

Verilog

一.Verilog简介

1.Verilog 模块

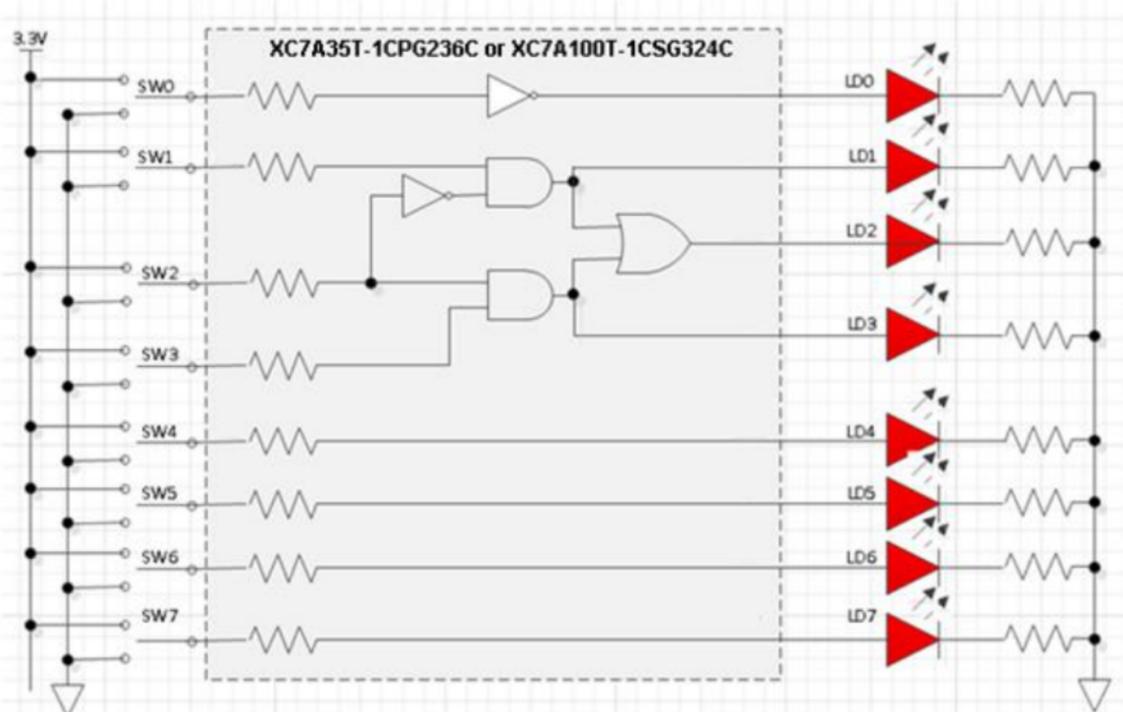
在Verilog语言中，一个电路就是一个module

```
module decoder_2_to_4 (A, D) ;//A,D表示两个接口,该句末尾要加分号

    input [1:0] A ;//两个输入引脚A0,A1,[1:0]对应的2位为二进制位
    output [3:0] D ;//四个输出引脚D0,D1,D2,D3

    assign D =      (A == 2'b00) ? 4'b0001 :
    //2'代表2位宽的数, b代表二进制, 00代表输入的值
    //?后面代表输出的值,:即为条件表达式
    //assign代表连续赋值语句,只要左边表达式值变化,就会立马更新D的值,若有多个assign语句则会
    同时执行
        (A == 2'b01) ? 4'b0010 :
        (A == 2'b10) ? 4'b0100 :
        (A == 2'b11) ? 4'b1000 : 4'b0000 ;
endmodule //模块结束
```

2.Lab1.v



```
'timescale 1ns / 1ps//1ns表示时延,1ps表示时延精度
////////////////////////////
// Module Name: lab1
```

```
module lab1(input [7:0] swt,output [7:0] led);

assign led[0] = ~swt[0];
assign led[1] = swt[1] & ~swt[2];
assign led[3] = swt[2] & swt[3];
assign led[2] = led[1] | led[3];

assign led[7:4] = swt[7:4];

endmodule
```

2.Lab1_tb.v

仿真电路用于进行测试

```

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Module Name: lab1_tb
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module lab1_tb(); //括号可以省略

    reg [7:0] switches; //reg为寄存器类型
    wire [7:0] leds; //wire为线型
    reg [7:0] e_led;

    integer i; //整型

    lab1 dut(.led(leds), .swt(switches)); //模块的输入端口为led, 输出端口为swt

    function [7:0] expected_led; //定义函数, 函数类型为一个8维向量
        input [7:0] swt; //至少需要定义一个输入端口
        //会隐式得声明一个与函数同名的寄存器变量
    begin
        expected_led[0] = ~swt[0]; //对寄存器赋值来返回函数值
        expected_led[1] = swt[1] & ~swt[2];
        expected_led[3] = swt[2] & swt[3];
        expected_led[2] = expected_led[1] | expected_led[3];
        expected_led[7:4] = swt[7:4];
    end
    endfunction

    initial //初始化语句, 仅执行一次
    begin
        for (i=0; i < 255; i=i+2)
        begin
            #50 switches=i; //表示时延, 在50个单位时间后将i赋给switches
            #10 e_led = expected_led(switches);
            if (leds == e_led)
                $display("LED output matched at", $time); //time为系统函数, display
            //类似于C语言的printf
            else
                $display("LED output mis-matched at ", $time, ": expected: %b,
actual: %b", e_led, leds);
        end
    end
endmodule

```

```

    end
end

endmodule

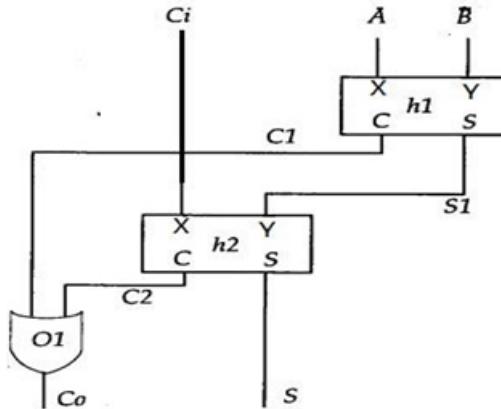
```

二.Verilog程序设计方法—组合电路

1.Verilog的三种描述方式

1) 结构化描述：使用实例化语句

以全加器的实现为例：下图的全加器通过两个半加器实现



```

//半加器模块
module half_add (X, Y, S, C);
    input X, Y ;
    output S, C ;
    xor SUM (S, X, Y); // 异或门实例语句
    and CARRY (C, X, Y); // 与门实例语句
endmodule

```

```

module full_add ( input A, B, CI, output S, CO) ;

    wire S1, C1, C2;      // 内部信号
    half_add h1 (A, B, S1, C1),h2 (S1, CI, S, C2); //两个 half_add半加器模块实例语句,
    端口位置关联
    or CARRY (CO, C2, C1); // or门实例语句

endmodule

```

除了端口位置关联，还可以利用端口名关联，格式如下：

```
.<port name>(<signal name>)
```

```

module Add_full(Co,S, A, B, Ci) ;
    output Co,S;  input A, B, Ci;  wire S1, C1, C2;

    Add_half h1( .C(C1), .S(S1), .X(A), .Y(B)); /* 端口名称关联 */
    Add_half h2( .C(C2), .S(S), .X(Ci), .Y(S1));
    or carry_bit(Co, C1, C2); /* 端口位置关联 */
endmodule

```

2) 数据流描述：使用连续赋值语句

全加器的数据流语句描述如下：

```

module fa_rtl (A, B, CI, S, CO) ;

    input A, B, CI ;//没说就默认1位宽
    output S, CO ;

    assign S = A & B & CI;
    assign CO = (A & B) | (A & CI) | (B & CI);
    //assign { CO , S } = A + B + CI ;//等价于该语句,CO代表高位, S代表低位
endmodule

```

3) 行为级描述：使用过程块语句

```

module fa_bhv (A, B, CI, S, CO) ;
    input A, B, CI;
    output S, CO; //也可以直接写成output reg S, CO;
    reg S, CO;// 只有reg型能够在 always 语句中被赋值
    always@(A or B or CI)// 由事件控制的always语句，always也一直在工作
        begin
            S = A & B & CI;
            CO = (A & B) | (A & CI) | (B & CI); // 过程性赋值语句,必须在 always 过程
       块中
        end
endmodule

```

只要有事件发生（列表中任何信号有变化），就执行begin...end 的语句

还可以利用混合方式进行程序设计：

```

module Add_half_bhv(c_out, sum, a, b);
    output reg sum, c_out;
    input a, b;
    always @ (a, b)
        begin
            sum = a & b;
            c_out = a & b;
        end
endmodule

```

```

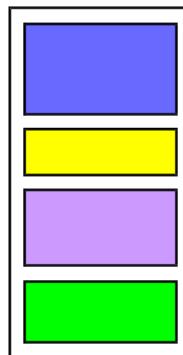
module Add_full_mix(c_out, sum, a, b, c_in) ;
    output sum, c_out;
    input a, b, c_in;
    wire w1, w2, w3;
    Add_half_bhv AH1(.sum(w1), .c_out(w2), .a(a), .b(b));
    Add_half_bhv AH2(.sum(sum), .c_out(w3), .a(c_in), .b(w1));
    assign c_out = w2 | w3;
endmodule

```

2. 层次化与源文件

多个module是放在同一个文件还是分开放?

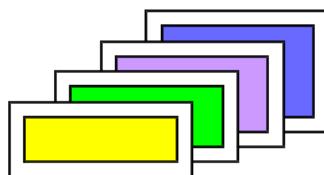
- 1) 可以放在同一个文件中: 模块次序自由, 适合于小设计, 不太适合模块重用 (剪切和粘贴)



- 2) 可以将模块分解成多个文件

良好习惯: 每个文件只包含一个module

有助于组织模块, 便于找到一个模块, 有利于模块重用 (将文件添加到项目中)



3.Verilog的行为描述

- 1) 行为建模的主要机制: initial and always

说明:

- ①所有其他的行为语句包含在这两个语句中, 如有多条行为语句, 要用begin/end
- ②各语句从0时刻开始并行执行
- ③initial and always 块不能嵌套
- ④左边变量必须是 reg 类型

- 2) initial语句: initial只执行一次, alway 语句重复执行, initial常用于仿真的Testbench模块中

说明: 所有initial从0时刻开始并行执行, initial的begin和end中的时延语句串行执行

```

module stim();
    reg m,a,b,x,y;
    m = 1'b0;
    initial begin
        #5 a = 1'b1; #25 b = 1'b1;
    end
    initial begin
        #10 x = 1'b1; #25 y = 1'b1;
    end
    initial
        #50 $finish;
endmodule

```

执行结果如下：

Time	Event
0	m = 1'b0
5	a = 1'b1
10	x = 1'b1
30	b = 1'b1
35	y = 1'b1
50	\$finish

3) always 语句：always语句重复执行，从0时刻开始不断循环执行；

可以使用触发/敏感事件列表来控制操作，类似于 @ (a or b or c)；没有触发事件，将在0时刻无限循环执行。

```

module clock_gen (output reg clock);
    initial
        clock = 1'b0; // must initialize in initial block
    always
        #10 clock = ~clock; // no trigger list for this always
                            // 带有时序控制
                            // 产生周期为20时间单位的波形
endmodule

```

4) always的事件控制方式

always是基于事件执行，有两种类型的事件控制方式

①边沿触发事件控制：用来描述时序逻辑电路，posedge代表上升沿，negedge代表下降沿触发

```

always @(posedge    clk)      // clk从低电平->高（正沿）
    cur_state =next_state;    // 就执行赋值语句
always @(negedge    reset)    // reset从高->低（负沿）
    count =0;                // 就执行赋值语句
always @ (posedge    clear or negedge reset)//不可以同时包括同一个信号的上升沿和下降沿,
所以要用or
    Q=0;

```

②电平敏感事件控制：用来描述组合逻辑电路

只要列表内有信号发生电平变化，就执行该always结构中的内容。

不可以同时包括电平敏感事件和边沿敏感事件

```
always @ (x1 or x2 or x3)//original way to specify trigger list  
always @ (x1, x2, x3)//in verilog 2001 can use "," instead of "or"  
always @ (*) //Verilog 2001 also has * for combinational only,"*"代表所有输入信号，可防止遗漏
```

不可以同时包括电平敏感事件和边沿敏感事件

三.Verilog程序设计方法—时序电路

1.参数parameter

有时候希望模块成为一般化的模块，即希望端口位数可选。 parameter可实现此功能，在调用模块时可改变该参数的值

parameter: 为一个常量，在本module中有效，其作用是模块间的参数传递

```
module mux2_1(out1, a, b, sel) ;  
    parameter N=2; //本模块内不可变  
    output [N-1:0] out1;  
    input [N-1:0] a, b;  
    input sel;  
    assign out1= sel ? b : a;  
endmodule
```

在顶层模块中：

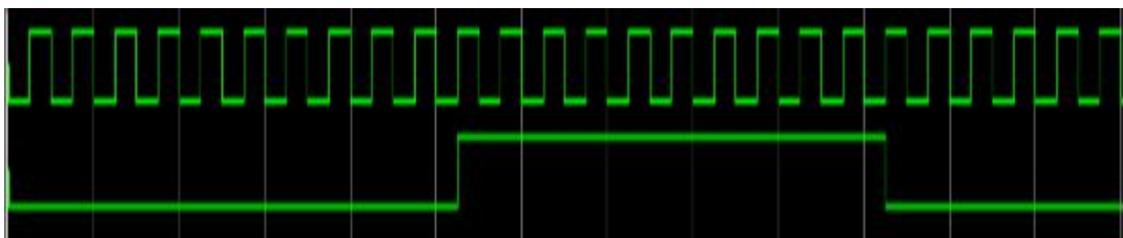
```
mux2_1 #(4) dut0( out, x,y,s ); //实例化时参数N值为4
```

2.分频器

分频：将信号的频率降低为原来的 $1/N$ ，就叫N分频。

分频器：实现分频的电路。

分频的实现：通过计数实现分频，当计数器从0计数到($N/2-1$)时，输出时钟翻转，计数器清零



20分频

```

module divider(input clk, output reg clk_N);
    parameter N=20; // 参数说明, 默认值20
    reg [31:0] count;
    always @(posedge clk) begin
        end
endmodule

```

```

//带参数值的模块引用
divider #(100) (clk100M,clk1M);
//or
divider #(100) D1(clk100M,clk1M);
//或者在调用divider的上层模块中用//defparam来改写参数值
defparam D1.N=100;
divider D1(clk100M,clk1M);

```

3.简单加法计数器

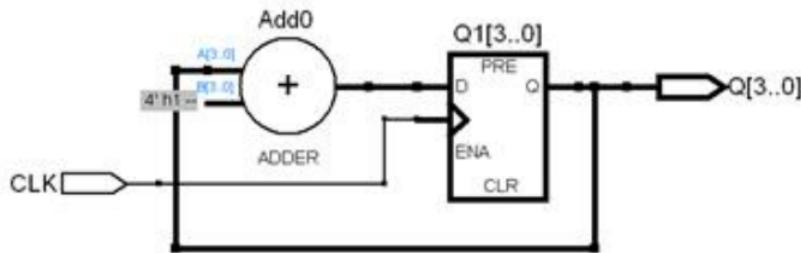


图 5-17 4 位加法计数器 RTL 电路图

```

module counter(clk, out);
    input clk; // 计数时钟
    output [2:0] out; // 计数值
    always @ (posedge clk) begin // 在时钟上升沿计数器加1
        // 功能实现
    end
endmodule

```

4.只读存储器

```

module rom8x4(addr, data);
    input [2:0] addr; // 地址
    output [3:0] data; // 地址addr处存储的数据
    reg [3:0] mem [7: 0]; // 8个4位的存储器
    initial begin // 初始化存储器
        ...
    end
    ... // 读取addr单元的值输出
endmodule

