# Lab3-1 Report

## 1.1使用enum语法FSM设计

typedef enum logic [1:0] {S0,S1,S2,S3} fsm\_state;

     fsm\_state state1;

当

## 1.2 综合实现有限状态机

always @(posedge clk or negedge rstn) begin

//使用异步复位

        if(~rstn) begin

            state1 <= S0;

        end

        else begin

            case(state1)//对State1的状态进行选择

                S0: begin

                    if(a) begin

                        state1 <= S1;

                    end

                    else begin

                        state1 <= S0;

                    end

                end

                S1: begin

                    if(a) begin

                        state1 <= S2;

                    end

                    else if(a == 0 && b == 1)begin

                        state1 <= S0;

                    end

                    else begin

                        state1 <= S1;

                    end

                end

                S2: begin

                    if(a) begin

                        state1 <= S3;

                    end

                    else if(b)begin

                        state1 <= S0;

                    end

                    else begin

                        state1 <= S2;

                    end

                end

                S3: begin

                    state1 <= S3;

                end

                default: state1 <= S0;

            endcase

        end

     end

     assign state = state1;

## 2. 思考比较enum + case的编程范式和数组查表的编程范式之间的优劣

**枚举 + case语句**

**优点**:

1. 使用枚举和case语句可以提高代码的可读性，使得代码易于理解。
2. 语句允许在每个分支中执行复杂的逻辑，并且可以轻松地添加或删除case，这使得代码更加灵活。

**缺点**:

1. 当有大量的case分支时，可能会影响性能，因为程序需要逐个检查每个case直到找到匹配项。
2. 随着case语句的增多，代码可能变得难以维护，尤其是当多个case分支执行相似操作时。

**数组查表**

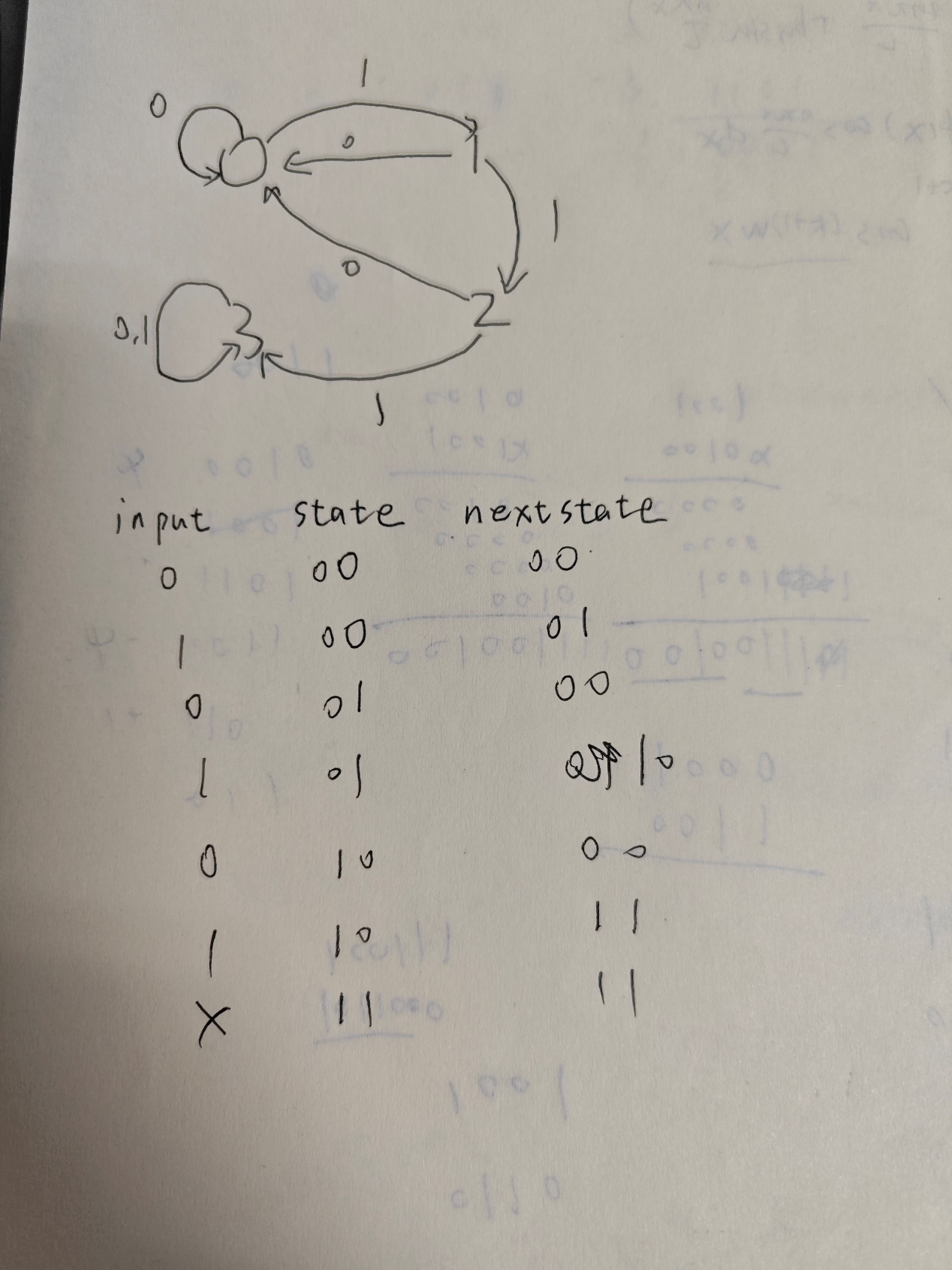
**优点**:

