课程名称：数字集成电路设计　　开课学院：微电子学院
姓名：xxx　　学号：xxxxxx
报告提交日期：2025.1.19

实验二 乘法器

一、实验目的

使用不同方法设计 8bit 无符号整数乘法器：

A. 基本乘法器构建方法：将被乘数和乘数按位与运算后形成部分积，并将部分积逐个相加，得到结果。运算过程如图 1 所示。

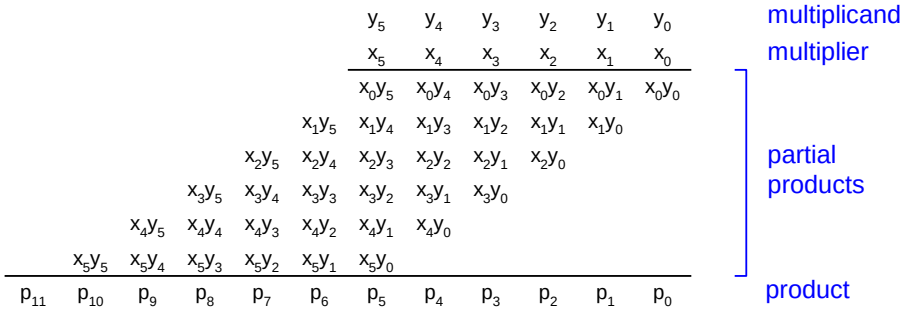


图 1. 乘法计算原理

B. 使用华莱士树构建方法：此方法的基本原理是采用进位保存加法器（CSA）将参与运算的三个数变成两个数输出，从而减少一个相加数，通过这种方法可以对图 1 中同一列（p）的部分积进行化简，化简的结果是最后每个位数的 S 构成一个加数，每个位数的进位 C 构成另外一个加数，将 S 和 C 通过高性能加法器相加即可求得结果。图 2 为 4 位华莱士树乘法器结构。

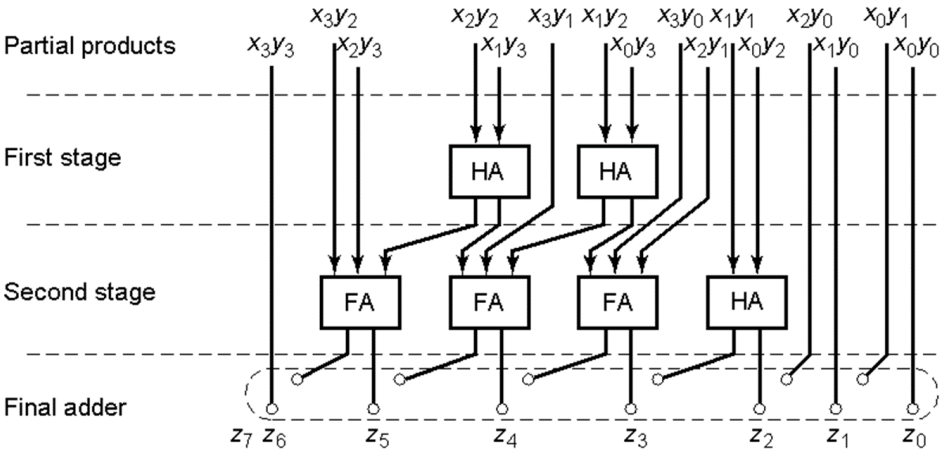


图 2. 4 位华莱士树结构乘法器二、实验原理与设计思路

（1）8 位基本乘法器

实验原理：

二进制乘法的基本操作是按位相乘并进行累加。假设我们要计算两个 8 位二进制数 $A = a_7a_6a_5a_4a_3a_2a_1a_0$和 ，其中$B = b_7b_6b_5b_4b_3b_2b_1b_0$ 每个数字

表示二进制位。乘法过程可以分为以下几个步骤：

首先，将第二个数 B 的每一位与第一个数 A 的每一位进行按位与（AND）运算，并生成部分积。具体来说：

$p_0 = A \times b_0$这相当于将 A 和 b_0 按位与操作得到的结果，表示 A 与 b_0 的乘积。

$p_1 = A \times b_1$ 这是将 A 和 b_1 按位与操作得到的结果。

以此类推，直到 $p_7 = A \times b_7$。

其次，每个部分积 p_i 是一个 16 位的数，表示 A 与 B 中相应位的乘积。每个部分积 p_i 都需要根据 B 中相应位的位置进行左移。对于 p_0，不需要移位，它在最低位；对于 p_1，需要左移 1 位，表示 b_1 参与了 AAA 的乘法，依此类推。

