

实验报告

|  |  |
| --- | --- |
| **学院** | 计算机科学与技术学院 |
| **班级** | CS1802 |
| **老师** | 班鹏新 |
| **姓名** | 李响 |
| **学号** | U201814531 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **2020** | **年** | **5** | **月** | **8** | **日** |

**目录**

**[实验2 简单组合电路设计 1](#_Toc19637)**

[任务描述 1](#_Toc6077)

[相关知识 1](#_Toc3274)

[实验步骤 2](#_Toc23156)

[编程要求 4](#_Toc7670)

[测试说明 4](#_Toc29985)

[源码 6](#_Toc21760)

[Testbench代码 8](#_Toc26098)

[控制台输出和波形图 8](#_Toc16224)

[遇到问题和解决方法 9](#_Toc23670)

[实验心得、意见和建议 9](#_Toc26520)

**[实验3 简单时序电路设计 10](#_Toc19398)**

[任务描述 10](#_Toc30321)

[相关知识 10](#_Toc6929)

[实验内容 11](#_Toc31637)

[遇到的问题及解决方法 18](#_Toc23930)

[实验心得、意见和建议 18](#_Toc9100)

**[实验4 数据通路和有限状态机设计 19](#_Toc6595)**

[任务描述 19](#_Toc8905)

[相关知识 19](#_Toc13800)

[实验内容 20](#_Toc24169)

[遇到的问题及解决方法 38](#_Toc14841)

[实验心得、意见和建议 38](#_Toc30721)

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

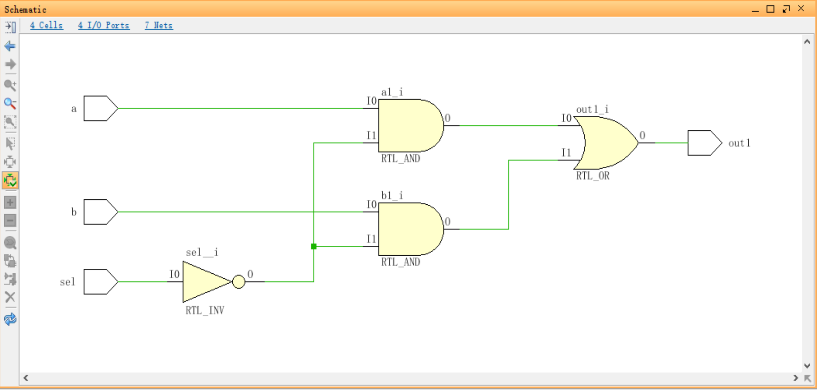


图2.2 2选1选择器Schematic

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。

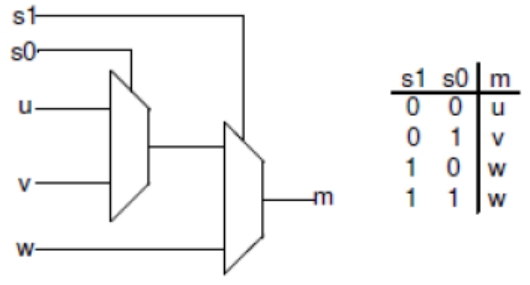


图2-3 3选1多路选择器电路和真值表

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。

## 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

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]

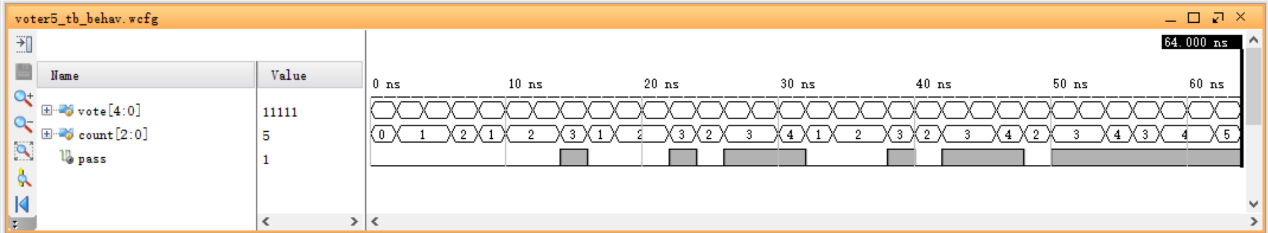
56000::[11100] [3] [1]

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

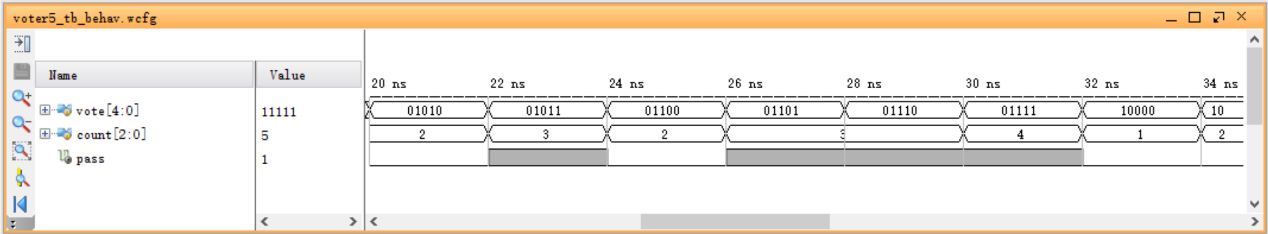
60000::[11110] [4] [1]

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. 任务1：2选1选择器—结构描述

module lab2\_1(a,b,sel,out1);

input a,b,sel;

output out1;

not (sel\_, sel);

and (a1, a, sel\_);

and (b1, b, sel);

or (out1, a1, b1);

endmodule

2. 任务2：2选1选择器—数据流描述

module lab2\_2(a,b,sel,out1);

input a,b,sel;

output out1;

assign out1=(sel & b) | (~sel & a);

endmodule

3. 任务3：2选1选择器—行为描述

module lab2\_3(a,b,sel,out1);

input a,b,sel;

output reg out1;

always @(\*)

begin

case (sel)

1'b0 : out1 = a;

