Lab3 report

Evelina

2026 年 4 月 20 日

目录

Chapter 1

Lab3-1

1.1 实验目的

- 学习使用 Verilog 语言进行时序电路设计的方法
- 掌握时序电路设计的基本思路
- 掌握有限状态机的设计和实现

1.2 实验过程

1.2.1 使用 enum 语法 FSM 设计

Listing 1.1: enum 实现 fsm

```
module FSM(
    input rstn,
    input clk,
    input a,
    input b,
    output [1:0] state
);
    typedef enum logic [1:0]{S0,S1,S2,S3} fsm_state;
    fsm_state curr_state;
    always @(posedge clk or negedge rstn) begin
        if(~rstn) begin
            curr_state <=S0;
        end
        else begin
            case(curr_state)
```

```
16                  S0: if(a) curr_state <=S1;
17                      else begin
18                          if (b) curr_state <= S0;
19                          else curr_state <= S0;
20                      end
21                  S1:
22                      if(a) curr_state <= S2;
23                      else begin
24                          if (b) curr_state <= S0;
25                          else curr_state <= S1;
26                      end
27                  S2: if(a) curr_state <= S3;
28                      else begin
29                          if (b) curr_state <= S0;
30                          else curr_state <= S2;
31                      end
32                  S3: curr_state <= S3;
33              endcase
34          end
35      end
36  assign state = curr_state;
37  endmodule
```

1.2.2 解释

使用 `typedef enum logic [1:0] {S0, S1, S2, S3} fsm_state;` 定义了一个枚举类型名，并声明 `curr_state` 为变量名。

