

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** | CS1806 |
| **老师** | 胡燏翀 |
| **姓名** | 赵哲泷 |
| **学号** | U201814660 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **2020** | **年** | **5** | **月** | **11** | **日** |

目录

[1 实验2 简单组合电路设计 3](#_Toc14985)

[1.1 任务描述 3](#_Toc24043)

[1.2 相关知识 3](#_Toc9730)

[1.3 实验步骤 4](#_Toc15353)

[1.4 编程要求 6](#_Toc24723)

[1.5 测试说明 6](#_Toc27573)

[1.6 源码 9](#_Toc6706)

[1.7 Testbench代码 12](#_Toc12031)

[1.8 控制台输出和波形图 12](#_Toc8182)

[1.9 遇到问题和解决方法 14](#_Toc20073)

[1.10实验心得、意见和建议 14](#_Toc20663)

[2 实验3 简单时序电路设计 15](#_Toc11212)

[2.1 任务描述 15](#_Toc5057)

[2.2 相关知识 15](#_Toc21169)

[2.3 实验内容 16](#_Toc18981)

[2.4 遇到的问题及解决方法 27](#_Toc15596)

[2.5 实验心得、意见和建议 27](#_Toc14176)

[3 实验4 数据通路和有限状态机设计 28](#_Toc29973)

[3.1 任务描述 28](#_Toc1142)

[3.2 相关知识 28](#_Toc19908)

[3.3 实验内容 29](#_Toc22710)

[3.4 遇到问题和解决方法 51](#_Toc23487)

[3.5 实验心得、意见和建议 51](#_Toc7610)

# 1 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

## 1.1 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。

## 1.2 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）将激励施加到被测试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）

//实例引用被测试模块（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。

## 1.3 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

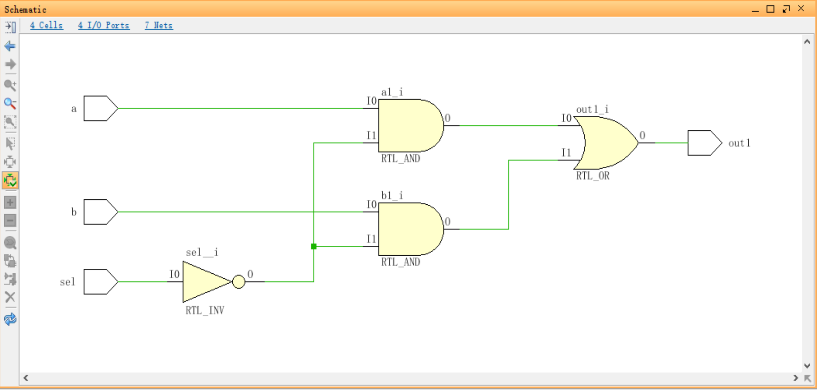


图2.2 2选1选择器Schematic

1. 编写2选1选择器电路的数据流描述模块，并生成Schematic。
2. 编写2选1选择器电路的行为描述模块，并生成Schematic。
3. 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。

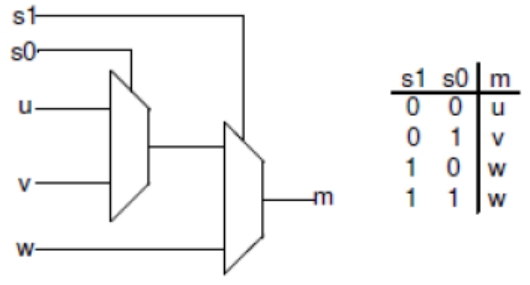


图2-3 3选1多路选择器电路和真值表

1. 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 |

1. 为上述3-8译码器编写Testbench并进行测试。

## 1.4 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。

## 1.5 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

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]

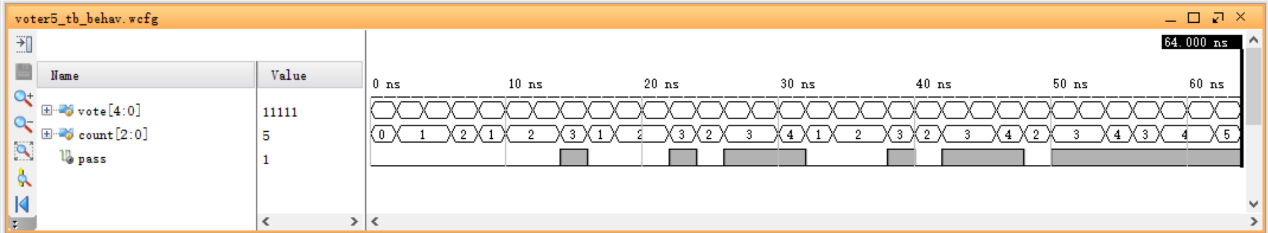
56000::[11100] [3] [1]

58000::[11101] [4] [1]

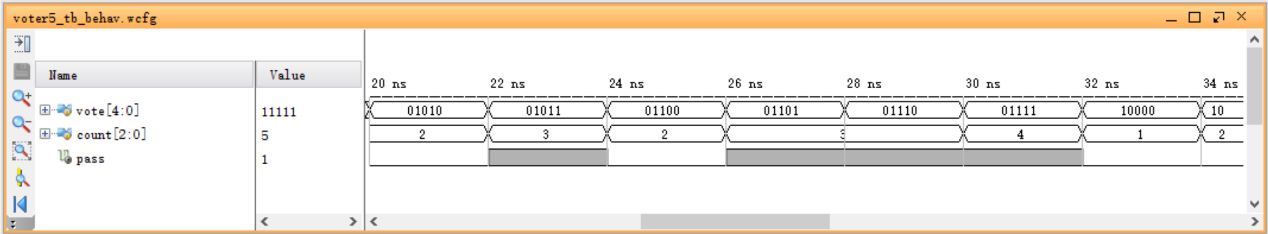
60000::[11110] [4] [1]