1. 对于大规模数据，查表通常比多个case语句执行得更快，因为可以直接通过索引访问数据，时间复杂度为O(1)。
2. 使用查表可以使代码更加简洁，特别是当处理大量固定的映射关系时。

**缺点**:

1. 查表可能会消耗更多的内存，特别是当表中有很多未使用的条目时。
2. 如果表的内容不是很直观，那么代码的可读性和维护性可能会下降。

## 绘制一个有限状态机的状态图和状态转移表



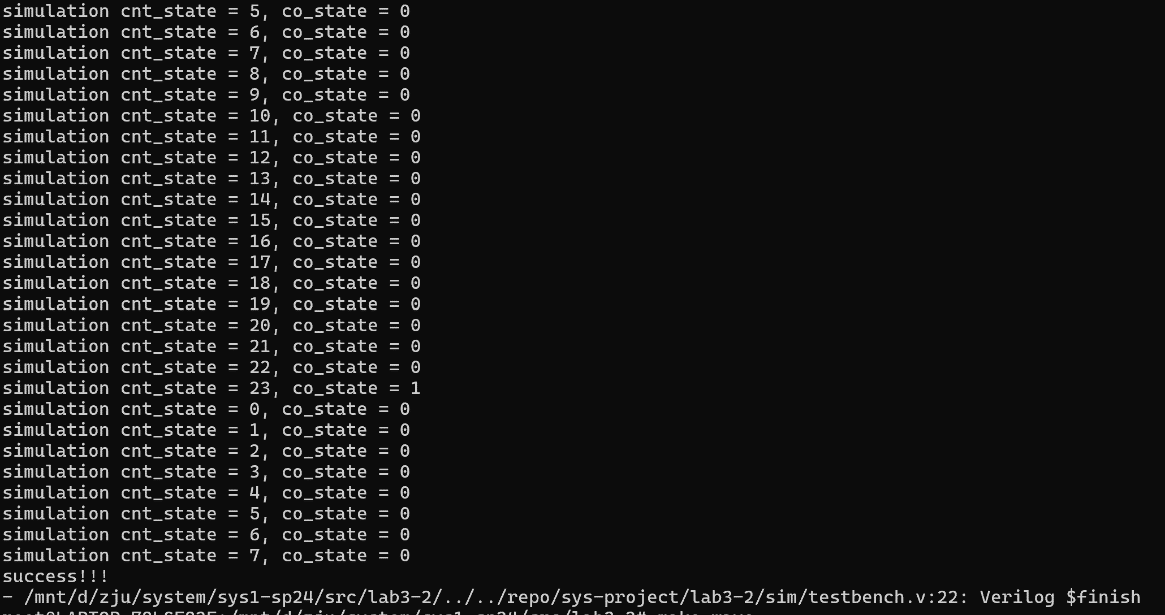
### 观察以下有限状态机电路实现存在的不足和不足的原因，如果电路不稳定可能会发生什么问题