其中`posedge clk or negedge rstn` 指的是再上升沿触发或者是在 reset 信号为 0 的时候触发

`case(curr_state)` 的循环中，依次遍历了 $S_0 \to S_3$ 的信息

下图是在 gtkwave 里面所呈现的波形图

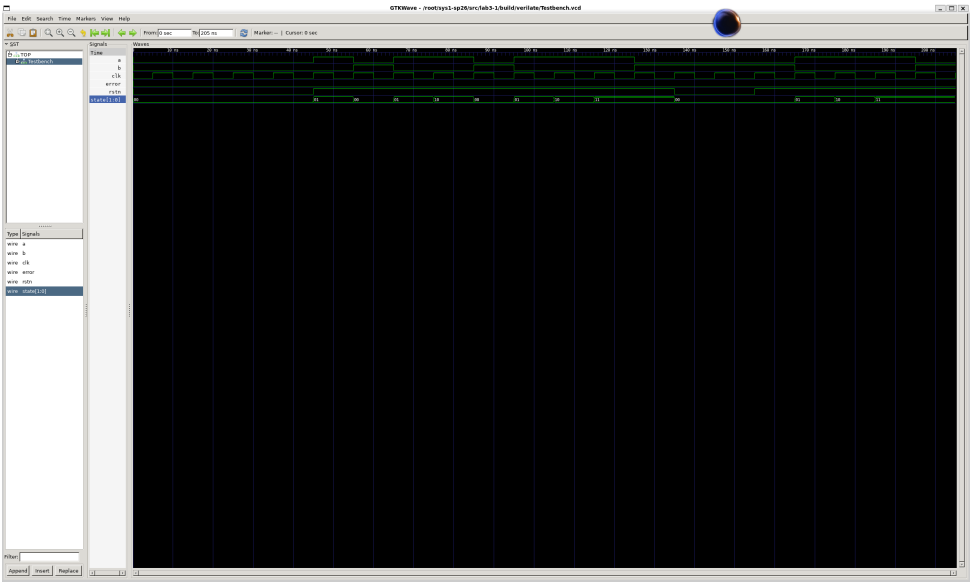

图 1.1: fsm 波形图 gtk

1.3 实验结果

如下图所示，是在上版之后的情况，其中按 N17 按钮会跳转数字，等到数字变到 3 之后就不会发生变化

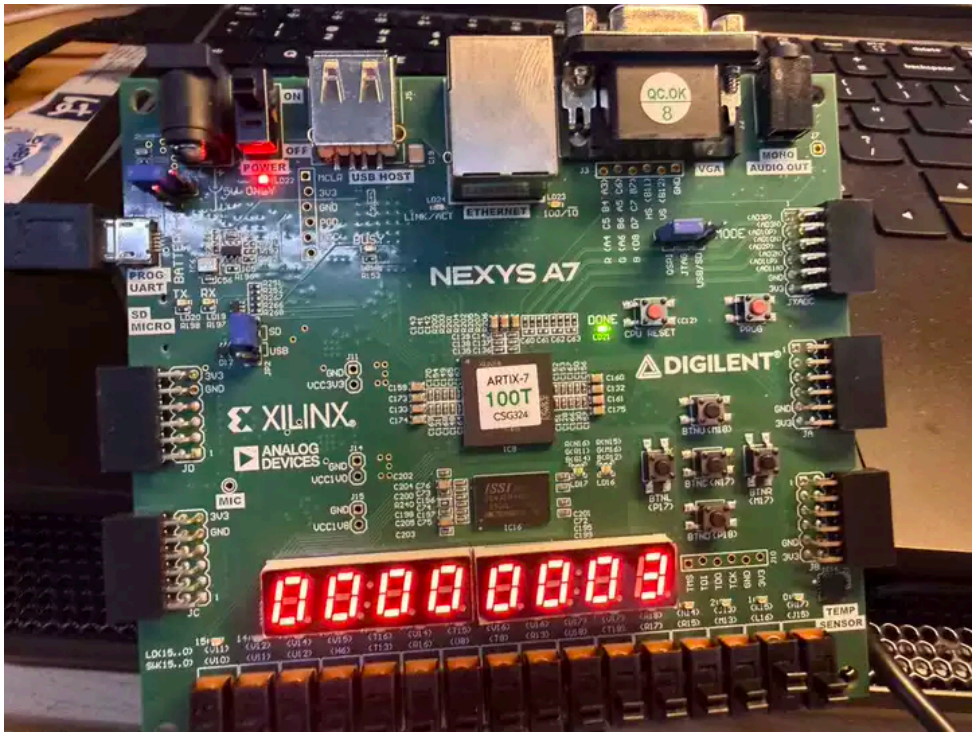

图 1.2: 综合上板实现

1.4 思考题

1.4.1 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

enum+case 的优势

- 可读性：代码的可读性非常强，而且逻辑性特别强。IDLE，WORK，FINAL 等等可以让代码处于哪个状态分析的特别清楚明白
- 可以写的很复杂：在 case 的每一种情况之下，可以嵌套 if else 语句，来实现复杂的电路。