```

5.Lab3

5.1 Led[0]闪烁

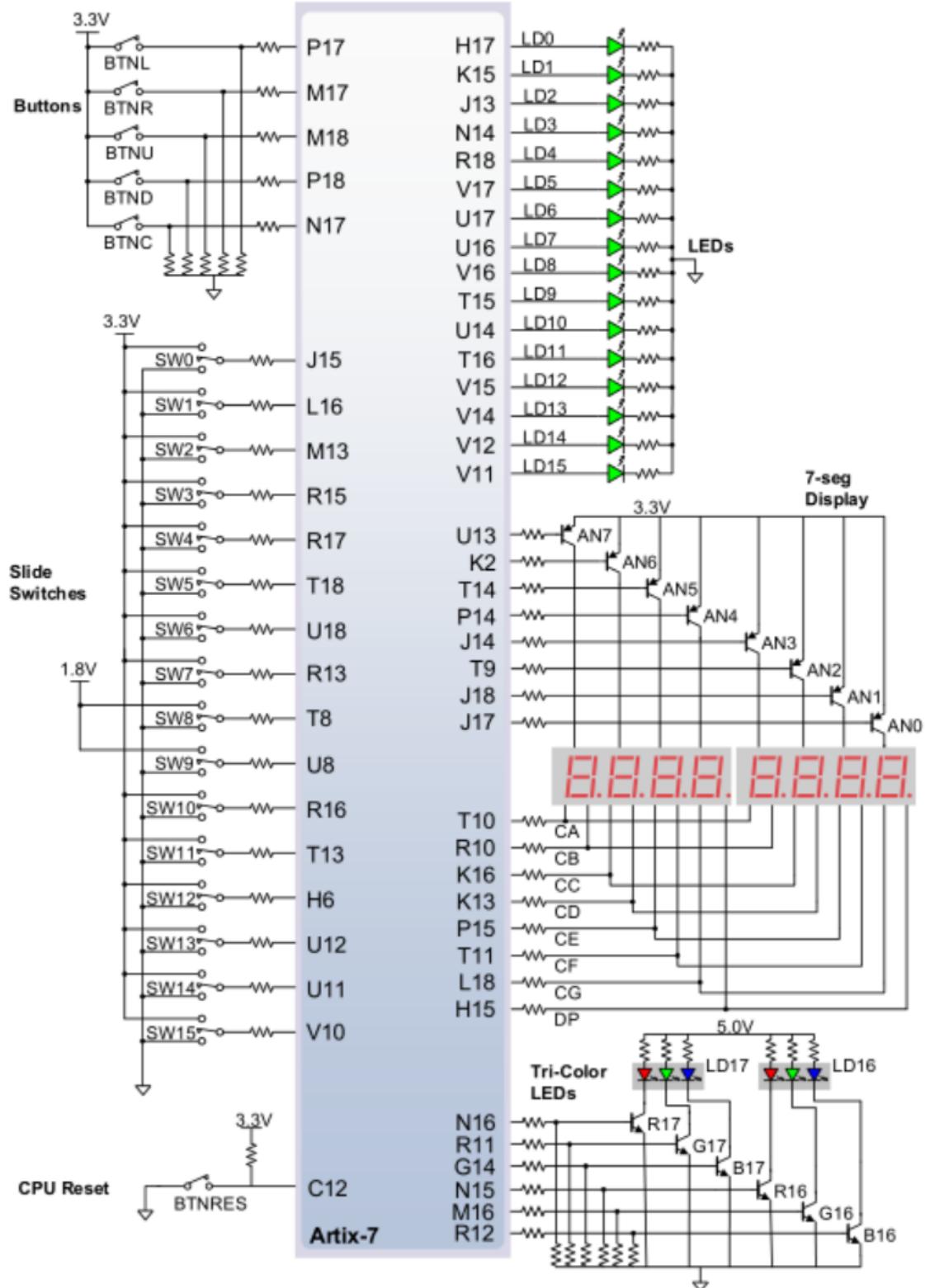
1HZ分频器：

```
module divider(clk, clk_N);
    input clk; // 系统时钟
    output reg clk_N; // 分频后的时钟
    parameter N = 50000000; // 1Hz的时钟，N=fclk/fclk_N
    reg [31:0] counter; /* 计数器变量，通过计数实现分频。当计数器从0计数到(N/2-1)时，输出时钟翻转，计数器清零 */
    always @(posedge clk) begin // 时钟上升沿
        counter <= counter+1'b1; // 功能实现
        if(counter==N)
            begin
                clk_N<=~clk_N;
                counter<=0;
            end
    end
endmodule
```

上层模块：

```
module Lab3_1(clk, LED);
    input clk;
    output [15:0] LED;
    wire clk_N;
    divider #(50000000) D1(clk, clk_N);
    assign LED[0]=clk_N;
endmodule
```

约束文件编写：



```

## This file is a general .xdc for the Nexys4 DDR Rev. C
## To use it in a project:
## - uncomment the lines corresponding to used pins
## - rename the used ports (in each line, after get_ports) according to the top
## level signal names in the project

## Clock signal
set_property -dict { PACKAGE_PIN E3      IO_STANDARD LVCMOS33 } [get_ports { clk
}]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports
{CLK100MHZ}];
## LEDs

```

```

set_property -dict { PACKAGE_PIN H17     IOSTANDARD LVCMOS33 } [get_ports { LED[0]
}]; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15     IOSTANDARD LVCMOS33 } [get_ports { LED[1]
}]; #IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13     IOSTANDARD LVCMOS33 } [get_ports { LED[2]
}]; #IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N14     IOSTANDARD LVCMOS33 } [get_ports { LED[3]
}]; #IO_L8P_T1_D11_14 Sch=led[3]
set_property -dict { PACKAGE_PIN R18     IOSTANDARD LVCMOS33 } [get_ports { LED[4]
}]; #IO_L7P_T1_D09_14 Sch=led[4]
set_property -dict { PACKAGE_PIN V17     IOSTANDARD LVCMOS33 } [get_ports { LED[5]
}]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
set_property -dict { PACKAGE_PIN U17     IOSTANDARD LVCMOS33 } [get_ports { LED[6]
}]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
set_property -dict { PACKAGE_PIN U16     IOSTANDARD LVCMOS33 } [get_ports { LED[7]
}]; #IO_L18P_T2_A12_D28_14 Sch=led[7]
set_property -dict { PACKAGE_PIN V16     IOSTANDARD LVCMOS33 } [get_ports { LED[8]
}]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
set_property -dict { PACKAGE_PIN T15     IOSTANDARD LVCMOS33 } [get_ports { LED[9]
}]; #IO_L14N_T2_SRCC_14 Sch=led[9]
set_property -dict { PACKAGE_PIN U14     IOSTANDARD LVCMOS33 } [get_ports {
LED[10] }]; #IO_L22P_T3_A05_D21_14 Sch=led[10]
set_property -dict { PACKAGE_PIN T16     IOSTANDARD LVCMOS33 } [get_ports {
LED[11] }]; #IO_L15N_T2_DQS_DOUT_CSO_B_14 Sch=led[11]
set_property -dict { PACKAGE_PIN V15     IOSTANDARD LVCMOS33 } [get_ports {
LED[12] }]; #IO_L16P_T2_CSI_B_14 Sch=led[12]
set_property -dict { PACKAGE_PIN V14     IOSTANDARD LVCMOS33 } [get_ports {
LED[13] }]; #IO_L22N_T3_A04_D20_14 Sch=led[13]
set_property -dict { PACKAGE_PIN V12     IOSTANDARD LVCMOS33 } [get_ports {
LED[14] }]; #IO_L20N_T3_A07_D23_14 Sch=led[14]
set_property -dict { PACKAGE_PIN V11     IOSTANDARD LVCMOS33 } [get_ports {
LED[15] }]; #IO_L21N_T3_DQS_A06_D22_14 Sch=led[15]

```

5.2 设计3位计数器

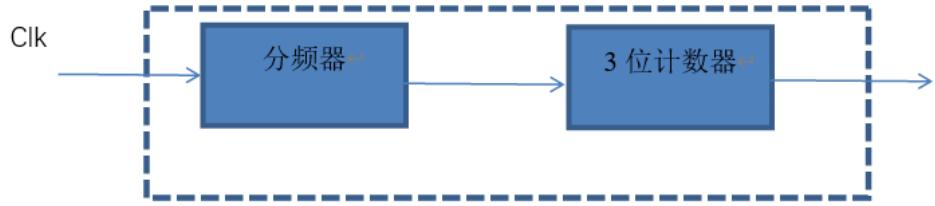
```

module counter(clk, out);
    input clk;                      // 计数时钟
    output reg [2:0] out;            // 计数值

    always @(posedge clk) begin // 在时钟上升沿计数器加1
        if(out<7) begin//计数到111B后归0
            out=<out+1'b1;// 功能实现
        end
        else begin
            out<=0;
        end
    end
endmodule

```

5.3 计数器控制LED灯闪亮



```
module Lab3_2(clk,LED);
    input clk;
    output [15:0] LED;
    wire clk_N;
    wire [2:0]out;
    divider #(50000000) D1(clk,clk_N);
    counter(clk_N,out);
    assign LED[2:0]=out[2:0];
endmodule
```

约束文件与5.1的相同

5.4 设计只读存储器 (ROM)

设计一个容量为8单元的4位只读存储器，里面存放待显示数字0、2、4、6、8、A、C、E的4位二进制编码

可在initial语句中对存储器进行初始化。

```
module rom8x4(addr, data);
    input [2:0] addr;           // 地址
    output reg [3:0] data;      // 地址addr处存储的数据

    reg [3: 0] mem [7: 0];     // 8个4位的存储器

    initial begin             // 初始化存储器
        mem[0]<=4'b0000;
        mem[1]<=4'b0010;
        mem[2]<=4'b0100;
        mem[3]<=4'b0110;
        mem[4]<=4'b1000;
        mem[5]<=4'b1010;
        mem[6]<=4'b1100;
        mem[7]<=4'b1110;
    end
    always @(addr) begin
        case(addr[2:0])// 读取addr单元的值输出
            3'b000: data[3:0]<=mem[0];
            3'b001: data[3:0]<=mem[1];
            3'b010: data[3:0]<=mem[2];
            3'b011: data[3:0]<=mem[3];
            3'b100: data[3:0]<=mem[4];
            3'b101: data[3:0]<=mem[5];
            3'b110: data[3:0]<=mem[6];
            3'b111: data[3:0]<=mem[7];
        endcase
    end
endmodule
```

约束文件:

```
## This file is a general .xdc for the Nexys4 DDR Rev. C
## To use it in a project:
## - uncomment the lines corresponding to used pins
## - rename the used ports (in each line, after get_ports) according to the top
##   level signal names in the project

## Clock signal
#set_property -dict { PACKAGE_PIN E3      IOSTANDARD LVCMOS33 } [get_ports {
CLK100MHZ }];
#IO_L12P_T1_MRCC_35 Sch=c1k100mhz
#create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports
{CLK100MHZ}];

##Switches

set_property -dict { PACKAGE_PIN J15      IOSTANDARD LVCMOS33 } [get_ports {
addr[0] }];
#IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16      IOSTANDARD LVCMOS33 } [get_ports {
addr[1] }];
#IO_L3N_T0_DQS_EMCCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13      IOSTANDARD LVCMOS33 } [get_ports {
addr[2] }];
#IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15      IOSTANDARD LVCMOS33 } [get_ports {
SW[3] }];
#IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17      IOSTANDARD LVCMOS33 } [get_ports {
SW[4] }];
#IO_L12N_T1_MRCC_14 Sch=sw[4]
set_property -dict { PACKAGE_PIN T18      IOSTANDARD LVCMOS33 } [get_ports {
SW[5] }];
#IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { PACKAGE_PIN U18      IOSTANDARD LVCMOS33 } [get_ports {
SW[6] }];
#IO_L17N_T2_A13_D29_14 Sch=sw[6]
set_property -dict { PACKAGE_PIN R13      IOSTANDARD LVCMOS33 } [get_ports {
SW[7] }];
#IO_L5N_T0_D07_14 Sch=sw[7]
set_property -dict { PACKAGE_PIN T8       IOSTANDARD LVCMOS18 } [get_ports {
SW[8] }];
#IO_L24N_T3_34 Sch=sw[8]
set_property -dict { PACKAGE_PIN U8       IOSTANDARD LVCMOS18 } [get_ports {
SW[9] }];
#IO_25_34 Sch=sw[9]
set_property -dict { PACKAGE_PIN R16      IOSTANDARD LVCMOS33 } [get_ports {
SW[10] }];
#IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
set_property -dict { PACKAGE_PIN T13      IOSTANDARD LVCMOS33 } [get_ports {
SW[11] }];
#IO_L23P_T3_A03_D19_14 Sch=sw[11]
set_property -dict { PACKAGE_PIN H6       IOSTANDARD LVCMOS33 } [get_ports {
SW[12] }];
#IO_L24P_T3_35 Sch=sw[12]
set_property -dict { PACKAGE_PIN U12      IOSTANDARD LVCMOS33 } [get_ports {
SW[13] }];
#IO_L20P_T3_A08_D24_14 Sch=sw[13]
set_property -dict { PACKAGE_PIN U11      IOSTANDARD LVCMOS33 } [get_ports {
SW[14] }];
#IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
set_property -dict { PACKAGE_PIN V10      IOSTANDARD LVCMOS33 } [get_ports {
SW[15] }];
#IO_L21P_T3_DQS_14 Sch=sw[15]

## LEDs

set_property -dict { PACKAGE_PIN H17      IOSTANDARD LVCMOS33 } [get_ports {
data[0] }];
#IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15      IOSTANDARD LVCMOS33 } [get_ports {
data[1] }];
#IO_L24P_T3_RS1_15 Sch=led[1]
```

```

set_property -dict { PACKAGE_PIN J13    IO_STANDARD LVCMOS33 } [get_ports {
data[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N14    IO_STANDARD LVCMOS33 } [get_ports {
data[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
set_property -dict { PACKAGE_PIN R18    IO_STANDARD LVCMOS33 } [get_ports { LED[4]
}]; #IO_L7P_T1_D09_14 Sch=led[4]
set_property -dict { PACKAGE_PIN V17    IO_STANDARD LVCMOS33 } [get_ports { LED[5]
}]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
set_property -dict { PACKAGE_PIN U17    IO_STANDARD LVCMOS33 } [get_ports { LED[6]
}]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
set_property -dict { PACKAGE_PIN U16    IO_STANDARD LVCMOS33 } [get_ports { LED[7]
}]; #IO_L18P_T2_A12_D28_14 Sch=led[7]
set_property -dict { PACKAGE_PIN V16    IO_STANDARD LVCMOS33 } [get_ports { LED[8]
}]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
set_property -dict { PACKAGE_PIN T15    IO_STANDARD LVCMOS33 } [get_ports { LED[9]
}]; #IO_L14N_T2_SRCC_14 Sch=led[9]
set_property -dict { PACKAGE_PIN U14    IO_STANDARD LVCMOS33 } [get_ports {
LED[10] }]; #IO_L22P_T3_A05_D21_14 Sch=led[10]
set_property -dict { PACKAGE_PIN T16    IO_STANDARD LVCMOS33 } [get_ports {
LED[11] }]; #IO_L15N_T2_DQS_DOUT_CSO_B_14 Sch=led[11]
set_property -dict { PACKAGE_PIN V15    IO_STANDARD LVCMOS33 } [get_ports {
LED[12] }]; #IO_L16P_T2_CSI_B_14 Sch=led[12]
set_property -dict { PACKAGE_PIN V14    IO_STANDARD LVCMOS33 } [get_ports {
LED[13] }]; #IO_L22N_T3_A04_D20_14 Sch=led[13]
set_property -dict { PACKAGE_PIN V12    IO_STANDARD LVCMOS33 } [get_ports {
LED[14] }]; #IO_L20N_T3_A07_D23_14 Sch=led[14]
set_property -dict { PACKAGE_PIN V11    IO_STANDARD LVCMOS33 } [get_ports {
LED[15] }]; #IO_L21N_T3_DQS_A06_D22_14 Sch=led[15]

```

5.5 设计3-8解码器和7段译码器

3-8解码器：

```

module decoder3_8(num, sel);
    input [2: 0] num;           // 数码管编号: 0~7
    output reg [7:0] sel;       // 7段数码管片选信号, 低电平有效
    always@(num) begin
        case (num[2:0])
            3'b000:sel=~8'b00000001;
            3'b001:sel=~8'b00000010;
            3'b010:sel=~8'b00000100;
            3'b011:sel=~8'b00001000;
            3'b100:sel=~8'b00010000;
            3'b101:sel=~8'b00100000;
            3'b110:sel=~8'b01000000;
            3'b111:sel=~8'b10000000;
        endcase
    end                                // 功能实现
endmodule

