

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** | 杨雨鑫 |
| **学号** | U201814655 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **2020** | **年** | **5** | **月** | **13** | **日** |

目录

**1** [**任务二描述 3**](#_Toc40295524)

**1.1** [**相关知识 3**](#_Toc40295525)

**1.2** [**实验步骤 4**](#_Toc40295526)

**1.3** [**编程要求 8**](#_Toc40295527)

**1.4** [**测试说明 8**](#_Toc40295528)

**1.5** [**源码 9**](#_Toc40295529)

**1.6** [**Testbench代码 12**](#_Toc40295530)

**1.7** [**控制台输出和波形图 13**](#_Toc40295531)

**1.8** [**遇到问题和解决方法 15**](#_Toc40295532)

**1.9** [**实验心得、意见和建议 15**](#_Toc40295533)

**2** [**任务三描述 17**](#_Toc40295534)

**2.1** [**相关知识 17**](#_Toc40295535)

**2.2** [**实验内容 18**](#_Toc40295536)

**2.3** [**遇到的问题及解决方法 31**](#_Toc40295537)

**2.4** [**实验心得、意见和建议 31**](#_Toc40295538)

**3** [**任务四描述 33**](#_Toc40295539)

**3.1** [**相关知识 33**](#_Toc40295540)

**3.2** [**实验内容 34**](#_Toc40295541)

**3.3** [**遇到问题和解决方法 59**](#_Toc40295542)

**3.4** [**实验心得、意见和建议 60**](#_Toc40295543)

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。

## 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

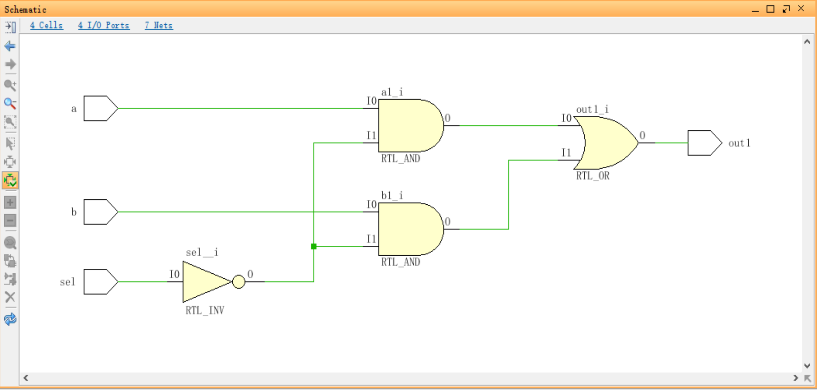


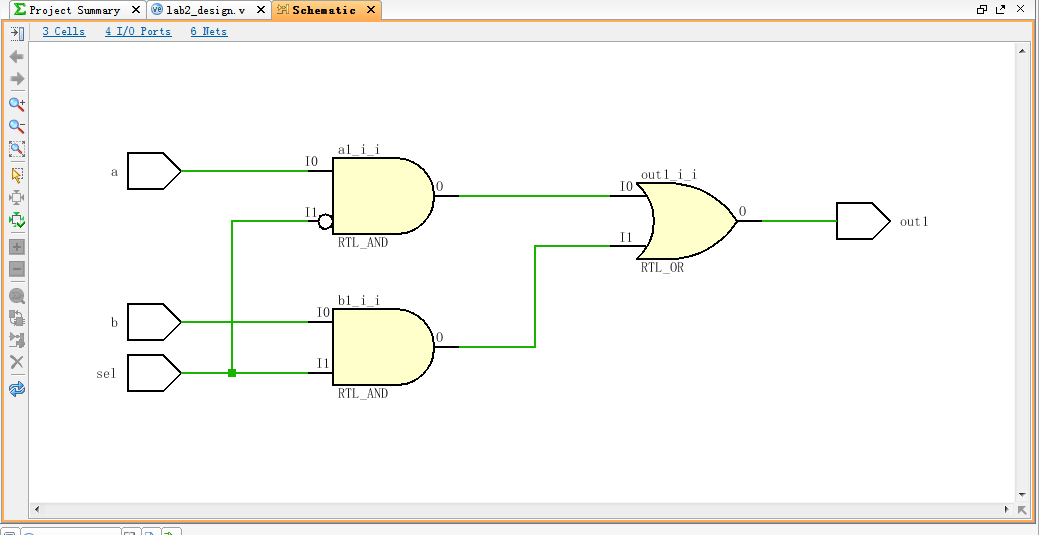
图2.2 2选1选择器Schematic

首先发现报告上的Schematic图给错了。

我的实验操作步骤：

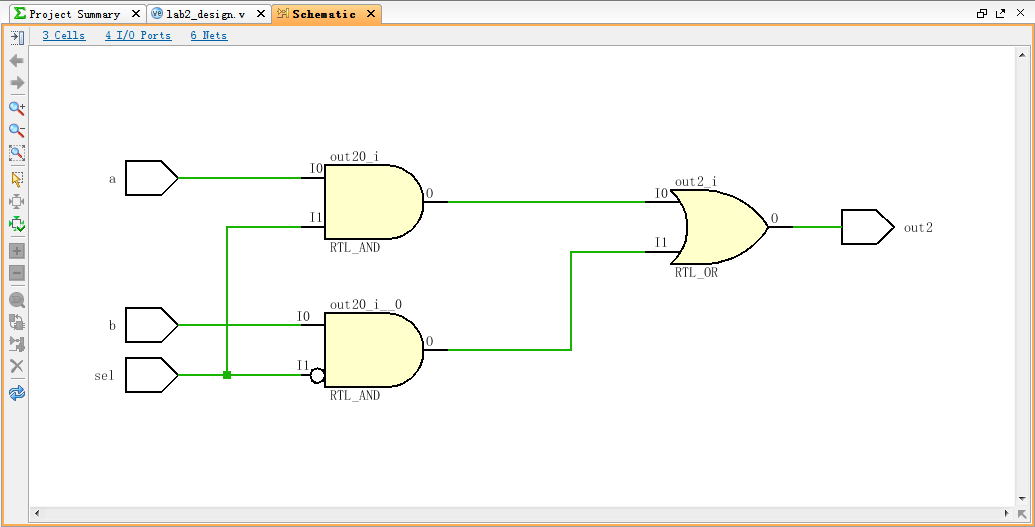
首先在vivado中创建一个空的工程，添加一个空的源文件lab2\_design，这里我还是选择的是和第一次试验相同的**XC7A100TCSG324-1**。