1'b1 : out1 = b;

endcase

end

endmodule

4. 任务4：用2选1多路选择器构造3选1多路选择器。

module lab2\_4(u,v,w,s0,s1,m);

input u,v,w,s0,s1;

output m;

lab2\_3(u,v,s0,out1);

lab2\_3(out1,w,s1,m);

endmodule

5. 任务5：设计一个3-8译码器模块

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

## Testbench代码

任务5：3-8译码器模块Testbench代码

`timescale 1ns / 1ps

module decoder\_38\_tb();

wire [7:0] F;

reg [2:0] CBA;

decoder\_38 TEST(.F(F), .CBA(CBA));

initial begin

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

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

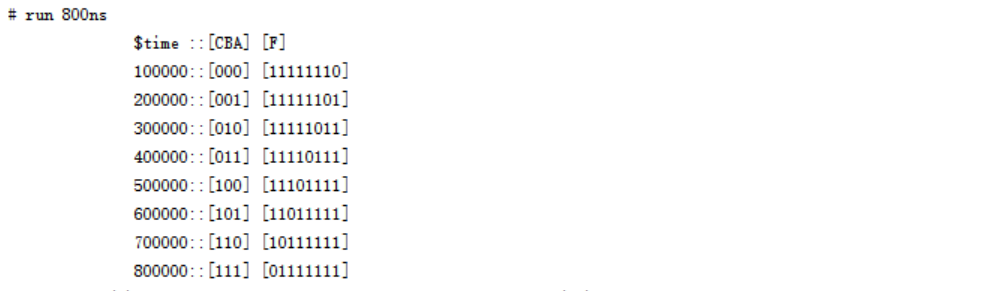
#100 $display ("%t::", $time, "[%b]\t[%b]",CBA,F);

end

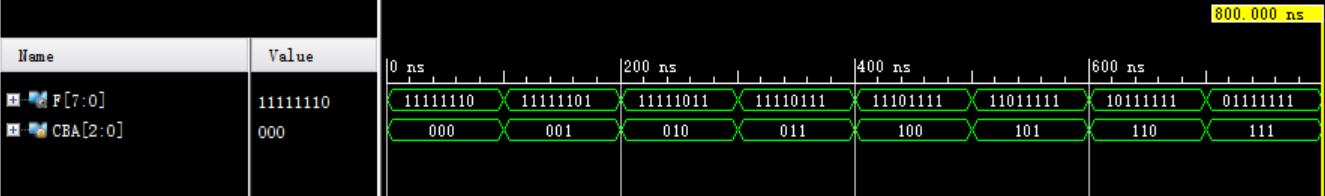
endmodule

## 控制台输出和波形图

任务5：3-8译码器模块的控制台输出与波形图如下图2.4所示，图a为控制台输出结果的截图，与实验要求进行比对，成功实现了译码功能；图b为波形图，与控制台输出相呼应，成功实现了译码功能。



（a）控制台输出结果显示图



（b）波形图

图2.4 3-8译码器模块的控制台输出与波形图

## 遇到问题和解决方法

由于Verilog语言的语法结构和C语言类似，再加上本次实验比较基础，本次实验碰到的问题不是特别多，主要集中在任务5设计3-8译码器模块并编写Testbench代码中，实验中遇到的主要问题和解决方法如下：

（1）由于初次接触Verilog语言，所以在编写的时候容易忘记begin和end的书写，导致程序在逻辑上出现问题，经过几次错误后，逐渐克服了这个问题。

（2）对于Testbench代码书写的不习惯，不小心把input写进了Testbench代码了，导致在进行仿真的时候总是出现问题，通过老师的提醒，最终修改了这个问题。

（3）对于initial语句以及Verilog语言模块化的、非流程性的语言设计风格不太适应，导致对程序的编写总是不是特别熟练，通过反复阅读资料和动手设计程序，最终克服了问题。

## 实验心得、意见和建议

本次实验是本课程第一次自己动手编写程序的实验，经过上几次实验课对Verilog语言的学习何对vivado程序设计流程的熟悉，本次实验得以比较顺利地进行。

由于Verilog语言的语法结构和C语言类似，再加上本次实验比较基础，本次实验上手较快，但是由于对Verilog语言非流程性的语言设计风格不是特别适应，导致我本次实验中产生了不少问题。通过本次实验，我基本掌握了基础的Verilog程序设计方式和语法（主要掌握了寄存器reg、网线wire以及always等语句的使用等，很多其他的类似循环和选择结构还有待进一步的学习），了解到了HDL类的语言的设计风格。

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事件控制。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。

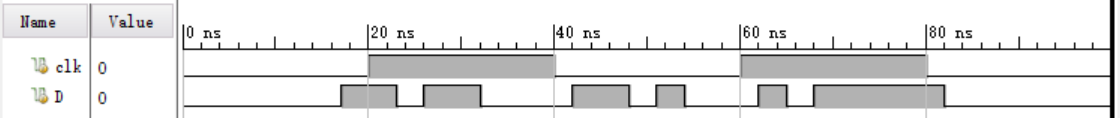
(1) 时钟下降沿触发的D触发器建模：

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

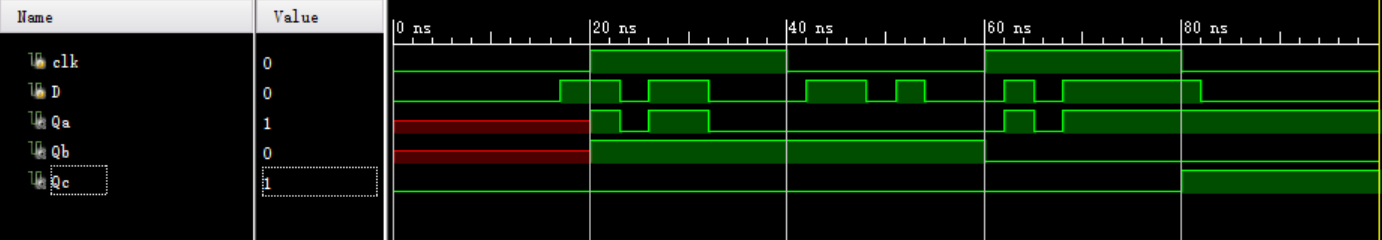
D\_latch myDlatch(clk, D, Qa);

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

always @(posedge clk)

if (rst) begin

Q <= 4'b0;

end

else begin

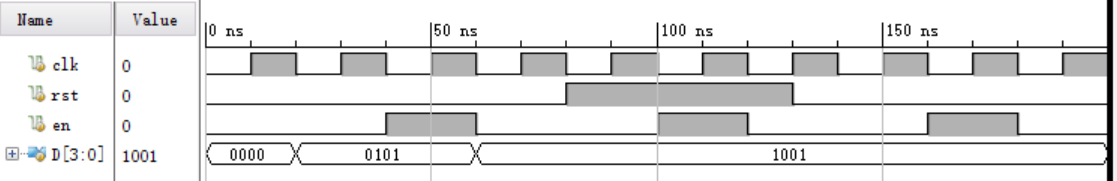
if (en) Q <= D;

else 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 clk, rst, en;

reg [3:0] D;

wire [3:0] Q;

initial begin

clk = 1'b0;

rst = 1'b0;

en = 1'b0;

