
华中科技大学

HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

实验报告

(包括实验 2, 3, 4)

学院	计算机科学与技术学院
班级	
老师	
姓名	
学号	

2020 年 05 月 12 日

目录

实验 2 简单组合电路设计.....	1
任务描述.....	1
相关知识.....	1
实验步骤.....	2
编程要求.....	4
测试说明.....	4
源码.....	6
Testbench 代码.....	10
控制台输出和波形图.....	10
遇到问题和解决方法.....	11
实验心得、意见和建议.....	13
实验 3 简单时序电路设计.....	14
任务描述.....	14
相关知识.....	14
实验内容.....	15
遇到的问题及解决方法.....	23
实验心得、意见和建议.....	23
实验 4 数据通路和有限状态机设计.....	24
任务描述.....	24
相关知识.....	24
实验内容.....	25
遇到问题和解决方法.....	43
实验心得、意见和建议.....	44

实验 2 简单组合电路设计

- 任务描述
 - 相关知识
 - 实验步骤
 - 编程要求
 - 测试说明
 - 源码
 - Testbench 代码
 - 控制台输出和波形图
 - 遇到问题和解决方法
 - 实验心得、意见和建议
-

任务描述

本关需要你根据所学的仿真测试的知识，完成选择器、译码器等组合电路的设计，对电路进行测试。熟悉 vivado 工具的操作；学习、掌握用 Verilog 语言设计组合逻辑电路的方法；掌握仿真测试方法，学习编写 testbench 并利用波形图进行测试。

相关知识

测试平台（Testbench）是用于测试和验证设计的正确性的程序。编写 Testbench 的主要目的是对使用硬件描述语言设计的电路进行仿真验证，测试设计电路的功能甚至部分性能是否与预期的目标相符。

测试一个实际功能电路需要用信号发生器来向电路输入测试信号、用示波器来观察电路的信号输出是否正确。一个待测的 Verilog HDL 模块就相当于一个功能电路，用 Testbench 对它进行仿真测试需要给待测模块输入激励、获取输出响应并作判断。Testbench 需要完成以下工作：

- (1) 产生仿真激励（波形）；
- (2) 将激励施加到被测模块端口并收集其输出响应；
- (3) 将输出响应与期望值进行比较，以判断是否符合预期目标。

典型的测试平台主要内容包括：

```
`timescale 1ns/100ps    //这里可适当指定仿真的“时间单位/时间精度”  
  
module XXX_tb;          //Testbench 模块，通常没有输入和输出端口  
  
    //局部 reg、wire 变量声明  
  
    //用 initial 和 always 等语句产生激励（波形）  
  
    //实例引用被测模块（籍以将激励自动施加其上）  
  
    //监视输出并与期望值做比较  
  
    //结束 testbench 程序的运行  
  
endmodule
```

其中许多内容书写的先后顺序不拘。

假若被测模块定义为

```
module M1(in1, in2, out1); //in1、in2 为 input 端口，out1 为 output 端口
```

则用来测试 M1 模块的 Testbench 模块，习惯上命名为 M1_tb，无输入无输出。Testbench 声明局部 reg、wire 变量时，应该包括（但不限于）一批与被测模块端口对应（不妨就同名）的变量，便于后面实例引用 M1 模块。并且与 input 端口、output 端口对应的变量分别声明成 reg 型和 wire 型。

实验步骤

请同学们根据实验任务细化实验步骤。

1. 编写如图 2.1 的 2 选 1 选择器电路的结构描述模块，并生成类似图 2.2 的原理图（RTL Analysis->Elaborated Design->Schematic）。

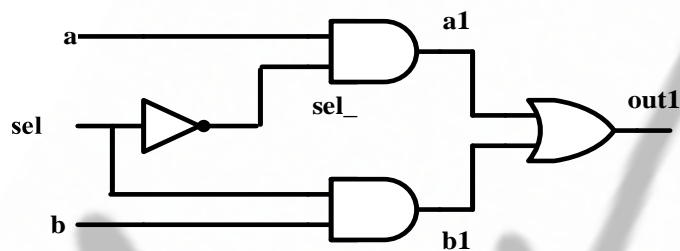


图 2.1 2 选 1 选择器电路

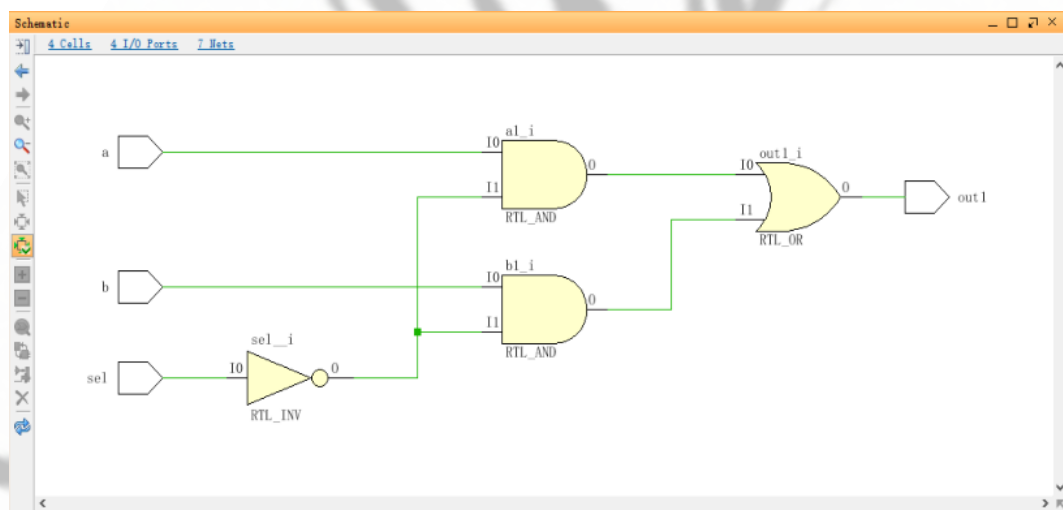


图 2.2 2 选 1 选择器 Schematic

2. 编写 2 选 1 选择器电路的数据流描述模块，并生成 Schematic。
3. 编写 2 选 1 选择器电路的行为描述模块，并生成 Schematic。
4. 用 2 选 1 多路选择器构造 3 选 1 多路选择器。顶层模块有 3 个数据输入端口 (u, v, w)、2 个选择输入端口 (s0, s1) 和 1 个输出端口 (m)。3 选 1 多路选择器的电路和真值如图 2-3 所示。请编写模块，并生成 Schematic。

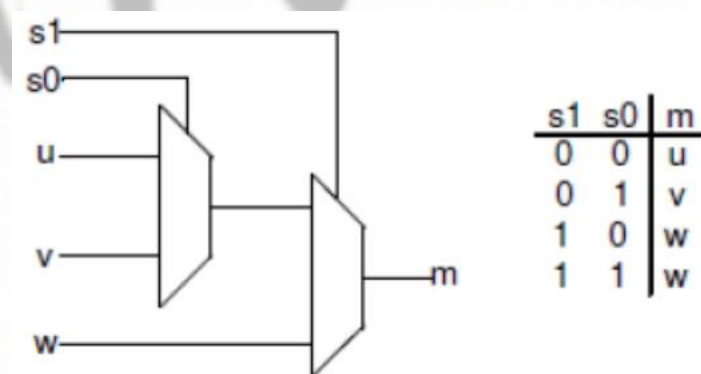


图 2-3 3 选 1 多路选择器电路和真值表

5. 设计一个 3-8 译码器模块，其真值表如表 2.1 所示。

模块请用以下格式：

```
module decoder_38(F, CBA);
```

```
input [2:0] CBA;
```



```

output reg [7:0] F;

.....

endmodule

```

表 2.1 译码器真值表

C	B	A	F7	F6	F5	F4	F3	F2	F1	F0
0	0	0	1	1	1	1	1	1	1	0
0	0	1	1	1	1	1	1	1	0	1
0	1	0	1	1	1	1	1	0	1	1
0	1	1	1	1	1	1	0	1	1	1
1	0	0	1	1	1	0	1	1	1	1
1	0	1	1	1	0	1	1	1	1	1
1	1	0	1	0	1	1	1	1	1	1
1	1	1	0	1	1	1	1	1	1	1

6. 为上述 3-8 译码器编写 Testbench 并进行测试。

编程要求

Testbench 要能生成包含各种输入值和对应输出值的波形图，并在 TCL 控制台打印各种输入值和对应输出值。

测试说明

以下是测试样例。

【例】五人投票表决器，过半数赞成则通过。仿真波形如图 2.4 所示。

```

module voter5(output pass, input vote);
    wire [4:0] vote;    //vote[i]表示第 i 人投票情况（1：赞成；0：反对）
    reg pass;           //最后结果（1：通过；0：不通过）
    reg [2:0] count;    //赞成票数

    integer i;
    always @(vote) begin
        count = 0;
        for (i = 0; i < 5; i = i+1) if (vote[i]) count = count + 1;
        if (count >= 3) pass = 1;    // 3 人以上赞成，则 pass=1
        else pass=0;
    end
end

```

```
endmodule
```

```
//仿真测试 Testbench 模块
```

```
`timescale 1ns / 100ps
```

```
module voter5_tb( );
```

```
    wire pass;
```

```
    reg [4:0] vote;
```

```
    voter5 M(.pass(pass), .vote(vote));
```

```
    initial begin
```

```
        $display ("Time::[vote] [count] [pass]-----");
```

```
        $monitor ("%t::", $time, "[%b]\t[%d]\t[%b]", vote,M.count,pass);
```

```
    end
```

```
    initial begin
```

```
        for (vote = 0; vote < 5'b11111; vote = vote + 1)
```

```
            #2;
```

```
        #2 $stop;
```

```
    end
```

```
endmodule
```

```
//TCL 控制台输出结果:
```

```
$time::[vote] [count] [pass]-----
```

```
0::[00000] [0] [0]
```

```
2000::[00001] [1] [0]
```

```
4000::[00010] [1] [0]
```

```
6000::[00011] [2] [0]
```

```
8000::[00100] [1] [0]
```

```
10000::[00101] [2] [0]
```

```
12000::[00110] [2] [0]
```

```
14000::[00111] [3] [1]
```

```
16000::[01000] [1] [0]
```

```
18000::[01001] [2] [0]
```

```
20000::[01010] [2] [0]
```

```
22000::[01011] [3] [1]
```

```
24000::[01100] [2] [0]
```

```
26000::[01101] [3] [1]
```

```
28000::[01110] [3] [1]
```

```
30000::[01111] [4] [1]
```

```
32000::[10000] [1] [0]
```

```
34000::[10001] [2] [0]
```

```
36000::[10010] [2] [0]
```

```
38000::[10011] [3] [1]
```

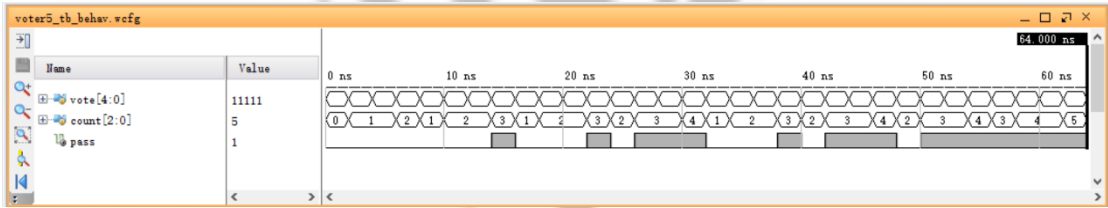
```
40000::[10100] [2] [0]
```

```
42000::[10101] [3] [1]
```