根据图2-1编写出结构描述模块，生成的Schematic图如下：



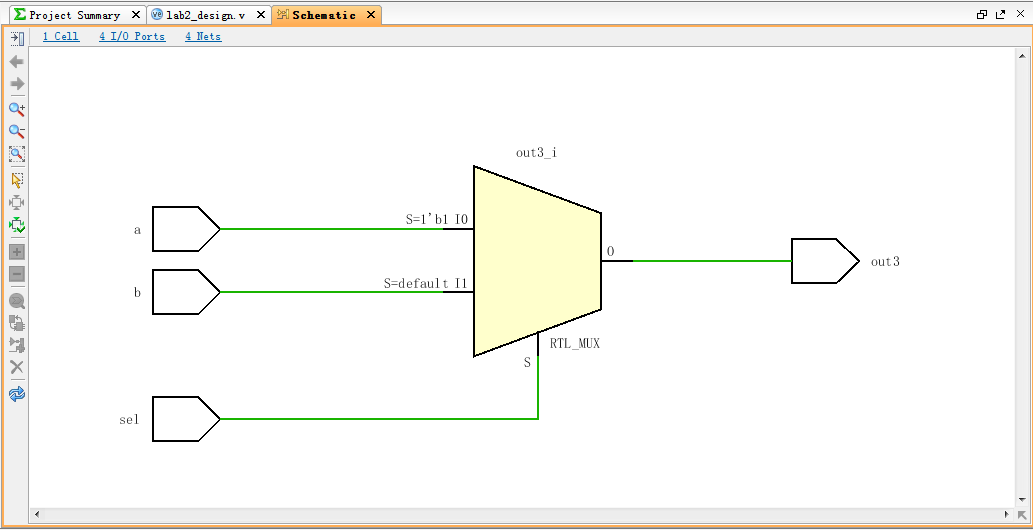
1. 编写2选1选择器电路的数据流描述模块，并生成Schematic。

根据图2-1编写出数据流描述模块，生成的Schematic图如下：



1. 编写2选1选择器电路的行为描述模块，并生成Schematic。

根据图2-1编写出行为描述模块，生成的Schematic图如下：



1. 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。

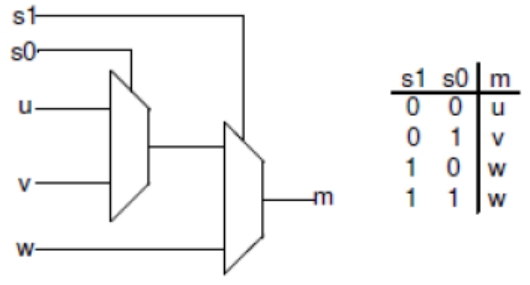
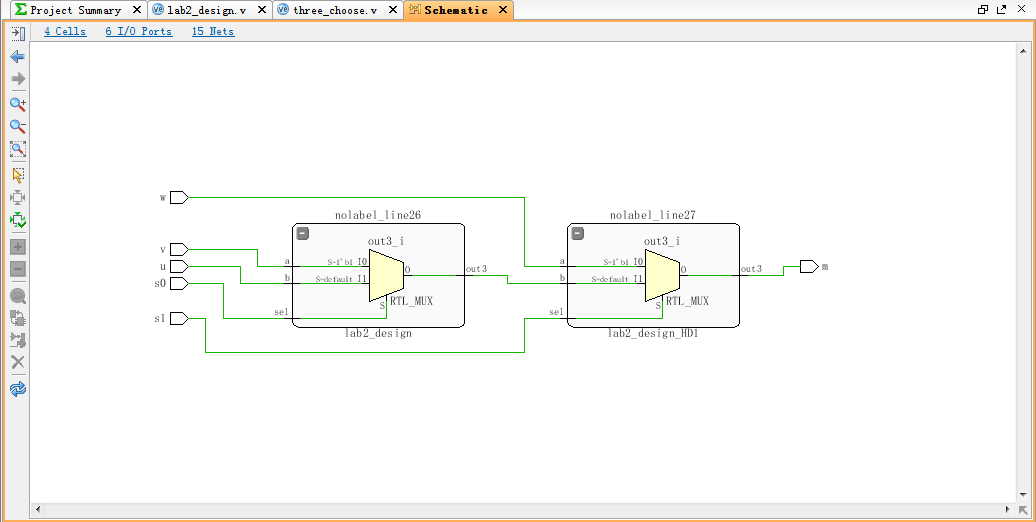


图2-3 3选1多路选择器电路和真值表

根据图2-3编写三选一多路选择器，这里我们是把两个文件放在同一个工程中，第二个便可以调用第一个模块。

生成的Schematic图如下：



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;

……

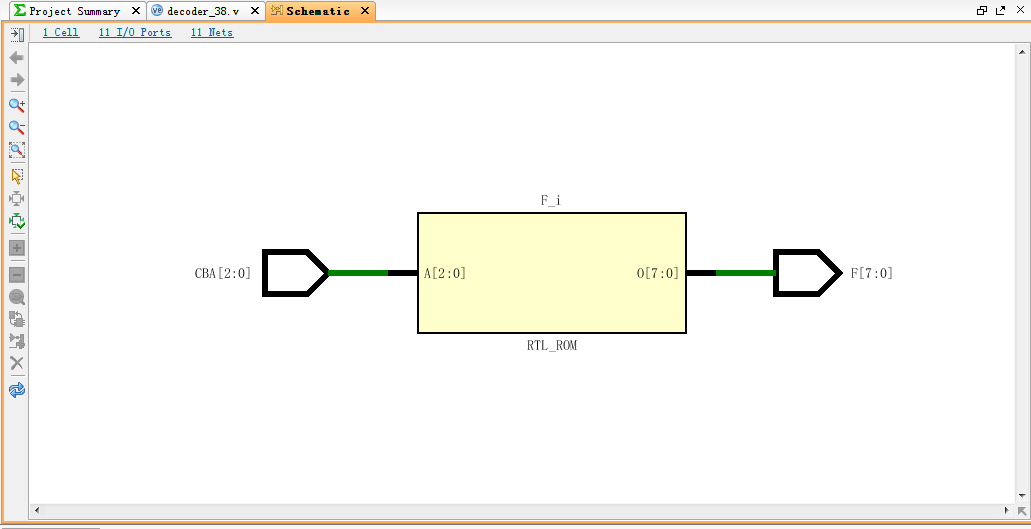
endmodule

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 |

根据所给提示要求生成的38译码器，源代码如下：

生成的Schematic图如下：



1. 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

3-8译码器模块，这里使用行为描述通过循环来给输入赋值。

控制台输出：

$time::[in] [out]------

0::[000] [11111110]

20::[001] [11111101]

40::[010] [11111011]

60::[011] [11110111]

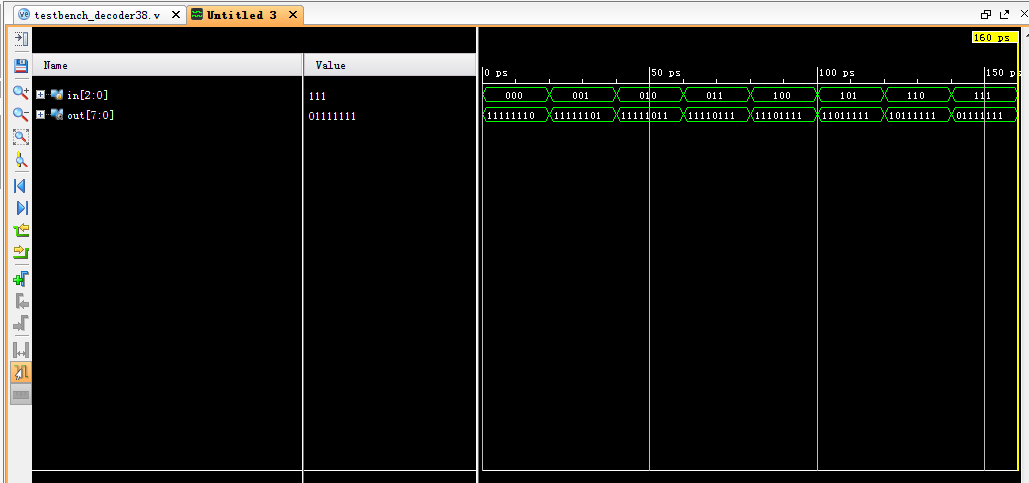
80::[100] [11101111]

100::[101] [11011111]

120::[110] [10111111]

140::[111] [01111111]

波形图：放大到比例适合：



## 源码

编写2选1选择器电路的结构描述模块

`timescale 1ns / 1ps

module lab2\_design(a,b,sel,out1);

input a,b,sel;

output out1;

wire nsel,a1,b1; //定义中间变量

not sel\_i(nsel,sel);

and a1\_i(a1,a,nsel);

and b1\_i(b1,b,sel);

or out1\_i(out1,a1,b1);

endmodule

编写2选1选择器电路的数据流描述模块：

`timescale 1ns / 1ps

module lab2\_design(a,b,sel,out2);

input a,b,sel;

output out2;

assign out2 = (a&sel) |(b&(~sel));

endmodule

编写2选1选择器电路的行为描述模块

`timescale 1ns / 1ps

module lab2\_design(a,b,sel,out3);

input a,b,sel;

output out3;

reg out3;

always @ (sel)

begin

if(sel == 1'b1)

out3 = a;

else

out3 = b;

end

endmodule

编写三选一多路选择器

`timescale 1ns / 1ps

module three\_choose(u,v,w,s0,s1,m);

input u,v,w,s0,s1;

output m;

lab2\_design(v,u,s0,temp1);

lab2\_design(w,temp1,s1,m);

endmodule

设计一个3-8译码器模块

`timescale 1ns / 1ps

module decoder\_38decoder\_38(F, CBA);

input [2:0] CBA;

output reg [7:0] F;

always @(CBA)

begin

case(CBA)

3'd0: F = 8'b1111\_1110;

3'd1: F = 8'b1111\_1101;

3'd2: F = 8'b1111\_1011;

3'd3: F = 8'b1111\_0111;

3'd4: F = 8'b1110\_1111;

3'd5: F = 8'b1101\_1111;

3'd6: F = 8'b1011\_1111;

3'd7: F = 8'b0111\_1111;

endcase

end

endmodule

（要标明是什么具体任务对应的源码）

## Testbench代码

这里是对于38译码器而写的testbench代码：

`timescale 10ps / 1ps

module testbench\_decoder38;

reg [2:0] in;

wire [7:0] out;

decoder\_38 test(.F(out), .CBA(in));

initial

begin

$display ("$time::[in] [out]------");

$monitor ("%t::", $time, "[%b]\t[%b]", in, out);

end

initial begin

for (in = 0; in <= 3'b111; in = in + 1)

#2;

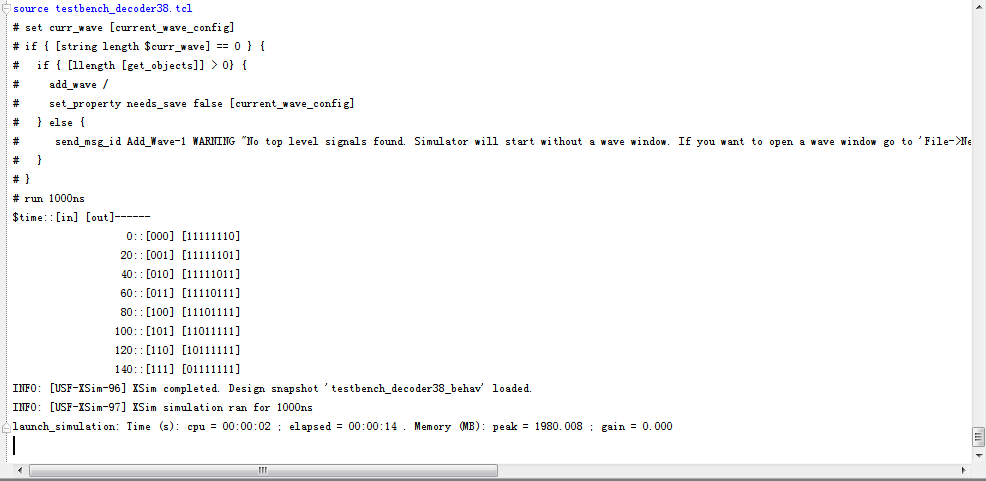
#2 $stop;