enum+case 的缺点

- 面积开销大：每一个 case 都会被转换成多路选择器和比较逻辑，分支越多，电路延迟越高。
- 代码复杂：假如要写多种状态，则语句代码就会显得冗长。

数组查表的编程范式的优势

- 方便修改输入的数据：可以直接在.hex 文件上进行修改输入数据
- 节省逻辑资源：使用的是 bram 块存储。

数组查表的编程范式的缺点

- 初始化会比较慢：`$readmemh` 在初始化的时候会加载很久

1.4.2 绘制一个有限状态机的状态图和状态转移表，如果输入的 01 字符串中 1 的个数大于 3 个，则 FSM 输出 1 表示接受，反之输出 0 表示拒绝

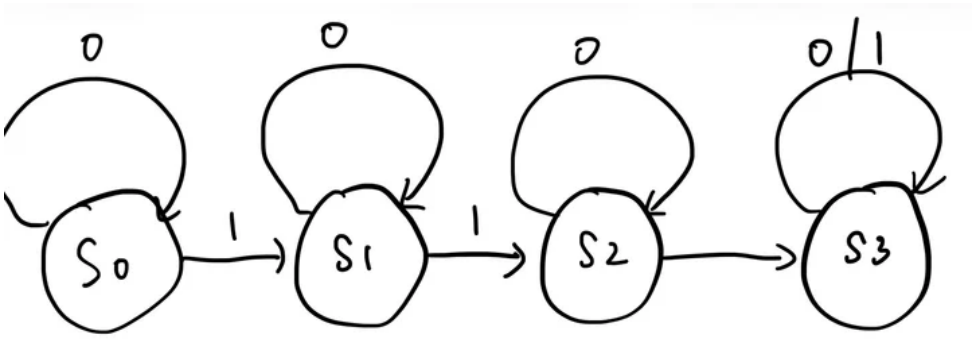

图 1.3: 有限状态机的状态图

其中 S_x 指的是状态机中有几个 1

表 1.1: 状态转移表

当前状态 (Current State)	输入 (Input X)	下一状态 (Next State)	输出 (Output Y)
S_0 (0 个 1)	0	S_0	0
	1	S_1	0
S_1 (1 个 1)	0	S_1	0
	1	S_2	0
S_2 (2 个 1)	0	S_2	0
	1	S_3	0
S_3 (3 个 1)	0	S_3	0
	1	S_4	0
S_4 (>3 个 1)	0	S_4	1
	1	S_4	1

状态转移表

1.4.3 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（bonus，分数不溢出 report）：

```
1        always@(posedge clk)begin
2        if(~rstn) state <= 2'b01;
```

```
3         else state[1:0] <= state[0:1];
4     end
```

连接的格式问题 state[1:0] 和 state[0:1] 并不是一个标准的格式。

电路会发生循环 电路会在两个状态之下反复循环。

Chapter 2

Lab3-2

2.1 实验目的

- 学习使用 Verilog 语言进行复杂电路设计的方法
- 进一步掌握组合电路、时序电路设计

2.2 实验过程

2.2.1 仿真通过，输出 success!!!

