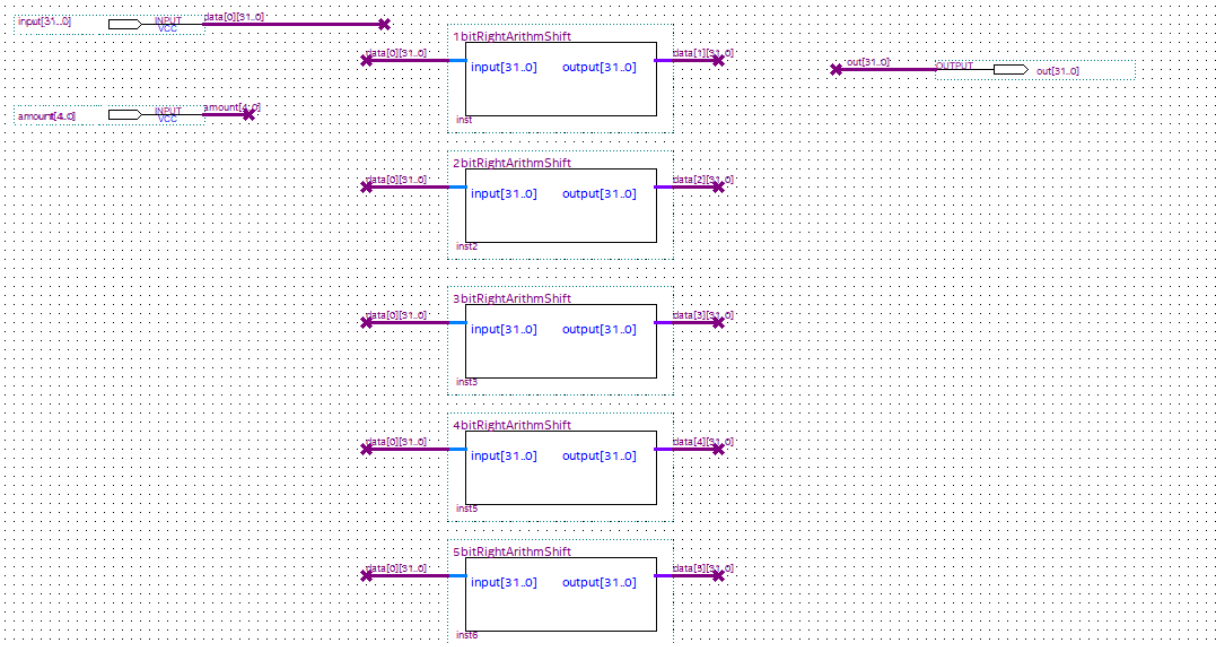

قسمتی از سخت‌افزار ضرب

- Sra:

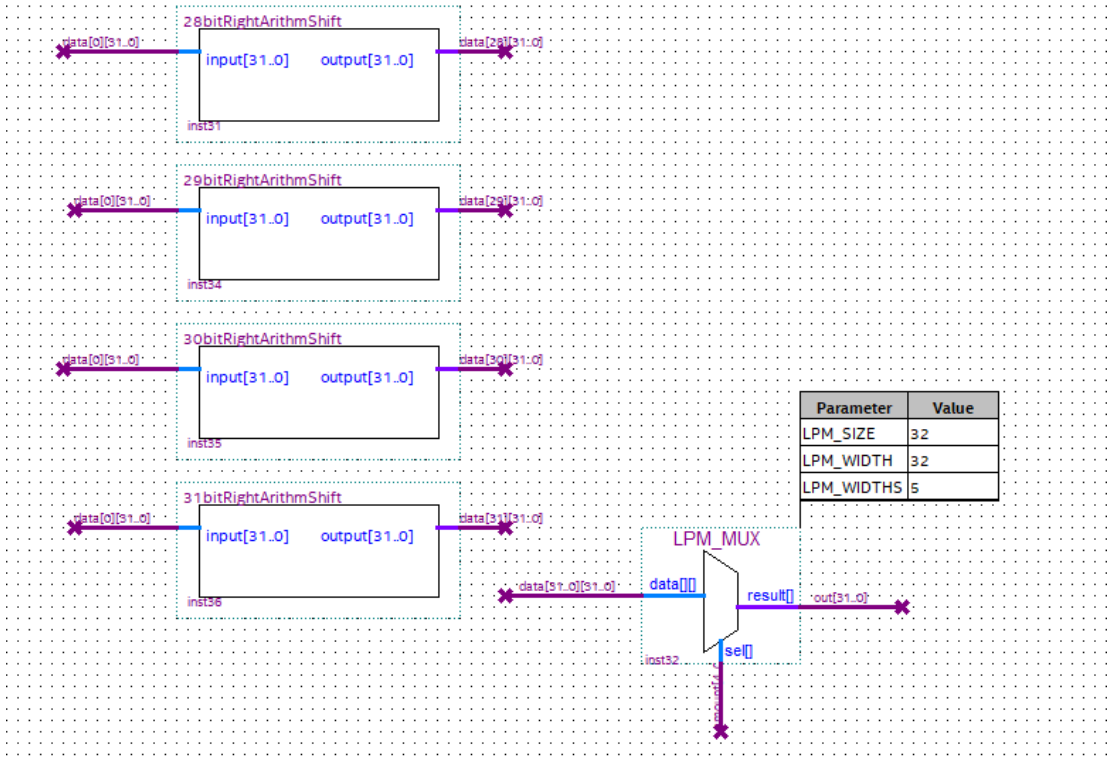
ابتدا شیفت یک بیتی ساخته می‌شود سپس با استفاده از آن شیفت های بزرگتر ساخته می‌شوند.برای سخت افزار شیفت حاصل همه ی شیفت های ۰ تا ۳۱ بیت محاسبه می شود سپس با multiplexer که selector آن shift_amount[4..0] است خروجی درست به دست می‌آید.




