实验4 数据通路和有限状态机设计

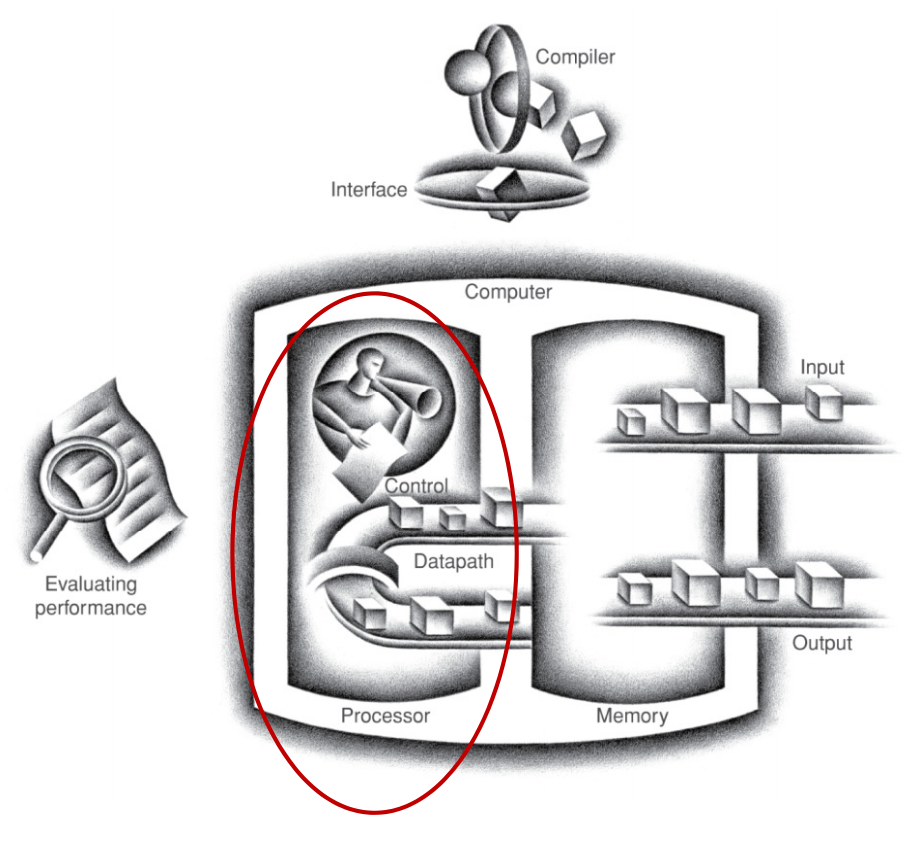
* 任务描述
* 相关知识
* 实验内容
* 遇到问题和解决方法
* 实验心得、意见和建议

