组成原理实验课程第二次实验报告

实验名称：数据运算：定点乘法　班级：李涛老师　学生姓名：陆皓喆　学号：2211044

指导老师：董前琨　实验地点：实验楼A306　实验时间：2024.03.28

一、实验目的

1. 理解定点乘法的不同实现算法的原理，掌握基本实现算法。
2. 熟悉并运用verilog语言进行电路设计。
3. 为后续设计cpu的实验打下基础。

二、实验内容说明

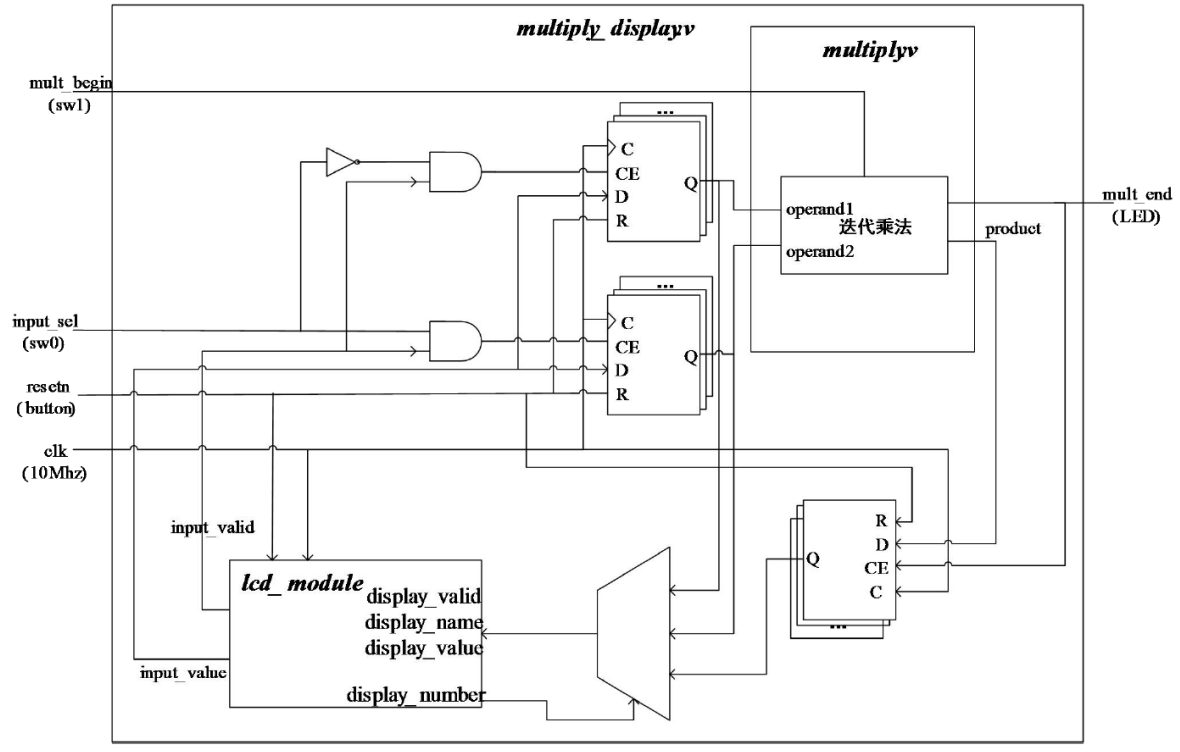
1.复现32位乘法器

复现指导书中的32位乘法器，使用两个32位寄存器，将其中的内容相乘，存储在一个64位寄存器中，并在实验箱上测试。

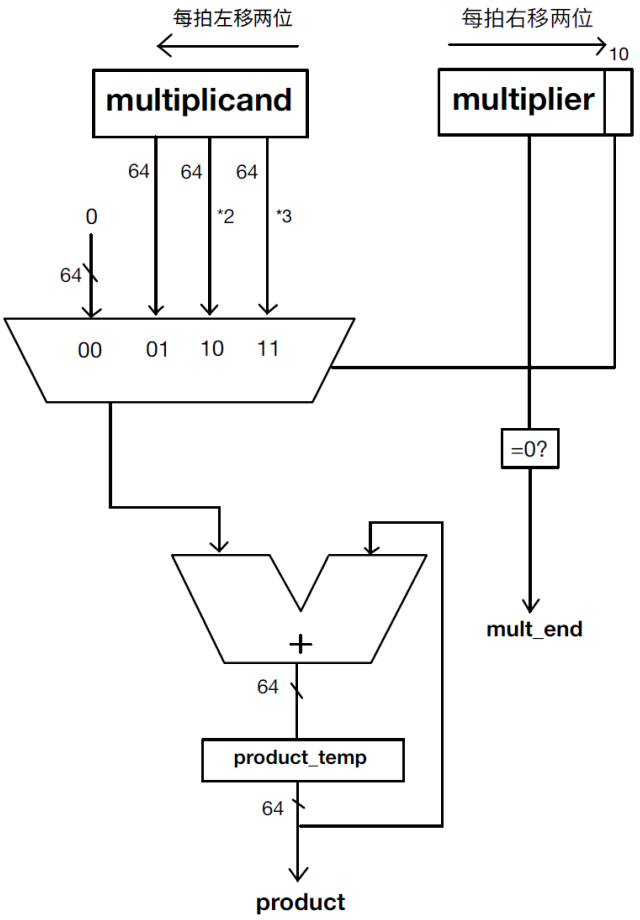
2.改进乘法器

要求在初始的乘法器的基础上进行改进，让算法能够在16个时钟节拍之内完成，提高乘法效率。上箱验证更改后的代码是否仍然正确。

三、实验原理图



整体的实验原理图与模板相同，只是在迭代乘法处修改了一部分内容，下面是我们对于迭代乘法中的具体修改。



想要缩短时钟节拍到16个，需要从每拍移动的位数入手。本次我们移动两位，就可以实现将时钟节拍缩短 。由于现在要判断乘数的最后 两位，因此数据选择器变成了四路数据选择器，对应着分别将被乘数乘上 0、1、 2、3倍然后累加到product_temp上 。这样，我们就可以一次性移动两位，可以在16个周期内完成计算。

四、实验步骤

1.multiply.v模块的修改

功能解释

本部分实现了输入两个数，进行相乘的操作。我们对原来的代码进行了修改，将移动的乘数和输出的结果都从一位变成了两位，从而实现了时钟周期的缩短。

代码部分

以下是我修改后的代码部分。

```
module multiply(
    input         clk,
    input         mult_begin,
    input  [31:0] mult_op1,
```

```
    input  [31:0] mult_op2,
    output [63:0] product,
    output        mult_end
);

    reg mult_valid;
    assign mult_end = mult_valid & ~(|multiplier);
    always @(posedge clk)
    begin
        if (!mult_begin || mult_end)
        begin
            mult_valid <= 1'b0;
        end
        else
        begin
            mult_valid <= 1'b1;
        end
    end

    wire        op1_sign;
    wire        op2_sign;
    wire [31:0] op1_absolute;
    wire [31:0] op2_absolute;
    assign op1_sign = mult_op1[31];
    assign op2_sign = mult_op2[31];