```

7段译码器：

```

module pattern(code, patt);
    input [3: 0] code;           // 16进制数字的4位二进制编码
    output reg [7:0] patt;       // 7段数码管驱动, 低电平有效
    always@(code) begin

```

```

    case (code[3:0])
        4'b0000:patt=8'b11000000;
        4'b0001:patt=8'b11111001;
        4'b0010:patt=8'b10100100;
        4'b0011:patt=8'b10110000;
        4'b0100:patt=8'b10011001;
        4'b0101:patt=8'b10010010;
        4'b0110:patt=8'b10000010;
        4'b0111:patt=8'b11111000;
        4'b1000:patt=8'b10000000;
        4'b1001:patt=8'b10011000;
        4'b1010:patt=8'b10001000;
        4'b1011:patt=8'b10000011;
        4'b1100:patt=8'b11000110;
        4'b1101:patt=8'b10100001;
        4'b1110:patt=8'b10000110;
        4'b1111:patt=8'b10001110;
    endcase
end
// 功能实现
endmodule

```

5.6 数码管动态显示

设计使数码管动态显示的顶层模块，实现八个数码管从右至左“分时”显示存储器中的数据
即mem[0]值在AN0显示，mem[1]值在AN1显示，.....

```

module dynamic_scan(clk, SEG, AN);
    input clk; // 系统时钟
    output [7:0] SEG; // 分别对应CA、CB、CC、CD、CE、CF、CG和DP
    output [7:0] AN; // 8位数码管片选信号
    wire clk_N;//1HZ信号
    wire [2:0]out; //计数器输出
    wire [4:0]data;//只读存储器输出
    divider #(50000000) d1(clk,clk_N);
    counter(clk_N,out); //计数器
    rom8x4(out, data); //只读存储器
    decoder3_8(out, AN); //3-8译码器
    pattern(data, SEG); //7段译码器
endmodule

```

约束文件如下：

```

## This file is a general .xdc for the Nexys4 DDR Rev. C
## To use it in a project:
## - uncomment the lines corresponding to used pins
## - rename the used ports (in each line, after get_ports) according to the top
## level signal names in the project

## Clock signal
set_property -dict { PACKAGE_PIN E3      IO_STANDARD LVCMOS33 } [get_ports { clk
}]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports
{CLK100MHZ}];

```

```

## LEDs

set_property -dict { PACKAGE_PIN H17      IOSTANDARD LVCMOS33 } [get_ports { LED[0] }
}; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15      IOSTANDARD LVCMOS33 } [get_ports { LED[1] }
}; #IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13      IOSTANDARD LVCMOS33 } [get_ports { LED[2] }
}; #IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N14      IOSTANDARD LVCMOS33 } [get_ports { LED[3] }
}; #IO_L8P_T1_D11_14 Sch=led[3]
set_property -dict { PACKAGE_PIN R18      IOSTANDARD LVCMOS33 } [get_ports { LED[4] }
}; #IO_L7P_T1_D09_14 Sch=led[4]
set_property -dict { PACKAGE_PIN V17      IOSTANDARD LVCMOS33 } [get_ports { LED[5] }
}; #IO_L18N_T2_A11_D27_14 Sch=led[5]
set_property -dict { PACKAGE_PIN U17      IOSTANDARD LVCMOS33 } [get_ports { LED[6] }
}; #IO_L17P_T2_A14_D30_14 Sch=led[6]
set_property -dict { PACKAGE_PIN U16      IOSTANDARD LVCMOS33 } [get_ports { LED[7] }
}; #IO_L18P_T2_A12_D28_14 Sch=led[7]
set_property -dict { PACKAGE_PIN V16      IOSTANDARD LVCMOS33 } [get_ports { LED[8] }
}; #IO_L16N_T2_A15_D31_14 Sch=led[8]
set_property -dict { PACKAGE_PIN T15      IOSTANDARD LVCMOS33 } [get_ports { LED[9] }
}; #IO_L14N_T2_SRCC_14 Sch=led[9]
set_property -dict { PACKAGE_PIN U14      IOSTANDARD LVCMOS33 } [get_ports {
LED[10] }];
#IO_L22P_T3_A05_D21_14 Sch=led[10]
set_property -dict { PACKAGE_PIN T16      IOSTANDARD LVCMOS33 } [get_ports {
LED[11] }];
#IO_L15N_T2_DQS_DOUT_CSO_B_14 Sch=led[11]
set_property -dict { PACKAGE_PIN V15      IOSTANDARD LVCMOS33 } [get_ports {
LED[12] }];
#IO_L16P_T2_CSI_B_14 Sch=led[12]
set_property -dict { PACKAGE_PIN V14      IOSTANDARD LVCMOS33 } [get_ports {
LED[13] }];
#IO_L22N_T3_A04_D20_14 Sch=led[13]
set_property -dict { PACKAGE_PIN V12      IOSTANDARD LVCMOS33 } [get_ports {
LED[14] }];
#IO_L20N_T3_A07_D23_14 Sch=led[14]
set_property -dict { PACKAGE_PIN V11      IOSTANDARD LVCMOS33 } [get_ports {
LED[15] }];
#IO_L21N_T3_DQS_A06_D22_14 Sch=led[15]

```

##7 segment display

```

set_property -dict { PACKAGE_PIN T10      IOSTANDARD LVCMOS33 } [get_ports { SEG[0] }
}; #IO_L24N_T3_A00_D16_14 Sch=ca
set_property -dict { PACKAGE_PIN R10      IOSTANDARD LVCMOS33 } [get_ports { SEG[1] }
}; #IO_25_14 Sch=cb
set_property -dict { PACKAGE_PIN K16      IOSTANDARD LVCMOS33 } [get_ports { SEG[2] }
}; #IO_25_15 Sch=cc
set_property -dict { PACKAGE_PIN K13      IOSTANDARD LVCMOS33 } [get_ports { SEG[3] }
}; #IO_L17P_T2_A26_15 Sch=cd
set_property -dict { PACKAGE_PIN P15      IOSTANDARD LVCMOS33 } [get_ports { SEG[4] }
}; #IO_L13P_T2_MRCC_14 Sch=ce
set_property -dict { PACKAGE_PIN T11      IOSTANDARD LVCMOS33 } [get_ports { SEG[5] }
}; #IO_L19P_T3_A10_D26_14 Sch=cf
set_property -dict { PACKAGE_PIN L18      IOSTANDARD LVCMOS33 } [get_ports { SEG[6] }
}; #IO_L4P_T0_D04_14 Sch=cg

set_property -dict { PACKAGE_PIN H15      IOSTANDARD LVCMOS33 } [get_ports { SEG[7] }
}; #IO_L19N_T3_A21_VREF_15 Sch=dp

```

```

set_property -dict { PACKAGE_PIN J17    IOSTANDARD LVCMOS33 } [get_ports { AN[0]
}]; #IO_L23P_T3_FOE_B_15 Sch=an[0]
set_property -dict { PACKAGE_PIN J18    IOSTANDARD LVCMOS33 } [get_ports { AN[1]
}]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
set_property -dict { PACKAGE_PIN T9     IOSTANDARD LVCMOS33 } [get_ports { AN[2]
}]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
set_property -dict { PACKAGE_PIN J14    IOSTANDARD LVCMOS33 } [get_ports { AN[3]
}]; #IO_L19P_T3_A22_15 Sch=an[3]
set_property -dict { PACKAGE_PIN P14    IOSTANDARD LVCMOS33 } [get_ports { AN[4]
}]; #IO_L8N_T1_D12_14 Sch=an[4]
set_property -dict { PACKAGE_PIN T14    IOSTANDARD LVCMOS33 } [get_ports { AN[5]
}]; #IO_L14P_T2_SRCC_14 Sch=an[5]
set_property -dict { PACKAGE_PIN K2     IOSTANDARD LVCMOS33 } [get_ports { AN[6]
}]; #IO_L23P_T3_35 Sch=an[6]
set_property -dict { PACKAGE_PIN U13    IOSTANDARD LVCMOS33 } [get_ports { AN[7]
}]; #IO_L23N_T3_A02_D18_14 Sch=an[7]

```

5.7进阶篇

增加三个开关，左中右；左右两个开关控制跑马灯的方向；中间开关控制停止跑动，八个数码管“同时”显示只读存储器中的数据

逆向计数器：

```

module reverse_counter(clk, out);
    input clk;                      // 计数时钟
    output reg [2:0] out;            // 计数值

    always @(posedge clk) begin // 在时钟上升沿计数器加1
        if(out>0) begin//计数到000B后归0
            out<=out-1'b1;// 功能实现
        end
        else begin
            out<=7;
        end
    end
endmodule

```

进阶动态显示数码管：

```

module dynamic_scan_advanced(clk, SW, SEG, AN);
    input clk;
    input [2:0] SW;
    output [7:0] SEG;
    output reg[7:0] AN;
    wire clk_N;
    wire clk_N1;
    wire [2:0]out1;
    wire [2:0]out2;
    wire [2:0]out3;
    reg [4:0]data;
    wire [4:0]data1;
    wire [4:0]data2;
    wire [4:0]data3;
    wire [7:0]AN1;
    wire [7:0]AN2;
    wire [7:0]AN3;

```

```

divider #(50000000) D1(clk,clk_N);
divider #(50000) D2(clk,clk_N1);
counter(clk_N,out1); //计数器
reverse_counter(clk_N, out2);
counter(clk_N1,out3);
decoder3_8(out1, AN1);
decoder3_8(out2, AN2);
decoder3_8(out3, AN3);
rom8x4(out1, data1);
rom8x4(out2, data2);
rom8x4(out3, data3);
always @(SW) begin
    if(SW==3'b001) begin
        AN[7:0]=AN1[7:0];
        data[4:0]=data1[4:0];
    end
    else if(SW==3'b100) begin
        AN[7:0]=AN2[7:0];
        data[4:0]=data2[4:0];
    end
    else if(SW==3'b010) begin
        AN[7:0]=AN3[7:0];
        data[4:0]=data3[4:0];
    end
    else begin
        AN=8'b11111111;
        data[4:0]=data1[4:0];
    end
end
pattern(data, SEG);
endmodule

```

四.简单数字电路设计

1.组合电路设计

1.1全加器

```

module full_adder(a, b, c_in, sum, c_out); //计算a+b+c_in=sum+c_out
    parameter WIDTH = 8; //本module内有效, 可用于参数传递, 即全加器的位宽
    input [WIDTH-1:0] a, b;
    input c_in;
    output [WIDTH-1:0] sum;
    output c_out;
    assign { c_out, sum } = a + b + c_in; //c_out,sum可以用{}衔接成1个WIDTH+1位数
endmodule
// 调用语法
full_adder #(4) add(x,y, ci, s,co); //调用时可以像端口信号传递一样进行参数传递
full_adder #(WIDTH(32)) add( x,y, ci, s,co);

```

1.2比较器

```

module comparator(A, B, is_equal, is_great, is_less); //比较A,B
parameter WIDTH = 8;
input [WIDTH-1:0] A, B;
output is_equal, is_great, is_less;
assign is_equal = (A==B)? 1'b1 : 1'b0;
assign is_great = (A>B)? 1'b1 : 1'b0;
assign is_less = (A<B)? 1'b1 : 1'b0;
endmodule

```

1.3译码器

译码器：是一种具有“翻译”功能的逻辑电路，能将输入二进制代码的各种状态，按照其原意翻译成对应的输出信号。译码器在数字系统中有广泛的用途，如，终端的数字显示，存贮器寻址等。

```

module decoder_38(out,in); //3-8译码器
output reg [7:0] out;
input [2:0] in;
always @(in) begin
  case(in)
    3'd0: out=8'b11111110;
    3'd1: out=8'b11111101;
    3'd2: out=8'b11111011;
    3'd3: out=8'b11110111;
    3'd4: out=8'b11101111;
    3'd5: out=8'b11011111;
    3'd6: out=8'b10111111;
    3'd7: out=8'b01111111;
  endcase
end
endmodule

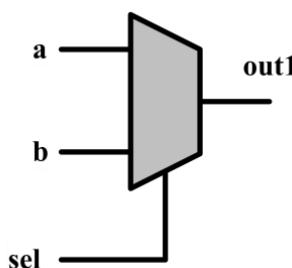
```

1.4选择器

```

module mux2_1(out1, a, b, sel) ;//2路选择器
parameter WIDTH = 8;
output [WIDTH-1:0] out1;
input [WIDTH-1:0] a, b;
input sel;
assign out1 = sel == 0: a? b ;
endmodule

```



```

module MUX4_1(out, in0, in1, in2, in3, sel);
output out;
input in0,in1,in2,in3;
input[1:0] sel;

```

```

reg out;
always @ (in0 or in1 or in2 or in3 or sel) begin
    case(sel)
        2'b00: out=in0;
        2'b01: out=in1;
        2'b10: out=in2;
        default: out=in3;
    endcase
end
endmodule

```

同理也可以用if-else语句实现4路选择器：

```

module MUX4_1(out, in0, in1, in2, in3, sel);
    output out;
    input in0, in1, in2, in3;
    input[1:0] sel;
    reg out;
    always @ (in0 or in1 or in2 or in3 or sel) begin
        if (sel==2'b00) out=in0;
        else if (sel==2'b01) out=in1;
        else if (sel==2'b10) out=in2;
        else out=in3;
    end
endmodule

```

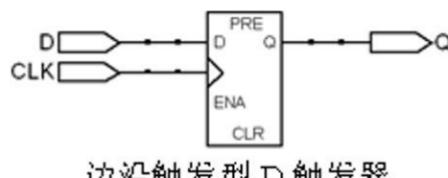
if-else和case的区别：

- ①case语句各个分支之间是平行的，优先级相同；if语句则具有优先级。
- ②每个else if 都可以判断不同的条件，比较灵活；case语句比较的是一个公共的控制表达式。
- ③if-else结构速度较慢；case结构速度较快

2.时序电路设计

2.1触发器

1) 基本D触发器： **触发器（Flip-flop）是具有记忆功能的二进制存储器，由时钟信号的跳变沿触发“存储”**



D 触发器时序波形

```

module DFF1(CLK,D,Q); //D触发器
    output Q;
    input CLK,D;
    reg Q;
    always@(posedge CLK) begin//上升沿触发
        Q<=D;
    end
endmodule

```

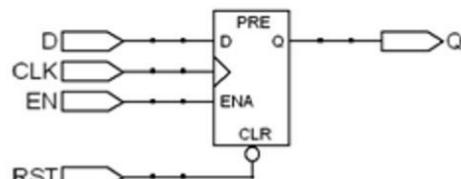
①`<=`: 非阻塞赋值, 使用在时序逻辑电路中

②`=`: 阻塞赋值, 用于组合逻辑电路

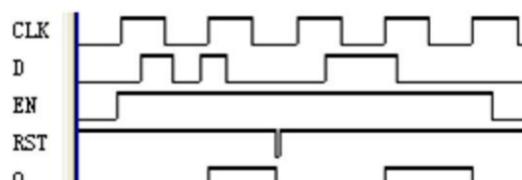
2) 含异步复位和时钟使能的D触发器:

①异步复位: 复位信号不受时钟控制, 通常“低电平”有效, 复位的优先级是最高

②使能信号: “启用”信号, 它有效时 (高电平), 触发器才工作



含使能和复位控制的 D 触发器



D 触发器的时序图

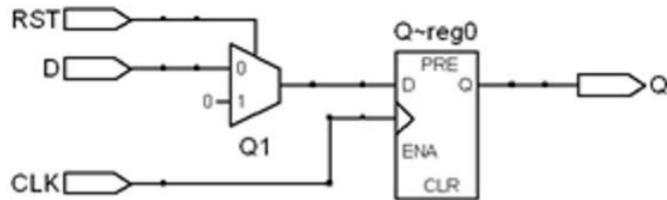
```

module DFF2(CLK,D,Q,RST,EN); //RST为复位端, EN为使能端
    output Q;
    input CLK,D,RST,EN;
    reg Q;
    always @(posedge CLK or negedge RST) begin//RST的下降沿对应RST由高电平到低电平,
即RST有效, 此时要进行复位
        if(!RST) Q<=0;//复位优先级高
        else if(EN) Q<=D;
    end
endmodule

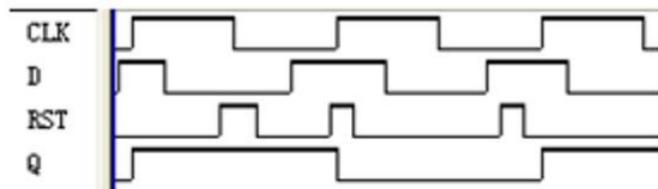
```

