计算机系统 I 实验报告 3

实验 3-1：时序电路设计之有限状态机

实验 3-2: 计数器 / 定时器设计与应用

实验 3-3: 乘法器

姓 名: 洪 奕 迅

学 号: 3230102930

班 号: 工信 2319

计算机系统 I

(春夏, 2024)

浙江大学

计算机学院

2024 年 4 月 17 日

目录

1 实验 3-1：时序电路设计之有限状态机

1.1 实验过程

1.1.1 代码设计

我使用了 **enum+case 的语法**来实现状态机不同状态之间的跳转，即设置一个遍历数组 *Count* 来遍历所有状态，其中的不同状态对应有限状态机的不同状态：

Code Snippet 1.1.1 ▶ src/lab3-1/submit/FSM.sv

```
typedef enum logic [1:0] {s0,s1,s2,s3} Count;
Count currentState;
```

代码核心部分就是模仿状态机的状态图来实现状态间的跳转，在判断时将每次的输入视为二进制串并按照其值选择对应字母或保持原有状态：

Code Snippet 1.1.2 ▶ src/lab3-1/submit/FSM.sv

```
always@(posedge clk or negedge rstn)begin
    if(~rstn) begin
        currentState<=s0;
    end else begin
        case(currentState)
            s0:case({a,b})
                no:currentState<=currentState;
                b1:currentState<=s0;
                a1:currentState<=s1;
                a2:currentState<=s1;
            endcase
            s1:case({a,b})
                no:currentState<=currentState;
                b1:currentState<=s0;
                a1:currentState<=s2;
                a2:currentState<=s2;
            endcase
```

```
            s2:case({a,b})
                no:currentState<=currentState;
                b1:currentState<=s0;
                a1:currentState<=s3;
                a2:currentState<=s3;
            endcase
            s3:case({a,b})
                no:currentState<=currentState;
                b1:currentState<=s3;
                a1:currentState<=s3;
                a2:currentState<=s3;
            endcase
        endcase
    end
end
assign state=currentState;
```

1.1.2 仿真上板

本次实验的仿真使用了 DPI-C 进行比对，由于是现成文件，因此不需要我们自己写，通过 Vivado 将 1.1.1 中 sv 文件结合所给文件得到比特流，以下为仿真结果：

```
make[1]: Leaving directory '/home/forever/sys1-sp24/src/lab3-1/build/verilate'
cd /home/forever/sys1-sp24/src/lab3-1/build/verilate; ./Testbench
success!!!
- /home/forever/sys1-sp24/src/lab3-1/../../repo/sys-project/lab3-1/sim/testbench
.v:49: Verilog $finish
```