    assign op1_absolute = op1_sign ? (~mult_op1+1) : mult_op1;
    assign op2_absolute = op2_sign ? (~mult_op2+1) : mult_op2;


    reg  [63:0] multiplicand;
    always @ (posedge clk)
    begin
        if (mult_valid)
        begin
            multiplicand <= {multiplicand[61:0],2'b00};
        end
        else if (mult_begin)
        begin
            multiplicand <= {32'd0,op1_absolute};
        end
    end

    reg  [31:0] multiplier;
    always @ (posedge clk)
    begin
        if (mult_valid)
        begin
            multiplier <= {2'b00,multiplier[31:2]};
        end
        else if (mult_begin)
        begin
            multiplier <= op2_absolute;
        end
    end

    wire [63:0] partial_product1;
```

```
    wire [63:0] partial_product2;
    assign partial_product1 = multiplier[0] ? multiplicand : 64'd0;
    assign partial_product2 = multiplier[1] ? {multiplicand[62:0],1'b0}:64'd0;

    reg [63:0] product_temp;
    always @ (posedge clk)
    begin
        if (mult_valid)
        begin
            product_temp <= product_temp + partial_product1+partial_product2;
        end
        else if (mult_begin)
        begin
            product_temp <= 64'd0;
        end
    end

    reg product_sign;
    always @ (posedge clk)
    begin
        if (mult_valid)
        begin
              product_sign <= op1_sign ^ op2_sign;
        end
    end

    assign product = product_sign ? (~product_temp+1) : product_temp;
endmodule
```