3) 含同步复位的D触发器

同步复位: 复位受时钟控制, 在时钟信号跳边沿检测复位信号



含同步清 0 控制的 D 触发器

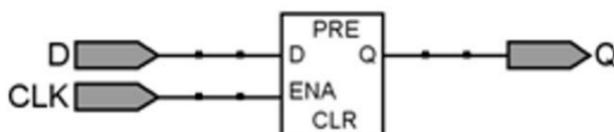


```
module DFF3(CLK,D,RST); //此时 RST高电平有效
  output Q;
  input CLK,D,RST;
  reg Q;
  always @ (posedge CLK) begin
    if(RST==1) Q<=0;
    else if(RST==0) Q<=D;
  end
endmodule
```

2.2 锁存器

1) 基本锁存器：

- ① 锁存器（Latch）也是具有记忆功能的二进制存储器
- ② 由电平信号触发“存储”



锁存器模块



锁存器的时序波形

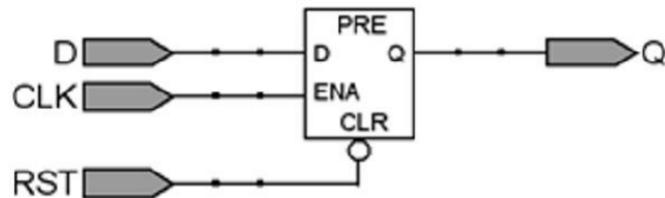
```
module LATCH1(CLK,D,Q);
  output Q;
  input CLK,D;
  reg Q;
  always@ (D or CLK) begin //此时 CLK为电平信号
    if(CLK) Q=D; // CLK为高电平
  end
endmodule
```

CLK 为高时，数据可以通过，输出端随输入而变，即存入新的数据

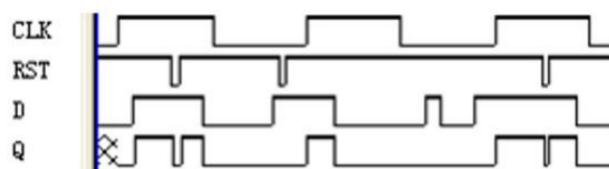
CLK 为低时，输出端被锁定，即记忆刚才存入的数据

CLK 被称为 LE (latch enable)

2) 含异步复位的锁存器：RST低电平有效



含异步清 0 的锁存器

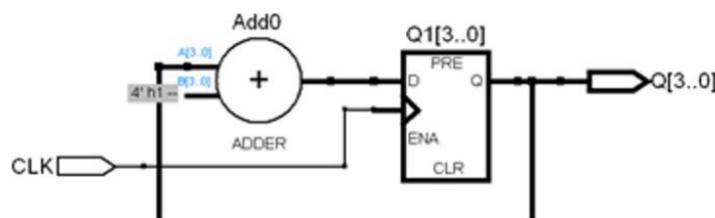


含异步清 0 的锁存器的仿真波形

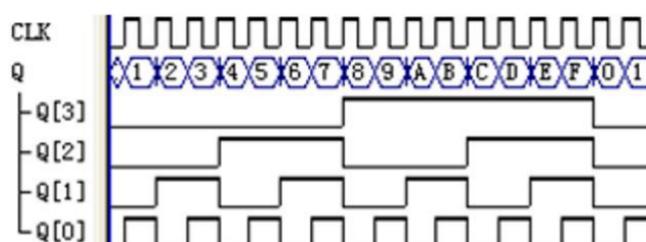
```
module LATCH2(CLK,D,Q,RST); //RST为异步复位信号
output Q;
input CLK,D,RST;
reg Q;
always @(D or CLK or RST) begin
  if(!RST) Q<=0;
  else if(CLK) Q<=D;
end
endmodule
```

2.3计数器

1) 简单加法计数器



4 位加法计数器 RTL 电路图



4 位加法计数器工作时序

```

module CNT1(CLK,Q);
    output [3:0]Q;
    input CLK;
    reg [3:0]Q;
    always@(posedge CLK) begin
        Q<=Q+1;
    end
endmodule

```

2) 实用加法计数器

- ①异步复位 RST, “低电平”有效 (RST_n)
- ②使能 EN
- ③数据加载 LOAD: 预置一个计数初值data, “低电平”有效
- ④计数初值DATA:
- ⑤计数溢出 COUT: 计数满时的输出信号

```

module CNT0(CLK,RST,EN,LOAD,COUT,DOUT,DATA); //模10计数器
    input CLK,RST,EN,LOAD;//时钟, 时钟使能, 复位, 数据加载控制信号
    input [3:0]DATA;//4位并行加载数据
    output [3:0]DOUT;//4位计数输出
    output COUT;//计数满信号
    reg [3:0]Q1;
    reg COUT;
    assign DOUT=Q1;//将内部寄存器计数结果输出至DOUT
    always@(posedge CLK or negedge RST) begin
        if(!RST) Q1<=0;//RST低电平有效, 优先级最高, 异步清零
        else if(EN) begin//使能端高电平时计数器正常工作
            if(!LOAD) Q1<=DATA;//数据加载
            else if(Q1<9) Q1<=Q1+1;//Q1<9时继续计数
            else Q1<=4'b0000;//计数到9则归0
        end
    end
    always@(Q1) begin//组合过程
        if(Q1==4'h9) COUT=1'b1;
        else COUT=1'b0;//用于重新计数后清0
    end
endmodule

```

2.4寄存器

用来暂存数据的存储单元，用触发器来实现

```

module register(clk, rst_n, en, d, q); //clk为时钟, rst_n为复位端, en为使能端, d为输入, q为寄存器输出
    parameter WIDTH = 8;
    input clk, rst_n, en;
    input [WIDTH-1:0] d;
    output reg [WIDTH-1:0] q;
    always @(posedge clk) begin
        if (!rst_n) q <=0;//rst_n低电平有效
        else if (en) q <= d;//使能端作用时才将d输出到q
    end
endmodule

```

2.5存储器

- ①用来存放数据的若干存储单元如：RAM， ROM
- ②有地址信号 addr
- ③声明 reg 型数组：数组大小size 取决于地址宽度addr_width: size=2^{addr_width}

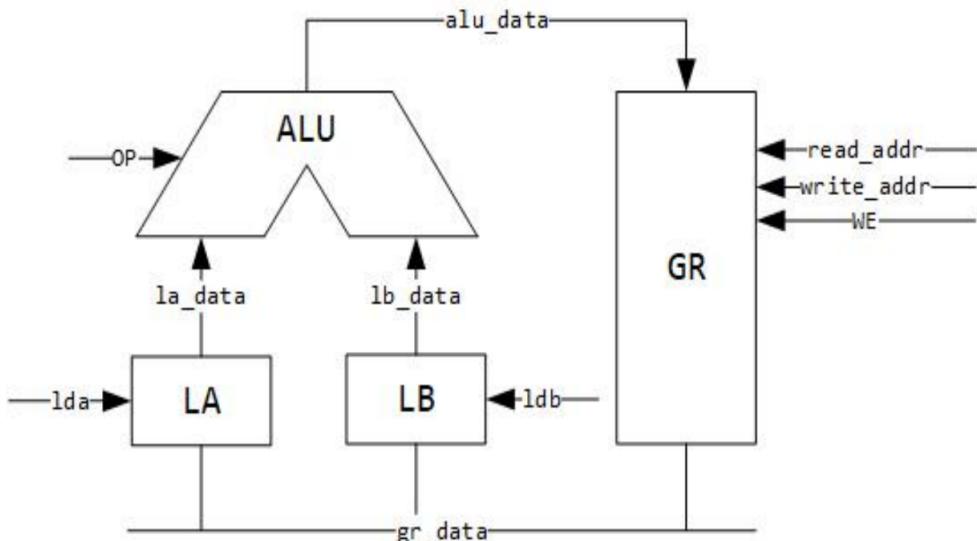
```
//n 为数据宽度, m为地址宽度
reg[n-1:0] mem[2**m-1:0]; // 2^m个n位的存储器
```

双端口存储器：同一个存储器具有两组相互独立的读写控制电路

```
module dual_port_ram( data, read_addr, write_addr, clk, we, q)
parameter DATA_WIDTH=8; //数据位宽
parameter ADDR_WIDTH=6; //地址位宽
input [DATA_WIDTH-1:0] data; //要处理的数据
input [ADDR_WIDTH-1:0] read_addr, write_addr; //读地址和写地址
input we, clk; //we为写信号端, 高电平有效
output reg [DATA_WIDTH-1:0] q;
reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0]; //存储寄存器
initial
$readmemh("ram_init.txt", ram); //从文件中读取数据到存储器ram
always @ (posedge clk) begin
if (we) ram[write_addr] <= data; //进行数据写入
q <= ram[read_addr]; //读出数据到q
end
endmodule
```

3. 数据通路

1) 数据通路示例



```
module alu_circuit(a,b,op,result); //运算器部件, 计算a op b=result, op为运算方式
parameter WIDTH=8;
input [WIDTH-1:0] a,b; //参与运算的数据
input [1:0] op; //运算方式
output reg [WIDTH-1:0] result;
always@(*) begin //always @ (*)代表所有输入信号
case(op) //根据op确定计算方式
2'b00: result=a+b;
2'b01: result=a&b;
2'b10: result=a^b;
```

```

    2'b11:result=a|b;
    default:result=0;
endcase
end
endmodule

```

```

module datapath_top(clk,rst,lda,ldb,read_addr,write_addr,we,op); //32位运算数据通路
实现
    input clk,rst,we,lda,ldb; //时钟, 复位端, 读写端, LA的使能端, LB的使能端
    input [4:0] read_addr,write_addr; //存储器的读地址和写地址, 地址位宽为5
    input [1:0] op;
    //声明数据通路中的连线
    wire [31:0] la_data,lb_data,alu_data,gr_data; //寄存器LA和LB的输出数据, ALU的计算结果, 存储器中存储的运算数据
    wire lda,ldb,we;
    wire [4:0] read_addr,write_addr; //存储器的读地址和写地址
    wire [1:0] op;

    register #(32) LA(clk,rst,lda,gr_data,la_data); //实例化寄存器LA
    register #(32) LB(clk,rst,ldb,gr_data,lb_data); //实例化寄存器LB
    //实例化32位数据位宽, 5地址位宽存储器
    dual_port_ram #(32, 5) GR(alu_data,read_addr, write_addr, clk, we, gr_data);
    alu_circuit #(32) ALU(la_data,lb_data,op,alu_data); //实例化运算器

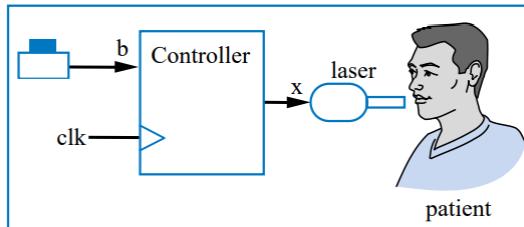
```

4.有限状态机

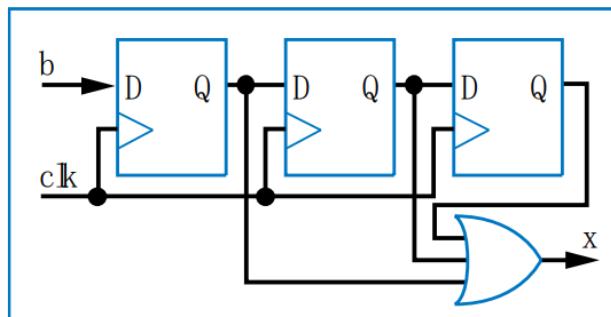
FSM: Finite State Mach

1) 激光计时器

要求: 按1次按钮b, x输出3个周期宽的1



方案1: 使用3个级联的D触发器和1个或门。

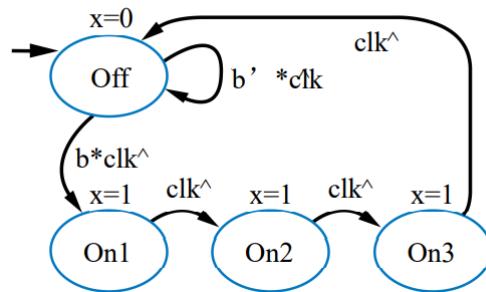


方案2: 使用有限状态机, **FSM是一种用来设计时序逻辑的通用模型**

①有限状态机的要素: **现态、条件(事件)、动作、次态; 当事件发生后, 根据当前状态, 决定执行的动作, 并设置下一个状态。**

②状态机解题的两个主要步骤: 确定系统总共有几个状态; 确定状态之间的迁移过程

Inputs: b; Outputs: x



③激光计时器：

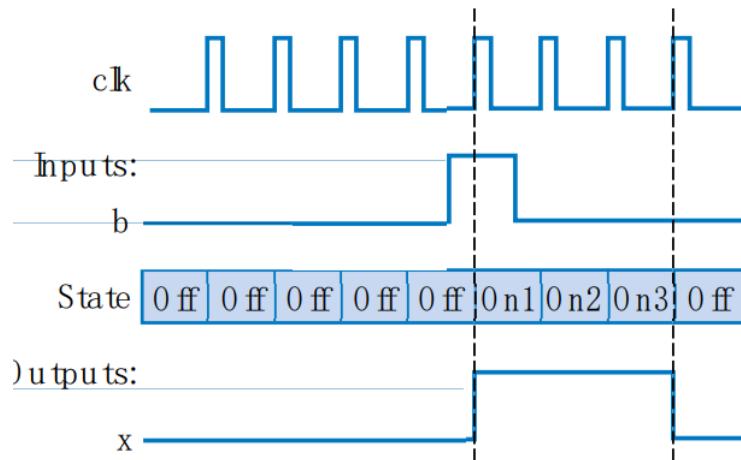
4个状态: {Off, On1, On2, On3}

b为0时“Off”态 (初始状态)