end

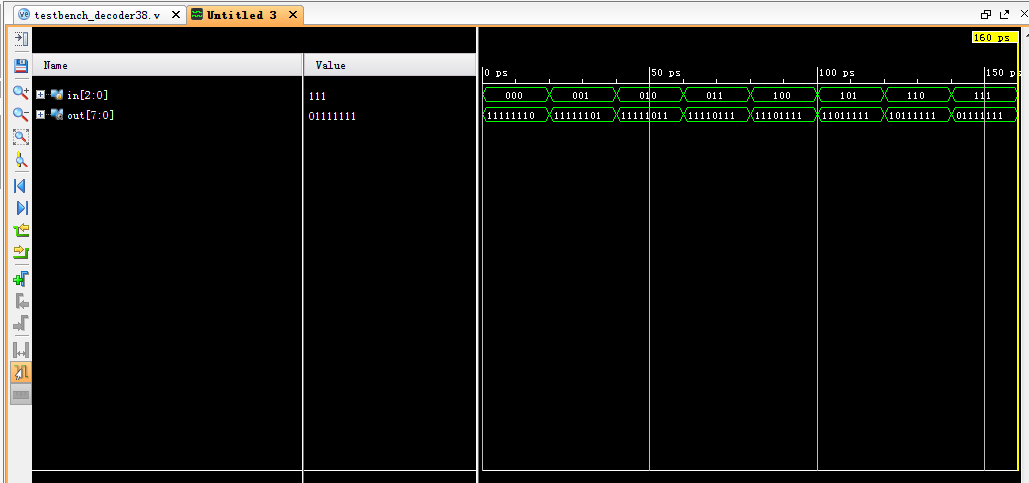
endmodule

## 控制台输出和波形图

这是控制台输出：



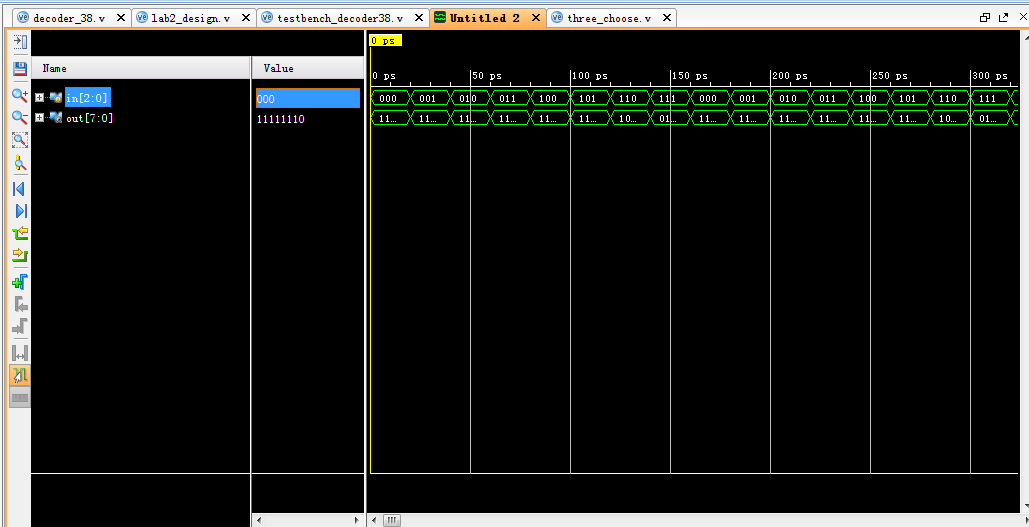
这是decoder\_38的输出波形：



修改循环条件后改为无限循环后的控制台输出：



修改条件后改为无限循环后的输出波形：



## 遇到问题和解决方法

这次实验，基本上除了前面四个模块是自己写的之外，testbench部分是使用老师的代码修改而得到的，因此遇到了报错，但是vivado有的报错会显示行数，但是有些报错就仅仅是报错，并不知道自己错在哪里，最后我只能重写了一遍。

编写前面几个的模块时候，在用多种方法描述的时候，我发现只能够把语句写成最简单的形式，例如a&b这种形式，系统才会使用与门这种基本电路来描述，否则直接就变成了个选择模块，并不知道内部的具体结构。

在生成Schematic图的时候，我发现必须把该文件set to top才能够生成该文件的Schematic图，此外，在一个模块中调用了该工程中的另外一个模块，会自动在source文件目录下看到生成了调用模块的子目录，方便我观察自己调用了哪些模块。

在第三次实验课上，老师单独对我的实验二进行了检查，并且在testbench文件中发现了我的问题，之前我设计for循环的时候，把判断条件直接设置成了3’b111，但是老师说希望我把循环条件改为<=3’b111这样的话就可以不断地重复输出译码值，修改后可以更加全面的看出来这是一个3-8译码器，进一步说明了这个系统在所有的输入可能情况下都可以正常工作。

这次试验的调试我开始也不是特别的熟练，在老师的指导下才知道看message框来寻找错误，但是有的时候报错只会在控制台上显示error，因此对于查看报错我才知道有两个地方可以看。

在一个工程中编写多个文件的时候，如果要生成一个设计文件的Schematic图，需要先把该文件set as top之后再查看原理图。在生成仿真的时候也是一样，需要把tb文件set as top。

## 实验心得、意见和建议

这次试验，编写的程序都是十分简单的，但是这也是我第一次动手编写verilog程序，对于一些语法还不是特别的熟悉，希望在之后的实验中能够逐渐熟悉起来。对于一些不是特别明确的报错还是尽量避免，对于仿真部分，下次可以考虑的更加详细，最好能找到自己程序可能产生的漏洞，比如万一发生了错误输入或是错误输出，最好有一个default的步骤，也可以在实际中避免一些考虑不周到的地方。

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：

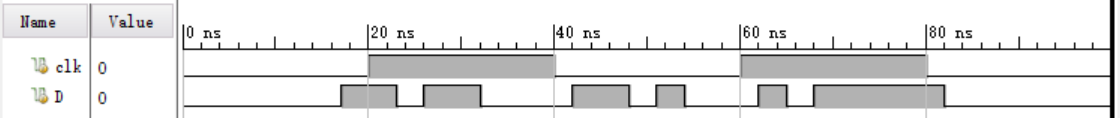
module D\_ff\_n(input clk, input D, output reg Q);

always @ (negedge clk) // 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

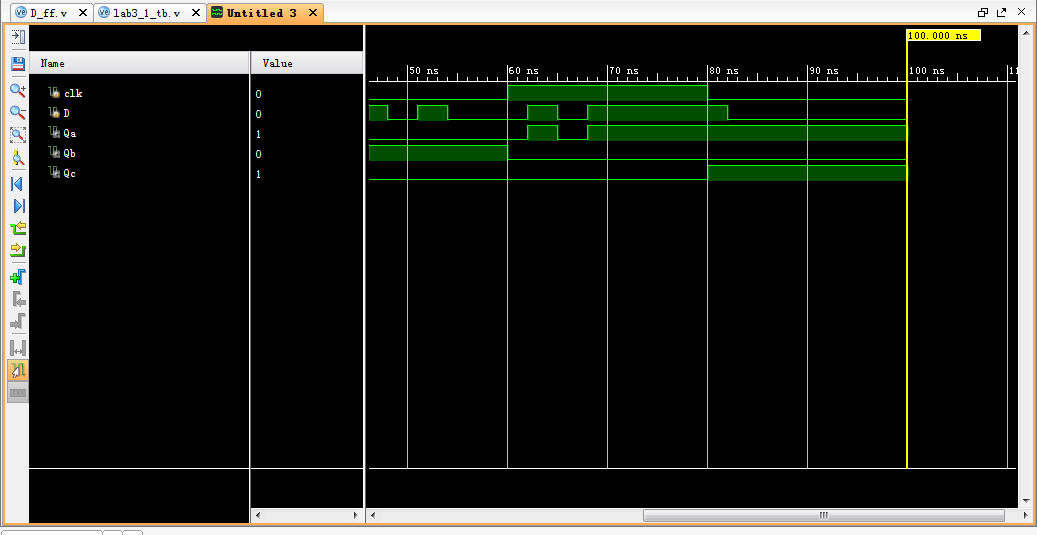
D\_latch myDlatch(clk, D, Qa);

D\_ff myDff(clk, D, Qb);

D\_ff\_n myDffn(clk, D, Qc);

endmodule

(3) 仿真波形图：



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

always @(posedge clk)

if (rst) begin // 同步复位

Q <= 4'b0;

end

else begin

if (en) begin

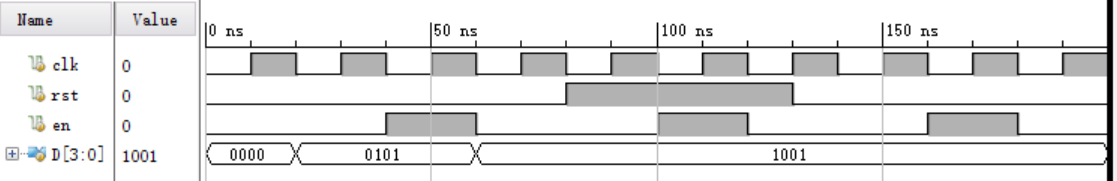
Q<=D;

end

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 in\_clk,in\_rst,in\_en;

reg [3:0] in\_D;

wire [3:0] out\_Q;