代码解释与修改

首先，我们修改了乘法模块中multplicand的模块，原本是只移动一位，现在变成了移动两位，下面是我的修改部分：

```
    multiplicand <= {multiplicand[61:0],2'b00};
```

可以发现，原来的代码实现的是，将被乘数的前62位取出，再将最低位的一位设置成0，这样就完成了移动一位的操作。现在，我们修改为，取出被乘数的前61位，再将最后两位设置成0，这样我们就完成了左移两位的操作。

```
    multiplier <= {2'b00,multiplier[31:2]};
```

这句代码我们实现了每次迭代后，乘数右移两位的操作，原来是右移一位，那么现在修改为取出迭代前的第31位到第2位数，然后将最高位的两位设置成0，这样我们就右移两位成功了。

通过上面的修改，我们发现，移动两位的同时，我们需要考虑两位数字的可能出现的取值。在之前一位数字的时候，只有0和1两种情况，现在出现两位数字的时候，我们就需要考虑0,1,2,3,（00,01,10,11）这四种情况了。所以我们将原来的流程部分进行修改，将原来的二路选择器修改为四路选择器，我们下面解释修改后的选择器的代码：

```
    wire [63:0] partial_product1;
    wire [63:0] partial_product2;
    assign partial_product1 = multiplier[0] ? multiplicand : 64'd0;
    assign partial_product2 = multiplier[1] ? {multiplicand[62:0],1'b0}:64'd0;
```

我们在原来的一个product的基础上又增加了一个product，我们使用两位数来存储我们应该乘上的数，在上面我们说过，存在四种不同的乘数，包括0,1,2,3这四个。第一条语句是对partial_product1赋值。我们根据的是乘数的第0位即最低的一位，若乘数第0位是0，那么 partial_product1的值赋值为0；若乘数第0位是1，那么对应着就给partial_product1赋值为1倍的被乘数。第二条语句是对partial_product2赋值。我们根据的是乘数的第1位即第二低的那位。若乘数第1位为0，那么partial_product2赋值为0；若乘数第1位为1，那么我们就将被乘数左移一位之后再赋值给partial_product2。因为左移一位就相当于*2，因此实际上是把被乘数的2倍赋值给了partial_product2。

接下来，我们只需要修改最后的赋值语句就可以了，将其赋值为temp与两个product相加即可。

```
    product_temp <= product_temp + partial_product1+partial_product2;
```

我们来做进一步的解释，把两个 partial_product均累加到 partial_temp上。其中当最低两位为00时，partial_temp加上0，即0倍的被乘数；当最低两位为01时，partial_product1保存着1倍的被乘数而partial_product2为0，实现了对partial_temp加上1倍的被乘数；当最低两位为10时，partial_product1为0而partial_product2保存着2倍的被乘数，实现对partial_temp加上 2倍的被乘数；当最低两位为11时，partial_product1保存1倍的被乘数， partial_product2保存2倍的被乘数，实现对partial_temp加上3倍的被乘数。通过以上的分析，我们就可以发现，在上述修改中我们完成了对乘法器的修改。

