

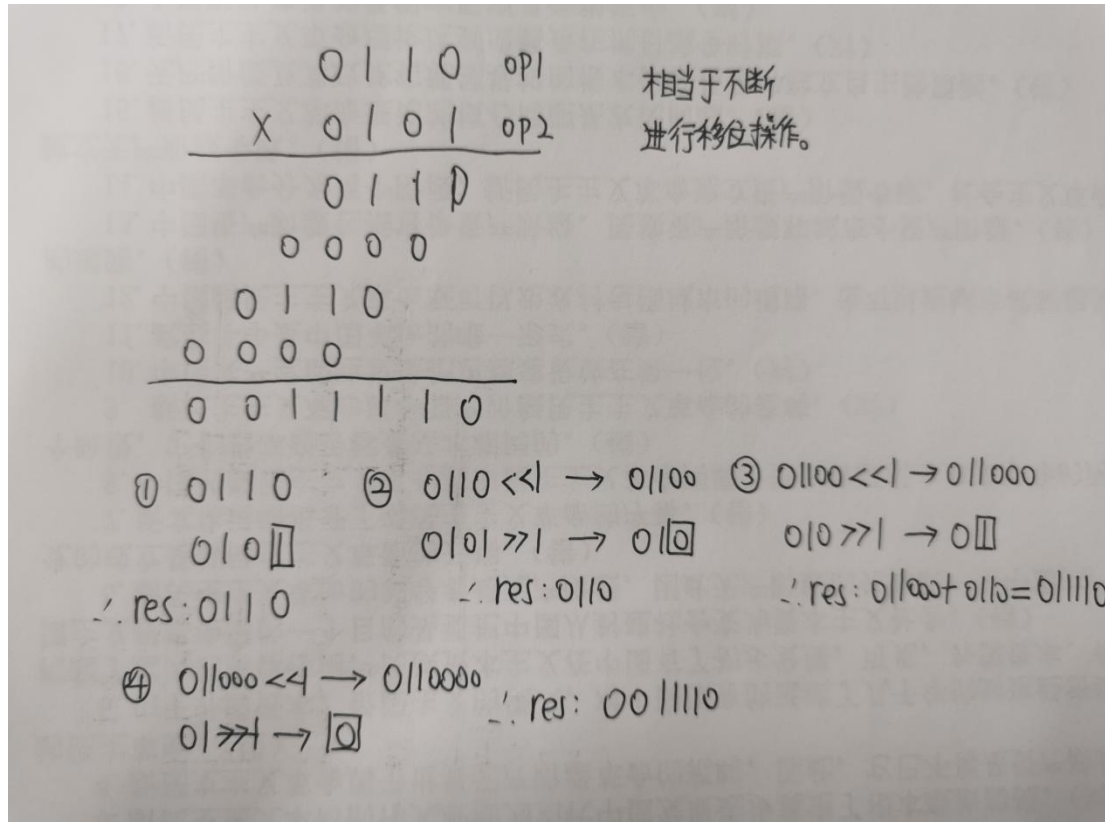
## 1、 乘法器以及合并乘除法器说明（可选模块）

### A)乘法器设计

#### 1) 乘法器文件：

● mul\_32\_1 (mul\_32\_1.v)

#### 2) 乘法器原理



乘法原理：op2 从最低位开始依次判断是否为 1，若为 1 就将 op1 最低位与 op2 当前位对齐，最后加到一起。因此，在流水中可以让 op2 与 op1 相反方向进行移位，从最低位开始，如果是 1 就将移位后的 op1 加入到结果中，否则保持原值不动，所以 op2 有几位，这个过程就进行几次，本实验中均是 32 位运算，所以循环要进行 32 次，这是判断乘法结束的重要条件。

#### 3) 代码解读：

```
module mul_32_1(
    input wire rst,
    input wire clk,
    input wire signed_mul_i,
    input wire[31:0] opdata1_i,
    input wire[31:0] opdata2_i,
    input wire start_i,           ///是否开始乘法运算
    input wire annul_i,           //是否取消乘法运算，1 位取消
    output reg signed [63:0] result_o,    ///乘法运算结果
    output reg ready_o
);
```