D = 4'b0;

#200 $stop;

end

always

#10 clk = !clk;

initial begin

#80 rst = !rst;

#50 rst = !rst;

end

always begin

#40 en = !en;

#20 en = !en;

end

initial begin

#20 D = 4'b0101;

#40 D = 4'b0101;

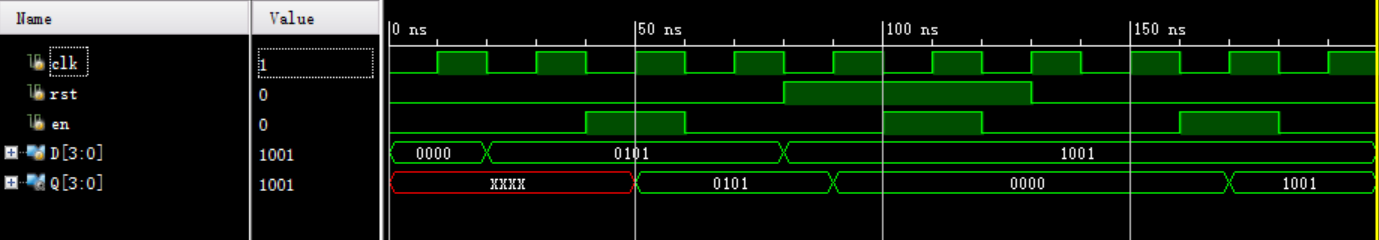
#20 D = 4'b1001;

end

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

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(clk, ShiftEn, ShiftIn, ParallelOut, ShiftOut);

input clk, ShiftIn, ShiftEn;

output [3:0] ParallelOut;

output ShiftOut;

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

reg shiftout = 1'b0;

always @(posedge clk)

if (ShiftEn) {shiftout, shift\_reg} <= {shift\_reg[3:0], ShiftIn};

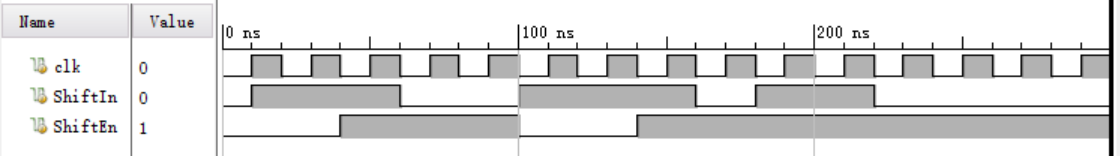
else shift\_reg <= shift\_reg;

assign ShiftOut = shiftout;

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, ShiftEn, ShiftIn;

wire [3:0]ParallelOut;

wire ShiftOut;

initial begin

clk = 1'b0;

ShiftIn = 1'b0;

ShiftEn = 1'b0;

#250 $stop;

end

always

#10 clk = !clk;

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;

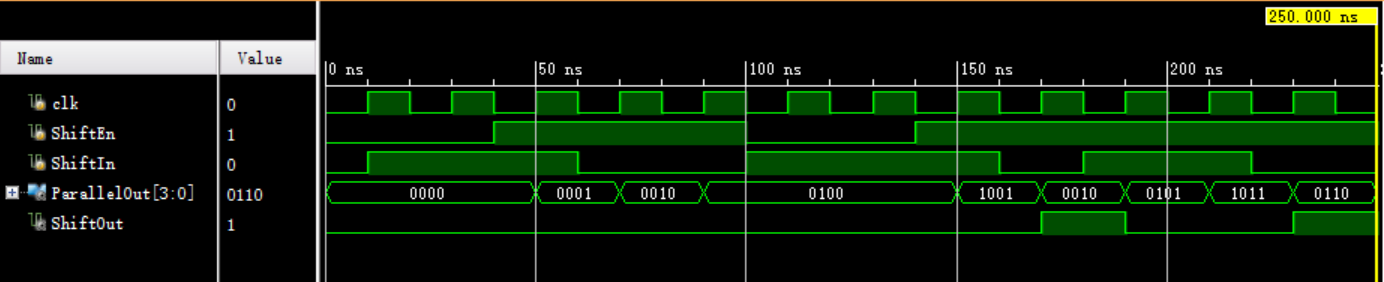
#40 ShiftEn = !ShiftEn;

end

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

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);

initial Q = 4'b0000;

always @(posedge clk)

if (rst) Q <= 4'b0000;

else begin

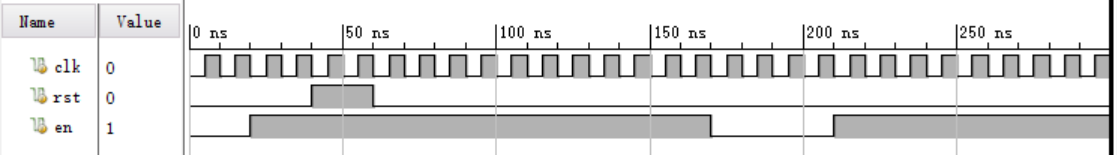
if (en) Q <= Q + 1;

else 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\_4\_tb();

reg clk, rst, en;

wire [3:0] Q;

initial begin

clk = 1'b0;

rst = 1'b0;

en = 1'b0;

#300 $stop;

end

always

#5 clk = !clk;

initial begin

#40 rst = !rst;

#20 rst = !rst;

end

initial begin

#20 en = !en;

#150 en = !en;