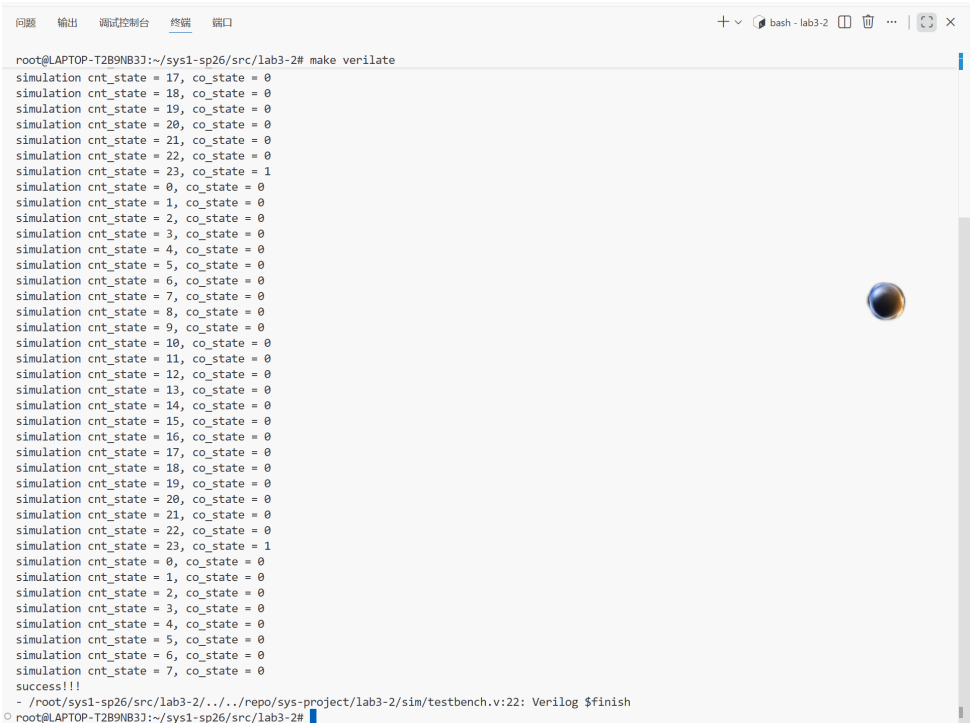

图 2.1: make verilate 出现 success!! 的图片

2.2.2 综合实现计数器

这里是我拍摄的实验视频的链接，请点击观看：https://www.bilibili.com/video/BV1eMoTBJEqt/?spm_id_from=333.1387.homepage.video_card.click&vd_source=cca86c858a3fea

本文档所提供的视频仅用于作业提交，提交完成后将会下架。

图 2.2: make verilate 出现 success!! 的图片

照片的形式

2.3 实验结果

2.3.1 代码展示

下面将展示实现计数器的核心代码：

Listing 2.1: Cnt.sv

```
1  module Cnt #(
2  parameter BASE = 10,
```

```
    parameter INITIAL = 0
) (
    input en,
    input clk,
    input rstn,
    input low_co,
    input high_rst,
    output co,
    output reg [3:0] cnt
);
    initial begin
        cnt = INITIAL[3:0];
    end
    always@(posedge clk or negedge rstn) begin
        if(~rstn)begin
            cnt<=INITIAL[3:0];
        end
        else if(high_rst)begin
                cnt<=4'b0000;
            end
        else if(en)begin
            if(low_co)begin
                if(cnt==BASE-1)begin
                    cnt<=4'b0000;
                end
                else begin
                    cnt<=cnt+1;
                end
            end

        end
    end
assign co=(cnt==BASE-1)&&low_co;
endmodule
```

Listing 2.2: Cnt2num.sv

```
    module Cnt2num #(
    parameter BASE = 24,
    parameter INITIAL = 16
)(
```

```
5       input en,
6       input clk,
7       input rstn,
8       input high_rst,
9       input low_co,
10      output co,
11      output [7:0] cnt
12  );
13
14      localparam HIGH_BASE = 10;
15      localparam LOW_BASE  = 10;
16      localparam HIGH_INIT = INITIAL/10;
17      localparam LOW_INIT  = INITIAL%10;
18         /* verilator lint_off WIDTHTRUNC */
19      localparam [3:0] HIGH_CO = (BASE-1)/10;
20      localparam [3:0] LOW_CO  = (BASE-1)%10;
21      /* verilator lint_on WIDTHTRUNC */
22      // fill the code
23      wire [3:0]high_cnt1;
24      wire [3:0]low_cnt1;
25      wire co_high;
26      wire co_low;
27      wire high_rst_internal;
28      assign high_rst_internal = (high_cnt1 == HIGH_CO) && (
           low_cnt1 == LOW_CO) && low_co;
29      Cnt #(
30          .BASE(HIGH_BASE),
31          .INITIAL(HIGH_INIT)
32      ) high_cnt(
33          .en(en),
34          .clk(clk),
35          .rstn(rstn),
36          .low_co(co_low),
37          .high_rst(high_rst_internal),
38          .co(co_high),
39          .cnt(high_cnt1)
40      );
41      Cnt #(
42          .BASE(LOW_BASE),
43          .INITIAL(LOW_INIT)
```

```
    ) low_cnt(
        .en(en),
        .clk(clk),
        .rstn(rstn),
        .low_co(low_co),
        .high_rst(high_rst_internal),
        .co(co_low),
        .cnt(low_cnt1)
    );

    assign co = (high_cnt1 == HIGH_CO) && (low_cnt1 == LOW_CO) &&
        low_co;
    assign cnt = (high_cnt1*10+low_cnt1)==BASE ? 8'b0000_0000 : {
       high_cnt1, low_cnt1};


endmodule
```

