

浙江大学



实验名称: 实验 12: 数字式秒表

实验地点: 东四-223

姓 名: 李昕

学 号: 3230103034

任课教师: 屈民军、唐奕

报告日期: 2025 年 6 月 7 日

第一部分： Lab 10 : 学号滚动显示实验

一、实验目的

1. 掌握用译码器、显示译码器、数据选择器、计数器/分频器等功能模块 Verilog HDL 描述的。
2. 掌握数码管的动态显示驱动方式。
3. 进一步掌握参数定义和参数传递的方法，进一步理解电路层次结构的设计。
4. 掌握具有大分频比的分频器模块的电路仿真。

二、实验要求

设计滚动显示自己学号的电路，要求：

1. 在 Basys3 开发板的 LED 数码管（4 位）显示学号（10 位），向左滚动，0.5s 滚动一位。
2. 在一次学号显示完毕后，插入 3 个“空格”，即相邻两次学号之间显示的三个数码管不亮。

三、实验原理

3.1 LED 数码管动态显示原理

数码管显示一般分静态显示和动态显示两种驱动方式。静态驱动方式的主要特点是，每个数码管都有相互独立的数据线，并且所有的数码管被同时点亮；这种驱动方式的缺点是占用 I/O 口线比较多。动态驱动方式则是所有数码管共用一组数据线（a~g），数码管轮流被点亮。下图为 Basys3 开发板的动态显示数码管的接法，四位 LED 数码显示器为共阳极数码管，由于 LED 数码管采用反相驱动，因此位选信号低电平有效。

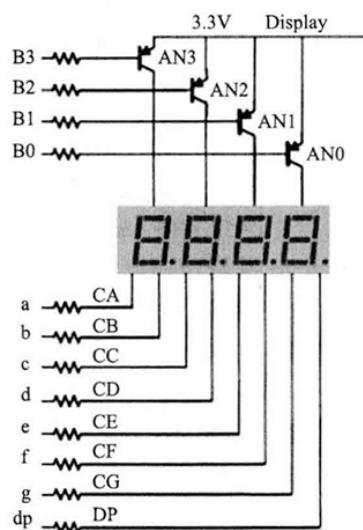


图 1 动态显示数码管的接法

驱动的时序要求如下图所示，**B0**~**B3**轮流输出低电平，依次点亮四位 LED 数码管。同时，要求在相应数码管点亮时输出该位数据的七段笔画码。为了使所有数码管稳定不闪烁地显示，每个数码管必须每隔 5~16 ms 点亮一次（刷新频率 60~200 Hz）。

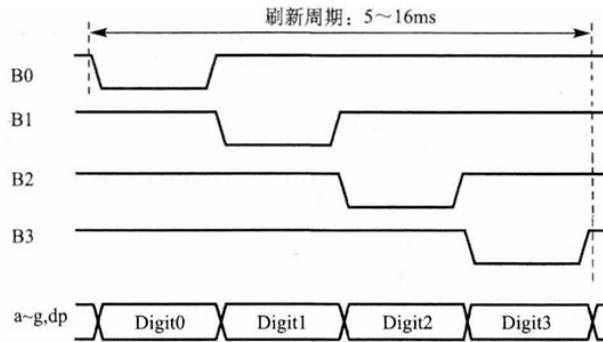


图 2 动态显示驱动的时序图

3.2、电路的总体设计

整个电路由分频器(**counter_n**)、循环移位器(**circ**)和动态显示器(**display**)构成。其原理框图如图所示：

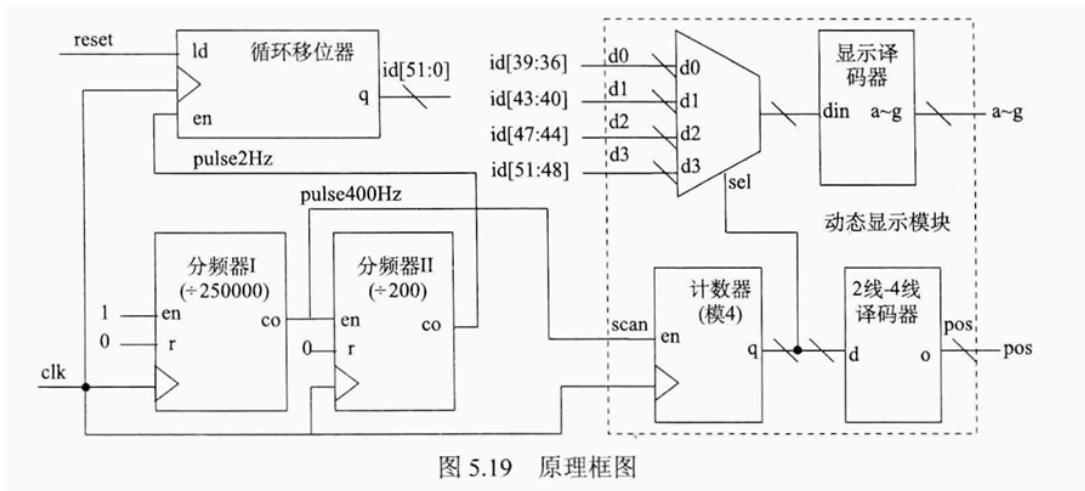


图 3 原理框图

3.3 分频器

由于系统时钟为 100MHz，因此，先由一个分频比为 250000 分频器 I 产生用于动态显示模块的 pulse400Hz 脉冲；再经分频器 II 对 pulse400Hz 进行 200 分频，产生 2Hz 脉冲信号 pulse2Hz，该脉冲信号控制滚动速度。

分频器(**counter_n.v**)主要通过计数器来实现。通过计数输入信号的脉冲数，当计数达到设定值时，输出一个脉冲，并将计数器清零。这样，输出信号的频率就是输入信号频率的 $1/N$ (N 为计数值)，从而实现分频。

3.4 循环移位器

循环移位器主要产生滚动显示数据，下面为其功能表，我的学号为“3230103034”。

| ld | cn | clk | q* | 功能 |
|----|----|-----|-----------------------|----------|
| 1 | × | ↑ | 52'haaa3230103034 | 同步置数 |
| 0 | 1 | ↑ | { q[47:0], q[51:48] } | 循环左移 4 位 |
| 0 | 0 | ↑ | q | 状态保持 |

表 1 循环移位器功能表

其中，同步置数的高三位用了三个 BCD 禁用码，即表示插入三个“空格”。

3.5 动态显示模块

动态显示模块主要由计数器、2 线-4 线译码器、显示译码器组成。四进制计数器状态 q 控制数据选择器依次选出当前显示的 BCD 码送入显示译码器；另外，计数器状态 q 同时表征数据显示在哪个数码管上，通过 2 线-4 线译码器输出位选信号 pos 控制对应的数码管点亮。注意，2 线-4 线译码器输出低电平有效。

七段译码器的功能表如下表所示。输入禁用码时，数码管全灭。

| din[3:0] | {a,b,c,d,e,f,g} | 显示编码 |
|----------|-----------------|------|
| 0000 | 0000001 | 0 |
| 0001 | 1001111 | 1 |
| 0010 | 0010010 | 2 |
| 0011 | 0000110 | 3 |
| 0100 | 1001100 | 4 |
| 0101 | 0100100 | 5 |
| 0110 | 0100000 | 6 |
| 0111 | 0001111 | 7 |
| 1000 | 0000000 | 8 |
| 1001 | 0000100 | 9 |
| 禁用码 | 1111111 | 不显示 |

表 2 数码管显示编码表

四、模块的 Verilog HDL 描述

4.1 顶层模块的 Verilog HDL 描述

StudentID 顶层模块中，输入为系统时钟 `clk` 和复位信号 `reset`，输出为数码管的段选信号 `a-g`、`dp` 以及位选信号 `pos`。模块内部包含两个分频器、一个循环移位器和一个动态显示模块。首先，分频器 1 (`counter_n`) 将 100MHz 的系统时钟分频为 400Hz，作为动态显示模块的扫描脉冲。分频器 2 再次将 400Hz 脉冲分频为 2Hz，作为学号滚动显示的控制脉冲。循环移位器 (`circ`) 在每个 2Hz 脉冲到来时，将学号数据循环左移 4 位，实现学号的滚动显示。动态显示模块 (`display`) 根据当前移位后的学号数据，依次将 4 位 BCD 码送入七段译码器，并控制数码管的显示内容和位置。通过顶层模块的集成，实现了学号在 4 位数码管上的动态滚动显示，并在每次完整显示后插入 3 个空白，实现题目要求的显示效果。

代码如下：

```
module StudentID(
    input clk,
    input reset,
    output a,
    output b,
    output c,
    output d,
    output e,
    output f,
    output g,
    output dp,
    output [3:0] pos
);

parameter sim = 0;//默认仿真

wire pulse400Hz;
wire pulse2Hz;
wire [51:0]id;

//n=4进制分频器
//sim=0, 实际使用代码, 分频比分别是25_0000/200
//sim=1, 测试代码,16进制, 分频比16/4

//分频器1
counter_n #(
    .n(sim?4:250000),
    .counter_bits(sim?2:18)
)
u_counter(
    .clk(clk),
    .en(1),
    .r(0),
    .q(),
    .co(pulse400Hz)
);

//分频器2
counter_n #(
    .n(sim?16:200),
    .counter_bits(sim?4:8)
)
u_counter2(
    .clk(clk),
    .en(pulse400Hz),
    .r(0),
    .q(),
    .co(pulse2Hz)
);

//循环移位器

circ u_circ(
    .ld(reset),
```

```

    .en(pulse2Hz),
    .clk(clk),
    .q(id)
);

//display模块
display u_display(
    .d0(id[39:36]),
    .d1(id[43:40]),
    .d2(id[47:44]),
    .d3(id[51:48]),
    .clk(clk),
    .scan(pulse400Hz),
    .a(a),
    .b(b),
    .c(c),
    .d(d),
    .e(e),
    .f(f),
    .g(g),
    .dp(dp),
    .pos(pos)
);
endmodule

```

4.2 display 模块的 Verilog HDL 描述

`display` 顶层模块用于多位数码管的动态显示控制。输入为 4 个 BCD 码 (`d0` `d3`)、时钟 `clk` 和扫描脉冲 `scan`，输出为数码管的段选信号 `a-g`、`dp` 及位选信号 `pos`。模块内部包含一个 2 位计数器（用于轮流选通 4 个数码管）、2-4 译码器（生成位选信号）、数据选择器（根据计数器状态选择当前显示的 BCD 码）、显示译码器（将 BCD 码转换为七段码）。通过扫描脉冲 `scan` 驱动计数器，实现 4 位数码管的动态轮流显示。

```

module display(
    input [3:0] d0,
    input [3:0] d1,
    input [3:0] d2,
    input [3:0] d3,
    input clk,
    input scan,
    output a,
    output b,
    output c,
    output d,
    output e,
    output f,
    output g,
    output dp,
    output [3:0] pos
);

wire [1:0] q;
//调用计数器

```

```

counter_n #( // 译码器
    .n(4), // 传入参数 n = 4
    .counter_bits(2) // 传入参数 counter_bits = 2
) counter_inst(
    .clk(clk),
    .en(scan),
    .r(r),
    .q(q),
    .co(co)
);

//译码器

decoder decoder_inst(
    .d(q),
    .pos(pos)
);

//数据选择器

wire [3:0] din;

select u_select(
    .sel(q),
    .d0(d0),
    .d1(d1),
    .d2(d2),
    .d3(d3),
    .num(din)
);

//显示译码器

displayDecoder u_displayDecoder(
    .din(din),
    .a(a),
    .b(b),
    .c(c),
    .d(d),
    .e(e),
    .f(f),
    .g(g),
    .dp(dp)
);

endmodule

```

4.3 其他功能模块设计

4.3.1 循环移位器

循环移位器使用时序逻辑电路。在每个时钟上升沿，移位器会根据控制信号决定是进行移位操作还是进行置数操作。

- 移位操作时，寄存器中的数据按预定方向循环移动一位，最后一位的数据会回到第一位，实现循环效果。

- 置数操作时，寄存器会被新的输入数据覆盖。通过这种时序逻辑，循环移位器能够在时钟的同步控制下有序地存储和转移数据。

代码如下：

```
//循环移位器
module circ(
    input ld,
    input en,
    input clk,
    output reg [51:0] q
);
    always @ (posedge clk)
    begin
        if (ld==1)
            q<=52'haaa3230103034;
        else begin
            if (en==1)
                q<={q[47:0],q[51:48]};
            else
                q<=q;
        end
    end
endmodule
```