当然，还有其他很多方法，比如我们可以先不看权重，对于赋值的时候将高位的部分修改为2倍即可。

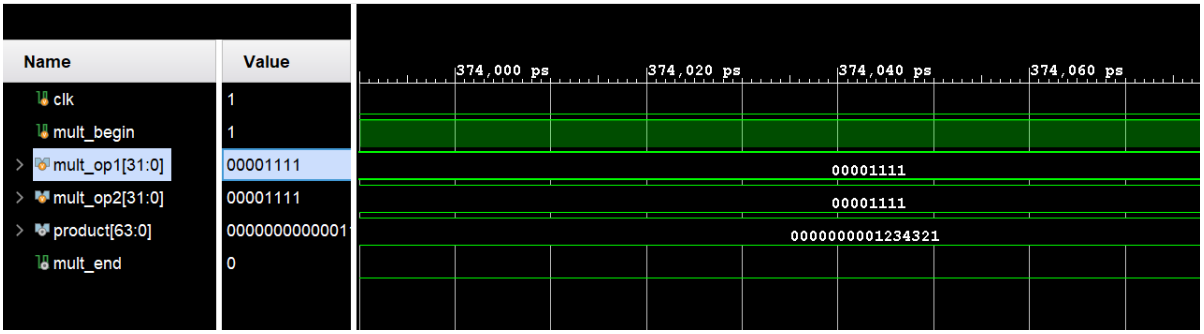
```
    wire [63:0] partial_product1;
    wire [63:0] partial_product2;
    assign partial_product1 = multiplier[0] ? multiplicand : 64'd0;
    assign partial_product2 = multiplier[1] ? multiplicand : 64'd0;
    product_temp <= product_temp + partial_product1+2*partial_product2;
```

2.其他三个模块

由于本次实验修改的是乘法器内部的语句，对于其他模块的功能没有需要修改的地方，所以不再展示。

五、实验结果分析

1.仿真验证



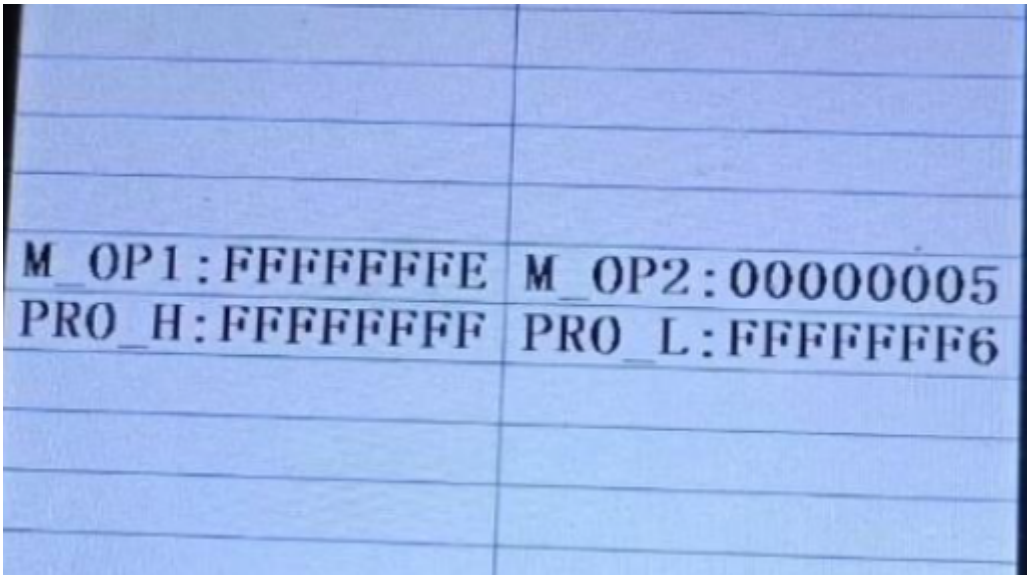
如上图所示，我们生成的两个随机数的十六进制为1111和1111，相乘后的结果为1234321，所以说明我们的仿真结果是正确的。