1.存在的问题：使用了同步初始化，但是如果rstn在clk的上升沿附近发生变化，可能会导致复位行为不确定。State只在两个状态之间转换，理论上仅需一位即可表示，使用两位会增加性能消耗。

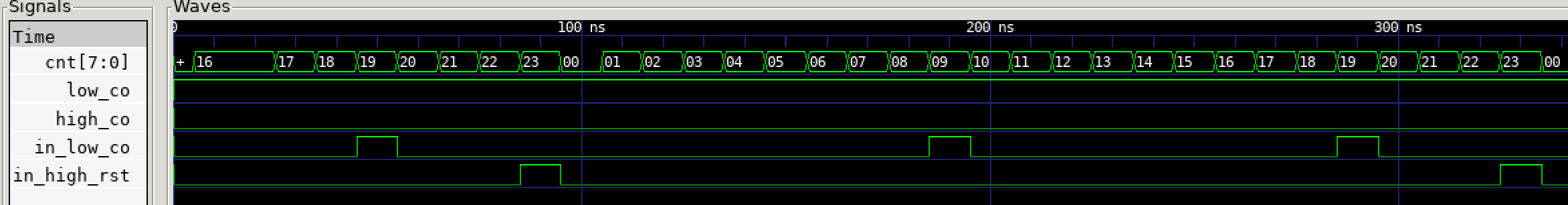
2.电路不稳定产生的问题：在复位和状态更新之间可能存在时序问题，尤其是在复位信号撤销的瞬间，如果时钟上升沿到达，可能会导致状态寄存器进入亚稳态，即其值既不是0也不是1.

# Lab3-2 Report

1. **success截图**



1. verilate仿真



1. 计数器代码解释

always @(posedge clk or negedge rstn) begin

        if(!rstn) begin

            out <= INITIAL[3:0];

        end

        else if(high\_rst) begin//如果高位复位，则将out置为0

            out <= 0;

        end

        else if(low\_co) begin//如果低位进位

            if(out == BASE-1) begin

                out <= 0;

            end

            else begin

                out <= out + 1;//此时将out+1

            end

        end

    end

    assign co = (out == BASE-1) ? 1 : 0;//表示是否需要进位

    assign cnt = out;

reg in\_high\_rst,in\_low\_co,high\_co;

    Cnt #(LOW\_BASE, LOW\_INIT) cnt\_low(//低位计数器

        .clk(clk),

        .rstn(rstn),

        .low\_co(1),

        .high\_rst(in\_high\_rst),

        .co(in\_low\_co),

        .cnt(cnt[3:0])

    );

    Cnt #(HIGH\_BASE, HIGH\_INIT) cnt\_high(//高位计数器

        .clk(clk),

        .rstn(rstn),

        .low\_co(in\_low\_co),

        .high\_rst(in\_high\_rst),

        .co(high\_co),

        .cnt(cnt[7:4])

    );

    assign in\_high\_rst = (cnt[7:4] == HIGH\_CO[3:0] && cnt[3:0] == LOW\_CO[3:0]) ? 1'b1 : 1'b0;

    //当达到23时停止，high\_rst=1,表示两个cnt全部复位

    assign co = (cnt[7:4] == HIGH\_CO[3:0] && cnt[3:0] == LOW\_CO[3:0]) ? 1'b1 : 1'b0;

    //当达到23时停止，co也需置为1

4.使用 Cnt2num 实现 1234 的 BCD 码计数器

要实现 1234的 BCD码计数器，需要两个 Cnt2num 模块，因为每个 Cnt2num 模块只能处理最多两位数的 BCD 码。将1234分为12和34，其中12用一个 Cnt2num 模块表示，34用另一个 Cnt2num 模块表示。