قسمتی از شیفت حسابی ۱ واحدی



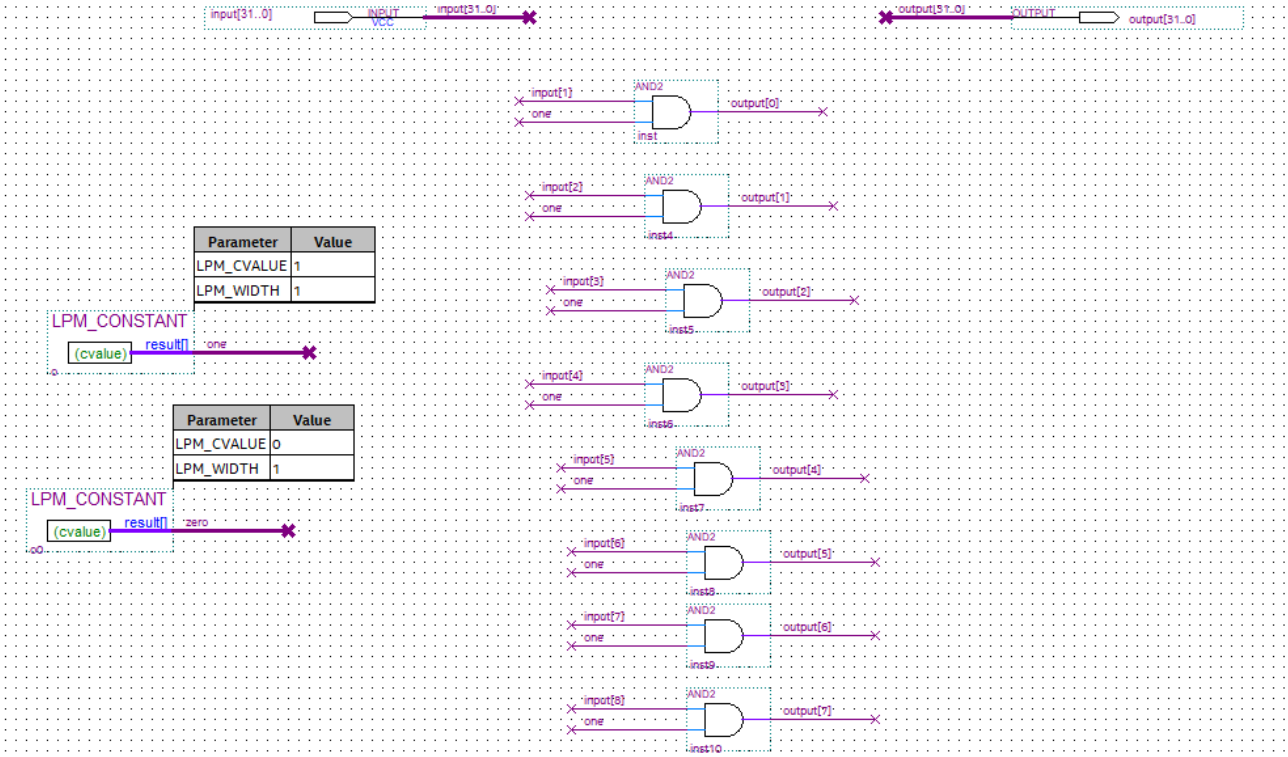

شیفت حسابی به راست – قسمت ۱



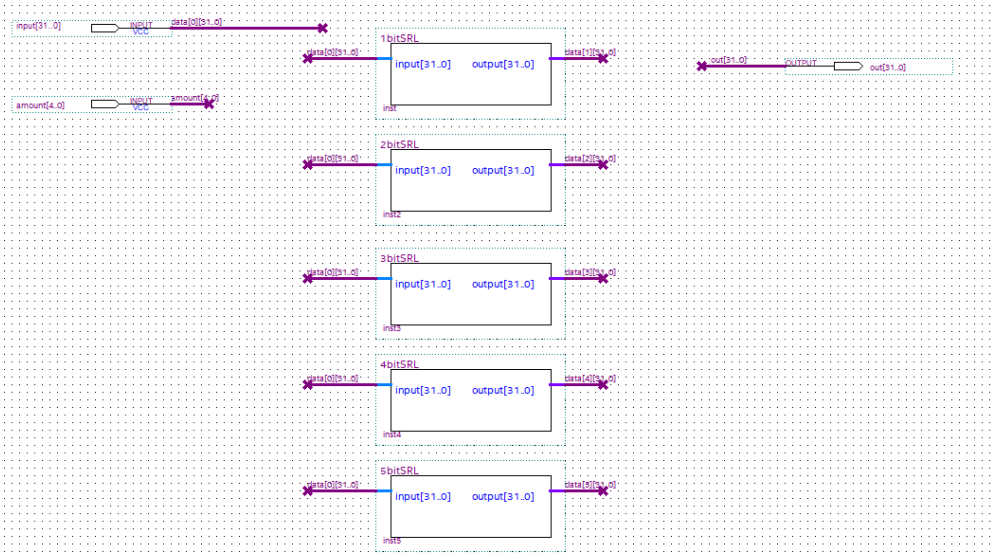

شیفت حسابی به راست – قسمت ۲

- Srl:

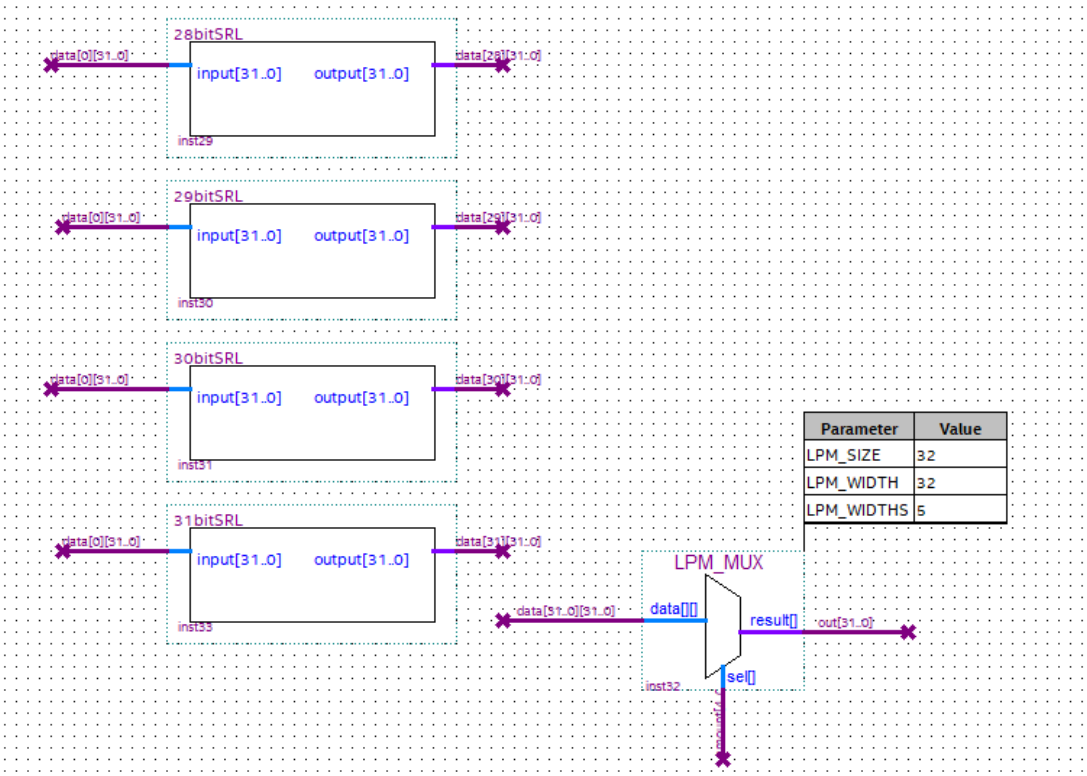
مانند قسمت قبل ابتدا شیفت یک واحدی ساخته می‌شود.سپس با استفاده از این شیفت ، شیفت های یک تا ۳۱ واحدی ، و سپس با multiplexer خروجی درست نسبت داده می‌شود.