2.上箱验证

(1)正数乘正数

我们手动输入两个正数进行相乘，输入4和7，发现相乘得28（1C），说明我们的结果是正确的。

(2)正数乘负数



我们输入一个正数和一个负数，输入-2（FFFFFFFE）和5（00000005），相乘得到结果-10（FFFFFFF6），说明我们的结果是正确的。

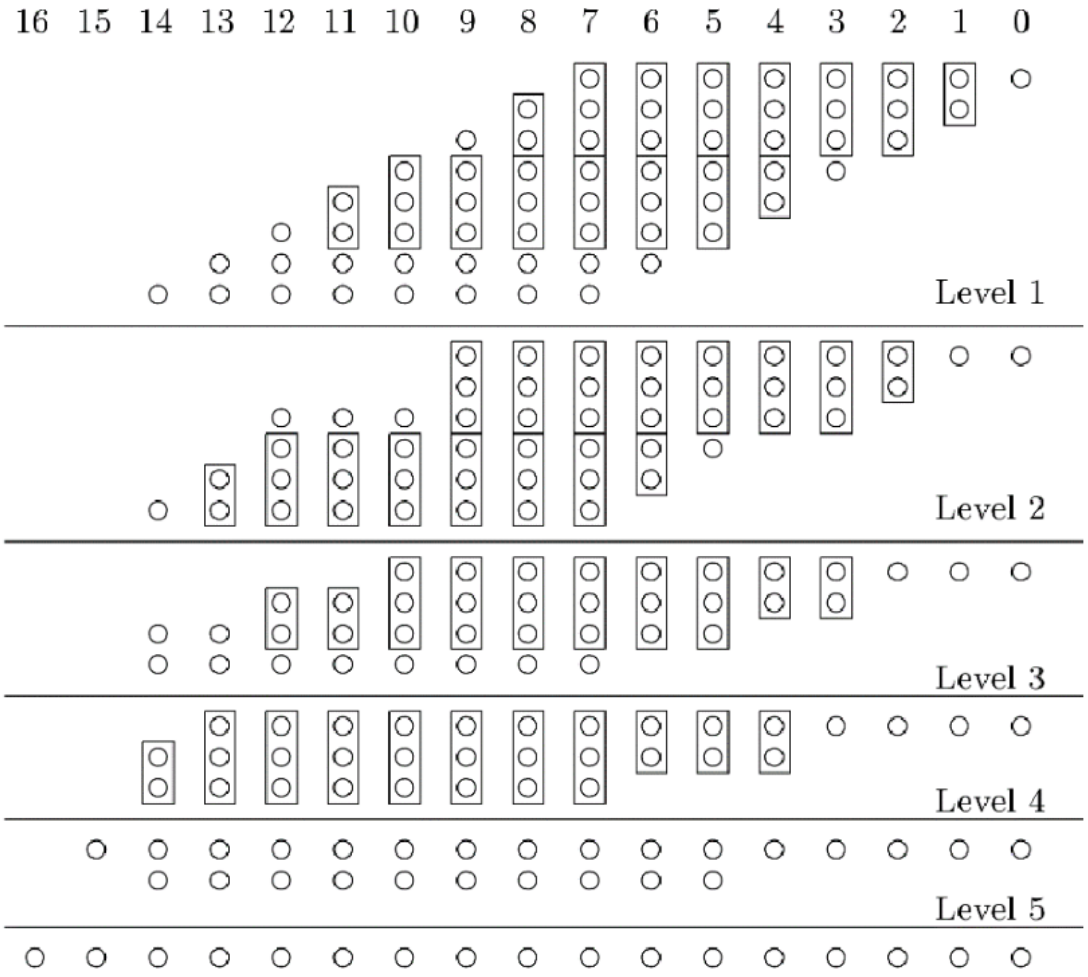
(3)负数乘负数

我们输入两个负数进行相乘，输入两个-1（FFFFFFFF），输出结果1（00000001），证明了我们的结果是正确的。

六、进一步分析

通过上网查询资料，我发现对于32位乘法器，还可以使用华莱士树的方法来进行优化。

华莱士树的主要实现方式利用多层来进行计算，从而加速时钟周期。



在github上，我找到了一个16位乘法器的代码库，我修改了他的代码，编写了以下32位乘法器的实现方式。修改的地方主要是，将PP输出的位宽以及输入的两个数的位宽进行了修改，然后对其他的部分进行相应的调试。