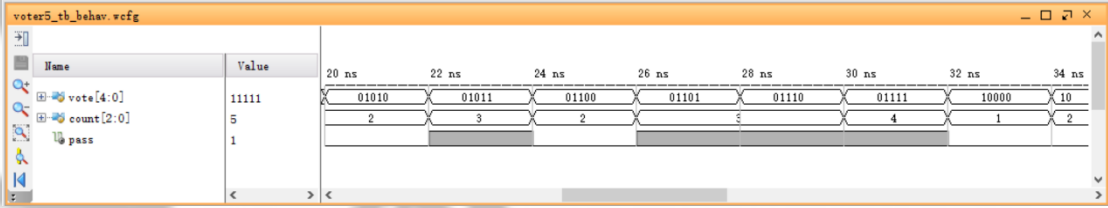
```
44000::[10110] [3] [1]
```

46000::[10111] [4] [1]
48000::[11000] [2] [0]
50000::[11001] [3] [1]
52000::[11010] [3] [1]
54000::[11011] [4] [1]
56000::[11100] [3] [1]
58000::[11101] [4] [1]
60000::[11110] [4] [1]
62000::[11111] [5] [1]

//波形图:



(a)波形



(b)波形（放大后）

图 2.3 仿真波形图

源码

1. 2 选 1 多路选择器的结构描述模块的源码

```
module mux_21_s(  
    input a,  
    input b,  
    input sel,  
    output out  
);  
  
    not(sel_, sel);  
    and(ao, a, sel_);  
    and(bo, b, sel);  
    or(out, ao, bo);
```

endmodule

生成的 Schematic 图见图 2.4。

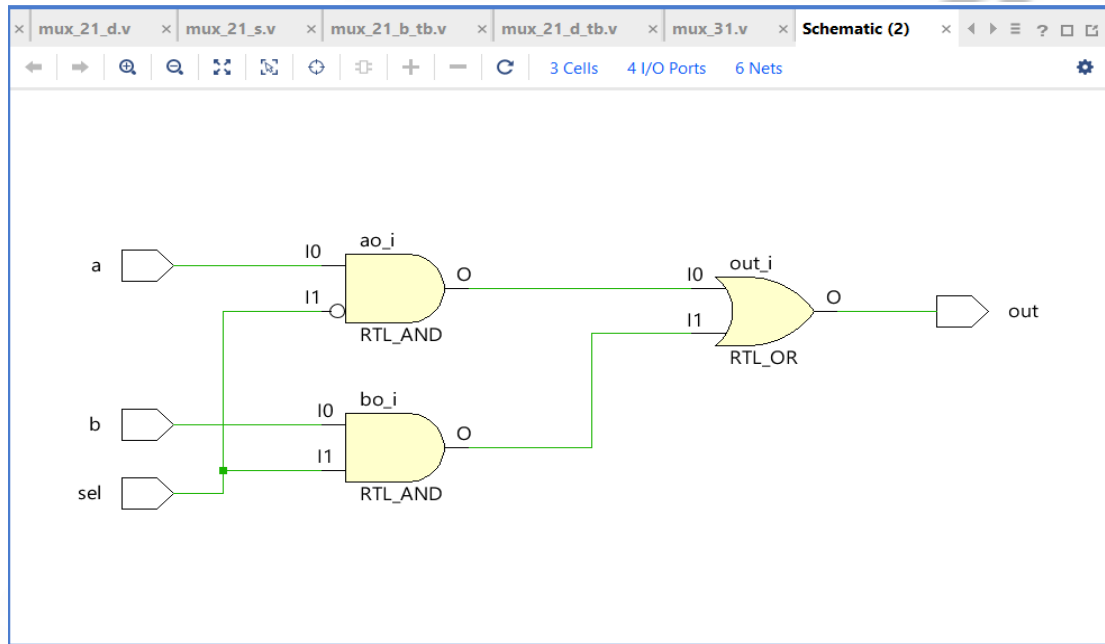


图 2.4 2 选 1 多路选择器的结构描述模块生成的 Schematic

2. 2 选 1 多路选择器的数据流描述模块的源码

```
module mux_21_d(
    input a,
    input b,
    input sel,
    output out
);

    assign out=(sel&b)|(!sel&a);
```

endmodule

生成的 Schematic 图见图 2.5。

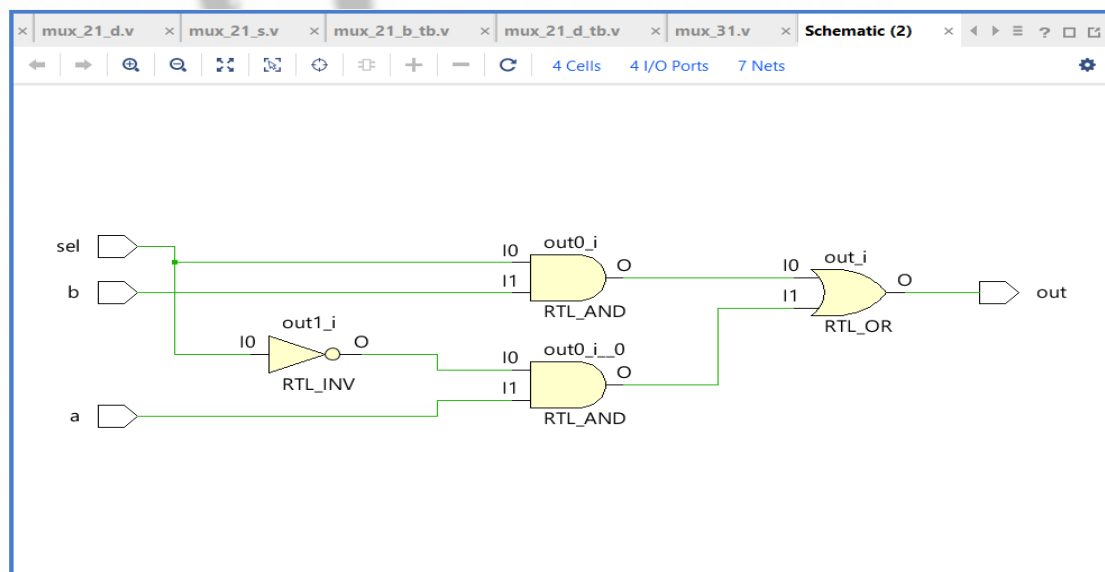


图 2.5 2 选 1 多路选择器的数据流描述模块生成的 Schematic

3. 2 选 1 多路选择器的行为描述模块的源码

```
module mux_2l_b(  
    input a,  
    input b,  
    input sel,  
    output reg out  
);  
  
always @(a, b, sel)  
begin  
    case(sel)  
        1'b1: out=b;  
        1'b0: out=a;  
    endcase  
end  
  
endmodule
```

生成的 Schematic 图见图 2.6。

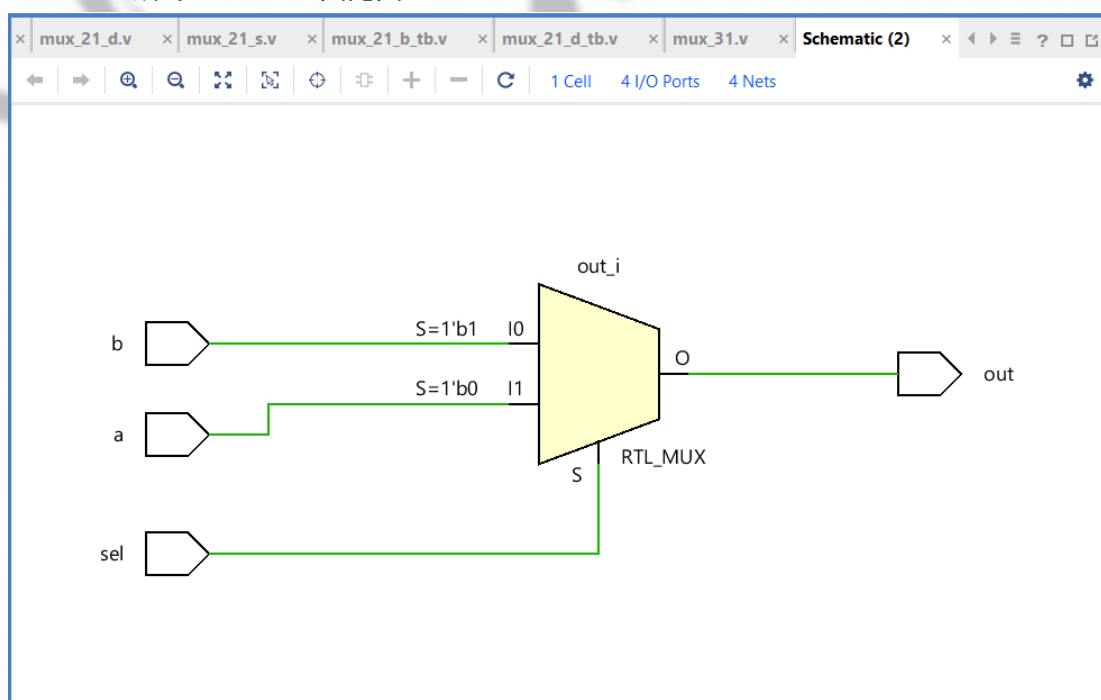


图 2.6 2 选 1 多路选择器的行为描述模块生成的 Schematic

4. 用 2 选 1 多路选择器实现 3 选 1 多路选择器的源码

```
module mux_3l(  
    input u,  
    input v,  
    input w,  
    input [1:0] sel,  
    output out  
);
```

```

mux_21_b mux_21_1(u,v,sel[0],t);
mux_21_b mux_21_2(t,w,sel[1],out);

```

endmodule

生成的 Schematic 图见图 2.7。

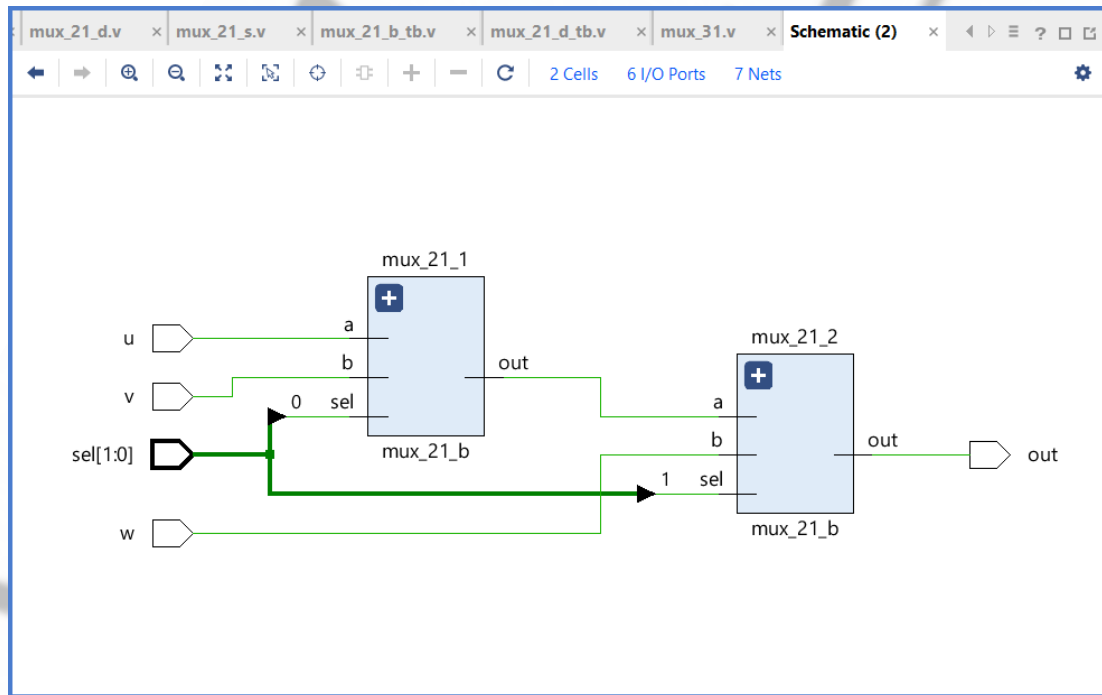


图 2.7 3 选 1 多路选择器生成的 Schematic

5. 3-8 线译码器模块的源码

```

module decoder_38(
    input [2:0] CBA,
    output reg [7:0] F
);

always @(CBA)
begin
    case(CBA)
        3'b000: F=8'b1111_1110;
        3'b001: F=8'b1111_1101;
        3'b010: F=8'b1111_1011;
        3'b011: F=8'b1111_0111;
        3'b100: F=8'b1110_1111;
        3'b101: F=8'b1101_1111;
        3'b110: F=8'b1011_1111;
        3'b111: F=8'b0111_1111;
    endcase
end