2.4 思考题

2.4.1 简述如何使用 Cnt2num 实现 1234 的 BCD 码计数器，并思考 Cnt2num 预留 co、low_co、high_rst 引脚的意义

Q1：实现方案 实现 1234 计数器需要级联两个 `Cnt2num` 模块，分别负责高两位（12）和低两位（34）的计数。当计数器数值达到设定的 `BASE`（即 1234）时，通过反馈逻辑将数值重置为 `4'b0000`。整体框架采用状态机（FSM）设计，利用 `enum` 定义状态，并在 `case` 语句中根据时钟信号进行状态转移。

Q2：引脚意义分析

- `co` (Carry Out)：溢出进位标志。当本级计数器达到最大值发生溢出时，`co` 置为 1，用作下一级计数器的使能信号，实现高位加 1。
- `low_co` (Low-bit Carry)：低位进位输入。用于描述来自低位计数器的进位状态。若 `low_co` 有效，说明本位计数器需要进入下一个计数状态。
- `high_rst` (High-level Reset)：高位复位信号。用于描述高位是否触发了整体复位条件。当 `high_rst` 为 1 时，系统强制将所有计数单元清零.

Chapter 3

Lab3-3

3.1 实验目的

- 继续学习时序电路设计的方法
- 设计实现乘法器

3.2 实验过程

3.2.1 仿真通过，输出 success!!!

```
root@LAPTOP-T2B9NB3J:~/sys1-sp26/src/lab3-3# make verilate
rameter -Wno-unused-variable    -DVL_DEBUG -DTOP=Testbench -std=c++17 -DVL_TIME_CONTEXT   -fcoroutines -c -o verilated_threads.
o /usr/local/share/verilator/include/verilated_threads.cpp
/usr/bin/python3 /usr/local/share/verilator/bin/verilator_includer -DVL_INCLUDE_OPT=include VTestbench.cpp VTestbench___024root
__DepSet_ha7d5f4a7__0.cpp VTestbench___024root__DepSet_h17da5287__0.cpp VTestbench___024unit__DepSet_h805a7447__0.cpp VTestbenc
h__main.cpp VTestbench__Dpi.cpp VTestbench__Trace__0.cpp VTestbench__ConstPool_0.cpp VTestbench___024root__Slow.cpp VTestbench_
__024root__DepSet_ha7d5f4a7__0__Slow.cpp VTestbench___024root__DepSet_h17da5287__0__Slow.cpp VTestbench___024unit__Slow.cpp VTe
stbench___024unit__DepSet_h3458d327__0__Slow.cpp VTestbench__Syms.cpp VTestbench__Trace__0__Slow.cpp VTestbench__TraceDecls__0_
_Slow.cpp > VTestbench__ALL.cpp
ccache g++ -Os  -I.  -MMD -I/usr/local/share/verilator/include -I/usr/local/share/verilator/include/vltstd -DVM_COVERAGE=0 -DVM
_SC=0 -DVM_TRACE=1 -DVM_TRACE_FST=0 -DVM_TRACE_VCD=1 -faligned-new -fcf-protection=none -Wno-bool-operation -Wno-overloaded-vir
tual -Wno-shadow -Wno-sign-compare -Wno-uninitialized -Wno-unused-but-set-parameter -Wno-unused-but-set-variable -Wno-unused-pa
rameter -Wno-unused-variable    -DVL_DEBUG -DTOP=Testbench -std=c++17 -DVL_TIME_CONTEXT   -fcoroutines -c -o VTestbench__ALL.o
VTestbench__ALL.cpp
echo "" > VTestbench__ALL.verilator_deplist.tmp