4.3.2 显示译码器

`displayDecoder` 的 `always` 块和 `case` 语句用于将 4 位 BCD 输入码转换为七段数码管的显示编码。使用了 `always @(*)` 代表组合逻辑，输入 `din` 变化就会触发逻辑，`case(din)` 根据不同输入值输出对应的七段码。这种结构可以高效地实现数字到七段显示的译码功能，输入为禁用码时，所有段均熄灭。

```
module displayDecoder(
    input [3:0] din,
    output reg a,
    output reg b,
    output reg c,
    output reg d,
    output reg e,
    output reg f,
    output reg g,
    output dp
);
    assign dp = 1;
    always @(*) begin
        case(din)
            4'b0000: {a,b,c,d,e,f,g} = 7'b0000001; // 显示数字 1
            4'b0001: {a,b,c,d,e,f,g} = 7'b1001111; // 显示数字 2
            4'b0010: {a,b,c,d,e,f,g} = 7'b0010010; // 显示数字 3
            4'b0011: {a,b,c,d,e,f,g} = 7'b0000110; // 显示数字 4
            4'b0100: {a,b,c,d,e,f,g} = 7'b1001100; // 显示数字 5
            4'b0101: {a,b,c,d,e,f,g} = 7'b0100100; // 显示数字 6
            4'b0110: {a,b,c,d,e,f,g} = 7'b0100000; // 显示数字 7
            4'b0111: {a,b,c,d,e,f,g} = 7'b0001111; // 显示数字 8
            4'b1000: {a,b,c,d,e,f,g} = 7'b0000000; // 显示数字 9
        endcase
    end
endmodule
```

```

        4'b1001: {a,b,c,d,e,f,g} = 7'b0000100; // 显示数字 0
        default: {a,b,c,d,e,f,g} = 7'b1111111; // 默认情况下熄灭所有段
    endcase
end
endmodule

```

4.3.3 2 线-4 线译码器

`decoder` 的逻辑与 `displayDecoder` 类似，都是使用组合逻辑，用 `always @(*)` 和 `case` 语句根据输入译码输出对应信号。

```

module decoder(
    input [1:0] d,
    output reg [3:0] pos
);
//2线-4线译码器
    always @(*) begin
        case(d)
            2'b00: pos = 4'b1110;
            2'b01: pos = 4'b1101;
            2'b10: pos = 4'b1011;
            2'b11: pos = 4'b0111;
        endcase
    end
endmodule

```

4.3.4 数据选择器

```

module select(
    input [1:0] sel,
    input [3:0] d0,
    input [3:0] d1,
    input [3:0] d2,
    input [3:0] d3,
    output reg [3:0] num
);

    always @(*) begin
        case(sel)
            2'b00: num = d0;
            2'b01: num = d1;
            2'b10: num = d2;
            2'b11: num = d3;
        endcase
    end
endmodule

```

4.3.5 n 进制分频器(计数器)

n 进制分频器使用了 lab1 中的设计。参数 `sim` 用于控制分频器的工作模式：

- 当 `sim=0` 时，分频器按实际硬件需求设置分频比（如 250000/200），用于板上运行；
- 当 `sim=1` 时，分频器采用较小的分频比（如 4/16），便于仿真测试，加快仿真速度。

```

module counter_n(clk,en,r,q,co);
    parameter n=4;
    parameter counter_bits=2;
    input clk,en,r ;

```

```

output co;
output [counter_bits-1:0] q;
reg [counter_bits-1:0] q=0;
assign co=(q==(n-1)) && en;
always @(posedge clk)
begin
    if(r) q=0;
    else if(en)
        begin
            if(q==(n-1)) q=0 ;
            else q=q+1;
        end
    end
endmodule

```

五、模块测试与分析

5.1 counter_n计数器测试

传入 `n=8, counter_bits=4` 进行测试：



图 4 counter_n 计数器 - 1

实验表明，计数器能够正确地从 0 计数到 7，计数到 7 时 `co=1` 并持续一个时钟周期，表明计数器功能正常。

检测复位信号：令 `r=1`，检测到计数器复位为 0，说明复位功能正常。

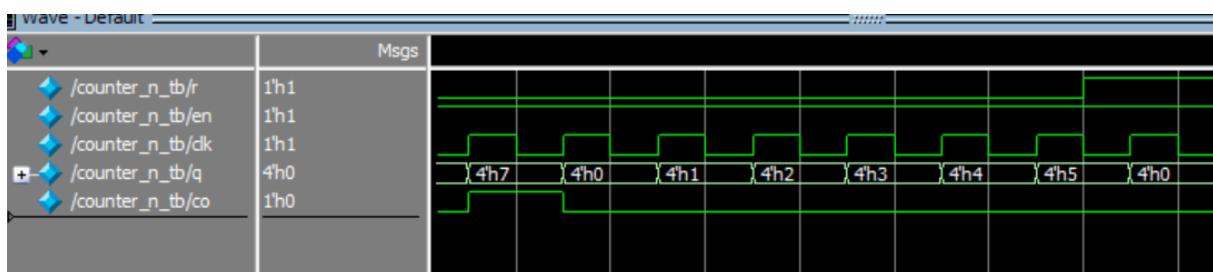


图 5 counter_n 计数器 - 2

5.2 circ 循环移位器测试

自主编写如下 `testbench` 进行测试：

```

module circ_tb;
parameter delay=10;
//declaration of signals
reg ld,en,clk;
wire [51:0] q; //output
//instantiation of the design
circ uut(.ld(ld),.en(en),.clk(clk),.q(q));

```

```

//this process sets up the free running clock
initial begin
    clk=0;
    forever #(dely) clk = ~clk;
end //clk周期是dely, 占空比50%
initial begin
    ld=0;
    en=0;
    #dely; //时钟稳定
    //测试en=1, 加载功能
    ld=1;
    #(2*dely);
    ld=0;
    //en=1, 循环移位
    en=1;
    #(4*dely);
    //测试en=0, 装填保持
    en=0;
    #(2*dely);
    $finish;//结束仿真
end
//监视波形:
initial begin
    $monitor("Time = %0t, q = %h, q_next = %h, en = %b, clk = %b", $time, q, en, ld, clk);
end
endmodule

```

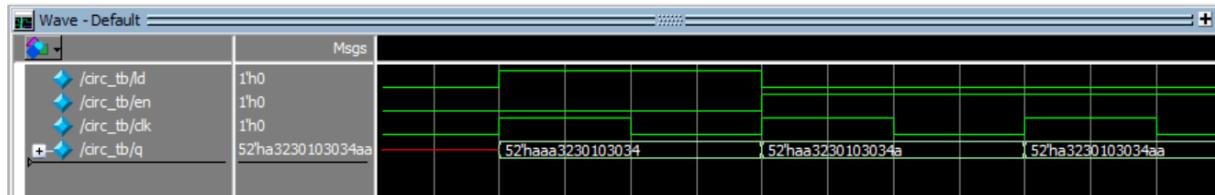


图 6 counter_n 计数器 - 2

仿真波形表示，循环移位器在 `[en=1]` 时能够实现学号数据的循环左移，每个时钟周期数据左移 4 位； `[ld=1]` 时数据被正确置数， `[en]` 信号切换时移位与保持功能均正常。

display 模块测试

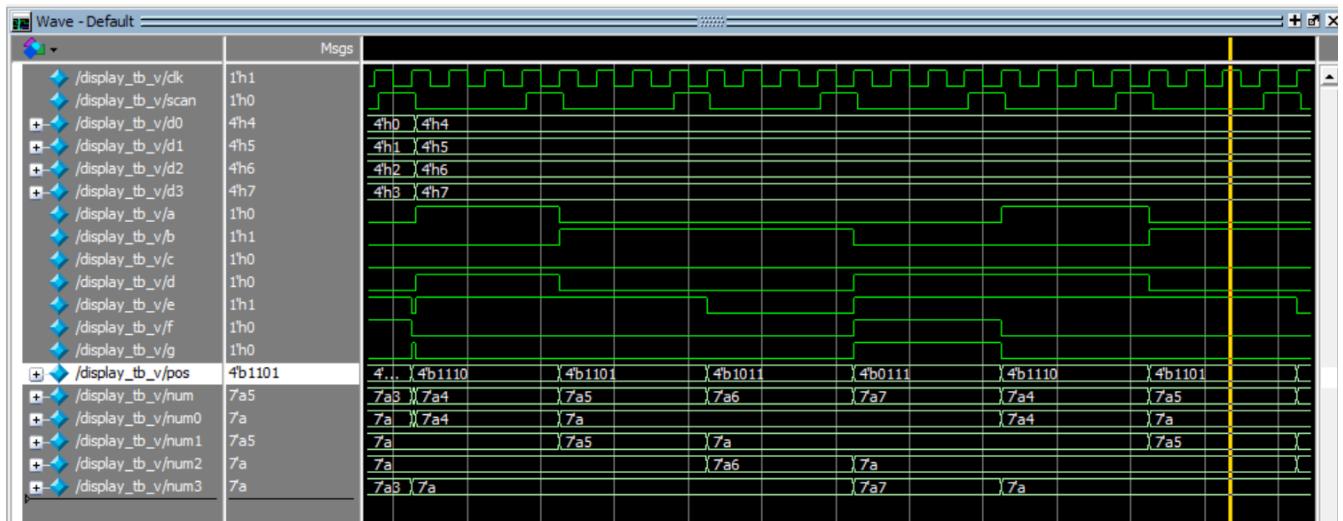


图 7 display 模块测试 - 1

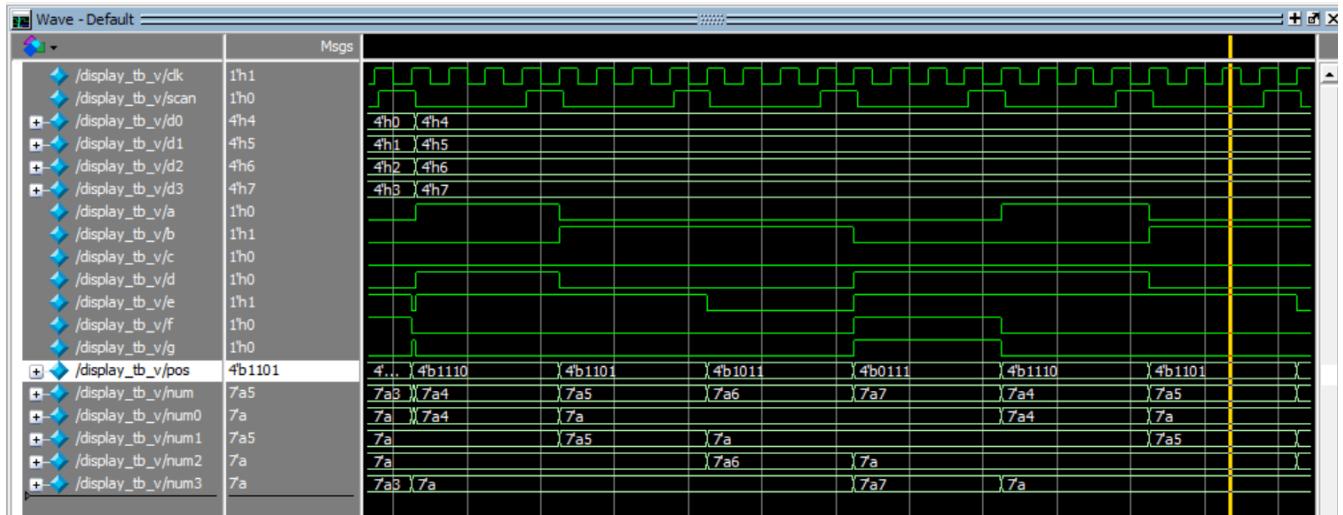


图 8 display 模块测试 - 2

使用 `display_tb_v` 对 `display` 模块进行了功能仿真，分别向 `d0` `d3` 输入不同的 BCD 码组合，配合 `scan` 信号模拟动态扫描过程。波形表明：

- 在 `scan` 信号有效时，译码器 `pos` 轮流选通 4 个数码管，实现动态显示；
- 数据选择器根据 `q` 的状态，依次将 `d0` `d3` 的值送入显示译码器，七段码输出与输入数据一致；
- 在仿真过程中，`num` 信号会根据 `pos` 的状态选择对应的输入数据，实现了数据选择器的功能；
- 七段码输出 `a` `g` 会根据当前 `num` 的值进行译码，正确显示对应的数字；