62000::[11111] [5] [1]

//波形图：



(a)波形



(b)波形（放大后）

图2.3 仿真波形图

## 1.6 源码

用结构描述风格编写要求的2-1选择器，源码如下：

`timescale 1ns/100ps

module mux2\_1jiegou(a,b,sel,out1);

input a,b,sel;

output out1;

wire sel\_,a1,b1;

not

RTL\_INV(sel\_,sel);

and

RTL\_AND1(a1,a,sel\_),

TRL\_AND2(b1,sel,b);

or

RTL\_OR(out1,a1,b1);

endmodule

用数据流描述风格，2选1选择器源码如下：

`timescale 1ns/100ps

module mux2\_1shujuliu(a,b,sel,out1);

input a,b,sel;

output out1;

wire sel\_,a1,b1;

assign sel\_= ~ sel;

assign a1= a & sel\_;

assign b1= sel & b;

assign out1= a1 | b1;

endmodule

用行为描述风格，2选1选择器源码如下：

`timescale 1ns/100ps

module mux2\_1xingwei(a,b,sel,out1);

input a,b,sel;

output reg out1;

reg sel\_,a1,b1;

always

@(a or b or sel) begin

sel\_<= ~ sel;

a1<=a & sel\_;

b1<=sel & b;

out1<= a1 | b1;

end

endmodule

3选1选择器源码如下：

`timescale 1ns/100ps

module mux3\_1(u,v,w,s,m);

input u,v,w;

input [1:0] s;

output wire m;

wire t1;

mux2\_1jiegou

h1(u,v,s[0],t1),

h2(t1,w,s[1],m);

endmodule

3-8译码器源码：

`timescale 1ns/100ps

module decoder\_38(F,CBA);

input [2:0] CBA;

output reg [7:0] F;

always

@(CBA) begin

case(CBA)

3'b000: F = 8'b11111110;

3'b001: F = 8'b11111101;

3'b010: F = 8'b11111011;

3'b011: F = 8'b11110111;

3'b100: F = 8'b11101111;

3'b101: F = 8'b11011111;

3'b110: F = 8'b10111111;

3'b111: F = 8'b01111111;

endcase

end

Endmodule

## 1.7 Testbench代码

3-8译码器的tb文件源码：

`timescale 1ns/100ps

module decoder\_38\_tb();

reg [2:0] CBA;

wire [7:0] F;

reg [3:0] pal;

decoder\_38 M(F,CBA);

initial begin

$display("$time::[CBA] [F]-------");

for(pal = 0;pal < 8;pal = pal + 1)

begin

CBA <= pal;

#50 $monitor("%t::",$time,"[%b]\t[%b]",CBA,F);

end

end

endmodule

## 1.8 控制台输出和波形图

如下是3-8译码器的仿真时控制台所显示的信息，可以看出在源码中设置了每50nm则pal的值加1，并赋给CBA，对应的控制台输出如下是正确的。

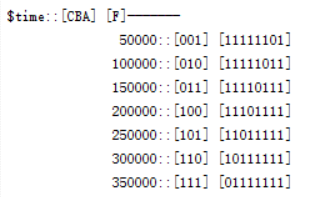


图2.4 译码器3-8的控制台输出

从如下波形图可以直观地看到，CBA随着pal的变化而变化，直到CBA的值为111b，而随着CBA的变化，P也随着题目所要求的那样变化，可见仿真结果是正确的。

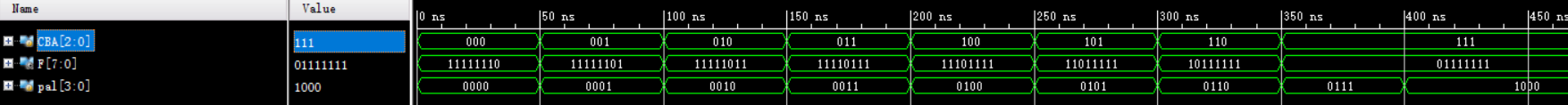


图2.5 译码器3-8的波形图

## 1.9 遇到问题和解决方法

1. 在进行行为描述的时候，因为语句中被赋值的变量类型不是寄存器型，导致出错。解决方法是改成reg型。
2. 在编写testbench代码时，在调用函数的过程中，我并没有将输出的类型设置为wire型，结果就一直进行仿真。
3. 在做3-8译码器时，系统始终会报如图2.6的两个错误，问了老师后，得知老师可以仿真成功，于是我重新创建了一个项目，结果就仿真成功了...按照老师所估计的，应该是vivado始终只允许cache中的内容，而不进行刷新。

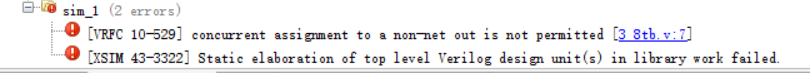


图2.6 系统报错

## 1.10实验心得、意见和建议

在本次实验过程中，我更深刻地理解了vivado的使用方法，在前两次课中，一直都是老师在讲或者是更着老师所给的资料，模仿着去做，而这些都不能让我们真正地学会去使用vivado。

然后经过本次的实验课，我切实地操作了软件，并使用自己编写的代码成功地完成了仿真，虽然在实验过程中遇到了许多的困难，比如说verilog语言中地一些具体语法没办法很好地掌握，因此在编程过程中会遇到许多的错误，但是有老师，同学以及网络上的许多资源，最终都可以解决掉这些问题，并且更深层次地理解知识。在自己动手操作之前，我总是觉得这门课不好学并且很枯燥，但当真正动手操作并且成功完成了仿真工作后，才发现这门课很有意思，也能从中学会很多知识。果然万事开头难，只要坚持下去，就能有收获。