(1)PP.v

```
module PP
    (
    input [31:0] A,
    input [31:0] B,

    output [63:0]PP0,
    output [63:0]PP1,
    output [63:0]PP2,
    output [63:0]PP3,
    output [63:0]PP4,
    output [63:0]PP5,
    output [63:0]PP6,
    output [63:0]PP7,
    output [63:0]PP8,
    output [63:0]PP9,
```

```
    output [63:0]PP10,
    output [63:0]PP11,
    output [63:0]PP12,
    output [63:0]PP13,
    output [63:0]PP14,
    output [63:0]PP15
    );

    wire [63:0]PP[31:0];

    genvar i;
    generate
        for(i = 0; i <= 31; i = i + 1)
        begin: PP_LOOP
            assign PP[i][63:0] = {32'h00000,(A[i] * B)} << i;
        end
    endgenerate

    assign PP0 = PP[0][63:0];
    assign PP1 = PP[1][63:0];
    assign PP2 = PP[2][63:0];
    assign PP3 = PP[3][63:0];
    assign PP4 = PP[4][63:0];
    assign PP5 = PP[5][63:0];
    assign PP6 = PP[6][63:0];
    assign PP7 = PP[7][63:0];
    assign PP8 = PP[8][63:0];
    assign PP9 = PP[9][63:0];
    assign PP10 = PP[10][63:0];
    assign PP11 = PP[11][63:0];
    assign PP12 = PP[12][63:0];
    assign PP13 = PP[13][63:0];
    assign PP14 = PP[14][63:0];
    assign PP15 = PP[15][63:0];

endmodule
```

(2)CSA.v

```
module CSA #(parameter N = 64)
    (
    input [N-1:0] A,
    input [N-1:0] B,
    input [N-1:0] C,

    output [N-1:0] SUM,
    output [N-1:0] CARRY
    );

    assign SUM = A ^ B ^ C;
    assign CARRY[0] = 0;
    assign CARRY[N-1:1] = (A&B) | (A&C) | (B&C);
```

```
endmodule
```

(3)CLA.v

```
module CLA (
                A,
                B,
                CIN,
                SUM,
                COUT);
    parameter SIZE = 64;

    input [SIZE - 1:0] A;
    input [SIZE - 1:0] B;
    input CIN;
    output [SIZE - 1:0] SUM;
    output COUT;

//assign {carry_out, sum} = A + B + CIN;

        wire [SIZE - 1:0] G;
        wire [SIZE - 1:0] P;
        wire [SIZE:0] C;

        genvar j, i;
        generate
 //assume C in is zero
        assign C[0] = CIN;

 //carry generator
        for(j = 0; j < SIZE; j = j + 1) begin: carry_generator
            assign G[j] = A[j] & B[j];
            assign P[j] = A[j] | B[j];
            assign C[j+1] = G[j] | P[j] & C[j];
        end

 //carry out
        assign COUT = C[SIZE];

 //calculate sum
 //assign sum[0] = A[0] ^ B ^ CIN;
        for(i = 0; i < SIZE; i = i+1) begin: sum_without_carry
            assign SUM[i] = A[i] ^ B[i] ^ C[i];
        end
    endgenerate
endmodule
```