5.3 SchoolID 模块测试

使用 `SchoolID_tb_v` 对 `SchoolID` 模块进行了功能仿真，波形如下：

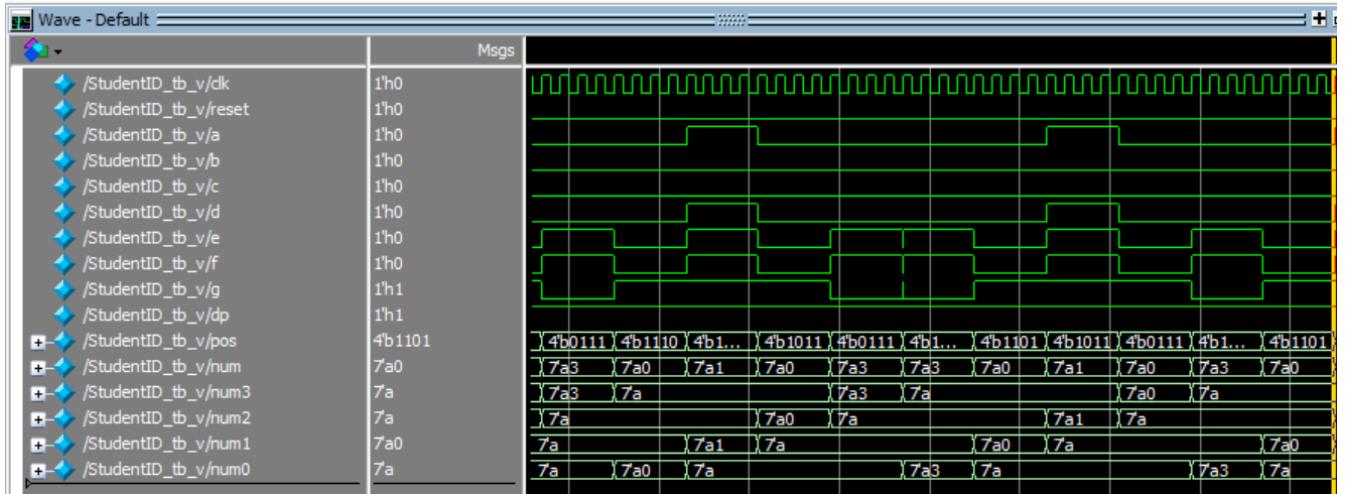


图 9 SchoolID 模块测试

波形表明，学号进行滚动显示，并且 `num` 按照 `pos` 依次显示对应的学号数字，说明 `SchoolID` 模块功能正常。

六、引脚约束

在 Vivado 软件内设置引脚约束如下表：

| 引脚名称 | I/O | 引脚编号 | 接口类型 | 说明 |
|--------|--------|------|---------|------------------|
| clk | Input | W5 | LVCMS33 | 系统 100MHz 主时钟 |
| reset | Input | U18 | LVCMS33 | 中间按钮 |
| a | Output | W7 | LVCMS33 | 七段码 |
| b | Output | W6 | LVCMS33 | 七段码 |
| c | Output | U8 | LVCMS33 | 七段码 |
| d | Output | V8 | LVCMS33 | 七段码 |
| e | Output | U5 | LVCMS33 | 七段码 |
| f | Output | V5 | LVCMS33 | 七段码 |
| g | Output | U7 | LVCMS33 | 七段码 |
| pos[0] | Output | U2 | LVCMS33 | 四个数码管的位选信号 |
| pos[1] | Output | U4 | LVCMS33 | 四个数码管的位选信号 |
| pos[2] | Output | V4 | LVCMS33 | 四个数码管的位选信号 |
| pos[3] | Output | W4 | LVCMS33 | 四个数码管的位选信号 |
| dp | Output | V7 | LVCMS33 | 可对 dp 置高电平来隐匿小数点 |

七、主要仪器设备

- 电脑（Modelsim、Vivado 软件）
- Basys3 开发板

八、上板实现

将 Verilog 代码综合、实现后，上板测试效果。

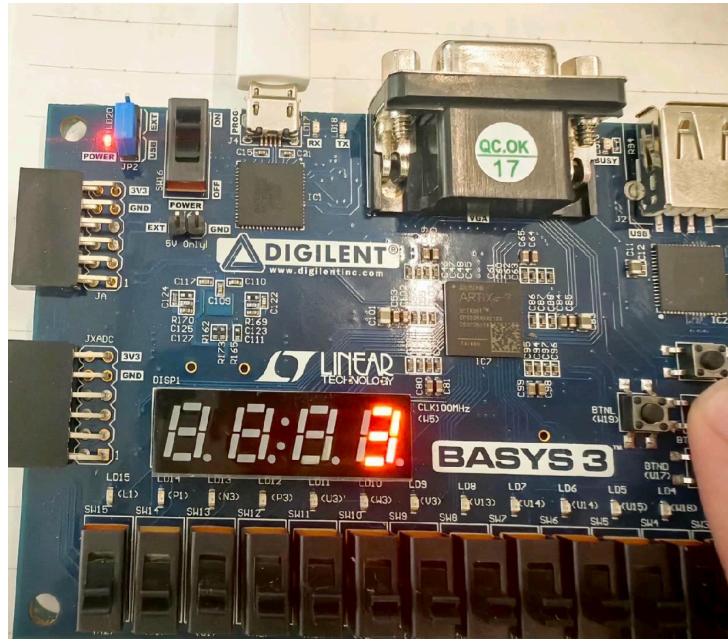


图 10 上板测试

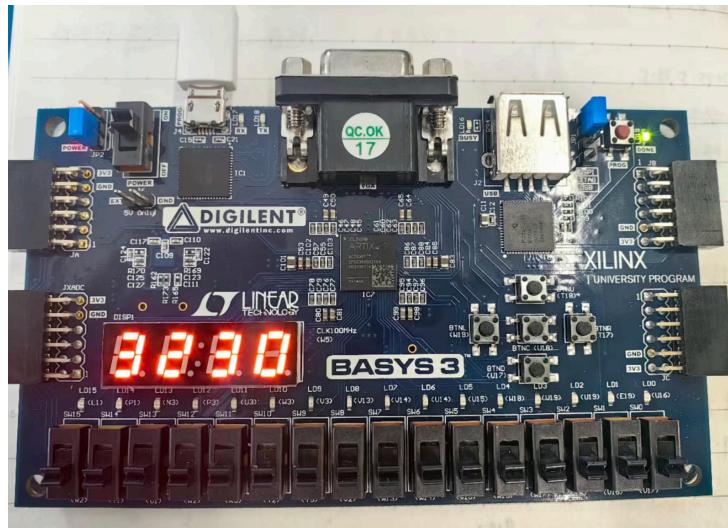


图 11 上板测试

上板后，学号依次滚动，实现了预期效果，本次实验完成。

九、思考题

Q: 动态显示有什么优缺点?

A:

优点:

1. **I/O 资源节约:** 动态驱动方式下，所有数码管共用一组数据线(a~g)，大幅减少了所需的 I/O 端口数量。对于多位数码管显示，静态驱动需要的 I/O 口数量为 $n \times 7$ ，而动态驱动仅需 $n+7$ 。
2. **硬件成本降低:**
 - 所需的驱动电路和连接线更少，降低了系统设计成本
 - 单个 FPGA 或微控制器可以驱动更多位数的数码管

3. 灵活性高:

- 便于实现字符滚动显示等动态效果
- 可以针对不同数码管位置显示不同内容

缺点:

1. 亮度降低:

- 由于每个数码管只在时间的一小部分被点亮，总体亮度低于静态驱动
- 驱动多位数码管时，亮度会随位数增加而降低($1/n$ 占空比)

2. 控制复杂度增加:

- 需要额外的控制逻辑来实现动态扫描
- 需要分频电路、计数器、译码器等附加电路

第二部分：Lab11：异步输入的同步器和开关防颤动电路设计

一、实验目的

1. 掌握减少亚稳态的方法，了解亚稳态给系统带来的危害。
2. 掌握开关颤动的概念和消除的方法。
3. 初步了解控制器的设计。

二、实验要求

1. 设计一个按键处理模块，要求每按一次按键，按键处理模块输出一个正脉冲信号，脉冲的宽度为一个 `c1k` 时钟周期。
2. 按照要求搭建一个按键处理模块测试电路，下载到开发板验证，即每按键一次，`led` 指示灯更改亮灭状态。

三、实验原理

1. 同步器的设计

在异步设计中，完全避免亚稳态是不可能的。因此，设计的基本思路应该是：首先尽可能减少出现亚稳态的可能性，其次是尽可能减少亚稳态给系统带来危害的可能性。如图 5.22 所示，采用双锁存器法，即将输入异步信号用两个锁存器连续锁存两次，理论研究表明这种设计可以将出现亚稳态的概率降低到一个很小的程度，但这种方法同时带来了对输入信号的延时，在设计时需要加以注意。

随着技术发展，工作时钟的周期越来越小，有时两级锁存还不能够解决亚稳态问题，因此目前在高速数字电路中会采用 3 级甚至 4 级锁存器级联来减少出现亚稳态的概率。

当异步脉冲宽度小于时钟周期时，采用图(a)所示电路可能无法采样到输入信号，这时应该采用图(b)所示的电路。由于异步输入 `asynch_in` 到来时间的不确定性，第 2 级也有可能出现亚稳态，可由第 3 级锁存器避免亚稳态。另外，该电路的输出 `synch_out` 的宽度为两个时钟周期。

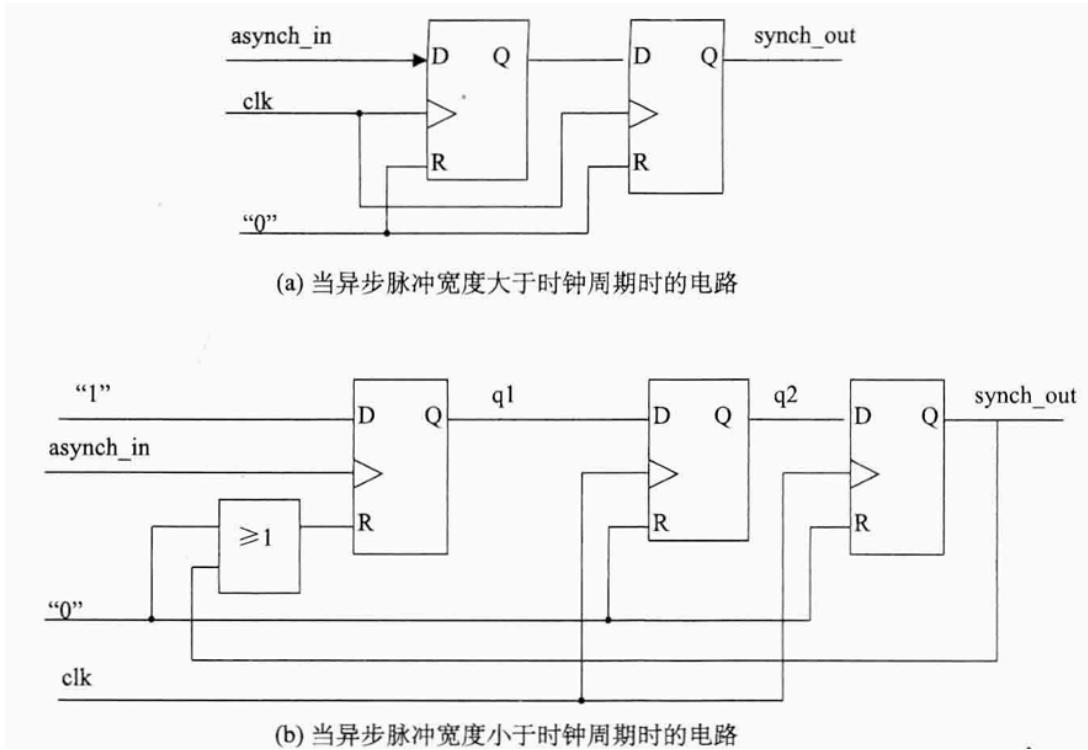


图 12 同步器的两种类型

2. 开关防颤动电路的设计

1. 开关的颤动及开关防颤动电路的功能

人们完成一次按键操作的指压力如图(b) 所示，按键开关从最初按下到接触稳定要经过数毫秒的颤动，按键松开时也有同样问题。因此，按键被按下或释放时，都有几毫秒的不稳定输出，从逻辑电平角度来看不稳定输出期间，其电平在“0”“1”之间无规则摆动。因此，一次按键操作的输出如(c)所示。按键操作时间 t_a 因人而异，一般开关 $t_a < 100ms$ 。

开关防颤动电路的目的是：按一次按键，输出一个稳定脉冲，即将开关的输出作为开关防颤动电路的输入，而开关防颤动电路的输出为理想输出，如下图(d)所示。

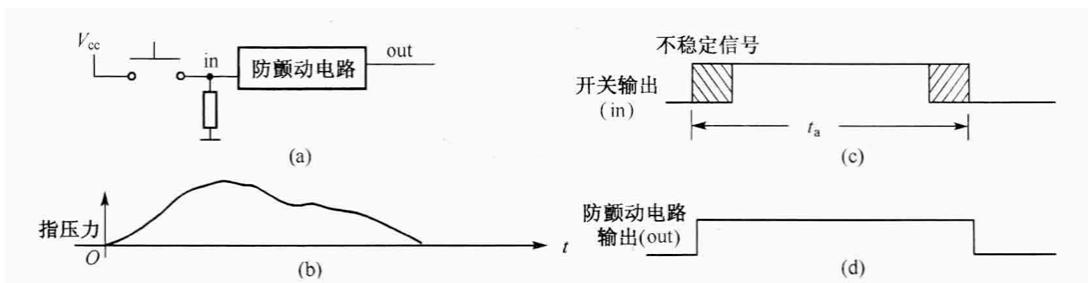


图 13 按键开关的颤动

2. 开关防颤动电路的设计

开关防颤动电路的关键是避免在颤动期采样。颤动期时间长短与开关类型及按键指压力有关，一般为 10ms 左右。据此可画出防颤动电路的原理框图，如下图所示。开发板输入时钟 `clk` 的频率为 100MHz， 10^5 分频器将产生周期为 1ms 的脉冲信号 `pulse1kHz`，`pulse1kHz` 的宽度为一个时钟 `clk` 周期，即 10ns。脉冲信号 `pulse1kHz` 用作定时器的基

准, **10ms** 定时器是用来定时颤动期, 启动信号 **timer_clr** 由控制器提供, 定时结束信号 **timer_done** 反馈给控制器。