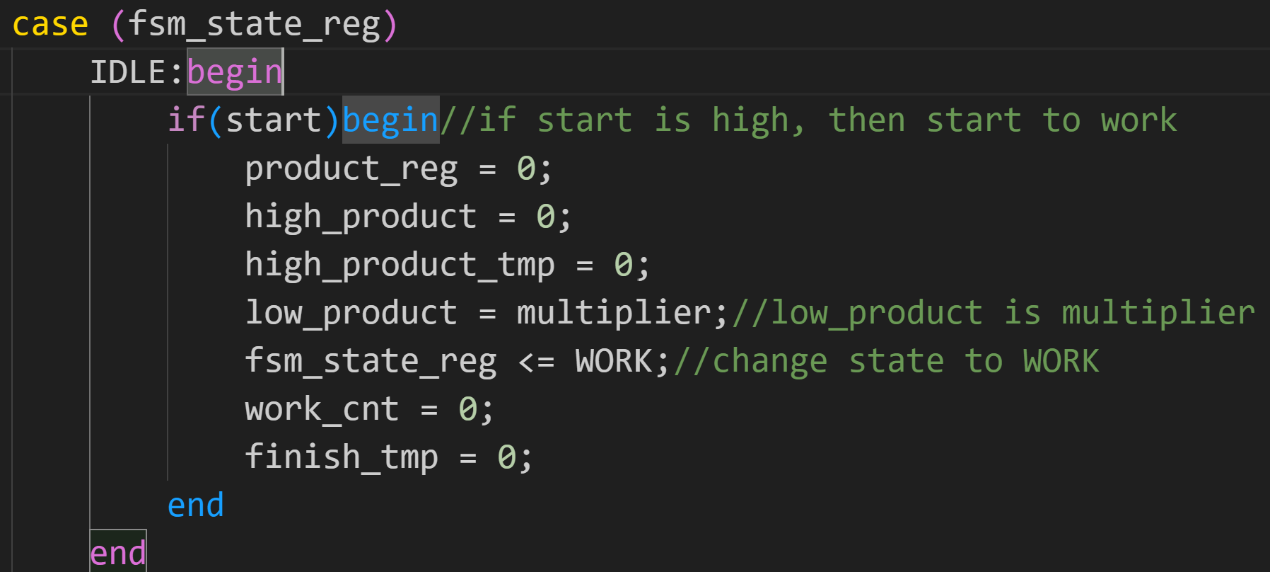
连接两个模块，当 Cnt2num\_34 达到最高计数34时，其 co 输出信号可以用来触发 Cnt2num\_12 模块的计数增加，实现12到下一个数的进位。low\_co 从 Cnt2num\_34 连接到 Cnt2num\_12 作为低位计数模块的进位输出，实现多级BCD计数器的级联。high\_rst 用于在计数器达到预设的最大计数时对计数器进行复位。