图 1.1: 仿真结果

以下是上板结果，输入序列 $S = abaaab$：

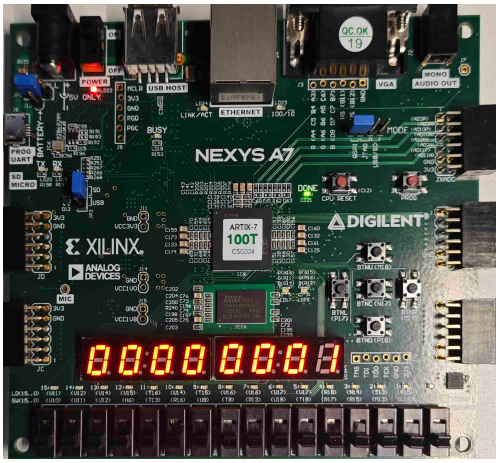

图 1.2: $S = a$

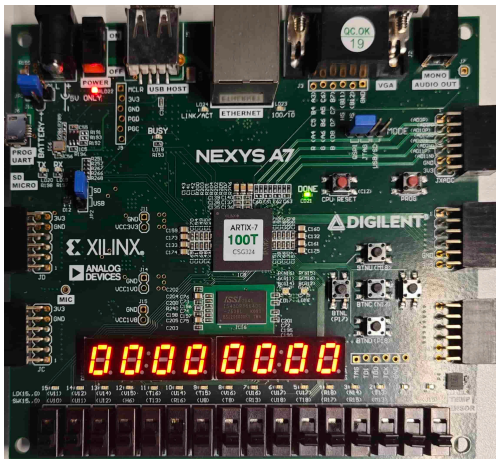

图 1.3: $S = ab$

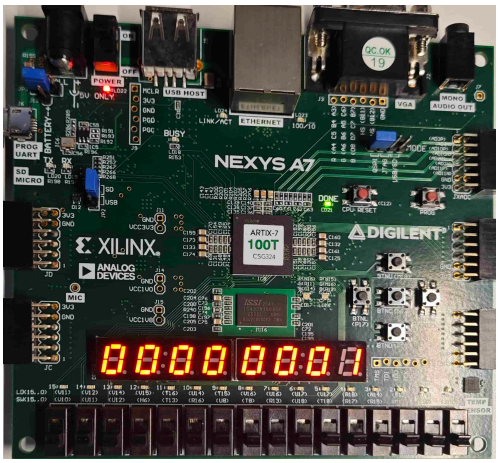

图 1.4: $S = aba$

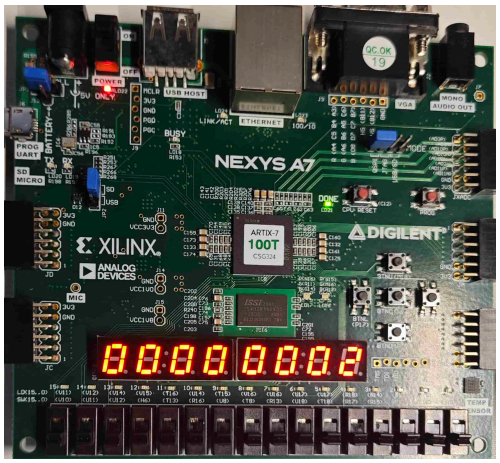

图 1.5: $S = abaa$

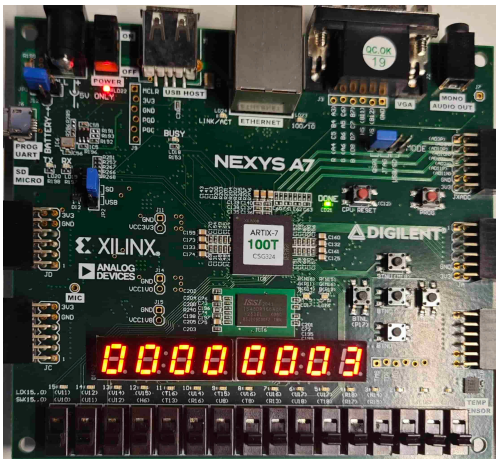

图 1.6: $S = abaaa$

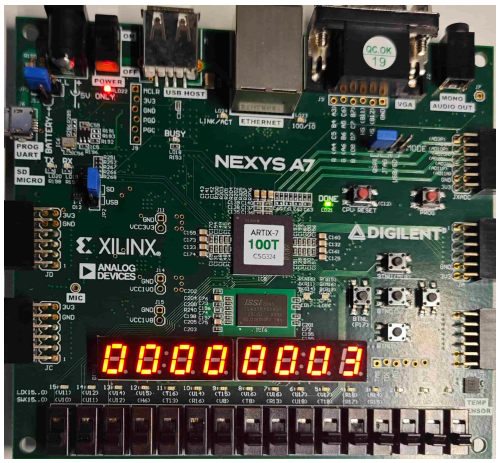

图 1.7: $S = abaaab$

注意状态机功能是是否存在序列 aaa，因此一旦出现后 b 将不再能重置七段管显示的连续出现的 a 的数目，而仿真上板的结果与这一设想相符。

1.2 编程范式比较

1.2.1 enum+case

enum 数组枚举有以下优缺点：

- 具有较强可读性：对于有限状态机而言，核心就是描述不同状态之间的跳转，因此通过枚举各个状态并表示跳转的情况非常符合有限状态机本身的逻辑，可读性强而且易于编写。
- 开销相对小：由于只需要列举需要的状态跳转，在大多数状态机中 enum 数组的 case 数会小于查表中存储的表，而这可以一定程度降低硬件开销。
- 维护性：数组枚举的维护性相对较好，只需要修改逻辑，这建立在可读性强上。
- 语句冗长：由于需要描述跳转的逻辑，enum+case 的语句会相对较冗长。

1.2.2 数组查表

数组查表有以下优缺点：

- 部分情况编写容易：查表法总是列举所有可能的情况，因此在跳转逻辑复杂时仍然可以以一样的方法编写。
- 跳转效率高：查表的跳转关系在表的下标和值中已经被表示，因此不需要复杂的逻辑跳转关系，因此效率更高。
- 可读性差：查表事实上列举了输入和转移的所有可能情况，而使用二进制串进行列举本身几乎丧失了所有的代码可读性。
- 维护性差：每次维护需要修改表，而表中的二进制数据的差可读性会造成维护困难。
- 硬件开销大：由于需要列举所有可能的跳转情况，因此对于 n 输入共需 2^n 长的表，会造成极大的硬件开销。

1.3 状态图/转移表

该状态机在输入为 0 时保持，输入为 1 时计数增加，得到以下状态转移表：

输入	状态	下一状态	输出
1	S0	$S1_1$	/
1	$S1_1$	$S1_2$	/
1	$S1_2$	$S1_3$	/
1	$S1_3$	$S2$	/
0	x	x	/
x	S2	/	1
x	输入结束	/	0