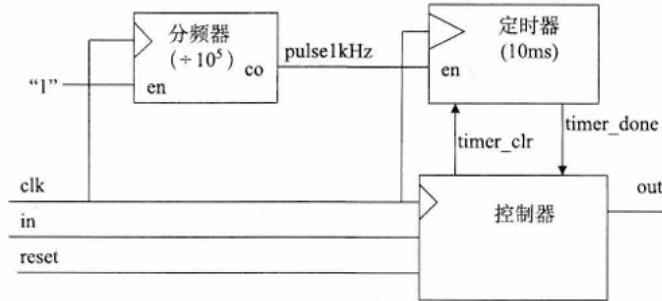


图 14 防颤动电路的原理框图

controller 是防颤动电路的核心, 综合开关防颤动原理可画出 **controller** 的算法流程图, 如下图所示。一般情况下电路在 **LOW** 状态下等待: 当按键按下 (检测到 **in=1**), 电路转到 **WAIT_HIGH** 状态, 启动 10ms 定时器; 当 10ms 定时结束 (**timer_done=1**), 电路进入 **HIGH** 状态, 采样开关输入, 等待按键释放; 当按键释放 (检测到 **in=0**), 电路转到 **WAIT_LOW** 状态, 启动 10ms 定时器, 消除按键释放颤动期; 定时结束后回到 **LOW** 状态下等待下次按键操作。

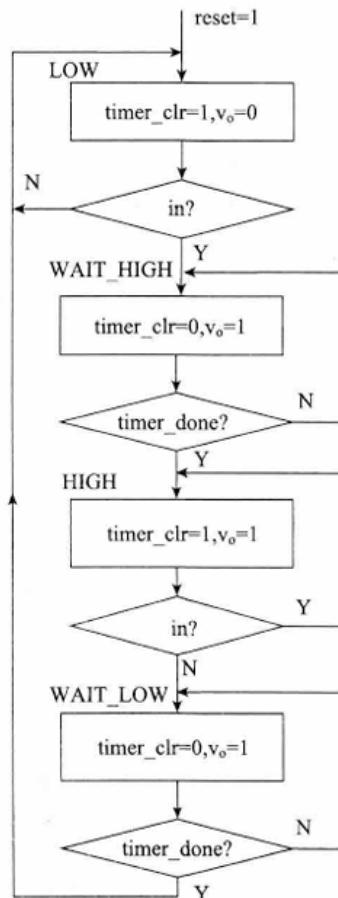


图 15 控制器的算法流程图

3. 脉宽变换电路的设计

从上面分析得知，开关或按键的输入信号宽度远大于一个时钟周期，脉宽变换电路的作用是将输入信号宽度转换成一个时钟周期。由于输入已是同步的且宽度大于一个时钟的脉冲，因此脉宽变换电路的设计就变得非常简单，下图所示为其原理图。

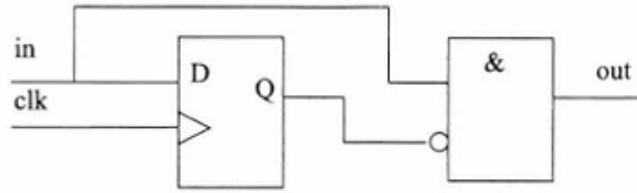


图 16 脉宽变换电路

四、模块的 Verilog HDL 描述

4.1 顶层模块的 Verilog HDL 描述

button_unit 顶层模块中，输入为系统时钟 `clk`、复位信号 `reset` 和按键输入信号 `ButtonIn`，输出为处理后的按键脉冲信号 `ButtonOut`。模块内部包含一个同步器、一个防颤电路和一个脉冲变换器。首先，同步器 (`sync`) 将异步的按键输入信号与系统时钟同步，输出同步后的信号 `sync_out`，避免亚稳态问题。防颤电路 (`defibrillate`) 接收同步后的信号，通过内部定时器产生 1kHz 脉冲进行采样，消除按键机械抖动，输出稳定的按键信号 `out`。脉冲变换器 (`pulsewidth`) 将防颤后的电平信号转换为单周期脉冲信号 `ButtonOut`，确保每次按键操作只产生一个有效脉冲。通过顶层模块的集成，实现了对机械按键信号的完整处理，从异步输入到同步化、去抖动再到脉冲输出，为数字系统提供可靠的按键控制信号。

代码如下：

```

//top module
module button_unit(
    input reset,
    input ButtonIn,
    input clk,
    output ButtonOut
);
    parameter sim=0;
    //同步器
    wire sync_out;
    sync unit_sync(
        .clk(clk),
        .sync_in(ButtonIn),
        .sync_out(sync_out)
    );
    //防颤电路
    wire out;
    wire timer_clr;
    wire timer_done;
    wire pulse1kHz;
    defibrillate #(
        .sim(0)

```

```

)
u_defibrillate(
    .clk(clk),
    .in(sync_out),
    .reset(reset),
    .out(out),
    .timer_clr(timer_clr),
    .timer_done(timer_done),
    .pulse1kHz(pulse1kHz)
);
//脉冲变换
// wire ButtonOut;
pulsewidth u_pulsewidth(
    .clk(clk),
    .in(out),
    .out(ButtonOut)
);
endmodule

```

4.2 各功能模块的 Verilog HDL 描述

4.2.1 同步器模块

同步器用于将异步输入信号与系统时钟同步，减少亚稳态的概率。采用两级 D 触发器级联实现，这种设计可以显著降低亚稳态发生的概率，提高系统稳定性。

```

module sync(
    input clk,
    input sync_in,
    output reg sync_out
);
    reg s1;
    always @ (posedge clk) begin
        s1 <= sync_in; // 第一级D触发器
        sync_out <= s1; // 第二级D触发器
    end
endmodule

```

在同步器中，异步输入信号 sync_in 首先被第一级 D 触发器采样，产生中间信号 s1，然后 s1 再被第二级 D 触发器采样，最终生成同步输出信号 sync_out。这种双触发器级联结构能够有效减少亚稳态传播到后续电路的可能性。

4.2.2 防颤动电路

防颤动电路由分频器、计数器和控制器组成，用于消除机械按键产生的抖动，输出稳定的按键信号。

防颤动电路集成了三个主要组件：分频器将 100MHz 时钟分频为 1kHz，用于按键状态采样；控制器实现了状态机逻辑，管理按键状态转换；定时器负责计算 10ms 的消抖时间。参数 sim 用于控制分频比，便于仿真测试。

```

module defibrillate(
    input clk,
    input in,
    input reset,
    output out,

```

```

    output timer_done,
    output pulse1kHz,
    output timer_clr
);

parameter sim=0;

//分频器1 - 产生1kHz脉冲信号
counter_n #(
    .n(sim?10:100000),
    .counter_bits(sim?4:17)
)
u_counter(
    .clk(clk),
    .en(1),
    .r(0),
    .q(),
    .co(pulse1kHz)
);

//控制器 - 负责状态转换与防抖
control ctrl(
    .clk(clk),
    .in(in),
    .reset(reset),
    .timer_clr(timer_clr),
    .timer_done(timer_done),
    .out(out)
);

//定时器 - 用于计时10ms消抖时间
count_time #(
    .n(10),
    .counter_bits(4))
u_count_time(
    .clk(clk),
    .en(pulse1kHz),
    .r(timer_clr),
    .done(timer_done)
);
endmodule

```

防颤动电路集成了三个主要组件：分频器将 100MHz 时钟分频为 1kHz，用于按键状态采样；控制器实现了状态机逻辑，管理按键状态转换；定时器负责计算 10ms 的消抖时间。参数 sim 用于控制分频比，便于仿真测试。

4.2.3 控制器模块

控制器是防颤动电路的核心，实现了状态机逻辑，根据输入信号和定时器状态控制整个按键处理流程。

```

module control(
    input clk,
    input reset,
    input in,
    input timer_done,

```

```

    output timer_clr,
    output out
);
//采用三段式状态机实现

//状态编码
parameter LOW = 2'b00;      // 低电平稳定状态
parameter WAIT_HIGH = 2'b01; // 等待高电平稳定
parameter HIGH = 2'b10;      // 高电平稳定状态
parameter WAIT_LOW = 2'b11; // 等待低电平稳定

//状态变量
reg [1:0] st_next; // 下一状态
reg [1:0] st_cur; // 当前状态

//输出寄存器
reg r_clr; // 定时器清零信号
reg r_v; // 输出信号

//1. 状态转移逻辑
always @(posedge clk or posedge reset) begin
    if(reset)
        st_cur <= LOW;
    else
        st_cur <= st_next;
end

//2. 状态切换逻辑
always @(*) begin
    // 默认保持当前状态
    st_next = st_cur;

    case(st_cur)
        LOW: begin
            case(in)
                1'b0: st_next = LOW;
                1'b1: st_next = WAIT_HIGH;
                default: st_next = LOW;
            endcase
        end
        WAIT_HIGH: begin
            case(timer_done)
                1'b0: st_next = WAIT_HIGH;
                1'b1: st_next = HIGH;
                default: st_next = WAIT_HIGH;
            endcase
        end
        HIGH: begin
            case(in)
                1'b1: st_next = HIGH;
                1'b0: st_next = WAIT_LOW;
                default: st_next = HIGH;
            endcase
        end
        WAIT_LOW: begin

```

```

        case(timer_done)
            1'b0: st_next = WAIT_LOW;
            1'b1: st_next = LOW;
            default: st_next = WAIT_LOW;
        endcase
    end
    default: st_next = LOW;
endcase
end

//3. 输出逻辑
always @(posedge clk or posedge reset) begin
    if(reset) begin
        r_clr <= 1;
        r_v <= 0;
    end
    else begin
        case(st_cur)
            LOW: begin
                r_clr <= 1; // 定时器清零
                r_v <= 0; // 输出低电平
            end
            WAIT_HIGH: begin
                r_clr <= 0; // 定时器开始计时
                r_v <= 0; // 输出保持低电平
            end
            HIGH: begin
                r_clr <= 1; // 定时器清零
                r_v <= 1; // 输出高电平
            end
            WAIT_LOW: begin
                r_clr <= 0; // 定时器开始计时
                r_v <= 1; // 输出保持高电平
            end
            default: begin
                r_clr <= 1;
                r_v <= 0;
            end
        endcase
    end
end

//输出连接
assign timer_clr = r_clr;
assign out = r_v;
endmodule

```

控制器采用三段式状态机设计，包括状态寄存器、组合逻辑状态转换和输出逻辑。状态机有四个状态：LOW（按键未按下）、WAIT_HIGH（等待按键稳定）、HIGH（按键已按下）和WAIT_LOW（等待按键释放稳定）。状态转换由输入信号 in 和定时器完成信号 timer_done 控制，输出包括定时器清零信号 timer_clr 和处理后的按键信号 out。

4.2.4 计时器模块

计时器模块接收 1kHz 脉冲信号，计数至 10ms 用于消抖。

计时器基于计数器实现，当输入使能信号 en 有效时进行计数，达到设定值 n 后输出完成信号 done。r 信号用于清零计数器和输出信号，通常由控制器的 timer_clr 驱动。

```

module count_time #(
    parameter n = 10,                      // 默认计数到10
    parameter counter_bits = 4              // 默认4位计数器
)()
(
    input clk,
    input en,   // 使能信号, 通常接1kHz脉冲
    input r,    // 复位信号
    output done // 计时完成信号
);
reg [counter_bits-1:0] count = 0;
reg done_reg = 0;

always @(posedge clk) begin
    if(r) begin
        // 复位逻辑
        count <= 0;
        done_reg <= 0;
    end
    else if(en) begin
        // 计数逻辑
        if(count == n-1) begin
            count <= count; // 计数到达上限后保持
            done_reg <= 1;   // 输出完成信号
        end
        else begin
            count <= count + 1; // 继续计数
            done_reg <= 0;      // 未完成
        end
    end
end

assign done = done_reg;
endmodule

```

4.2.5 脉冲变换模块

脉冲变换电路用于将防颤后的按键信号转换为单周期脉冲，便于数字系统处理。

脉冲变换器使用 D 触发器检测输入信号的上升沿，当检测到按键信号从 0 变为 1 时，生成一个持续一个时钟周期的脉冲。

```

module pulsewidth(
    input clk,
    input in,
    output out
);
wire q;
// 使用D触发器实现边沿检测
dffre #(n(1)) ff(
    .d(in),
    .en(1'b1),
    .r(1'b0),
    .clk(clk),

```

```

    .q(q)
);

// 检测上升沿：当前输入为1且前一周期为0时产生脉冲
assign out = in & (~q);
endmodule

```

4.2.6 D 触发器模块

为支持脉冲变换器并进行测试，实现了具有使能和复位功能的 D 触发器模块。

D 触发器模块支持参数化位宽，具有同步使能和异步复位功能，为系统提供基础的时序存储单元。

```

module dffre #( 
    parameter n = 1 // 触发器位宽
) (
    input [n-1:0] d,
    input en,
    input r,
    input clk,
    output reg [n-1:0] q
);
    always @(posedge clk) begin
        if(r)
            q <= 0;
        else if(en)
            q <= d;
    end
endmodule

```

五、模块测试与分析

5.1 同步器测试

使用自主编写的 testbench 对同步器进行功能验证：

```

module sync_tb;
    reg clk;
    reg sync_in;
    wire sync_out;

    sync uut(
        .clk(clk),
        .sync_in(sync_in),
        .sync_out(sync_out)
    );

    // 时钟生成
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    // 测试序列
    initial begin
        sync_in = 0;

```

```

#15 sync_in = 1;
#7 sync_in = 0;
#3 sync_in = 1;
#2 sync_in = 0;
#10 sync_in = 1;
#20 $finish;
end
endmodule

```

使用 Modelsim 仿真，波形如下：



图 17 同步器测试波形

测试结果表明，当输入 `sync_in` 后，时钟上升沿时将输出一个时钟周期的信号。因此，同步器成功将异步输入信号与时钟同步，输出信号 `sync_out` 只在时钟上升沿变化，有效减少了亚稳态风险。