说明：因为受到除法器的启发，因此在输入输出端口与除法器没有太大区别，无非是改变了

变量名，变量含义已在注释中。

```
reg [5:0] cnt; //记录乘法进行了几轮
reg [1:0] state; //乘法器处于的状态
reg[63:0] mult1_shift;
reg[31:0] mult2_shift;
reg[63:0] mult1_acc;
reg[31:0] neg1;
reg[31:0] neg2;
always @ (posedge clk) begin
    if (rst) begin
        state <= 2'b00;
        result_o <= {32'b0,32'b0};
        ready_o <= 1'b0;
    end else begin
        case(state)
            2'b00: begin //空闲
                if (start_i == 1'b1 && annul_i == 1'b0) begin
                    state <= 2'b10;
                    cnt <= 6'b000000;
                    if(signed_mul_i == 1'b1 && opdata1_i[31] == 1'b1) begin
                        mult1_shift <= {32'b0, (~opdata1_i + 1)};
                    end else begin
                        mult1_shift <= {32'b0, opdata1_i};
                    end
                    if (signed_mul_i == 1'b1 && opdata2_i[31] == 1'b1 ) begin
                        mult2_shift <= (~opdata2_i + 1);
                    end else begin
                        mult2_shift <= opdata2_i;
                    end
                    begin
                        mult1_acc <= 64'b0;
                    end
                end else begin
                    ready_o <= 1'b0;
                    result_o <= {32'b0, 32'b0};
                end
            end
            2'b10: begin
                if(annul_i == 1'b0) begin //进行乘法运算
                    if(cnt != 6'b100000) begin
                        mult1_acc
                        <= (mult2_shift[0] == 1'b1) ? (mult1_acc + mult1_shift) : mult1_acc;
```

```

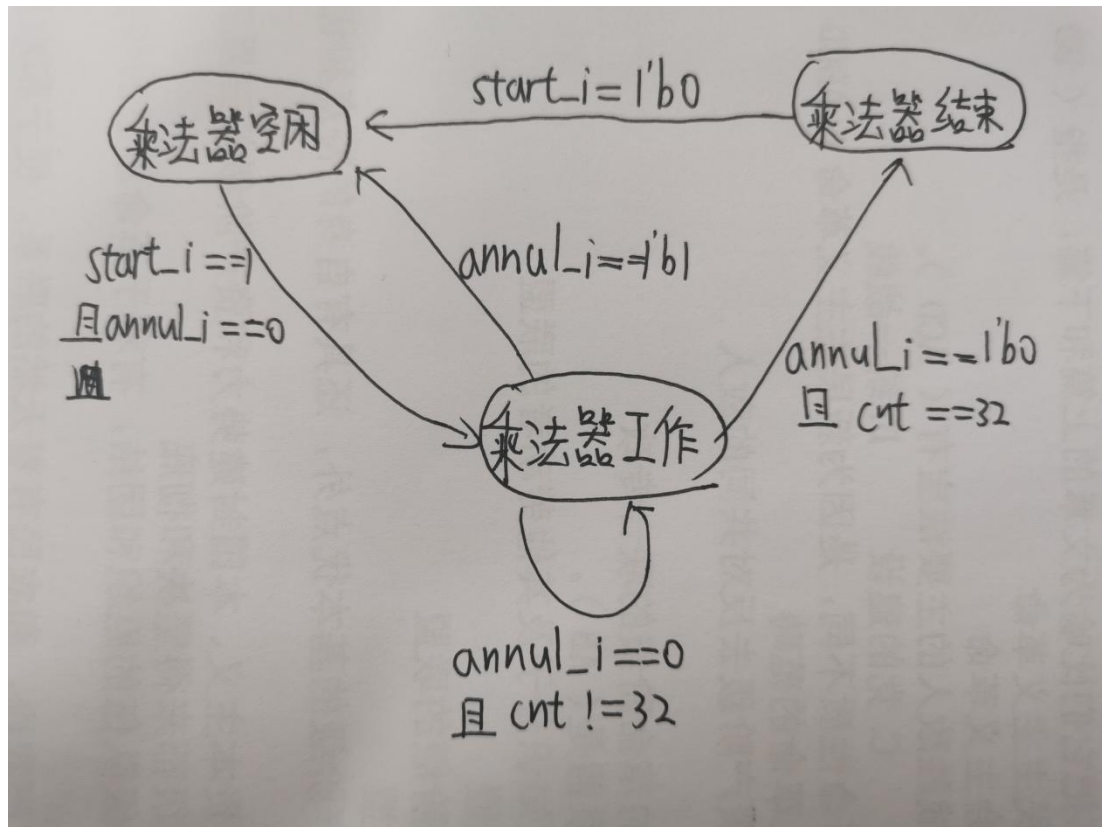
        emp_op1<={emp_op1[62:0],1'b0};
        temp_op2<={1'b0,temp_op2[31:1]};

        cnt <= cnt +1;      //乘法运算次数
    end else begin
        state <= 2'b11;
        cnt <= 6'b0000000;
        end

    end else begin
        state <= 2'b00;
    end
end
2'b11: begin      //乘法结束
    result_o  <=  (((opdata1_i[31]  ==  1'b1&&opdata2_i[31]  ==
1'b1)|| (opdata1_i[31] == 1'b0&&opdata2_i[31] == 1'b0)&&signed_mul_i == 1'b1)||signed_mul_i
== 1'b0)?mult1_acc:(~mult1_acc+1);
    ready_o <= 1'b1;
    if (start_i == 1'b0) begin
        state <= 2'b00;
        ready_o <= 1'b0;
        result_o <= {32'b0, 32'b0};
    end
end
end
endcase
end
end
endmodule