```

```
endmodule
```

Testbench 代码

1. 为 3-8 线译码器编写的 testbench 代码

```
`timescale 1ns / 1ps

module decoder_38_tb();
    reg [2:0] CBA;
    wire [7:0] F;

    decoder_38 dut(CBA, F);

    initial begin
        $display ("Time::[CBA] [F] -----");
        $monitor ("%t::", $time, "[%b]\t[%b]", CBA,F);
    end

    initial begin
        CBA=3'b000;

        #15 CBA=3'b001;
        #15 CBA=3'b011;
        #15 CBA=3'b010;
        #15 CBA=3'b110;
        #15 CBA=3'b111;
        #15 CBA=3'b101;
        #15 CBA=3'b100;
    end
end

endmodule
```

控制台输出和波形图

1. 3-8 线译码器的 testbench 的控制台输出如下，波形图见图 2.8。

```
Time::[CBA] [F] -----
      0::[000]    [11111110]
    15000::[001]    [11111101]
    30000::[011]    [11110111]
    45000::[010]    [11111011]
    60000::[110]    [10111111]
    75000::[111]    [01111111]
```

```

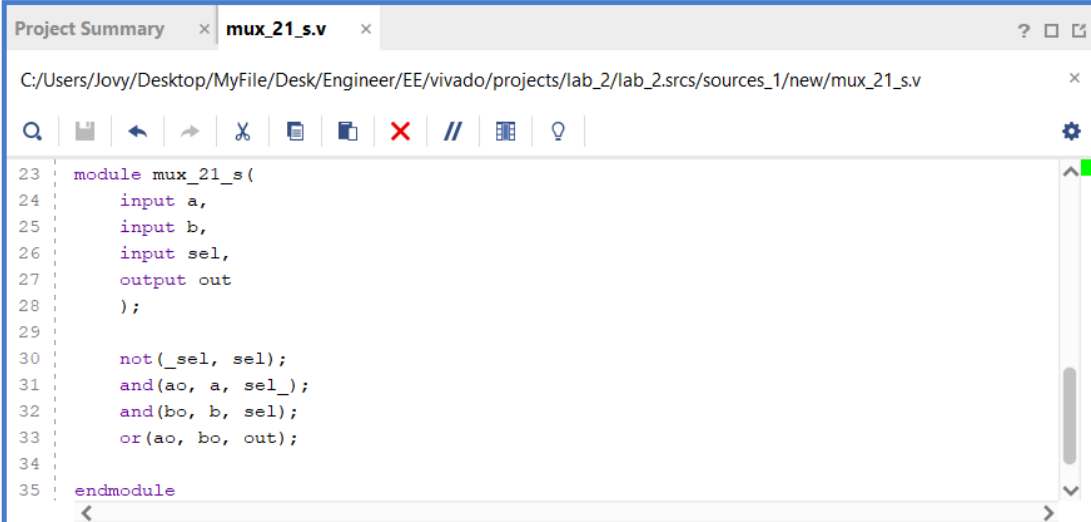
90000::[101]    [11011111]
105000::[100]    [11101111]

```

图 2.8 3-8 线译码器的 testbench 输出的波形图

遇到问题和解决方法

1.在自己编写第一个 verilog 程序时,也就是 2 选 1 多路选择器的结构描述模块,对于 verilog 的语法和格式还不是特别习惯,尤其是使用与门、或门、非门时常常不知道输入输出的顺序,一开始写的时候写成了图 2.9 所示的模样。



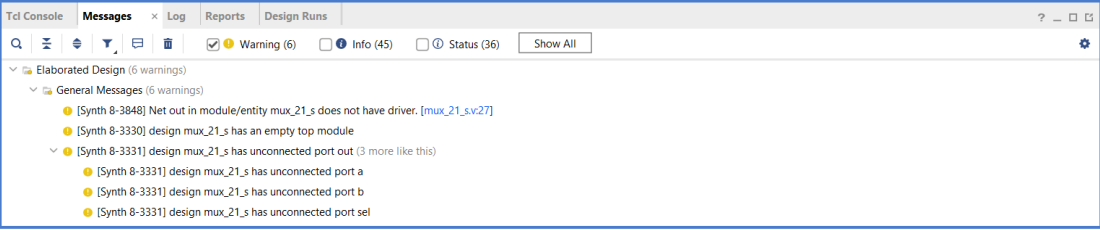
```

23 module mux_21_s (
24     input a,
25     input b,
26     input sel,
27     output out
28 );
29
30 not(_sel, sel);
31 and(ao, a, sel_);
32 and(bo, b, sel);
33 or(ao, bo, out);
34
35 endmodule

```

图 2.9 程序一开始的写法

存盘之后就报了如图 2.10 这些错。



```

Tcl Console Messages x Log Reports Design Runs
[Synth 8-3848] Net out in module/entity mux_21_s does not have driver. [mux_21_sv:27]
[Synth 8-3330] design mux_21_s has an empty top module
[Synth 8-3331] design mux_21_s has unconnected port out (3 more like this)
[Synth 8-3331] design mux_21_s has unconnected port a
[Synth 8-3331] design mux_21_s has unconnected port b
[Synth 8-3331] design mux_21_s has unconnected port sel

```

图 2.10 存盘后报错

一开始我都不知道哪里有错,但发现好像跟输出 out 有很强的关联,仔细检查程序中关于输出 out 部分才发现或门那里的输入输出写反了,改过来之后就不再报错了,于是开始生成原理图,生成之后发现还是不对(见图 2.11)。

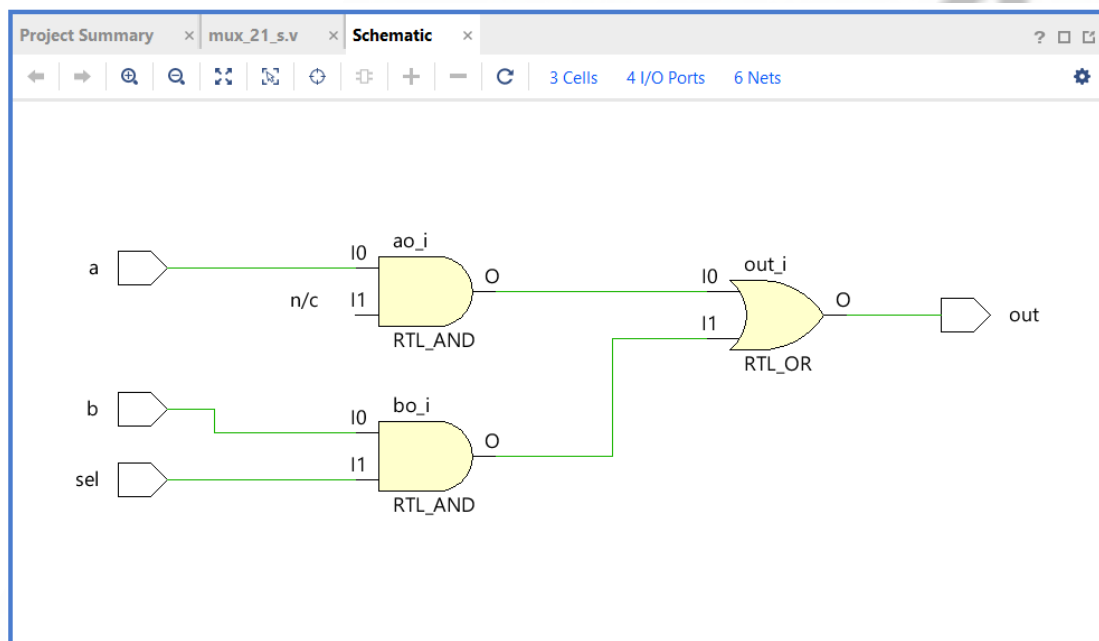


图 2.11 生成的原理图

从图 2.11 很明显可以看出有一个端口悬空了，而且没有出现结构描述过的非门，于是我去检查 a 的那个与门和 sel 的那个非门，检查后发现一个用的是 sel_，一个用的是_sel，两个不一样自然导致悬空。将这个错误改过来之后就得到了正确的原理图。

2. 编写 3-8 线译码器的 testbench 时比较粗心，少打了一个分号，存盘之后报了错（见图 2.12），但不是少了分号的那一行，而是下一行，这时候还不太熟悉 verilog 的报错方式，只是看到它提示 '#' 附件有错误，但不知道要往上一行找，我还以为是第 37 行中间某个字符是中文字符，找了半天也没找到，后来在网上搜了一下这个报错代码，才发现是上一行少了一个分号，顿时感觉自己又粗心又愚蠢。

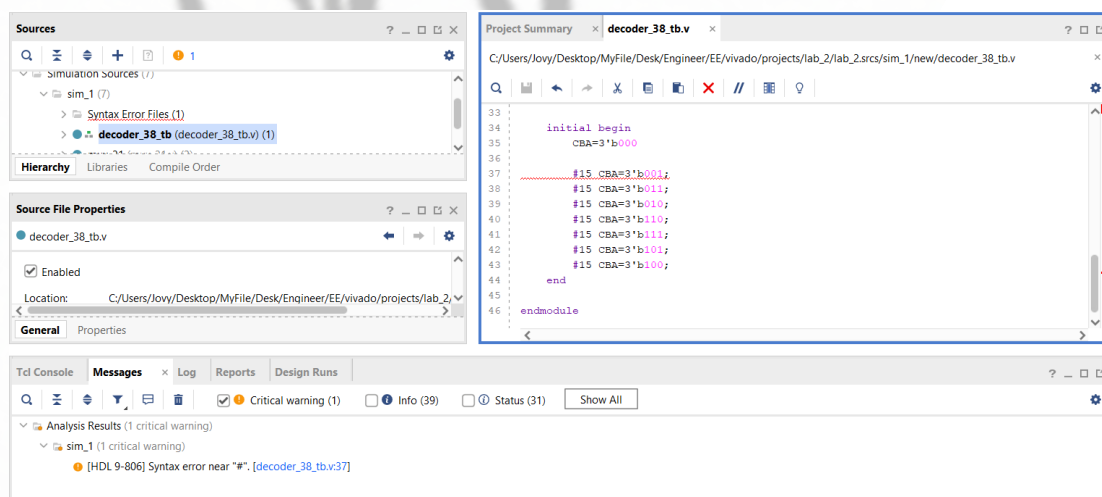


图 2.11 少打了一个分号

实验心得、意见和建议

1. 无论是在编写软件程序还是硬件程序，我们都必须要保持细心，这是一个受益终身的好习惯。有时候一个细微的错误可能导致非常严重的漏洞，还有时候这些错误非常隐蔽，需要我们耗费大量的时间去排查，耗时又费力，往往得不偿失。对于这些问题最好的解决方案就是在一开始编写的时候就要细心，不给这些错误可乘之机。
2. Verilog 和 C 很像，和 C 也很不一样，这可能是软件和硬件思维的区别。一开始接触到 Verilog 时，觉得其语法和 C 语言非常相近，给我造成的一种错觉就是它和 C 语言非常类似，没有什么太大区别。但在实验过程中慢慢体会到 Verilog 和 C 有非常大的本质上的区别，其中最大的感受就是软件和硬件思维上的区别。大部分软件处理程序基本上还是串行为主，但是硬件天生就有并行的优势，这是个人认为很重要的一点区别。
3. 熟能生巧。一开始接触 Verilog HDL 不管是对语言本身还是对 Vivado 套件肯定还是有点不适应，但慢慢地写了几个小模块之后开始逐渐熟悉这个流程，对这些操作也越来越熟练。在后面学习过程中更是要多加实践，才能更好的理解用 Verilog HDL 实现硬件开发的流程。