5.2 状态机控制器测试

状态机控制器是整个按键处理电路的核心，测试重点关注其状态转换和输出行为。使用自行编写的测试代码进行测试：

```

module control_tb;
parameter delay = 10;
//declaration of signals
reg clk;
reg reset;
reg in;
reg timer_done;
//output
wire timer_clr;
wire out;

//instantiation of the design
control uut(
    .clk(clk),
    .reset(reset),
    .in(in),
    .timer_done(timer_done),
    .timer_clr(timer_clr),
    .out(out)
);
//initial clk

initial begin
    clk=0;
    forever #(delay) clk = ~clk;
end //clk周期是delay, 占空比50%

initial begin

```

```

//初始化，并等待一段时间
clk=0;
reset=0;
in=0;
timer_done=0;
 #(4*dely);
//reset=1, LOW->LOW
reset=1;
in=0;
#(2*dely);
reset=0;
//LOW->WAIT-HIGH
in=1;
#(2*dely);
//WAIT-HIGH->WAIT-HIGH
timer_done=0;
#(2*dely);
//WAIT-HIGH->HIGH
timer_done=1;
#(2*dely);
//HIGH->HIGH
in=1;
#(2*dely);
//HIGH->WAIT-LOW
in=0;
#(2*dely);
//WAIT-LOW->WAIT-LOW
timer_done=0;
#(2*dely);
//WAIT-LOW->LOW
timer_done=1;
#(2*dely);
$finish;
end

initial begin
$monitor("Time = %0t, clk = %b, reset = %b, in = %b, timer_done = %b, timer_clr = %b,
out = %b", $time, clk, reset, in, timer_done, timer_clr, out);
end

endmodule

```

利用测试代码，测试得到的状态机波形如下：

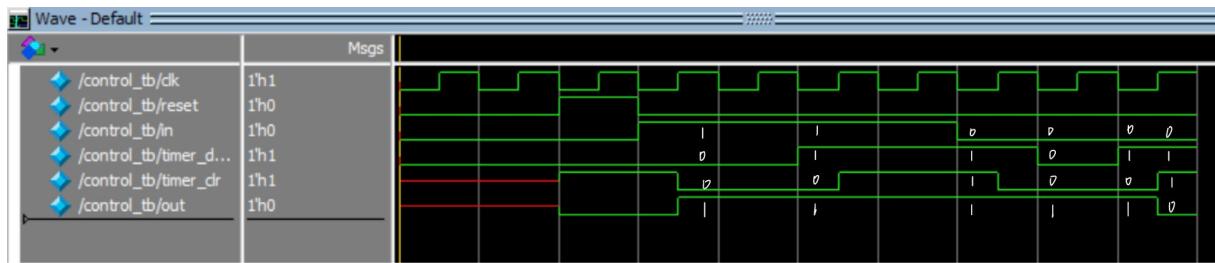


图 18 控制器状态机测试波形

控制台输出的信号如下：

```

#
run -all
...
# Time = 30, clk = 1, reset = 0, in = 0, timer_done = 0, timer_clr = x, out = x
# Time = 40, clk = 0, reset = 1, in = 0, timer_done = 0, timer_clr = 1, out = 0
# Time = 50, clk = 1, reset = 1, in = 0, timer_done = 0, timer_clr = 1, out = 0
# Time = 60, clk = 0, reset = 0, in = 1, timer_done = 0, timer_clr = 1, out = 0
# Time = 70, clk = 1, reset = 0, in = 1, timer_done = 0, timer_clr = 0, out = 1
# Time = 80, clk = 0, reset = 0, in = 1, timer_done = 0, timer_clr = 0, out = 1
# Time = 90, clk = 1, reset = 0, in = 1, timer_done = 0, timer_clr = 0, out = 1
# Time = 100, clk = 0, reset = 0, in = 1, timer_done = 1, timer_clr = 0, out = 1
# Time = 110, clk = 1, reset = 0, in = 1, timer_done = 1, timer_clr = 1, out = 1
# Time = 120, clk = 0, reset = 0, in = 1, timer_done = 1, timer_clr = 1, out = 1
# Time = 130, clk = 1, reset = 0, in = 1, timer_done = 1, timer_clr = 1, out = 1
# Time = 140, clk = 0, reset = 0, in = 0, timer_done = 1, timer_clr = 1, out = 1
# Time = 150, clk = 1, reset = 0, in = 0, timer_done = 1, timer_clr = 0, out = 1
# Time = 160, clk = 0, reset = 0, in = 0, timer_done = 0, timer_clr = 0, out = 1
# Time = 170, clk = 1, reset = 0, in = 0, timer_done = 0, timer_clr = 0, out = 1
# Time = 180, clk = 0, reset = 0, in = 0, timer_done = 1, timer_clr = 0, out = 1
# Time = 190, clk = 1, reset = 0, in = 0, timer_done = 1, timer_clr = 1, out = 0
# * Note: $finish      : D:/MyRepository/SystemDesignLab/3230103034/
lab11_button_process_unit/src/control_tb.v(62)
#     Time: 200 ns  Iteration: 0  Instance: /control_tb

```

波形显示，控制器状态机正确实现了四状态转换机制：

- 低电平稳定状态(`LOW`)检测到按键按下后，转入等待高电平稳定状态(`WAIT_HIGH`)
- 计时器完成 10ms 计时后，进入高电平稳定状态(`HIGH`)
- 检测到按键释放后，转入等待低电平稳定状态(`WAIT_LOW`)
- 计时器再次完成 10ms 计时后，回到低电平稳定状态(`LOW`)
- 状态机输出信号 `r_clr` 和 `r_v` 随状态变化正确生成，实现了预期的按键去抖动功能。

5.3 计时模块测试

为了验证该模块的功能，编写的测试代码如下，进行了以下测试：

1. **分频测试：**使用 `counter_n` 模块生成模拟的 1kHz 脉冲信号 `pulse1kHz`，在仿真模式下通过参数 `sim=1` 将分频比设置为较小的值(10)，加快仿真速度
2. **计时功能验证：** `count_time` 模块接收 `pulse1kHz` 作为使能信号，计数至 10 后生成 `done` 信号，测试其计时精度和复位功能
3. **重置功能测试：** 通过控制 `time_clr` 信号，验证计时器在收到复位信号后能否正确清零并重启计时

```

module timer_tb;// output declaration of module defibrillate

// module timer(clk,en,r,q,co);
parameter dely = 10;
reg clk;
reg en;
// wire [3:0] q;
wire timer_done;

wire pulse1kHz;

```

```

//wire timer_clr;
reg time_clr;
// wire timer_done;
parameter sim=1;
//分频器1
counter_n #(
    .n(sim?10:100000),
    .counter_bits(sim?4:17)
)
u_counter(
    .clk(clk),
    .en(1),
    .r(0),
    .q(),
    .co(pulse1kHz)
);

count_time #(
    .n(10),
    .counter_bits(4))
u_count_time(
    .clk(clk),
    .en(pulse1kHz),
    .r(time_clr),
    // .q(q),
    .done(timer_done)
);

//this process sets up the free running clock
initial begin
    clk=0;
    forever #(dely) clk = ~clk;
end //clk周期是dely, 占空比50%

initial begin
    //初始化
    time_clr=0;
    en=0;
    #(5*dely);

    //复位
    time_clr=1;
    #(2*dely);
    time_clr=0;
    #(5*dely);
    //下面观察变化
    #(200*dely);
    $finish;
end

initial begin
    $monitor("Time = %0t, clk = %b, en = %b, r = %b, co = %b", $time, clk, en, time_clr,
    timer_done);
end
endmodule

```

测试得到的波形如下：

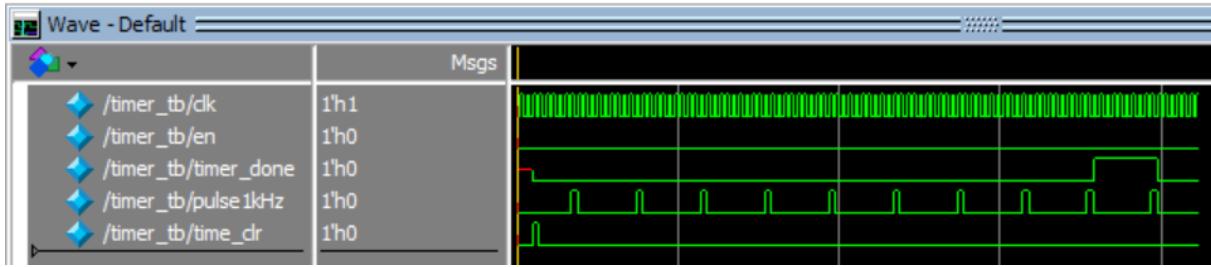


图 19 控制器状态机测试波形

观察波形可以发现，`timer_clr`时开始计时，每隔 10 个时钟周期，`pulse_1kHz`输出一次信号，每隔 10 个`pulse_1kHz`信号输出一次计时器完成`timer_done`信号，说明电路功能正常。

5.4 脉冲变换模块测试

自主编写脉冲变换模块测试代码如下；输入一定长度的信号，观察输出。

```
module pulsedwidth_tb;
parameter dely = 10;
//port name
reg in;
reg clk;
wire out;
//模块名称
pulsedwidth u_pulsedwidth(
    .clk(clk),
    .in(in),
    .out(out)
);

initial begin
    clk=0;
    forever #(dely) clk = ~clk;
end //clk周期是dely, 占空比50%

initial begin
    in=0;
    //初始化
    #(5*dely);
    //输入宽度
    in=1;
    #(5*dely);
    in=0;
    #(10*dely);
    $finish;
end

initial begin
    $monitor("Time = %0t, clk = %b, in = %b, out = %b", $time, clk, in, out);
end

endmodule
```

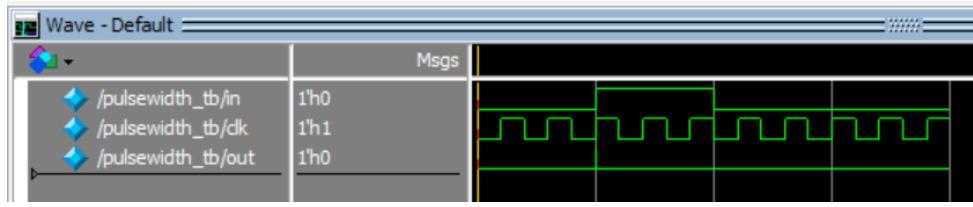


图 20 脉冲变换模块测试波形

波形表明，脉冲变换电路在输入信号上升沿将信号变换为了一个上升脉冲信号，说明电路功能正常。

5.5 整体按键处理单元测试

使用所给的测试代码，完整的按键处理单元集成测试结果如下。其中，为了将脉宽信号变换为能使 Led 亮起的信号，`ButtonOut`后接了一个`dffre`触发器级联实现。

```
button_unit #(.sim(1)) button_unit(
    .clk(clk),
    .reset(reset),
    .ButtonIn(ButtonIn),
    .ButtonOut(ButtonOut)
);
dffre #(.n(1)) d(
    .d(ButtonOut),
    .clk(clk),
    .r(0),
    .q(o),
    .en(1)
);
```

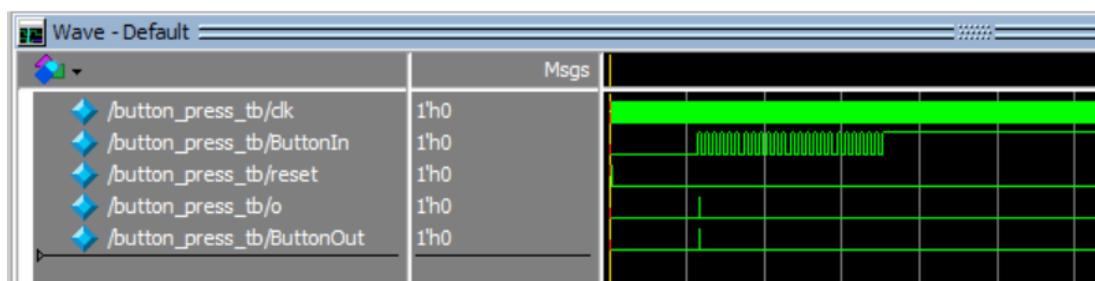


图 21 整体按键处理单元测试波形

测试结果表明，当输入抖动的时候，输出信号`o`只会输出一次正信号，这说明防抖模块的功能正常。

六、引脚约束

按照下表在 Vivado 内为 Basys 3 设置引脚约束。

| 引脚名称 | I/O | 引脚编号 (Basys 3) | 电气特性 | 说明 |
|----------|--------|----------------|----------|---------------|
| clk | Input | W5 | LVCMOS33 | 系统 100MHz 主时钟 |
| reset | Input | U18 | LVCMOS33 | 开发板上的中间按键 |
| ButtonIn | Input | T18 | LVCMOS33 | 开发板上的上边按键 |
| led | Output | U16 | LVCMOS33 | LED0 指示灯 |