# 2 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

## 2.1 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。

## 2.2 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。

## 2.3 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：

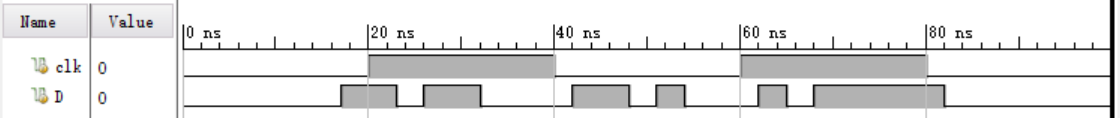
module D\_ff\_n(input clk, input D, output reg Q);

always @(negedge clk) //时钟下降沿触发

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：

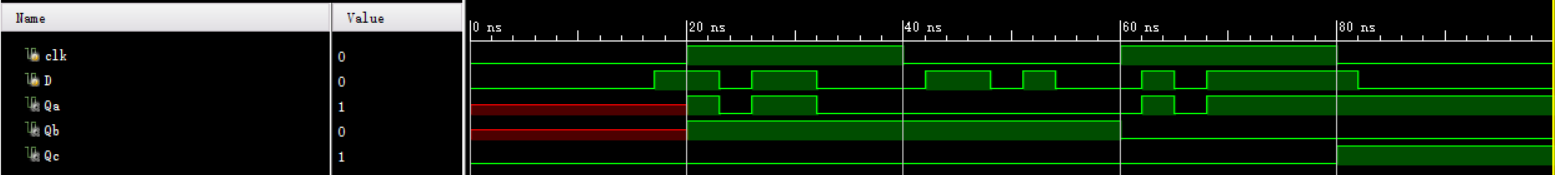


图3-1 lab3\_1的仿真波形

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) begin

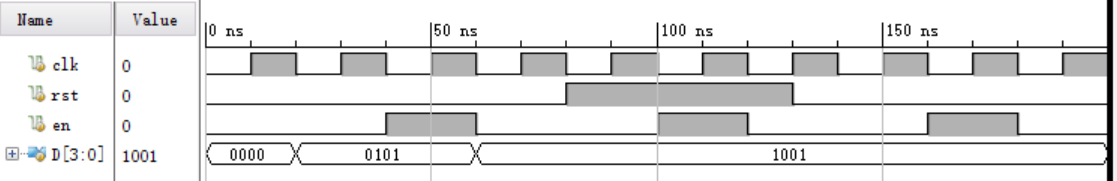
if(rst) Q <= 4'b0;

else if(en) Q <= D;

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 clk;

reg [0:3] D;

wire [0:3] Q;

initial begin

clk =0;

rst =0;

en =0;

#200 $stop;

end

always begin

#10 clk=!clk;

#80 rst=!rst;

#40 en=!en;

end

initial begin

D=4'b0;

#20 D=4'b0101;

#40 D=4'b1001;

end

Register\_synch\_reset\_load h1(D,clk,rst,en,Q);

endmodule

(3) 仿真波形图：

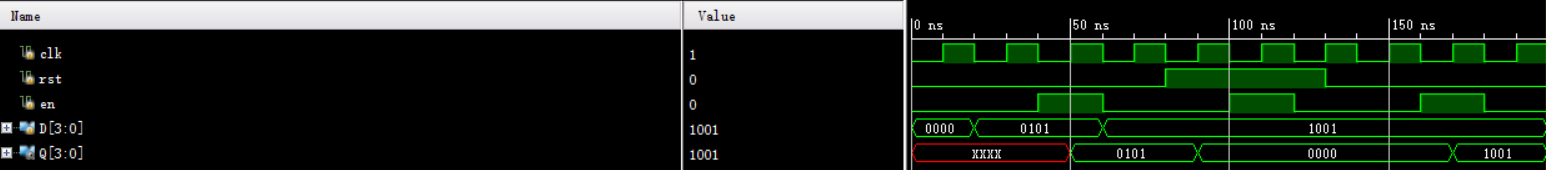


图3-2 lab3\_2的仿真波形

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 = shift\_reg[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骞惰杈撳嚭淇″彿

output reg ShiftOut; // 绉讳綅杈撳嚭淇″彿

reg [3:0] shift\_reg = 4'b0000;

always @(posedge clk)

if(ShiftEn) begin

ShiftOut <= shift\_reg[3];

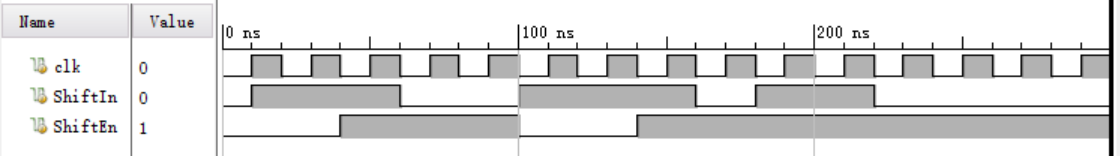
shift\_reg <= {shift\_reg[2:0],ShiftIn};

end

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,ShiftIn,ShiftEn;

wire ShiftOut;

wire [3:0] ParallelOut;

initial begin

clk=0;

ShiftIn=0;

ShiftEn=0;

#300 $stop;

end

always begin

#10 clk=!clk;

end

initial begin

#10 ShiftIn=!ShiftIn;

#50 ShiftIn=!ShiftIn;

#40 ShiftIn=!ShiftIn;

#60 ShiftIn=!ShiftIn;

#20 ShiftIn=!ShiftIn;

#40 ShiftIn=!ShiftIn;

end

initial begin

#40 ShiftEn=!ShiftEn;

#60 ShiftEn=!ShiftEn;

