Lab 7 report

PB2111711 陈昕琪

实验目的与内容

> 1. 基于加法运算的逻辑电路以及有限状态机的内容，实现一个较为简单的乘法模块。
> 2. 以组合乘法器为例，对电路进行时间性能和资源占用的分析。为更好地理解硬件电路的生成与设计打下基础。
> 3. 进一步掌握时序逻辑电路设计、有限状态机设计和基本算术电路的设计方法。

1. 必做内容：乘法器基础版

> **要求：**
>
> 根据实验文档中给出的移位乘法器设计方法，设计一个基础的参数化移位乘法器。

> **逻辑实现：**
>
> 本题主要是基于有限状态机，并根据实验文档中对乘法器的介绍，实现的乘法器。
> 对于每个状态，需要在对应的时序模块中处理。

- `IDLE`状态：只需判断`start`信号即可。
- `INIT`状态：初始化各寄存器的值，即将乘数和被乘数存进相应的寄存器。
- `CALC`状态：进行计算，判断乘数寄存器的最低位后进行相应操作，并将被乘数寄存器左移，将乘数寄存器右移一位。同时加入一个`n`用于判断次数，当判断到循环达到`N`次的时候则跳转到`DONE`。
- `DONE`状态：表示计算结束，输出结果。

> 最终根据状态判断是否赋值给`res`。

代码如下：

```
module MUL #(
    parameter                               WIDTH = 32
) (
    input                   [ 0 : 0]            clk,
    input                   [ 0 : 0]            rst,
    input                   [ 0 : 0]            start,
    input                   [WIDTH-1 : 0]       a,
    input                   [WIDTH-1 : 0]       b,
    output      reg         [2*WIDTH-1:0]       res,
    output      reg         [ 0 : 0]            finish
);
reg [2*WIDTH-1 : 0]     multiplicand;       // 被乘数寄存器
reg [  WIDTH-1 : 0]     multiplier;         // 乘数寄存器
reg [2*WIDTH-1 : 0]     product;            // 乘积寄存器
localparam IDLE = 2'b00;            // 空闲状态。这个周期寄存器保持原值不变。当 start 为 1 时跳转到 INIT。
localparam INIT = 2'b01;            // 初始化。下个周期跳转到 CALC
```

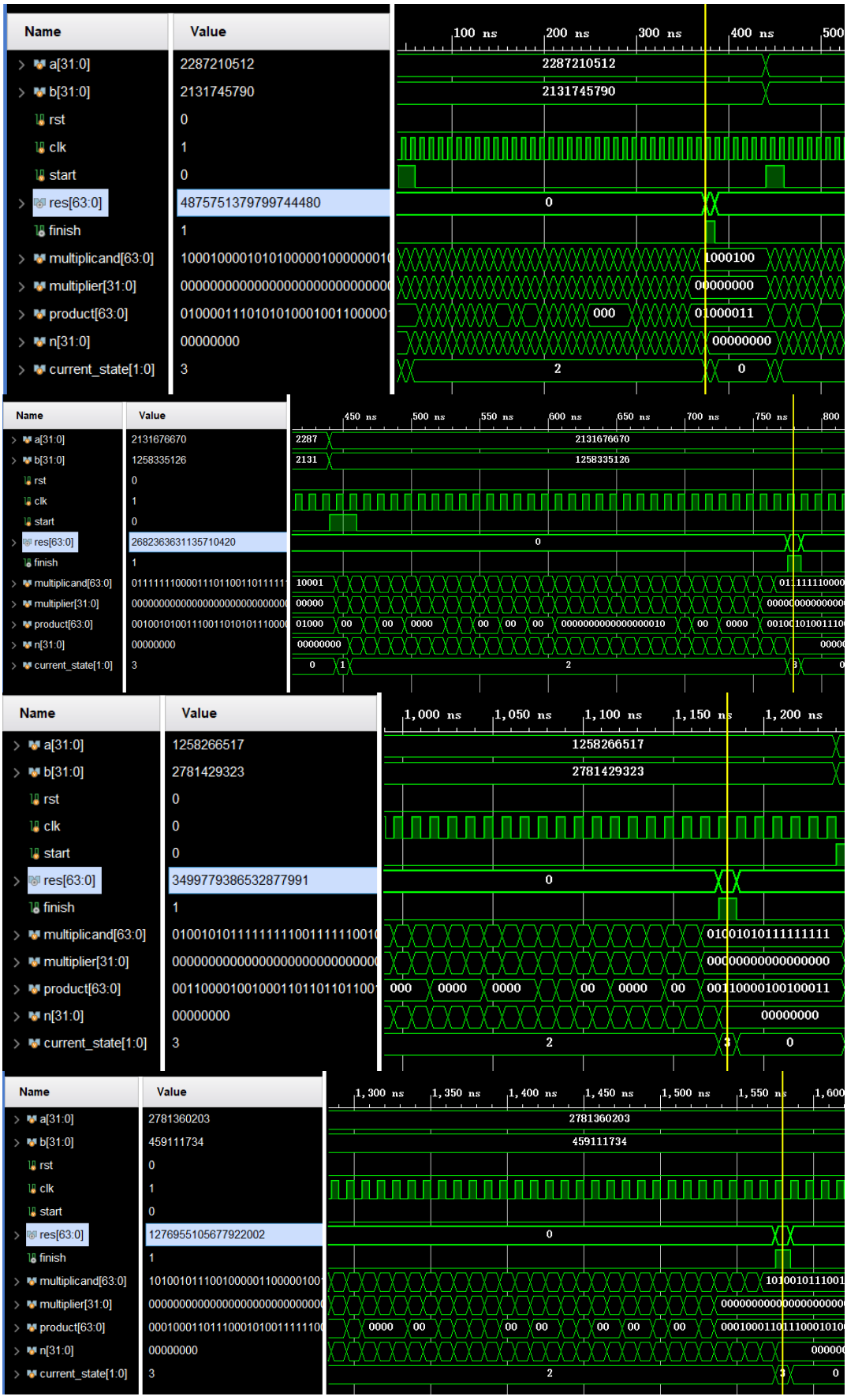
```
localparam CALC = 2'b10;            // 计算中。计算完成时跳转到 DONE
localparam DONE = 2'b11;            // 计算完成。下个周期跳转到 IDLE
reg [1:0] current_state, next_state;
reg [31:0] n;
always @(posedge clk) begin
    if(rst) begin
        current_state <= IDLE;
        multiplicand <= 0;
        multiplier <= 0;
        product <= 0;
        res <= 0;
        n <=0;
    end else  begin
        current_state <= next_state;
    end
    if(next_state == CALC) begin
        if(multiplier[0] == 1) begin
            product <= multiplicand + product;
        end else product <= product;
        multiplicand <= {multiplicand[2*WIDTH-2:0],1'b0};
        multiplier <= {1'b0,multiplier[WIDTH-1:1]};
    end
    if(next_state == CALC && n != WIDTH) begin
        n <= n + 1;
    end else if(n == WIDTH) begin
        n <= 0;
    end
end
// 实现 FSM
always @(*) begin
    case(current_state)
        IDLE: begin
            if (start) next_state = INIT;
            else next_state = IDLE;
        end
        INIT: begin
            multiplicand = a;
            multiplier = b;
            product = 0;
            n = 0;
            next_state = CALC;
        end
        CALC: begin
            if (n == WIDTH) begin
                next_state = DONE;
            end else next_state = CALC;
        end
        DONE: begin
            next_state = IDLE;
        end
    endcase
end
always @(*)begin
    finish = (current_state == DONE);
    res = finish ? product : 0;
```

```
end
endmodule
```