Archive ar -rcs VTestbench__ALL.a VTestbench__ALL.o
g++    judge.o verilated.o verilated_dpi.o verilated_vcd_c.o verilated_timing.o verilated_threads.o VTestbench__ALL.a    -pthre
ad -lpthread -latomic   -o Testbench
rm VTestbench__ALL.verilator_deplist.tmp
make[1]: Leaving directory '/root/sys1-sp26/src/lab3-3/build/verilate'
cd /root/sys1-sp26/src/lab3-3/build/verilate; ./Testbench
simulation multiplicand = 797673c4, multiplier = 0b1e5d9c, product = 05467f450934bf70
simulation multiplicand = 9efdd502, multiplier = 48c2e0e4, product = 2d306befaff775c8
simulation multiplicand = 92178378, multiplier = bbd58ebc, product = 6b310c0d5f091c20
simulation multiplicand = caa49eb6, multiplier = 2a14ffe4, product = 214fa174b6eca418
simulation multiplicand = 7d6ff3b7, multiplier = fcc47153, product = 7bda7525d1fbcb55
simulation multiplicand = 6f9e1721, multiplier = 8a9a7de1, product = 3c6e9473ff177101
simulation multiplicand = df8cafc5, multiplier = 1253f382, product = 1001339ce728410a
simulation multiplicand = a451ff62, multiplier = 73581635, product = 4a096029e09c4b4a
simulation multiplicand = adfbc912, multiplier = 62f9ab56, product = 434411511866920c
simulation multiplicand = b4323bfd, multiplier = dc6b41f4, product = 9b26aaf8ffdb6a24
simulation multiplicand = 06adf53b, multiplier = c66d4d58, product = 052d65c1aaaae0b48
simulation multiplicand = e9cf04b2, multiplier = 56d1085c, product = 4f4a75391fdd3ff8
simulation multiplicand = 5a1b851e, multiplier = ad1f459e, product = 3cef9acb6c7f3e84
simulation multiplicand = 3df8999e, multiplier = 7b331cee, product = 1dd2d156526618e4
simulation multiplicand = 983b6b10, multiplier = 40fb7f76, product = 26a468d19e304960
simulation multiplicand = 5a50ed63, multiplier = 599fa2c0, product = 1f9e7435ce67b040
simulation multiplicand = 01c8bb1d, multiplier = c11b0f22, product = 0158855798ae8cda
simulation multiplicand = ef938f06, multiplier = 85ee5b81, product = 7d56ba1f42c63406
simulation multiplicand = 792d3263, multiplier = c454c278, product = 5ceeb97561daa468
success!!!
- /root/sys1-sp26/src/lab3-3/sim/testbench.v:30: Verilog $finish
root@LAPTOP-T2B9NB3J:~/sys1-sp26/src/lab3-3#
```

