lab 3-1: 时序电路设计之有限状态机

1. 实验环境

1. 操作系统：Ubuntu 24.04
2. VHDL：Verilog，SystemVerilog

2. 实验目的

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

3. 实验报告 30%

3.1 每一步的过程&截图和解释 15%

3.1.1 使用 enum 语法 FSM 设计 5%

1. 使用`enum`语法实现`FSM.sv`，部分代码：

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

采用“状态寄存器 + 条件转移”写法。

`curr_state`在时钟上升沿更新，低电平复位`rstn`有效时回到`S0`。当输入`a`有效时，状态沿`S0->S1->S2->S3`推进；当输入`b`有效时，状态回到`S0`。

仿真通过，输出`success!!!`：

```
parallels@ubuntu-linux-2404:~/sys1-sp26/src/lab3-1$ make verilate
mkdir -p /home/parallels/sys1-sp26/src/lab3-1/build/verilate
verilator -Wno-STMTDLY --timescale 1ns/10ps --trace --cc --exe  --main --timing --Mdir /home/parallels/sys1-sp26/src/lab3-1/buil
d/verilate --top-module Testbench -o Testbench -I/home/parallels/sys1-sp26/src/lab3-1/include -CFLAGS "-DVL_DEBUG -DTOP=Testbenc
h -std=c++17"  /home/parallels/sys1-sp26/src/lab3-1/../../repo/sys-project/lab3-1/sim/judge.v /home/parallels/sys1-sp26/src/lab3
-1/../../repo/sys-project/lab3-1/sim/judge.cpp /home/parallels/sys1-sp26/src/lab3-1/../../repo/sys-project/lab3-1/sim/testbench.
v /home/parallels/sys1-sp26/src/lab3-1/submit/FSM.sv +define+TOP_DIR=\"/home/parallels/sys1-sp26/src/lab3-1/build/verilate\" +de
fine+VERILATE
make -C /home/parallels/sys1-sp26/src/lab3-1/build/verilate -f VTestbench.mk Testbench
make[1]: 进入目录“/home/parallels/sys1-sp26/src/lab3-1/build/verilate”
make[1]: “Testbench”已是最新。
make[1]: 离开目录“/home/parallels/sys1-sp26/src/lab3-1/build/verilate”
cd /home/parallels/sys1-sp26/src/lab3-1/build/verilate; ./Testbench
success!!!
- /home/parallels/sys1-sp26/src/lab3-1/../../repo/sys-project/lab3-1/sim/testbench.v:49: Verilog $finish
```