仿真结果与分析

> 直接使用实验文档中的仿真文件

> 仿真得出的波形图如下：



> 通过计算器验算得知程序正确。

2. 必做内容：乘法器优化版

> **逻辑实现：**
>
> 根据实验文档中的优化方法改进移位乘法器。要求通过仿真验证其正确性。
> 同时，需要在FPGAOL平台上用数码管验证其正确性。

代码如下：(以下是`32`位乘法器，四位乘法器只需更改`WIDETH`的值即可)

```
module MUL #(
    parameter                               WIDTH = 32
) (
    input                   [ 0 : 0]            clk,
    input                   [ 0 : 0]            rst,
    input                   [ 0 : 0]            start,
    input                   [WIDTH-1 : 0]       a,
    input                   [WIDTH-1 : 0]       b,
    output      reg         [2*WIDTH-1:0]       res,
    output      reg         [ 0 : 0]            finish
);
reg [  WIDTH-1 : 0]     multiplicand;
reg [2*WIDTH   : 0]     product;
localparam IDLE = 2'b00;
localparam INIT = 2'b01;
localparam CALC = 2'b10;
localparam DONE = 2'b11;
reg [1:0] current_state, next_state;
reg [31:0] n;
reg [2*WIDTH : 0]     temp;
always @(*) begin
    temp = {1'b0,multiplicand[WIDTH-1:0],{WIDTH{1'b0}}};
end
always @(posedge clk) begin
    if (rst) begin
        current_state <= IDLE;
        multiplicand <= 0;
        product <= 0;
        n <= 1'b0;
    end else  begin
        current_state <= next_state;
        if(next_state == CALC && n != WIDTH) begin
            n <= n + 1;
        end else if(n == WIDTH) begin
            n <= 0;
        end
        if (next_state == CALC) begin
            if (product[0] == 1) begin
                product <= {1'b0,{temp[2*WIDTH:1]}} + {1'b0,product[2*WIDTH:1]};
            end else product <= {1'b0,product[2*WIDTH:1]};
        end else if(next_state == INIT) begin
            multiplicand <= a;
```

```
            product <= b;
            n <= 0;
        end
    end
end
// 实现 FSM
always @(*) begin
    case(current_state)
        IDLE: begin
            if (start) next_state = INIT;
            else next_state = IDLE;
        end
        INIT: begin
            next_state = CALC;
        end
        CALC: begin
            if (n == WIDTH) begin
                next_state = DONE;
            end else begin
                next_state = CALC;
            end
        end
        DONE: begin
            next_state = IDLE;
        end
    endcase
end
always @(*)begin
    finish = (current_state == DONE);
    res = finish ? product : 0;
end
endmodule
```

> 而想要在FPGAOL平台上运行，则需要添加`Top`模块并调用之前写过的数码管模块。以下是`Top`模块。

```
module Top(
    input                   clk,
    input                   btn,
    input  [7:0]            sw,
    output [2:0]            seg_an,
    output [3:0]            seg_data
);
wire [7:0] res;
reg [7:0] res_temp;
initial begin
    res_temp = 0;
end
    MUL_4 mul(
    .clk(clk),
    .rst(sw[7]),
    .a({{1'b0},{sw[6:4]}}),
    .b(sw[3:0]),
    .start(btn),
    .res(res),
```

```
    .finish()
    );
always @(posedge clk) begin
    if(res != 0) begin
        res_temp <= res;
    end else res_temp <= res_temp;
end
    Segment segment(
        .clk(clk),
        .rst(btn),
        .output_data({{24'b0},{res_temp[7:0]}}),
        .seg_data(seg_data),
        .seg_an(seg_an)
    );
endmodule
```

> 其中，由于`res`是瞬时输出的，所以用了一个`res_temp`用来储存`res`的值用于显示(但是写的时候由于没有用时序逻辑电路出现了锁存器，在助教的指导下更正过来了)
> 这里的`MUL_4`模块和32位优化乘法器是一样的，只是更改了`WIDTH`的值。`Segment`模块运用了之前写过的模块实现数码管显示。