b为1时 (且时钟上升沿), 迁移到 “On1”态, x输出1

接着的2个时钟沿, 分别迁移到 “On2”, “On3”, x均输出1

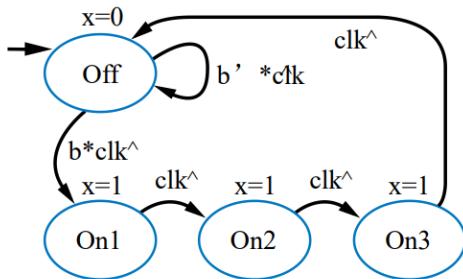
按钮b按下后, x=1保持了3个时钟周期



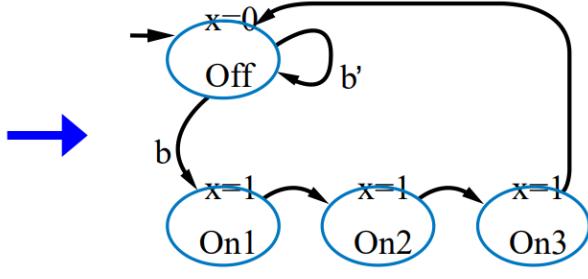
2) 基于FSM的激光计时器

①对于同步时序电路, clk^\wedge 是必然迁移条件, 因此在状态图中可以忽略 clk

Inputs: b; Outputs: x



Inputs: b; Outputs: x

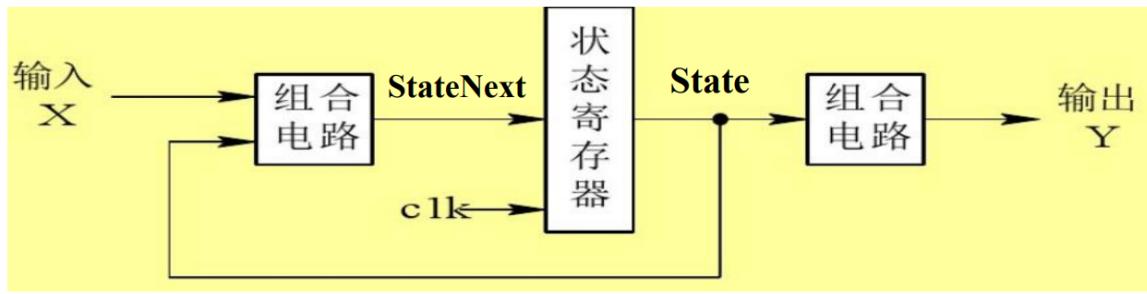


②FSM的设计涉及三方面内容：

实现状态迁移：在 clk 的上升沿将次态迁移到现态，是典型的“同步时序逻辑”。

确定下一个状态：与现态及输入条件有关，与 clk 无关，是“组合逻辑”。

确定输出：输出仅和现态有关 (Moore型FSM)，输出和现态、输入有关 (Mealy)，是“组合逻辑”。



Moore型 FSM的典型结构

3) FSM的Verilog实现

①一段式：把 FSM 的 3 部分内容写到一个always块里；

缺点：结构不清晰，不容易调试，易出错。

②两段式：用两个always块，**其中一个采用同步时序描述状态转移；另一个采用组合逻辑判断状态转移条件，描述状态转移规律和输出**。优点：结构清晰，容易调试。

③三段式：用三个always块，一个采用同步时序描述状态转移，一个采用组合逻辑判断状态转移条件，描述状态转移规律，另一个采用同步时序描述输出。

优点：相比两段式，输出毛刺少，更适用于全同步时序电路

二段式FSM的模板：**不能用initial对有限状态机进行初始化**

```
module LaserTimer(Clk,Rst,B,X); //激光计数器
    input Clk, Rst; //时钟与复位端
    input B; //按钮B的状态
    output reg X; //输出的信号输出
    localparam Off = 0, On1 = 1, On2 = 2, On3 = 3; //定于本模块中的局部参数，局部参数不可用于参数传递
    reg [1:0] State, StateNext; //2位宽的State和StateNext记录现态和次态
    // 组合逻辑判断状态转移条件，描述状态转移规律和输出
    always @(State, B) begin
        case (State)
            off: begin
                X = 0; // 输出
                if (B == 0) //状态转移条件
                    StateNext = Off;
                else
                    StateNext = On1;
            end
            On1: begin
                X = 1;
                StateNext = On2;
            end
            On2: begin
                X = 1;
                StateNext = On3;
            end
            On3: begin
                X = 1;
                StateNext = Off;
            end
        endcase
    end
    // 时序逻辑描述状态转移
    always @(posedge Clk) begin
        if (Rst == 1) //同步复位使状态初始化
            State <= Off; //FSM的初始态
    end
endmodule
```

```

    else
        State <= StateNext; //状态转移
    end
endmodule

```

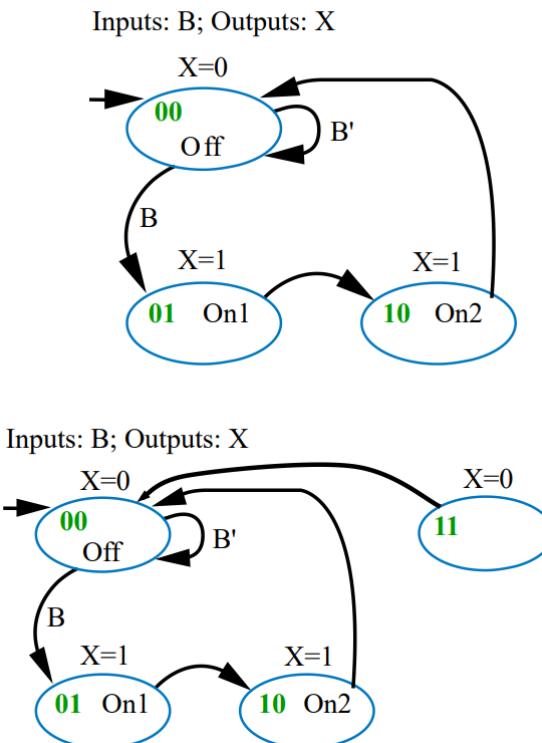
4) 安全的 FSM : 如果进入非法状态, 可以自动转移到合法状态。

示例: 如图状态机只有三个状态

①2-bit 状态码, 其中11是非法状态 (不可达状态)

②正常情况下, 状态机不会进入该状态, 但是噪声等干扰会引起误码, 0->1 或 1->0

安全设计: 能够从错误中恢复, 转移到合法状态



5.Lab4

5.1 数据通路设计

根据图1给出的数据通路 (图中SUM和NEXT是寄存器, Memory是存储器, +是加法器, ==0是比较器, 其它则是多路选择器), 完成相应的Verilog程序设计。具体要求如下:

①图中数据线的宽度和各个器件的数据线宽度初始设计时均为8位, 要求构成数据通路时可以扩充至16位或者是32位;

(利用parameter)

②设计的数据通路能够正确综合, Vivado所示的电路原理图与图1给出的一致。

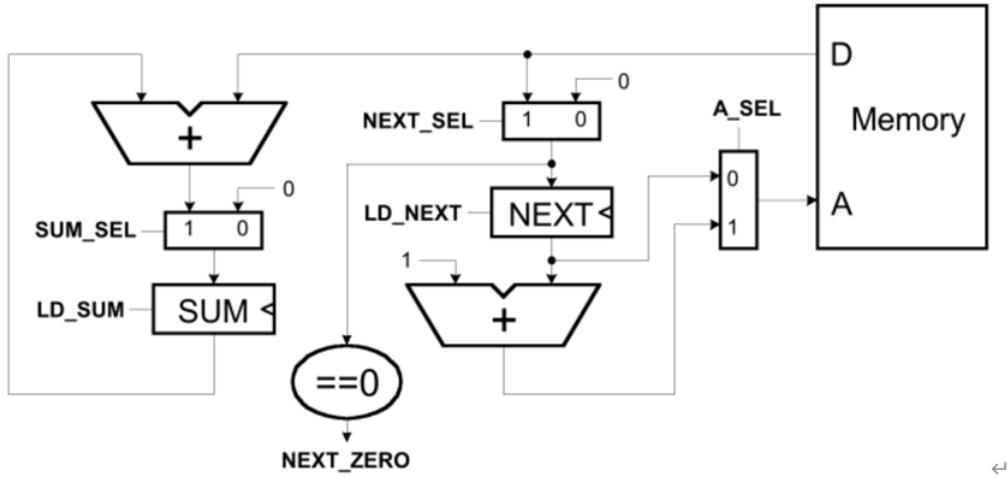
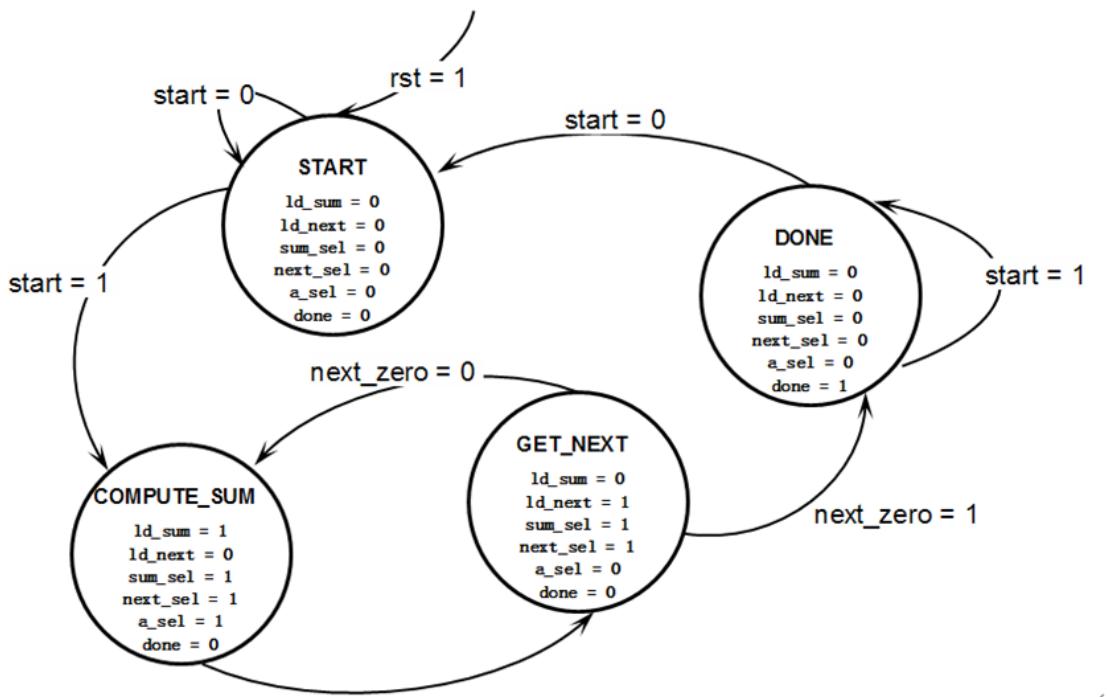


图1 数据通路图

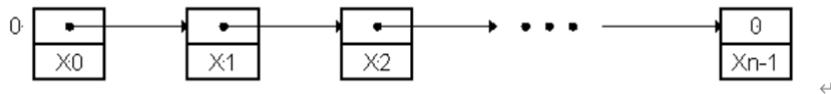
对于数据通路的分析，可以综合第二题的FSM：



提示：

- 1) 分别设计n位加法器模块，n位2选1多路选择器模块，n位比较器模块。（用parameter传参来扩展）
- 2) 设计一个含同步复位rst和加载load端的n位寄存器模块
当load=1时，对输入的n位数据进行同步寄存，即让输入D的值赋给输出Q；
- 3) 设计一个n位存储器模块，存储器中存放有一条至少包含3个节点的数据链表（如图3所示），链表第1个节点在0号地址。存放的链表可以参考后面所示的文本文件用系统函数\$readmemh进行初始化，也可以自行设计数据。
- 4) 根据图1实例化以上模块完成数据通路模块的设计
输入端口有：时钟clk，复位rst，加载信号LD_SUM, LD_NEXT,
输出端口有：链尾标志NEXT_ZERO

链表的各个节点在存储器中不是连续存放，各节点的第一个地址存放下一个节点的地址，各节点的第二个地址中存放着要进行求和运算的数据，当下一个节点的地址为0时，表示到达链表的结尾，求和运算结束。



5.1.1 n位加法器模块

该加法器模块不考虑进位

```
module add(a, b,sum); //计算a+b=sum
parameter WIDTH = 8; //本module内有效, 可用于参数传递, 即全加器的位宽
input [WIDTH-1:0] a, b;
output [WIDTH-1:0] sum;
assign sum = a + b;
endmodule
```

5.1.2 n位2选1多路选择器模块

```
module mux2_1(out, a, b, sel) ;//2路选择器
parameter WIDTH = 8;
output [WIDTH-1:0] out;
input [WIDTH-1:0] a, b;
input sel;
assign out = sel == 0? a: b ;//sel=0时, 输出a, 否则输出b
endmodule
```

5.1.3 n位比较器模块

```
module comparator(A, B, is_equal); //比较A,B,相等时is_equal=1, 否则is_equal=0
parameter WIDTH = 8;
input [WIDTH-1:0] A, B;
output is_equal;
assign is_equal = (A==B)? 1'b1 : 1'b0;
endmodule
```

5.1.4 n位寄存器模块

load=1时, 对输入的n位数据进行同步寄存, 即让输入D的值赋给输出Q (load端即为使能端)

```
module register(clk, rst_n, load, d, q); //clk为时钟, rst_n为复位端, en为使能端, d为输入, q为寄存器输出
parameter WIDTH = 8;
input clk, rst_n, load;
input [WIDTH-1:0] d;
output reg [WIDTH-1:0] q;
initial begin
    q=0;
end
always @(posedge clk) begin
    if (rst_n) q <=0; //rst_n高电平有效
    else if (load) q <= d; //使能端作用时才将d输出到q
end
endmodule
```

5.1.5 n位存储器模块

```

module dual_port_ram( read_addr, clk, data);
    parameter DATA_WIDTH=8;//数据位宽
    parameter ADDR_WIDTH=3;//地址位宽
    input [DATA_WIDTH-1:0] read_addr;//读地址, 其有效位宽只有ADDR_WIDTH位
    input clk;
    output reg [DATA_WIDTH-1:0] data ;//输出端
    reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];//存储寄存器
    initial
        $readmemh("ram_init.txt", ram); //从文件中读取数据到存储器ram
    always @ (negedge clk) begin
        data <= ram[read_addr];//读出数据到data
    end
endmodule

```

5.1.6 数据通路

```

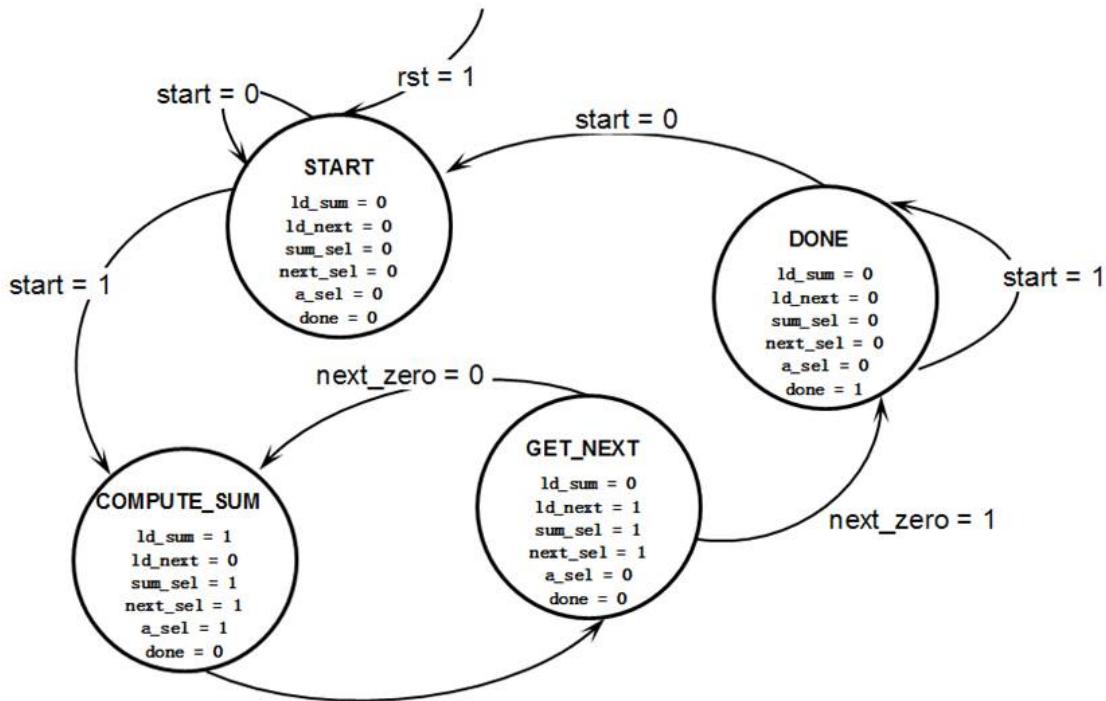
module
Datapath(CLK,RST,LD_SUM,LD_NEXT,SUM_SEL,NEXT_SEL,A_SEL,NEXT_ZERO,SUM_OUT);
//参数
parameter DATA_WIDTH=32;//数据位宽
parameter ADDR_WIDTH=4;//地址位宽
//输入输出
input CLK,RST,LD_SUM,LD_NEXT,SUM_SEL,NEXT_SEL,A_SEL;
output NEXT_ZERO;//链尾标志
output [DATA_WIDTH-1:0] SUM_OUT;//求和结果
//线路
wire [DATA_WIDTH-1:0] memory_data;//存储器输出数据
wire [DATA_WIDTH-1:0] sum_data,next_data,add1_data,add2_data;
wire [DATA_WIDTH-1:0] mux1_data,mux2_data,mux3_data;
//初始化
initial begin
    memory_data=0;
    sum_data=0;next_data=0;add1_data=0;add2_data=0;
end
//实例化
dual_port_ram #(DATA_WIDTH,ADDR_WIDTH) Memory( mux3_data, CLK, memory_data);
add #(DATA_WIDTH) ADD1(sum_data,memory_data,add1_data);
add #(DATA_WIDTH) ADD2(next_data,1,add2_data);
mux2_1 #(DATA_WIDTH) MUX1(mux1_data, 0, add1_data, SUM_SEL);
mux2_1 #(DATA_WIDTH) MUX2(mux2_data, 0, memory_data, NEXT_SEL);
mux2_1 #(DATA_WIDTH) MUX3(mux3_data, next_data, add2_data, A_SEL);
register #(DATA_WIDTH) SUM(CLK, RST, LD_SUM, mux1_data, sum_data);
register #(DATA_WIDTH) NEXT(CLK, RST, LD_NEXT, mux2_data, next_data);
comparator #(DATA_WIDTH) CMP(mux2_data, 0, NEXT_ZERO);
assign SUM_OUT=sum_data;
endmodule