Image 1

3. 波形图：其中

   (a) `a=1`、`b=0`：输入的是字符`a`

   (b) `a=0`、`b=1`：输入的是字符`b`

   (c) `a=1`、`b=1`：输入的是字符`a`

   (d) `a=0`、`b=0`：没有输入，状态保持不变

> 当输入连续三个`a`后，`state=3`，需要`rstn`复位才会重置为`0`。

从波形可见，状态变化都发生在`clk`上升沿，电路行为符合预期。

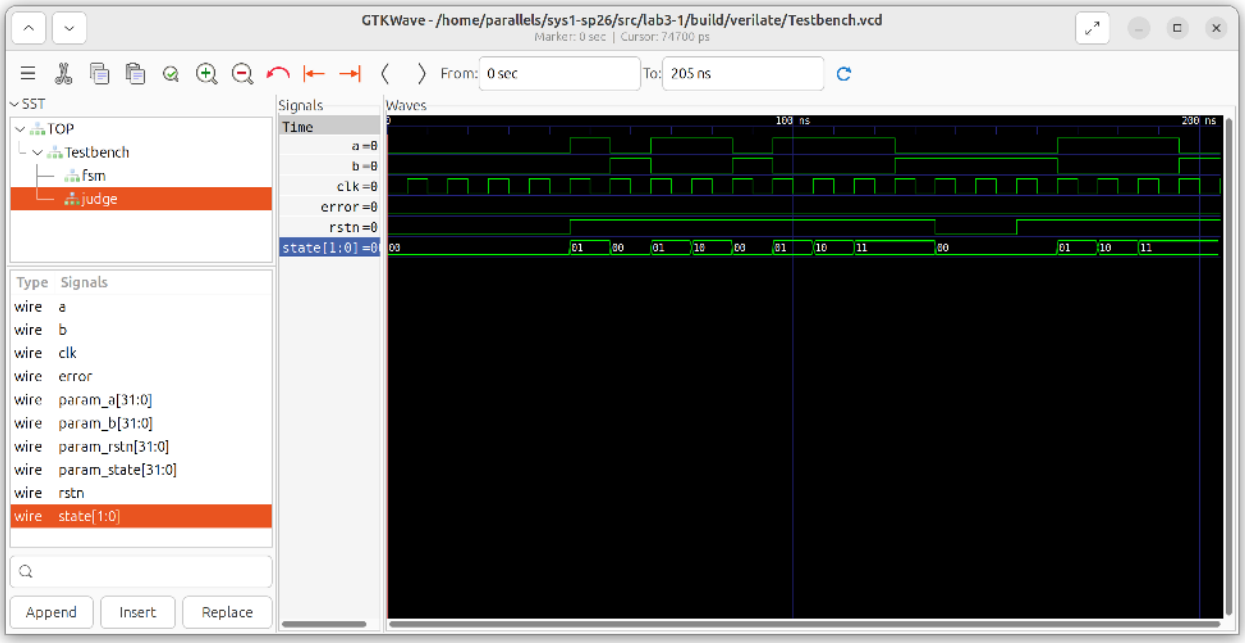


Image 2

3.1.2 综合实现有限状态机 10%

下板图片如下：

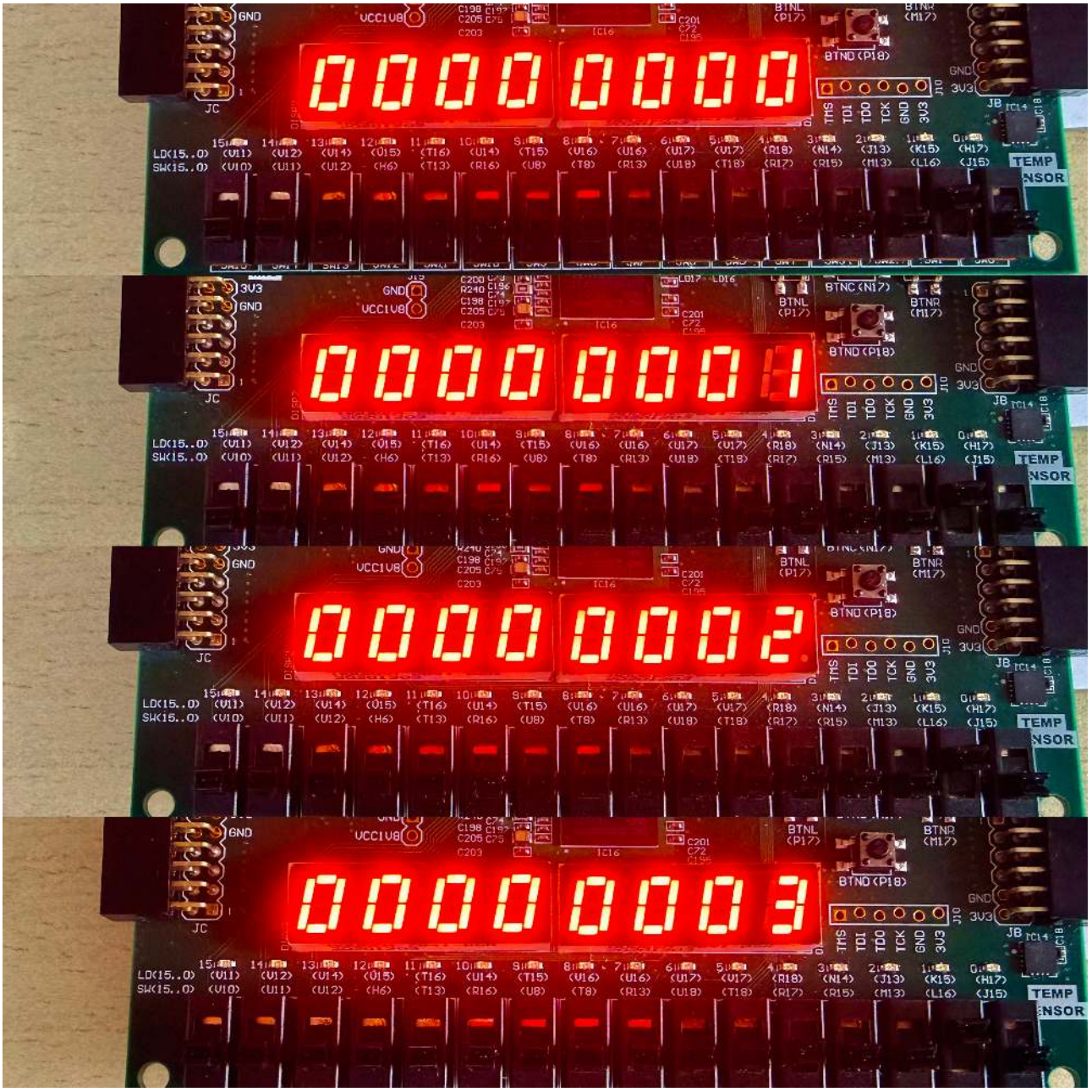


Image 3

下板现象与仿真一致，按键输入对应 `a` 、 `b` 后，状态按设计进行推进或回退。

3.2 思考比较 `enum + case` 的编程范式和数组查表的编程范式的优劣 5%

1. 数组查表
   (a) 优点是存储效率更高、代码简洁；
   (b) 缺点是依赖 `hex` 等外部数据文件，若无自动生成脚本，后续改动和调试较困难。
2. `enum + case`
   (a) 优点是代码可读性强、语义直观；
   (b) 缺点是存储效率较低，且需要多个条件判断结构。