表 3 FPGA 引脚约束内容

七、主要仪器设备

- 电脑 (Modelsim、Vivado 软件)
- Basys3 开发板

八、上板验证

代码烧录后，按下上面按键，按下时指示灯亮起。且按键抖动不妨碍功能实现，说明本次实验正常。



图 22 按键处理上板验证

九、思考题

1. 同步器中的锁存器级数与什么因素有关？

同步器中锁存器的级数主要与系统时钟频率、亚稳态允许概率和输入信号的异步性有关。时钟频率越高、对亚稳态的容忍度越低时，需要更多级锁存器以降低亚稳态传播概率。

2. 在本实验中，如果对 UP 开关信号不作脉宽变换处理，那么将会造成什么结果？

如果不对 UP 开关信号进行脉宽变换，宽度大于一个时钟周期的信号会被计数器多次采样，导致一次按键可能被识别为多次触发，产生多次计数或误操作。

3. 为什么对系统复位信号 reset 不需要进行防颤动处理？

系统复位信号通常由电路或专用复位电路产生，具有明确的时序和稳定性，不会像机械按键那样产生抖动，因此不需要防颤动处理。

第三部分：Lab 12：数字式秒表实现（包含个性化要求）

一、实验目的

1. 掌握计数器的功能和应用。
2. 理解开关防颤动的必要性。
3. 掌握简单控制器的设计方法。

二、实验任务

1. 基本要求

设计一个数字秒表电路，具体要求如下：

- **计时范围：** $0'0.0'' \sim 9'59.9''$ （分辨率 0.1 秒），使用数码管显示计时值。
- **功能按键（ButtonIn）控制逻辑：**
 - 初始状态：第一次按下按键 → 开始自动计时。
 - 第二次按下 → 停止计时。
 - 第三次按下 → 计数器复位至 $0'0.0''$ ，回到初始状态。

2. 个性化要求（双计时功能）

设计支持记录甲、乙两物体运动时间的秒表（假设甲先到达终点），功能逻辑如下：

- **初始状态：** 第一次按下按键 → 开始自动计时。
- **甲到达终点：** 第二次按下 → 保存甲的计时值，但内部继续为乙累积计时。
- **乙到达终点：** 第三次按下 → 停止计时，显示乙的计时值。
- **第四次按下 → 显示之前保存的甲的计时值。**
- **第五次按下 → 计数器复位至 $0'0.0''$ ，回到初始状态。**

三、实验原理

3.1 顶层设计

秒表电路由分频模块、按键处理模块、控制器、计时器和动态显示模块组成。其中，分频模块、按键处理模块和动态显示模块在之前的实验中都已实现过。本实验主要需要额外实现控制器、计时器模块。

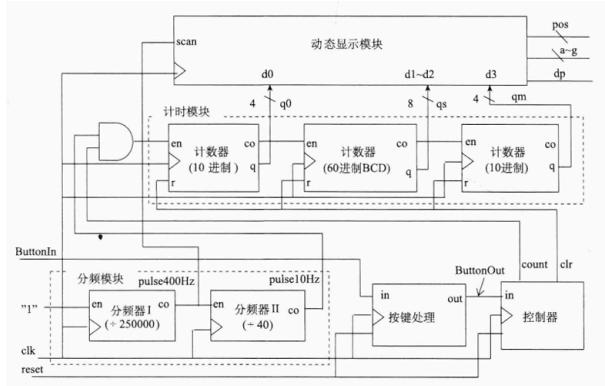


图 23 秒表电路的原理框图

- 分频器模块 产生其他模块所需的脉冲信号，其中输出的 `pulse400Hz` 信号是频率为 400Hz、宽度为一个 `c1k` 周期的脉冲信号，该信号用于 动态显示扫描模块。而 `pulse10Hz` 信号是频率为 10Hz、宽度为一个 `c1k` 周期的脉冲信号，该信号为 秒表 的计时基准信号。
- 按键处理模块 完成按键输入的 同步器、开关防颤动 和 脉冲宽度变换 等功能，即当按键一次，输出一个宽度为 `c1k` 周期的脉冲信号 `Buttonout`，该模块的设计方法参见实验 11。
- 计时模块 可由 1 个十进制计数器（十分之一秒计时）、1 个六十进制 BCD 码 秒计数器 和 1 个十进制 分计数器 级联而成。
- 显示采用数码管动态显示技术，设计原理参见实验 10。与实验 10 所不同，本实验应在第 2 位显示小数点，所示 `dp` 可由 `dp=pos[1]` 赋值。

3.2 控制器的原理

控制器是电路的核心，在按键控制下，输出 `clr` 和 `count` 两个信号分别控制计时模块的工作状态。控制器的算法流程如下图所示，`RESET`、`TIMING`、`STOP` 分别代表秒表的初始、计时和停止三个状态，相应地控制计数器模块清零、计数和保持三种功能。

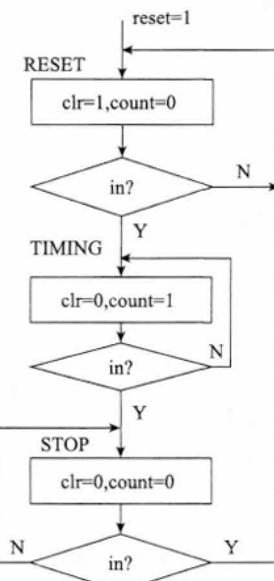


图 24 控制器的 ASM 图

四、模块的 Verilog HDL 描述

4.1 顶层模块的 Verilog HDL 描述

`stopwatch` 秒表顶层模块中，输入为系统时钟 `clk`、按键输入 `ButtonIn` 和复位信号 `reset`，输出为七段数码管的段选信号 `a-g`、`dp` 和位选信号 `pos`。

模块内部集成了分频器、按键处理、控制器、计时器和显示模块。首先，两级分频器将 100MHz 系统时钟分别分频为 400Hz（扫描脉冲）和 10Hz（计时脉冲）。按键处理模块 (`button_unit`) 对输入按键进行同步化、去抖动和脉冲转换。控制器 (`controller`) 根据处理后的按键信号实现状态机控制，输出计数使能和清零信号。计时器 (`timing`) 在控制信号作用下进行时间计数。动态显示模块 (`display`) 将计时结果通过数码管动态显示。

```
module controller(
    input in,
    input clk,
    input reset,
    output reg count,
    output reg clr
);
    //三段式
    //状态编码
    parameter RESET = 2'b00;
    parameter TIMING = 2'b01;
    parameter STOP = 2'b10;

    ///machine variable
    reg [1:0] st_next;//next state
    reg [1:0] st_cur;//current state

    //1. state transfer
    always @(posedge clk or posedge reset)begin
        if(reset)
            st_cur=RESET;
        else
            st_cur=st_next;
    end

    //2.state switch
    always @(*)begin
        case(st_cur)
            RESET:
                case(in)
                    1'b0: st_next=RESET;
                    1'b1: st_next=TIMING;
                endcase
            TIMING:
                case(in)
                    1'b0: st_next=TIMING;
                    1'b1: st_next=STOP;
                endcase
            STOP:
                case(in)
                    1'b1: st_next=RESET;
                    1'b0: st_next=STOP;
                endcase
        endcase
    end
endmodule
```

```

        endcase
    endcase
end
//3. output logic
always @ (posedge clk or posedge reset) begin
    if (reset) begin
        clr = 1;
        count = 0;
    end
    //如果没开始，就不用计时，输出也是0
    else begin
        case (st_cur)
            RESET: begin
                clr <= 1;
                count <= 0;
            end
            TIMING: begin
                clr <= 0;
                count <= 1;
            end
            STOP: begin
                clr <= 0;
                count <= 0;
            end
        endcase
    end
end
endmodule

```

为了添加个性要求，我们在顶层模块设计了冻结逻辑 `display_d0` ~ `display_d3`，在第一次按下到第二次按下的时间内，不更新显示值。冻结逻辑的工作原理是：当状态机处于 `FIRST` 状态时，`first` 信号置高，此时显示寄存器停止更新，保持第一次按键时的计时值；而计时器模块继续在后台运行计数；当第二次按键后进入 `STOP` 状态，`first` 信号清零，显示寄存器恢复更新，显示最终的计时结果。通过这种设计，实现了显示冻结与后台计时分离的功能，满足了个性化需求。

```

//This is TOP MODULE!
module stopwatch(
    input clk,
    input ButtonIn,
    input reset,
    output a,
    output b,
    output c,
    output d,
    output e,
    output f,
    output g,
    output dp,
    output [3:0] pos
);
    parameter sim = 0;
    wire pulse400Hz;
    wire pulse10Hz;

```

```

//n=4进制分频器
//sim=0, 实际使用代码, 分频比分别是25_0000/200
//sim=1, 测试代码,16进制, 分频比16/4

//分频器1
counter_n #(
    .n(sim?2:250000),
    .counter_bits(sim?2:18)
)
u_counter(
    .clk(clk),
    .en(1),
    .r(0),
    .q(),
    .co(pulse400Hz)
);

//分频器2
counter_n #(
    .n(sim?10:40),
    .counter_bits(sim?4:6)
)
u_counter2(
    .clk(clk),
    .en(pulse400Hz),
    .r(0),
    .q(),
    .co(pulse10Hz)
);
//按键处理模块

wire ButtonOut;

button_unit #(
    .sim(sim)
)
u_button(
    .clk(clk),
    .ButtonIn(ButtonIn),
    .reset(reset),
    .ButtonOut(ButtonOut)
);
wire count;
wire clr;
wire first;
//控制器
controller u_controller(
    .in(ButtonOut),
    .clk(clk),
    .reset(reset),
    .count(count),
    .clr(clr),
    .first(first)
);
//计时器

```

```

wire [3:0] d0;
wire [7:0] d1;
wire [3:0] d3;
//冻结寄存器
reg [3:0] display_d0;
reg [7:0] display_d1;
reg [3:0] display_d3;

timing u_timer(
    .clk(clk),
    .en(pulse10Hz&count),
    .r(clr),
    .d0(d0),
    .d1(d1),
    .d3(d3)
);

// 添加显示冻结逻辑
always @ (posedge clk or posedge reset) begin
    if(reset) begin
        display_d0 <= 0;
        display_d1 <= 0;
        display_d3 <= 0;
    end
    // first=0, 不更新显示值, 其他时候实时更新
    else if(!first) begin
        display_d0 <= d0;
        display_d1 <= d1;
        display_d3 <= d3;
    end
    // 在STOP状态, 不更新显示值, 但计时器继续运行
end

//动态显示模块
display u_display(
    .d0(display_d0),
    .d1(display_d1[3:0]),
    .d2(display_d1[7:4]),
    .d3(display_d3),
    .clk(clk),
    .scan(pulse400Hz),
    .a(a),
    .b(b),
    .c(c),
    .d(d),
    .e(e),
    .f(f),
    .g(g),
    .dp(dp),
    .pos(pos)
);

```

endmodule

4.2 控制器模块的 Verilog HDL 描述

控制器模块中，输入为按键信号 `in`、系统时钟 `clk` 和复位信号 `reset`，输出为计数使能信号 `count` 和清零信号 `clr`。模块采用三段式状态机设计，包含三个状态：复位状态（`RESET`）、计时状态（`TIMING`）和停止状态（`STOP`）。状态转移逻辑通过按键输入控制：在复位状态下按键启动计时，在计时状态下按键停止计时，在停止状态下按键重新复位。输出逻辑根据当前状态控制计时器的工作：复位状态输出清零信号，计时状态输出计数使能信号，停止状态保持计时结果不变。通过状态机的设计，实现了秒表的启动、停止和复位功能控制。

```
module controller(
    input in,
    input clk,
    input reset,
    output reg count,
    output reg clr
);
    //三段式
    //状态编码
    parameter RESET = 2'b00;
    parameter TIMING = 2'b01;
    parameter STOP = 2'b10;

    ///machine variable
    reg [1:0] st_next;//next state
    reg [1:0] st_cur;//current state

    //1. state transfer
    always @(posedge clk or posedge reset)begin
        if(reset)
            st_cur=RESET;
        else
            st_cur=st_next;
    end

    //2.state switch
    always @(*)begin
        case(st_cur)
            RESET:
                case(in)
                    1'b0: st_next=RESET;
                    1'b1: st_next=TIMING;
                endcase
            TIMING:
                case(in)
                    1'b0: st_next=TIMING;
                    1'b1: st_next=STOP;
                endcase
            STOP:
                case(in)
                    1'b1: st_next=RESET;
                    1'b0: st_next=STOP;
                endcase
        endcase
    end
endmodule
```

```

    end
    //3. output logic
    always @(posedge clk or posedge reset) begin
        if(reset) begin
            clr = 1;
            count = 0;
        end
        //如果没开始，就不用计时，输出也是0
        else begin
            case(st_cur)
                RESET: begin
                    clr <= 1;
                    count <= 0;
                end
                TIMING: begin
                    clr <= 0;
                    count <= 1;
                end
                STOP: begin
                    clr <= 0;
                    count <= 0;
                end
            endcase
        end
    end
endmodule

```

为了添加个性要求，在原有三状态基础上增加了 **FIRST** 状态，形成四状态控制器。新的状态转移逻辑为：**RESET**→**TIMING**→**FIRST**→**STOP**→**RESET**，其中 **FIRST** 状态用于控制显示冻结功能。在 **FIRST** 状态下，**first** 输出信号置高，指示顶层模块停止更新显示寄存器，而计时器继续运行，实现显示冻结与后台计时的分离控制。

```

module controller(
    input in,
    input clk,
    input reset,
    output reg count,
    output reg clr,
    output reg first
);
    //三段式
    //状态编码
    parameter RESET = 2'b00;
    parameter TIMING = 2'b01;
    parameter FIRST = 2'b10;
    parameter STOP = 2'b11;

    ///machine variable
    reg [1:0] st_next;//next state
    reg [1:0] st_cur;//current state

    //1. state transfer
    always @(posedge clk or posedge reset)begin
        if(reset)
            st_cur=RESET;