قسمتی از شیفت منطقی یک واحدی به راست




شیفت منطقی به راست – قسمت ۱



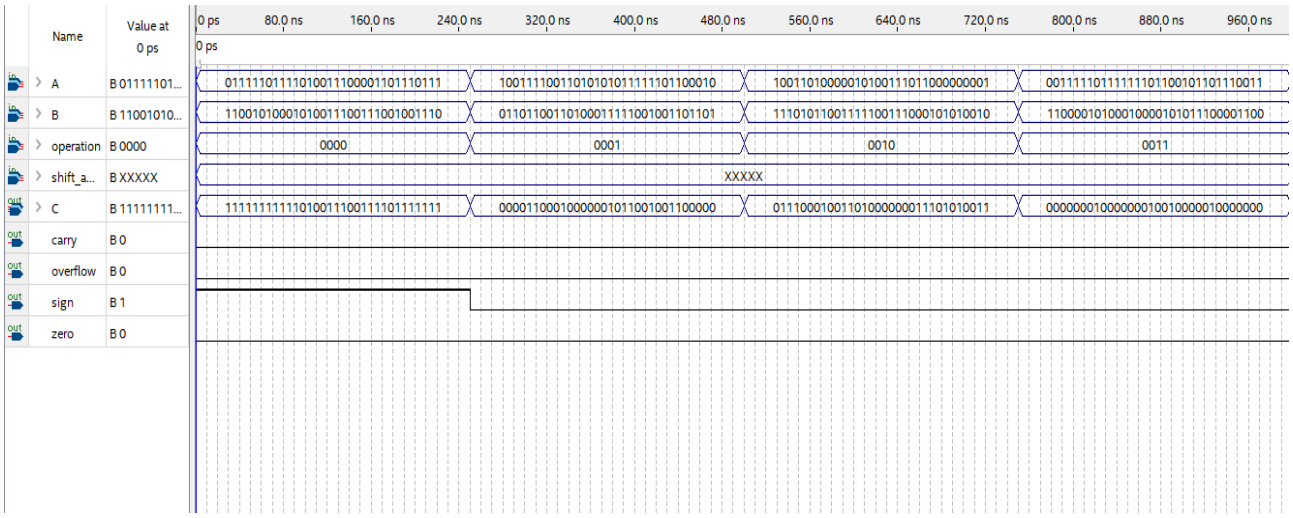

شیفت منطقی به راست – قسمت ۲

- SII:

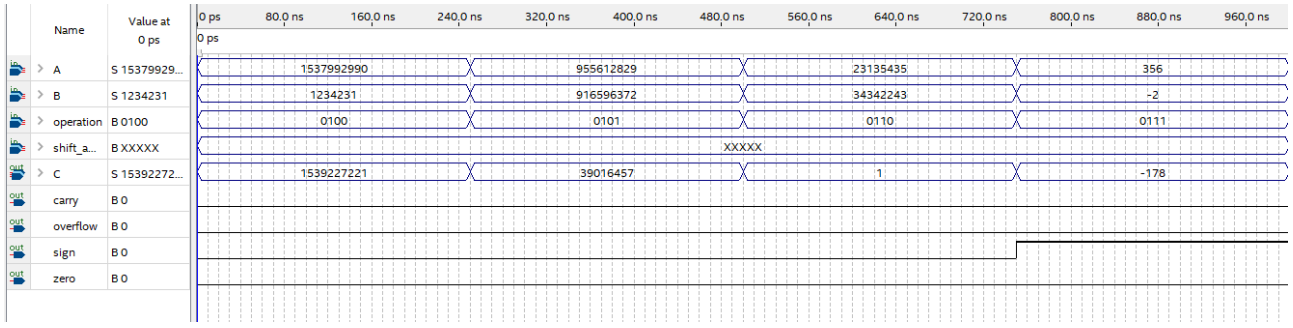
پیاده سازی آن دقیقا مانند شیفت منطقی به راست است.