实验中优先使用 `enum + case` 更合适，因为它更便于展示FSM设计思路与验证过程。

3.3 绘制一个有限状态机的状态图和状态转移表 10%

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。

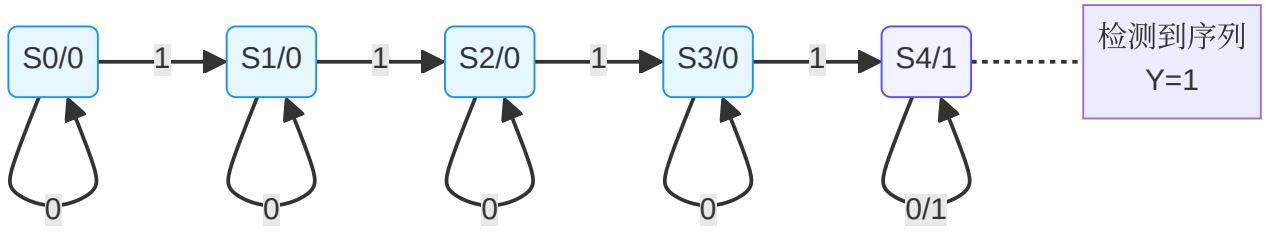
状态含义可理解为“当前已累计到的`1`数量区间”：`S0/S1/S2/S3`分别表示累计到`0/1/2/3`个`1`，`S4`表示“至少4个`1`”。因此一旦进入`S4`，在后续输入`0/1`下都保持接受。

状态转移表：

当前状态	下一状态 x = 0	下一状态 x = 1	输出 y
S0	S0	S1	0
S1	S1	S2	0
S2	S2	S3	0
S3	S3	S4	0
S4	S4	S4	1

Table 1

流程图：



3.4 有限状态机电路实现的不足

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）：+10%

```
always@(posedge clk)begin
    if(~rstn) state <= 2'b01;
    else state[1:0] <= state[0:1];
end
```