```

5.2 有限状态机设计

设有限状态机的状态转移图如图2所示。根据状态转移图, 按照有限状态机 (FSM) 标准的实现模式来编写Verilog程序代码。

如果要在数据通路中完成图3所示的数据链表的求和运算, 需要一个控制器, 用我们课上讲的有限状态机(FSM)来实现该控制器。



【提示】该控制器模块的端口有：

输入端口：时钟clk，复位rst，启动求和start，链尾标志next_zero

输出端口：LD_SUM, LD_NEXT, , 求和结束done

```

module
FSM(clk,rst,start,next_zero,LD_SUM,LD_NEXT,SUM_SEL,NEXT_SEL,A_SEL,done); //控制器
有限状态机
//输入输出
input clk,rst,start,next_zero;
output reg LD_SUM,LD_NEXT,SUM_SEL,NEXT_SEL,A_SEL,done;
//状态标号
localparam START = 0, COMPUTE_SUM = 1,GET_NEXT = 2, DONE = 3;
//现态与次态记录
reg [1:0] State, StateNext;//2位宽的state和StateNext记录现态和次态
//时序逻辑实现状态转移规律和输出
always@(State,start,next_zero) begin
    case(State)
        START: begin
            LD_SUM=0;LD_NEXT=0;SUM_SEL=0;NEXT_SEL=0;A_SEL=0;done=0;
            if (start==0)
                StateNext=START;
            else
                StateNext=COMPUTE_SUM;
        end
        COMPUTE_SUM: begin
            LD_SUM=1;LD_NEXT=0;SUM_SEL=1;NEXT_SEL=1;A_SEL=1;done=0;
            StateNext=GET_NEXT;
        end
        GET_NEXT: begin
            LD_SUM=0;LD_NEXT=1;SUM_SEL=1;NEXT_SEL=1;A_SEL=0;done=0;
            if (next_zero==0)
                StateNext=COMPUTE_SUM;
            else
                StateNext=DONE;
        end
        DONE: begin

```

```

LD_SUM=0;LD_NEXT=0;SUM_SEL=0;NEXT_SEL=0;A_SEL=0;done=1;
if (start==0)
    StateNext=START;
else
    StateNext=DONE;
end
endcase
end
// 时序逻辑描述状态转移
always @(posedge clk) begin
    if (rst == 1) //同步复位使状态初始化
        State <= START; //FSM的初始态
    else
        State <= StateNext; //状态转移
end
endmodule

```

Test_bench

```

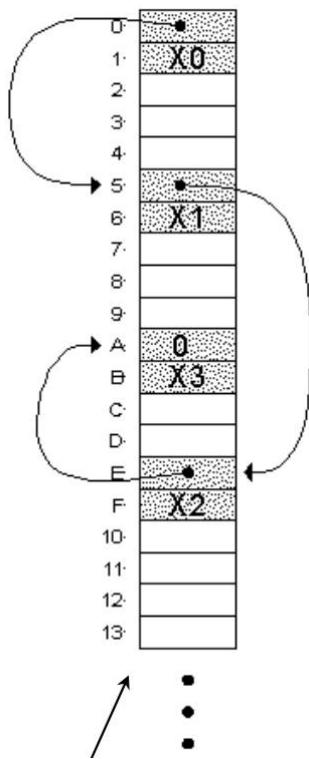
`timescale 1ns / 1ps
module FSM_tb();//测试模型
    reg clk,rst,start,next_zero;//时钟, 复位, 开始信号
    wire LD_SUM,LD_NEXT,SUM_SEL,NEXT_SEL,A_SEL,done;
    FSM(clk,rst,start,next_zero,LD_SUM,LD_NEXT,SUM_SEL,NEXT_SEL,A_SEL,done);
    //产生时钟脉冲
    initial clk=0;
    always #5 clk=~clk;//时钟信号
    //测试模拟
    initial begin
        rst=0;start=0;next_zero=0;
        LD_SUM=0;LD_NEXT=1;SUM_SEL=1;NEXT_SEL=1;A_SEL=0;done=0;
        //复位测试
        #5 rst=1;//进行复位
        #12 rst=0;//解除复位
        //开始状态转移
        #20 start=1;//START->COMPUTE_SUM
        #50 next_zero=1;//GET_NEXT->DONE
        #80 start=0;//DONE->START
    end
endmodule

```

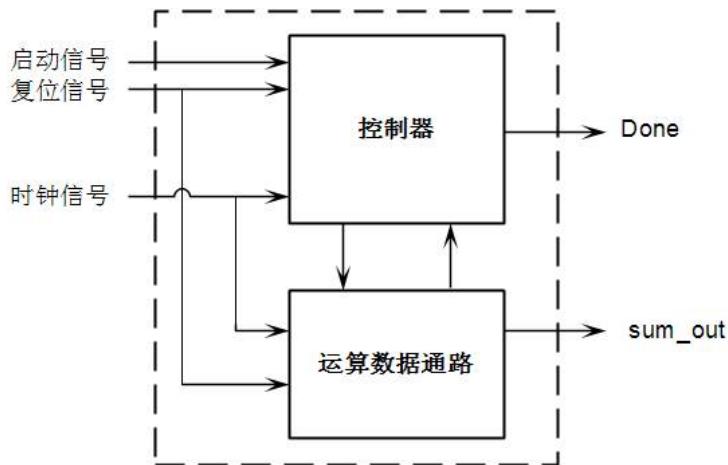
5.3 自动运算电路的设计

在存储器中存放的数据链表其结构如下图3所示，链表的各个节点在存储器中不是连续存放，各节点的第一个地址存放下一个节点的地址，各节点的第二个地址中存放着要进行求和运算的数据，当下一个节点的地址为0时，表示到达链表的结尾，求和运算结束。

存储的链表如图所示：



模块的结合可以参考如下：



因为要对求和结果进行输出，因此为数据通路添加一个新的输出端。

```

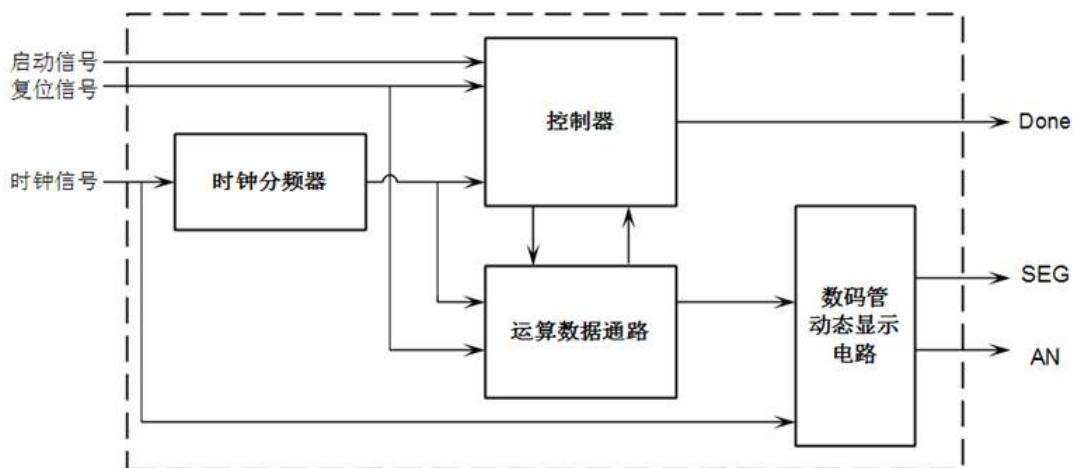
module Auto_Cal(CLK,RST,START,DONE,SUM_OUT); //自动运算电路
  //参数
  parameter DATA_WIDTH=32; //数据位宽
  parameter ADDR_WIDTH=4; //地址位宽
  //输入输出
  input CLK,RST,START; //时钟信号, 复位信号, 启动信号
  output DONE; //运算完成标志
  output [DATA_WIDTH-1:0] SUM_OUT;
  //线路
  wire NEXT_ZERO,LD_SUM,LD_NEXT,SUM_SEL,NEXT_SEL,A_SEL;
  //实例化
  FSM
  Controller(CLK,RST,START,NEXT_ZERO,LD_SUM,LD_NEXT,SUM_SEL,NEXT_SEL,A_SEL,DONE);
  Datapath #(DATA_WIDTH,ADDR_WIDTH)
  Cal(CLK,RST,LD_SUM,LD_NEXT,SUM_SEL,NEXT_SEL,A_SEL,NEXT_ZERO,SUM_OUT);
endmodule

```

5.4 带数码管显示的自动运算电路

将上面设计实现的自动运算电路配上实验2、3设计的数码管动态显示电路，设计一个能够通过数码管显示求和结果的电路，为了能够看清每步求和的中间结果，需要调慢数据通路和控制器的时钟让运算速度变慢（建议分频器的输出时钟为1Hz），该电路的总体框架如图5所示。具体要求如下：

- 完成带数码管显示的自动运算求和电路的设计，能够正确综合；
- 修改约束文件、生成比特流文件；
- 下载到Nexys4 DDR开发板上运行，**下载后拨动一个开关开始求和运算，每步的求和结果在数码管上正确显示，整个求和运算结束后一个LED指示灯亮，表明运算完成，数码管上显示最终的和，整个求和运算步数不能少于3次。**
- 如果能够用10进制显示求和结果更好。



5.4.1 二进制转BCD码模块

首先要将SUM_OUT由二进制数转换为BCD码进行输出：利用逢5加3的方式

Operation	Hundreds	Tens	Units	Binary	
				F	F
HEX					
Start				1 1 1 1	1 1 1 1
Shift 1			1	1 1 1 1	1 1 1
Shift 2			1 1	1 1 1 1	1 1
Shift 3			1 1 1	1 1 1 1	1
Add 3			1 0 1 0	1 1 1 1	1
Shift 4		1	0 1 0 1	1 1 1 1	
Add 3		1	1 0 0 0	1 1 1 1	
Shift 5		1 1	0 0 0 1	1 1 1	
Shift 6		1 1 0	0 0 1 1	1 1	
Add 3		1 0 0 1	0 0 1 1	1 1	
Shift 7	1	0 0 1 0	0 1 1 1	1	
Add 3	1	0 0 1 0	1 0 1 0	1	
Shift 8	1 0	0 1 0 1	0 1 0 1		
BCD	2	5	5		

```

module BinToBCD(Bin,Hundreds,Tens,Ones); //二进制转3个BCD码
//参数
parameter DATA_WIDTH=32; //数据位宽
integer i;
//输入输出
input [DATA_WIDTH-1:0] Bin; //输入的二进制数
output reg[3:0] Hundreds,Tens,Ones; //输出的3个BCD码
always@(Bin) begin
    Hundreds=4'd0;
    Tens=4'd0;
    Ones=4'd0;
    for(i=DATA_WIDTH-1;i>=0;i=i-1)
        begin

```

```

if(Hundreds>=5)
Hundreds=Hundreds+3;
if(Tens>=5)
Tens=Tens+3;
if(Ones>=5)
Ones=Ones+3;

Hundreds=Hundreds<<1;
Hundreds[0]=Tens[3];
Tens=Tens<<1;
Tens[0]=Ones[3];
Ones=Ones<<1;
Ones[0]=Bin[i];
end
end
endmodule

```

5.4.2 3-8译码器

```

module decoder3_8(num, sel);
    input [2: 0] num;           // 数码管编号: 0~7
    output reg [7:0] sel;       // 7段数码管片选信号, 低电平有效
    always@(num) begin
        case (num[2:0])
            3'b000:sel=~8'b00000001;
            3'b001:sel=~8'b00000010;
            3'b010:sel=~8'b00000100;
            3'b011:sel=~8'b00001000;
            3'b100:sel=~8'b00010000;
            3'b101:sel=~8'b00100000;
            3'b110:sel=~8'b01000000;
            3'b111:sel=~8'b10000000;
        endcase
    end                                // 功能实现
endmodule

```

5.4.3 7段译码器

```

module pattern(code, patt);
    input [3: 0] code;          // 16进制数字的4位二进制编码
    output reg [7:0] patt;      // 7段数码管驱动, 低电平有效
    always@(code) begin
        case (code[3:0])
            4'b0000:patt=8'b11000000;
            4'b0001:patt=8'b11111001;
            4'b0010:patt=8'b10100100;
            4'b0011:patt=8'b10110000;
            4'b0100:patt=8'b10011001;
            4'b0101:patt=8'b10010010;
            4'b0110:patt=8'b10000010;
            4'b0111:patt=8'b11111000;
            4'b1000:patt=8'b10000000;
            4'b1001:patt=8'b10011000;
            4'b1010:patt=8'b10001000;
            4'b1011:patt=8'b10000011;
            4'b1100:patt=8'b11000110;
            4'b1101:patt=8'b10100001;
        endcase
    end
endmodule

```

```

        4'b1110:patt=8'b10000110;
        4'b1111:patt=8'b10001110;
    endcase
end                                // 功能实现
endmodule

```

5.4.4 计数器

计数器，用于0~2之间循环计数，控制显示器的输出，利用高频时钟实现不同数码管显示不同的数

```

module counter(clk, out);
    input clk;                      // 高频计数时钟
    output reg [2:0] out;            // 计数值

    always @(posedge clk) begin     // 在时钟上升沿计数器加1
        if(out<2) begin//计数到0010B后归0
            out<=out+1'b1; // 功能实现
        end
        else begin
            out<=0;
        end
    end
endmodule

```

5.4.5 显示器

根据计数器的不同，选择在数码管上显示不同的数

```

module display(out,Hundreds,Tens,Ones,data); //根据out选择合适的数显示
    input [2:0]out;
    input [3:0]Hundreds,Tens,Ones;
    output reg[3:0]data;
    always@(out) begin
        case(out[2:0]) // 读取addr单元的值输出
            3'b000:data=Ones;
            3'b001:data=Tens;
            3'b010:data=Hundreds;
            default:data=4'b0000;
        endcase
    end
endmodule

```

5.4.6 分频器

对时钟信号进行分频，参数50000000会获得1Hz的时钟

```

module divider(clk, clk_N);
    parameter N = 50000000;      // 1Hz的时钟，N=fclk/fclk_N
    input clk;                  // 系统时钟
    output reg clk_N;           // 分频后的时钟
    reg [31:0] counter;         /* 计数器变量，通过计数实现分频。当计数器从0计数到
(N/2-1)时，                                     输出时钟翻转，计数器清零 */
    always @(posedge clk) begin // 时钟上升沿
        counter <= counter+1'b1 ; // 功能实现
        if(counter==N)

```

```

begin
    c1k_N<=~c1k_N;
    counter<=0;
end
endmodule

```

5.4.7 数码管动态显示电路

```

module Dig_Tube_Display(SUM_OUT,CLK,SEG,AN);
//参数
parameter DATA_WIDTH=32;//数据位宽
//输入输出
input CLK;
input [DATA_WIDTH-1:0] SUM_OUT;//计算结果
output [7:0] SEG;//SEG表示输出的值, AN表示选择哪个数码管进行输出
output [7:0] AN;//片选信号
//线路
wire CLK_N;//高频时钟信号
wire [3:0]Hundreds,Tens,Ones,data;//BCD码和显示器输出数字
wire [2:0]out;//计数器输出, 该数在0-2间循环
//实例化
divider #(50000) D(CLK,CLK_N);//获得一个高频信号c1k_N
BinToBCD #(DATA_WIDTH) BCD(SUM_OUT,Hundreds,Tens,Ones);//二进制转BCD码, 不妨用
高频信号
counter(CLK_N,out);//高频计数器
decoder3_8(out, AN);//3-8译码器, 选择哪个数码管输出数字
display(out,Hundreds,Tens,Ones,data);
pattern(data, SEG);//7段译码器, 根据显示器的选择输出数字
endmodule