然后由状态转移表得到状态转移图：

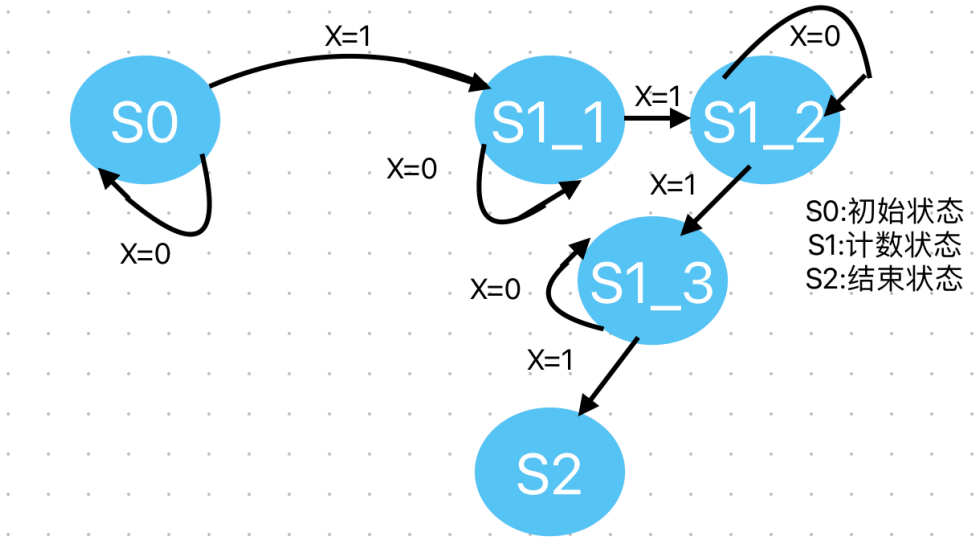

图 1.8: 状态转移图

1.4 Bonus：电路分析

Code Snippet 1.4.1 ▶ bonus

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

1.4.1 不足和原因

这段代码希望实现将向量翻转的功能，这样的代码在仿真中并不存在问题，但在真正的时序电路中可能产生预期之外的结果例如变为了 11，这是因为时序电路的非堵塞操作依赖于寄存器实现，因此操作总是存在时延，不能保证在一个时钟周期内完成目标操作。

1.4.2 电路不稳定结果

在电路不稳定时，这样的翻转可能并不能成功，例如在一个充满毛刺的不稳定的时序下，可能出现在某次翻转中只完成了 $state[0] <= state[1]$，而在这个毛刺以后 *state* 变为了 {11}，此后即使时序稳定，完成的操作仍然是不符合我们的期望的，而绝对稳定的时序显然是很难存在的，因此在一个相对高频或有毛刺的电路中这样的翻转操作极其容易造成数据丢失。

2 实验 3-2: 计数器 / 定时器设计与应用

2.1 实验过程

2.1.1 代码设计

计数器本质实现了一个每按一次按钮向下一个状态跳转的状态机，其中计数状态只需要记录次数而不需要作为多个状态，因此在代码中使用了 if-else 分支结构结合 cnt 计数实现了计数器：

Code Snippet 2.1.1 ▶ src/lab3-2/submit/Cnt.sv

```
always @(posedge clk or posedge high_rst or negedge rstn) begin
    if (~rstn) begin
        cnt <= INITIAL[3:0];
    end else if (high_rst&&en) begin
        cnt <= 0;
    end else if (low_co && (cnt == BASE - 1)&&en) begin
        cnt <= 0;
    end else if (low_co&&en) begin
        cnt <= cnt + 1;
    end
end

assign co = (cnt == BASE - 1) ? 1 : 0;
```

而对于要实现的 24 计数器，由于使用了十进制，因此我们在代码中使用了两个模十计数器，而对于个位只需要根据按键的输入不断增长即可，而对于十位，需要考虑在什么情况下的按钮输入值应该使十位增长？这里就需要考虑逐级计数器的使能信号 low_co，由于报告要求，会在后续再涉及，此处只给出代码，我们发现我们总是把低级的进位信号 co 作为高位的局部使能信号 low_co：

Code Snippet 2.1.2 ▶ src/lab3-2/submit/Cnt2num.sv

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

```
    input en,
    input clk,
    input rstn,
    input high_rst,
    input low_co,
    output co,
    output [7:0] cnt
);

    localparam HIGH_BASE = 10;
    localparam LOW_BASE  = 10;
    localparam HIGH_INIT = INITIAL/10;
    localparam LOW_INIT  = INITIAL%10;
    localparam HIGH_CO   = (BASE-1)/10;
    localparam LOW_CO    = (BASE-1)%10;

    logic co1;
    logic hrst;
    Cnt #(
        .BASE(LOW_BASE),
        .INITIAL(LOW_INIT)
    )cnt_low(
        .en(en),
        .clk(clk),
        .rstn(rstn),
        .low_co(low_co),
        .high_rst(hrst),
        .co(co1),
        .cnt(cnt[3:0])
    );
    logic co2;
    Cnt #(
        .BASE(HIGH_BASE),
        .INITIAL(HIGH_INIT)
    )cnt_high(
        .en(en),
        .clk(clk),
        .rstn(rstn),
```

```
43          .low_co(co1),
44          .high_rst(hrst),
45          .co(co2),
46          .cnt(cnt[7:4])
47      );
48      assign hrst=(cnt[7:4] == HIGH_CO[3:0] && cnt[3:0] == LOW_CO[3:0]) ? 1 : 0;
49      assign co = (cnt[7:4] == HIGH_CO[3:0] && cnt[3:0] == LOW_CO[3:0]) ? 1 : 0;
50
51  endmodule
```