#40 ShiftEn=!ShiftEn;

end

Serial\_in\_Parallel\_out\_enable h1(clk,ShiftEn,ShiftIn,ParallelOut,ShiftOut);

endmodule

(3) 仿真波形图：

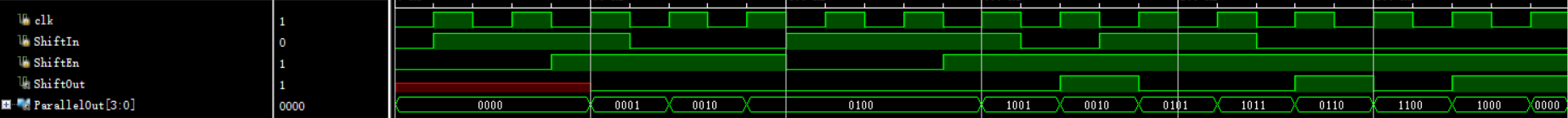


图3-3 lab3\_3的波形图

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加法计算器（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);

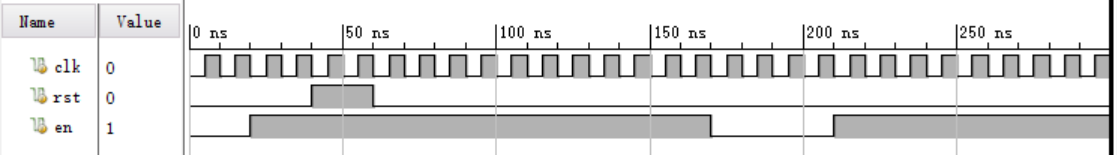
always @(posedge clk)

if(rst) Q<=0;

else if(en) Q<=Q+1;

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,rst,en;

wire [3:0] Q;

initial begin

clk=0;

rst=0;

en=0;

#300 $stop;

end

always

#5 clk=!clk;

initial begin

#40 rst=!rst;

#20 rst=!rst;

end

initial begin

#20 en=!en;

#150 en=!en;

#40 en=!en;

end

CNT4\_synch\_reset\_enable h1(clk,rst,en,Q);

endmodule

(3) 仿真波形图：

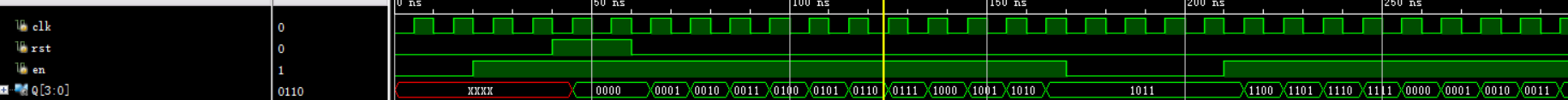


图3-4 lab3\_4的波形图

## 2.4 遇到的问题及解决方法

1. 在编写代码时，出现了没有加分号或者是”initial”单词拼写错误。
2. 出现仿真错误，如下图，仔细检查代码，发现是在仿真模块没有对部分变量进行说明。

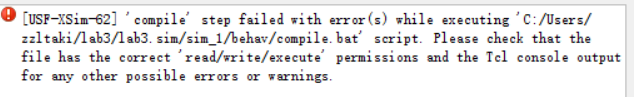


图3-5 仿真过程遇到的问题

1. 在进行实验3\_2时，我发现输出波形与预想的有挺大的偏差，如下图。首先是输入波形的偏差。仔细查看代码，发现是对阻塞性过程性赋值的理解不够深刻，后面的延迟赋值必须要在前面的赋值完成后才能进行。

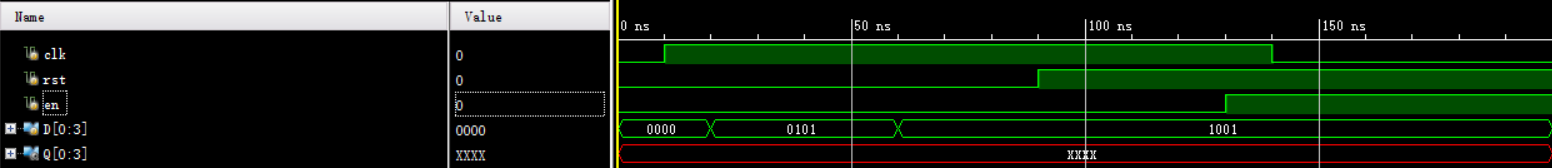


图3-6 由于时延的利用不当导致的错误

1. 在我完成了一个了一个实验任务后，进行下一个任务时，当我对任务建模后输出原理图，一直是输出上一个任务的模拟图。检查是否设置set as top以及reload，结果发现全部都进行过的。然后再仔细检查系统报错时才发现我有个变量的名字拼写错了，当我改正并保存后，再输出原理图就OK了。

## 2.5 实验心得、意见和建议

本次实验主要是对一些简单时序逻辑电路的元件进行了建模、仿真和调试。

经过了上次的实验，感觉这次的实验还是比较轻松的。在本次实验中，我了解到了如何编写D触发器、D锁存器、寄存器和加法器等，有了上一节实验课的基础，在本次实验中没有遇到特别多的语法错误，只是在一些小的细节上没有把握好。同时，在这堂课上练习了许多的仿真操作对调用模块也是更加的熟练了。