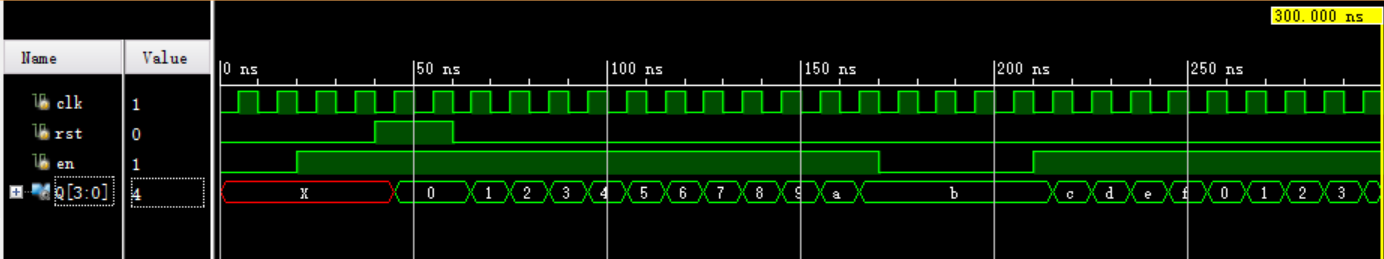
#40 en = !en;

end

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

endmodule

(3) 仿真波形图：



## 遇到的问题及解决方法

本次实验的难度较上一次有较大的提高，主要体现在4个方面，第一，出现了较多比较实用且具有实际使用价值的元器件的设计；第二，出现了上升沿时间控制和下降沿时间控制的概念，第三，也是最重要的一点，采用了大量的选择结构和循环结构；第四，Testbench代码编写复杂度提高。因此本次实验的主要问题也出在这几个方面，具体遇到的问题和解决方法如下所示：

（1）对选择结构的理解不够深入，且总是忘记加begin和end语句，导致出现逻辑上和语法上的问题，通过阅读书本和资料，逐渐克服了这一问题。

（2）对always语句中需要赋值的模块的类型理解不足，忘记需要采用reg类型才可进行赋值，导致程序出问题，通过反复调试和测验，最终找到了这个问题并解决了这个问题。

## 实验心得、意见和建议

本次实验的难度虽然较上一次有所提高，但是仍在可以接受的范围之内，通过本次实验，我基本掌握了简单的组合电路的设计和编写流程，并且顺利完成了Testbench代码的编写任务以及仿真过程。

实验4 数据通路和有限状态机设计

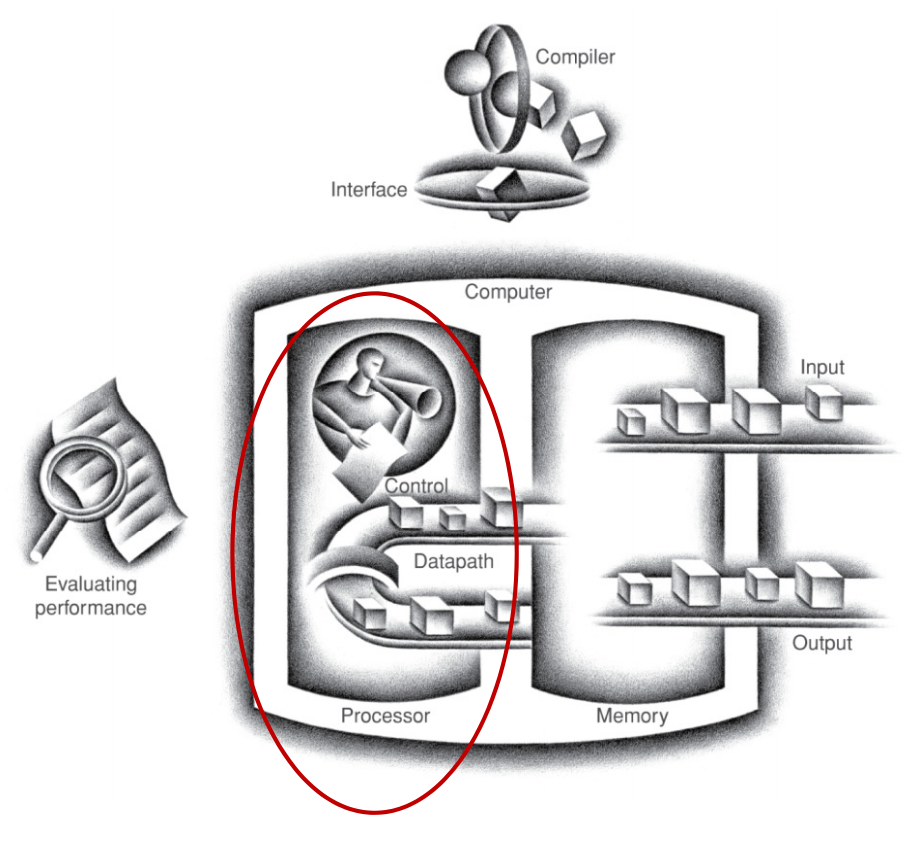
* 任务描述
* 相关知识
* 实验内容
* 遇到问题和解决方法
* 实验心得、意见和建议

## 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）。



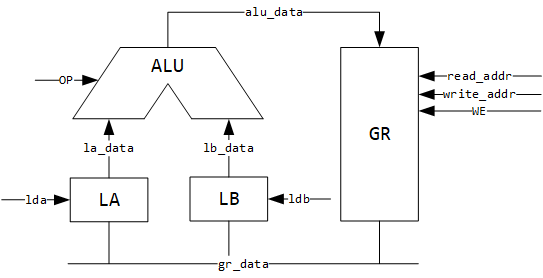
**图4.1处理器由数据通路和控制器构成**

## 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4.2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);**

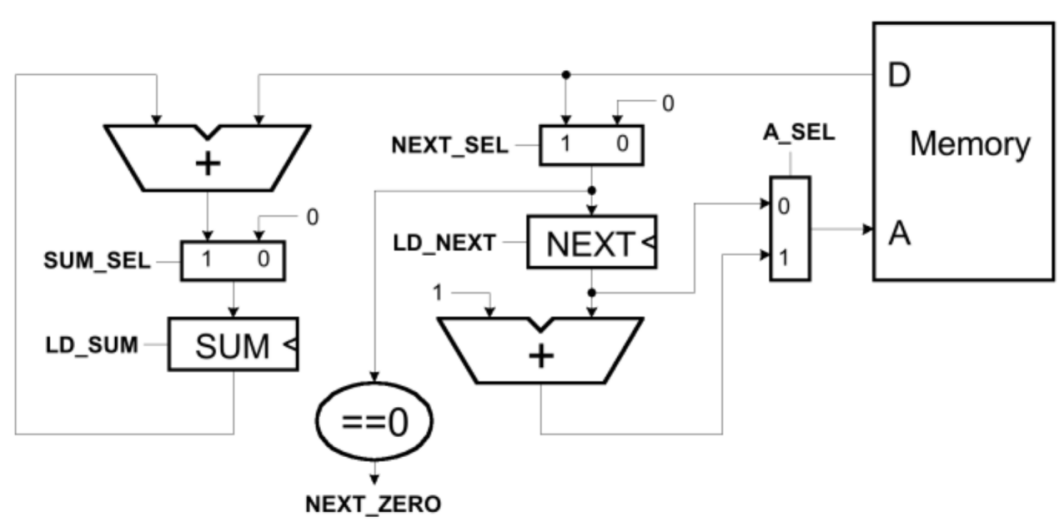
**endmodule**

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4.3 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. 各个单元模块的代码