```

5.4.8 带数码管显示的自动运算电路

```

module Auto_Cal_with_Tube(CLK,RST,START,DONE,SEG,AN);
//输入输出
input CLK,RST,START;//时钟信号, 复位信号, 启动信号
output DONE;//运算完成标志
output [7:0]SEG;//数码管输出控制
output [7:0]AN;//片选信号
//线路
wire NEXT_ZERO,LD_SUM,LD_NEXT,SUM_SEL,NEXT_SEL,A_SEL,CLK_1HZ;
wire [31:0]SUM_OUT;
//实例化
divider #(50000000) D1(CLK,CLK_1HZ);//产生1Hz时钟信号
FSM
Controller(CLK_1HZ,RST,START,NEXT_ZERO,LD_SUM,LD_NEXT,SUM_SEL,NEXT_SEL,A_SEL,DON
E);
Datapath #(32,4)
Cal(CLK_1HZ,RST,LD_SUM,LD_NEXT,SUM_SEL,NEXT_SEL,A_SEL,NEXT_ZERO,SUM_OUT);
Dig_Tube_Display#(32) Dig_Tube(SUM_OUT,CLK,SEG,AN);//此处还用原来的时钟信号
endmodule

```

约束文件:

```

## This file is a general .xdc for the Nexys4 DDR Rev. C
## To use it in a project:

```

```

## - uncomment the lines corresponding to used pins
## - rename the used ports (in each line, after get_ports) according to the top
level signal names in the project

## Clock signal
set_property -dict { PACKAGE_PIN E3      IOSTANDARD LVCMOS33 } [get_ports { CLK
}]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports
{CLK100MHZ}];

##Switches

set_property -dict { PACKAGE_PIN J15      IOSTANDARD LVCMOS33 } [get_ports { START
}]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16      IOSTANDARD LVCMOS33 } [get_ports { RST
}]; #IO_L3N_T0_DQS_EMCCCLK_14 Sch=sw[1]

## LEDs

set_property -dict { PACKAGE_PIN H17      IOSTANDARD LVCMOS33 } [get_ports { DONE
}]; #IO_L18P_T2_A24_15 Sch=led[0]

##7 segment display

set_property -dict { PACKAGE_PIN T10      IOSTANDARD LVCMOS33 } [get_ports { SEG[0]
}]; #IO_L24N_T3_A00_D16_14 Sch=ca
set_property -dict { PACKAGE_PIN R10      IOSTANDARD LVCMOS33 } [get_ports { SEG[1]
}]; #IO_25_14 Sch=cb
set_property -dict { PACKAGE_PIN K16      IOSTANDARD LVCMOS33 } [get_ports { SEG[2]
}]; #IO_25_15 Sch=cc
set_property -dict { PACKAGE_PIN K13      IOSTANDARD LVCMOS33 } [get_ports { SEG[3]
}]; #IO_L17P_T2_A26_15 Sch=cd
set_property -dict { PACKAGE_PIN P15      IOSTANDARD LVCMOS33 } [get_ports { SEG[4]
}]; #IO_L13P_T2_MRCC_14 Sch=ce
set_property -dict { PACKAGE_PIN T11      IOSTANDARD LVCMOS33 } [get_ports { SEG[5]
}]; #IO_L19P_T3_A10_D26_14 Sch=cf
set_property -dict { PACKAGE_PIN L18      IOSTANDARD LVCMOS33 } [get_ports { SEG[6]
}]; #IO_L4P_T0_D04_14 Sch=cg

set_property -dict { PACKAGE_PIN H15      IOSTANDARD LVCMOS33 } [get_ports { SEG[7]
}]; #IO_L19N_T3_A21_VREF_15 Sch=dp

set_property -dict { PACKAGE_PIN J17      IOSTANDARD LVCMOS33 } [get_ports { AN[0]
}]; #IO_L23P_T3_FOE_B_15 Sch=an[0]
set_property -dict { PACKAGE_PIN J18      IOSTANDARD LVCMOS33 } [get_ports { AN[1]
}]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
set_property -dict { PACKAGE_PIN T9       IOSTANDARD LVCMOS33 } [get_ports { AN[2]
}]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
set_property -dict { PACKAGE_PIN J14      IOSTANDARD LVCMOS33 } [get_ports { AN[3]
}]; #IO_L19P_T3_A22_15 Sch=an[3]
set_property -dict { PACKAGE_PIN P14      IOSTANDARD LVCMOS33 } [get_ports { AN[4]
}]; #IO_L8N_T1_D12_14 Sch=an[4]
set_property -dict { PACKAGE_PIN T14      IOSTANDARD LVCMOS33 } [get_ports { AN[5]
}]; #IO_L14P_T2_SRCC_14 Sch=an[5]

```

```
set_property -dict { PACKAGE_PIN K2      IO_STANDARD LVCMOS33 } [get_ports { AN[6] }]; #IO_L23P_T3_35 sch=an[6]
set_property -dict { PACKAGE_PIN U13      IO_STANDARD LVCMOS33 } [get_ports { AN[7] }]; #IO_L23N_T3_A02_D18_14 sch=an[7]
```

五.Educoder

5.1 全加器

```
module full_adder(
    input [2:0] a, b,
    input cin,
    output cout,
    output [2:0] sum
);

// 请完成模块的设计
assign {cout,sum}=a+b+cin;

endmodule
```

5.2 多路选择器

```
module mux21(a,b,s,y);
    input a,b,s;
    output y;

    // 请在下面添加代码
    assign y=s==1?a:b;

endmodule
```

5.3 译码器

```
module sevenseg_dec(
    input [3:0]      data,
    output reg [6:0]  segments
);

    always@(data) begin
        case(data)
            4'b0000:segments=8'b0000001;
            4'b0001:segments=8'b1001111;
            4'b0010:segments=8'b0010010;
            4'b0011:segments=8'b0000110;
            4'b0100:segments=8'b1001100;
            4'b0101:segments=8'b0100100;
            4'b0110:segments=8'b0100000;
            4'b0111:segments=8'b0001111;
            4'b1000:segments=8'b0000000;
            4'b1001:segments=8'b0001100;
            4'b1010:segments=8'b0001000;
            4'b1011:segments=8'b1100000;
```

```

        4'b1100:segments=8'b1110010;
        4'b1101:segments=8'b1000010;
        4'b1110:segments=8'b0110000;
        4'b1111:segments=8'b0111000;
    endcase
end
endmodule

```

5.4 无符号数扩展

```

module mips_unsignal_extend
(
    inmediate_data ,
    unsig_extend_data
);
    input [15:0] inmediate_data;
    output [31:0] unsig_extend_data;
    assign unsig_extend_data={16'b0000000000000000,inmediate_data};
endmodule

```

5.5 有符号数扩展

```

module mips_signal_extend
(
    inmediate_data ,
    extend_data
);
    input[15:0] inmediate_data;
    output reg[31:0] extend_data;

    always@(inmediate_data) begin
        if(inmediate_data[15]==1)
            extend_data={16'b1111111111111111,inmediate_data};
        else
            extend_data={16'b0000000000000000,inmediate_data};
    end
endmodule

```

5.6 单周期MIPS CPU的控制器

序号	指令	输入操作码		输出控制信号												
		OP (6位) (10进制)	FUNC (6位) (10进制)	ALU_OP (4位) (10进制)	MemtoReg	MemWrite	ALU_SRC	RegWrite	SYSCALL	SignedExt	RegDst	BEQ	BNE	JR	JUMP	JAL
1	SLL	0	0	0					1			1				
2	SRA	0	3	1					1			1				
3	SRL	0	2	2					1			1				
4	ADD	0	32	5					1			1				
5	ADDU	0	33	5					1			1				
6	SUB	0	34	6					1			1				
7	AND	0	36	7					1			1				
8	OR	0	37	8					1			1				
9	NOR	0	39	10					1			1				
10	SLT	0	42	11					1			1				
11	SLTU	0	43	12					1			1				
12	JR	0	8	0										1	1	
13	SYSCALL	0	12	0						1						
14	J	2	X	0												1
15	JAL	3	X	0					1							1 1
16	BEQ	4	X	11							1		1			
17	BNE	5	X	11							1				1	
18	ADDI	8	X	5				1	1		1					
19	ANDI	12	X	7				1	1							
20	ADDIU	9	X	5				1	1		0					
21	SLTI	10	X	11				1	1		1					
22	ORI	13	X	8				1	1							
23	LW	35	X	5	1			1	1		1					
24	SW	1	43	X	5		1	1			1					

```

module mips_1stage_decoder(
    input [5:0]          i_op,
    input [5:0]          i_func,
    output wire [3:0]    o_alu_op,
    output wire          o_MemtoReg,
    output wire          o_MemWrite,
    output wire          o_ALU_SRC,
    output wire          o_RegWrite,
    output wire          o_SYSCALL,
    output wire          o_SignedExt,
    output wire          o_RegDst,
    output wire          o_BEQ,
    output wire          o_BNE,
    output wire          o_JR,
    output wire          o_JUMP,
    output wire          o_JAL
);
    reg[15:0] signal;//12个信号
    assign
    {o_alu_op,o_MemtoReg,o_MemWrite,o_ALU_SRC,o_RegWrite,o_SYSCALL,o_SignedExt,o_RegDst,o_BEQ,o_BNE,o_JR,o_JUMP,o_JAL}=signal;
    always@(i_op,i_func) begin
        case(i_op)
            6'b000000: begin
                case(i_func)
                    6'b000000:signal=16'b0000000100100000; //SLT
                    6'b000011:signal=16'b0001000100100000; //SAR
                    6'b000010:signal=16'b0010000100100000; //SRL
                    6'b100000:signal=16'b0101000100100000; //ADD
                    6'b100001:signal=16'b0101000100100000; //ADDU
                    6'b100010:signal=16'b0110000100100000; //SUB
                    6'b100100:signal=16'b0111000100100000; //AND
                    6'b100101:signal=16'b1000000100100000; //OR
                    6'b100111:signal=16'b1010000100100000; //NOR
                    6'b101010:signal=16'b1011000100100000; //SLT
                    6'b101011:signal=16'b1100000100100000; //SLTU
                    6'b001000:signal=16'b00000000000000110; //JR
                endcase
            endcase
        end
    end

```

```

        6'b0001100:signal=16'b0000000010000000; //SYSCALL
    endcase
end
6'b000010:signal=16'b0000000000000010; //J
6'b000011:signal=16'b0000000100000011; //JAL
6'b000100:signal=16'b1011000001010000; //BEQ
6'b000101:signal=16'b1011000001001000; //BNE
6'b001000:signal=16'b0101001101000000; //ADDI
6'b001100:signal=16'b0111001100000000; //ANDI
6'b001001:signal=16'b0101001100000000; //ADDIU
6'b001101:signal=16'b1011001101000000; //SLTI
6'b001101:signal=16'b1000001100000000; //ORI
6'b100011:signal=16'b0101101101000000; //LW
6'b101011:signal=16'b0101011001000000; //SW
endcase
end
endmodule

```

5.7 MIPS ALU设计

引脚与功能描述

引脚	输入/输出	位宽	功能描述
a_i	输入	32	操作数 a
b_i	输入	32	操作数 b
alu_op	输入	4	运算器功能码, 具体功能见下表
result_1	输出	32	ALU 运算结果
result_2	输出	32	ALU 结果第二部分, 用于乘法指令结果高位或除法指令的余数位, 其它运算时值为零
over_flow	输出	1	有符号加减运算溢出标记, 其它运算时值为零
unsig_over_flow	输出	1	无符号加减运算溢出标记, 其它运算时值为零 溢出条件(加法和小于加数, 减法差大于被减数)
equal	输出	1	equal = (a_i == b_i)? 1 : 0, 对所有运算均有效

运算符功能

alu_op	十进制	运算功能		
0000	0	result_1 = a_i << b_i	逻辑左移 (b_i 取低五位)	result_2=0
0001	1	result_1 = a_i >>>b_i	算术右移 (b_i 取低五位)	result_2=0
0010	2	result_1 = a_i >> b_i	逻辑右移 (b_i 取低五位)	result_2=0
0011	3	result_1 = (a_i * b_i)[31:0]; result_2=(a_i * b_i)[63:32]	无符号乘法	
0100	4	result_1 = a_i / b_i;	result_2= a_i % b_i	无符号除法
0101	5	result_1 = a_i + b_i	(Set OF/UOF)	
0110	6	result_1 = a_i - b_i	(Set OF/UOF)	
0111	7	result_1 = a_i & b_i	按位与	
1000	8	result_1 = a_i b_i	按位或	
1001	9	result_1 = a_i ⊕ b_i	按位异或	
1010	10	result_1 = ~ (a_i b_i)	按位或非	
1011	11	result_1 = (a_i < b_i) ? 1 : 0	符号比较	
1100	12	result_1 = (a_i < b_i) ? 1 : 0	无符号比较	

```

module mips_alu
(

```

```

    a_i          ,
    b_i          ,
    alu_op       ,
    result_1     ,
    result_2     ,
    over_flow    ,
    unsig_over_flow ,
    equal
);
input [31:0] a_i,b_i;
input [3:0] alu_op;
output reg [31:0] result_1,result_2;
output reg over_flow,unsig_over_flow,equal;
always@(a_i,b_i,alu_op) begin
  case(alu_op)
    4'b0000:begin
      result_1=a_i<<b_i[4:0];
      result_2=0;
    end
    4'b0001:begin
      result_1=$signed(a_i)>>>b_i[4:0];
      result_2=0;
    end
    4'b0010:begin
      result_1=a_i>>b_i[4:0];
      result_2=0;
    end
    4'b0011:{result_2,result_1}=a_i*b_i;
    4'b0100:begin
      result_1=a_i/b_i;
      result_2=a_i%b_i;
    end
    4'b0101:begin
      {unsig_over_flow,result_1}=a_i+b_i;
      over_flow=(a_i[31]&b_i[31]&~result_1[31])|
      (~a_i[31]&~b_i[31]&result_1[31]);
    end
    4'b0110:begin
      result_1=a_i-b_i;
      over_flow=(a_i[31]&~b_i[31]&~result_1[31])|
      (~a_i[31]&b_i[31]&result_1[31]);
      unsig_over_flow=(a_i<b_i)?0:1;
    end
    4'b0111:result_1=a_i&b_i;
    4'b1000:result_1=a_i|b_i;
    4'b1001:result_1=a_i^b_i;
    4'b1010:result_1=~(a_i|b_i);
    4'b1011:result_1=($signed(a_i)<$signed(b_i))?1:0;
    4'b1100:result_1=(a_i<b_i)?1:0;
  endcase
end
endmodule

```

5.8 时钟分频器

```

module divider (clk, clk_N);
    input      clk;
    output reg   clk_N;
    parameter   N = 50_000_000;
    reg [31:0]  counter;
    initial begin
        clk_N=0;
        counter=0;
    end
    always@(posedge clk) begin
        counter=counter+1;
        if(counter==N+1) begin
            clk_N=~clk_N;
            counter=0;
        end
    end
endmodule

```

5.9 计数器

```

module counter(
    input clk,
    input rst,
    input start,
    input up,
    output [3:0] counter_out
);

// 请完成模块的设计
always@(posedge clk) begin
    if(rst) counter_out=0;
    else if(start) begin
        if(up==1) counter_out=counter_out+1;
        else counter_out=counter_out-1;
    end
end
endmodule

```

5.10 移位寄存器

输入							输出
清零	控制信号		串行输入		时钟	并行输入	$Q_0^{n+1} \sim Q_7^{n+1}$
\overline{CR}	S1	S0	Dsr	Dsl	CP	$DI_0 \sim DI_7$	
L	x	x	x	x	x	x	L
H	L	L	x	x	x	x	$Q_0^n \sim Q_7^n$
H	L	H	L	x	↑	x	$L, Q_0^n \sim Q_6^n$
H	L	H	H	x	↑	x	$H, Q_0^n \sim Q_6^n$
H	H	L	x	L	↑	x	$Q_1^n \sim Q_7^n, L$
H	H	L	x	H	↑	x	$Q_1^n \sim Q_7^n, H$
H	H	H	x	x	↑	$DI_0^* \sim DI_7^*$	$DI_0 \sim DI_7$