# 3 实验4 数据通路和有限状态机设计

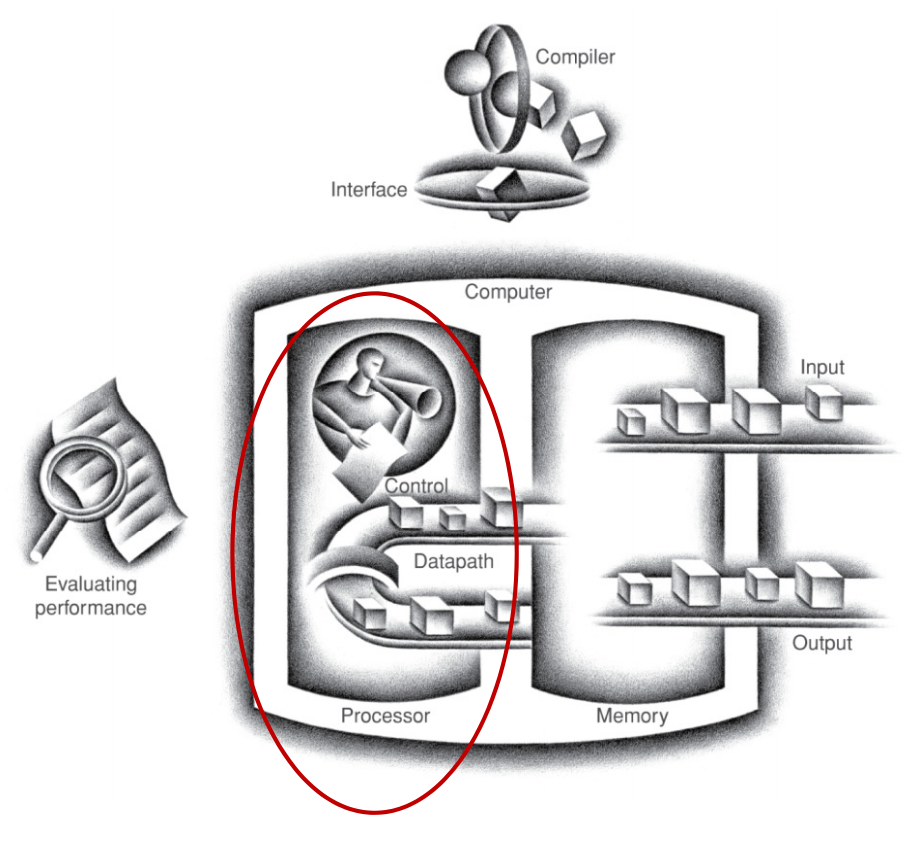
* 任务描述
* 相关知识
* 实验内容
* 遇到问题和解决方法
* 实验心得、意见和建议

## 3.1 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）。

## 3.2 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）。



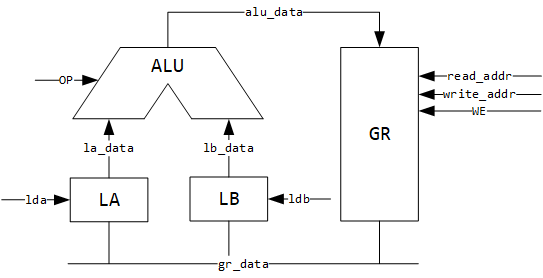
**图1处理器由数据通路和控制器构成**

## 3.3 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**

首先实现各个单元的模块：

**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 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;**

**2'b11: q = a | b;**

**default: q = 0;**

**endcase**

**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];**

**initial begin //对存储器初始化**

**$readmemh("ram\_init.txt", ram); end**

**always @(posedge clk) begin**

**if (we)**

**ram[write\_addr] <= data;**

**q <= ram[read\_addr];**

**end**

**endmodule**

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);**

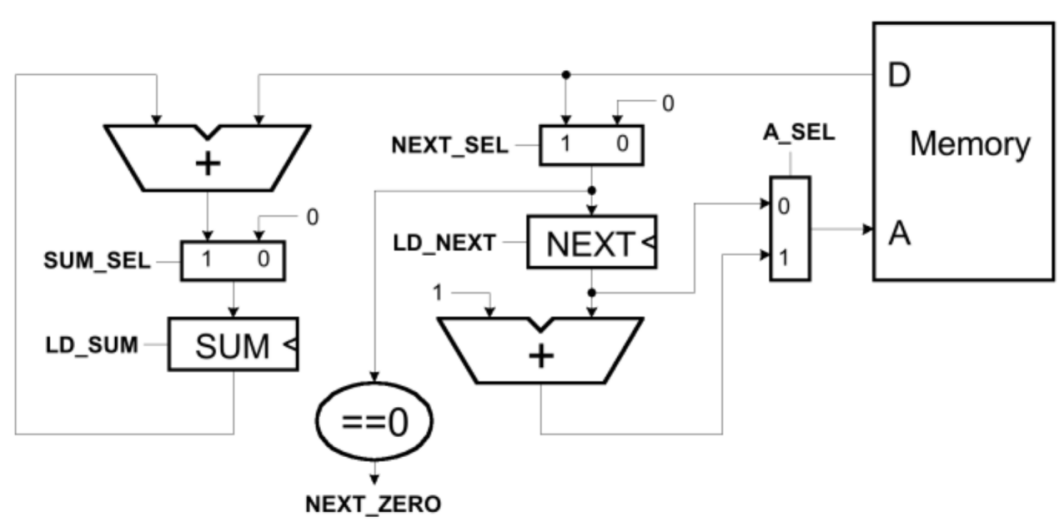
**endmodule**

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。



**sum\_out**

**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。

1. 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。

1. 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）

module mux2\_1(a,b,sel,q);

parameter WIDTH = 8;

input [WIDTH-1:0] a,b;

input sel;

output reg [WIDTH-1:0] q;

always @(\*) begin

if(sel==0) q=a;

else q=b;

end

endmodule

module compare(a,b,NEXT\_ZERO);

parameter WIDTH=8;

input [WIDTH-1:0] a,b;

output reg NEXT\_ZERO;

always @(\*) begin

if(a==b) NEXT\_ZERO=1;

else NEXT\_ZERO=0;

end

endmodule

module register(clk, rst, load, D, Q);

parameter WIDTH = 8;

input clk, rst, load;

input [WIDTH-1:0] D;

output reg [WIDTH-1:0] Q;

always @(posedge clk) begin

if (rst) Q <=0;

else if (load) Q <= D;

end

endmodule

module add(a,b,sum);

parameter WIDTH = 8;

input [WIDTH-1 : 0] a,b;

output reg [WIDTH-1 : 0] sum;

always @(\*) begin

sum=a+b;

end

endmodule

module ram(read\_addr,data);

parameter DATA\_WIDTH = 8;

parameter ADDR\_WIDTH = 3;

input [ADDR\_WIDTH-1:0] read\_addr;

output reg [DATA\_WIDTH-1:0] data;

reg [DATA\_WIDTH-1:0] ram[2\*\*ADDR\_WIDTH-1:0];

initial begin

$readmemh("C:/Users/zzltaki/Desktop/ram\_init.txt",ram);

end

always @(\*) begin

data<=ram[read\_addr];