1. FSM需要有输入`X`，在这里只是翻转，无法表达有条件的状态转移或输出逻辑，不是真正的`FSM`。
2. 切片语法必须升序，`state[0:1]`应改为`{state[0], state[1]}`。
3. 未处理非法状态。

- 不稳定（上电、复位发生毛刺等）时电路可能会出现非法的未定义状态`00`和`11`。

lab 3-2: 计数器 / 定时器设计与应用

1. 实验环境

1. 操作系统：Ubuntu 24.04
2. VHDL：Verilog，SystemVerilog

2. 实验目的

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

3. 实验报告 30%

3.1 每一步的过程&截图和解释 20%

3.1.1 仿真通过，输出 `success!!!` 10%

`Cnt.sv` 部分

```
always@(posedge clk) begin                  //此处有问题，将在下文指出！
    if(~rstn) cnt <= INITIAL[3:0];
    else if(high_rst) cnt <= 0;
    else if(en) begin
        if(low_co) begin
            if(cnt == BASE-1) cnt <= 0;
            else cnt <= cnt + 1;
        end
    end
end
assign co = (cnt == BASE-1) && low_co && en;
```

> ⚠ **Warning**
>
> 上板后发现**复位无效**！发现板子的时钟分频器在reset时会复位，导致reset不会被识别到，这和仿真时的行为不同。尽管仿真时success，上板时复位功能失效。
>
> 在这里需要用异步复位。
>
> 将 `always@(posedge clk) begin` 改为
>
> `always@(posedge clk or negedge rstn) begin` 即可解决上板时的复位问题。

`Cnt.sv` 实现了带使能 `en` 、可置零 `high_rst` 和进位输入 `low_co` 的基础计数单元；

`Cnt2num.sv` 部分

```
logic co_low_cnt;
Cnt #(.BASE(LOW_BASE), .INITIAL(LOW_INIT)) low_cnt(
    .en(en),
    .clk(clk),
```

```
    .rstn(rstn),
    .low_co(low_co),
    .high_rst(high_rst||co),
    .co(co_low_cnt),
    .cnt(cnt[3:0])
);
Cnt #(.BASE(HIGH_BASE), .INITIAL(HIGH_INIT)) high_cnt(
    .en(en),
    .clk(clk),
    .rstn(rstn),
    .low_co(co_low_cnt),
    .high_rst(high_rst||co),
    .co(),
    .cnt(cnt[7:4])
);
assign co = (cnt == {HIGH_CO[3:0], LOW_CO[3:0]}) && en && low_co;
```

`Cnt2num.sv` 通过高低位级联实现两位计数。低位溢出后通过 `co_low_cnt` 驱动高位进位。

仿真通过，输出 `success!!!`

```
parallels@ubuntu-linux-2404:~/sys1-sp26/src/lab3-2$ make verilate
make[1]: “Testbench”已是最新。
make[1]: 离开目录“/home/parallels/sys1-sp26/src/lab3-2/build/verilate”
cd /home/parallels/sys1-sp26/src/lab3-2/build/verilate; ./Testbench
init cnt_state = 16
simulation cnt_state = 16, co_state = 0
simulation cnt_state = 17, co_state = 0
simulation cnt_state = 18, co_state = 0
simulation cnt_state = 19, co_state = 0
simulation cnt_state = 20, co_state = 0
simulation cnt_state = 21, co_state = 0
simulation cnt_state = 22, co_state = 0
simulation cnt_state = 23, co_state = 1
simulation cnt_state = 0, co_state = 0
simulation cnt_state = 1, co_state = 0
simulation cnt_state = 2, co_state = 0
simulation cnt_state = 3, co_state = 0
simulation cnt_state = 4, co_state = 0
simulation cnt_state = 5, co_state = 0
simulation cnt_state = 6, co_state = 0
simulation cnt_state = 7, co_state = 0
simulation cnt_state = 8, co_state = 0
simulation cnt_state = 9, co_state = 0
simulation cnt_state = 10, co_state = 0
simulation cnt_state = 11, co_state = 0
simulation cnt_state = 12, co_state = 0
simulation cnt_state = 13, co_state = 0
simulation cnt_state = 14, co_state = 0
simulation cnt_state = 15, co_state = 0
simulation cnt_state = 16, co_state = 0
simulation cnt_state = 17, co_state = 0
simulation cnt_state = 18, co_state = 0
simulation cnt_state = 19, co_state = 0
simulation cnt_state = 20, co_state = 0
simulation cnt_state = 21, co_state = 0
simulation cnt_state = 22, co_state = 0
simulation cnt_state = 23, co_state = 1
simulation cnt_state = 0, co_state = 0
simulation cnt_state = 1, co_state = 0
simulation cnt_state = 2, co_state = 0
simulation cnt_state = 3, co_state = 0
simulation cnt_state = 4, co_state = 0
simulation cnt_state = 5, co_state = 0
simulation cnt_state = 6, co_state = 0
simulation cnt_state = 7, co_state = 0
success!!!
- /home/parallels/sys1-sp26/src/lab3-2/../../repo/sys-project/lab3-2/sim/testbench.v:22: Verilog $finish
```