```

说明：乘法器与除法器最大的不同在于少了除数为零的状态，状态图如下：、



有了状态图我们了解到了最重要的信息：主要的运算在乘法器工作状态完成，其他部分只负责控制，因此我们只需要在工作状态进行修改，其他的不需要改变即可：

```

2'b00: begin
    //空闲
    if (start_i == 1'b1 && annul_i == 1'b0) begin
        state <= 2'b10;
        cnt <= 6'b000000;
        if(signed_mul_i == 1'b1 && opdata1_i[31] == 1'b1) begin
            //负数
            mult1_shift <= {32'b0, (~opdata1_i + 1)};
        end else begin
            mult1_shift <= {32'b0, opdata1_i};
        end
        if (signed_mul_i == 1'b1 && opdata2_i[31] == 1'b1 ) begin
            //负数
            mult2_shift <= (~opdata2_i + 1);
        end else begin
            mult2_shift <= opdata2_i;
        end
        begin
            mult1_acc <= 64'b0;
        end
    end else begin
        ready_o <= 1'b0;
        result_o <= {32'b0, 32'b0};
    end
end

```

```

end
End

```

当乘法器处于空闲时，会先进行操作数处理，如果是有符号运算，那么先将操作数转换成补码，并将累加结果变量 **acc** 初始化，将乘法次数清零，将状态变为 **10**，即将开始乘法计算。

```

2'b10: begin
    if(annul_i == 1'b0) begin          //进行乘法运算
        if(cnt != 6'b1000000) begin
            mult1_acc<=(mult2_shift[0]==1'b1)?(mult1_acc+mult1_shift):mult1_acc;
            mult1_shift<={mult1_shift[62:0],1'b0};
            mult2_shift<={1'b0,mult2_shift[31:1]};
            cnt <= cnt +1;              //乘法运算次数
        end else begin
            state <= 2'b11;
            cnt <= 6'b0000000;
        end
    end else begin
        state <= 2'b00;
    end
end
end

```

当 **cnt** 不为 32 时，执行乘法运算——**mult1\_shift** 是储存操作数 1 的移位结果，**mult2\_shift** 储存操作数 2 的移位结果，**mult1\_acc** 储存累加结果，当 **acc==32** 后，**state** 会转变为 11，意思是进入乘法结束状态，并将 **cnt** 重新清零。

```

2'b11: begin          //乘法结束
    result_o <= (((opdata1_i[31] == 1'b1&&opdata2_i[31] == 1'b1)|| (opdata1_i[31] ==
1'b0&&opdata2_i[31] == 1'b0)&&signed_mul_i == 1'b1)|| signed_mul_i ==
1'b0)?mult1_acc:(~mult1_acc+1);
    ready_o <= 1'b1;
    if (start_i == 1'b0) begin
        state <= 2'b00;
        ready_o <= 1'b0;
        result_o <= {32'b0, 32'b0};
    end
end
end

```

当状态达到结束的时候，若是有符号数，则根据正负数选择 **acc** 或 **acc** 补码赋值，若是无符号数，则直接将 **acc** 赋给 **result**，输出至乘法器外部。