$display("%t: %b %b",$time,read\_addr,data);

end

endmodule

1. 数据通路的代码：

module datapath(clk,rst,SUM\_SEL,NEXT\_SEL,A\_SEL,LD\_SUM,LD\_NEXT,NEXT\_ZERO,sum\_out);

input clk,rst,SUM\_SEL,NEXT\_SEL,A\_SEL,LD\_SUM,LD\_NEXT;

output NEXT\_ZERO;

output [31:0] sum\_out;

wire [3:0] read\_addr;

wire [31:0] next1,next2,data,add\_out,sum\_in,NEXT\_IN,ad\_data;

mux2\_1 #(32) m1(32'h0000,data,NEXT\_SEL,NEXT\_IN);

compare #(32) com(NEXT\_IN,32'h0000,NEXT\_ZERO);

register #(32) NEXT(clk,rst,LD\_NEXT,NEXT\_IN,next1);

add #(32) a1(32'h0001,next1,next2);

mux2\_1 #(32) m2(next1,next2,A\_SEL,ad\_data);

assign read\_addr=ad\_data[3:0];

ram #(32,4) Memory(read\_addr,data);

add #(32) a2(data,sum\_out,add\_out);

mux2\_1 #(32) m3(32'h0000,add\_out,SUM\_SEL,sum\_in);

register #(32) SUM(clk,rst,LD\_SUM,sum\_in,sum\_out);