Register\_synch\_reset test(.D(in\_D), .clk(in\_clk), .rst(in\_rst), .en(in\_en), .Q(out\_Q));

initial begin

in\_clk=1'b0;

end

always

#10 in\_clk= !in\_clk;

initial begin

in\_rst=1'b0;

#80 in\_rst= !in\_rst;

#50 in\_rst= !in\_rst;

#70 $stop;

end

initial begin

in\_en=1'b0;

#40 in\_en= !in\_en;

#20 in\_en= !in\_en;

#40 in\_en= !in\_en;

#20 in\_en= !in\_en;

#40 in\_en= !in\_en;

#20 in\_en= !in\_en;

#20 $stop;

end

initial begin

in\_D=4'b0000;

#20 in\_D=4'b0101;

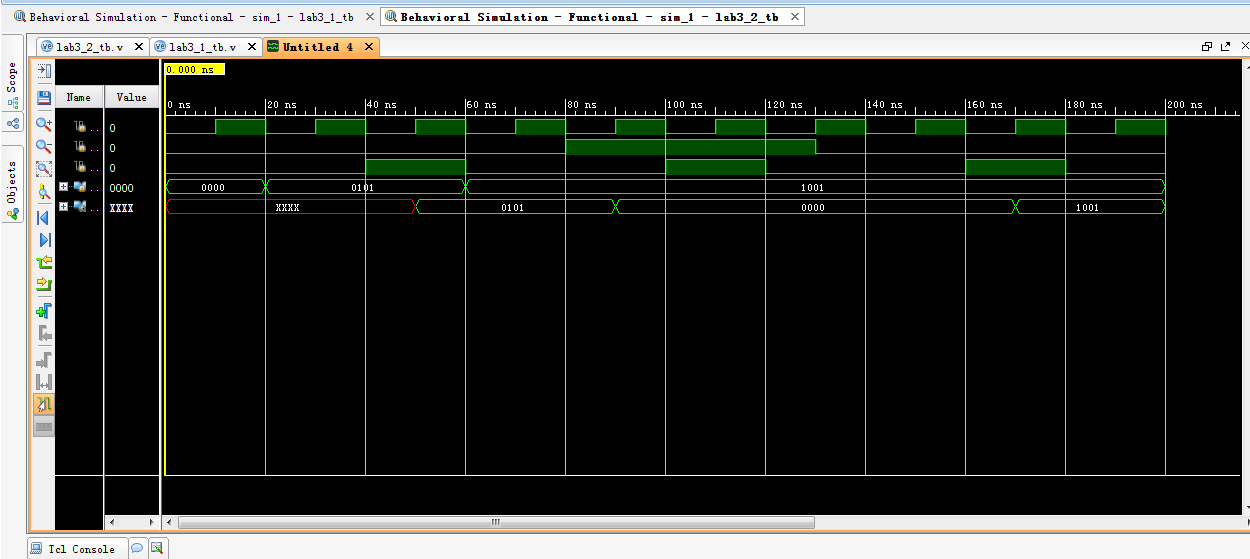
#40 in\_D=4'b1001;

#140 $stop;

end

endmodule

(3) 仿真波形图：



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 = 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(input clk,input ShiftEn,input ShiftIn,output [3:0] ParallelOut,output reg ShiftOut);

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

always @(posedge clk)

if (ShiftEn) begin

ShiftOut = shift\_reg[3];

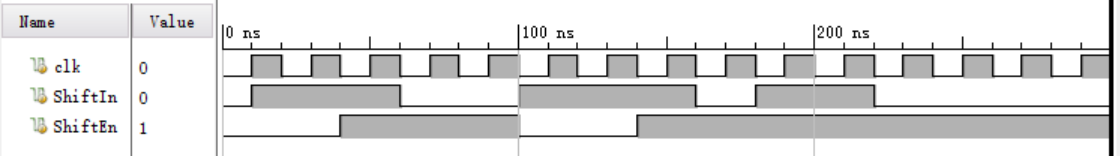
shift\_reg <= {shift\_reg[2:0], ShiftIn};

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 in\_clk,in\_ShiftEn,in\_ShiftIn;

wire [3:0] ParallelOut\_out;

wire ShiftOut\_out;

Serial\_in\_Parallel\_out\_enable test(.clk(in\_clk),.ShiftEn(in\_ShiftEn),.ShiftIn(in\_ShiftIn),.ShiftOut(ShiftOut\_out),.ParallelOut(ParallelOut\_out));

initial begin

in\_clk=1'b0;

end

always

#10 in\_clk= !in\_clk;

initial begin

in\_ShiftIn=1'b0;

#10 in\_ShiftIn= !in\_ShiftIn;

#50 in\_ShiftIn= !in\_ShiftIn;

#40 in\_ShiftIn= !in\_ShiftIn;

#60 in\_ShiftIn= !in\_ShiftIn;

#20 in\_ShiftIn= !in\_ShiftIn;

#40 in\_ShiftIn= !in\_ShiftIn;

#80 $stop;

end

initial begin

in\_ShiftEn=1'b0;

#40 in\_ShiftEn= !in\_ShiftEn;

#60 in\_ShiftEn= !in\_ShiftEn;

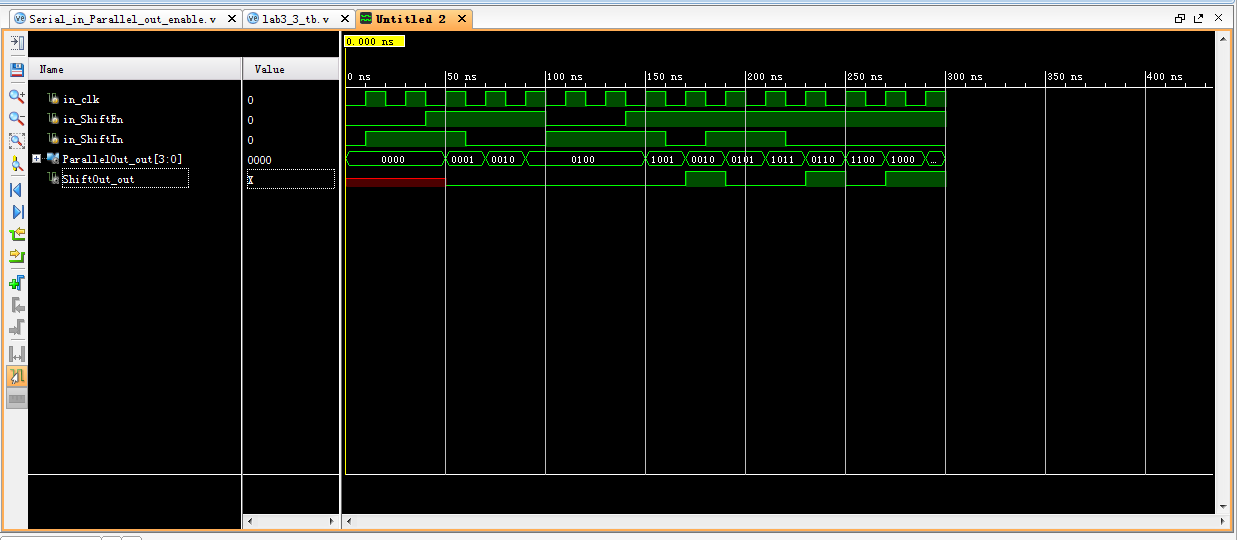
#40 in\_ShiftEn= !in\_ShiftEn;

#160 $stop;

end

endmodule

(3) 仿真波形图：



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) begin // 同步复位

Q <= 4'b0;

end

else begin

if (en) begin

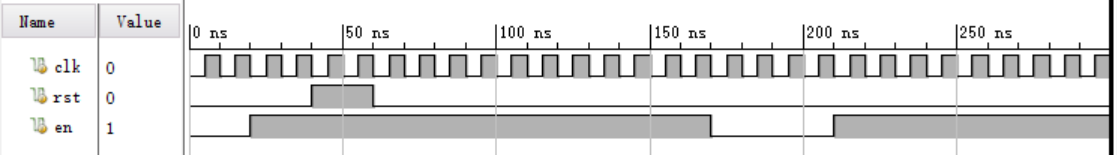
Q<=Q+1;

end

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\_4\_tb();

reg clk;

reg en,rst;

wire [3:0] Q;

initial begin

clk = 1'b0;

end

always

#5 clk = !clk;

initial begin

rst = 1'b0;

#40 rst = !rst;

#20 rst = !rst;

#240 $stop;

end

initial begin

en = 1'b0;

#20 en = !en;

#150 en = !en;

#40 en = !en;