图 3.1: make verilate 出现 success!! 的图片

3.2.2 综合实现乘法器

下面将展示其中一组乘法器的结果

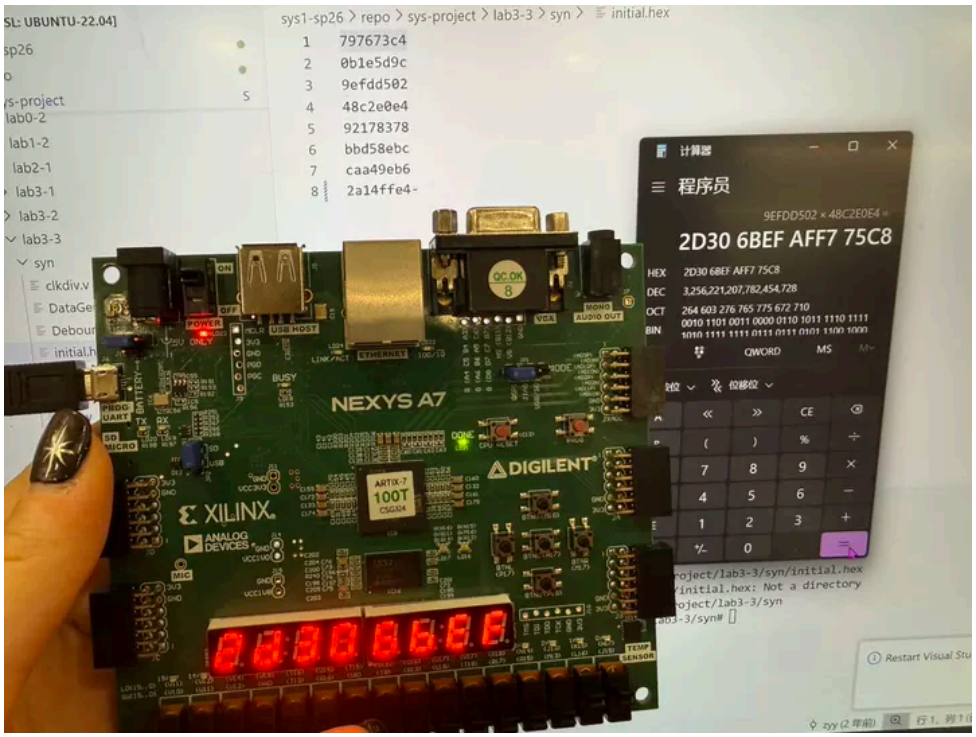

图 3.2: make verilate 出现 success!! 的图片

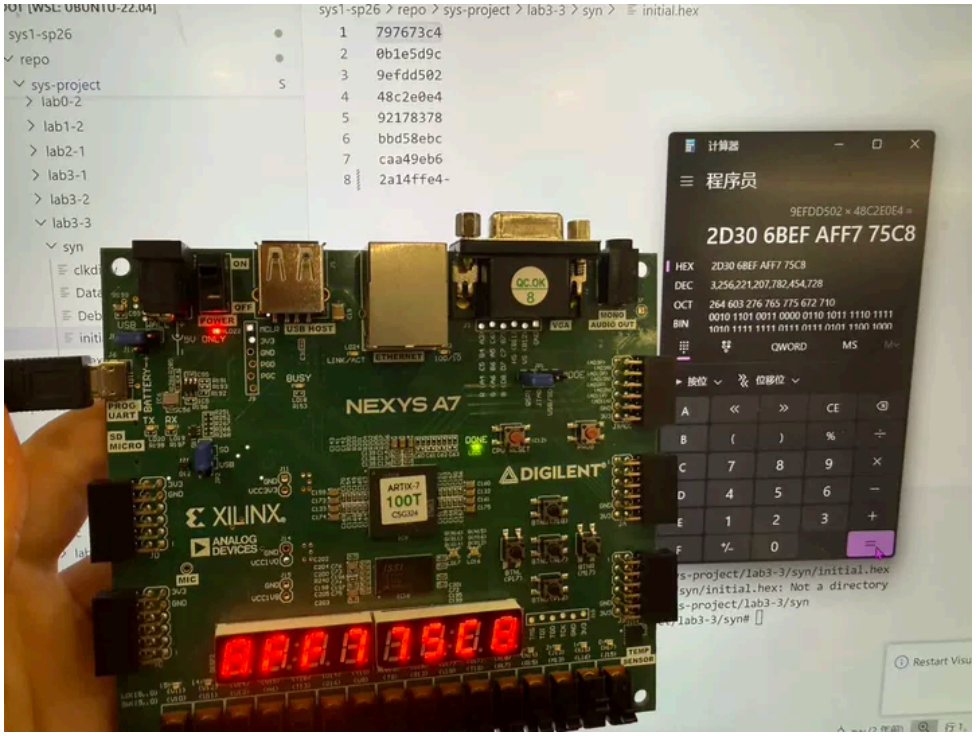

图 3.3: make verilate 出现 success!! 的图片

3.3 实验结果

实验结果部分将展示核心代码：

Listing 3.1: Multiplier.sv

```
    module Multiplier #(
    parameter LEN = 32
    // 乘数、被乘数的长度参数
) (
    input clk,
    input rst,
    // 使用 rst, rst = 1 复位
    input [LEN-1:0] multiplicand,
    input [LEN-1:0] multiplier,
    input start,

    output [LEN*2-1:0] product,
    output finish
);

    localparam PRODUCT_LEN = LEN*2;
    // 表示乘积的位宽
    logic [LEN-1:0] multiplicand_reg;
    logic [PRODUCT_LEN-1:0] product_reg;

    localparam CNT_LEN = $clog2(LEN);
    // 表示迭代计数器的位宽
    localparam CNT_NUM = LEN - 1;
    // 迭代需要的立即数，大家按需使用
    logic [CNT_LEN-1:0] work_cnt;

    typedef enum [1:0] {IDLE, WORK, FINAL} fsm_state;
    // 有限状态机
    fsm_state fsm_state_reg;
always @(posedge clk or posedge rst) begin
    if(rst) begin
        fsm_state_reg <= IDLE;
    end
    else begin
        case(fsm_state_reg)
```

```
36          IDLE:begin
37              if(start) begin
38              multiplicand_reg<=multiplicand;
39              product_reg <= 0;
40              work_cnt <=0;
41              fsm_state_reg <= WORK;
42              end
43          end
44          WORK: begin
45              logic [32:0] next_sum;
46              next_sum=multiplier[work_cnt]?({1'b0, product_reg
                    [63:32]} + {1'b0, multiplicand_reg}):{1'b0,
                    product_reg[63:32]};
47              product_reg<={next_sum,product_reg[31:1]};
48              work_cnt<=work_cnt+1;