```

```

    else
        st_cur=st_next;
end

//2.state switch
always @(*)begin
    case(st_cur)
        RESET:
            case(in)
                1'b0: st_next=RESET;
                1'b1: st_next=TIMING;
            endcase
        TIMING:
            case(in)
                1'b0: st_next=TIMING;
                1'b1: st_next=FIRST;
            endcase
        FIRST:
            case(in)
                1'b0: st_next=FIRST;
                1'b1: st_next=STOP;
            endcase
        STOP:
            case(in)
                1'b1: st_next=RESET;
                1'b0: st_next=STOP;
            endcase
    endcase
end
//3. output logic
always @ (posedge clk or posedge reset) begin
    if(reset) begin
        clr = 1;
        count = 0;
    end
    //如果没开始, 就不用计时, 输出也是0
    else begin
        case(st_cur)
            RESET: begin
                clr <= 1;
                count <= 0;
                first <=0;
            end
            TIMING: begin
                clr <= 0;
                count <= 1;
                first <=0;
            end
            FIRST: begin
                clr <= 0;
                count <= 1;
                first <=1;
            end
            STOP: begin
                clr <= 0;
                count <= 0;
            end
        end
    end

```

```

        first <=0;
    end
endcase
end
end
endmodule

```

4.3 计时器模块的 Verilog HDL 描述

模块 `timing` 顶层模块中，输入为系统时钟 `clk`、使能信号 `en` 和复位信号 `r`，输出为三个时间计数值：4 位的 `d0`、8 位的 `d1` 和 4 位的 `d3`。模块内部包含两个十进制计数器和一个 60 进制 BCD 计数器，构成级联的时间计数系统。

十进制计数器 1 (`timer_counter1`) 以系统时钟和使能信号进行 0-9 的循环计数，输出 4 位计数值 `d0` 作为最低位时间单位，并在计满时产生进位信号 `co1`。60 进制 BCD 计数器 (`u_counter_bcd`) 接收 `co1` 作为使能信号，实现 0-59 的 BCD 码计数，输出 8 位数据 `d1` 表示中间位时间单位，计满时产生进位信号 `co2`。十进制计数器 2 (`timer_counter2`) 接收 `co2` 作为使能信号，输出 4 位计数值 `d3` 作为最高位时间单位。

```

module timing(
    input clk,
    input en,
    input r,
    output [3:0] d0,
    output [7:0] d1,
    output [3:0] d3
);
    wire co1;
    wire co2;
    //10进制计数器
    counter_n #(
        .n(10),
        .counter_bits(4)
    )
    timer_counter1(
        .clk(clk),
        .en(en),
        .r(r),
        .q(d0),
        .co(co1)
    );
    //60进制BCD计数器
    counter_bcd u_counter_bcd(
        .clk(clk),
        .en(co1),
        .r(r),
        .co(co2),
        .d(d1)
    );
    //10进制计数器
    counter_n #(
        .n(10),
        .counter_bits(4)
    )
    timer_counter2(

```

```

    .clk(clk),
    .en(co2),
    .r(r),
    .q(d3),
    .co()
);
endmodule

```

4.4 60 进制 BCD 计数器 Verilog HDL 描述

模块名 `counter_bcd`，为 60 进制 BCD 计数器。输入信号有系统时钟 `clk`、使能信号 `en` 和复位信号 `r`，输出为进位信号 `co` 以及 8 位 BCD 码数据 `d`。模块内部使用 8 位寄存器 `cnt` 实现 0-59 的循环计数，并通过 `carry` 标志位产生进位。当 `r` 有效时，`cnt` 和 `carry` 清零；当 `en` 有效时，`cnt` 递增，达到 59 时清零并置位 `carry`。输出逻辑将 `cnt` 的二进制值转换为 BCD 码：`d[3:0]` 输出个位 (`cnt % 10`)，`d[7:4]` 输出十位 (`cnt / 10`)。进位信号 `co` 在溢出且 `en` 有效时输出，为上级计数器提供时钟。该设计实现了适用于时间计数的 60 进制 BCD 计数功能，满足分钟和秒钟的计数需求。

```

module counter_bcd(
    input clk,
    input en,
    input r,
    output co,
    output [7:0] d
);
reg [7:0] cnt = 0;
reg carry = 0;
always @(posedge clk)begin
    if(r)begin
        cnt <= 0;
        carry <= 0;
    end
    else if (en) begin
        if(cnt==59)begin
            cnt <= 0;
            carry <= 1;
        end
        else begin
            cnt <= cnt+1;
            carry <= 0;
        end
    end
    assign d[3:0] = cnt%10;
    assign d[7:4] = cnt/10;
    assign co = carry & en;
endmodule

```

五、模块测试与分析

5.1 60 进制 BCD 计数器测试

使用自主编写的 testbench 对 60 进制 BCD 计数器进行功能验证：

```
module counter_bcd_tb;
```

```

parameter delay = 10;
reg clk;
reg en;
reg r;
wire co;
wire [7:0] d;

//实例化

counter_bcd u_counter(
    .clk(clk),
    .en(en),
    .r(r),
    .co(co),
    .d(d)
);

initial begin
    clk = 0;
    forever #(delay) clk = ~ clk;
end

initial begin
    //初始化
    en=0;
    r=1;
    #(5*delay);
    en=1;
    r=0;
    #(3000*delay);
    $finish;
end
endmodule

```

实验波形如下。由波形，当d计到60时，有进位信号输出，并持续一个时钟周期，说明功能正常。

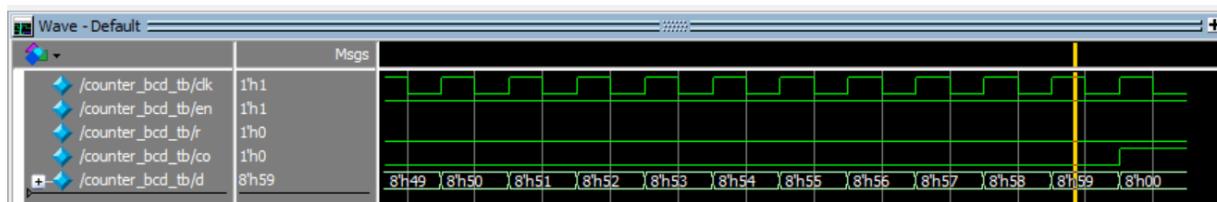


图 25 60 进制 BCD 码

5.2 控制器测试

根据本实验要求的控制器模块，依次测试不同输出及不同状态下的控制器，代码如下：

```

module controller_tb;
parameter delay = 10;
reg in;
reg clk;
reg reset;

```

```

wire clr;
wire count;
//实例化

controller u_controller(
    .clk(clk),
    .in(in),
    .reset(reset),
    .count(count),
    .clr(clr)
);

initial begin
    clk = 0;
    forever #(dely) clk = ~clk;
end

initial begin
    //初始化
    in=0;
    reset = 1;
    #(5*dely);
    reset=0;
    #(5*dely);
    in=0;
    #(5*dely);
    in=1;
    #(5*dely);
    in=0;
    #(5*dely);
    in=1;
    #(5*dely);
    in=0;
    #(5*dely);
    in=1;
    #(5*dely);
    $finish;
end

initial begin
    $monitor("Time = %0t, clk = %b, reset = %b, in = %b, clr = %b, count = %b", $time, clk,
    reset, in, clr, count);
end
endmodule

```

实验测试波形如下，并观察控制台输出。

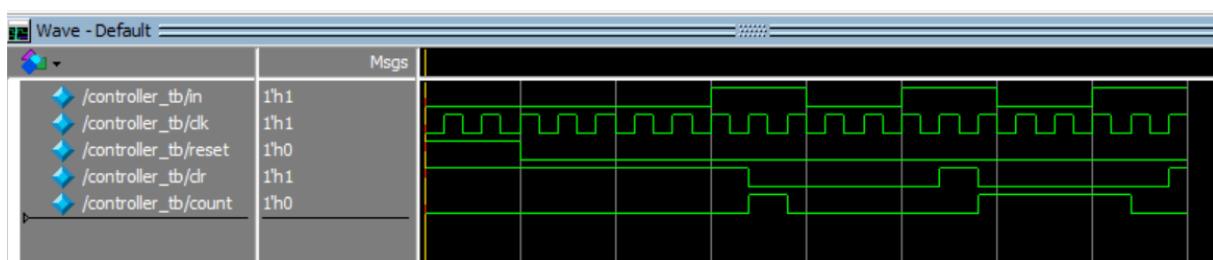


图 26 控制器测试

```

# Loading work.controller
add wave sim:/controller_tb/*
VSIM 34> run -all
# Time = 0, clk = 0, reset = 1, in = 0, clr = 1, count = 0
# Time = 10, clk = 1, reset = 1, in = 0, clr = 1, count = 0
# Time = 20, clk = 0, reset = 1, in = 0, clr = 1, count = 0
# Time = 30, clk = 1, reset = 1, in = 0, clr = 1, count = 0
# Time = 40, clk = 0, reset = 1, in = 0, clr = 1, count = 0
# Time = 50, clk = 1, reset = 0, in = 0, clr = 1, count = 0
# Time = 60, clk = 0, reset = 0, in = 0, clr = 1, count = 0
# Time = 70, clk = 1, reset = 0, in = 0, clr = 1, count = 0
# Time = 80, clk = 0, reset = 0, in = 0, clr = 1, count = 0
# Time = 90, clk = 1, reset = 0, in = 0, clr = 1, count = 0
# Time = 100, clk = 0, reset = 0, in = 0, clr = 1, count = 0
# Time = 110, clk = 1, reset = 0, in = 0, clr = 1, count = 0
# Time = 120, clk = 0, reset = 0, in = 0, clr = 1, count = 0
# Time = 130, clk = 1, reset = 0, in = 0, clr = 1, count = 0
# Time = 140, clk = 0, reset = 0, in = 0, clr = 1, count = 0
# Time = 150, clk = 1, reset = 0, in = 1, clr = 1, count = 0
# Time = 160, clk = 0, reset = 0, in = 1, clr = 1, count = 0
# Time = 170, clk = 1, reset = 0, in = 1, clr = 0, count = 1
# Time = 180, clk = 0, reset = 0, in = 1, clr = 0, count = 1
# Time = 190, clk = 1, reset = 0, in = 1, clr = 0, count = 0
# Time = 200, clk = 0, reset = 0, in = 0, clr = 0, count = 0
# Time = 210, clk = 1, reset = 0, in = 0, clr = 0, count = 0
# Time = 220, clk = 0, reset = 0, in = 0, clr = 0, count = 0
# Time = 230, clk = 1, reset = 0, in = 0, clr = 0, count = 0
# Time = 240, clk = 0, reset = 0, in = 0, clr = 0, count = 0
# Time = 250, clk = 1, reset = 0, in = 1, clr = 0, count = 0
# Time = 260, clk = 0, reset = 0, in = 1, clr = 0, count = 0
# Time = 270, clk = 1, reset = 0, in = 1, clr = 1, count = 0
# Time = 280, clk = 0, reset = 0, in = 1, clr = 1, count = 0
# Time = 290, clk = 1, reset = 0, in = 1, clr = 0, count = 1
# Time = 300, clk = 0, reset = 0, in = 0, clr = 0, count = 1
# Time = 310, clk = 1, reset = 0, in = 0, clr = 0, count = 1
# Time = 320, clk = 0, reset = 0, in = 0, clr = 0, count = 1
# Time = 330, clk = 1, reset = 0, in = 0, clr = 0, count = 1
# Time = 340, clk = 0, reset = 0, in = 0, clr = 0, count = 1
# Time = 350, clk = 1, reset = 0, in = 1, clr = 0, count = 1
# Time = 360, clk = 0, reset = 0, in = 1, clr = 0, count = 1
# Time = 370, clk = 1, reset = 0, in = 1, clr = 0, count = 0
# Time = 380, clk = 0, reset = 0, in = 1, clr = 0, count = 0
# Time = 390, clk = 1, reset = 0, in = 1, clr = 1, count = 0
# ** Note: $finish : D:/MyRepository/SystemDesignLab/3230103034/lab12_stopwatch/src/controller_tb.v(45)
#   Time: 400 ns  Iteration: 0  Instance: /controller_tb
# 1
# Break in Module controller_tb at D:/MyRepository/SystemDesignLab/3230103034/lab12_stopwatch/src/controller_tb.v line 45

```

图 27 控制器测试

分析状态变化：

1. **0-50ns:** `reset=1`, 状态机处于 RESET 状态, `clr=1, count=0`
2. **50-150ns:** `reset=0, in=0`, 状态机保持在 RESET 状态, `clr=1, count=0`
3. **150-170ns:** `in=1`, 但状态还未更新
4. **170ns:** 时钟上升沿, 检测到 `in=1`, 状态从 RESET 变为 TIMING, 输出变为 `clr=0, count=1`
5. **190ns:** 时钟上升沿, 检测到 `in=1`, 状态从 TIMING 变为 STOP, 输出变为 `clr=0, count=0`
6. **200-250ns:** `in=0`, 状态机保持在 STOP 状态, `clr=0, count=0`
7. **250-270ns:** `in=1`, 但状态还未更新
8. **270ns:** 时钟上升沿, 检测到 `in=1`, 状态从 STOP 变为 RESET, 输出变为 `clr=1, count=0`
9. **290ns:** 时钟上升沿, 检测到 `in=1`, 状态从 RESET 变为 TIMING, 输出变为 `clr=0, count=1`
10. **300-350ns:** `in=0`, 状态机保持在 TIMING 状态, `clr=0, count=1`