## 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）。



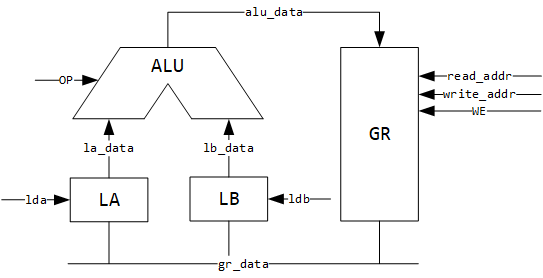
**图1处理器由数据通路和控制器构成**

## 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);**

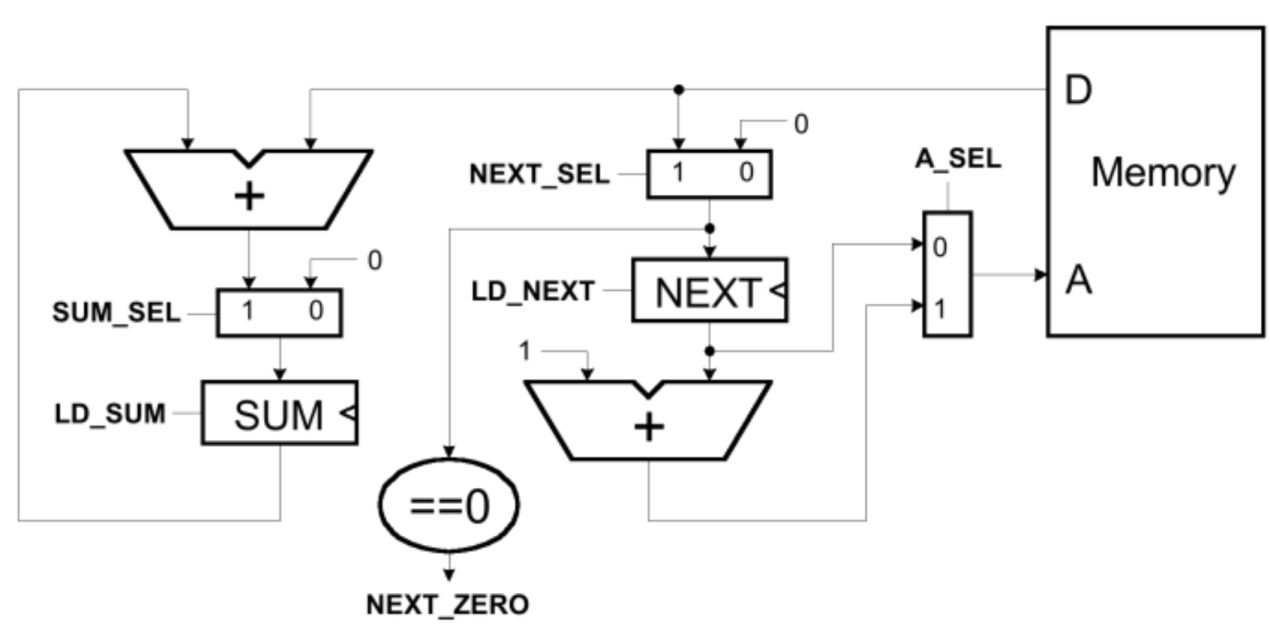
**endmodule**

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));

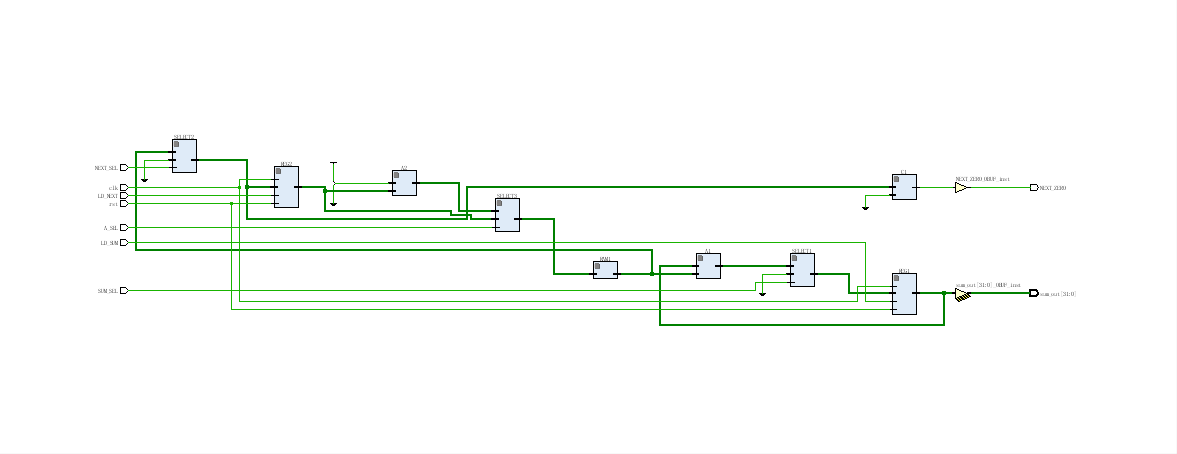
select\_2\_to\_1 SELECT2(.a(MEMORY\_DATA), .b(0), .c\_out(SELECT\_OUT2), .sel(NEXT\_SEL));