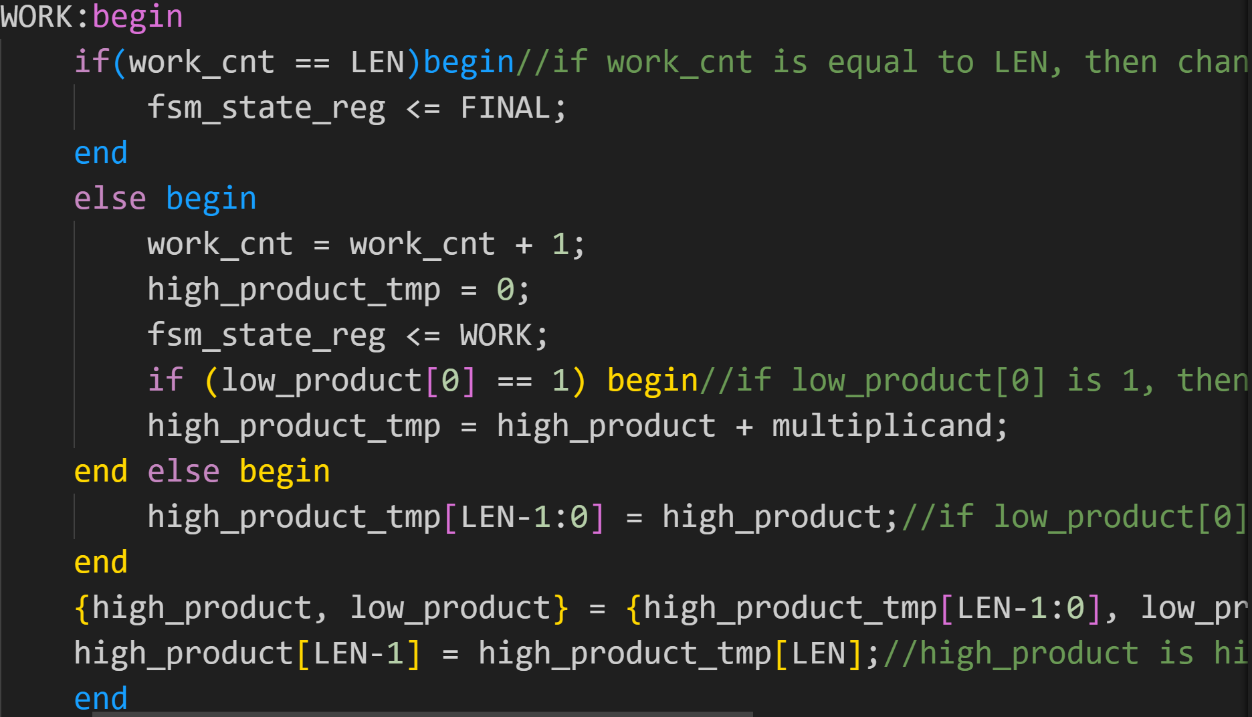
# Lab 3-3 Report

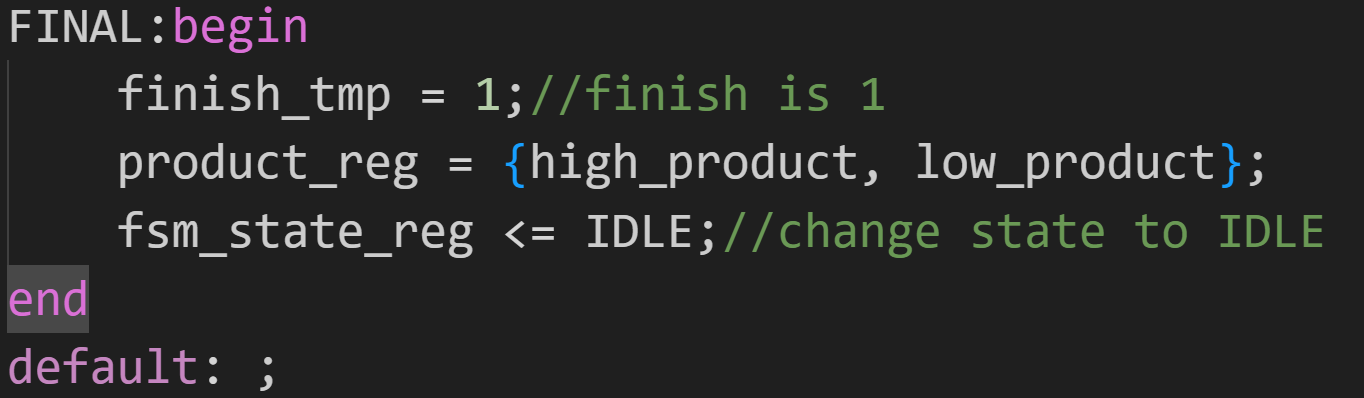
## 仿真测试

## 