（1）n位加法器模块

`timescale 1ns / 1ps

module n\_adder(a,b,out);

parameter WIDTH = 8;

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

output [WIDTH-1:0] out;

assign out = a + b;

endmodule

（2）n位2选1多路选择器模块

`timescale 1ns / 1ps

module n\_mux\_21(a, b, sel, out);

parameter WIDTH = 8;

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

input sel;

output [WIDTH-1:0] out;

assign out = (sel == 0) ? a : b;

endmodule

（3）n位比较器模块

`timescale 1ns / 1ps

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

parameter WIDTH = 8;

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

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

assign is\_equal = (a == b) ? 1'b1 : 1'b0;

assign is\_great = (a > b) ? 1'b1 : 1'b0;

assign is\_less = (a < b) ? 1'b1 : 1'b0;

endmodule

（4）含同步复位rst和加载load端的n位寄存器模块

`timescale 1ns / 1ps

module n\_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;

else q <= q;

end

endmodule

（5）n位存储器模块

`timescale 1ns / 1ps

module ram(addr, data, clk);

parameter DATA\_WIDTH = 8;

parameter ADDR\_WIDTH = 4;

input clk;

input [ADDR\_WIDTH-1:0] addr;

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

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

initial

$readmemh("C:/Users/new/Desktop/Verilog/lab4/ram\_init.txt", ram);

always @(negedge clk)

begin

data <= ram[addr];

end

endmodule

1. 数据通路的代码：

`timescale 1ns / 1ps

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 [31:0] next\_data,D\_data;

wire [31:0] sum1\_data,sum2\_data,mux1\_data,mux2\_data,mux3\_data;

n\_register #(32) SUM(clk, rst, LD\_SUM, mux1\_data, sum\_out);

n\_register #(32) NEXT(clk, rst, LD\_NEXT, mux2\_data, next\_data);

n\_adder #(32) sum1(sum\_out, D\_data, sum1\_data);

n\_adder #(32) sum2(next\_data, 32'b1, sum2\_data);

n\_mux\_21 #(32) mux1(32'b0, sum1\_data, SUM\_SEL, mux1\_data);

n\_mux\_21 #(32) mux2(32'b0, D\_data, NEXT\_SEL, mux2\_data);

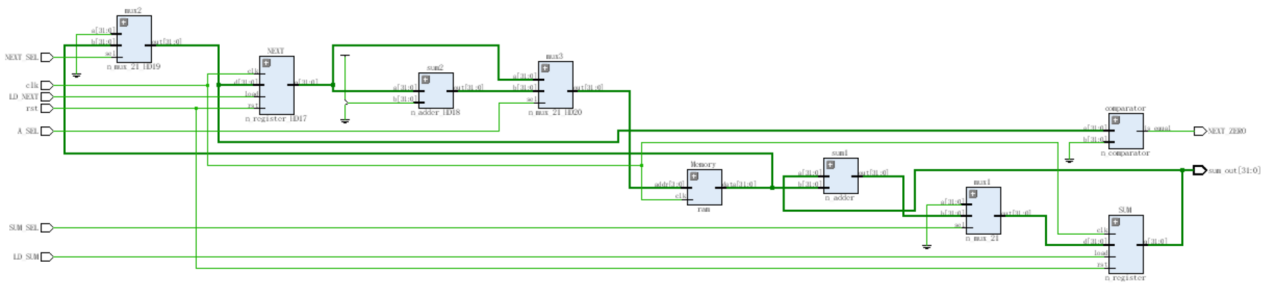
n\_mux\_21 #(32) mux3(next\_data, sum2\_data, A\_SEL, mux3\_data);