仿真结果与分析

> 根据所写的代码，编写相应的仿真文件验证其正确性。
> 注：这里更改了`Segement`模块中计时器部分的参数，便于观察仿真结果。

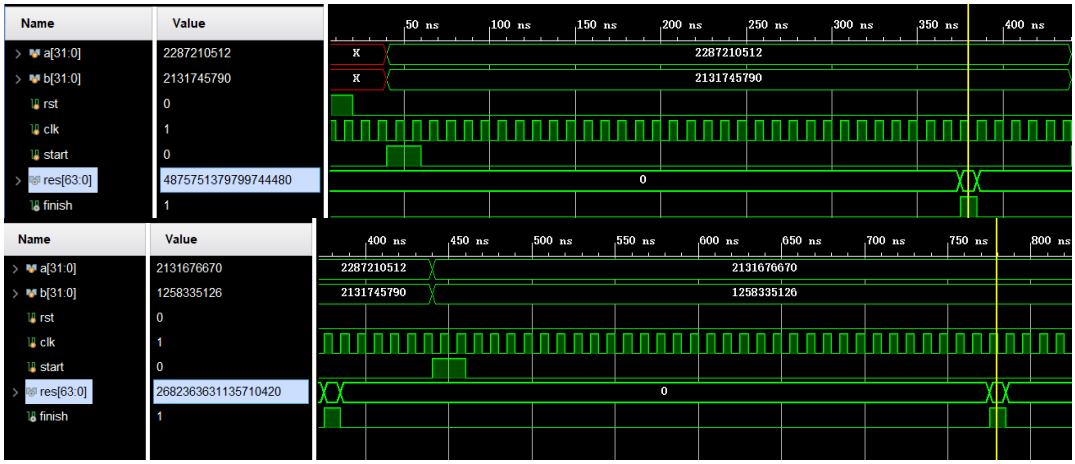
仿真文件代码如下：

```
module Top_tb();
reg                 clk;
reg                 btn;
reg  [7:0]          sw;
wire [2:0]          seg_an;
wire [3:0]          seg_data;
initial begin
    clk = 0;
    btn = 0;
    sw = 8'b1_001_0001;
    #10;
    sw = 8'b0_001_0001;
    btn = 1;
    #10 btn = 0;
    #2000;
    sw = 8'b1_011_0001;
    #10;
    sw = 8'b0_011_0001;
    btn = 1;
    #10 btn = 0;
    #2000;
end
always #5 clk = ~clk;
Top top(
    .clk(clk),
    .btn(btn),
    .sw(sw),
```

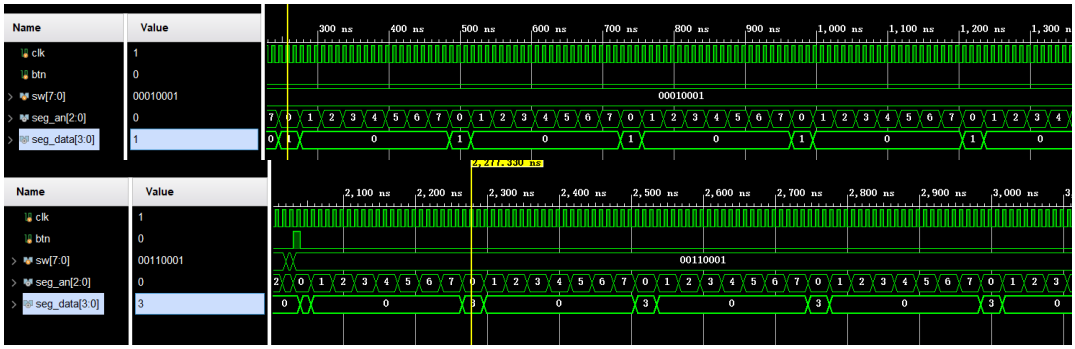
```
    .seg_an(seg_an),
    .seg_data(seg_data)
);
endmodule
```