最后，将所有的部分积按适当的位置加到一起，得到最终的乘积。这里的加法不是简单的按位加法，而是带进位的加法。每次加法都会处理进位，并将结果存储在一个更高位的存储单元中。

设计思路：

代码主要分为两个模块，分别是 multiply 模块和 full_add 模块。multiply 模块负责计算乘法运算，并且根据乘法的规律生成部分积，进行左移操作，最后通过加法器将部分积累加，得到最终的乘积结果。而 full_add 模块则用于实现二进制数的加法。**为了更加准确地比较两种乘法器的区别，我们自己实现加法器，不使用+，确保两个乘法器的加法器综合后的电路一致。**

在 multiply 模块中，首先定义了多个寄存器数组 temp 和 temp_reg 来分别存储未移位和已移位的部分积。通过对 a 和 b 按位与操作，生成每一位的部分积，存储在 temp 中。然后，将每个部分积根据其对应的位置进行左移操作，并将结果保存在 temp_reg 中。移位的数量由乘数 b 的位数决定。之后，利用 generate 语句，调用 full_add 模块对移位后的部分积进行逐步加法操作。每次加法都会处理进位，最终得到一个累加的结果。

在加法操作的实现中，full_add 模块逐位进行加法。通过遍历每一位，进行按位加法并处理进位。进位的计算通过按位与运算实现，并且每次加法操作都会将进位传递到下一位。最终的加法结果即为两个加数的和，它被输出为 sum。

（2）8 位华莱士树乘法器

实验原理：

课程名称：数字集成电路设计　　开课学院：微电子学院
姓名：xxx　　学号：xxxxxx
报告提交日期：2025.1.19

（1）部分积的产生与问题

在乘法运算中，以两个 n 位二进制数相乘为例，设这两个数为$A=a_{n-1}a_{n-2}\cdots a_0$和 $B=b_{n-1}b_{n-2}\cdots b_0$，根据乘法规则，其结果是多个部分积（Partial Product）之和。具体计算过程为：

每一项$a_ib_j2^{i+j}$即为一个部分积。例如，当$n=4$ 时,$A=a_3a_2a_1a_0,B=b_3b_2b_1b_0$相乘后会产生$4\times4=16$个部分积。这些部分积的求和若采用传统的逐位相加方式，随着位数增加，运算复杂度将大幅上升，导致运算速度变慢。

（2）华莱士树的核心思想——分组并行相加

华莱士树的核心思想是通过分组并行相加的方式，减少加法运算的级数，从而提高运算速度。具体操作如下：

第一轮分组：将所有部分积按三个一组进行分组（若部分积数量不是 3 的倍数，最后一组可能不足三个）。例如，假设有 9 个部分积 $P_1,P_2,\cdots,P_9$ ，可分为三组：$(P_1,P_2,P_3),(P_4,P_5,P_6),(P_7,P_8,P_9)$ 。

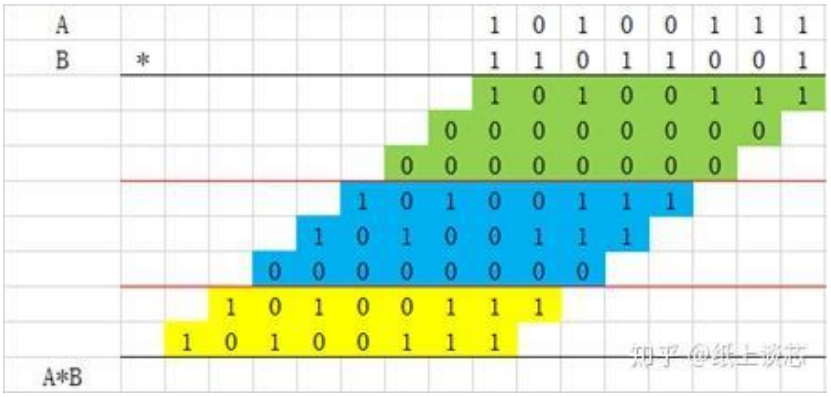
组内运算：对于每一组，使用全加器（Full - Adder，FA）进行运算。全加器有三个输入（三个部分积）和两个输出：和（Sum） 与进位（Carry） 。其逻辑表达式为：