Image 4

3.1.2 综合实现计数器 10%

上板截图如下，取复位瞬间：

复位瞬间观测到计数值清零，`co = 1`，且后续按时钟递增，和仿真结果一致。

Image 5

3.2 四位BCD码计数器 10%

> 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

与二位BCD码类似，实现`1234`可将两个`Cnt2num`继续级联：低两位负责个位/十位，高两位负责百位/千位，并保持同样的进位连接方式。

`co`、`low_co`、`high_rst`预留的意义在于模块化扩展：

- `low_co`：接收低位的进位；
- `co`：向更高位输出进位，便于继续拼接更长位宽，
  - 最高位的`co`也可作为到达边界时的溢出标志，实现计数器的清零；
- `high_rst`：提供统一控制，以便于计数器复位。

lab 3-3: 乘法器

1. 实验环境

1. 操作系统：Ubuntu 24.04
2. VHDL：Verilog，SystemVerilog

2. 实验目的

1. 继续学习时序电路设计的方法
2. 设计实现乘法器

3. 实验报告 40%

3.1 每一步的过程&截图和解释 20%

3.1.1 仿真通过，输出 `success!!!` 10%

`Multiplier.sv` 部分代码：

```
logic finish_reg;
logic [LEN-1:0] add;     //实际上是wire
logic carry;             //wire
assign {carry,add} = product_reg[PRODUCT_LEN-1:LEN] + multiplicand_reg;
always@(posedge clk or posedge rst) begin
    if(rst) begin//重置
        fsm_state_reg <= IDLE;
        multiplicand_reg <= 0;
        product_reg <= 0;
        work_cnt <= 0;
    end
    else begin
        case(fsm_state_reg)
            IDLE: begin
                finish_reg <= 1'b0;
                if(start) begin//start握手信号
                    fsm_state_reg <= WORK;
                    multiplicand_reg <= multiplicand;//被乘数
                    product_reg <= {{LEN{1'b0}}, multiplier};//乘数->积的
                    低位，高位补0
                    work_cnt <= CNT_NUM[CNT_LEN-1:0];
                end
            end
            WORK: begin
                if(work_cnt == 0)begin
                    fsm_state_reg <= FINAL;
                end
                if(product_reg[0] == 1'b1) begin
                    product_reg <= {carry, add, product_reg[LEN-
                    1:1]};//[:1]实现右移，避免位宽不匹配的警告
```

```
                end
                else begin
                    product_reg <= (product_reg >> 1);
                end
                work_cnt <= work_cnt - 1;
            end
            FINAL:begin
                finish_reg <= 1'b1;//finish握手信号
                fsm_state_reg <= IDLE;
            end
            default: fsm_state_reg <= IDLE;
        endcase
    end
end
assign finish = finish_reg;//输出finish信号
assign product = product_reg;
```

移位加法乘法器：在 `WORK` 状态中，每拍检查 `product_reg[0]`：若为 `1`，先加被乘数再右移；若为 `0`，直接右移。循环 `LEN` 次后进入 `FINAL` 并拉高 `finish`，最后回到 `IDLE` 等待下一次 `start`。