> 仿真得出的波形图如下：

32位优化版加法器：



4位优化版乘法器数码管显示版



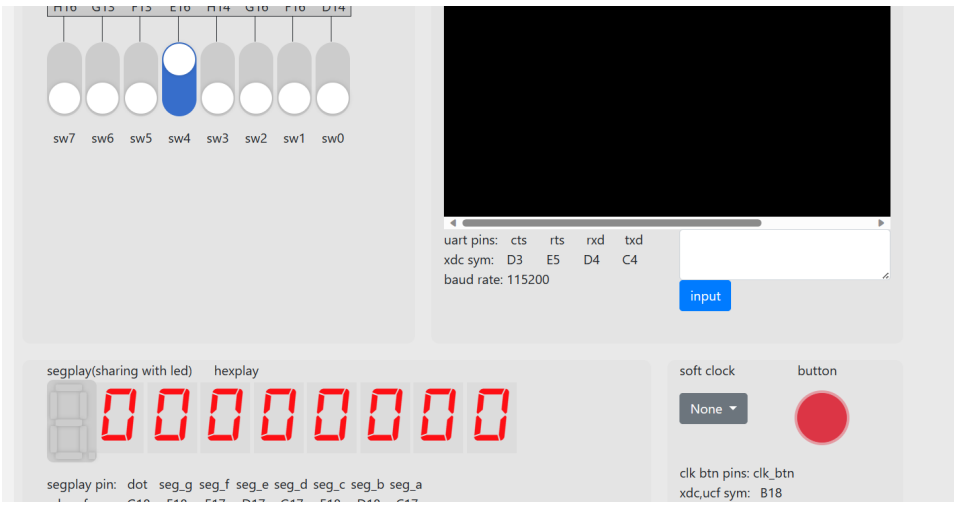
> 对比分析得知程序正确。

测试

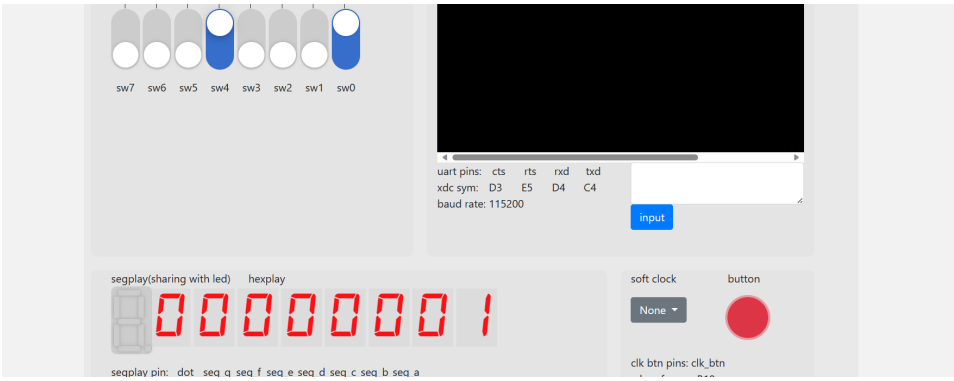
上板结果：

> 烧写完比特流文件后进行上板查看，测试如下：

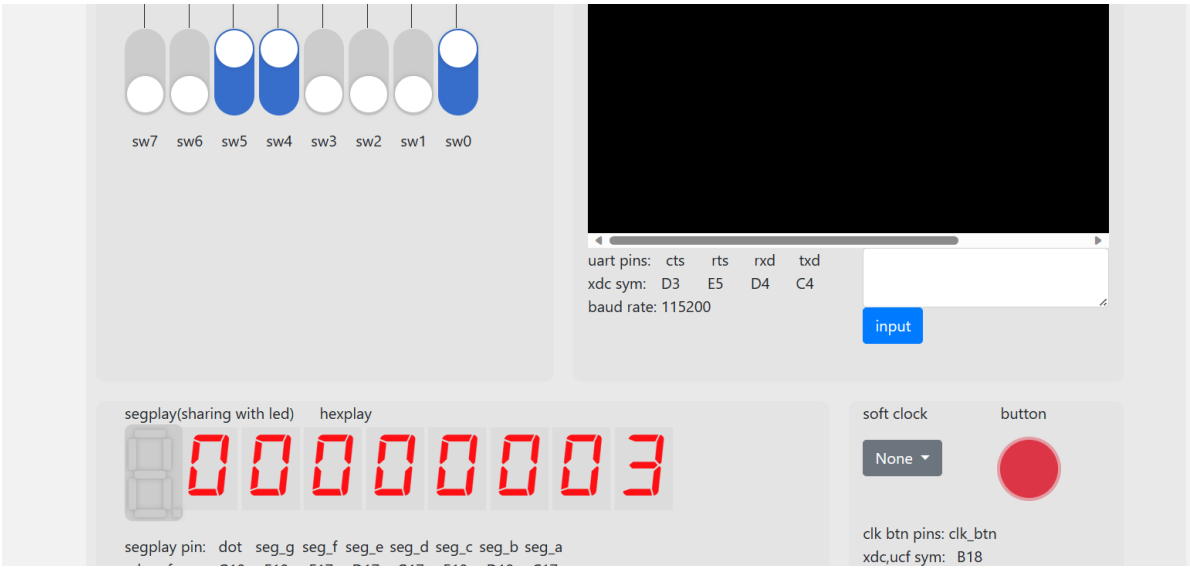
1. 测试1×0



2. 测试1×1

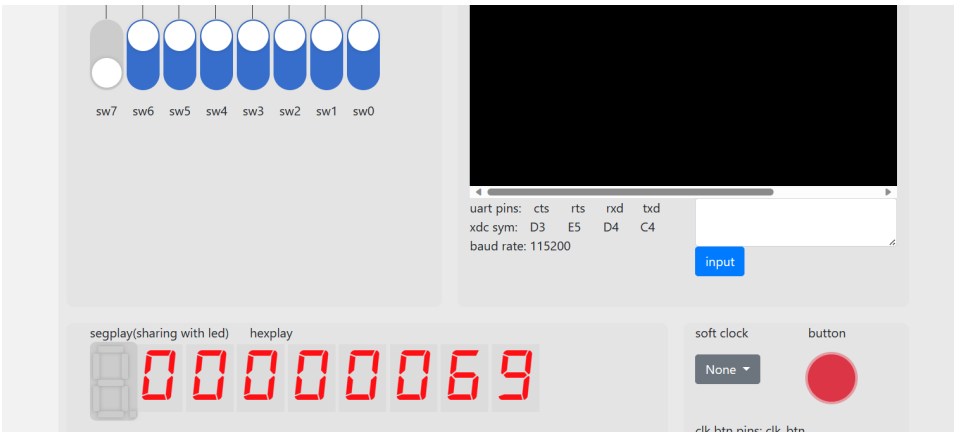


3. 测试3×1



4. 测试7×15

> 注：这里是直接将计算得出的数字传入数码管了，所以显示的是结果的十六进制。



> 对比分析程序正确

3. 必做内容：性能比较

> **题目要求：**
>
> 将自己写的优化版乘法器与 Verilog 中自带的乘法进行性能比较，大致估计两个乘法器的运行时间。

1. 自己写的优化版乘法器

```
5
6  ## Clock signal
7  set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { clk }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
8  create_clock -add -name sys_clk_pin -period 3.60 -waveform {0 1.80} [get_ports {clk}];
9
```

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.025 ns	Worst Hold Slack (WHS):	0.281 ns	Worst Pulse Width Slack (WPWS):	1.300 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	90	Total Number of Endpoints:	90	Total Number of Endpoints:	47

All user specified timing constraints are met.