求和： $S=A\oplus B\oplus C$

求进位： C=(A&B)|(A&C)|(B&C)

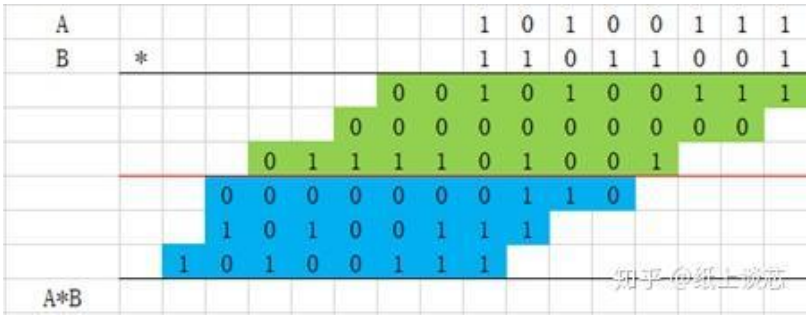
其中，A、B、C 为三个部分积输入，。通过全加器，每组三个部分积可转化为一个和与一个进位，这样部分积的数量大致减少为原来的 2/3。

（3）迭代分组：将第一轮得到的和与进位作为新的输入，再次按三个一组进行分组，并重复上述全加器运算过程。不断迭代此过程，直到剩余的部分积数量减少到可以用常规加法器（如并行加法器）高效处理的程度。

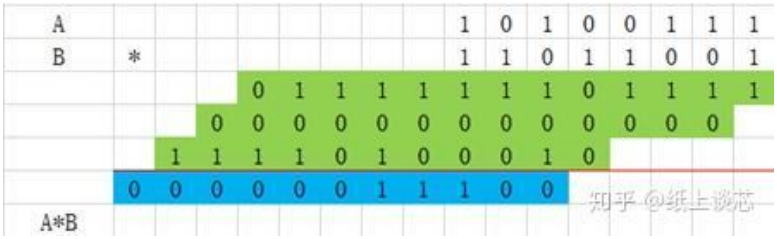


第一层加法器，处理(P_1,P_2,P_3)与(P_4,P_5,P_6)，产生C_1,S_1 和C_2,S_2

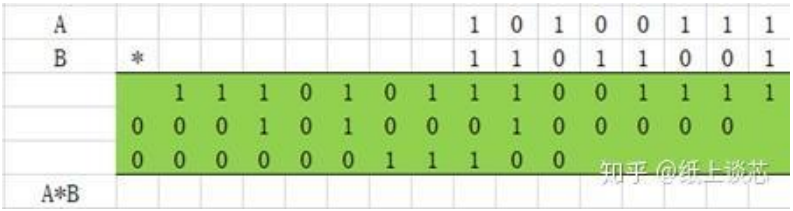
课程名称：数字集成电路设计　　开课学院：微电子学院
姓名：xxx　　学号：xxxxxx
报告提交日期：2025.1.19



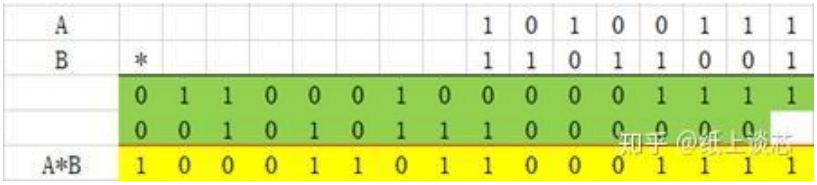
第二层加法器，处理(C_1,S_1,S_2)与(P_6,P_7,C_2)，产生C_3,S_3 和C_4,S_4



第三层加法器，处理(C_3,S_3,S_4)，产生C_5,S_5 。



第四层加法器，处理(C_5,S_5,C_4)，产生C_6,S_7 。



第五层加法器，处理$(C_6,S_7,0)$，输出最终结果。

设计思路：

代码主要分为两个模块，分别是 multiply 模块和 full_add 模块。multiply 模块负责计算乘法运算，并且根据乘法的规律生成部分积，进行左移操作，最后通过多层加法器级联将部分积累加，得到最终的乘积结果。而 full_add 模块则用于 16 位和与进位输出。