نتایج شبیه سازی:

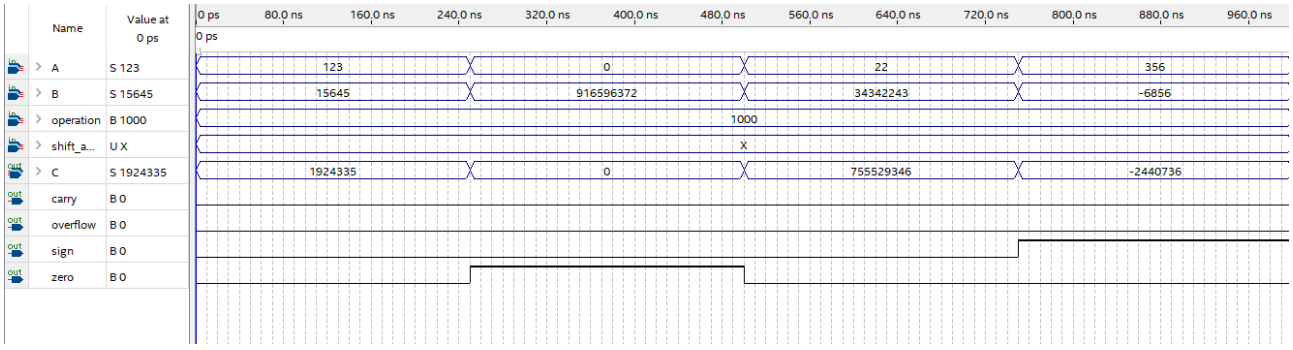
## Or,and,xor,nor



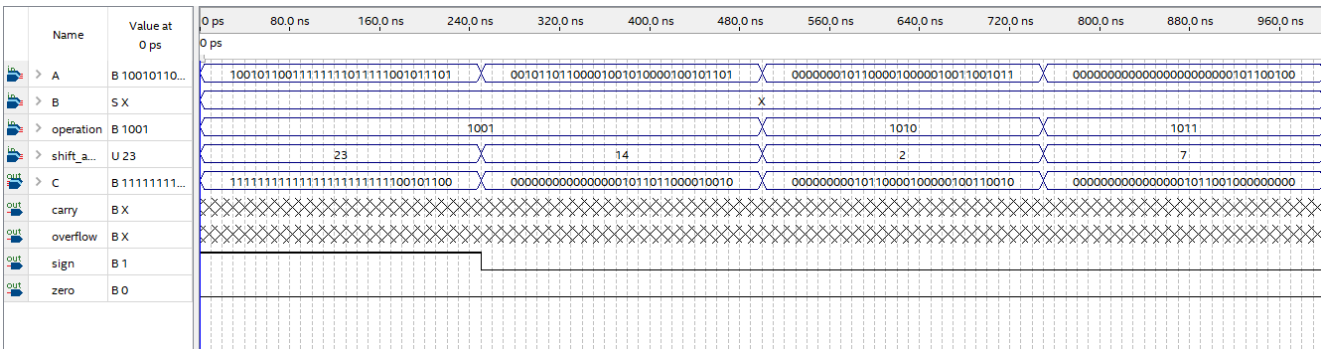

## Add,sub,min,div




## mul



Sra,srl,sll



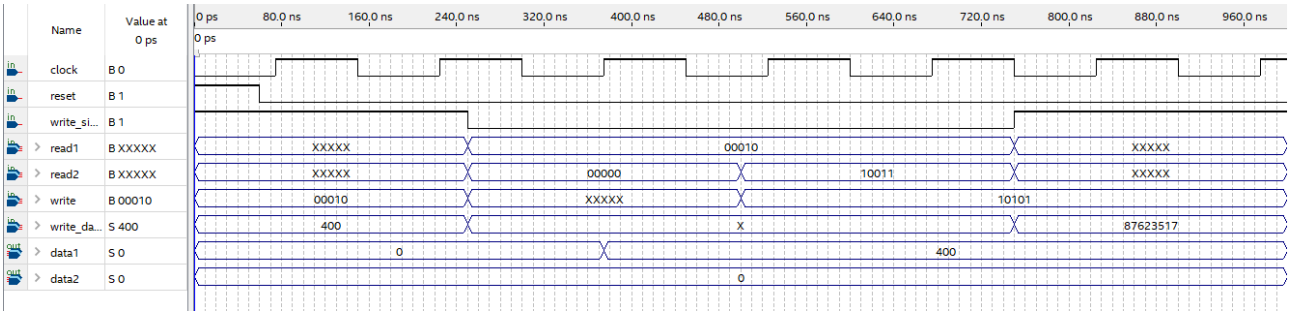

## گام دوم : طراحی رجیستر فایل

همیشه در لبه‌ی clock ،اگر reset = 1 باشد مقادیر تمام رجیستر ها برابر صفر می شود.اگر reset = 0 بود طبق سیگنال های ورودی یا از رجیستر خوانده می شود و یا بر روی یک رجیستر اطلاعات نوشته می شود.