2.1.2 仿真上板

以下为仿真和上板的实验截图和上板图片，该仿真文件遍历了 16-23，0-23 全过程从而检验了计数器的正确性：

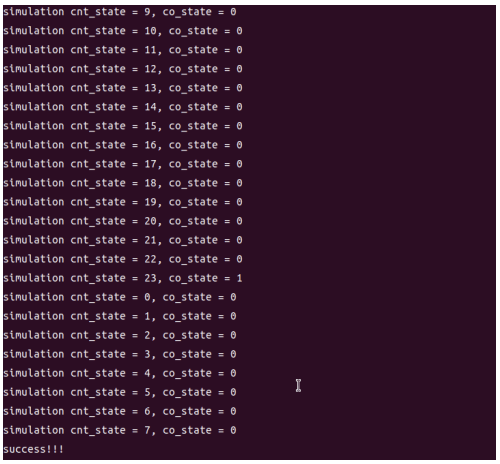

图 2.1: 仿真截图

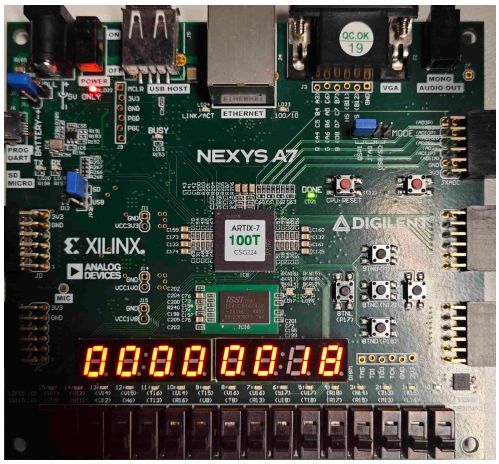

图 2.2: $n = 18$

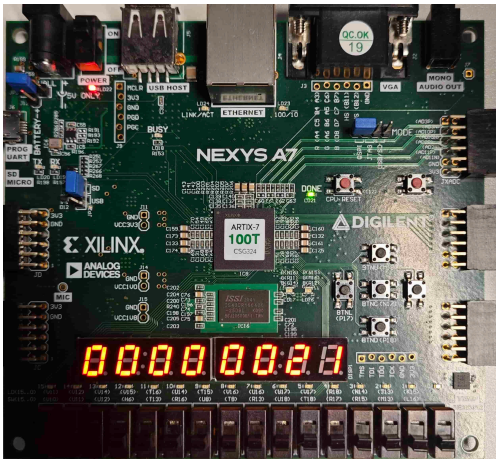

图 2.3: $n = 21$

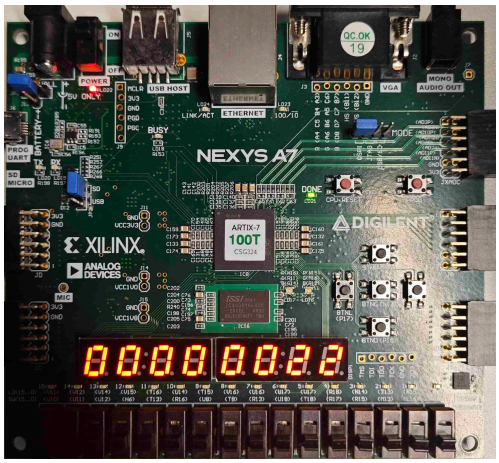

图 2.4: $n = 22$

2.2 1234 计数器实现

Technique 2.2.1 ▶ 如何实现任意 k 进制计数器

对于任意一个十进制数 $\{A_{n-1}...A_1, A_0\}_k$ 都可以被拆分成各个数位，因此最直接的我们使用 n 个模 k 计数器来负责各个数位的统计。

但是并非所有的数位都在每次输入加一，因此需要考虑到我们此前提到的我们应该在什么时候使能某一位的计数器，或者说，我们应该考虑每一位的计数器到底各自计数了几次?

从最低位开始考虑，最低位事实上只需要在每次输入信号的上升沿一直加一即可，其显示范围会由计数器本身的模 n 特性而被有效限制。

但从最低位往上我们发现，越高的的位数需要增长的次数越少，对于第 n 位而言，其每次增长对应的步长为 k^{n-1} ，而计数器只在计数到该级步长次时计数，因此我们只需要在总进行步长次时触发一次该位的使能信号即可。而在实际实现中我们总是使用一种规格的计数器，迭代的讲，对于十进制，该级计数器总是在上级计数器被使能 k 次时触发使能。

现在我们有了一个能计数 $0 - k^N - 1$ 的十进制计数器，接下来只需要引入一个全局复位信号 *high rst* 控制在达到我们希望的计数上限时重置即可。

于是如果我们想要实现 1234 计数器，只需要仿照此前的 24 计数器，将四个 10 进制计数器按位串联，低位进位作为高位使能即可。如何对于复位信号只需要调整为同时判断四位是否满足 1234 条件即可，代码不必赘述。