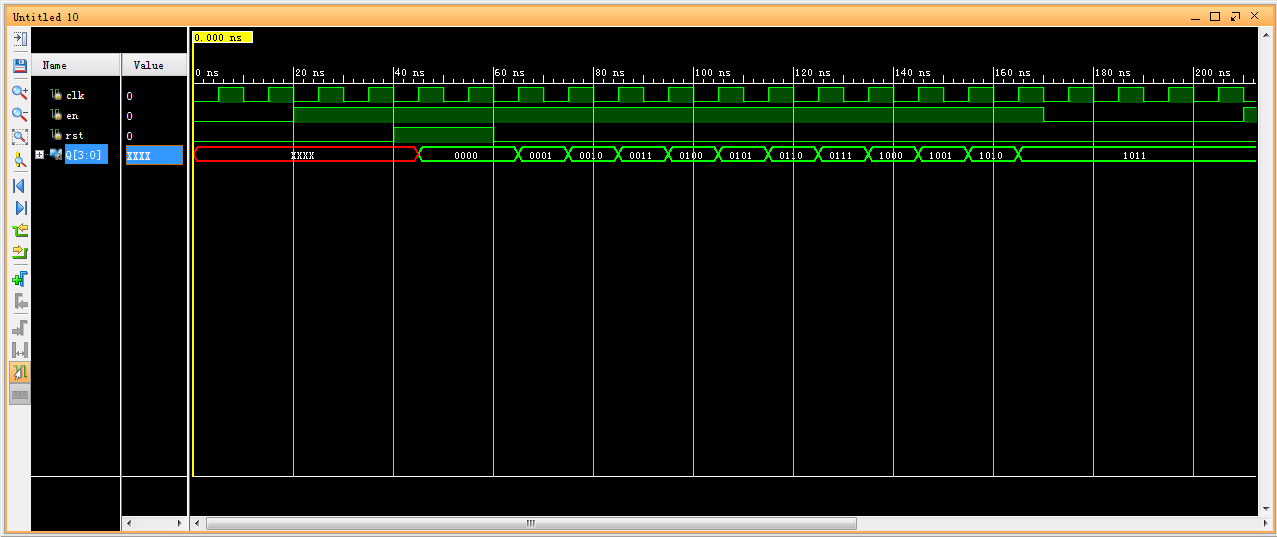
#90 $stop;

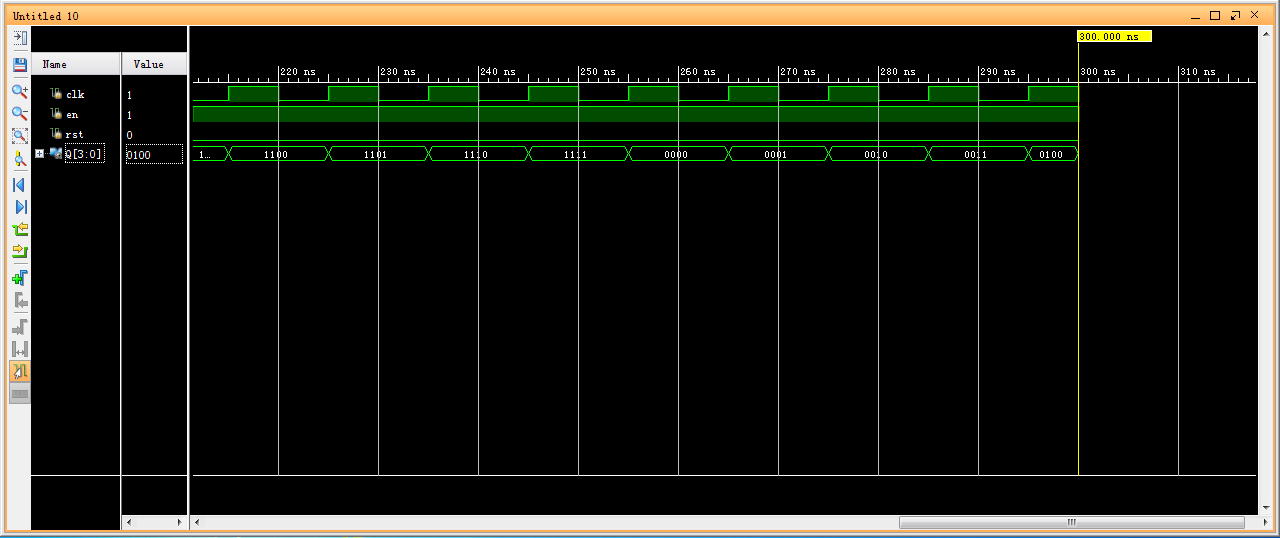
end

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

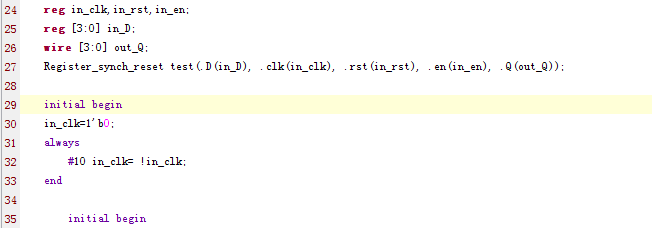
Endmodule

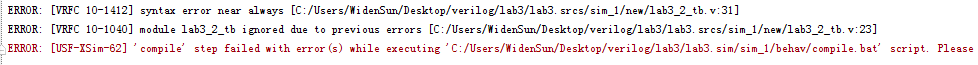
(3) 仿真波形图：





## 遇到的问题及解决方法





这次试验在编写第一个原件电路是比较顺利，但是在编写测试文件的时候，有一个报错我看了好久也没明白怎么回事，一直说我的always语句有问题，开始我以为是漏了end或是begin，最后才发现，原来always语句不能在嵌套中使用，果然，修改过后，报错消失了。

编写第二个文件和测试比较顺利。但是在第三个任务的时候我卡住了，怎么都没想明白这个任务是让我怎么做，后来我反复观看了老师给我的教学视频后，才渐渐理解了如何做到串行输入，并行输出。

这次实验在编写测试文件的时候我有个疑问，在always语句中，如果我只让程序循环10次，应该怎样让它停下来呢，最后我只想到了再多申请一个寄存器变量用来作为计数器，判断语句就写成计数器是否清0就可以了，当然个人认为还是比较浪费空间的，希望能够像汇编语言一样专门分配寄存器用来做循环的计数器就好了。

## 实验心得、意见和建议

这次实验课，总的来说自己已经上路了，可以独立的写出自己需要的电路并编写测试文件。当然这次我遇到了应该如何在自己希望的地方终止always，目前的解决办法是换成循环再加个计数器判定，此外感觉对于仿真波形图，vivado只能一级一级放大缩小，没有像word那样可以按照百分比放缩，我经常不能让图像放大到自己最满意的状态，希望程序能够改进的更加人性化。

实验4 数据通路和有限状态机设计

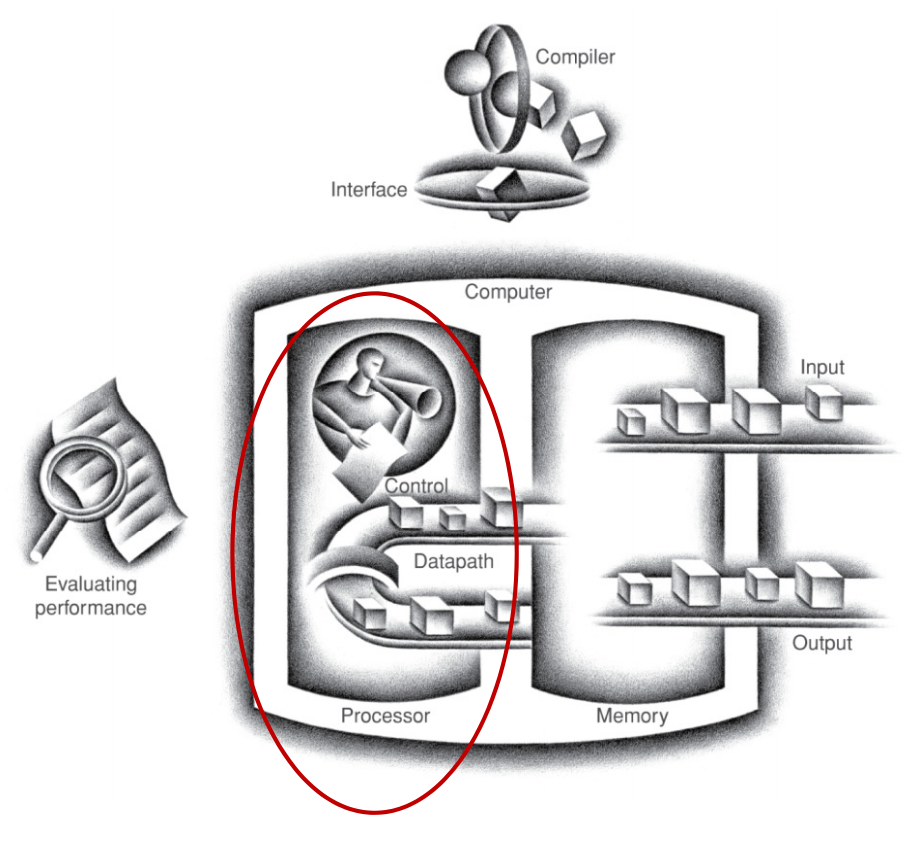
* 任务描述
* 相关知识
* 实验内容
* 遇到问题和解决方法
* 实验心得、意见和建议