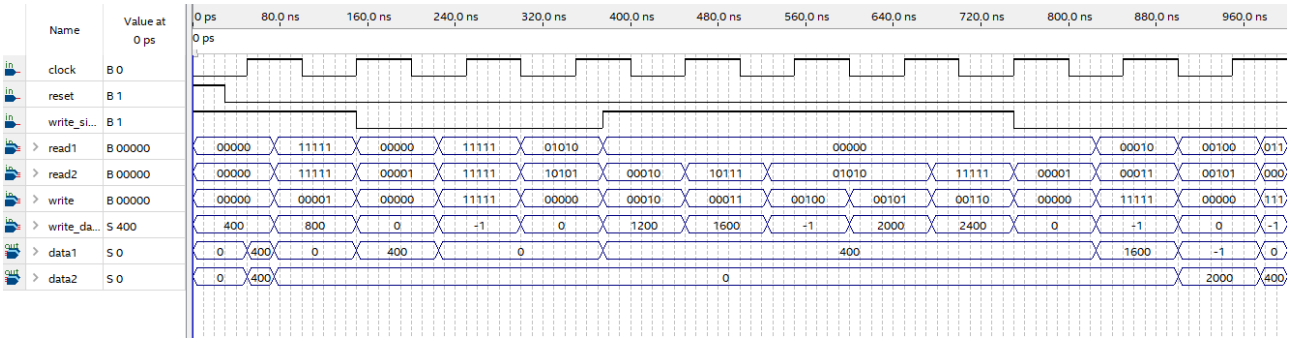
کد وریلاگ:

```
module regfile(clock,reset,read1,read2,write,write_data,write_signal,data1,data2);
input clock;
input reset;
input [4:0] read1,read2,write;
input [31:0] write_data;
input write_signal;
output [31:0] data1,data2;
reg [31:0] registers[31:0];
reg [31:0] data1;
reg [31:0] data2;
integer i;
always @(posedge clock) begin
if (reset)begin
   for(i = 0; i < 32; i = i+1)
      begin
           registers[i] = 0;
      end
   end
   else begin
   if (write_signal)begin
      $monitor("REG[%d]=%d", write, registers[write]);
      registers[write] <= write_data;
      $monitor("REG[%d]=%d", write, write_data);
   end
   else begin
      data1 <= registers[read1];
      data2 <= registers[read2];
      $monitor("REG[%d]=%d", read1, registers[read1]);
      $monitor("REG[%d]=%d", read2, registers[read2]);
   end
   end
end
endmodule
```

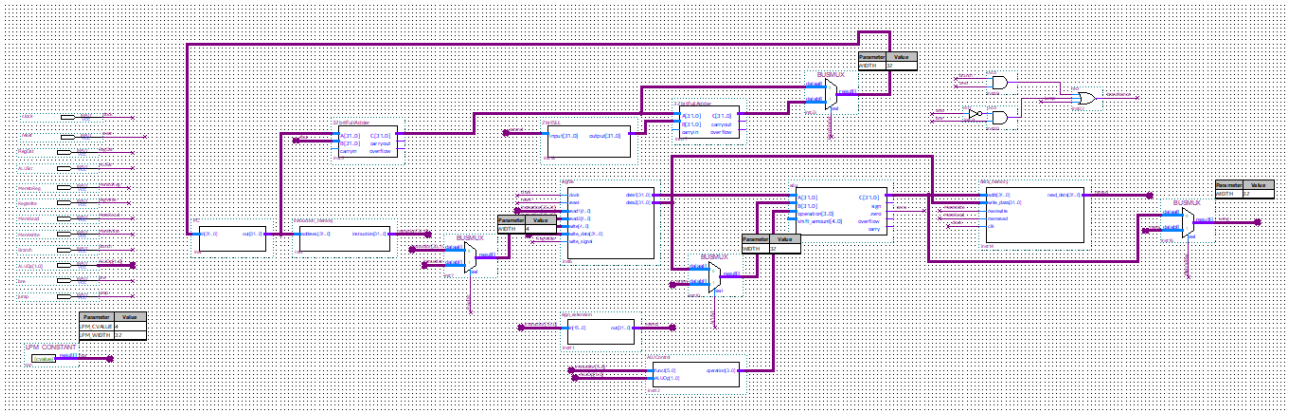
مثال اول از رجیستر فایل :




مثال دوم از رجیستر فایل:

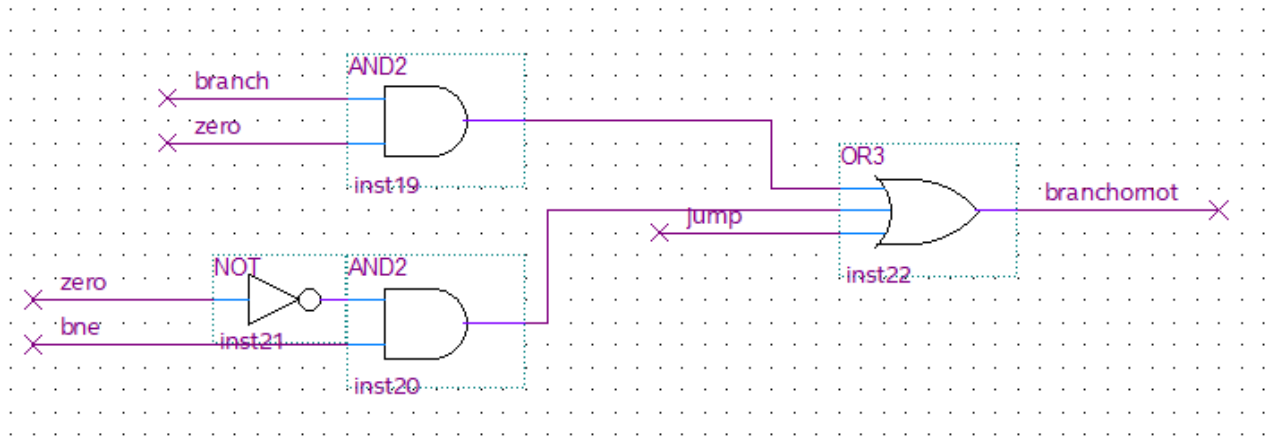



## Datapath:



پیاده سازی datapath مانند datapath کتاب است با این تفاوت که دو سیگنال کنترلی bne و jump هم اضافه شده است.

تصمیم اینکه انشعب انجام شود یا خیر با مدار زیر است:




اجزای datapath :

PC:

یک رجیستر است که آدرس دستوری که باید خوانده شود را در خود نگه می دارد.

Instruction memory :

حافظه‌ای که دستورات در آن قرار دارند و با استفاده از PC آدرس دستور خوانده می شود.

کد وریلاگ :

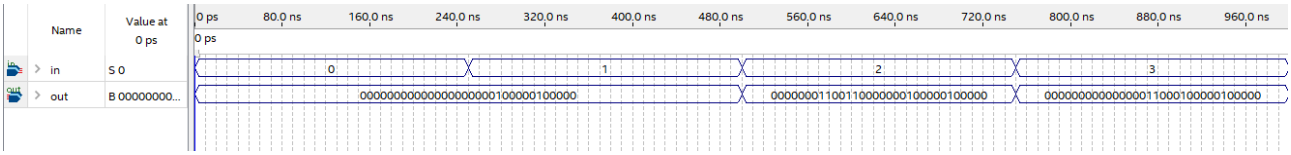
```
module instruction_memory

(
    address,
    instruction
);

    input     [31:0]   address;
    output    [31:0]   instruction;
    reg   [31:0] instruction_memory  [255:0];
    initial
    begin
        $readmemh("program.mips",instruction_memory);
    end
    assign instruction = instruction_memory[address[9:2]];

 endmodule
```

شبیه سازی:

تنها تفاوت این است که از address[7:0] خوانده می شود و آدرس یکی یکی اضافه می‌شود.(فقط برای این شبیه سازی)

فایل program.mips :

```
00000000000000000000100000100000
00000000000000000000100000100000
00000001100110000000100000100000
00000000000000011000100000100000
00000010100000000000100000100000
```




Data memory :

حافظه‌ای که برای دستوراتی همچون lw و sw استفاده می‌شود.

کد وریلاگ:

```
module dmemory (
input wire [31:0] addr,
input wire [31:0] write_data,
input wire memwrite, memread,
input wire clk,
output [31:0] read_data
);

reg [31:0] MEMO[65535:0];  // 256 words of 32-bit memory

integer i;

initial begin
  for (i = 0; i < 256; i = i + 1) begin
    MEMO[i] = i;
  end
end


always @(posedge clk) begin
  if (memwrite == 1'b1) begin
    MEMO[addr] <= write_data;
  end
end
assign read_data = (memread == 1'b1) ? MEMO[addr]:32'd0;

endmodule
```

ALU control :