注: DI_N^* 表示CP脉冲上升沿之前瞬间 DI_N 的电平

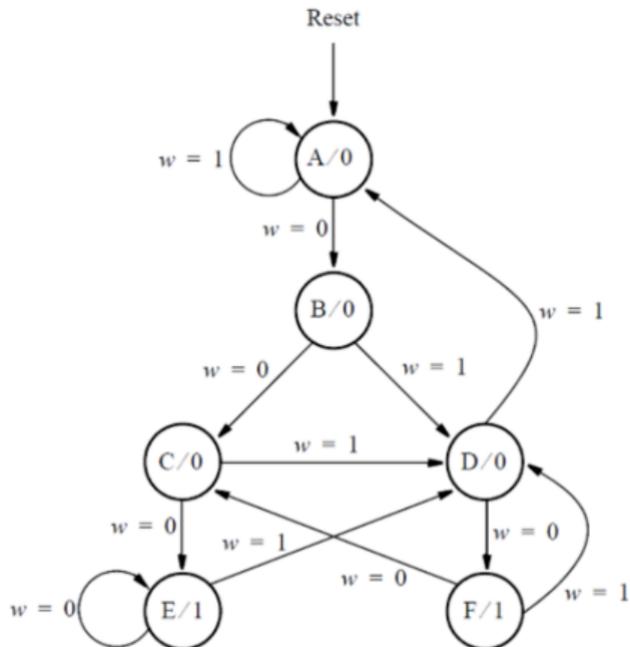
```

module shifter(din,s,srsi,slsi,clk,clr,dout);
    input [7:0] din;
    input [1:0] s;
    input srsi,slsi,clk,clr;
    output [7:0] dout;
    reg [7:0] dout;

// 请在下面添加代码，完成 8 位双向移位寄存器功能
always@(clr,s) begin
    if(!clr) dout=0;
    else if(s[1]==0&&s[0]==0) dout=dout;
end
always@(posedge clk) begin
    if(!clr) dout=0;
    else if(s[1]==0&&s[0]==0) dout<=dout;
    else if(s[1]==0&&s[0]==1) dout<={srsi,dout[6:0]};
    else if(s[1]==1&&s[0]==0) dout<={dout[7:1],slsi};
    else if(s[1]==1&&s[0]==1) dout<=din;
end
endmodule

```

5.11 FSM



```

module finite_state_machine(
    input clk,
    input reset,
    input w,
    output [2:0] state,
    output [2:0] next_state,
    output z
);

// 请完成模块的设计
localparam A=0,B=1,C=2,D=3,E=4,F=5;
always@(state,w) begin
    case(state)
        A:begin
            z=0;
            if(w==1) next_state=A;
            else next_state=B;
        end
        B:begin
            z=0;
            if(w==1) next_state=D;
            else next_state=C;
        end
        C:begin
            z=0;
            if(w==1) next_state=D;
            else next_state=E;
        end
        D:begin
            z=0;
            if(w==1) next_state=A;
            else next_state=F;
        end
        E:begin
            z=1;
            if(w==1) next_state=D;
            else next_state=E;
        end
    endcase
end

```

```

    end
  F:begin
    z=1;
    if(w==1) next_state=D;
    else next_state=C;
  end
endcase
end
always@(posedge clk) begin
  if(reset==1) state=A;
  else state=next_state;
end
endmodule

```

六. 交通信号灯

6.1 七段译码器

用于数码管显示，不过多解释

```

module seg7(in,out);
  input[3:0] in;
  output[6:0] out;
  reg[6:0] out;
  always@(in)
    case(in)
      4'd0: out = 7'b1000000;
      4'd1: out = 7'b1111001;
      4'd2: out = 7'b0100100;
      4'd3: out = 7'b0110000;
      4'd4: out = 7'b0011001;
      4'd5: out = 7'b0010010;
      4'd6: out = 7'b0000010;
      4'd7: out = 7'b1111000;
      4'd8: out = 7'b0000000;
      4'd9: out = 7'b0010000;
      4'd10: out = 7'b0001000;
      4'd11: out = 7'b0000011;
      4'd12: out = 7'b1000110;
      4'd13: out = 7'b0100001;
      4'd14: out = 7'b00000110;
      4'd15: out = 7'b0001110;
      default: out = 7'b1111111;
    endcase
endmodule

```

6.2 分频器

分频器，原本时钟频率50MHz，N=100M，频率降为50/100=0.5Hz，周期为2s，代码不作解释

```

module divclk(rst_n,clk_in,clk_out);
    input rst_n,clk_in;
    output clk_out;
    reg[26:0] cnt;
    always@(posedge clk_in or negedge rst_n)
        if(!rst_n) //复位端低电平有效
            cnt <= 27'd0;
        else if(cnt == 27'd99999999)
            cnt <= 27'd0;
        else
            cnt <= cnt+27'd1;
    assign clk_out = (cnt == 27'd99999999) ? 1'b1 : 1'b0;
endmodule

```

6.3 扫描数码管

```

module scan_seg7(clk,rst_n,com_pos);
    input clk,rst_n;           //时钟端、复位端
    output[3:0] com_pos;       //选中的位置
    reg[3:0] com_pos_inv;
    wire[3:0] com_pos;
    reg[15:0] cnt;             //用于计数的寄存器
    wire clk_div;              //时钟端：根据注释，这是一个周期为1ms的时钟端

    //*****这一部分也是个分频器*****
    always@(posedge clk or negedge rst_n) //N=50000*2=100000
        if(!rst_n)
            cnt <= 16'd0;
        else if(cnt == 16'd49999)
            cnt <= 16'd0;
        else
            cnt <= cnt+16'd1;
    assign clk_div = (cnt == 16'd49999) ? 1'b1 : 1'b0; //Tclk = 1ms
    //*****



    always@(posedge clk_div or negedge rst_n)
        if(!rst_n)
            com_pos_inv <= 4'b0001;
        else if(com_pos_inv == 4'b1000)
            com_pos_inv <= 4'b0001;
        else
            com_pos_inv <= com_pos_inv << 1'b1;
    //com_pos_inv在1ms内在0001-0010-0100-1000之间循环变化
    assign com_pos = ~com_pos_inv; //取反也许是因为低电平有效
endmodule

```

6.4 信号灯

```

module traffic_light(clk,rst_n,cnt_main_1,cnt_main_0,cnt_branch_1,cnt_branch_0,
                     light_main_g,light_main_y,light_main_r,
                     light_branch_g,light_branch_y,light_branch_r);

    input clk,rst_n;//时钟端、复位端

```

```

    output
light_main_g,light_main_y,light_main_r,light_branch_g,light_branch_y,light_branch_r;
    //主干道和支干道的红黄绿灯信号
    output[3:0] cnt_main_1,cnt_main_0,cnt_branch_1,cnt_branch_0;
    //主干道和支干道的计数器,1代表十位数,0代表个位数
    reg[5:0] cnt;    //辅助计数器

//-----信号灯控制逻辑-----
//60s循环计数
always@(posedge clk or negedge rst_n)
    if(!rst_n)
        cnt <= 6'd0;
    else if(cnt==6'd59)
        cnt <= 6'd0;
    else
        cnt <= cnt+6'd1;
//主干道(main) : green(30s),yellow(5s),red(25s)
//支干道(branch): red(35s),green(20s),yellow(5s)
assign
{light_main_g,light_main_y,light_main_r,light_branch_g,light_branch_y,light_branch_r}=
    (cnt>=0 &&cnt<=29) ? 6'b100001 :
    (cnt>=30&&cnt<=34) ? 6'b010001 :
    (cnt>=35&&cnt<=54) ? 6'b001100 :
    (cnt>=55&&cnt<=59) ? 6'b0001010 : 6'b000000 ;
//0 -29s: main:green ; branch:red
//30-34s: main:yellow; branch:red
//35-54s: main:red    ; branch:green
//55-59s: main:red    ; branch:yellow
//-----

//-----倒计时控制逻辑-----
//main绿灯倒计时:29-0,模30计数
reg[3:0] cnt_main_g_0,cnt_main_g_1;//主干道绿灯倒计时的个位与十位
always@(posedge clk or negedge rst_n)//cnt_main_g_0:9-0循环计数
if(!rst_n)
    cnt_main_g_0 <= 4'd9;
else if(cnt_main_g_0==4'd0)
    cnt_main_g_0 <= 4'd9;
else
    cnt_main_g_0 <= cnt_main_g_0-4'd1;
always@(posedge clk or negedge rst_n)//cnt_main_g_1:2-0循环计数
if(!rst_n)
    cnt_main_g_1 <= 4'd2;
else if(cnt_main_g_1==4'd0&&cnt_main_g_0==0)
    cnt_main_g_1 <= 4'd5;
else if(cnt_main_g_0==0)
    cnt_main_g_1 <= cnt_main_g_1-4'd1;
//后续原理完全相同, 不过多解释
//main红灯倒计时:24-0,模25计数
reg[3:0] cnt_main_r_0,cnt_main_r_1;//主干道红灯倒计时的个位与十位
always@(posedge clk or negedge rst_n)
if(!rst_n)
    cnt_main_r_0 <= 4'd9;
else if(cnt_main_r_0==4'd0)
    cnt_main_r_0 <= 4'd9;
else

```

```

        cnt_main_r_0 <= cnt_main_r_0-4'd1;
always@(posedge clk or negedge rst_n)
    if(!rst_n)
        cnt_main_r_1 <= 4'd5;
    else if(cnt_main_r_1==4'd0&&cnt_main_r_0==4'd0)
        cnt_main_r_1 <= 4'd5;
    else if(cnt_main_r_0==0)
        cnt_main_r_1 <= cnt_main_r_1-4'd1;
//branch红灯倒计时:34-0,模35计数
reg[3:0] cnt_branch_r_0,cnt_branch_r_1;//支干道红灯倒计时的个位与十位
always@(posedge clk or negedge rst_n)
    if(!rst_n)
        cnt_branch_r_0 <= 4'd4;
    else if(cnt_branch_r_0==4'd0)
        cnt_branch_r_0 <= 4'd9;
    else
        cnt_branch_r_0 <= cnt_branch_r_0-4'd1;
always@(posedge clk or negedge rst_n)
    if(!rst_n)
        cnt_branch_r_1 <= 4'd3;
    else if(cnt_branch_r_1==4'd0&&cnt_branch_r_0==0)
        cnt_branch_r_1 <= 4'd5;
    else if(cnt_branch_r_0==0)
        cnt_branch_r_1 <= cnt_branch_r_1-4'd1;
//branch绿灯倒计时:19-0,模20计数
reg[3:0] cnt_branch_g_0,cnt_branch_g_1;//支干道绿灯倒计时的个位与十位
always@(posedge clk or negedge rst_n)
    if(!rst_n)
        cnt_branch_g_0 <= 4'd4;
    else if(cnt_branch_g_0==4'd0)
        cnt_branch_g_0 <= 4'd9;
    else
        cnt_branch_g_0 <= cnt_branch_g_0-4'd1;
always@(posedge clk or negedge rst_n)
    if(!rst_n)
        cnt_branch_g_1 <= 4'd5;
    else if(cnt_branch_g_1==4'd0 && cnt_branch_g_0==0)
        cnt_branch_g_1 <= 4'd5;
    else if(cnt_branch_g_0==0)
        cnt_branch_g_1 <= cnt_branch_g_1-4'd1;
//branch & main黄灯倒计时:4-0,模5计数
reg[3:0] cnt5_0;//黄灯倒计时个位
wire[3:0] cnt5_1;//黄灯倒计时十位
always@(posedge clk or negedge rst_n)
    if(!rst_n)
        cnt5_0 <= 4'd4;
    else if(cnt5_0==4'd0)
        cnt5_0 <= 4'd4;
    else
        cnt5_0 <= cnt5_0-4'd1;
assign cnt5_1 = 4'd0;//4-0,倒计时十位数恒为0
//根据时间设置主干道倒计时和支干道倒计时的十位数和个位数显示
assign {cnt_main_1,cnt_main_0,cnt_branch_1,cnt_branch_0} = (cnt>=0&&cnt<=29)
?{cnt_main_g_1,cnt_main_g_0,cnt_branch_r_1,cnt_branch_r_0} :
                                         (cnt>=30&&cnt<=34) ?
{cnt5_1,cnt5_0,cnt_branch_r_1,cnt_branch_r_0} :
                                         (cnt>=35&&cnt<=54) ?
{cnt_main_r_1,cnt_main_r_0,cnt_branch_g_1,cnt_branch_g_0} :

```

(cnt>=55&&cnt<=59) ?

```
{cnt_main_r_1,cnt_main_r_0,cnt5_1,cnt5_0} : 16'd0;  
endmodule
```

6.6 top模块

```
module  
traffic_light_top(clk,rst_n,cnt,com_pos,light_main_g,light_main_y,light_main_r,  
light_branch_g,light_branch_y,light_branch_r);  
  
input clk,rst_n;//时钟端和复位端  
output[6:0] cnt;//计数器  
output[3:0] com_pos;//片选  
//主/支干道的红黄绿灯  
output  
light_main_g,light_main_y,light_main_r,light_branch_g,light_branch_y,light_branch_r;  
//主/支干道到的倒计时十位与个位  
wire[3:0] com_pos,cnt_main_1,cnt_main_0,cnt_branch_1,cnt_branch_0;  
//倒计时数码管显示对应的7段译码  
wire[6:0] cnt_main_11,cnt_main_00,cnt_branch_11,cnt_branch_00;  
reg[6:0] cnt;//辅助计数器  
//实例化分频器0，得到0.5Hz的时钟clk_o  
divclk divclk_0(.rst_n(rst_n),.clk_in(clk),.clk_out(clk_o)  
//实例化交通信号灯  
traffic_light  
traffic_light_0(.clk(clk_o),.rst_n(rst_n),.cnt_main_1(cnt_main_1),.cnt_main_0(cnt_main_0),  
.cnt_branch_1(cnt_branch_1),.cnt_branch_0(cnt_branch_0),.light_main_g(light_main_g),.light_main_y(light_main_y),.light_main_r(light_main_r),.light_branch_g(light_branch_g),.light_branch_y(light_branch_y),.light_branch_r(light_branch_r));  
//实例化主干道和支干道交通信号灯倒计时的十位与个位的7段译码  
seg7 seg7_0(.in(cnt_main_0),.out(cnt_main_00));  
seg7 seg7_1(.in(cnt_main_1),.out(cnt_main_11));  
seg7 seg7_2(.in(cnt_branch_0),.out(cnt_branch_00));  
seg7 seg7_3(.in(cnt_branch_1),.out(cnt_branch_11));  
//实例化扫描数码管模块  
scan_seg7 scan_seg7_0(.clk(clk_o),.rst_n(rst_n),.com_pos(com_pos));  
//利用cnt输出计数  
always@(com_pos or cnt_main_00 or cnt_main_11 or cnt_branch_00 or  
cnt_branch_11)  
    case(com_pos)  
        4'b1110: cnt = cnt_branch_11;  
        4'b1101: cnt = cnt_branch_00;  
        4'b1011: cnt = cnt_main_11;  
        4'b0111: cnt = cnt_main_00;  
        default: cnt = 7'b11111111;  
    endcase  
endmodule
```

6.7 设计思路

首先归结一下我们设计的交通信号灯系统，它分别控制着主干道和支干道的交通信号，交通信号设定如下：

```
//主干道(main) : green(30s),yellow(5s),red(25s)
//支干道(branch): red(35s),green(20s),yellow(5s)
```

我们要利用开发板上的led灯与主干道和支干道的红黄绿灯对应，定时显示固定的颜色，同时要利用数码管显示倒计时。

seg7、divclk：既然要用到数码管，那么7段译码器 seg7 是必备的模块，它用于将数字信号转为7段译码器的显示信号，同时开发板所提供的时钟源是50MHz的，这显然不是我们所要用到的，所以我们也需要用到分频器 divclk 以得到所需频率的时钟信号。

scan_seg7：4数码管的显示是共同的，这就导致4个数码管同时只能显示相同的数字，而我们的倒计时是需要在四个数码管上显示不同的数字的，这就是我们编写scan_seg7模块的目的，该模块在1ms间以走马灯的形式快速循环得选中4个数码管，而在top模块中，选中不同的数码管时显示不同数字，1ms间的变化是肉眼无法察觉的，因此我们实现了4个数码管显示不同数字的功能。

traffic_light：该模块用于控制信号灯的亮起和倒计时的显示，具体设计请查看注释。