## 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）。



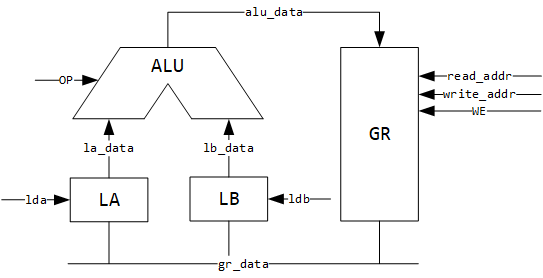
**图1处理器由数据通路和控制器构成**

## 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);**

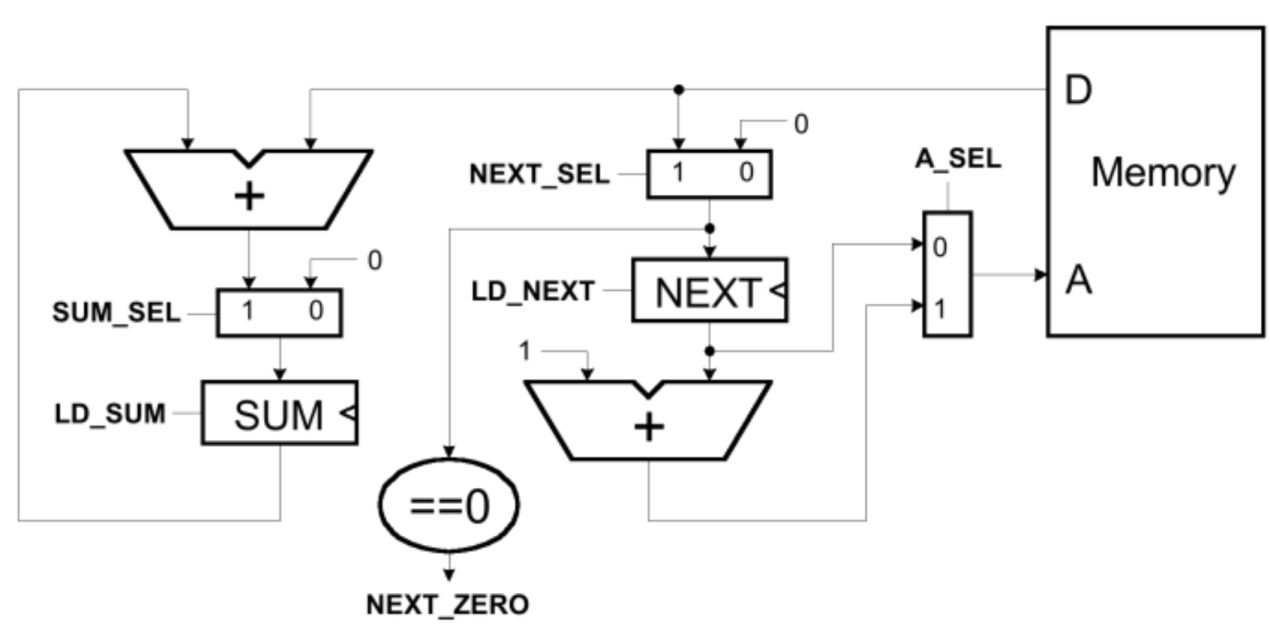
**endmodule**

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）

存储器模块：

module memory(address, data);

parameter WIDTH = 32;

input [WIDTH-1:0] address;

output [WIDTH-1:0] data;

reg [WIDTH-1:0] ram[4\*WIDTH-1:0];

initial begin

$readmemh("C:/Users/WidenSun/Desktop/verilog/lab4/lab4.srcs/sources\_1/new/ram\_init.txt", ram);

end

assign data =ram[address];

endmodule

加法器模块：

module adder(a\_in, b\_in, c\_out);

parameter WIDTH = 32;

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

output [WIDTH-1:0] c\_out;

assign c\_out = a\_in +b\_in;

endmodule

寄存器模块：

module register(clk, rst, load, d, q);

parameter WIDTH = 32;

input clk,rst,load;

input [WIDTH-1:0] d;

output reg [WIDTH-1:0] q;

initial begin

q<=0;

end

always @(posedge clk) begin

if(rst==1)

q<=0;

else if (load==1)

q<=d;

end

endmodule

二选一模块：

module select\_2\_1(a, b, c\_out, sel);

parameter WIDTH = 32;

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

input sel;

output [WIDTH-1:0] c\_out;

assign c\_out = (sel)? a: b;

endmodule

比较器模块：

module comparator(a,b,is\_great,is\_less,is\_equal );

parameter WIDTH = 32;

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

output reg is\_great,is\_less,is\_equal;

initial begin

is\_great<=0;

is\_less<=0;

is\_equal<=0;

end

always @(a,b) begin

if(a==b) begin

is\_equal=1;

is\_great=0;

is\_less=0;

end

if(a>b) begin

is\_equal=0;

is\_great=1;

is\_less=0;

end

if(a<b) begin

is\_equal=0;

is\_great=0;

is\_less=1;

end

end

endmodule

1. 数据通路的代码：

`timescale 1ns / 1ps

module datapath\_top(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 [31:0] ADD\_OUT1,ADD\_OUT2;

wire [31:0] REG\_OUT1,REG\_OUT2;

wire [31:0] SELECT\_OUT1,SELECT\_OUT2,SELECT\_OUT3;

wire [31:0] MEMORY\_DATA;

assign sum\_out=REG\_OUT1;

memory RAM1(.address(SELECT\_OUT3),.data(MEMORY\_DATA));

adder A1(.a\_in(REG\_OUT1),.b\_in(MEMORY\_DATA),.c\_out(ADD\_OUT1));

adder A2(.a\_in(1), .b\_in(REG\_OUT2), .c\_out(ADD\_OUT2));

register REG1(.clk(clk), .rst(rst), .load(LD\_SUM), .d(SELECT\_OUT1), .q(REG\_OUT1));

register REG2(.clk(clk), .rst(rst), .load(LD\_NEXT), .d(SELECT\_OUT2), .q(REG\_OUT2));

select\_2\_to\_1 SELECT1(.a(ADD\_OUT1), .b(0), .c\_out(SELECT\_OUT1), .sel(SUM\_SEL));

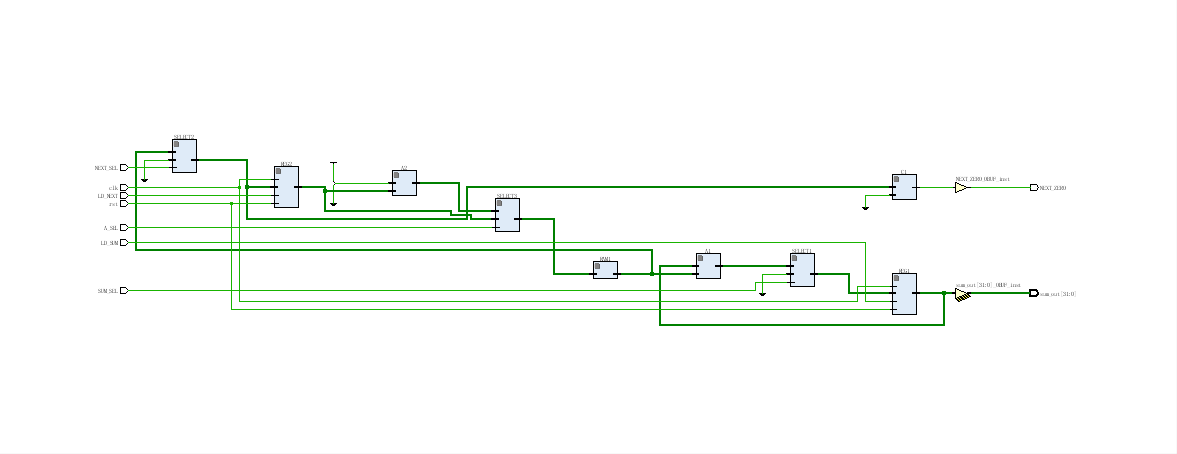
select\_2\_to\_1 SELECT2(.a(MEMORY\_DATA), .b(0), .c\_out(SELECT\_OUT2), .sel(NEXT\_SEL));

select\_2\_to\_1 SELECT3(.a(ADD\_OUT2), .b(REG\_OUT2), .c\_out(SELECT\_OUT3), .sel(A\_SEL));

comparator C1(.a(SELECT\_OUT2),.b(0),.is\_equal(NEXT\_ZERO));