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)。

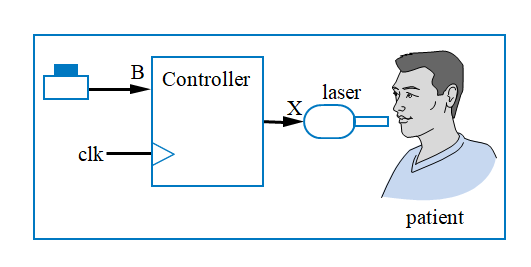


图4 激光计时器

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。

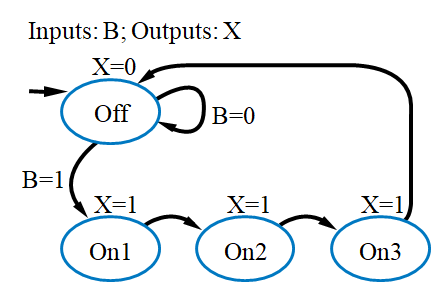


图5 该激光计时器的有限状态机

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);**

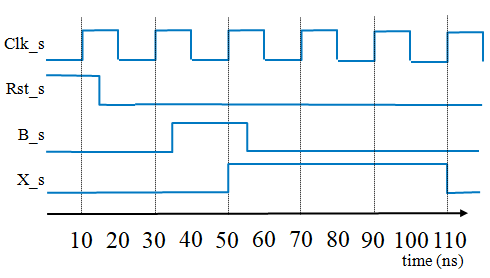
**#5 if (X\_s != 0)** //验证状态Off

**$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：

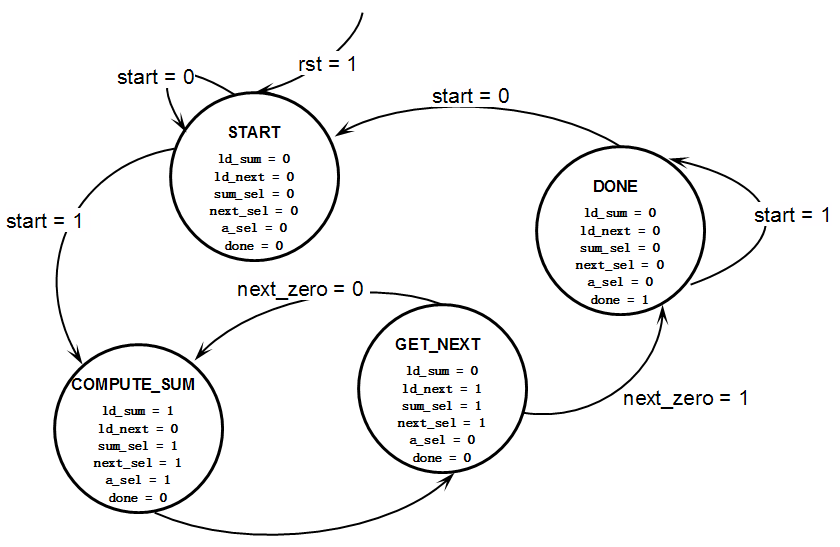
* 设计的有限状态机（FSM）能够正确综合；
* 编写有限状态机的仿真程序，完成有限状态机（FSM）的功能仿真，有限状态机功能仿真正确。