endmodule

1. 数据通路的电路原理图（生成Schematic）：

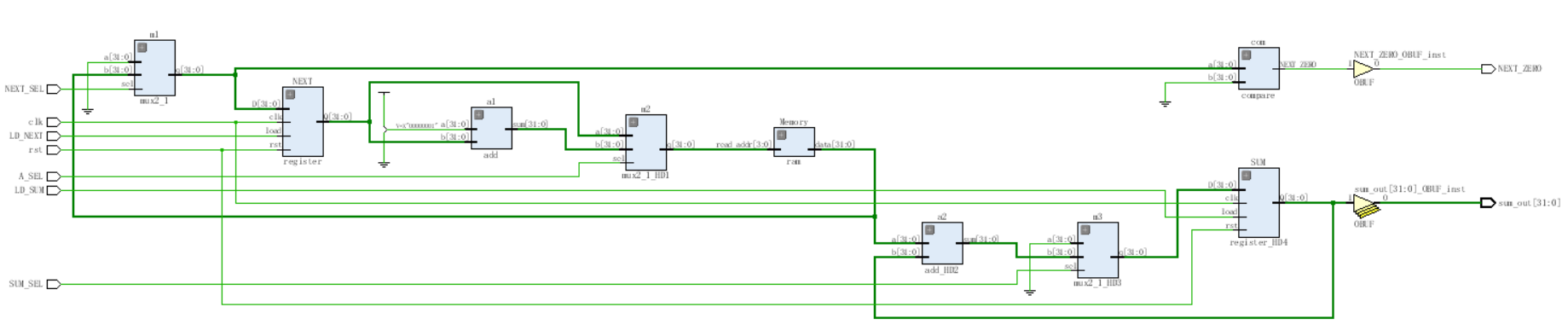


图4.1 数据通路原理图

**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)。

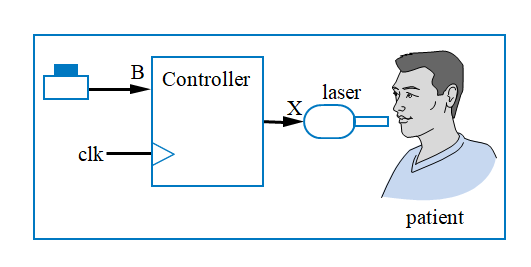


图4 激光计时器

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。

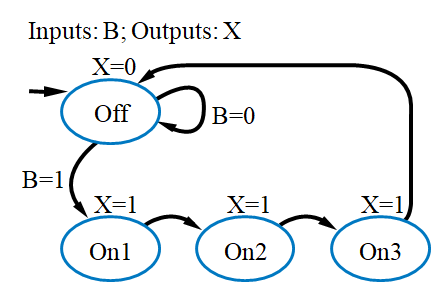


图5 该激光计时器的有限状态机

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;**//对应四个状态

**reg [1:0] State, StateNext;** //当前状态和下一个时钟周期的状态

(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

**StateNext <= S\_Off;** //下一个状态自动迁移为Off

**end**

**endcase**

**end**

**// StateReg**

**always @(posedge Clk) begin**

**if (Rst == 1 )**

**State <= S\_Off;** //激光器复位

**else**

**State <= StateNext;** //迁移到下一个状态

**end**

**endmodule**

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);**

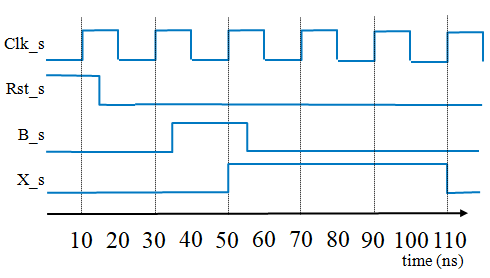
**#5 if (X\_s != 0)** //验证状态Off

**$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：

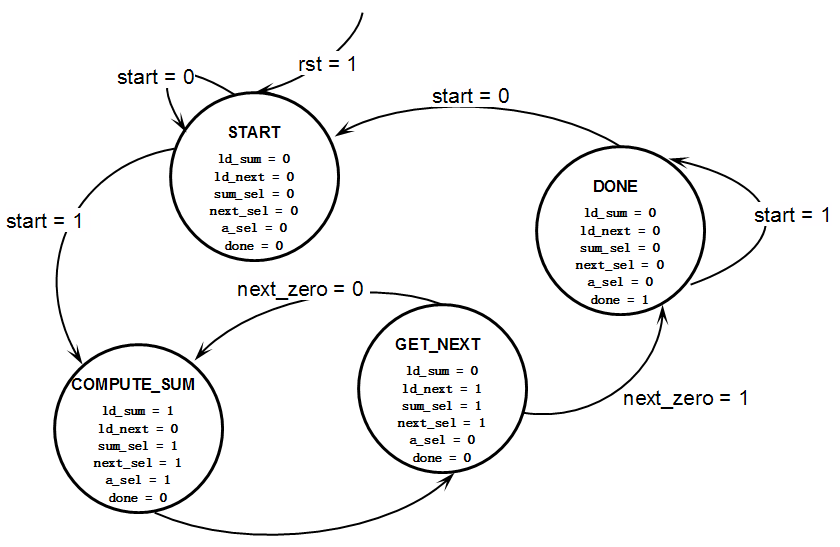
* 设计的有限状态机（FSM）能够正确综合；
* 编写有限状态机的仿真程序，完成有限状态机（FSM）的功能仿真，有限状态机功能仿真正确。

【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);

input clk,rst,start,next\_zero;

output reg LD\_SUM,LD\_NEXT,SUM\_SEL,NEXT\_SEL,A\_SEL,DONE;

parameter START=0,COMPUTE\_SUM=1,GET\_NEXT=2,DO\_NE=3;

reg [1:0] State,StateNext;

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==1)

StateNext<=COMPUTE\_SUM;

else

StateNext<=START;

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<=DO\_NE;

end

DO\_NE:begin

LD\_SUM<=0;

LD\_NEXT<=0;

SUM\_SEL<=0;

NEXT\_SEL<=0;

A\_SEL<=0;

DONE<=1;

if(start==1)

StateNext<=DO\_NE;

else

StateNext<=START;

end

endcase

end

always @(posedge clk) begin

if(rst==1)

State<=START;

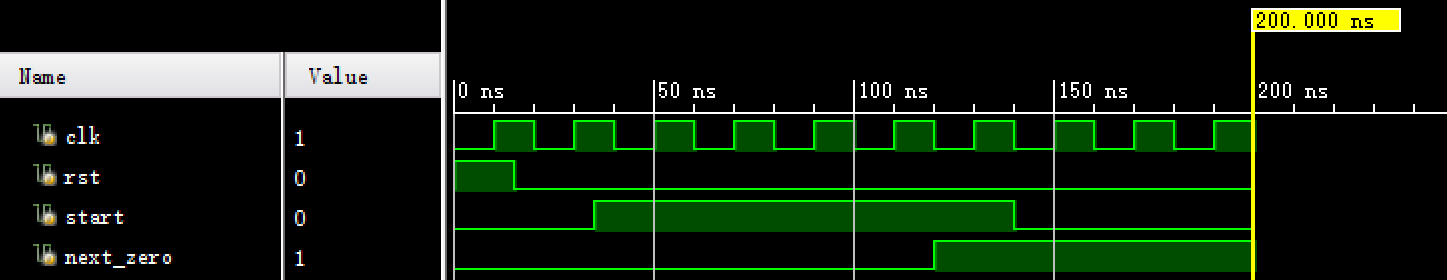
else

State<=StateNext;

end

endmodule

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;

wire LD\_SUM,LD\_NEXT,SUM\_SEL,NEXT\_SEL,A\_SEL,DONE;

FSM dut(clk,rst,start,next\_zero,LD\_SUM,LD\_NEXT,SUM\_SEL,NEXT\_SEL,A\_SEL,DONE);

always begin

clk<=0;

#10;

clk<=1;

#10;

end

initial begin

rst<=1;

#15 rst<=0;

end

initial begin

start<=0;

#35 start<=1;

#105 start<=0;

end

initial begin

next\_zero<=0;

#120 next\_zero<=1;