在 multiply 模块中，首先定义了寄存器数组 temp_reg 来存储移位后的部分积。之后，调用 full_add 模块，以前文分析的加法器层次对部分积进行逐步加法操作，最终得到一个累加的结果。

在加法操作的实现中，full_add 模块并不是一位全加器的级联，而是全加

器的并联实现，同时输入三个 16 位输出，输出和与进位。

二、编码

（1）8 位基本乘法器

RTL 设计代码：

```
`timescale 1ns/1ps
module multiply(
    input [7:0] a,              // 8 位输入 a
    input [7:0] b,              // 8 位输入 b
    output [15:0] sum           // 16 位结果
);

// 存储部分乘积
reg [15:0] temp[7:0];
reg [15:0] temp_reg[7:0];  // 存储偏移后的部分乘积
integer i, j;
// 计算部分乘积
always @(*) begin
    for (i = 0; i < 8; i = i + 1) begin
        temp[i] = 16'b0;            // 清空临时存储
        for (j = 0; j < 8; j = j + 1) begin
            // 计算部分乘积
            temp[i][j] = a[j] & b[i];
        end
    end
    // 将乘积按位数移位
    temp_reg[0] = temp[0];
    temp_reg[1] = temp[1] << 1;
    temp_reg[2] = temp[2] << 2;
    temp_reg[3] = temp[3] << 3;
    temp_reg[4] = temp[4] << 4;
    temp_reg[5] = temp[5] << 5;
    temp_reg[6] = temp[6] << 6;
    temp_reg[7] = temp[7] << 7;
end
// 中间信号定义
wire [15:0] sum_1, sum_2, sum_3, sum_4, sum_5, sum_6, sum_7;
// 加法模块实例化
full_add u0(.a(temp_reg[0]), .b(temp_reg[1]), .sum(sum_1));
```

```
full_add u1(.a(sum_1), .b(temp_reg[2]), .sum(sum_2));
full_add u2(.a(sum_2), .b(temp_reg[3]), .sum(sum_3));
full_add u3(.a(sum_3), .b(temp_reg[4]), .sum(sum_4));
full_add u4(.a(sum_4), .b(temp_reg[5]), .sum(sum_5));
full_add u5(.a(sum_5), .b(temp_reg[6]), .sum(sum_6));
full_add u6(.a(sum_6), .b(temp_reg[7]), .sum(sum_7));
// 最终结果
assign sum = sum_7;
endmodule
module full_add(
      input [15:0]a,
      input [15:0]b,
      output    reg [15:0]sum
      );
reg [15:0] c;
integer i;
always@(*)
begin
      c[0]=1'b0;
      for(i=0;i<=15;i=i+1)
      begin
            c[i+1]=(a[i]&b[i])|((a[i]|b[i])&c[i]);
            sum[i]=a[i]^b[i]^c[i];

      end
end
endmodule
```

Testbench 代码：

```
`timescale 1ns/1ps
module multiply_tb;

      // 输入信号
      reg [7:0] a;
      reg [7:0] b;

      // 输出信号
      wire [15:0] multi;   // 修改为 16 位

      // 实例化 multiply 模块
      multiply uut (
            .a(a),
```

```
        .b(b),
        .sum(multi)
    );

    // 时钟和初始化信号
    initial begin
        // 初始化输入信号
        a = 8'b00000000;
        b = 8'b00000000;

        // 测试不同的输入值
        #10; a = 8'b00000001; b = 8'b00000001;   // 1 * 1 = 1
        #10; a = 8'b00000010; b = 8'b00000011;   // 2 * 3 = 6s
        #10; a = 8'b00000101; b = 8'b00000011;   // 5 * 3 = 15
        #10; a = 8'b00001111; b = 8'b00000111;   // 15 * 7 = 105
        #10; a = 8'b11111111; b = 8'b11111111;   // 255 * 255 = 65025
        #10; a = 8'b10101010; b = 8'b11110000;   // 170 * 204 = 34680
        #10; a = 8'b11110000; b = 8'b10101010;   // 240 * 170 = 40800
        #10; a = 8'b00000000; b = 8'b11111111;   // 0 * 0 = 0
        #10; a = 8'b11111111; b = 8'b00000001;   // 255 * 1 = 255
        #10; a = 8'b11111111; b = 8'b00000000;   // 255 * 0 = 0
        // 测试结束
        $finish;
    end
    // 监视输出
    initial begin
        $monitor("Time = %0t, a = %d, b = %d, multi = %d", $time, a, b, multi);
    end
 initial begin
        $fsdbDumpfile("a.fsdb");
        $fsdbDumpvars();
    end
endmodule
```