【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

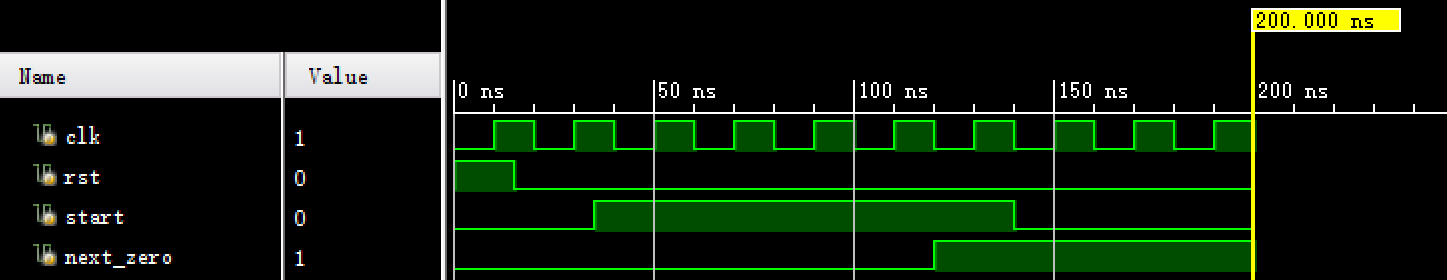
endcase

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

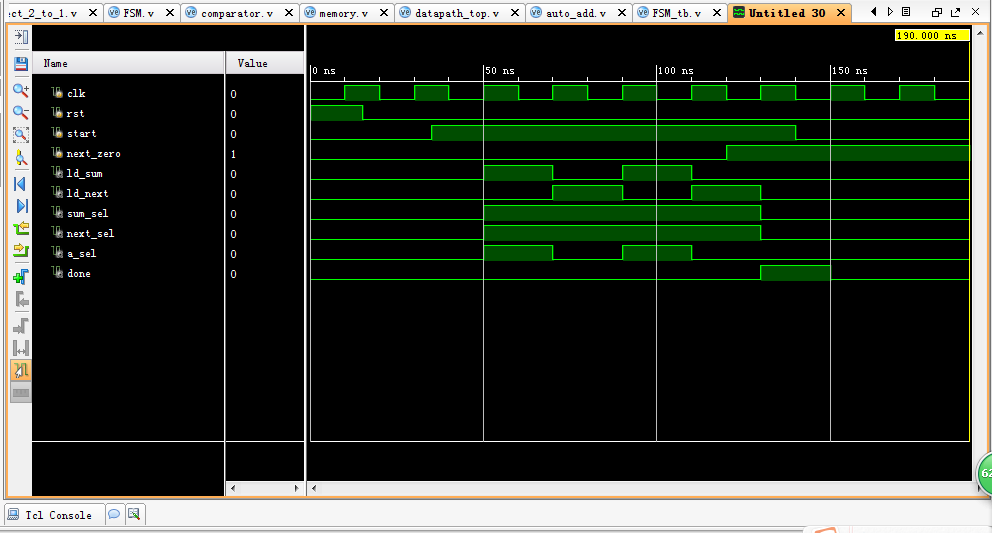
always begin

#10 clk <= ~clk;

end

endmodule

1. 仿真结果图：

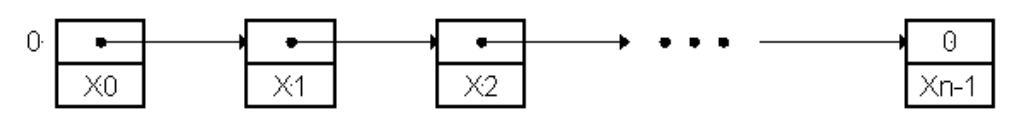


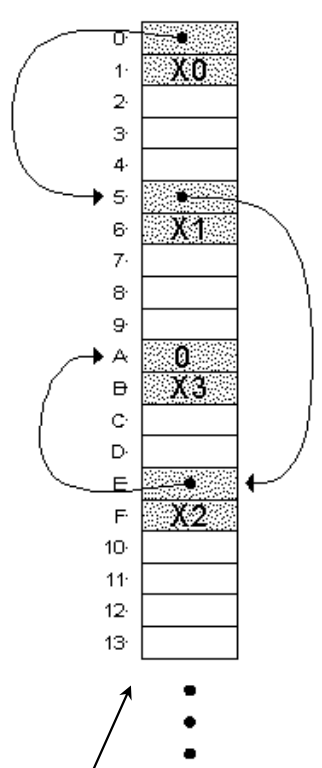
**3. 自动运算电路的设计（步骤3）**

【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。

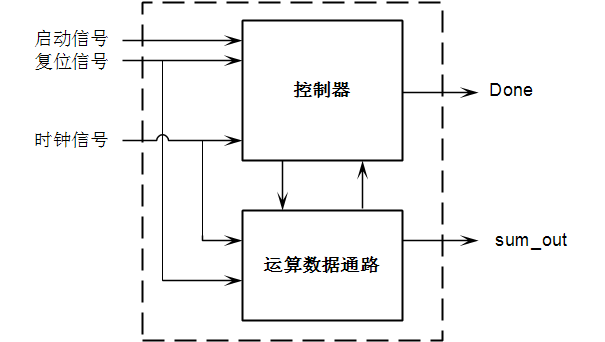




**图7 数据链表及其在存储器中的存放格式**

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;

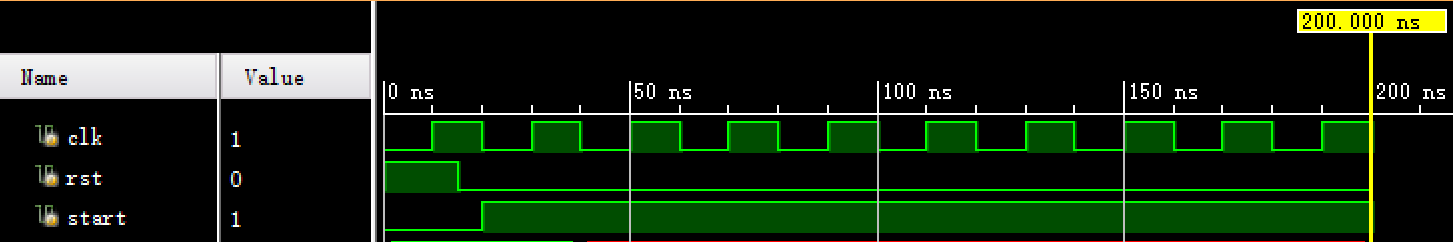
wire a\_sel;