end

endmodule

1. 仿真结果图：

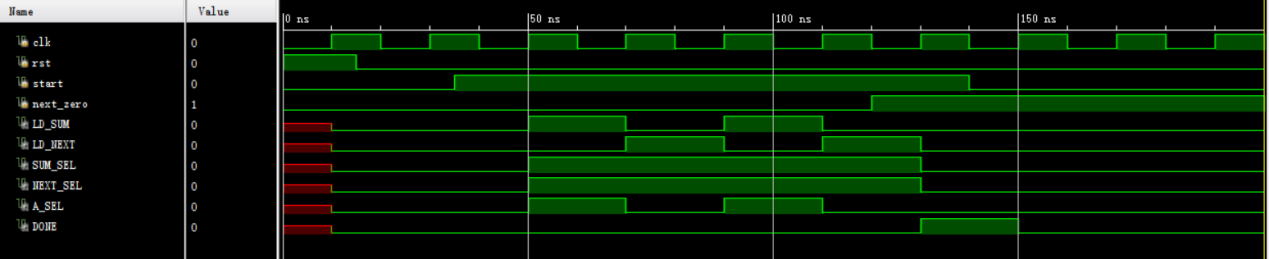


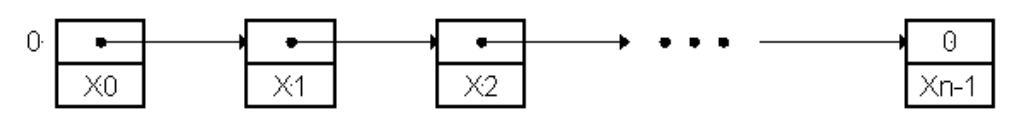
图4.2 有限状态机仿真图

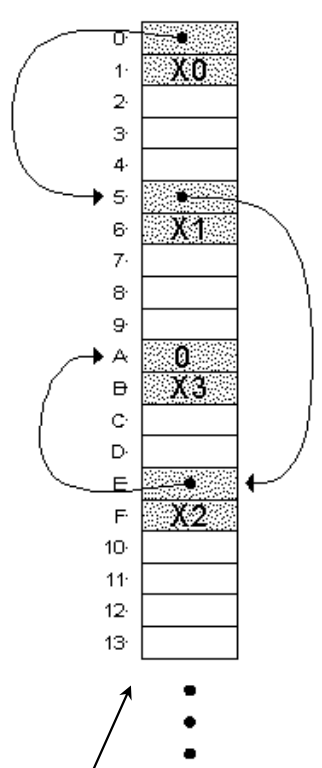
**3. 自动运算电路的设计（步骤3）**

【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。

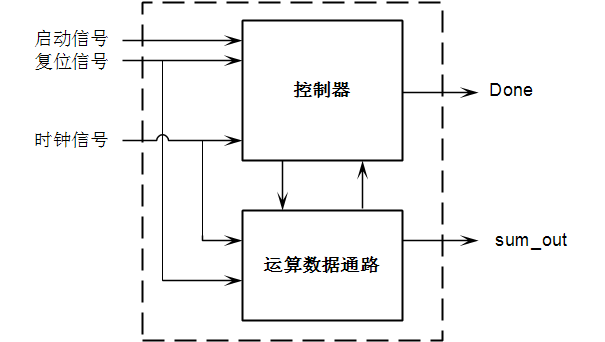




**图7 数据链表及其在存储器中的存放格式**

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);

input clk,rst,start;

output DONE;

output [31:0] sum\_out;

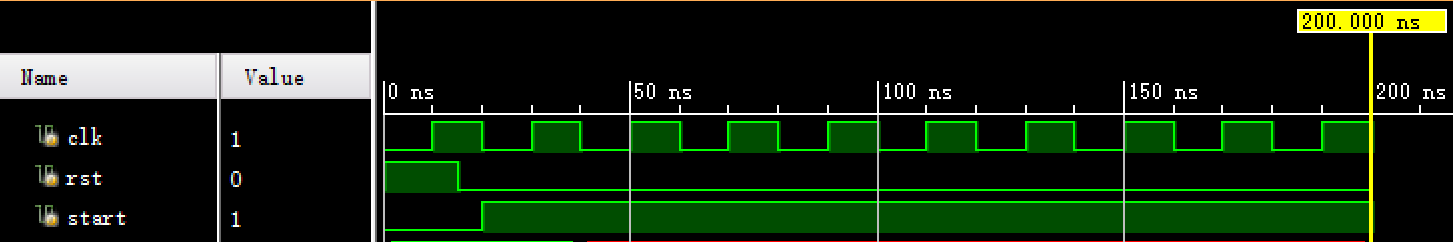
wire LD\_SUM,LD\_NEXT,SUM\_SEL,NEXT\_SEL,A\_SEL,NEXT\_ZERO;

FSM dut1(clk,rst,start,NEXT\_ZERO,LD\_SUM,LD\_NEXT,SUM\_SEL,NEXT\_SEL,A\_SEL,DONE);

datapath dut2(clk,rst,SUM\_SEL,NEXT\_SEL,A\_SEL,LD\_SUM,LD\_NEXT,NEXT\_ZERO,sum\_out);

endmodule

1. 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 DONE;

wire [31:0] sum\_out;

auto\_add h1(clk,rst,start,DONE,sum\_out);

always begin

clk<=0;

#10;

clk<=1;

#10;

end

initial begin

rst<=1;

#15 rst<=0;

end

initial begin

start<=0;

#20 start<=1;

end

endmodule

1. 仿真结果图：

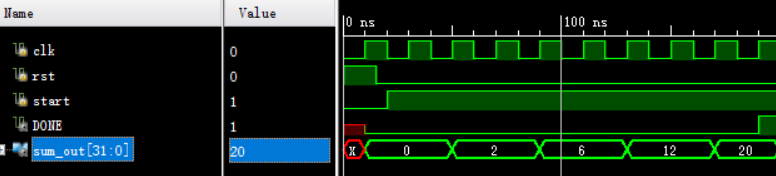


图4.3 自动运算电路仿真图

## 3.4 遇到问题和解决方法

首先在实现数据通路中的存储器时，我开始一直考虑存储器应该有两个输出端口，一个用于累加，一个用于获取下一个地址，而没有想到这里巧妙地使用了一个二选一选择器，并通过寄存器锁存和开启的方式来控制。

在实现自动运算电路时，选择的文件我是以绝对路径的方式读取的，但在写路径时，我使用了符号“\”，于是仿真时的输出波形如图4.4所示。不曾想在vivado中需要使用“/”间隔符。

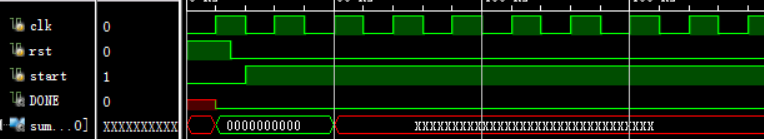


图4.4 实现自动运算电路时出错

## 3.5 实验心得、意见和建议

本次实验是一个综合性实验，层层递进，分别从数据通路和有限状态机入手，然后将两者结合，推进到自动运算电路，让学生能够接受。

在这个综合性的实验中，跟着实验要求一步一步慢慢走，遇到了一些棘手的问题，但通过询问老师顺利地解决了。当完成了实验后，感觉有点不可思议，回想起来感觉实验难度很大。完成实验后很有成就感。