仿真通过，输出 `success!!!`

```
parallels@ubuntu-linux-2404:~/sys1-sp26/src/lab3-3$ make verilate
mkdir -p /home/parallels/sys1-sp26/src/lab3-3/build/verilate
verilator -Wno-STMTDLY --timescale 1ns/10ps --trace --cc --exe  --main --timing --Mdir /home/parallels/sys1-sp26/src/lab3-3/build/verilate --top-module Testbenc
h -o Testbench -CFLAGS "-DVL_DEBUG -DTOP=Testbench -std=c++17"  /home/parallels/sys1-sp26/src/lab3-3/sim/judge.v /home/parallels/sys1-sp26/src/lab3-3/sim/judge.
cpp /home/parallels/sys1-sp26/src/lab3-3/sim/testbench.v /home/parallels/sys1-sp26/src/lab3-3/submit/Multiplier.sv +define+TOP_DIR=\"/home/parallels/sys1-sp26/s
rc/lab3-3/build/verilate\" +define+VERILATE
make -C /home/parallels/sys1-sp26/src/lab3-3/build/verilate -f VTestbench.mk Testbench
make[1]: 进入目录“/home/parallels/sys1-sp26/src/lab3-3/build/verilate”
make[1]: “Testbench”已是最新。
make[1]: 离开目录“/home/parallels/sys1-sp26/src/lab3-3/build/verilate”
cd /home/parallels/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 = 1001339ce720410a
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 = 052d65c1aaae0b48
simulation multiplicand = e9cf04b2, multiplier = 56d1085c, product = 4f4a75391fdd3ff8
simulation multiplicand = 5a1b851e, multiplier = ad1f459e, product = 3cef9acb6c7f3e84
simulation multiplicand = 3df8999e, multiplier = 7b331cee, product = 1dd2d156526618c4
simulation multiplicand = 983b6b10, multiplier = 40fb7f76, product = 26a468d19e304960
simulation multiplicand = 5a50ed63, multiplier = 599fa2c0, product = 1f9e7435ce67b040
success!!!
- /home/parallels/sys1-sp26/src/lab3-3/sim/testbench.v:33: Verilog $finish
parallels@ubuntu-linux-2404:~/sys1-sp26/src/lab3-3$
```

Image 6

3.1.2 综合实现乘法器 10%

计算四组数据的乘法结果：

```
parallels@ubuntu-linux-2404: ~
parallels@ubuntu-linux-2404:~$ \
echo "ibase=16; 797673C4 * 0B1E5D9C" | bc | xargs printf "乘积1: %016X\n"
echo "ibase=16; 9EFDD502 * 48C2E0E4" | bc | xargs printf "乘积2: %016X\n"
echo "ibase=16; 92178378 * BBD58EBC" | bc | xargs printf "乘积3: %016X\n"
echo "ibase=16; CAA49EB6 * 2A14FFE4" | bc | xargs printf "乘积4: %016X\n"
乘积1: 05467F450934BF70
乘积2: 2D306BEFAFF775C8
乘积3: 6B310C0D5F091C20
乘积4: 214FA174B6ECA418
parallels@ubuntu-linux-2404:~$
```