实验 3 简单时序电路设计

- 任务描述
 - 相关知识
 - 实验内容
 - 遇到的问题及解决方法
 - 实验心得、意见和建议
-

任务描述

1. 掌握 Verilog 语言的简单时序电路的设计、实现、仿真、调试方法。
2. 掌握锁存器、触发器、简单寄存器、移位寄存器和计数器等器件的建模和使用，了解这些器件带复位、使能、加载等功能的用法。
3. 掌握用测试平台（test bench）对模块进行测试和验证的方法。
4. 通过仿真波形图分析所设计模块功能的正确性。

相关知识

设计中经常用到时序电路，为保证时序正确，需要进行时序控制。时序控制可以与过程语句关联，时序控制有延迟控制和事件控制两种形式。

(1) 延迟控制

格式为：`#delay 过程语句`

比如：`#10 Q = 4'b1001;` 表示等待 10 个时间单位后执行赋值。

(2) 事件控制

事件控制又分跳变沿敏感事件控制和电平敏感时间控制。所谓跳变沿是指信号由低电平变为高电平（上升沿）或由高电平变为低电平（下降沿）的那一瞬间。

跳变沿敏感事件控制格式为： *@event 过程语句*

比如， `@(posedge clock) curr_state = next_state;` 表示在 clock 信号上出现了正跳变沿（上升沿），就执行赋值语句；否则，赋值语句被挂起。负跳变沿事件的表示是在信号前面加 `negedge`，比如， `@(negedge clock)` 表示 clock 信号出现负跳变沿的事件。

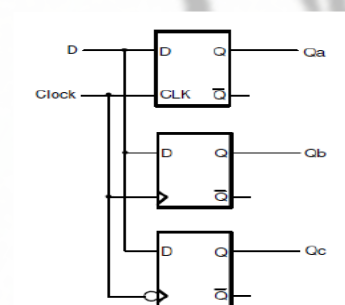
事件控制中的敏感事件可由多个表达式组成，用 `or` 或逗号把它们隔开，形成敏感事件列表。`@ *` 表示隐含地把过程语句中所有变量和线网都包含在敏感事件列表中。

实验内容

1、锁存器和触发器是时序电路中常用的存储器件。下面分别给出了 D 锁存器和 D 触发器（时钟上升沿触发）的行为建模。

```
module D_latch(input clk, input D, output reg Q);
    always @ (clk or D)
        if (clk) begin
            Q <= D;
        end
endmodule
```

```
module D_ff(input clk, input D, output reg Q);
    always @ (posedge clk) // 时钟上升沿触发
        Q <= D;
endmodule
```

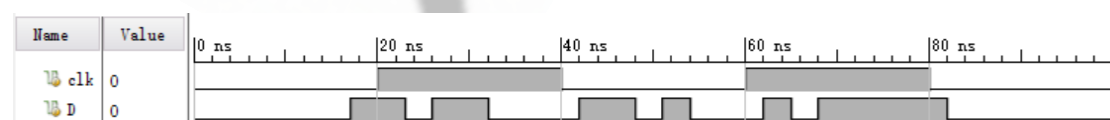


左图电路中，从上到下分别是 D 锁存器、时钟上升沿触发的 D 触发器和时钟下降沿触发的 D 触发器。为了弄清这三种器件在功能上的区别，本实验题首先要求对此电路建模，然后用下面给出的测试平台对设计进行仿真测试，将得到的波形图截图后粘贴在下面，对照波形图分析三种器件的功能。

(1) 时钟下降沿触发的 D 触发器建模：

```
module D_ff_n(input clk, input D, output reg Q);
    always @ (negedge clk)
        Q <= D;
endmodule
```

(2) 测试平台：



```
`timescale 1ns / 1ps

module lab3_1_tb( );
    reg clk;
    reg D;
    wire Qa, Qb, Qc;

    initial begin
        clk = 1'b0;
        #100 $stop;
    end

    always
        #20 clk = !clk;

    initial begin
        D = 1'b0;
        #17 D = !D;
        #6  D = !D;
        #3  D = !D;
        #6  D = !D;
        #10 D = !D;
        #6  D = !D;
        #3  D = !D;
        #3  D = !D;
        #8  D = !D;
        #3  D = !D;
        #3  D = !D;
        #14 D = !D;
    end

    D_latch myDlatch(clk, D, Qa);
    D_ff myDff(clk, D, Qb);
    D_ff_n myDffn(clk, D, Qc);
endmodule
```

(3) 仿真波形图：

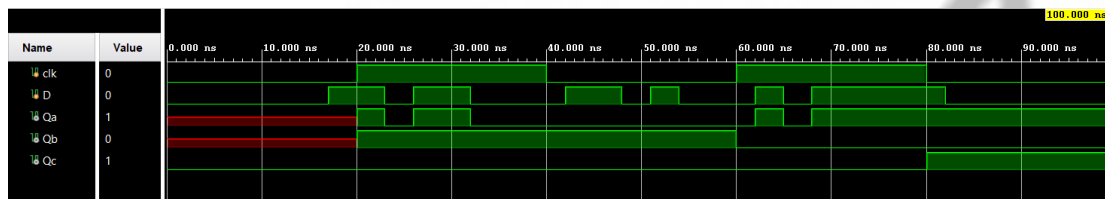


图 3-1 D 锁存器、上升沿 D 触发器、下降沿 D 触发器波形图

2、将几个触发器组合在一起并使用公共时钟，以此保存相关信息，这样的电路称为寄存器。以下是一个带同步复位功能的 4bit 寄存器。

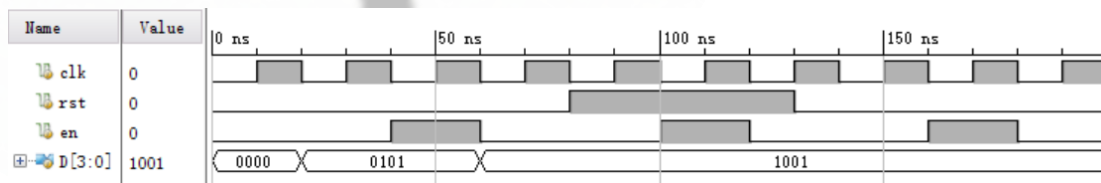
```
module Register_synch_reset(input [3:0] D, input clk, input rst, output reg [3:0] Q);
    always @(posedge clk)
        if (rst) begin           // 同步复位
            Q <= 4'b0;
        end else begin
            Q <= D;
        end
endmodule
```

下面首先需要在此基础上设计一个带同步复位和使能功能的 4bit 寄存器，复位信号的优先级要高于使能信号。非复位状态下，该器件在使能信号为高电平时，将输入信号 D 加载到输出端口 Q；否则，输出端口 Q 不变化。接着，设计测试平台对该寄存器进行仿真测试，观察并分析仿真波形图，验证其功能。

(1) 设计一个带同步复位和使能功能的 4bit 寄存器：

```
module Register_synch_reset_load(input [3:0] D, input clk, input rst, input en, output reg [3:0] Q);
    always @(posedge clk)
        if (rst) begin
            Q <= 4'b0000;
        end else if (en) begin
            Q <= D;
        end else begin
            Q <= Q;
        end
endmodule
```

(2) 设计测试平台进行仿真测试，输入信号波形如下图：



`timescale 1ns / 1ps

```

module lab3_2_tb();
    reg [3:0] D;
    reg clk;
    reg rst;
    reg en;
    wire [3:0] Q;

    initial begin
        clk = 1'b0;
        en = 1'b0;
        #200 $stop;
    end

    always
        #10 clk = !clk;

    always begin
        #40 en=!en;
        #20 en=!en;
    end

    initial begin
        rst = 1'b0;
        #80 rst = !rst;
        #50 rst = !rst;
    end

    initial begin
        D = 4'b0000;
        #20 D = 4'b0101;
        #40 D = 4'b1001;
    end

    Register_synch_reset_load myRegister(D, clk, rst, en, Q);

endmodule

```

(3) 仿真波形图：

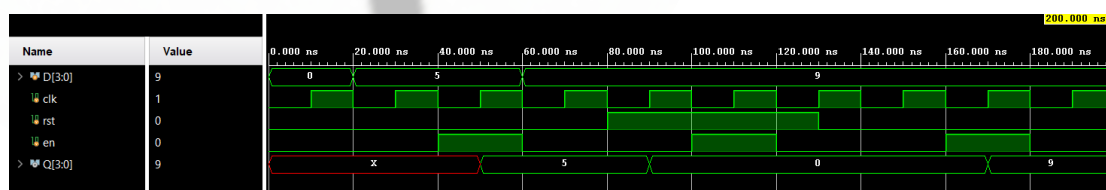


图 3-2 带同步复位和使能功能的四位寄存器测试波形图

3、下面的代码模拟了一个带加载和移位使能信号的 4bit 并行输入左移寄存器。

```
module Parallel_in_serial_out_load_enable(clk, ShiftIn, ParallelIn, load, ShiftEn, ShiftOut,
RegContent);
    input clk, ShiftIn, load, ShiftEn;
    input [3:0] ParallelIn;
    output ShiftOut;
    output [3:0] RegContent;

    reg [3:0] shift_reg;

    always @(posedge clk)
        if (load)
            shift_reg <= ParallelIn;
        else if (ShiftEn)
            shift_reg <= {shift_reg[2:0], ShiftIn};

    assign ShiftOut = RegContent[3];
    assign RegContent = shift_reg;
endmodule
```

下面设计一个 4bit 串入并出移位寄存器，并用测试平台仿真，输出仿真波形图验证其功能。

(1) 对 4bit 串入并出移位寄存器建模：

```
module Serial_in_Parallel_out_enable(clk, ShiftEn, ShiftIn, ParallelOut, ShiftOut);
    input clk, ShiftIn, ShiftEn;
    output [3:0] ParallelOut;           // 4bit 并行输出信号
    output ShiftOut;                   // 移位输出信号

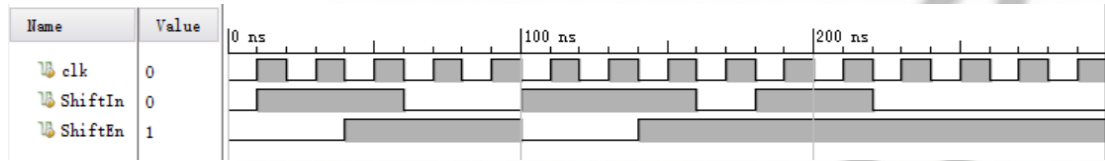
    reg [3:0] shift_reg;

    always @ (posedge clk)
        if(ShiftEn) begin
            shift_reg <= {ParallelOut[2:0], ShiftIn};
        end

    assign ShiftOut = ParallelOut[3];
    assign ParallelOut = shift_reg;

endmodule
```