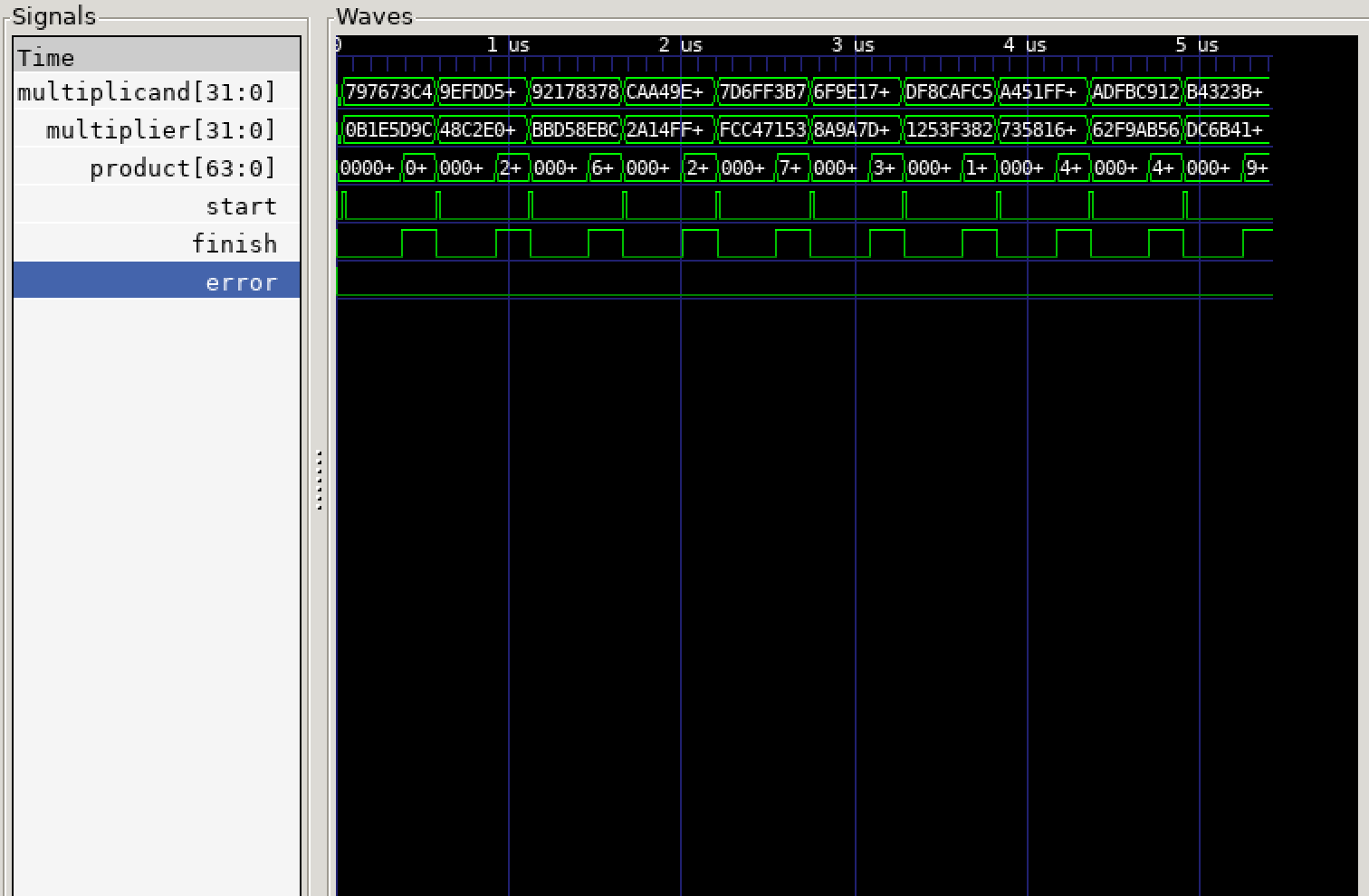
## 代码解释：

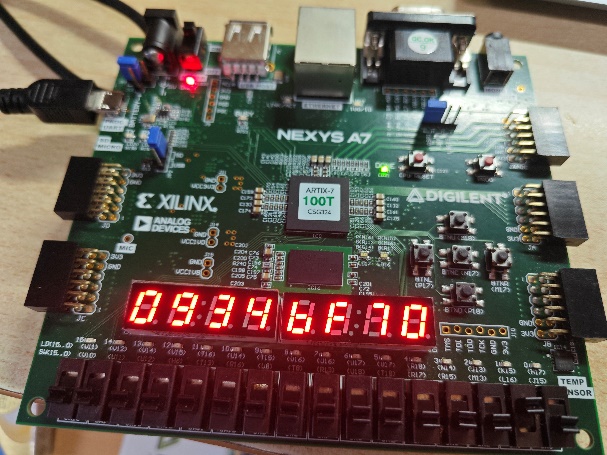
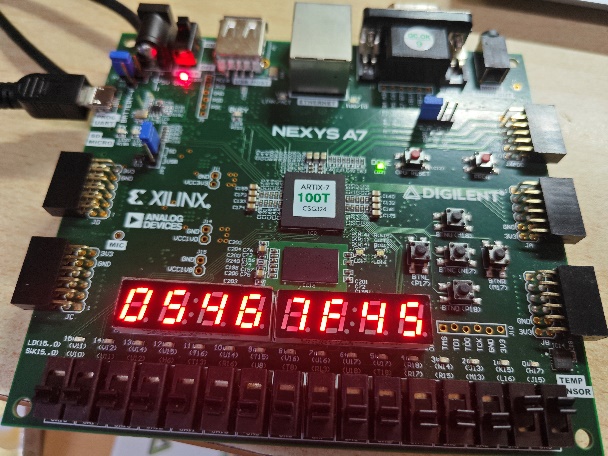