当然我们也可以使用其他计数器作为 1234 的基，这样从数字角度可以减少总体数字的浪费，但由于数码管本身是十进制的，所以这并不合适。

2.3 预留引脚含义

2.3.1 *co*/*low_co*

从算数角度这两个引脚可以被理解成进位，但是从计数器的实现上来说，使能信号可能更符合他们的实际作用。

在上一节提到了按位划分后的计数器的每一位都需要一个信号来驱动其跟随输入上升而增长，而在这里我们的使能信号就是 *co*/*low_co*。其中 *co* 是该级模 n 计数器完成了 n 次计数后向上一级传递的使能信号，而 *low_co* 则是来自上一级的使能信号，因此我们看到在代码中 *low_co* 被作为计数器是否转入计数状态的条件，也就是在上一节中说到的**计数器只在计数到该级步长次时计数**。

2.3.2 *high_rst*

在上一节其实事实上我们已经讨论到了这个问题，由于模 n 计数器本身的模 n 特性，因此对于上一节我们构建的计数器有天然的 $0 - k^{N-1} - 1$ 的范围限制。因此假设我们希望构建一个模 99 计数器，我们并不需要一个全局复位引脚。

但由于我们希望的上限和进制本身上限不一致，因此我们引入了 *high_rst* 引脚来控制全局的重置，这相当于赋予了这整个计数器模块一个模 n 的限制。

3 实验 3-3: 乘法器

3.1 实验过程

3.1.1 代码设计

乘法器本质是一个反复重复的加法器，因此我们需要设计一个有限状态机，其中的工作状态就是反复按照我们的设计进行加法器操作，而加法器在此处使用了 Verilog 中的加法运算符，这在后续综合中会被综合为一个加法器，以下为乘法器状态机部分的代码：

Code Snippet 3.1.1 ▶ src/lab3-3/Multiplier.sv

```
always @(posedge clk or negedge rst) begin
    if(rst) begin
        fsm_state_reg<=IDLE;
    end else begin
        case(fsm_state_reg)
            IDLE:begin
                finish<=1'b0;
                if(start)begin
                    product_reg={(PRODUCT_LEN+1){1'b0}};
                    multiplicand_reg<=multiplicand;
                    product_reg[CNT_NUM:0]=multiplier;
                    product_reg[PRODUCT_LEN-1:LEN]={LEN{1'b0}};
                    Count<=0;
                    fsm_state_reg<=WORK;
                    high_product_temp={(LEN+1){1'b0}};
                end else begin
                    fsm_state_reg<=IDLE;
                end
            end
            WORK:begin
                if(Count==LEN)begin
                    fsm_state_reg<=FINAL;
                end else begin
```

```
                    if(product_reg[0] & 1)begin
                        high_product_temp={1'b0,product_reg[PRODUCT_LEN-1:LEN]}
                        +{1'b0,multiplicand_reg};
                    end else begin
                        high_product_temp={1'b0,product_reg[PRODUCT_LEN-1:LEN]};
                    end
                    {product_reg[PRODUCT_LEN:LEN],product_reg[CNT_NUM:0]}=
                    {high_product_temp,product_reg[CNT_NUM:0]}>>1;
                    Count<=Count+1;
                end
            end
            FINAL:begin
                product<=product_reg[PRODUCT_LEN-1:0];
                finish<=1'b1;
                fsm_state_reg<=IDLE;
                Count<=0;
            end
            NULL:fsm_state_reg<=IDLE;
        endcase
    end
end
```

3.1.2 仿真上板

仿真阶段没，首先我们需要一个对照来判断我们乘法器代码的正确性，也就是我们使用的 DPI-C 外部库：

Code Snippet 3.1.2 ▶ src/lab3-3/sim/judge.cpp

```
#include <svdpi.h>
#include <stdio.h>

extern "C" unsigned int mul_judge (
    unsigned int multiplicand,
    unsigned int multiplier,
    unsigned long long int product
){

```

```
    unsigned long long int simulate_result=(unsigned long long
     ↪ int)multiplicand*multiplier;
    bool right= simulate_result==product;

    if(!right){
        printf("*********error***********\n");
        printf("simulation multiplicand = %08x, multiplier = %08x, product =
         ↪ %016llx\n", multiplicand, multiplier, simulate_result);
        printf("hardware   multiplicand = %08x, multiplier = %08x, product =
         ↪ %016llx\n", multiplicand, multiplier, product);
    }else{
        printf("simulation multiplicand = %08x, multiplier = %08x, product =
         ↪ %016llx\n", multiplicand, multiplier, simulate_result);
    }

    return right;
}
```

在这段对拍器中我们将 C 算子得到的乘法结果同硬件结果进行比对，来判断乘法器的正确性，而这样一个外部 C 函数，会被以这样的方式引入 Verilog 编写的判断程序中：

Code Snippet 3.1.3 ▶ src/lab3-3/sim/judge.v

```
import "DPI-C" function int unsigned mul_judge(
    input int unsigned multiplicand,
    input int unsigned multiplier,
    input longint unsigned product
);
```