（2）8 位华莱士树乘法器

RTL 设计代码：

```
`timescale 1ns/1ps
module wallace_tree(
    input [7:0] a,              // 8 位输入 a
    input [7:0] b,              // 8 位输入 b
```

```
    output [15:0] multi          // 16 位结果
);
reg [15:0] temp_reg[7:0];       // 存储偏移后的部分乘积
integer i, j;
// 计算部分乘积
always@(*)
begin
    for (i = 0; i < 8; i = i + 1) begin
        temp_reg[i] = 16'b0;
        if(b[i]==1) begin
            temp_reg[i]=a << i ; // 计算部分乘积
        end
    end
end
// 中间信号定义
wire [15:0] carry_1, carry_2, carry_3, carry_4, carry_5, carry_6;
wire [15:0] sum_1, sum_2, sum_3, sum_4, sum_5, sum_6;
// 使用全加器计算各部分的和
full_add                                                    inst_full_add1
(.a(temp_reg[0]), .b(temp_reg[1]), .c(temp_reg[2]), .carry(carry_1), .sum(sum_1));
full_add                                                    inst_full_add2
(.a(temp_reg[3]), .b(temp_reg[4]), .c(temp_reg[5]), .carry(carry_2), .sum(sum_2));
full_add inst_full_add3 (.a(sum_1), .b(carry_1<<1), .c(sum_2), .carry(carry_3), .sum(sum_3));
full_add                                                    inst_full_add4
(.a(temp_reg[6]), .b(temp_reg[7]), .c(carry_2<<1), .carry(carry_4), .sum(sum_4));
full_add inst_full_add5 (.a(carry_3<<1), .b(sum_3), .c(sum_4), .carry(carry_5), .sum(sum_5));
full_add inst_full_add6 (.a(carry_4<<1), .b(carry_5<<1), .c(sum_5), .carry(carry_6), .sum(sum_6));
full_add inst_full_add7 (.a(carry_6<<1), .b(sum_6),.c(16'b0), .sum(multi));
endmodule


module full_add(
    input [15:0]a,
    input [15:0]b,
    input [15:0]c,
    output   reg [15:0]carry,
    output   reg [15:0]sum
    );
integer i;
always@(*)
begin
    for(i=0;i<=15;i=i+1)
    begin
        sum[i] = a[i] ^ b[i] ^ c[i];   // 求和
```

```
            carry[i] = (a[i] & b[i]) | (b[i] & c[i]) | (a[i] & c[i]);  // 进位
        end
    end
    endmodule
```

Testbench 代码：

```
`TIMESCALE 1NS/1PS

MODULE WALLACE_TREE_TB;

    // 输入信号
    REG [7:0] A;
    REG [7:0] B;
    // 输出信号
    WIRE [15:0] MULTI;
    // 实例化 WALLACE_TREE 模块
    WALLACE_TREE UUT ( .A(A), .B(B), .MULTI(MULTI) );
    // 时钟和初始化信号
    INITIAL BEGIN
        // 初始化输入信号
        A = 8'B00000000;
        B = 8'B00000000;
        // 等待几时间单位以确保初始状态
        #10;
        // 测试开始

        // 测试不同的输入值
        // 用于验证的不同输入
        #10; A = 8'B00000001; B = 8'B00000001;  // 1 * 1 = 1
        #10; A = 8'B00000010; B = 8'B00000011;  // 2 * 3 = 6S
        #10; A = 8'B00000101; B = 8'B00000011;  // 5 * 3 = 15
        #10; A = 8'B00001111; B = 8'B00000111;  // 15 * 7 = 105
        #10; A = 8'B11111111; B = 8'B11111111;  // 255 * 255 = 65025
        #10; A = 8'B10101010; B = 8'B11110000;  // 170 * 204 = 34680
        #10; A = 8'B11110000; B = 8'B10101010;  // 240 * 170 = 40800
        #10; A = 8'B00000000; B = 8'B11111111;  // 0 * 0 = 0
        #10; A = 8'B11111111; B = 8'B00000001;  // 255 * 1 = 255
        #10; A = 8'B11111111; B = 8'B00000000;  // 255 * 0 = 0
        // 测试结束
        $FINISH;
    END
    // 监视输出
```

```
    INITIAL BEGIN
        $MONITOR("TIME = %0T, A = %D, B = %D, MULTI = %D", $TIME, A, B, MULTI);
    END
ENDMODULE
```

