实验4 数据通路和有限状态机设计

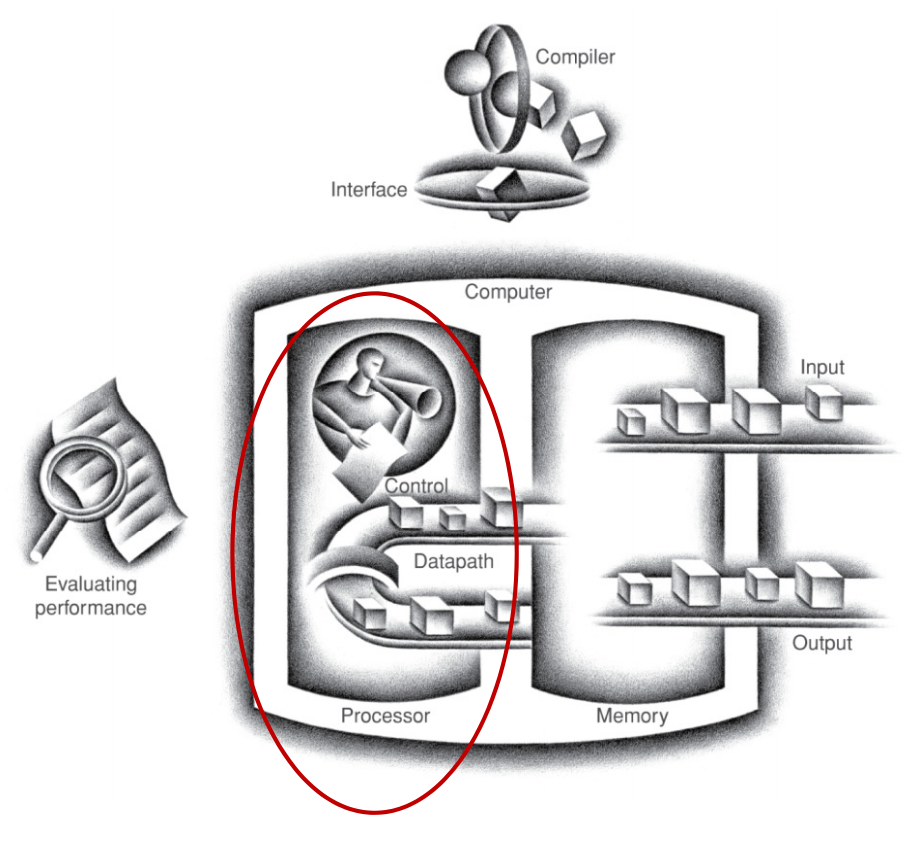
* 任务描述
* 相关知识
* 实验内容
* 遇到问题和解决方法
* 实验心得、意见和建议

## 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）。



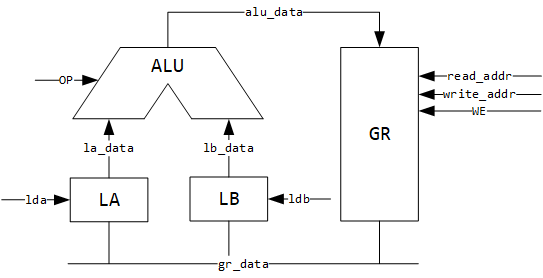
**图1处理器由数据通路和控制器构成**

## 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];**

**always @(posedge clk) begin**

**if (we)**

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

**initial begin //对存储器初始化**

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

**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);**

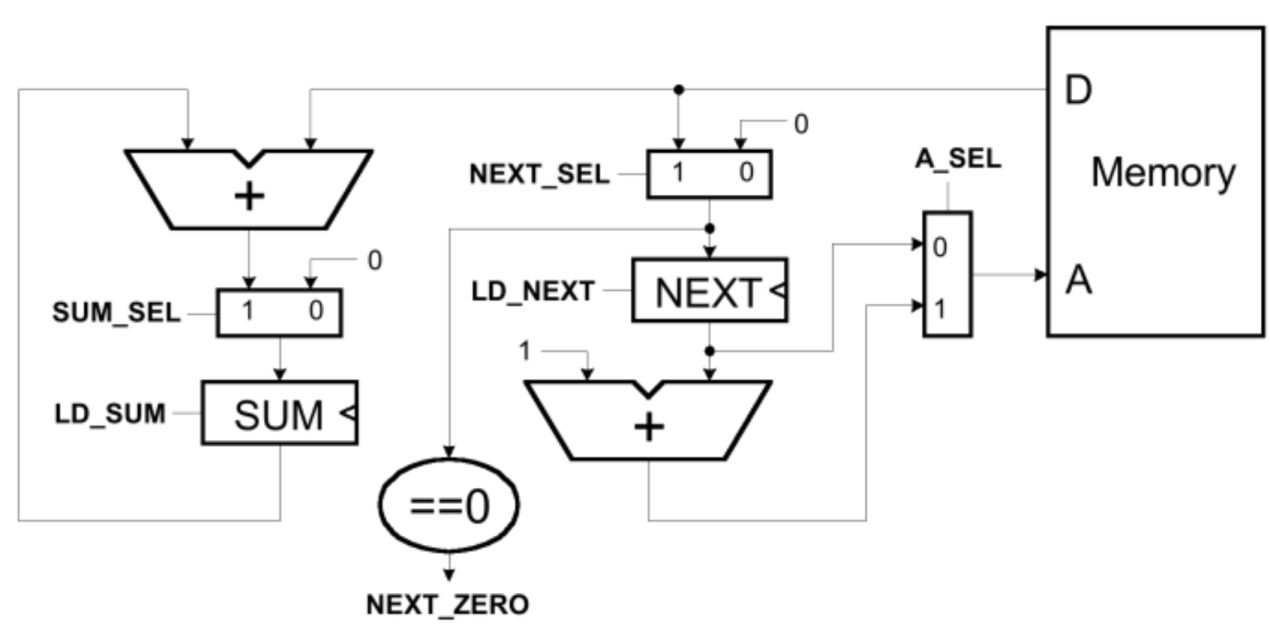
**endmodule**

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。



**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）

（1）加法器

module sum(a, b, c);