这里值得注意的是 C 和 Verilog 对于变量类型的名称并不相同，因此需要区分。

在得到对拍器后，我们需要编写 *testbench.v* 来实现多组随机样例测试，并且需要满足 start-finish 协议，这里具体的分析在后续章节，但是核心会是对于 @() 语句的使用，来实现最关键的只有在得到回复后才发出下一次启动信号，这里给出部分代码和仿真成功的截图：

Code Snippet 3.1.4 ▶ src/lab3-3/sim/testbench.v

```
initial begin
    clk=0;
    rst=1;
```

```
    #20;
    rst=0;
    for(i=0;i<16;i=i+1)begin
        start=1;
        multiplicand = $random(seed);
        multiplier = $random(seed);
        #10;
        start=0;
        @(posedge finish);
        #25;
        end
    $display("success!!!");
    $finish;
end
```

```
bb220
simulation multiplicand = 000dfa90, multiplier = 001bf520, product = 00000186cfb
b2200
simulation multiplicand = 0037ea40, multiplier = ffefd480, product = 0037e6b7dbb
22000
simulation multiplicand = ffdfa900, multiplier = ffc0adff, product = ffa05efec4f
e5700
simulation multiplicand = ff015bff, multiplier = fe02b7ff, product = fd060e91b51
bec01
simulation multiplicand = fc056fff, multiplier = f80adfff, product = f42ff9392de
fb001
simulation multiplicand = f015bfff, multiplier = e02b7fff, product = d23bd3b04fb
ec001
simulation multiplicand = 3fd6ffff, multiplier = 802dffff, product = 1ff6f8a13ff
b0001
simulation multiplicand = ffdbffff, multiplier = 0037fffe, product = 0037f81e001
00002
simulation multiplicand = ffeffffc, multiplier = ffdfffff8, product = ffd001f4010
00020
simulation multiplicand = 003ffff0, multiplier = fffffe0, product = 003fffeff80
00200
success!!!
```

图 3.1: 仿真成功截图

3.2 start-finish 协议分析

Theorem 3.2.1 ▶ 协议分析

start-finish 协议的主要特点可以被概括成：

1. start 信号与 finish 信号只在发出周期内有效
2. 在得到信号的周期内，数据必须立刻被接发送/接收
3. 信号总是成对出现，即在得到上一个 finish 信号前不会发出下一个 start 信号

3.2.1 满足协议分析

我们首先给出仿真的波形图：

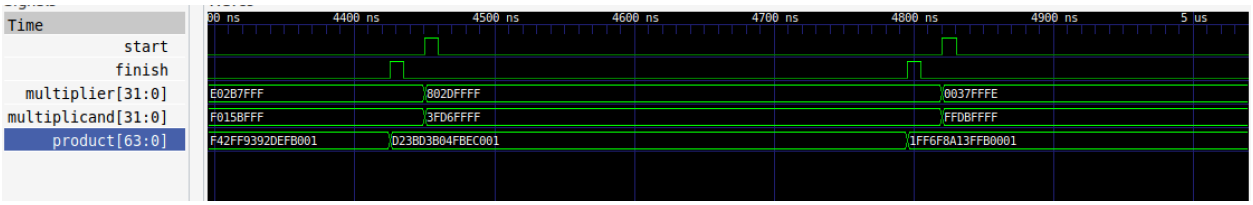

图 3.2: 仿真波形图

可以发现我们的仿真代码满足了协议的所有要求，而这在代码上，通过以下方式实现：

- @(*posedge finish*); 语句保证了握手信号的成对出现
- 在给出一个周期 start 高电平后将其重置以及在收到 finish 信号后等待一段时间，保证有效性和接收即刻性，这里具体的保证方式与乘法器状态机的设计有关，在状态机中开始状态被定义为收到 start 高电平，同时最后预留了超过一个周期时间作为不同计算间的间隔，避免出现数据即刻更新导致的结果与输入错位。

而对于硬件的顶层设计，我们可以发现在 *top.v* 中调用了 *DataDenerator.v*。而后者确保了顶层结构设计与握手协议是符合的，这里需要结合代码进行分析：

Code Snippet 3.2.2 ▶ repo/sys-project/lab3-3/syn/DataGenerator.v

```
always@(posedge clk or negedge rstn)begin
    if(~rstn) start <= 1'b0;
    else if(next_test) start <= 1'b1;
    else start <= 1'b0;
end
```

由于实现中我们并不是和仿真一样循环产生随机样例测试，而是根据按钮产生的指定样例计算，因此不需要考虑自动变化，只需要根据按钮输入产生 start 即可，而对于 finish 和 product 的对齐由我们的乘法器本身保证，这也是这个数据发生器所实现的，在去抖后发现按钮被触发则启动一次 start 传递给乘法器，这样保证了我们的顶层架构满足了这一协议。