## Verilate仿真





下板验证 1

1. **解释仿真测试样例和下板的顶层结构为什么满足 start-finish 握手协议。尝试给出 start-finish 握手协议存在的缺点和改进的方法。**

对于testbench，每次生成完样例之后，都会将start置为1，等待一个周期之后再将其置为0.之后进行等待，等待finish为1之后，再进行下一次循环。caller 模块无法知道 callee 模块是否处于空闲状态，只能采用保守的策略————当 finish=1 之后再次使能 start 信号，确保 callee 一定处于空闲状态。这使得 caller 在 start 发送到 finish 返回的时间区间只能执行一次 callee 调用，但是很多时候 callee 也许可以连续执行多次 caller 调用，因此 caller 期望 callee 可以提供一个信号来表示自己是否空闲。callee 模块无法知道 caller 模块是否准备接受数据，只能采用冒险的策略————当 finish=1 的时候，返回数据只保证一个周期有限，默认 caller 会立刻接受返回的计算结果。这要求 caller 的部分组件时刻准备着接收 callee 的结果，不然可能会错过 callee 返回的结果。因此 caller 期望向 callee 提供一个信号来表示自己是否准备就绪，防止 callee 在 caller 接受之前无效掉返回结果。

**改进方法：valid-ready 握手协议**

对于任何需要被发送的数据组合，发送方提供一个 valid 信号，接受方提供一个 ready 信号。

发送方当数据线上的数据准备就绪的时候将 valid 设置为 1，表示发送方数据准备就绪，然后持续等待接收方接收数据。

接收方准备接受数据就绪时将 ready 设置为 1，表示接收方准备就绪，然后持续等待接收方接收数据。

当 valid=1 且 ready=1 时，本次握手正式完成，在下个时钟上升沿接收方就会载入发送方放置在数据线上的数据。valid 信号可以在 ready 信号之前、之后或者同时使能，两者保持独立。

握手完成时，接收方必须立刻接收数据线上的数据，如果无法继续接受数据应将 ready 设置为 0，如果不想继续接受数据可将 ready 设置为 0，如果还能接收数据且愿意接受数据则可以将 ready 继续保持为 1。

握手完成时，发送方本次发送的数据已经被接收，此时如果不打算继续发送数据则将 valid 设置为 1；如果想继续发送数据则需要立刻撤换数据线上的数据，否则接收方会再次接受原来的数据请求，重复处理。

# 32bit 无符号整数除法器

**流程图：**

1.初始化：设置初始状态，清零商、余数寄存器。

2.输入检查：如果开始信号激活，则从输入端读取被除数和除数。

3.位处理：对于每个位（从最高位到最低位）：

4.将除数左移至相对于当前处理的位对齐。

5.如果对齐后的除数小于或等于当前余数，则从余数中减去对齐后的除数，并在当前商的相应位置置位。

6.输出结果：输出商和余数。

**伪代码：**

module Divider\_32(input clk, input rst, input start, input [31:0] dividend, input [31:0] divisor, output finish, output [31:0] quotient, output [31:0] remainder)

{

if (rst) {

quotient = 0;

remainder = 0;

finish = false;

}

if (start) {

quotient = 0;

remainder = dividend; // 初始化余数为被除数

temp\_divisor = divisor;

for (i = 31; i >= 0; i--) {

temp\_divisor = divisor << i; // 将除数左移i位

if (remainder >= temp\_divisor) {

remainder -= temp\_divisor;

quotient |= (1 << i); // 在商的第i位设置1

}

}

finish = true; // 设置完成标志

}

output finish, quotient, remainder;

}

## 尝试改进目前的有限状态机，使得一次乘法操作或者连续乘法操作消耗的时钟周期数可以减少。

1. 在乘法器中的加法部分使用超前进位加法器运算。
2. 在WORK状态中，可以同时处理多个位，而不是逐位处理，如可以考虑每次处理两位，可以将迭代次数减少一半。
3. 将乘法器分隔为多个模块，运用分治算法，在每个周期计算多组结果，最后再进行汇总。
4. 使用超前进位乘法器（理论可行）