کد وریلاگ :

```
    reg [3:0] operation;
    always@(*)begin
        if (ALUOp == 2'b00 && funct == 6'b100000)begin
            operation = 4'b0000;
        end
        else if (ALUOp == 2'b00 && funct == 6'b100001)begin
            operation = 4'b0001;
        end
        else if (ALUOp == 2'b00 && funct == 6'b100010)begin
            operation = 4'b0010;
        end
        else if (ALUOp == 2'b00 && funct == 6'b100011)begin
            operation = 4'b0011;
        end
        else if(ALUOp == 2'b00 && funct == 6'b010000)begin
            operation = 4'b0100;
        end
        else if(ALUOp == 2'b00 && funct == 6'b010001)begin
            operation = 4'b0101;
        end
        else if(ALUOp == 2'b00 && funct == 6'b010010)begin
            operation = 4'b0110;
        end
        else if(ALUOp == 2'b00 && funct == 6'b001000)begin
            operation = 4'b0111;
        end
        else if(ALUOp == 2'b00 && funct == 6'b001001)begin
            operation = 4'b1000;
        end
        else if(ALUOp == 2'b00 && funct == 6'b000100)begin
            operation = 4'b1001;
        end
        else if(ALUOp == 2'b00 && funct == 6'b000101)begin
            operation = 4'b1010;
        end
        else if (ALUOp == 2'b00 && funct == 6'b000110)begin
            operation = 4'b1011;
        end
        else if (ALUOp == 2'b11) begin
            operation = 4'b0100;
        end
        else if (ALUOp == 2'b10)begin
            operation = 4'b0101;
        end
    end
endmodule
```

## واحد کنترل :

کد وریلاگ :

```
module controlUnit(op,RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp,bne,jump);
    input [5:0] op;
    output RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,bne,jump;
    output [1:0] ALUOp;
    reg RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,bne,jump;
    reg [1:0] ALUOp;
    always@(*) begin
        if (op == 6'b000000) begin//R-Type
            ALUOp = 2'b00;
            ALUSrc = 1'b0;
            RegDst = 1'b1;
            MemtoReg = 1'b0;
            RegWrite = 1'b1;
            MemRead = 1'b0;
            MemWrite = 1'b0;
            Branch = 1'b0;
            bne = 1'b0;
            jump = 1'b0;
        end
        else if(op == 6'b100011)begin//LW
            ALUOp = 2'b11;
            ALUSrc = 1'b1;
            RegDst = 1'b0;
            MemtoReg = 1'b1;
            RegWrite = 1'b1;
            MemRead = 1'b1;
            MemWrite = 1'b0;
            Branch = 1'b0;
            bne = 1'b0;
            jump = 1'b0;
        end
        else if(op == 6'b101011)begin//SW
            ALUOp = 2'b11;
            ALUSrc = 1'b1;
            RegDst = 1'bx;
            MemtoReg = 1'bx;
            RegWrite = 1'b0;
            MemRead = 1'b0;
            MemWrite = 1'b1;
            Branch = 1'b0;
            bne = 1'b0;
            jump = 1'b0;
        end
        else if(op == 6'b000100)begin//beq
            ALUOp = 2'b10;
            ALUSrc = 1'b1;
```

```
47          RegDst = 1'bx;
48          MemtoReg = 1'bx;
49          RegWrite = 1'b0;
50          MemRead = 1'b0;
51          MemWrite = 1'b0;
52          Branch = 1'b1;
53          bne = 1'b0;
54          jump = 1'b0;
55       end
56       else if(op == 6'b000101)begin//bne
57          ALUOp = 2'b10;
58          ALUSrc = 1'b1;
59          RegDst = 1'bx;
60          MemtoReg = 1'bx;
61          RegWrite = 1'b0;
62          MemRead = 1'b0;
63          MemWrite = 1'b0;
64          Branch = 1'b0;
65          bne = 1'b1;
66          jump = 1'b0;
67       end
68       else if(op == 6'b000010)begin//jump
69          ALUOp = 2'b11;
70          ALUSrc = 1'b1;
71          RegDst = 1'bx;
72          MemtoReg = 1'bx;
73          RegWrite = 1'b0;
74          MemRead = 1'b0;
75          MemWrite = 1'b0;
76          Branch = 1'b0;
77          bne = 1'b0;
78          jump = 1'b1;
79       end
80    end
81 endmodule
82
```

شبیه سازی :

فایل program.mips :

دستور add :

مقدار اولیه رجیستر ۰ام ۱۰ است.

Add $0,$0,$0

```
1  00000000000000000000000000010000
2  00000000000000000000000000010000
3  00000000000000000000000000010000
4  00000000000000000000000000010000
```