三、仿真验证

（1）8 位基本乘法器

基本**乘法**测试：

测试：a = 1, b = 1

预期输出：multi = 1

测试：a = 2，b = 3

预期输出：multi = 6

测试：a = 15，b = 7

预期输出：multi = 105

边界值测试：

测试：a = 255，b = 255

预期输出：multi = 65025

测试：a = 0，b = 0

预期输出：multi = 0

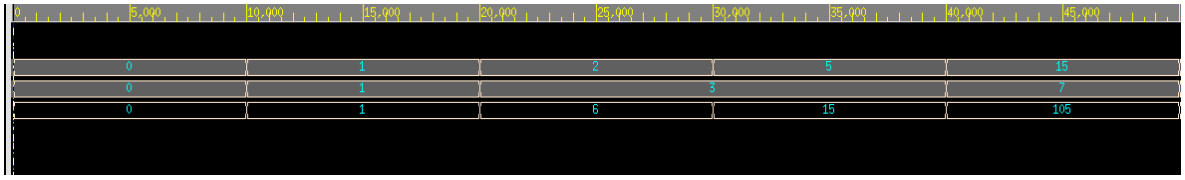
乘零测试：

测试：a = 0，b = 255

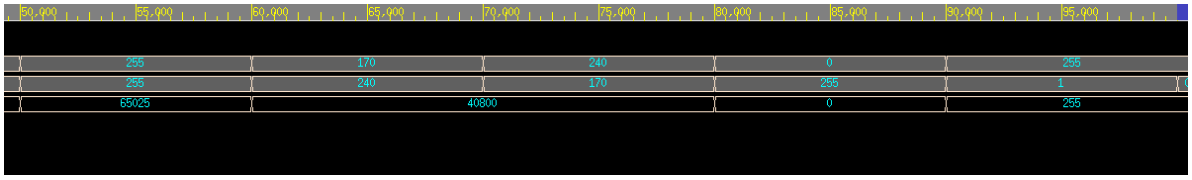
预期输出：multi = 0

```
Time = 0, a =   0, b =   0, multi =     0
Time = 10000, a =   1, b =   1, multi =     1
Time = 20000, a =   2, b =   3, multi =     6
Time = 30000, a =   5, b =   3, multi =    15
Time = 40000, a =  15, b =   7, multi =   105
Time = 50000, a = 255, b = 255, multi = 65025
Time = 60000, a = 170, b = 240, multi = 40800
Time = 70000, a = 240, b = 170, multi = 40800
Time = 80000, a =   0, b = 255, multi =     0
Time = 90000, a = 255, b =   1, multi =   255
```