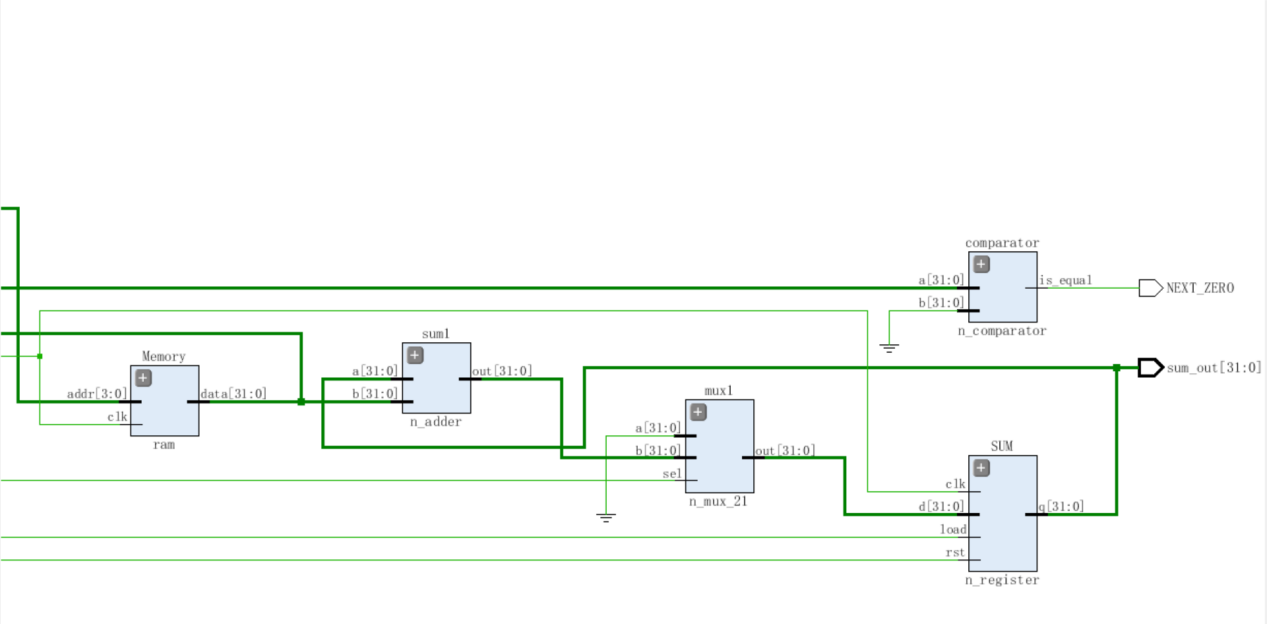
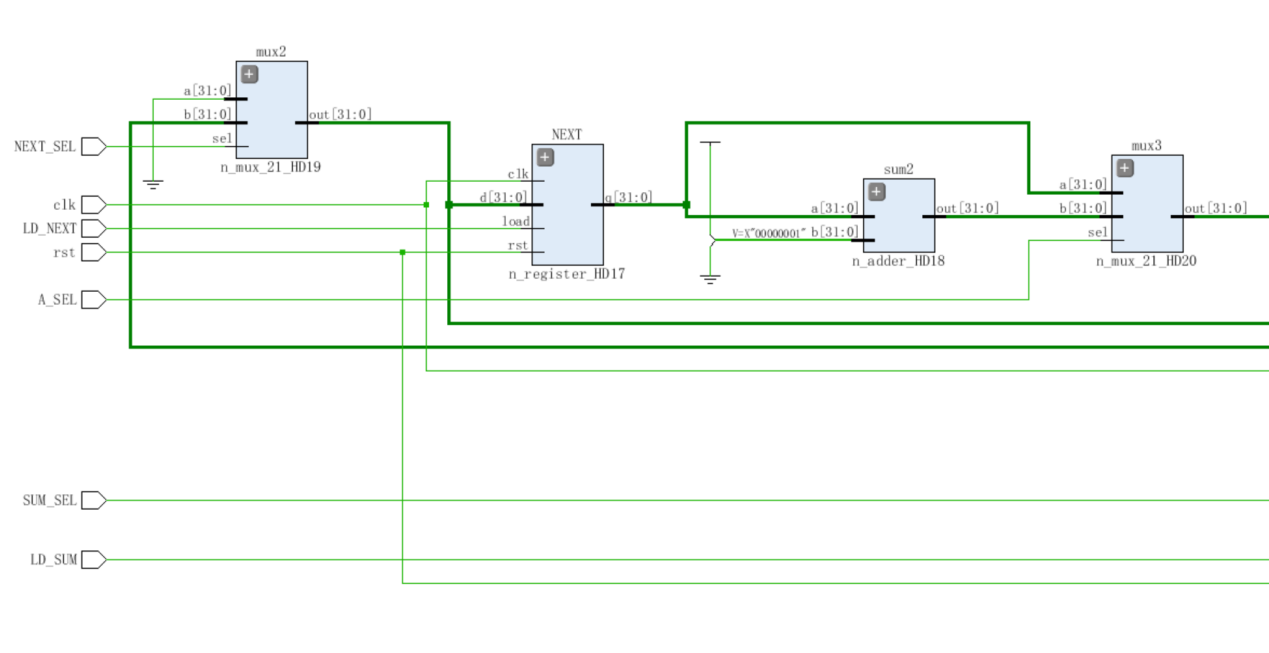
ram #(32,4) Memory(mux3\_data,D\_data,clk);

n\_comparator #(32) comparator(mux2\_data, 32'b0, NEXT\_ZERO,,);

endmodule

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)。

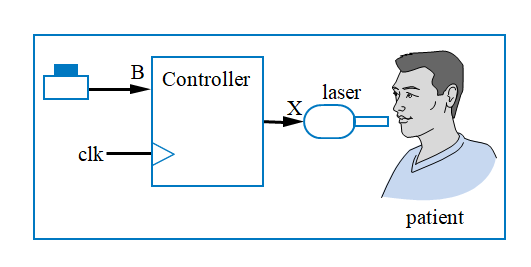


图4.4 激光计时器

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。

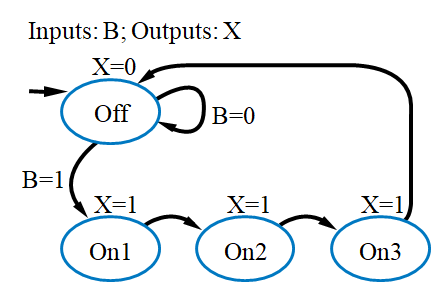


图4.5 该激光计时器的有限状态机

图4.5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);**

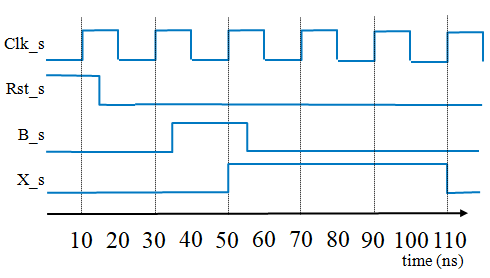
**#5 if (X\_s != 0)** //验证状态Off

**$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：

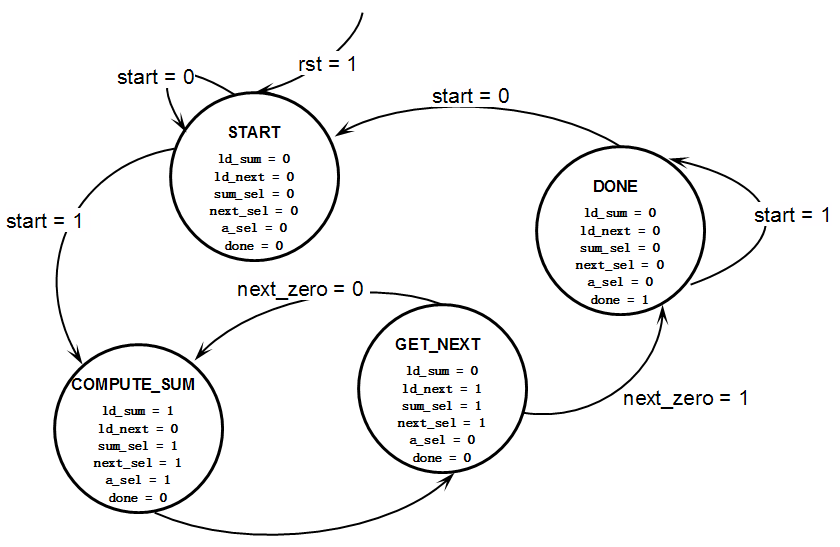
* 设计的有限状态机（FSM）能够正确综合；
* 编写有限状态机的仿真程序，完成有限状态机（FSM）的功能仿真，有限状态机功能仿真正确。