49              if (work_cnt==5'd31) begin
50              fsm_state_reg <= FINAL;
51      end
52      end
53          FINAL:begin
54              fsm_state_reg<=IDLE;
55          end
56          default: begin
57          fsm_state_reg <= IDLE;
58      end
59      endcase
60  end
61  end
62
63  assign product=product_reg;
64  assign finish=fsm_state_reg==FINAL?1'b1:1'b0;
65  endmodule
```

3.4 思考题

3.4.1 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。

在顶层设计中，只有在 btn 的信号为 1 的时候才会进行计算。当计算结果结束之后，才可以继续有效接受 btn 的信号，也就是相当于 finish。

在上板中，只有按下了 N17 的按钮之后才会开始进行计算，其中 N17 的按钮即为 start 信号。而之后在计算完毕显示玩数字之后才可以继续按下 N17 的按钮，继续下一组的计算。那么计算完毕的返回信号即为 finish 信号。

所以整体是满足握手协议的。

缺点：

- 高延迟：每一次状态的改变，都要接收到 finish 信号才可以进行下一次的运算。耗时较长
- 数据传输量低下：由于发送端在收到确认信号前必须保持阻塞状态，总线或信道在握手期间大部分时间处于闲置。
- 功耗大：每一次传输伴随着四次信号的变化。

改进：我觉得可以在 start 还没有的得到 finish 的指令时候之前进行改进。在电路的空闲部分可以提前进行下一次数据的处理，来保证电路硬件结构的不浪费。

3.4.2 请仿照乘法器的设计方法和我们手动计算除法的方式,设计 32bit 无符号整数除法器，你只需要给出设计思路即可。流程图和伪代码是推荐的描述形式。

```
1   module Divider_32(
2       input     clk,
3       input     rst,
4
5       input     start,
6       input     [31:0] dividend,
7       input     [31:0] divisor,
8
9       output     finish,
10      output     [31:0] quotient,
11      output     [31:0] remainder
```

```
        );

endmodule
```

Listing 3.2: 伪代码

```
define [31:0]devisor_t={divisor,0000……} //将divisor中的元素移到开头。
define i=31;
while decidend>=decisor do
if devisor_t>devidend do
    remainder[i] =1
    end
else do
    reminder[i]=0
end
i=i-1;
end
remainder=devidend;
quotient={0000…… ,quetient[31:31-i]}
```

3.4.3 bonus

我觉得可以删除 final 的状态，直接在 work 的状态里加一个判断，直接将 finish 置为一，这样可以节省一个时钟周期。

3.5 心得体会

我尝试这书写了伪代码!!! 学会了乘法器!!