(2) 设计测试平台进行仿真测试，输入信号波形如下图：



```
`timescale 1ns / 1ps
```

```
module lab3_3_tb( );
```

```
    reg clk;
```

```
    reg ShiftEn;
```

```
    reg ShiftIn;
```

```
    wire [3:0] ParallelOut;
```

```
    wire ShiftOut;
```

```
    initial begin
```

```
        clk = 1'b0;
```

```
        #300 $stop;
```

```
    end
```

```
    always
```

```
        #10 clk = !clk;
```

```
    initial begin
```

```
        ShiftIn = 1'b0;
```

```
        #10 ShiftIn = 1'b1;
```

```
        #50 ShiftIn = 1'b0;
```

```
        #40 ShiftIn = 1'b1;
```

```
        #60 ShiftIn = 1'b0;
```

```
        #20 ShiftIn = 1'b1;
```

```
        #40 ShiftIn = 1'b0;
```

```
    end
```

```
    initial begin
```

```
        ShiftEn = 1'b0;
```

```
        #40 ShiftEn = 1'b1;
```

```
        #60 ShiftEn = 1'b0;
```

```
        #40 ShiftEn = 1'b1;
```

```
    end
```

```
    Serial_in_parallel_out_enable mySReg(clk, ShiftEn, ShiftIn, ParallelOut, ShiftOut);
```

```
endmodule
```

(3) 仿真波形图:

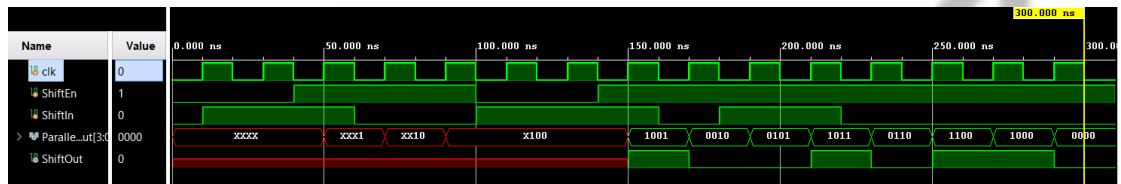


图 3-3 四位串入并出移位寄存器测试波形图

4、下面给出的是一个简单的 4bit 加法计数器：

```
module CNT4(input clk, outut [3:0] Q);
```

```
    reg [3:0] Q1;
```

```
    always @(posedge clk)
```

```
        Q1 <= Q1 + 1;
```

```
    assign Q = Q1;
```

```
endmodule
```

(1) 设计一个带同步复位和使能功能的 4bit 加法计数器（复位优先级高于使能优先级）：

```
module CNT4_synch_reset_enable(input clk, input rst, input en, output reg [3:0] Q);
```

```
    reg [3:0] Q1;
```

```
    always @ (posedge clk)
```

```
        if (rst) begin
```

```
            Q1 <= 4'b0000;
```

```
        end else if (en) begin
```

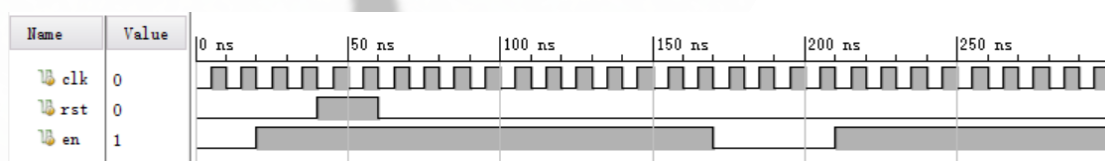
```
            Q1 <= Q1+1;
```

```
        end
```

```
    assign Q = Q1;
```

```
endmodule
```

(2) 设计测试平台进行仿真测试，输入信号波形如下图：



```
`timescale 1ns / 1ps
```

```
module lab3_4_tb( );
```

```
    reg clk;
```



```

reg rst;
reg en;
wire [3:0] Q;

initial
    clk = 1'b0;

always
    #5 clk = !clk;

initial begin
    rst = 1'b0;
    #40 rst = 1'b1;
    #20 rst = 1'b0;
end

initial begin
    en = 1'b0;
    #20 en = 1'b1;
    #150 en = 1'b0;
    #40 en = 1'b1;
end

CNT4_synch_reset_enable myCNT(clk, rst, en, Q);

endmodule

```

(3) 仿真波形图：

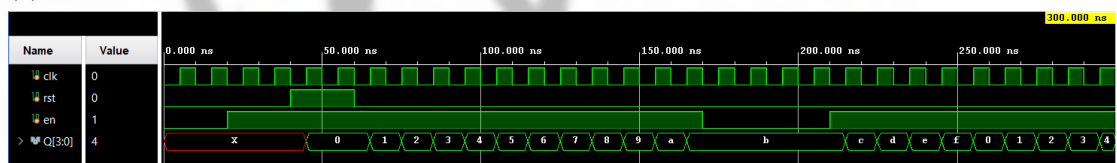


图 3-4 带同步复位和使能功能的四位加法计数器测试波形图

遇到的问题及解决方法

经过前两次实验的磨合，这次实验碰到的问题不太多，但还是有一些细节问题要多加注意，避免出现问题。

首先，就是关于行为描述中的初始化问题。由于 `always` 语句的触发是由其后的 `trigger list` 决定的，在时序逻辑电路的初始状态（也就是首次达到触发条件之前），电路的输出是处于不确定的状态的，也就是输出的波形图当中的红色部分。



图 3-5 未初始化的情况下的波形图

我一开始自作聪明，通过 `initial` 语句或者在声明寄存器类变量时给变量赋一个初值，于是波形图中不再出现红色部分了。虽然我们在设计电路的时候，可以通过加上 `initial` 语句给变量赋初值，但是考虑到在实际电路设计过程中是没有行为描述的初始化的，因而我们不应该这样做。

其次，就是变量类型的问题。在第一次实验的过程中，我们就了解到了变量也是分类型的，不同的描述方法对变量类型的要求也不同，我们需要做好区分，从而方便我们编写正确的程序。

实验心得、意见和建议

这一次实验最明显的感受便是对 `vivado` 套件的使用越来越熟练，从第一次实验的需要对照老师的视频边看边操作，到后来第二次实验尝试自己操作，不清楚了再去查看文档，直至这一次实验可以完全自己操作了，在这个过程中也对用 `vivado` 设计电路的流程更加熟悉了。

另一方面，这一次实验对于三大描述方法尤其是数据流描述和行为描述的理解更加深刻了。个人认为，数据流描述根本在于输出随输入随时变化；而行为描述的根本就是抓住了触发条件，可以是边沿触发也可以是电平触发，其最大的特点在于输出只会在特点的触发点变化。

实验 4 数据通路和有限状态机设计

- 任务描述
 - 相关知识
 - 实验内容
 - 遇到问题和解决方法
 - 实验心得、意见和建议
-

任务描述

综合应用掌握的简单组合电路（实验 2）和简单时序电路（实验 3）的设计方法，完成一个数据通路的设计，并为该数据通路配上一个控制器（有限状态机），最后将所有的实验综合起来，实现一个简单的处理器（自动运算电路）。

相关知识

在学习完实验 3（简单时序电路设计）后，实验 4 将尝试较为复杂的时序电路设计（比如处理器）。此类电路设计主要包含“数据通路”（Datapath）和“控制器”（Controller）两大部分，在经典计算机模型中，处理器部分如图 1 红框所示。其中，数据通路负责数据的操作，包括算术运算和传输数据；控制器负责数据的控制，通常以有限状态机（FSM: Finite State Machine）方式实现，包括控制流的输入、输出，以及控制数据通路中数据的传输顺序。另外，处理器旁通常会有一个“存储器”（Memory），可根据地址存取程序指令和数据。注意，数据通路自身并不能工作，只能通过控制器输出控制信号，输入到数据通路的各个单元，才能完成处理器的工作。因此，一个经典处理器通常是由数据通路和控制器组合完成的；与之对应的，本实验共包含三个步骤：数据通路（步骤 1），有限状态机（步骤 2），和自动运算处理器（步骤 3）。

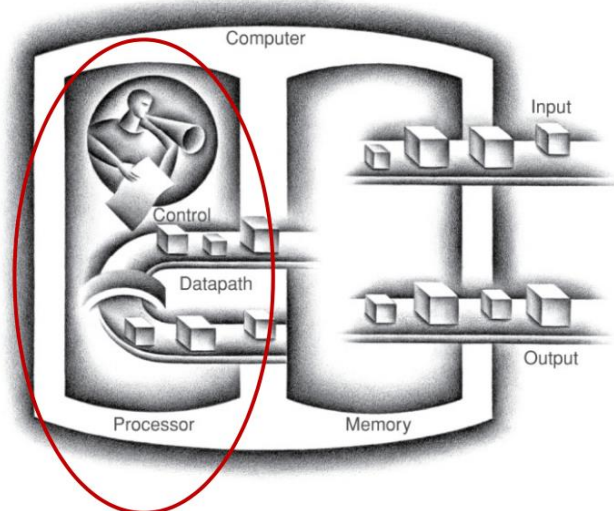


图 1 处理器由数据通路和控制器构成

实验内容

1. 数据通路设计（步骤 1）

【实验样例】

图 2 所示处理器中，有 4 个逻辑单元，包括一个计算单元 ALU，两个寄存器单元 LA 和 LB，以及一个双端口存储器 GR。各个单元的外部控制信号包括 OP, lda, ldb, read_addr, write_addr, WE 等，数据通路内部传输数据包括 la_data, lb_data, gr_data 和 alu_data。

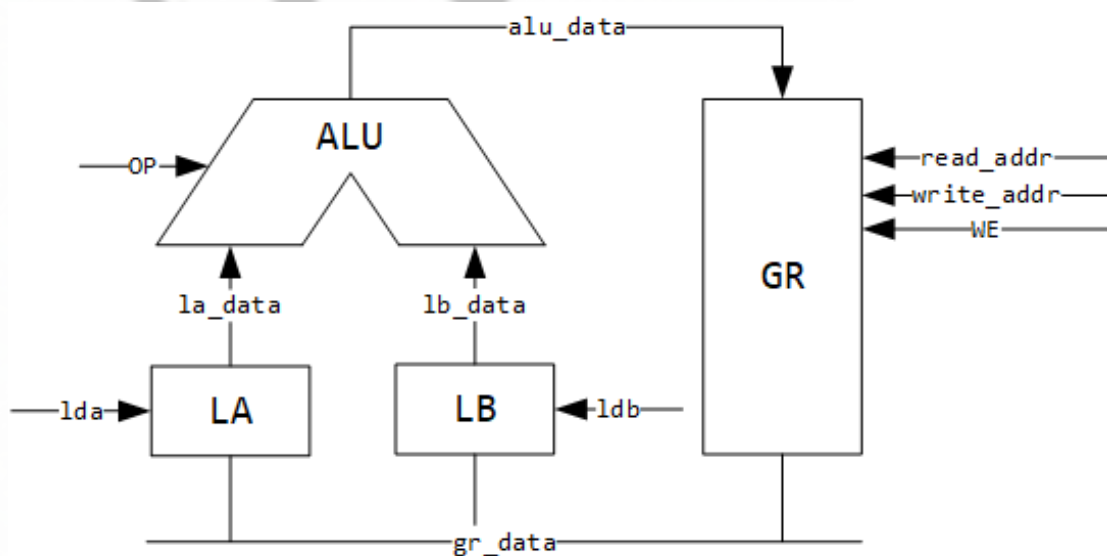


图 2 某处理器的数据通路

首先实现各个单元模块：

```
module alu(a, b, op, q);  
    parameter WIDTH = 8;  
    input [WIDTH-1:0] a, b;  
    input [1:0] op;  
    output reg [WIDTH-1:0] q;  
    always @(*) begin  
        case(op)  
            2'b00: q = a + b;  
            2'b01: q = a & b;  
            2'b10: q = a ^ b;
```

```
module register(clk, rst_n, en,  
d, q);  
    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;  
        else if (en) q <= d;  
    end  
endmodule
```

```
module ram(data, read_addr, write_addr, clk, we, q);  
    parameter DATA_WIDTH = 8;  
    parameter ADDR_WIDTH = 3;  
  
    input clk, we;  
    input [DATA_WIDTH-1:0] data;  
    input [ADDR_WIDTH-1:0] read_addr, write_addr;  
    output reg [DATA_WIDTH-1:0] q;  
  
    // 申明存储器数组  
    reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];  
  
    always @(posedge clk) begin  
        if (we)  
            ram[write_addr] <= data;
```