【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: LD\_SUM,LD\_NEXT,SUM\_SEL,NEXT\_SEL,A\_SEL,DONE。



**图4.6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

`timescale 1ns / 1ps

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 S\_sta = 0,S\_com = 1,S\_next = 2,S\_done = 3;

reg [1:0] State, StateNext;

always @(State, start, next\_zero) begin

case (State)

S\_sta: begin

LD\_SUM <= 0;

LD\_NEXT <= 0;

SUM\_SEL <= 0;

NEXT\_SEL <= 0;

A\_SEL <= 0;

DONE <= 0;

if (start == 1) StateNext <= S\_com;

else StateNext <= S\_sta;

end

S\_com: begin

LD\_SUM <= 1;

LD\_NEXT <= 0;

SUM\_SEL <= 1;

NEXT\_SEL <= 1;

A\_SEL <= 1;

DONE <= 0;

StateNext <= S\_next;

end

S\_next: begin

LD\_SUM <= 0;

LD\_NEXT <= 1;

SUM\_SEL <= 1;

NEXT\_SEL <= 1;

A\_SEL <= 0;

DONE <= 0;

if (next\_zero == 0) StateNext <= S\_com;

else StateNext <= S\_done;

end

S\_done: begin

LD\_SUM <= 0;

LD\_NEXT <= 0;

SUM\_SEL <= 0;

NEXT\_SEL <= 0;

A\_SEL <= 0;

DONE <= 1;

if (start == 0) StateNext <= S\_sta;

else StateNext <= S\_done;

end

endcase

end

always @(posedge clk) begin

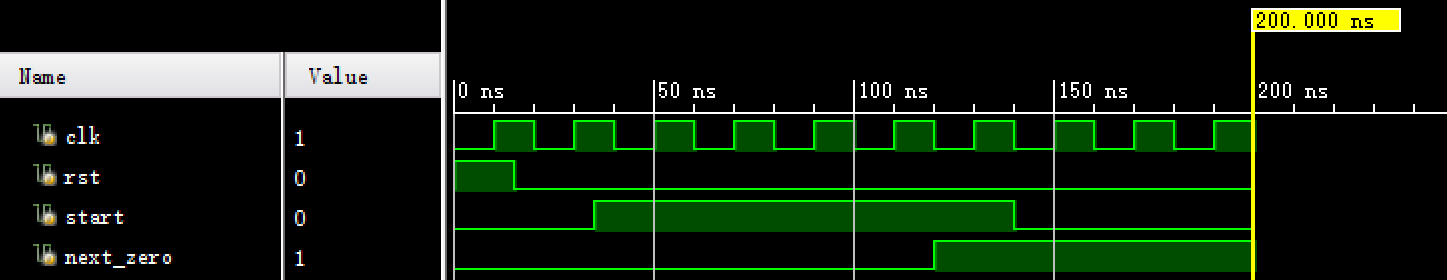
if ( rst == 1 ) State <= S\_sta;

else State <= StateNext;

end

endmodule

1. 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;

initial begin

clk = 1'b0;

rst = 1'b1;

start = 1'b0;

next\_zero = 1'b0;

#200 $stop;

end

always

#10 clk = !clk;

initial begin

#15 rst = !rst;

end

initial begin

#35 start = !start;

#105 start = !start;

end

initial begin

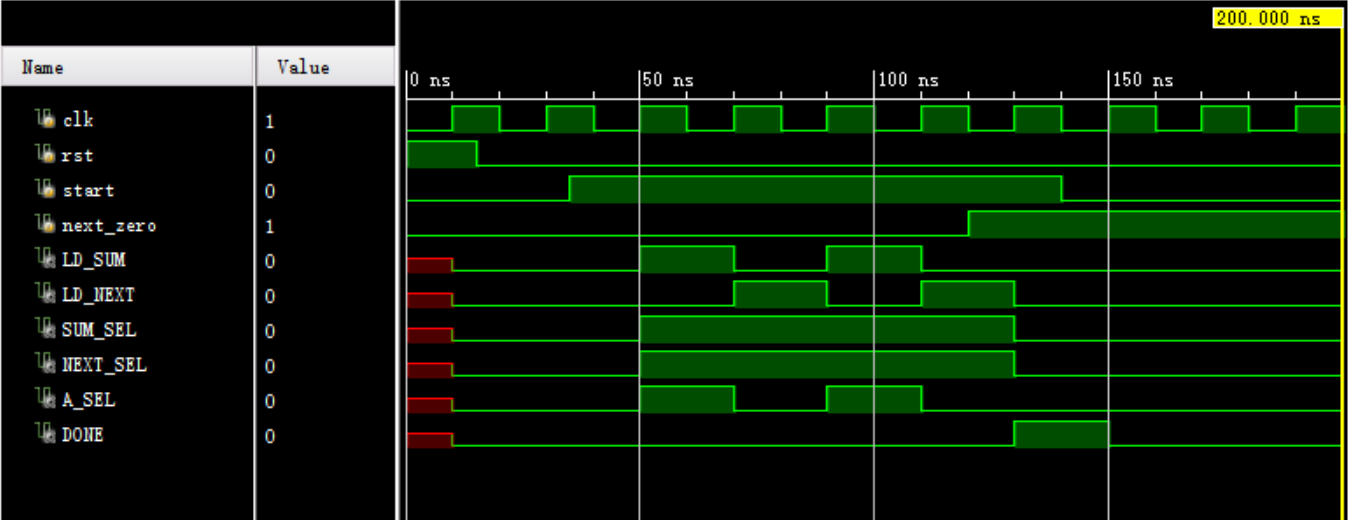
#120 next\_zero = !next\_zero;