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

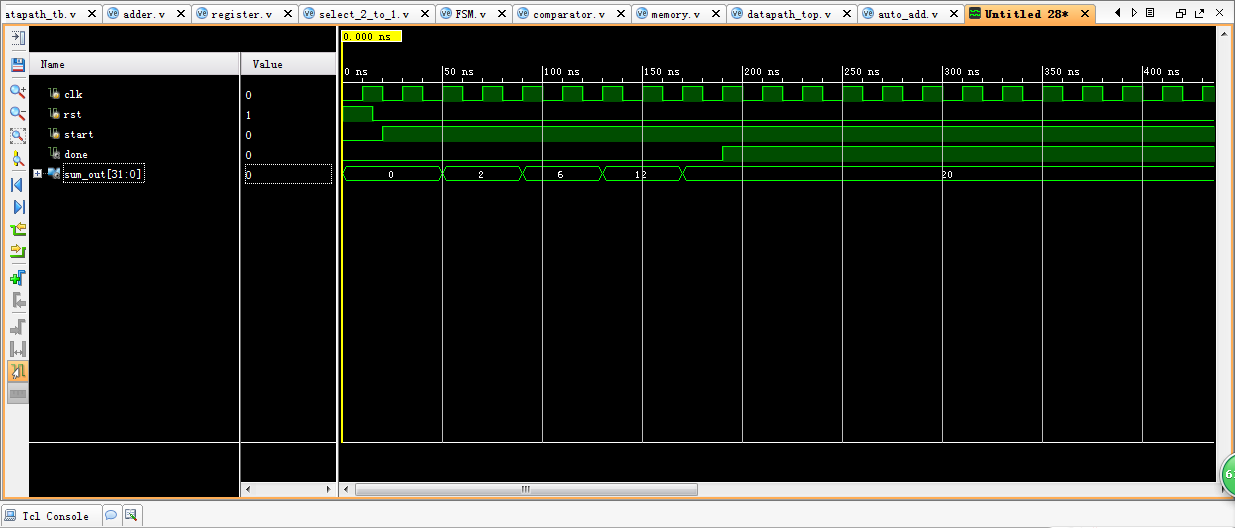
always begin

#10 clk<=~clk;

end

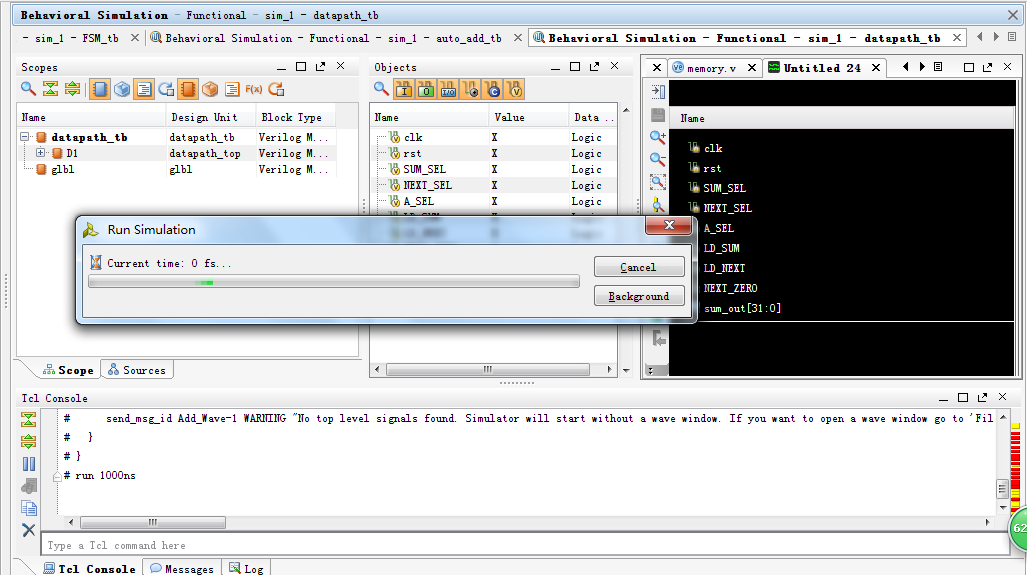
endmodule

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。