endmodule

1. 数据通路的电路原理图（生成Schematic）：



**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)。

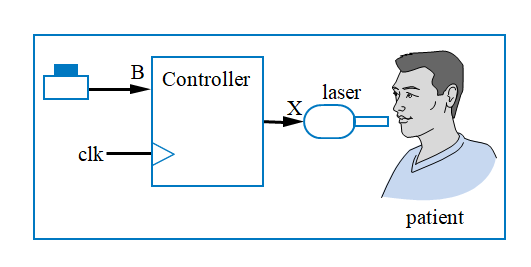


图4 激光计时器

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。

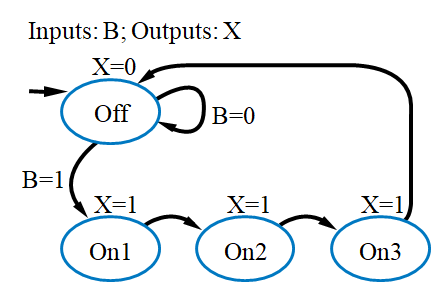


图5 该激光计时器的有限状态机

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);**

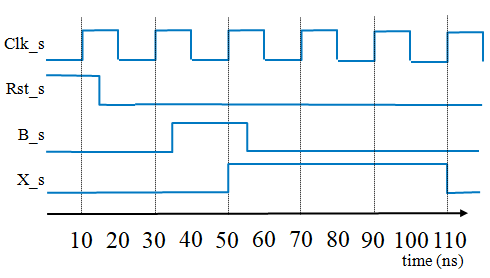
**#5 if (X\_s != 0)** //验证状态Off

**$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：

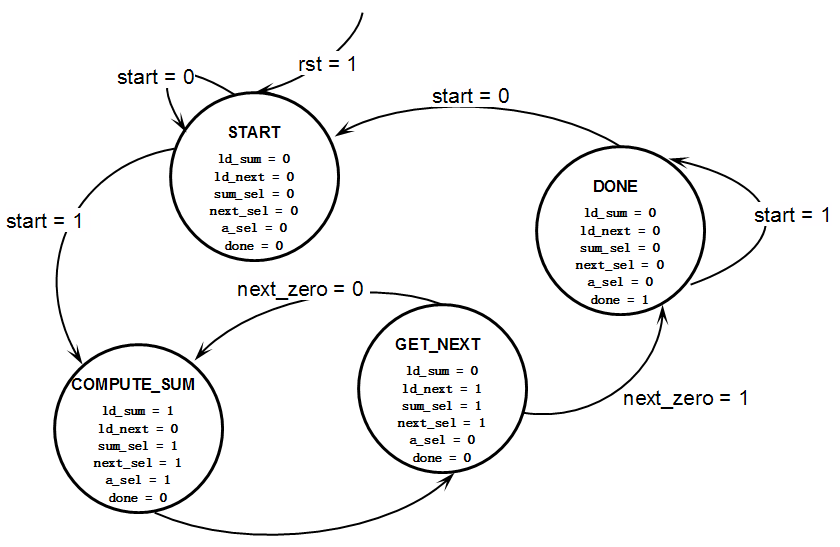
* 设计的有限状态机（FSM）能够正确综合；
* 编写有限状态机的仿真程序，完成有限状态机（FSM）的功能仿真，有限状态机功能仿真正确。

【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;

input rst, start;

input next\_zero;

output ld\_sum, ld\_next;

output sum\_sel, next\_sel;

output a\_sel;

output done;

parameter START = 4'b1000;

parameter COMPUTE\_SUM = 4'b0100;

parameter GET\_NEXT = 4'b0010;

parameter DONE = 4'b0001;

reg [3:0] state;

reg [5:0] output\_value;

assign {ld\_sum, ld\_next, sum\_sel, next\_sel, a\_sel, done} = output\_value;

initial begin

state <= START;

output\_value <= 6'b000000;

end

always @(posedge clk) begin

if (rst) begin

state <= START;

output\_value <= 6'b000000;

end

else begin

case (state)

START:begin

if (start == 0) begin

state <= START;

output\_value <= 6'b000000;

end

else if (start == 1) begin

state <= COMPUTE\_SUM;

output\_value <= 6'b101110;

end

end

COMPUTE\_SUM:begin

state <= GET\_NEXT;

output\_value <= 6'b011100;

end

GET\_NEXT:begin

if (next\_zero == 0) begin

state <= COMPUTE\_SUM;

output\_value <= 6'b101110;

end

else if (next\_zero == 1) begin

state <= DONE;

output\_value <= 6'b000001;

end

end

DONE:begin

if (start == 0) begin

state <= START;

output\_value <= 6'b000000;

end

else if (start == 1) begin

state <= DONE;

output\_value <= 6'b000001;

end

end

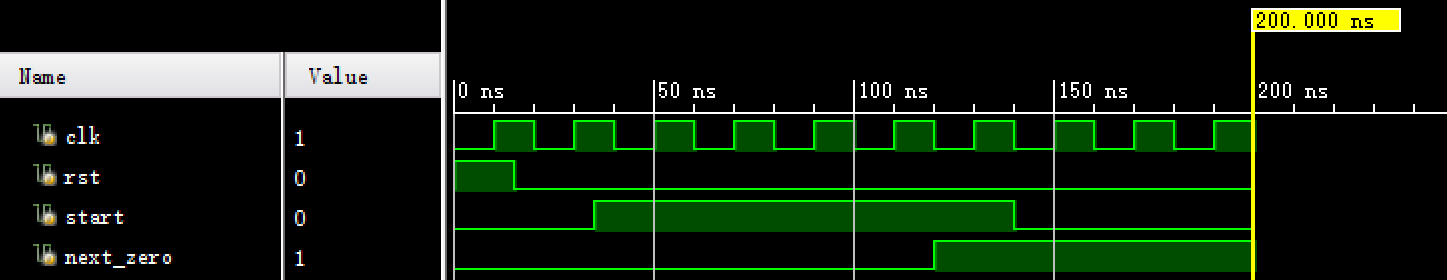
endcase

end

end

endmodule

1. 设计testbench进行仿真测试，输入信号波形如下图：



module FSM\_tb();

reg clk;

reg rst, start;

reg next\_zero;

wire ld\_sum, ld\_next, sum\_sel, next\_sel, a\_sel, done;

FSM FSM1(.clk(clk),

.rst(rst),

.start(start),

.next\_zero(next\_zero),

.ld\_sum(ld\_sum),

.ld\_next(ld\_next),

.sum\_sel(sum\_sel),

.next\_sel(next\_sel),

.a\_sel(a\_sel),

.done(done));

initial begin

clk <= 0;

rst <= 1;

start <= 0;

next\_zero =0;

#15 rst<=0;

#20 start<=1;

#95 next\_zero =1;

#20 start<=0;

#50 $stop;

end

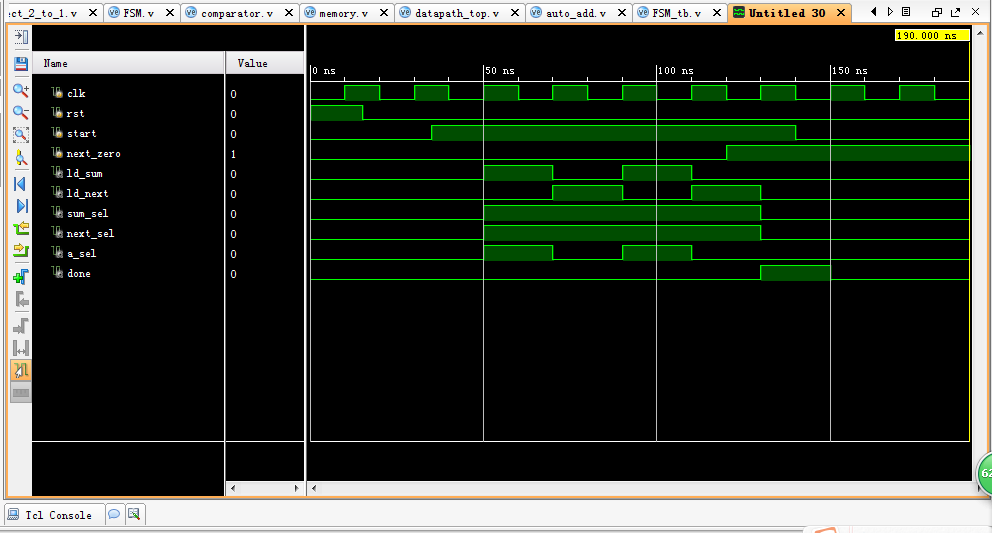
always begin

#10 clk <= ~clk;

end

endmodule

1. 仿真结果图：

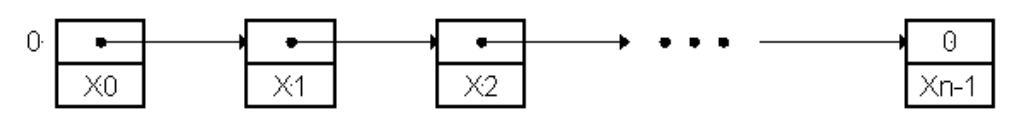


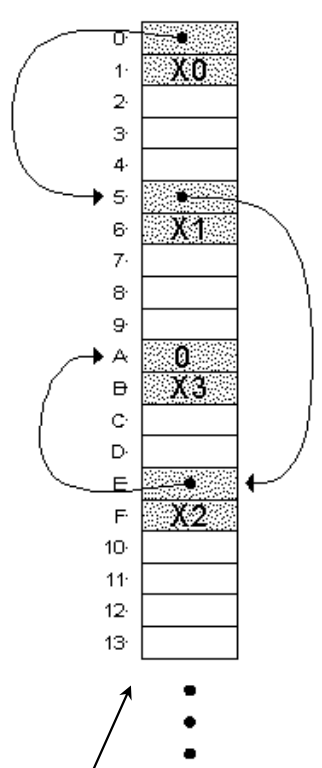
**3. 自动运算电路的设计（步骤3）**

【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。

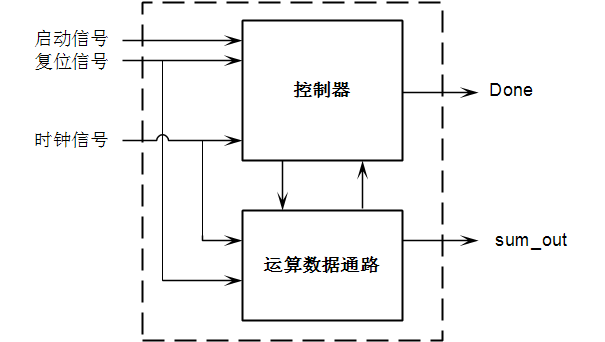




**图7 数据链表及其在存储器中的存放格式**

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);

input clk,rst,start;

output done;

output [31:0] sum\_out;

wire next\_zero;

wire ld\_sum,ld\_next;

wire sum\_sel, next\_sel;