Image 7

```
0546 7F45 0934 BF70;
2D30 6BEF AFF7 75C8;
6B31 0C0D 5F09 1C20;
214F A174 B6EC A418
```

上板进行对照，数据一致：

Image 8

3.2 start-finish 握手协议 10%

> 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。

仿真与顶层结构均由 `start` 信号发起操作请求，模块在操作完成后通过 `finish` 信号给出完成应答。

模块仅在检测到有效 `start` 后才开始工作，外部也仅在 `finish` 有效时读取结果，形成完整的握手时序。

缺点：协议只能单向信息传输，无法知道接收方是否接收。如果接收方繁忙，而发送方未同步该状态，发送方将持续等待finish信号，导致卡死。

可以采用三次握手协议，先询问是否空闲，再改变start，双向状态同步。

3.3 32bit 无符号整数除法器 10%

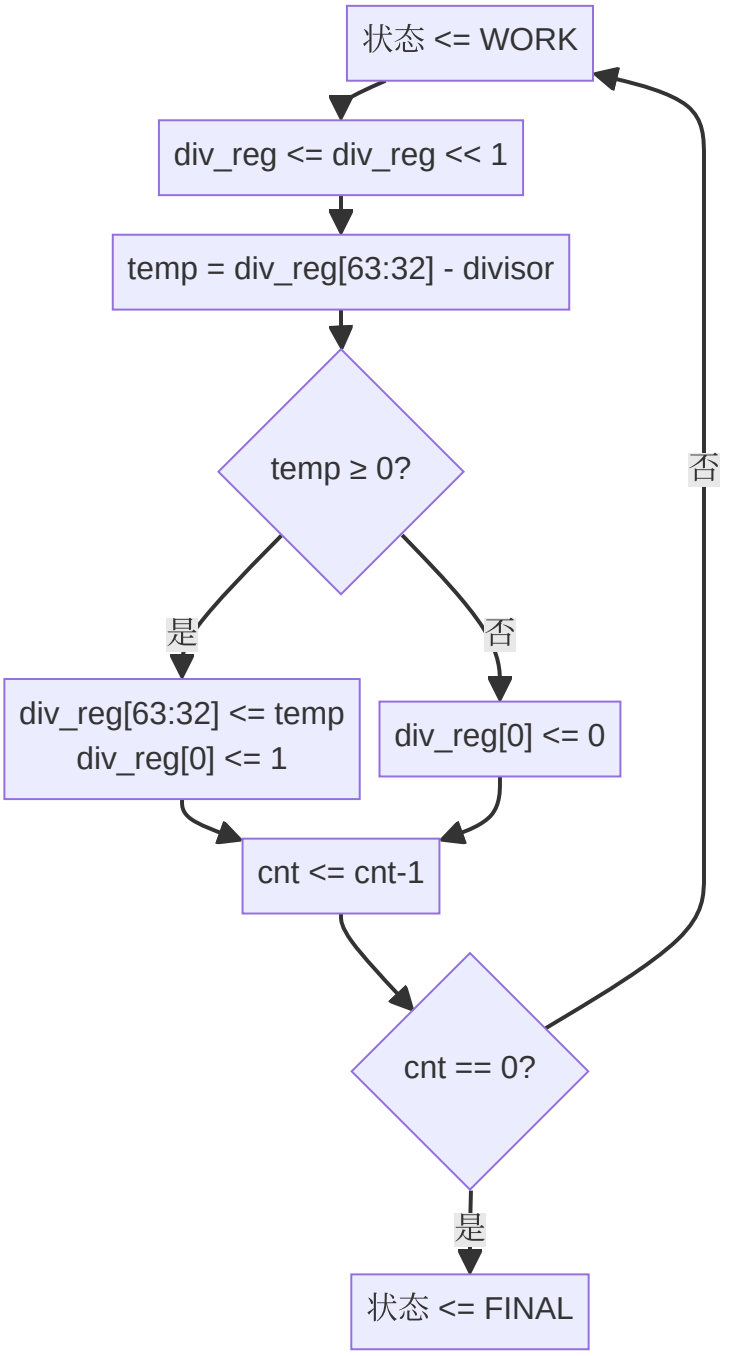
> 请仿照乘法器的设计方法和我们**手动计算除法**的方式，设计**32bit 无符号整数除法器**，你只需要给出设计思路即可。流程图和伪代码是推荐的描述形式。

类似之前的乘法器，定义64位的div_reg，高32位初始化为0，低32位初始化为被除数。

用div_reg的高32位减除数（试探），如果大于等于0执行操作。

最后低32位存放商，高32位存放余数。

这里给出 `WORK` 状态的流程图。



3.4 改进有限状态机 +15%

> (bonus) 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。

单次：可以定义多个Adder，将每次右移的位数变为2或4，以达成速度与门开销的平衡。

连续：pipeline？不太懂。