测出的$T_{min} = 3.60ns$,因此运行时间
$t = WIDTH/f_{max2} = 32 \times 3.60 \times 10^-9 = 1.152 \times 10^-7s$。

2. `Verilog`自带乘法器

```
## Clock signal
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { clk }]; #IO_L12P_T1_MRCC_35 Sch=clk
create_clock -add -name sys_clk_pin -period 3.00 -waveform {0 1.50} [get_ports {clk}];
```

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.006 ns	Worst Hold Slack (WHS):	0.380 ns	Worst Pulse Width Slack (WPWS):	0.845 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	8	Total Number of Endpoints:	8	Total Number of Endpoints:	18

All user specified timing constraints are met.

测出的$T_{min} = 3.00ns$,因此运行时间$t = 1/f_{max1} = 3.00 \times 10^-9s$。

4. 必做内容：有符号乘法

> **要求：**
>
> 基于自己写的优化乘法器，设计有符号乘法运算，即输入输出都视为有符号数。通过仿真测试有符号乘法的正确性。
>
> **逻辑实现：**
>
> 有符号的乘法器首先需要判断乘数被乘数符号，若有负数则需取反加一再放进寄存器中。
> 这时放入寄存器中的值都是正数，所以只需要正常进行乘法操作。
> 最后赋值给res的时候若两个数异号则取反加一赋值给res，否则直接赋值给res。

代码如下：(这里仅展示核心代码)

1. `INIT`状态赋值

```
INIT: begin
            if(a[WIDTH-1] == 1) begin
                multiplicand = ~a + 1;
            end else multiplicand = a;
            if(b[WIDTH-1] == 1) begin
                product[WIDTH-1:0] = ~b + 1;
                product[2*WIDTH:WIDTH] = 0;
            end else product = b;
            temp = {1'b0,multiplicand[WIDTH-1:0],32'b0};
            finish = 0;
            n = 0;
            next_state = CALC;
        end
```

2. `res` 赋值:

```
always @(*)begin
    finish = (current_state == DONE);
    if(finish == 1) begin
        if(a[WIDTH-1] == b[WIDTH-1]) begin
            res = product;
        end else res = {1'b1,(~product[2*WIDTH-2:0]+1)};
    end else res = 0;
end
```

仿真结果与分析

> 为了测试符号判断的正确性，自己写了几个数据测试，测试了两正，一正一负，两负三种情况。

仿真文件代码如下:

```
module MUL_tb #(
    parameter WIDTH = 32
) ();
reg  [WIDTH-1:0]    a, b;
reg                 rst, clk, start;
wire [2*WIDTH-1:0]  res;
wire                finish;
initial begin
    clk = 0;
    forever begin
        #5 clk = ~clk;
    end
end

initial begin
    rst = 1;
    start = 0;
    #20;
    rst = 0;
    #20;
    a = 32'h12345678;
    b = 32'h23456789;
    start = 1;
```

```
    #20 start = 0;
    #380;
    a = 32'hf2345678;
    b = 32'h23456789;
    start = 1;
    #20 start = 0;
    #380;
    a = 32'hf2345678;
    b = 32'hf3456789;
    start = 1;
    #20 start = 0;
    #380;
    $finish;
end

MUL mul(
    .clk        (clk),
    .rst        (rst),
    .start      (start),
    .a          (a),
    .b          (b),
    .res        (res),
    .finish     (finish)
);
endmodule
```