图表 1：仿真输出



图表 2：仿真波形一



图表 3：仿真波形二

由仿真结果可知，加法器逻辑正确。

（2）8 位华莱士树乘法器

基本**乘法**测试：

测试：a = 1, b = 1

预期输出：multi = 1

测试：a = 2, b = 3

预期输出：multi = 6

测试：a = 15, b = 7

预期输出：multi = 105

边界值测试：

测试：a = 255, b = 255

预期输出：multi = 65025

测试：a = 0, b = 0

预期输出：multi = 0

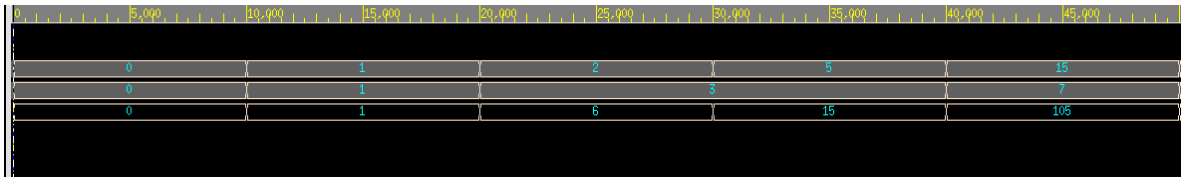
乘零测试：

测试：a = 0, b = 255

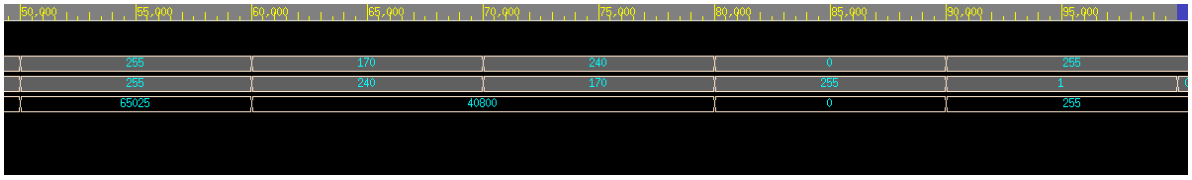
预期输出：multi = 0

```
Time = 0, a =   0, b =   0, multi =     0
Time = 10000, a =   1, b =   1, multi =     1
Time = 20000, a =   2, b =   3, multi =     6
Time = 30000, a =   5, b =   3, multi =    15
Time = 40000, a =  15, b =   7, multi =   105
Time = 50000, a = 255, b = 255, multi = 65025
Time = 60000, a = 170, b = 240, multi = 40800
Time = 70000, a = 240, b = 170, multi = 40800
Time = 80000, a =   0, b = 255, multi =     0
Time = 90000, a = 255, b =   1, multi =   255
```

图表 4：仿真输出



图表 5：仿真波形一



图表 6：仿真波形二

由仿真结果可知，加法器逻辑正确。

四、综合

（1）8 位基本乘法器

4.1.1 约束条件：

虚拟时钟 virtual_clk，其周期为 40ns（即频率为 25MHz）,无时钟抖动。

输入信号相对于 virtual_clk 时钟的最大延迟为 3ns，最小延迟为 0ns。

输出信号相对于 virtual_clk 时钟的最大延迟为 3ns，最小延迟为 0ns。

无面积、功耗约束。

4.1.2 综合结果与分析：

能达到的最大频率

a[0] (in)	0.00	23.00 f
……		
data arrival time		36.48 f

通过 8 位基本乘法器的最长延时为 12.7ns，故乘法器能达到的最大频率约为 74MHz（忽略保持时间和建立时间）。由于基本乘法器的输出是由 8 个 16 位加法器级联形成，故延时较长。

面积

Number of ports:	368
Number of nets:	846
Number of cells:	501
Number of combinational cells:	493
Number of sequential cells:	1
Number of macros/black boxes:	0
Number of buf/inv:	114
Number of references:	9

```
Combinational area:                5467.583024
Buf/Inv area:                       517.446004
Noncombinational area:                0.000000
Macro/Black Box area:                 0.000000
Net Interconnect area:      undefined  (Wire load has zero net
area)

Total cell area:                   5467.583024
```

8 位基本乘法器需要总的逻辑门面积约为 5467 单位。由于基本乘法器需要 8 个 16 位加法器，所以消耗面积比较大。

（2）8 位华莱士树乘法器

4.2.1 约束条件：

时钟 clk，其周期为 20ns（即频率为 50MHz），无时钟抖动。

输入信号相对于 virtual_clk 时钟的最大延迟为 3ns，最小延迟为 0ns。

输出信号相对于 virtual_clk 时钟的最大延迟为 3ns，最小延迟为 0ns。

无面积、功耗约束。

4.2.2 综合结果与分析：

能达到的最大频率

```
a[3] (in)                                0.00       23.00 f
……
Data arrival time                                    27.36
```

通过 8 位华莱士树乘法器的最长延时为 4.36ns，故乘法器能达到的最大频率约为 229MHz（忽略保持时间和建立时间）。由于华莱士树乘法器的输出是由五层加法器级联形成，且单个 16 位加法器模块与常规 16 位加法器模块不同，不是一位加法器的级联而是并联输出 16 位的和与进位，故延时时间较基本乘法器大大减小。