11. **350-370ns**: `in=1`, 但状态还未更新
12. **370ns**: 时钟上升沿, 检测到 `in=1`, 状态从 TIMING 变为 STOP, 输出变为 `clr=0, count=0`
13. **390ns**: 时钟上升沿, 检测到 `in=1`, 状态从 STOP 变为 RESET, 输出变为 `clr=1, count=0`

因此:

- 在 RESET 状态: `clr=1, count=0` ✓
- 在 TIMING 状态: 预期 `clr=0, count=1` ✓
- 在 STOP 状态: 预期 `clr=0, count=0` ✓

说明控制器功能正确。

5.3 计时器测试

使用所给的测试代码进行测试, 结果如下。

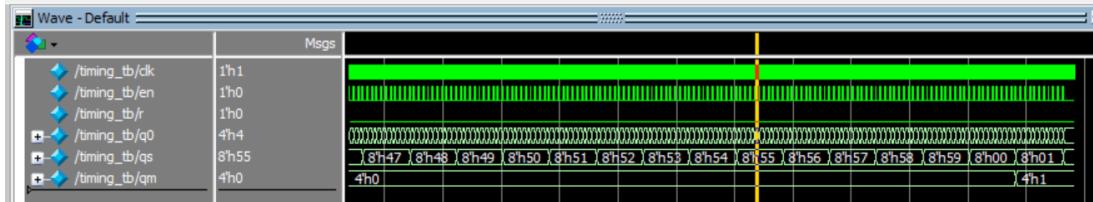


图 28 计时器测试

根据测试结果, 计时器能正常计时, 并且在需要进位的时候, `qm`输出进位信号, 说明计时器整体功能正常。

5.4 数字秒表测试

使用测试代码进行测试:

计时开始时, 显示的数字是 0:

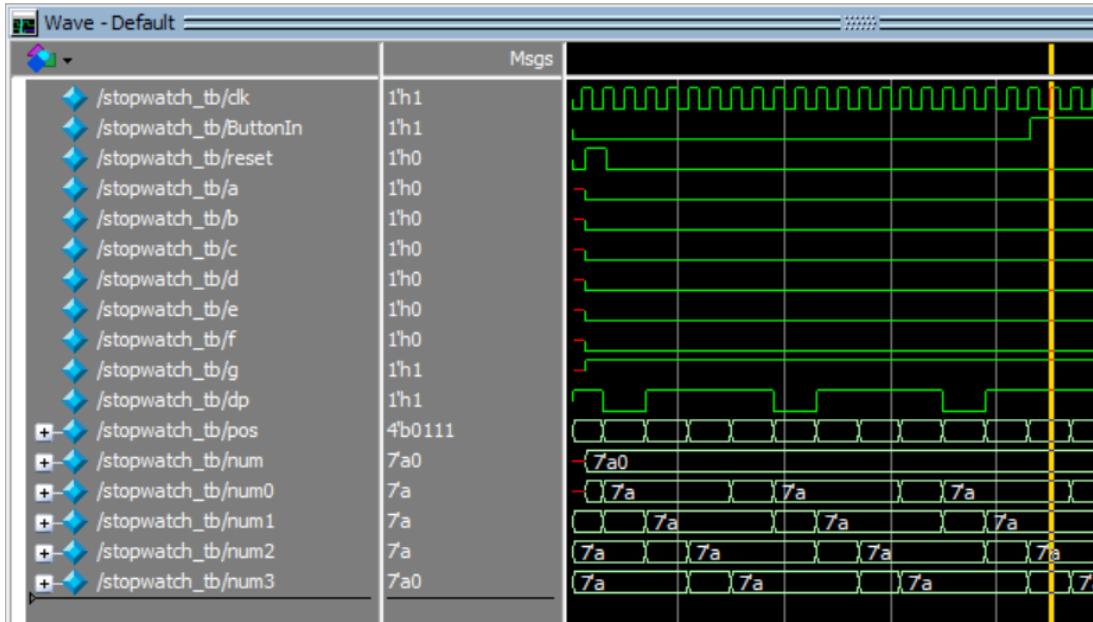


图 29 计时器测试

计时结束时，秒表显示数字：

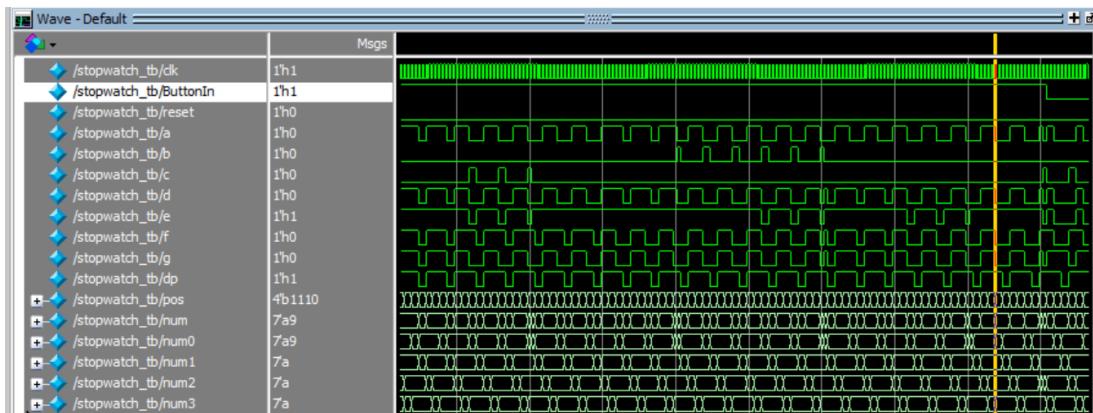


图 30 计时器测试

在个性化要求中，两次按键处理中 num 未发生改变，说明总体数字秒表的功能正常。

六、引脚约束

根据下面的引脚约束表格，在 Vivado 中设置引脚约束：

| 引脚名称 | I/O | 引脚编号 | 电气特性 | 说明 |
|-------------|--------|------|---------|---------------|
| clk | Input | W5 | LVCMS33 | 系统 100MHz 主时钟 |
| ButtonIn | Input | T18 | LVCMS33 | 上边按键 |
| reset | Input | U18 | LVCMS33 | 中间按键 |
| a | Output | W7 | LVCMS33 | 七段码 |
| b | Output | W6 | LVCMS33 | 七段码 |
| c | Output | U8 | LVCMS33 | 七段码 |
| d | Output | V8 | LVCMS33 | 七段码 |
| e | Output | U5 | LVCMS33 | 七段码 |
| f | Output | V5 | LVCMS33 | 七段码 |
| g | Output | U7 | LVCMS33 | 七段码 |
| position[0] | Output | U2 | LVCMS33 | 4 个数码管的点亮控制端 |
| position[1] | Output | U4 | LVCMS33 | 4 个数码管的点亮控制端 |
| position[2] | Output | V4 | LVCMS33 | 4 个数码管的点亮控制端 |
| position[3] | Output | W4 | LVCMS33 | 4 个数码管的点亮控制端 |
| dp | Output | V7 | LVCMS33 | 小数点 |

表 4 FPGA 引脚约束内容

七、主要仪器设备

- 电脑（Modelsim、Vivado 软件）
- Basys3 开发板

八、上板实现

代码烧录后，第一次按下按键计时开始，并以 0.1s 的刷新率刷新，且进位正常。第二次按下按键显示当前时间保持不动；第三次按下按键显示最终时间。说明实验要求完成。

- 计时中：

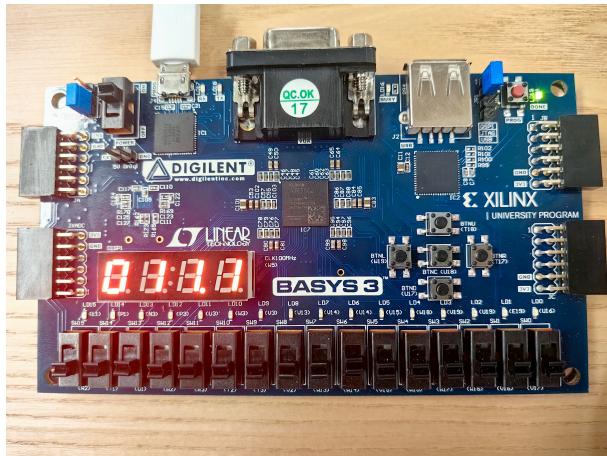


图 31 计时中

- 第一次暂停：

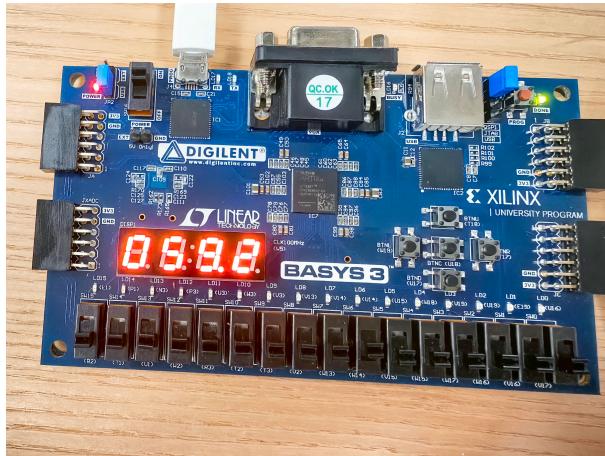


图 32 第一次暂停

- **第二次暂停:**

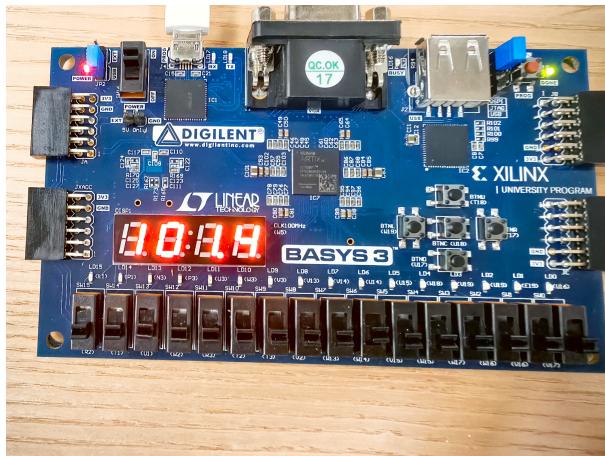


图 33 第二次暂停

九、思考题

1. **分频器二进制编码:** 分频器的主要功能是对输入的时钟信号进行计数，然后输出一个频率较低的信号。在这个过程中，计数器的内部运算是基于二进制的，使用二进制编码最直接、最高效。数字电路的底层逻辑运算都是基于二进制的，采用二进制编码可以简化电路设计，减少逻辑门数量，提高运行速度。

二进制编码是最紧凑的数字表示方式，可以最大限度地利用硬件资源，减少存储单元和逻辑电路的开销。

2. **计时模块 BCD 编码:** 计时模块（如数字时钟、秒表）的最终目的是将时间信息显示给用户。人类习惯于使用十进制来表示时间（例如，0-9 秒，0-59 分，0-23 小时等）。如果计时模块的计数状态采用二进制编码，那么在显示时就需要额外的二进制到十进制的转换电路，这会增加设计的复杂性和成本。

同时，BCD（Binary-Coded Decimal，二进制编码的十进制）编码是将每个十进制数字独立地用四位二进制数表示。例如，十进制的“5”在 BCD 码中是 0101，而十进制的“12”在 BCD 码中是 0001 0010。这种编码方式使得直接驱动七段数码管等显示器变得非常简单，因为每个 BCD 码位可以直接对应一个数码管的显示段。

第四部分：碰到的问题和解决方法

1. 在顶层模块中使用 `reg` 传递信号报错。即使子模块里面定义的是 `reg`，但在顶层应该用 `wire` 来传递信号。
2. 引脚约束的时候没有选择 Basys3 相应的开发板，所以导致找不到列表中的引脚。
3. 单纯分析所给出的模块发现出错，因此养成了写一个模块就完成对应测试代码的习惯。在最终顶层出错之后，修改所给的测试代码，可以多输出几个中间使用的信号。
4. 按键处理模块，一开始发现电路板无法显示输出的脉冲信号，后来接入了 `dffre` 就正常了。
5. 测试代码里面没写 `$stop` 或是 `$finish`, `run-all` 之后仿真停不下来。一开始仿真波形看的头疼，后面才学会在控制台打印信号的值对比检查。
6. 对于 `verilog` 还有 vivado 的不熟悉，导致一直出现各种语法还有调试的错误，有些还挺蠢的，有点难以一一列举了。

第五部分：心得与体会

这次数电实验是我第一次从新建文件夹开始写这么大的工程，每一个模块我几乎都是从 verilog 代码的语法开始学，然后逐一测试的。所幸我觉得我的整个过程还挺顺利的，几乎没有发生太多错误，虽然也有卡住的时候，但一路写一路调完成的相当顺利。

发现 verilog 比数电大多数内容好像都好理解了很多，总体也觉得 Verilog HDL 硬件描述挺好上手，而且挺感兴趣的。因此下学期选了计算机组成与设计，希望能继续学习这些知识！