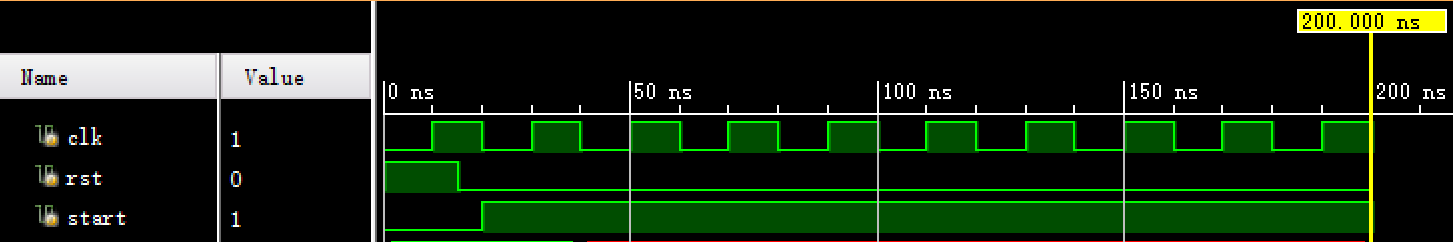
wire a\_sel;

datapath\_top D1 (.clk(clk), .rst(rst), .SUM\_SEL(sum\_sel), .NEXT\_SEL(next\_sel), .A\_SEL(a\_sel), .LD\_SUM(ld\_sum), .LD\_NEXT(ld\_next), .NEXT\_ZERO(next\_zero), .sum\_out(sum\_out));

FSM F1 (.clk(clk),.rst(rst),.start(start),.next\_zero(next\_zero), .ld\_sum(ld\_sum), .ld\_next(ld\_next), .sum\_sel(sum\_sel), .next\_sel(next\_sel), .a\_sel(a\_sel), .done(done));

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 A1 (clk,rst,start,done,sum\_out);

initial begin

clk<=0;

rst<=1;

start<=0;

#15 rst<=0;

#5 start<=1;

#980 $stop;

end

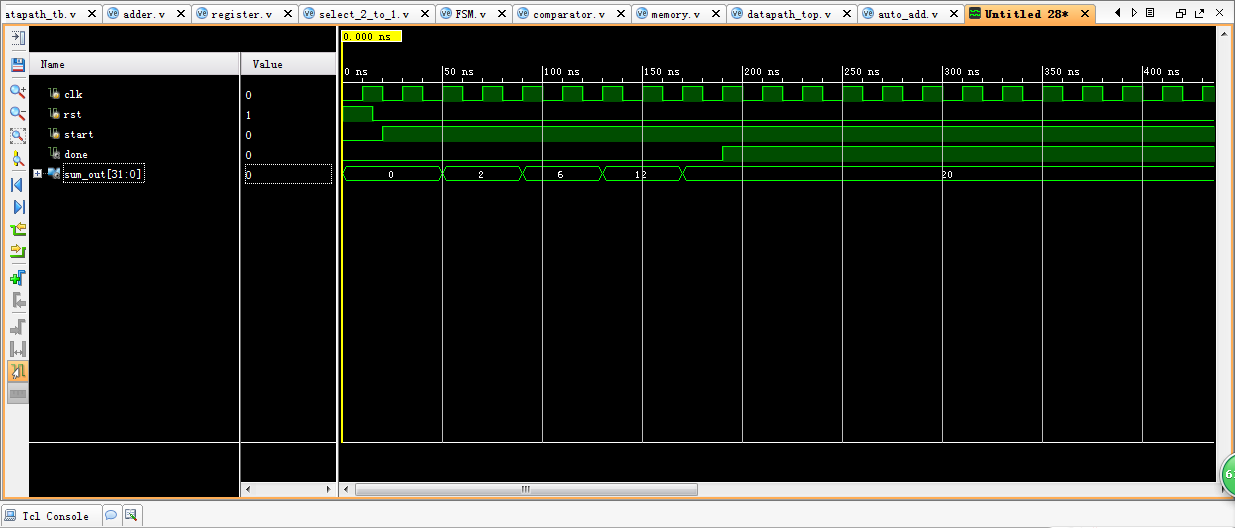
always begin

#10 clk<=~clk;

end

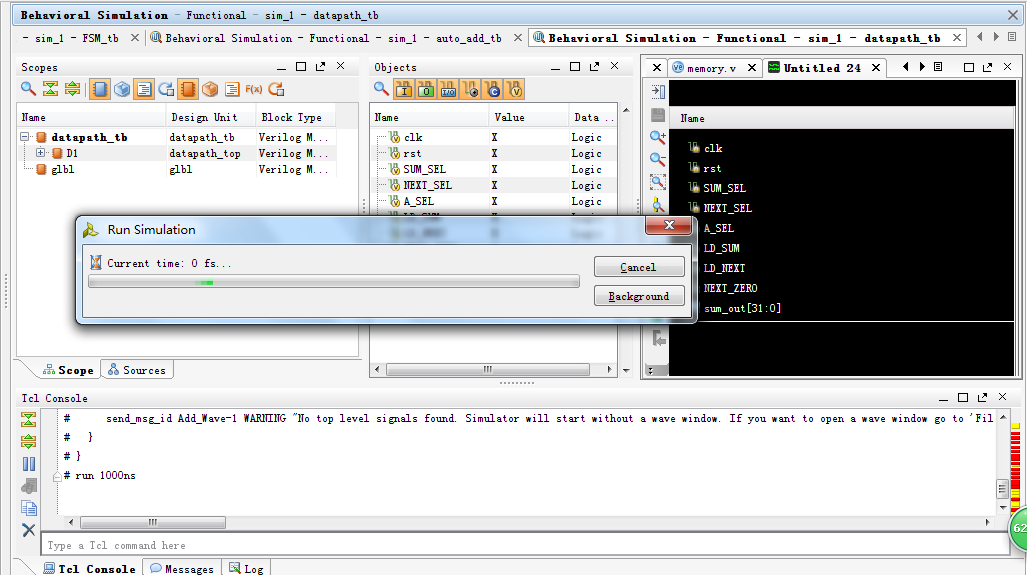
endmodule

1. 仿真结果图：



## 遇到问题和解决方法

这次实验的难度一下子上来了，我刚开始看第一个数据通路的时候无从下手，后来发现一个一个模块写起来其实还比较容易，首先我遇到的难点就是关于那个readmeh函数的使用，刚开始我是把txt文件放在source文件中，后来报错打不开。我于是又在网上查了，发现应该把文件放在sim文件夹里面。我先自己写了testbench测试了一下，发现我的程序半天不出来仿真结果：



后来我在群里面询问了老师，知道这个报错原来还是我的程序进入了死循环。最后我发现是我的比较器函数出了问题，导致模块死循环了，后来我直接拿if语句改写了，仿真结果成功输出。在设计memory模块中，我也是了解了为什么我32位的读取，却要准备128位的地址，不然也会报错。

FSM模块的编写过程，刚开始我也是十分的苦恼，不知道每个状态那么多输出怎么解决，后来才想到了直接把每个状态的信息存起来，到了那个状态直接把所有的变量重新赋值就行了，帮助我优化了代码，避免了我像以前一样使用那么多语句。

最后的自动计算电路，我就是简单的把之前编写好的两个模块的接口连接好，就可以编写仿真程序。仿真结果应该很快就出来了，但是又给我报错，我文件太多字符？

这搞得我有点迷茫，难道又是那个txt文件出了错？最后我干脆直接把txt文件换成了直接路径打开了，但是又继续报错，说我文件打不开。我去xlinx论坛上逛了好久，发现再输入绝对路径的时候要使用“/”，而不是“\”，最后成功的解决了这个报错。在老师的帮助下，我把输出成功的显示成了10进制，得到了仿真效果图。

## 实验心得、意见和建议

这次试验我花了很长的时间观看老师给的视频知道和ppt的样例，首先在第一次课上我先是把老师给的例子整合到一起跑了一遍，然后根据老师给的要求图在纸上列出了需要编写的元器件，真正编写起来的时候便是十分容易了。

这次遇到的报错也是有几个我之前没有见过的，那个死循环我一直以为系统能够像c语言一样有中断机制，超过一定时间就会自动退出，可是事实却是程序直接卡死了，希望vivado也能够添加这种中断机制，方便我及时了解程序错误。还有一个问题就是错误的报错定位搞得我很困惑，每次看报错的地方，我反复想了好久都觉得没问题，但是就是调不通，最后只能选择重新写一遍，感觉很是不舒服。这次也是我第一次尝试文件读取，除了readmeh函数，还有很多文件打开函数，类似于c语言的fopen函数这一类的，并且这类函数也会配备文件指针方便我进行文件访问。

每次做实验，Vivado功能和函数之多让我佩服，但是感觉和真正编程语言的成熟的开发环境还是有很多差距，希望能够增加一定的容错率以及减少一些莫名其妙的bug。