其次利用结构描述设计方法，利用以上模块实现图 2 的数据通路：

```
module datapath_top(clk, rst, lda, ldb, read_addr, write_addr, we,
op);

input clk, rst, lda, ldb, we;
input [4:0] read_addr, write_addr;
input [1:0] op;

wire [31:0] gr_data, alu_data;
wire [31:0] la_data, lb_data;

register #(32) LA (clk, rst, lda, gr_data, la_data);
register #(32) LB (clk, rst, ldb, gr_data, lb_data);
ram #(32, 5) GR (alu_data, read_addr, write_addr, clk, we,
gr_data);

alu #(32) ALU (la_data, lb_data, op, alu_data);

endmodule
```

注：register #(32) LA (clk, rst, lda, gr_data, la_data)传递参数 32 到 LA 模块，使得其 WIDTH = 32。

【实验要求】

请参照实验样例，实现图 3 所示的数据通路。图 3 给出的数据通路里，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。具体要求如下：

- 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；
- 设计的数据通路能够正确综合，Vivado 所示的电路原理图与图 3 给出的一致。

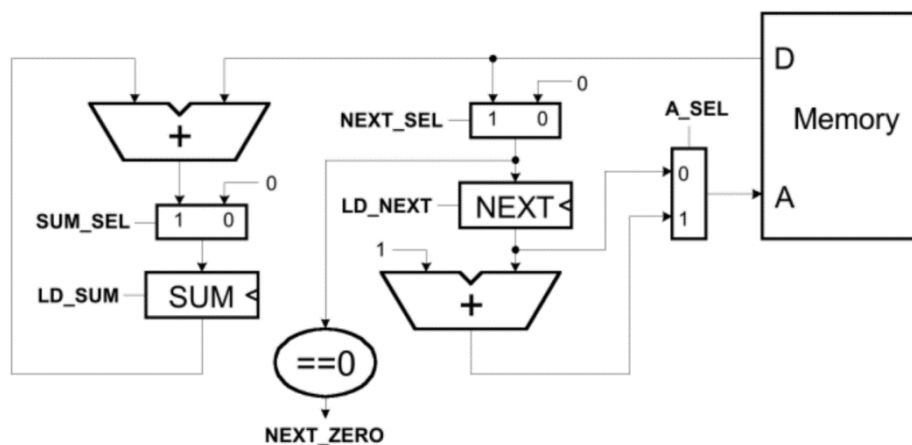


图 3 数据通路图

【实验提示】

(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, 存储器中存放如下的链表 (具体见图 7), 链表第 1 个节点在 0 号地址, 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。

00000003

00000002

00000000

00000007

00000004

00000000

00000000

0000000b

00000006

00000000

00000000

00000000

00000008

00000000

00000000

00000000

注：存储器存放该链表的过程可以如下实现：1) 将该链表存入一个文本文件；
2) 用系统函数\$readmemh 读该文本文件对存储器进行初始化。具体可见教材 readmemh 的语法。

(4) 利用以上模块完成图 3 的数据通路模块的设计

输入端口有：时钟 clk，复位 rst，加载信号 SUM_SEL, NEXT_SEL, A_SEL, LD_SUM, LD_NEXT。

输出端口有：链尾标志 NEXT_ZERO，求和结果 sum_out。

【实验填写】

参照实验样例，根据实验提示完成实验要求，包括：

1. 图 3 各个单元模块的代码（参考课堂 PPT）

(1) 加法器

```
module sum(a, b, c);  
    parameter WIDTH = 8;  
  
    input [WIDTH-1:0] a;  
    input [WIDTH-1:0] b;  
    output [WIDTH-1:0] c;  
  
    assign c = a+b;  
endmodule
```

(2) 寄存器

```
module register(clk, rst_n, en, d, q);  
    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;  
        else if (en) q <= d;  
    end  
endmodule
```

(3) 2 选 1 多路选择器

```
module mux_21(a, b, sel, c);  
    parameter WIDTH = 8;
```

```
    input [WIDTH-1:0] a, b;  
    input sel;  
    output reg [WIDTH-1:0] c;
```

```
    always @(a, b, sel) begin  
        case(sel)  
            1'b0: c = a;  
            1'b1: c = b;  
        endcase  
    end
```

```
endmodule
```

(4) rom

```
module rom(read_addr, data);  
    parameter DATA_WIDTH = 8;  
    parameter ADDR_WIDTH = 8;  
    parameter INIT_FILE = "sum_init.mem";
```

```
    input [ADDR_WIDTH-1:0] read_addr;  
    output [DATA_WIDTH-1:0] data;
```

```
    reg [DATA_WIDTH-1:0] rom[15:0];
```

```
    initial begin  
        $readmemh(INIT_FILE, rom);  
    end
```

```
    assign data = rom[read_addr];  
endmodule
```

(5) 比较器

```
module is_equal(a, b, c);  
    parameter WIDTH = 8;
```

```
    input [WIDTH-1:0] a;  
    input [WIDTH-1:0] b;  
    output c;
```

```
    assign c = (a==b);  
endmodule
```

2. 数据通路的代码:

```
module datapath_top(clk, rst, SUM_SEL, NEXT_SEL, A_SEL, LD_SUM, LD_NEXT, NEXT_ZERO, sum_out);
    parameter WIDTH = 32;
    parameter ADDR_WIDTH = 32;
    parameter SUM_FILE = "sum_init.mem";

    input clk, rst, SUM_SEL, NEXT_SEL, A_SEL, LD_SUM, LD_NEXT;
    output NEXT_ZERO;
    output [WIDTH-1:0] sum_out;

    wire [WIDTH-1:0] tmp, next_addr, read_addr, sum_tmp, data, next_addr_tmp, data_addr;

    wire [WIDTH-1:0] low = 'h0;
    wire [WIDTH-1:0] unit = 'h1;

    register #(WIDTH) SUM(clk, rst, LD_SUM, tmp, sum_out);
    register #(WIDTH) NEXT(clk, rst, LD_NEXT, next_addr_tmp, next_addr);
    rom #(WIDTH, ADDR_WIDTH, SUM_FILE) GR(read_addr, data);
    mux_21 #(WIDTH) SUM_MUX(low, sum_tmp, SUM_SEL, tmp);
    mux_21 #(WIDTH) NEXT_MUX(low, data, NEXT_SEL, next_addr_tmp);
    mux_21 #(WIDTH) A_MUX(next_addr, data_addr, A_SEL, read_addr);
    sum #(WIDTH) data_sum(sum_out, data, sum_tmp);
    sum #(WIDTH) inc_addr(unit, next_addr, data_addr);
    is_equal #(WIDTH) endpoint(next_addr_tmp, low, NEXT_ZERO);
endmodule
```

3. 数据通路的电路原理图 (生成 Schematic):

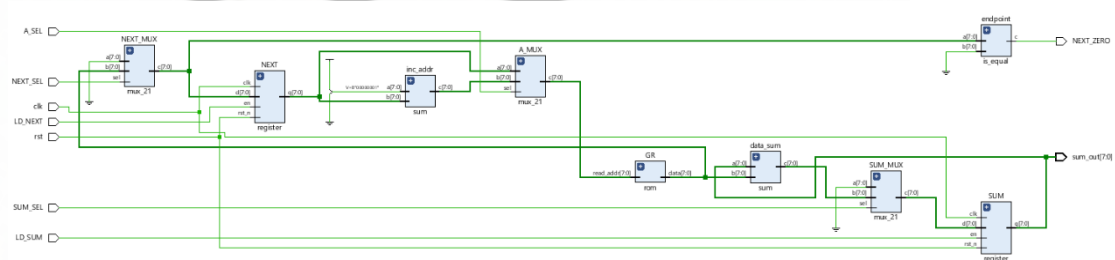


图 4-1 数据通路的电路原理图

2. 有限状态机设计（步骤 2）

【实验样例】

给定某一类激光计时器（图 4），不按按钮(即 $B=0$)，激光器关闭(即 $X=0$)；按了按钮(即 $B=1$)，激光器会发射 3 个周期(即 $X=1$)；3 个周期后激光器关闭(即 $X=0$)。

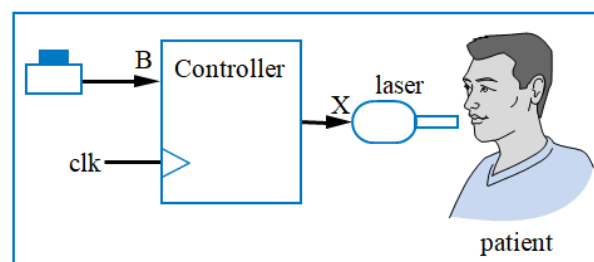


图 4 激光计时器

该类激光计时器的有限状态机如图 5 所示，拥有 Off（关闭），On1~On3（第 1~3 个周期激光发射）一共四个状态。每一个时钟周期都触发一次状态迁移。

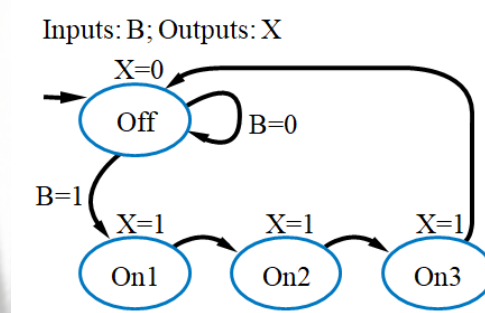


图 5 该激光计时器的有限状态机

图 5 所示状态机的代码如下：

```
module LaserTimer(B, X, Clk, Rst);

    input B;
    output reg X;
    input Clk, Rst;

    parameter S_Off = 0, S_On1 = 1, S_On2 = 2, S_On3 = 3; //对应四个状
```

(接上页)

```
// CombLogic
always @(State, B) begin
    case (State)
        S_Off: begin
            X <= 0;           //初始状态 Off: 关闭
            if (B == 0)
                StateNext <= S_Off; //不按按钮, 保持关闭
            else
                StateNext <= S_On1; //按了按钮, 下一个状态为
On1
        end
        S_On1: begin
            X <= 1;           //激光发射第 1 个周期
            StateNext <= S_On2; //下一个状态自动迁移为
On2
        end
        S_On2: begin
            X <= 1;           //激光发射第 2 个周期
            StateNext <= S_On3; //下一个状态自动迁移为
On3
        end
        S_On3: begin
            X <= 1;           //激光发射第 3 个周期
```