(4)Wallace_Tree_Mult.v

```
module Wallace_Tree_Mult(
    input [31:0]A,
    input [31:0]B,
    output [63:0]C,
    output carry
```

```
    );

    wire  [63:0] pp [31:0];

    PP partial_products(
        A,
        B,
        pp[0][63:0],
        pp[1][63:0],
        pp[2][63:0],
        pp[3][63:0],
        pp[4][63:0],
        pp[5][63:0],
        pp[6][63:0],
        pp[7][63:0],
        pp[8][63:0],
        pp[9][63:0],
        pp[10][63:0],
        pp[11][63:0],
        pp[12][63:0],
        pp[13][63:0],
        pp[14][63:0],
        pp[15][63:0],

        );

    // STAGE1
    wire [63:0] sum_s11, carry_s11, sum_s12, carry_s12, sum_s13, carry_s13,
sum_s14, carry_s14, sum_s15, carry_s15;
    CSA s11 (pp[0][63:0], pp[1][63:0], pp[2][63:0], sum_s11[63:0],
carry_s11[63:0]);
    CSA s12 (pp[3][63:0], pp[4][63:0], pp[5][63:0], sum_s12[63:0],
carry_s12[63:0]);
    CSA s13 (pp[6][63:0], pp[7][63:0], pp[8][63:0], sum_s13[63:0],
carry_s13[63:0]);
    CSA s14 (pp[9][63:0], pp[10][63:0], pp[11][63:0], sum_s14[63:0],
carry_s14[63:0]);
    CSA s15 (pp[12][63:0], pp[13][63:0], pp[14][63:0], sum_s15[63:0],
carry_s15[63:0]);

    //STAGE2
    wire [63:0] sum_s21, carry_s21, sum_s22, carry_s22, sum_s23, carry_s23;
    CSA s21 (sum_s11[63:0], carry_s11[63:0], sum_s12[63:0], sum_s21[63:0],
carry_s21[63:0]);
    CSA s22 (carry_s12[63:0], sum_s13[63:0], carry_s13[63:0], sum_s22[63:0],
carry_s22[63:0]);
    CSA s23 (sum_s14[63:0], carry_s14[63:0], sum_s15[63:0], sum_s23[63:0],
carry_s23[63:0]);

    //STAGE3
    wire [63:0] sum_s31, carry_s31, sum_s32, carry_s32;
    CSA s31 (sum_s21[63:0], carry_s21[63:0], sum_s22[63:0], sum_s31[63:0],
carry_s31[63:0]);
    CSA s32 (carry_s22[63:0], sum_s23[63:0], carry_s23[63:0], sum_s32[63:0],
carry_s32[63:0]);
```

```
    //STAGE4
    wire [63:0] sum_s41, carry_s41, sum_s42, carry_s42;
    CSA s41 (sum_s31[63:0], carry_s31[63:0], sum_s32[63:0], sum_s41[63:0],
carry_s41[63:0]);
    CSA s42 (carry_s32[63:0], carry_s15[63:0], pp[15][63:0], sum_s42[63:0],
carry_s42[63:0]);

    //STAGE5
    wire [63:0] sum_s51, carry_s51;
    CSA s51 (sum_s41[63:0], carry_s41[63:0], sum_s42[63:0], sum_s51[63:0],
carry_s51[63:0]);

    //STAGE6
    wire [63:0] sum_s61, carry_s61;
    CSA s61 (sum_s51[63:0], carry_s51[63:0], carry_s42[63:0], sum_s61[63:0],
carry_s61[63:0]);

    CLA Carry_Lookahead_Adder(
        .A(sum_s61),
        .B(carry_s61),
        .CIN(1'b0),
        .COUT(carry),
        .SUM(C)
        );


endmodule
```

我们运行工程，发现时钟周期确实得到了提升，从16周期来到了8周期。

七、总结感想

1. 通过本次实验，我进一步掌握了vivado的使用以及 Verilog语句的编写，对数据的移位操作有了比较好的理解与应用，同时对乘法器的原理以及硬件实现有了 较好的掌握。
2. 对于时钟周期的优化，我们要考虑到移动位数的变化以及通过位数的改变能否影响到具体的时钟周期，我们通过将32个时钟周期长度进行一次性移动两位，实现了时钟周期的缩短，完成了优化。我们发现，其实在组成原理的代码中，我们通过修改每次移动的位数来完成程序性能的优化，也体会到了对于同一个问题，不仅仅是表面的代码区别，在内部也有很大的区别。
3. 在对自己的16周期进行优化的时候，在网上学习了WallaceTree的用法，并且对其进行了修改。