3.2.2 协议缺点

start-finish 存在以下缺点：

- 我们注意到 start 必须等待 finish 然后才能启动，这表明了一个问题，即 caller 模块并不知道 callee 的状态，因此只能采用被动等待的方式，这决定了 start-finish 协议下两个使能信号之间只可能有一次完整的操作
- 最大的问题在于 caller 和 callee 是双盲的，互相不知道对方状态，因此我们必须使用在得到 finish 信号后立刻接受结果这样的策略，而此时 caller 的状态是未知的，如果这一接受不有效，start-finish 协议并不尝试处理问题，而是直接进行下一次使能

3.2.3 改进方法

对应以上缺点，我们有以下想要改进的：

- caller 希望 callee 能提供一个信号表明自己是否空闲，这样就不再需要被动等待 callee 完成计算并传输 finish 信号
- callee 希望 caller 提供一个信号表明自己是否空闲，避免结果的传输发生意外而直接进入下一周期

于是可以使用 valid-ready 握手协议：

Definition 3.2.3 ▶ valid-ready 握手协议

***Valid-ready* 协议**希望解决 caller 和 callee 双盲的现状，提供一组信号来表明各自模块的状态，从而使整个实现过程不是被动的受限于时钟周期。这两个信号满足以下规定：

- caller 提供一个 valid 信号，同时 callee 提供一个 ready 信号
- valid=1 代表 caller 准备好了发送数据，反之代表数据没有准备好或者不想发送
- ready=1 代表 callee 准备好了接受数据，反之代表无法或不想接受
- 在 valid=ready=1 时，握手成功，在该周期内进行数据的传输
- 在握手成功的时间内数据传输会被重复进行，因此数据线上的数据应当跟随时钟进行撤换避免重复处理
- 不同于 start-finish 协议中 finish 总是与 start 匹配出现，valid-ready 协议中信号最好相互独立，最起码 ready 信号不能依赖于 valid 信号避免出现 start-finish 协议中 callee 错误导致的死锁

3.3 无符号 32bit 除法器设计

3.3.1 原理分析

我们希望模仿乘法器和我们日常的计算过程来实现除法器：

Definition 3.3.1 ▶ 无符号保留余数除法器分析

无符号有余除法器的原理类似于乘法器的逆过程，即我们可以采取以下方式来得到商和余数：

1. 首先需要准备两个额外寄存器用于存储商和余数
2. 从被除数高位取位宽等于除数的部分与除数比较，若前者不小于后者，向当前位商寄存器载入 1，两者做差后得到初始余数；若前者大与后者，直接将高位作为初始余数
3. 此后每次取余数和被除数剩余高位拼接后与除数比较，参照上一步得到新的余数和商
4. 重复上一步，直至被除数最后一位参与计算，此时的余数寄存器中存储了全程计算的余数

二进制与十进制相比的另一个好处是不再需要判断该位的商，只需要比较大小，因为某位的商只可能是 0/1。

3.3.2 实现方式简述

采用流程图的方式来展现上述算法的实现：

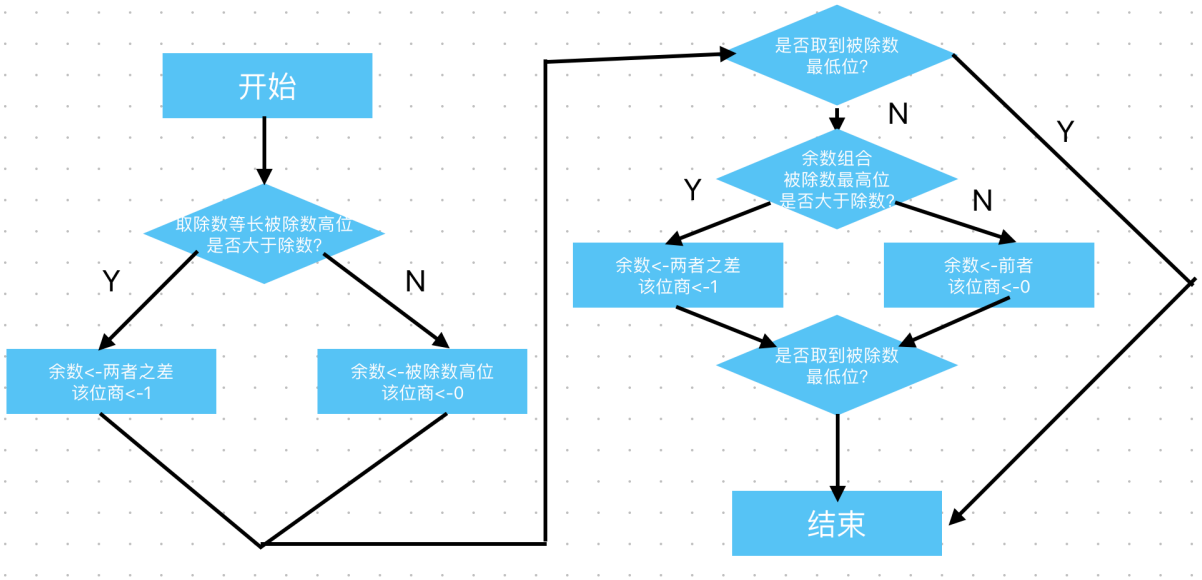

图 3.3: 除法器流程图表示

3.4 Bonus: 乘法器优化

3.4.1 不改变原理减少运算次数

首先应该意识到，我们的乘法器复杂程度事实上与乘数和被乘数位数没有关系，因为我们总是按照寄存器的最大大小来进行多次计算，这符合我们想要的操作的简易性和统一性，但是不符合我们现实中计算的操作，在手动计算时我们总是会忽略前导零，从这一角度出发可以按照这一路径进行优化：