乘法外部的暂停与除法的暂停相同，因此会在乘除法合并时阐述。

## B)合并乘除法器

思路：正如乘法器中讲的一样，除了工作状态有改变之外，其余状态没有区别（乘法只少了除数为零这一状态），因此主要的改变在 **DIVON**：

### 1) 文件

● **u\_div : div (div.v)**

### 2) 代码：

```

module div(

```

```

input wire rst,
input wire clk,
input wire signed_div_i,
input wire[31:0] opdata1_i,
input wire[31:0] opdata2_i,
input wire start_i,          ///是否开始除法运算
input wire annul_i,          ///是否取消除法运算，1 位取消
output reg[63:0] result_o,    ///除法运算结果
output reg ready_o,          ///除法运算是否结束
input wire [1:0]sel
);

```

除法器的模块，只多了 sel 输入，目的是标志运算类型：01 为乘法、10 为除法、默认 00。

```

wire [32:0] div_temp;
reg [5:0] cnt;                ///记录试商法进行了几轮
reg[64:0] dividend;          ///低 32 位保存除数、中间结果，第 k 次
迭代结束的时候 dividend[k:0]保存的就是当前得到的中间结果，
//dividend[31:k+1]保存的是被除数没有参与运算的部分，dividend[63:32]是每次迭代时的
的被减数
reg [1:0] state;              ///除法器处于的状态
reg[31:0] divisor;
reg[63:0] emp_op1;
reg[31:0] temp_op1;//shift1
reg[31:0] temp_op2;//shift2
reg[63:0] mult1_acc;
assign div_temp = {1'b0, dividend[63: 32]} - {1'b0, divisor};

```

变量声明里我们只加入了 emp\_op1、mult1\_acc,分别用于储存 64 位零扩展的操作数 1、乘法累加结果。

```

`DivFree: begin                ///除法器空闲
    if (start_i == `DivStart && annul_i == 1'b0) begin
        if(opdata2_i == `ZeroWord&&sel==2'b10) begin          ///如果除数为 0
            state <= `DivByZero;
        end else begin
            state <= `DivOn;                                     ///除数不为 0
            cnt <= 6'b000000;
            if(signed_div_i == 1'b1 && opdata1_i[31] == 1'b1) begin
                ///被除数为负数
                temp_op1 = ~opdata1_i + 1;
                emp_op1 = {32'b0,~opdata1_i + 1};
            end else begin
                emp_op1 = {32'b0,opdata1_i};
                temp_op1 = opdata1_i;
            end
            if (signed_div_i == 1'b1 && opdata2_i[31] == 1'b1 ) begin
                ///除数为负数

```

```

        temp_op2 = ~opdata2_i + 1;
    end else begin
        temp_op2 = opdata2_i;
    end
    mult1_acc <= 64'b0;
    if(sel==2'b10)begin
        dividend <= {`ZeroWord, `ZeroWord};
        dividend[32: 1] <= temp_op1;
        divisor <= temp_op2;
    end
end
end else begin
    ready_o <= `DivResultNotReady;
    result_o <= {`ZeroWord, `ZeroWord};
end
end
End

```

乘除法器空闲时，先做判断，如果是除法操作且除数为零，进入“除数为零”状态，否则进行数据处理，这个部分乘除法没有区别，只是乘法需要扩充 32 个 0 至 emp\_op1（64 位，给 op1 移位预留空间），且 acc 需要清零。如果是除法（10），则按照 if 语句正常进行。

```

`DivByZero: begin
    //除数为 0
    dividend <= {`ZeroWord, `ZeroWord};
    state <= `DivEnd;
end

`DivOn: begin
    //除数不为 0
    if(annul_i == 1'b0) begin
        //进行除法运算
        if(cnt != 6'b1000000&&sel==2'b10) begin
            if (div_temp[32] == 1'b1) begin
                dividend <= {dividend[63:0],1'b0};
            end else begin
                dividend <= {div_temp[31:0],dividend[31:0], 1'b1};
            end
            cnt <= cnt +1;
            //除法运算次数
        end else if(sel==2'b10) begin
            if ((signed_div_i == 1'b1) && ((opdata1_i[31] ^ opdata2_i[31]) == 1'b1)) begin
                dividend[31:0] <= (~dividend[31:0] + 1);
            end
            if ((signed_div_i == 1'b1) && ((opdata1_i[31] ^ dividend[64]) == 1'b1)) begin
                dividend[64:33] <= (~dividend[64:33] + 1);
            end
            state <= `DivEnd;
            cnt <= 6'b0000000;
        end
        if(cnt != 6'b1000000&&sel==2'b01) begin