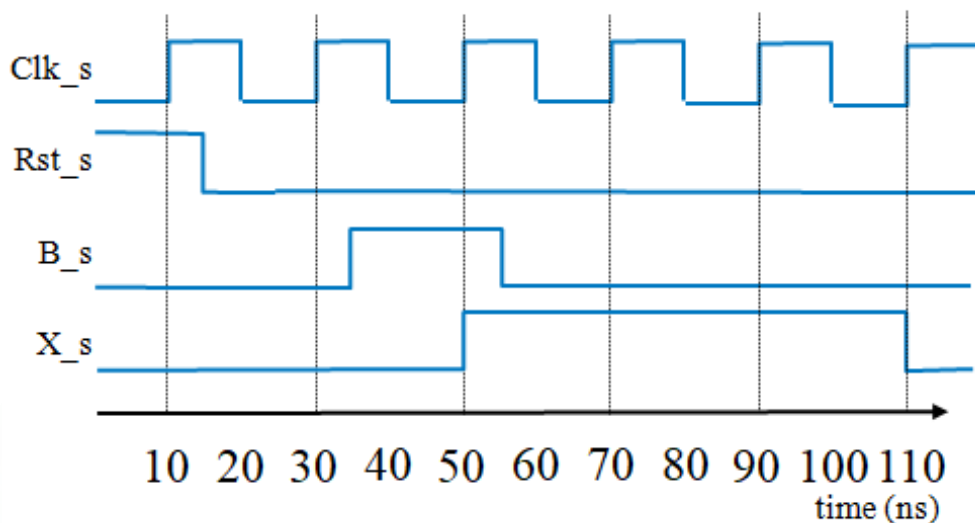
该状态机的 testbench 代码如下：

```
`timescale 1ns / 1ps

module laser_timer_tb();
    reg Clk_s, Rst_s, B_s;
    wire X_s;
    LaserTimer dut(Clk_s, Rst_s, B_s, X_s);
    always begin // 周期为 20ns 的时钟
        Clk_s <= 0;
        #10;
        Clk_s <= 1;
        #10;
    end

    initial begin
        Rst_s <= 1; //复位启动
        B_s <= 0; //按钮未按下
        @(posedge Clk_s); //到达下一个时钟上升沿
        #5 if (X_s != 0) //延迟 5ns 后验证复位是否成功
            $display("%t: Reset failed", $time);
        Rst_s <= 0; //复位关闭
        @(posedge Clk_s);
        #5 B_s <= 1; //按下按钮
        @(posedge Clk_s);
        #5 B_s <= 0; //松开按钮
        if (X_s != 1) //验证状态 On_1
            $display("%t: First X=1 failed", $time);
        @(posedge Clk_s);
        #5 if (X_s != 1) //验证状态 On_2
            $display("%t: Second X=1 failed", $time);
        @(posedge Clk_s);
        #5 if (X_s != 1) //验证状态 On_3
            $display("%t: Third X=1 failed", $time);
        @(posedge Clk_s);
        #5 if (X_s != 0) //验证状态 Off
            $display("%t: Final X=0 failed", $time);
    end
endmodule
```

仿真结果如下：



【实验要求】

假设有限状态机的状态转移图如图 6 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。具体要求如下：

- 设计的有限状态机（FSM）能够正确综合；
- 编写有限状态机的仿真程序，完成有限状态机（FSM）的功能仿真，有限状态机功能仿真正确。

【实验提示】

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：控制信号 LD_SUM, LD_NEXT, SUM_SEL, NEXT_SEL, A_SEL，求和结束 DONE。

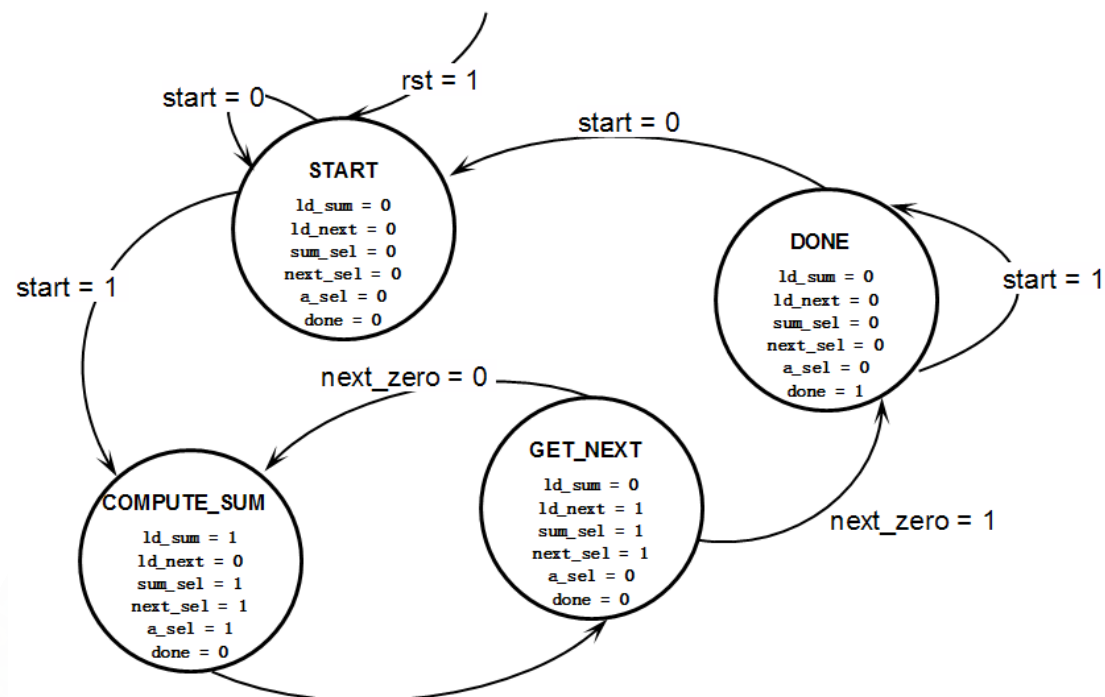


图 6 状态转移图

【实验填写】

参照实验样例，根据实验提示完成实验要求，包括：

1. 图 6 的 Verilog 程序代码

```
module FSM(clk,rst,start,next_zero, LD_SUM, LD_NEXT, SUM_SEL, NEXT_SEL, A_SEL,
DONE, State);
```

```
    input clk,rst,start,next_zero;
```

```
    output reg LD_SUM,LD_NEXT,SUM_SEL,NEXT_SEL,A_SEL,DONE;
```

```
    output reg [1:0] State;
```

```
    parameter START=2'h0, COMPUTE_SUM=2'h1, GET_NEXT=2'h2, FINISHED=2'h3;
```

```
    reg [1:0] NextState;
```

```
    // CombLogic
```

```
    always @(State, start, next_zero) begin
```

```
        case (State)
```

```
            START: begin
```

```
                LD_SUM <= 1'b0;
```

```
                LD_NEXT <= 1'b0;
```

```
                SUM_SEL <= 1'b0;
```

```
                NEXT_SEL <= 1'b0;
```

```

        A_SEL <= 1'b0;
        DONE <= 1'b0;
        if (start == 2'b0)
            NextState <= START;
        else
            NextState <= COMPUTE_SUM;
        end

    COMPUTE_SUM: begin
        LD_SUM <= 1'b1;
        LD_NEXT <= 1'b0;
        SUM_SEL <= 1'b1;
        NEXT_SEL <= 1'b1;
        A_SEL <= 1'b1;
        DONE <= 1'b0;
        NextState <= GET_NEXT;
    end

    GET_NEXT: begin
        LD_SUM <= 1'b0;
        LD_NEXT <= 1'b1;
        SUM_SEL <= 1'b1;
        NEXT_SEL <= 1'b1;
        A_SEL <= 1'b0;
        DONE <= 1'b0;
        if (next_zero == 1'b0)
            NextState <= COMPUTE_SUM;
        else
            NextState <= FINISHED;
        end
    end

    FINISHED: begin
        LD_SUM <= 1'b0;
        LD_NEXT <= 1'b0;
        SUM_SEL <= 1'b0;
        NEXT_SEL <= 1'b0;
        A_SEL <= 1'b1;
        DONE <= 1'b1;
        if (start == 1'b0)
            NextState <= START;
        else
            NextState <= FINISHED;
        end
    end
endcase

```

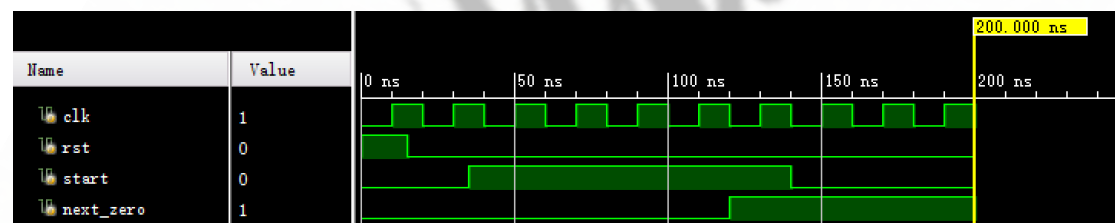
```

end

// StateReg
always @(posedge clk) begin
    if (rst == 1'b1 )
        State <= START;
    else
        State <= NextState;
    end
endmodule

```

2. 设计 testbench 进行仿真测试，输入信号波形如下图：



```

`timescale 1ns / 1ps
module FSM_tb();
    reg clk, rst, start, next_zero;

    FSM myFSM(clk,rst,start,next_zero, LD_SUM, LD_NEXT, SUM_SEL, NEXT_SEL,
A_SEL, DONE);

    initial begin
        clk = 1'b0;
        #200 $stop;
    end

    always
        #10 clk = !clk;

    initial begin
        rst = 1'b1;
        #15 rst = 1'b0;
    end

    initial begin
        start = 1'b0;
        #35 start = 1'b1;
        #105 start = 1'b0;
    end

    initial begin

```

```

        next_zero = 1'b0;
        #110 next_zero = 1'b1;
    end
endmodule

```

3. 仿真结果图：



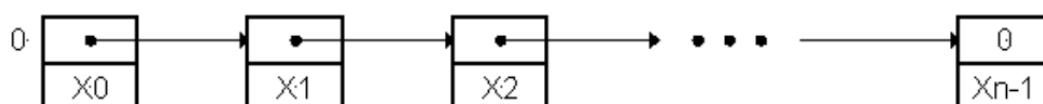
图 4-2 有限状态机仿真结果图

3. 自动运算电路的设计（步骤 3）

【实验要求】

将实验步骤 1 实现的数据通路与实验步骤 2 实现的有限状态机（FSM）结合起来，可以进行以链表方式存储的数据的求和运算。

在存储器中存放的数据链表（第 5 页所示链表）其结构如下图 7 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。



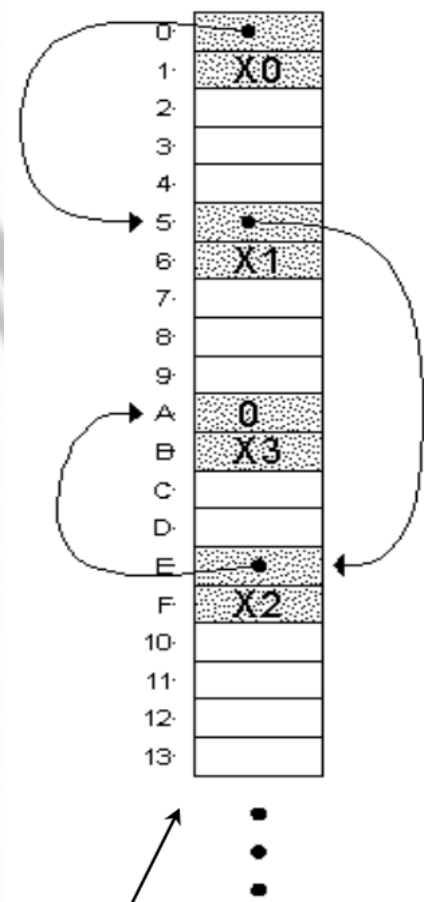


图 7 数据链表及其在存储器中的存放格式

利用上面设计的数据通路、有限状态机，将它们集成起来，设计并实现一个能够进行上述图 7 所示链表数据的自动求和运算，该电路的总体框架如图 8 所示。

具体要求如下：

- 完成自动运算求和电路的设计，能够正确综合；
- 编写仿真程序，进行功能仿真，仿真结果正确；

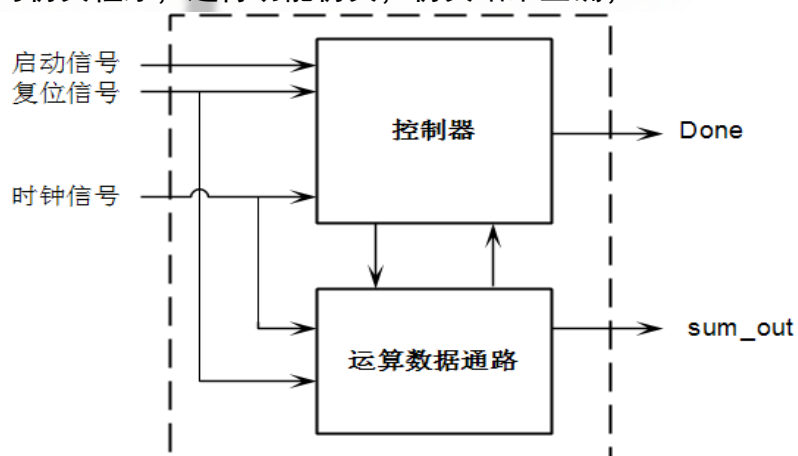


图 8 自动运算电路模块构成图

存储器初始化文件（存储器每个存储单元 32 位，共有 16 个存储单元，最后的求和运算结果 $= 2+4+6+8 = 20$ ）：

00000003
00000002
00000000
00000007
00000004
00000000
00000000
0000000b
00000006
00000000
00000000
00000000
00000008
00000000
00000000
00000000