面积

```
Number of ports:                          592
Number of nets:                          1083
Number of cells:                          536
Number of combinational cells:            528
Number of sequential cells:                 1
Number of macros/black boxes:               0
Number of buf/inv:                        128
Number of references:                       9

Combinational area:               5277.007904
Buf/Inv area:                      580.992004
Noncombinational area:               0.000000
Macro/Black Box area:                0.000000
Net Interconnect area:      undefined  (Wire load has zero net
area)

Total cell area:                  5277.007904
```

8 位华莱士树乘法器需要总的逻辑门面积约为 5277 单位，与基本乘法器相当，这是因为华莱士树乘法器仍使用了 7 个 16 位加法器，华莱士树乘法器优化的是加法器之间的关系而不是个数。

五、静态时序分析

（1）8 位基本乘法器

由于 8 位基本乘法器是组合电路，故没有静态时序分析。

（2）8 位华莱士树乘法器

由于 8 位华莱士树乘法器是组合电路，故没有静态时序分析。

六、最大延时路径

（1）8 位基本乘法器

```
Startpoint: a[0] (clock source 'virtual_clk')
Endpoint: sum[15] (output port clocked by virtual_clk)
Path Group: virtual_clk
Path Type: max
```

最大延时路径为 a[0]至 sum[15]，延时为 13.5ns。因为基本乘法器本质是全加器的级联，所以前一级计算完毕之后下一级才能开始计算，延时较大。

基本乘法器的优化方向是改为流水线，避免加法器的闲置状态，或者改为华莱士树乘法器，优化加法逻辑。同时，也可以优化基本乘法器的 16 位加法器，这也可以大大提升基本乘法器的速度。

（2）8 位华莱士树乘法器

```
Startpoint: a[3] (clock source 'virtual_clk')
Endpoint: multi[8] (output port clocked by virtual_clk)
Path Group: virtual_clk
Path Type: max
```

最大延时路径为 a[3]至 multi[8]，延时为 4.36ns。因为 8 位华莱士树乘法器仍是五层全加器的级联，所以前一层计算完毕之后下一层才能开始计算。

因为8位华莱士树乘法器仍是有多层加法器,所以优化方向是改为流水线，避免加法器的闲置状态。

七、实验总结

通过本次数字集成电路设计实验二，我对 8 位无符号整数乘法器的设计有了深入的理解和实践。在实验过程中，我成功设计并实现了两种乘法器：基本乘法器和华莱士树乘法器。

基本乘法器的设计相对简单，它通过按位与运算生成部分积，并逐个相加得到最终结果。虽然这种方法直观易懂，但在实际应用中存在一些局限性。从综合结果来看，基本乘法器能达到的最大频率约为 74MHz，其延时较长，主要原因是输出由 8 个 16 位加法器级联形成。此外，基本乘法器消耗的面积较大，需要总

的逻辑门面积约为 5467 单位。这表明，尽管基本乘法器在小规模或低性能要求的应用中可能足够使用，但在对速度和面积有严格要求的场景下，其性能可能不尽如人意。

华莱士树乘法器则采用了更为先进的设计理念。它通过分组并行相加的方式，将多个部分积压缩成两个数，从而减少了加法运算的级数，提高了运算速度。实验结果表明，华莱士树乘法器能达到的最大频率约为 229MHz，远高于基本乘法器。其最长延时为 4.36ns，这一显著的性能提升得益于其独特的加法器级联结构。在面积方面，华莱士树乘法器需要的逻辑门面积约为 5277 单位，与基本乘法器相当。这说明华莱士树乘法器在不增加过多面积开销的情况下，实现了性能的大幅提升。

在实验过程中，我还进行了仿真验证，确保两种乘法器在不同输入条件下的正确性。通过对比两种乘法器的性能，我深刻认识到优化算法和电路结构对于提升集成电路性能的重要性。基本乘法器的优化方向可以是改为流水线结构，避免加法器的闲置状态，或者采用华莱士树乘法器的加法逻辑。而对于华莱士树乘法器，尽管其性能已经较为出色，但仍有进一步优化的空间，例如通过流水线技术进一步提高速度。

总的来说，本次实验不仅让我掌握了两种乘法器的设计方法，还让我学会了如何通过综合和仿真分析来评估电路的性能。这些知识和技能将为我未来在数字集成电路设计领域的学习和研究打下坚实的基础。