```

```

mult1_acc <= (temp_op2[0]==1'b1)?(mult1_acc+emp_op1):mult1_acc;
    emp_op1 <= {emp_op1[62:0],1'b0};
    temp_op2 <= {1'b0,temp_op2[31:1]};
    cnt <= cnt +1;    //乘法运算次数
end else if(sel==2'b01) begin
    state <= 2'b11;
    cnt <= 6'b000000;
end

end else begin
    state <= `DivFree;
end

End

```

如果除数为零，进入相应状态，当乘除法器启动后，进入 ON 状态，根据 sel 进行选择，如果是 10，则进入除法的 if 语句，如果是 01，则进入乘法的 if 语句，乘法计算过程与 32 周期乘法器 mul\_32\_1 完全相同。无论是哪种计算，运算时 cnt 都会累加，结束后都会进入 END 状态。

```

`DivEnd: begin    //除法结束
    result_o <= (sel==2'b10)?{dividend[64:33], dividend[31:0]}:
        (sel==2'b01)?(((opdata1_i[31] == 1'b1&&opdata2_i[31] == 1'b1)|| (opdata1_i[31] ==
1'b0&&opdata2_i[31] == 1'b0)&&signed_div_i == 1'b1)|| signed_div_i ==
1'b0)?mult1_acc:(~mult1_acc+1):64'b0;
    ready_o <= `DivResultReady;
    if (start_i == `DivStop) begin
        state <= `DivFree;
        ready_o <= `DivResultNotReady;
        result_o <= {`ZeroWord, `ZeroWord};
    end

End

```

运算结束时，根据运算类型进行输出，如果为除法，按照第一行输出，如果是乘法，按照第二行输出，赋值规则与乘法器相同。

### 3) 乘除法器外部 (mul\_div.v)

```

assign sel=(inst_div|inst_divu)?2'b10:
    (inst_mult|inst_multu)?2'b01:
    2'b00;
assign stallreq_for_ex=(stallreq_for_div|stallreq_for_mul)?1'b1:1'b0;
always @ (*) begin
    if (rst) begin
        stallreq_for_div = `NoStop;
        div_opdata1_o = `ZeroWord;
        div_opdata2_o = `ZeroWord;
        div_start_o = `DivStop;
        signed_div_o = 1'b0;
    end
end

```



```

else begin
    stallreq_for_div = `NoStop;
    div_opdata1_o = `ZeroWord;
    div_opdata2_o = `ZeroWord;
    div_start_o = `DivStop;
    signed_div_o = 1'b0;
    case ({(inst_div | inst_mult), (inst_divu | inst_multu)})
        2'b10: begin
            if (div_ready_i == `DivResultNotReady) begin
                div_opdata1_o = data1;
                div_opdata2_o = data2;
                div_start_o = `DivStart;
                signed_div_o = 1'b1;
                stallreq_for_div = `Stop;
            end
            else if (div_ready_i == `DivResultReady) begin
                div_opdata1_o = data1;
                div_opdata2_o = data2;
                div_start_o = `DivStop;
                signed_div_o = 1'b1;
                stallreq_for_div = `NoStop;
            end
            else begin
                div_opdata1_o = `ZeroWord;
                div_opdata2_o = `ZeroWord;
                div_start_o = `DivStop;
                signed_div_o = 1'b0;
                stallreq_for_div = `NoStop;
            end
        end
    end
    2'b01: begin
        if (div_ready_i == `DivResultNotReady) begin
            div_opdata1_o = data1;
            div_opdata2_o = data2;
            div_start_o = `DivStart;
            signed_div_o = 1'b0;
            stallreq_for_div = `Stop;
        end
        else if (div_ready_i == `DivResultReady) begin
            div_opdata1_o = data1;
            div_opdata2_o = data2;
            div_start_o = `DivStop;
            signed_div_o = 1'b0;
            stallreq_for_div = `NoStop;
        end
    end
end

```

```

        end
    else begin
        div_opdata1_o = `ZeroWord;
        div_opdata2_o = `ZeroWord;
        div_start_o = `DivStop;
        signed_div_o = 1'b0;
        stallreq_for_div = `NoStop;
    end
end
end
default:begin
end
endcase
end
End

```

Sel 的赋值细节如上，ex 段的暂停由乘除法器控制（mul 在乘除法器合并后没有使用，但是可以起到标示作用，就没删除），时序逻辑未做太大改变，只是将判断条件改为：

```
case ({(inst_div | inst_mult),(inst_divu | inst_multu)})
```