end

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

endmodule

1. 仿真结果图：

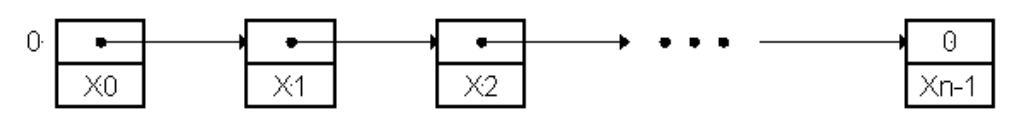


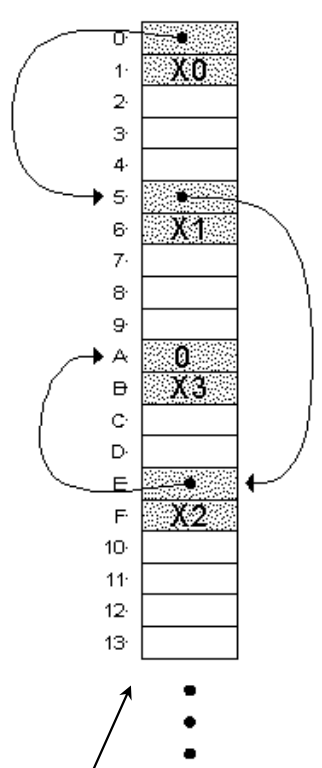
**3. 自动运算电路的设计（步骤3）**

【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。

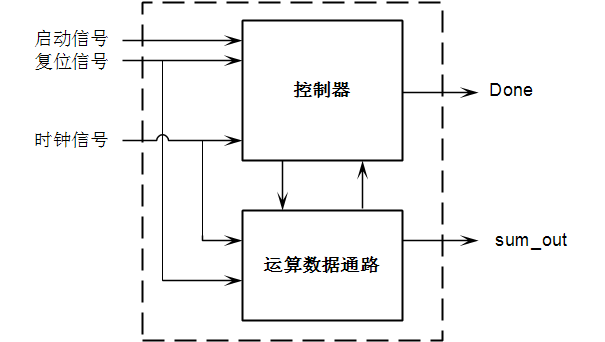




**图7 数据链表及其在存储器中的存放格式**

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

`timescale 1ns / 1ps

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

input clk,rst,start;

output DONE;

output [31:0] sum\_out;

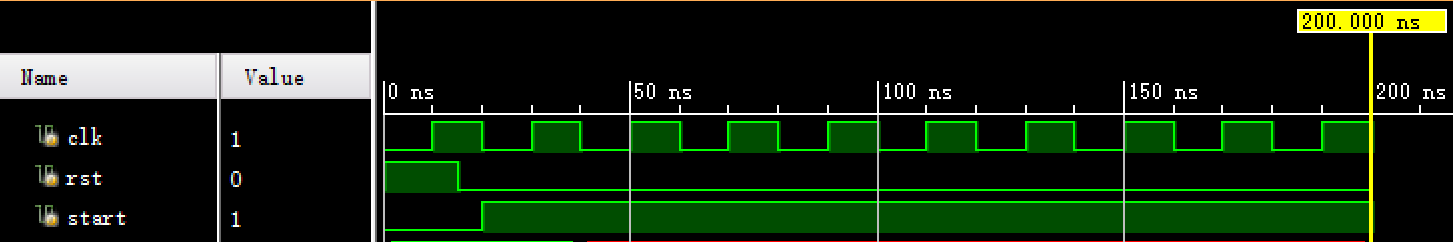
wire LD\_SUM, LD\_NEXT, SUM\_SEL, NEXT\_SEL, A\_SEL,DONE,NEXT\_ZERO;

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

FSM MyFSM (clk, rst, start, NEXT\_ZERO, LD\_SUM, LD\_NEXT, SUM\_SEL, NEXT\_SEL, A\_SEL,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 [31:0]sum\_out;

wire DONE;

initial begin

clk = 1'b0;

rst = 1'b1;

start = 1'b0;

#200 $stop;

end

always #10 clk = !clk;

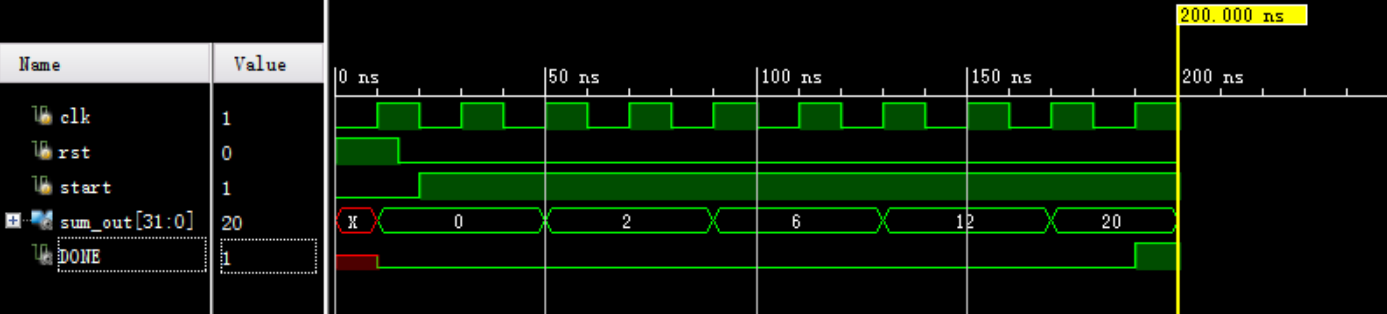
initial #15 rst = !rst;

initial #20 start = !start;

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

endmodule

1. 仿真结果图：



## 遇到的问题及解决方法

本次实验的难度是三次实验中最高的一次，主要体现在3个方面，第一，首次编写这种含有多个模块，输入输出端口极多的电路程序，并不太适应这种较为复杂的电路的编写和设计过程；第二，接触到了两个新的概念，数据通路和有限状态机，这两种器件虽然在数电中已经有所学习，但是使用Verilog语言编写仍然是第一次，导致我在编写的时候比较生疏；第三，本次编写的输入输出端口较多，导致容易出错，且不容易进行调试。

因此本次实验的主要问题也出在这几个方面，具体遇到的问题和解决方法如下所示：

（1）因为中间的连线非常多，输入输出端口也较为复杂，使得我经常性的连接错误，通过反复的检查，解决了这个问题。

（2）由于对有限状态机不够熟悉，再加上对Verilog语言中的保留字没有注意，导致出现了一些奇怪的错误，通过反复的检查发现可能是由于状态与保留字存在重合导致出错，通过修改状态名称解决问题。

## 实验心得、意见和建议

本次实验的难度较上一次又有了较大幅度的提高，接触到了比较困难且实际的应用，所幸设计流程都已经非常清晰的显示在任务书中，不需要自行设计电路，仅仅需要将设计完成的图纸转化为电路即可，即便如此，我也遇到了不少麻烦，花费了非常多的精力和时间再非常低级的错误上。通过本次实验，我初步掌握了实际简单工程的设计流程和代码编写方式，虽然不熟练，但是基本掌握了编写的流程，有了使用软件解决实际问题的基础，接下里就是要反复使用练习了。