- 首先要统计除数和被除数的实际位数，当然这会占用一定的时钟周期
- 我们发现乘法器的周期次数实际上取决于乘数的有效位数，因此在被乘数位数更小情况下，我们可以将被乘数与乘数进行交换

但必须注意，这两个步骤本身会消耗一定时钟周期，因此适用于乘数或被乘数存在显著短于最大长度时，同时如果确定乘数和被乘数的有效位数，则在连续乘法中，这样的做法会大大减小消耗的时钟周期。

3.4.2 Booth 算数乘法器

而在不改变原理的情况下，我们的改进总是有限的，甚至于在 lab 中的所有样例，上面的优化甚至会造成时间的劣化，因此我们引入一种新的乘法算法进行原理上的优化：

Theorem 3.4.1 ▶ Booth 算数乘法器

Booth 算数乘法器事实上是对于二进制编码的一种改进，减少可能的加法操作，对于基 2-Booth 编码，优化后的乘法器可以减少加法部分积的计算；而对于基 4-Booth 编码，优化后的乘法器只需要原来一般次数的部分积累加操作就可以完成运算。

当然 Booth 编码优化针对的是有符号数，对于我们所做的无符号数自然可以当成正数加上符号位进行操作，接下来来考虑基 2/4-Booth 编码如何加速了乘法运算：

对于一个 N 位有符号数 X，有：

$$X = -2^{N-1}x_{N-1} + 2^{N-2}x_{N-2} \cdots + 2^0 x_0$$

考虑以下变换，为了保证形式的统一，我们规定 $x_{-1} = 0$:

$$X = -2^{N-1}x_{N-1} + 2^{N-2}x_{N-2} \cdots + 2^0 x_0$$

$$= -2^{N-1}x_{N-1} + 2^{N-1}x_{N-2} - 2^{N-2}x_{N-2} \cdot -2^0 x_0 + 2^0 x_{-1}$$

$$= 2^{N-1}(-x_{N-1} + x_{N-2}) + 2^{N-2}(-x_{N-2} + x_{N-3}) \cdot +2^0(-x_0 + x_{-1})$$

于是我们成功得到了基 2 的 Booth 编码，不难发现状态空间 $[x_n, x_{n-1}] = [11, 10, 01, 00]$ 在这种编码下能够减少需要进行加法操作的次数，例如 01111110 = 100000(−1)0 其中-1 为正常意义下的负一，而在补码运算中相反数并不是一个难以处理的问题，考虑此前的乘法器，不难发现现在我们少了很多 1，也就代表乘数寄存器参与运算次数大大减少，而使用 Booth 编码后原来状态空间中的每一种状态都对应一种明确的操作：

- 00/11: 右移一位
- 01: 部分积加乘数补码后右移
- 10: 部分积减乘数补码后右移

第一眼看到 Booth 乘法器感到非常抽象，但事实上这些运算规则的本质是与我们的推导相对应的，我们相当于把原来的每两位作为一个整体重新定义其运算。

当然基 2-Booth 并不能减少部分积累加次数，我们可以参照其推导过程得到基 4-Booth 编码和其对应状态空间 $[x_n, x_{n-1}, x_{n-2}] = \{0-7\}_2$ 所需要进行的操作，即：

$$X = -2^{N-1}x_{N-1} + 2^{N-2}x_{N-2} \cdots + 2^0 x_0$$

$$= 2^{N-2}(-2x_{n-1} + x_{n-2} + x_{n-2}) \cdots + (-2x_1 - x_0 - x_{-1})$$

于是我们也有了对应的操作表，按照这样的操作，可以减少我们的循环次数到原来的一半，并且表格中乘 2 操作也可以通过左移很好的实现：

x_{2i+1}	x_{2i}	x_{2i-1}	PP_i
0	0	0	0
0	0	1	Y
0	1	0	Y
0	1	1	2Y
1	0	0	−2Y
1	0	1	−Y
1	1	0	−Y
1	1	1	0

其中的 Y 代表乘数。

3.4.3 CSA

Definition 3.4.2 ▶ CSA

进位保留加法器 (Carray Save Adder, CSA) 事实上就是一个全加器，能够将三位输入压缩为输出和进位。

不难发现在部分积的累加中又出现了行波进位加法器存在的延迟问题，因此我们希望能够减少这一延迟，而 CSA 能很好实现这一点，对于三组部分积我们可以作为 CSA 的输入从而产生输出和进位，这样的压缩方式能够减少部分积累加的延迟。

当然也可以模仿 CSA 做法实现 4-2 压缩，能力有限，此处略过不谈。