【实验提示】

可参照实验 2（简单组合电路设计）中的第四步“用 2 选 1 多路选择器构造 3 选 1 多路选择器。”利用结构描述，结合步骤 1 和步骤 2 的数据通路模块和有限状态机模块，构造自动运算电路，完成图 7 所示的数据链表的求和运算。

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

输出端口：求和结束 DONE，求和结果 sum_out

【实验填写】

根据实验提示完成实验要求，包括：

1. 图 8 的 Verilog 程序代码

```
module auto_add(clk,rst,start,DONE,sum_out);  
`timescale 1ns / 1ps  
module auto_add(clk,rst,start,DONE,sum_out);  
    parameter WIDTH = 32;  
    parameter ADDR_WIDTH = 32;  
    parameter SUM_FILE = "sum_init.mem";  
    input clk,rst,start;
```



```

output DONE;
output [WIDTH-1:0] sum_out;

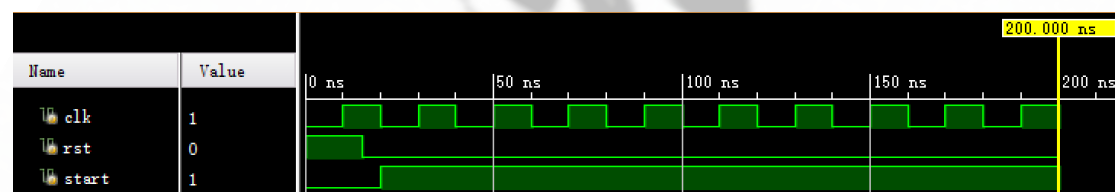
wire [1:0] State;

FSM myFSM(clk,rst,start, NEXT_ZERO, LD_SUM, LD_NEXT, SUM_SEL, NEXT_SEL,
A_SEL, DONE, State);
datapath_top #(WIDTH, ADDR_WIDTH, SUM_FILE) myDataPath(clk, rst, SUM_SEL,
NEXT_SEL, A_SEL, LD_SUM, LD_NEXT, NEXT_ZERO, sum_out);

assign next_zero = NEXT_ZERO ? 1 : 0;
endmodule

```

2. 设计 testbench 进行仿真测试，输入信号波形如下图：



```

`timescale 1ns / 1ps
module auto_add_tb();
    reg clk, rst, start;
    wire [7:0] sum_out;
    wire DONE;

    auto_add #(32, 32, "sum_init.mem") myAdder(clk,rst,start,DONE,sum_out);

    initial begin
        clk = 1'b0;
        #200 $stop;
    end

    always begin
        #10 clk = !clk;
    end

    initial begin
        rst = 1'b1;
        #15 rst = 1'b0;
    end

    initial begin
        start = 1'b0;
        #20 start = 1'b1;
    end
end

```

endmodule

3. 仿真结果图：

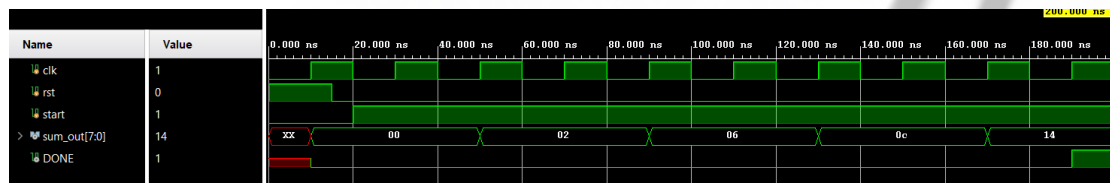


图 4-3 自动加法器仿真结果图

遇到问题和解决方法

刚刚把数据通路写出来的时候生成了电路图之后看了一下，貌似没有太大问题，就开始做有限状态机了，有限状态机比较简单，写好之后用 testbench 测试没有问题，心想着这次试验基本大功告成了，但是把数据通路和有限状态机连接起来之后发现问题很大，很多输出都是红色的，没有有效输出，而且有限状态机的变化也有问题（见图 4-4）。

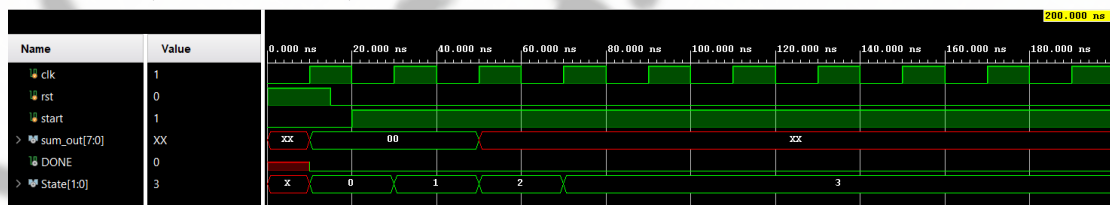


图 4-4 错误的输出

我觉得既然有限状态机测试没有问题那就只能是数据通路的问题了，于是开始找数据通路中的问题。虽然这个数据通路的电路很基础也很简单，但涉及到的结点也不少，我觉得很可能是我用结构化描述的时候把结点给接错了，于是我用了“最笨”的方法来检测，把结点的名称标在电路图上面（见图 4-5），然后对着结构化描述一个一个检查。在这个过程发现很多结点都接错了（与原理图不符），于是把它们一一改正，再次进行测试，还是有问题。

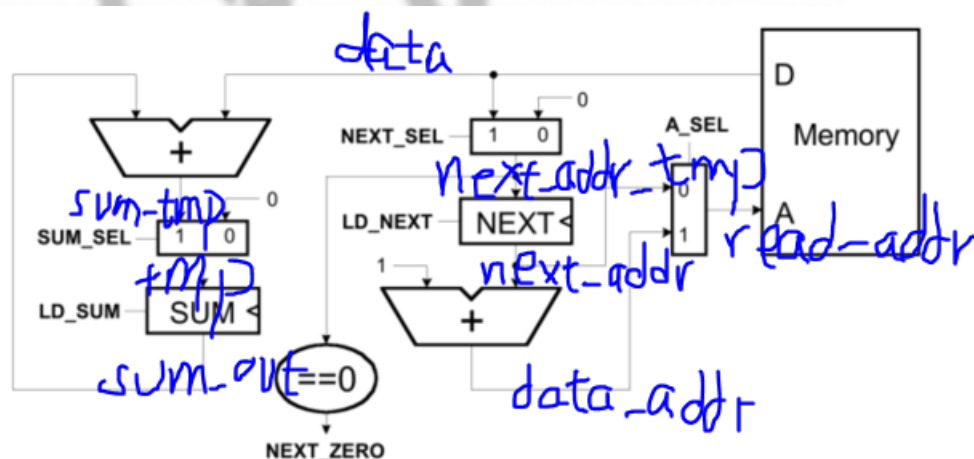


图 4-5 标结点检查结构描述

这个时候我觉得如果数据通路按照电路图接好了，有限状态机也测试过，那会不会是把它们两个接起来的时候出问题了，于是去看 auto_add.v，果不其然，数据通路和有限状态机的 next_zero 一个大写一个小写，两个端子没有接在一起。不幸的是，把这个错误纠正过来之后再去测试，仍然不正确。

于是我猜测会不会是 rom 里面的数据有问题，于是查看了 rom 的波形图，发现 rom 里面的数据没有问题，但是 rom 的 read_addr 有问题，没有初始值（见图 4-6）。



图 4-6 rom 波形图

我开始找为什么 read_addr 没有初值，是需要程序中人为赋初值吗？于是我又回到电路图那里去想这个自动加法器的初始过程，发现这个 rom 的 read_addr 是通过寄存器和多路选择器来控制初值的，于是我去观察 NEXT 选择器, NEXT 寄存器, 地址加法器, A 选择器这几个部件的波形图（见图 4-7）。

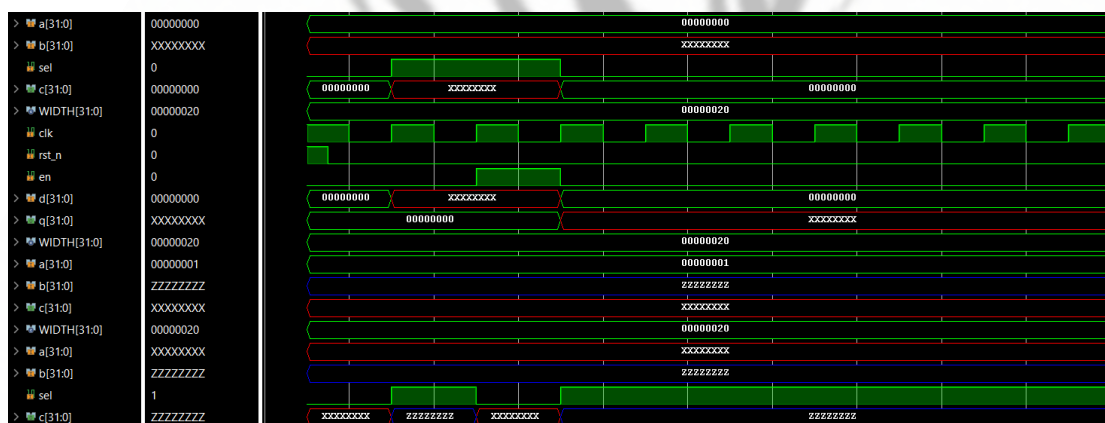


图 4-7 有关 read_addr 的波形图

这个时候我才发现是地址加法器出了问题导致 A_SEL 多路选择器的输入有问题，于是我去检查源代码中关于地址加法器的结构化描述，发现了一个致命的错误，一个输入和输出（next_addr 和 data_addr）写反了，导致有一个输入无效，从而导致 A_SEL 的输入出错，因而 rom 的 read_addr 无有效初值。把这个错误改过来之后整个电路就能正常工作了。

实验心得、意见和建议

这次实验涉及到两个非常重要的概念——数据通路和有限状态机，这有点像是程序设计的雏形，我们只需要把数据输入就能得到我们想要的结果，当然，前提是电路的设计能够实现我们预设的功能。因而，深刻理解并实现数据通路和有限状态机具有及其重要的意义。

对于数据通路的实现，我认为关键在于准确的结构化描述，尤其注意连着的结点名称必须完全相同，否则两个端子是断开的。除此之外，还要注意模块实例化时输入输出的排列顺序必须和定义模块时完全一致，切忌张冠李戴。这次实验过程中就是犯了这两低级错误导致出现严重的问题，并且后期花费很多时间来找到这个问题，事倍功半，得不偿失。

有限状态机的实现并不是很复杂，但有限状态机的设计必须要合理，要保证数据通路能够按照一定步骤顺序完成对应的功能，这样数据通路和有限状态机才能一起完成既定的功能。