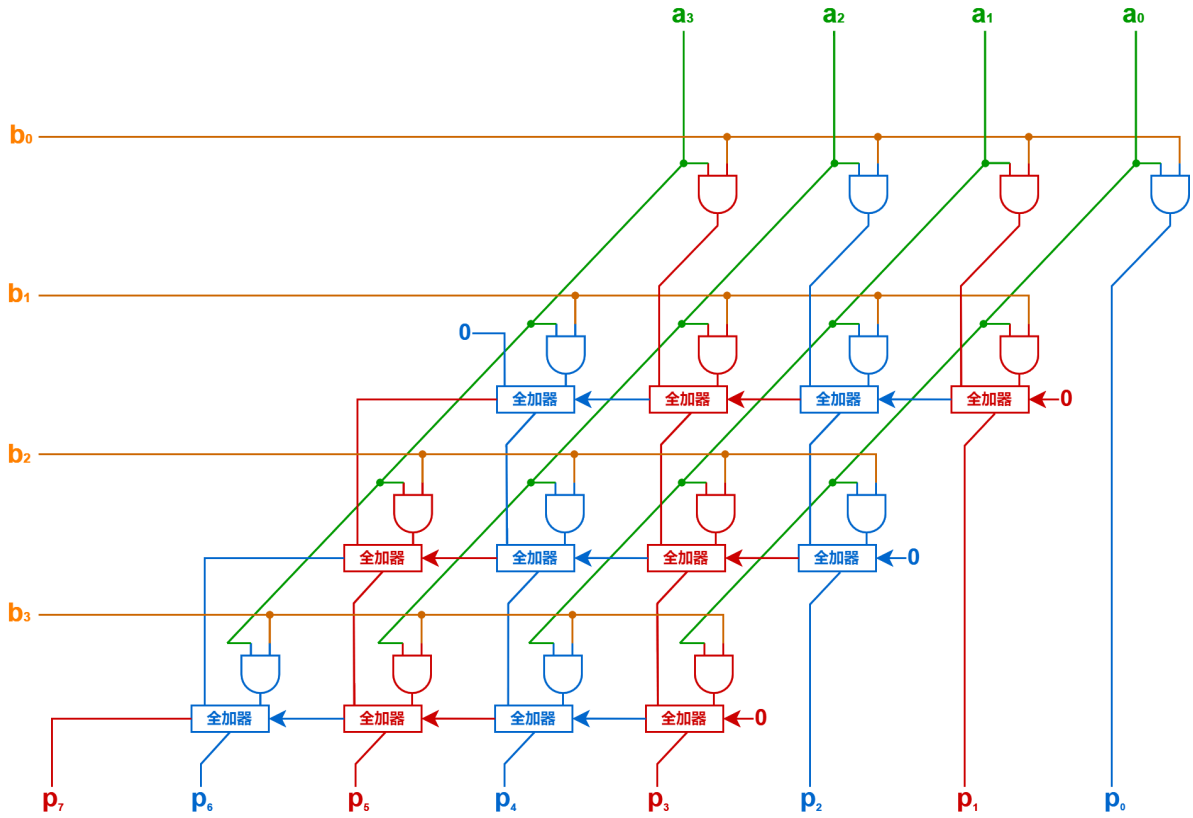
> 仿真得出的波形图如下：

> 计算器验算得知程序正确。

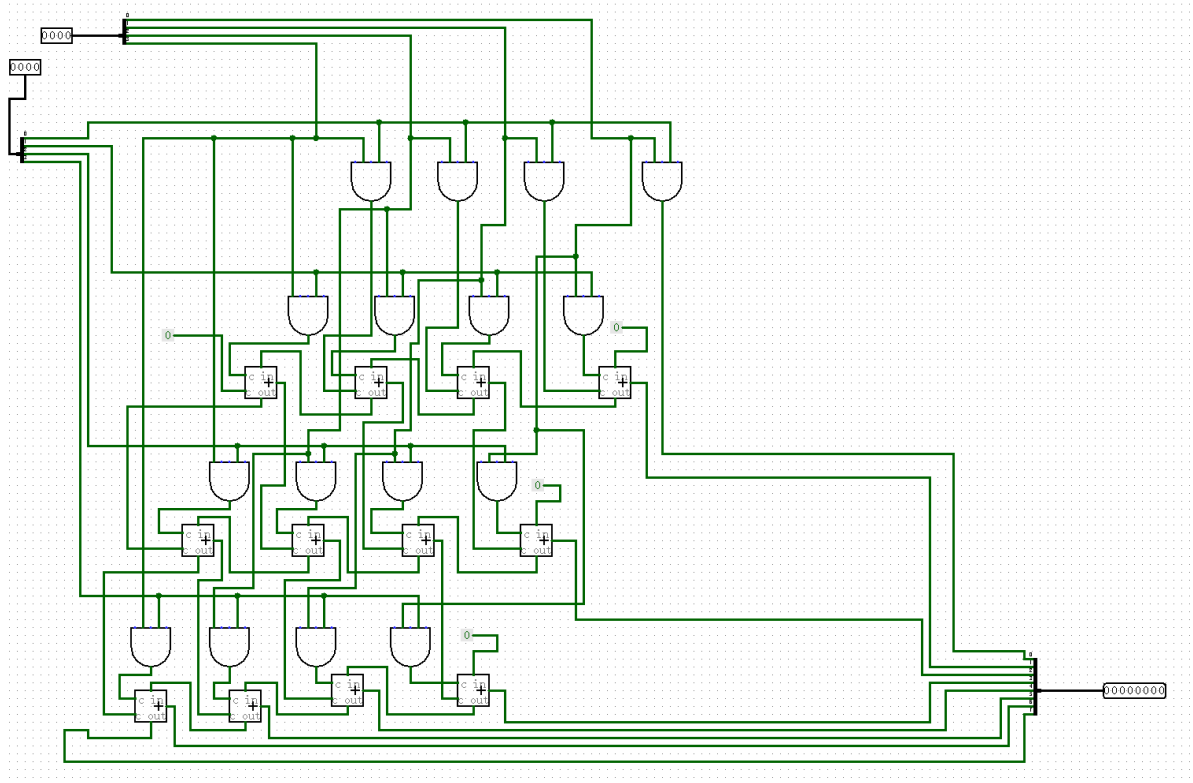
5. 选择性必做内容：Logisim 搭建乘法器

> **要求：参考文档中四位乘法器的实现，在 Logisim 中搭建一个四位乘法器**
>
> **逻辑实现：**
>
> 乘法计算过程共分为三步：

- 第一步计算移位，即将乘数的副本左移$0-n-1$位；
- 第二步将移位结果AND被乘数对应位置的数值；
- 第三步计算每一位的最终和。
  并根据四位无符号二进制数乘法电路的结构可以用logisim搭建乘法器



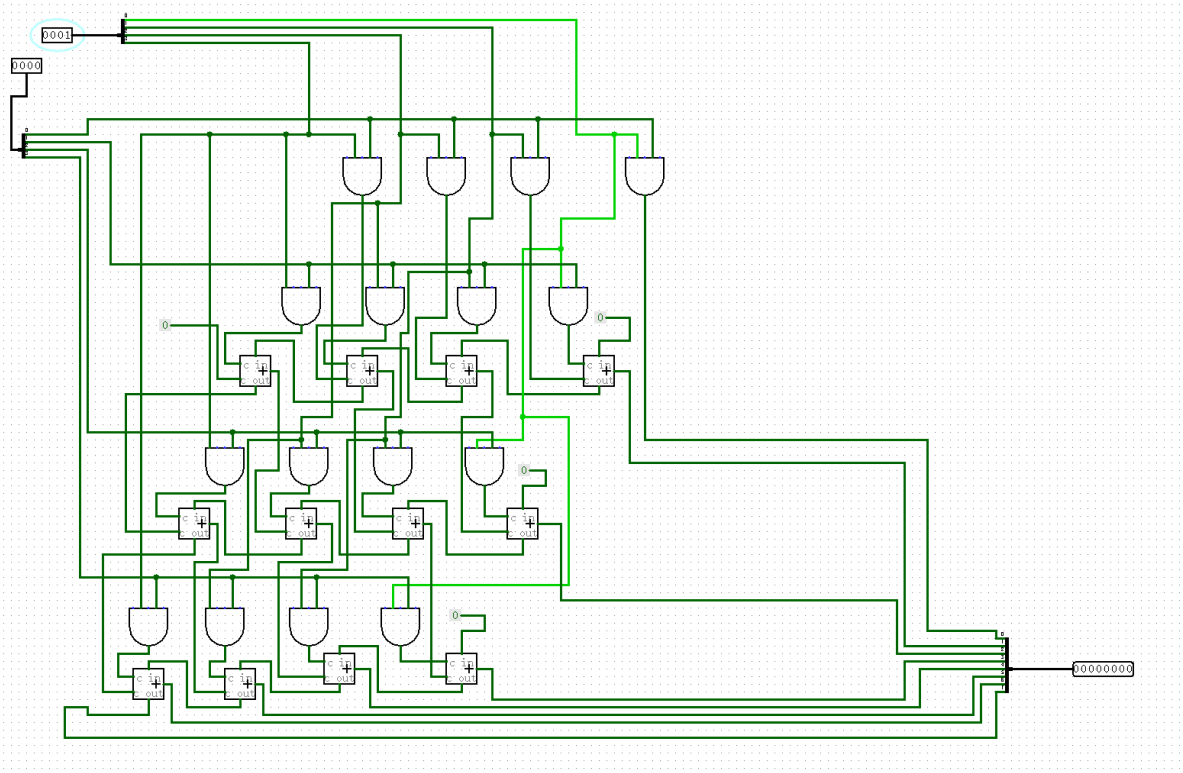
logisim搭建的乘法器：



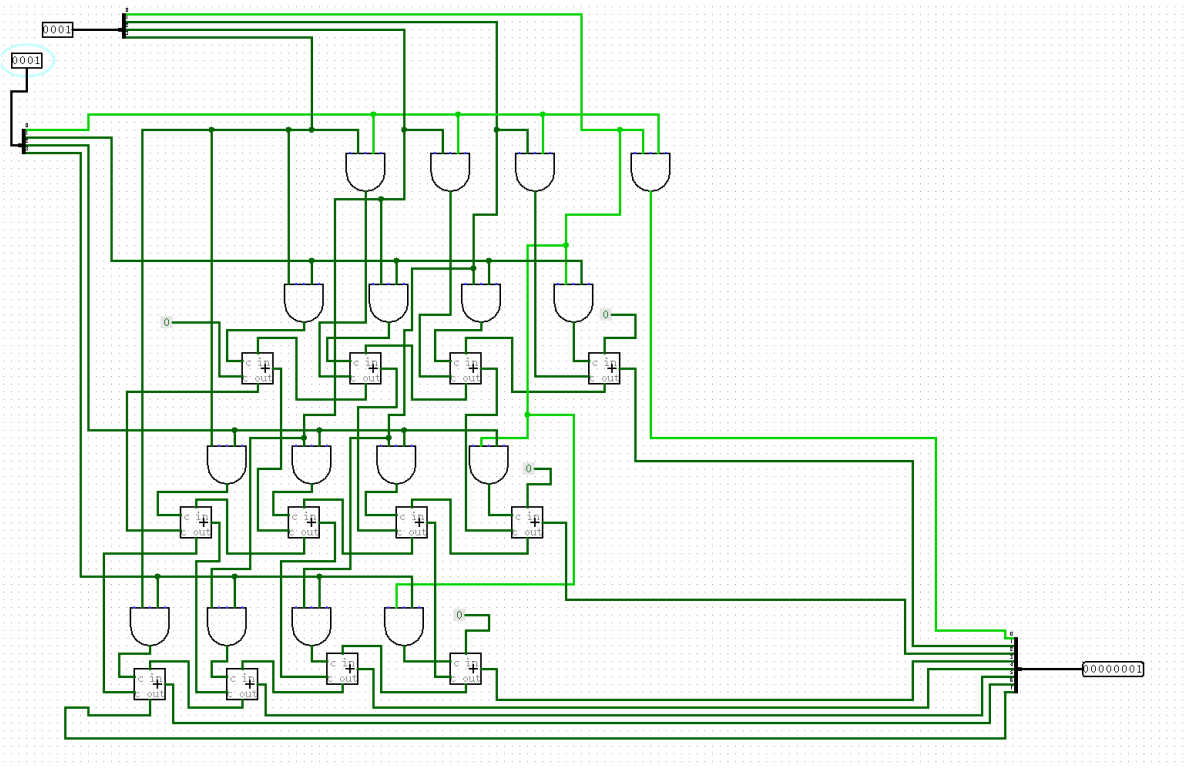
测试结果

测试结果如下：

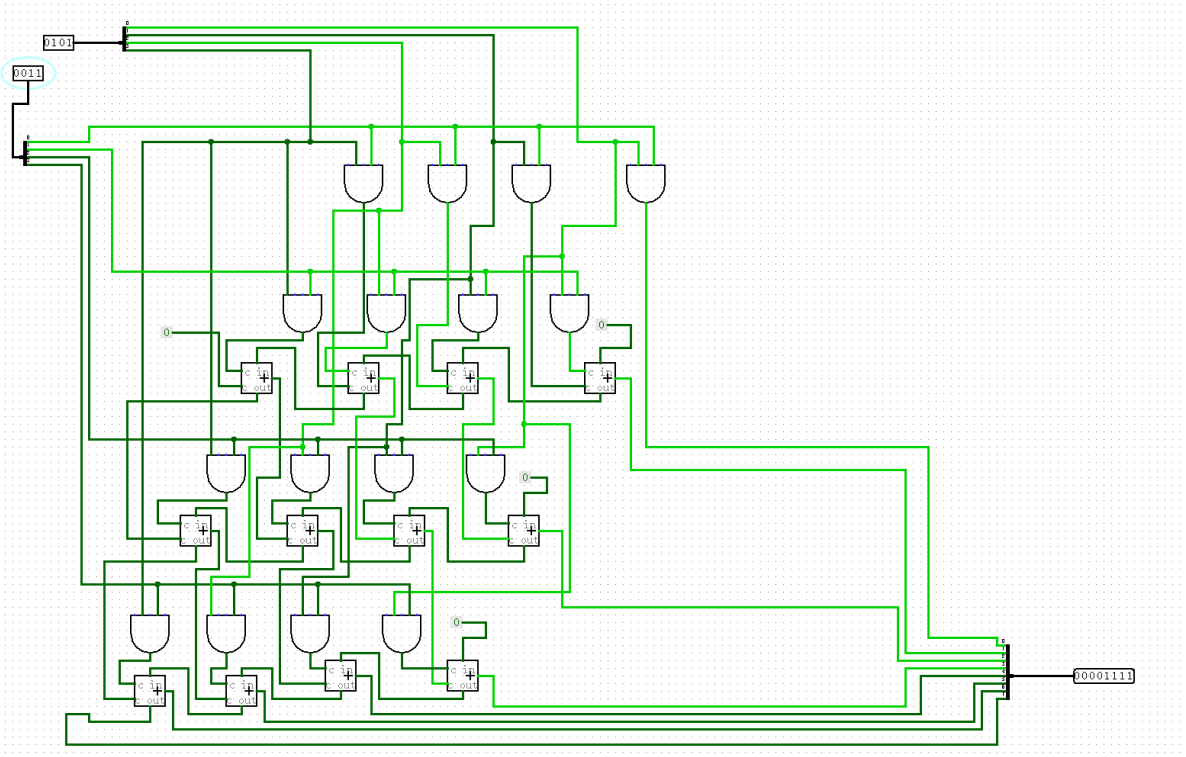
1. 1×0



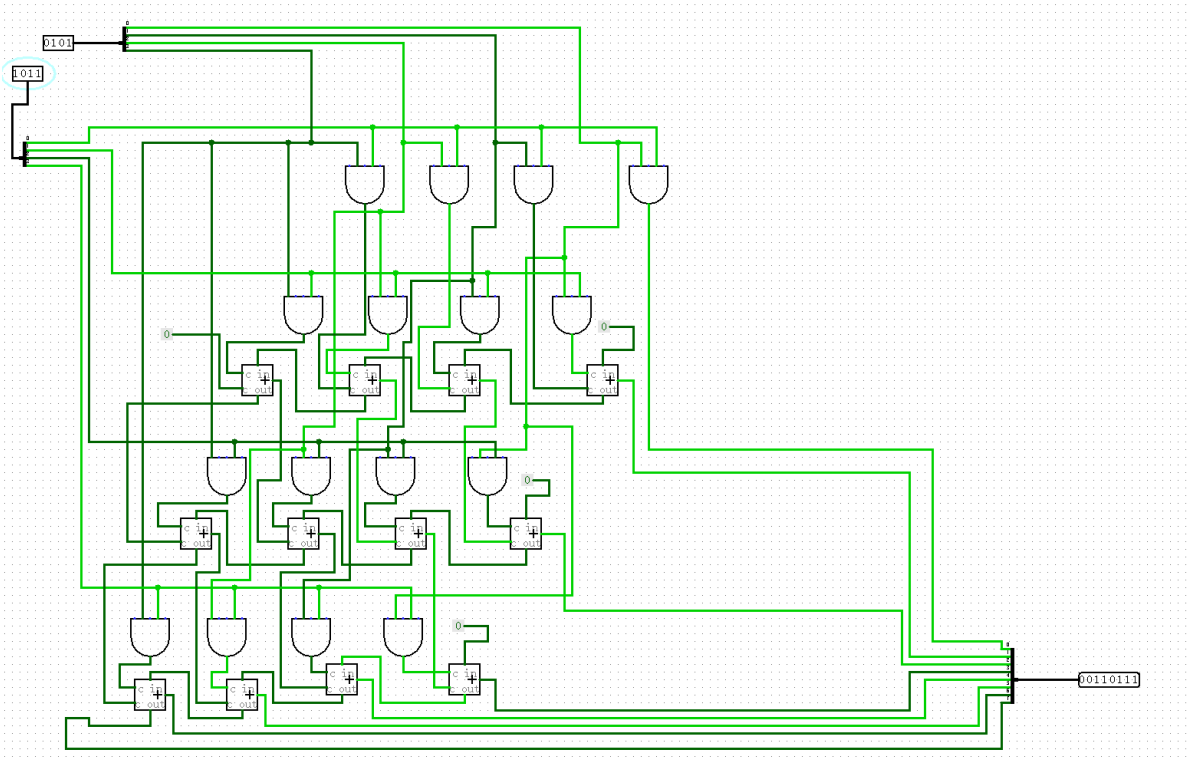
2. 1×1



3. 5×3



4. 5×11



> 观察比对得知程序正确。

总结与反思

> 1. 本次实验，对于 Verilog 语言有了更深入的了解，同时学会利用状态机数码管等曾经学过的知识编写电路。
> 2. 本次实验最大的收获是学会了看`warning`，这里要感谢助教的push，之前都是看波形图查看程序的错误，但是往往有很多问题仿真不能看出来但是会无法生成比特流文件。这也是之前写实验的时候没有关注到的点，很幸运能在第七次实验学会看warning，也加深了组合环、多驱动等概念的理解（应该好好看看实验文档了qwq）。

3. 对于实验部分，学会了乘法器的组成及原理，并学会分析电路性能，为今后的实验打下基础。同时，在程序出现问题或者仿真综合出现问题时，要及时回去细致的查看代码，这次实验因为几个问题卡了好久。不过为今后的实验积累了一点经验（毕竟计科还要手搓cpu呜呜）。
4. 最后感谢助教的耐心讲解与解答，一直在帮同学们答疑解惑，助教们辛苦哩!