    parameter WIDTH = 8;

    input [WIDTH-1:0] a;

    input [WIDTH-1:0] b;

    output [WIDTH-1:0] c;

    assign c = a+b;

endmodule

（2）寄存器

module register(clk, rst\_n, en, d, q);

  parameter WIDTH = 8;

  input clk, rst\_n, en;

  input [WIDTH-1:0] d;

  output reg [WIDTH-1:0] q;

  always @(posedge clk) begin

    if (rst\_n) q <=0;

    else if (en) q <= d;

  end

endmodule

（3）2选1多路选择器

module mux\_21(a, b, sel, c);

    parameter WIDTH = 8;

    input [WIDTH-1:0] a, b;

    input sel;

    output reg [WIDTH-1:0] c;

    always @(a, b, sel) begin

        case(sel)

            1'b0: c = a;

            1'b1: c = b;

        endcase

    end

endmodule

（4）rom

module rom(read\_addr, data);

  parameter DATA\_WIDTH = 8;

  parameter ADDR\_WIDTH = 8;

  parameter INIT\_FILE = "sum\_init.mem";

  input [ADDR\_WIDTH-1:0] read\_addr;

  output [DATA\_WIDTH-1:0] data;

  reg [DATA\_WIDTH-1:0] rom[15:0];

  initial begin

        $readmemh(INIT\_FILE, rom);

  end

  assign data = rom[read\_addr];

endmodule

（5）比较器

module is\_equal(a, b, c);

    parameter WIDTH = 8;

    input [WIDTH-1:0] a;

    input [WIDTH-1:0] b;

    output c;

    assign c = (a==b);

endmodule

1. 数据通路的代码：

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

    parameter WIDTH = 32;

    parameter ADDR\_WIDTH = 32;

    parameter SUM\_FILE = "sum\_init.mem";

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

    output NEXT\_ZERO;

    output [WIDTH-1:0] sum\_out;

    wire [WIDTH-1:0] tmp, next\_addr, read\_addr, sum\_tmp, data, next\_addr\_tmp, data\_addr;

    wire [WIDTH-1:0] low = 'h0;

    wire [WIDTH-1:0] unit = 'h1;

    register #(WIDTH) SUM(clk, rst, LD\_SUM, tmp, sum\_out);

    register #(WIDTH) NEXT(clk, rst, LD\_NEXT, next\_addr\_tmp, next\_addr);

    rom #(WIDTH, ADDR\_WIDTH, SUM\_FILE) GR(read\_addr, data);

    mux\_21 #(WIDTH) SUM\_MUX(low, sum\_tmp, SUM\_SEL, tmp);

    mux\_21 #(WIDTH) NEXT\_MUX(low, data, NEXT\_SEL, next\_addr\_tmp);

    mux\_21 #(WIDTH) A\_MUX(next\_addr, data\_addr, A\_SEL, read\_addr);

    sum #(WIDTH) data\_sum(sum\_out, data, sum\_tmp);

    sum #(WIDTH) inc\_addr(unit, next\_addr, data\_addr);

    is\_equal #(WIDTH) endpoint(next\_addr\_tmp, low, NEXT\_ZERO);

endmodule

1. 数据通路的电路原理图（生成Schematic）：

A screenshot of a social media post

Description automatically generated

图4-1 数据通路的电路原理图

**2. 有限状态机设计（步骤2）**

【实验样例】

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

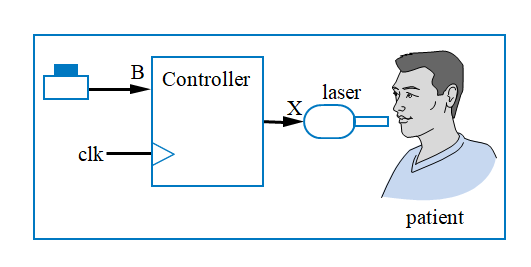


图4 激光计时器

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

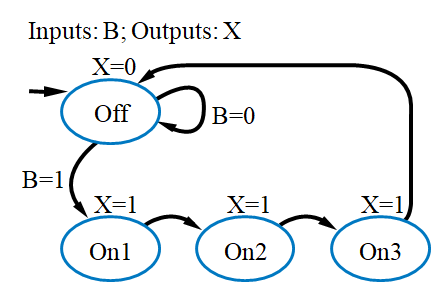


图5 该激光计时器的有限状态机

图5所示状态机的代码如下：

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

**input B;**

**output reg X;**

**input Clk, Rst;**

**parameter S\_Off = 0, S\_On1 = 1, S\_On2 = 2, S\_On3 = 3;**//对应四个状态

**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);**

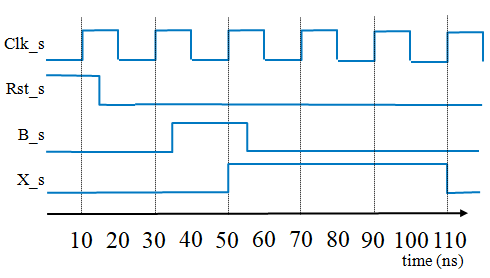
**#5 if (X\_s != 0)** //验证状态Off

**$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：

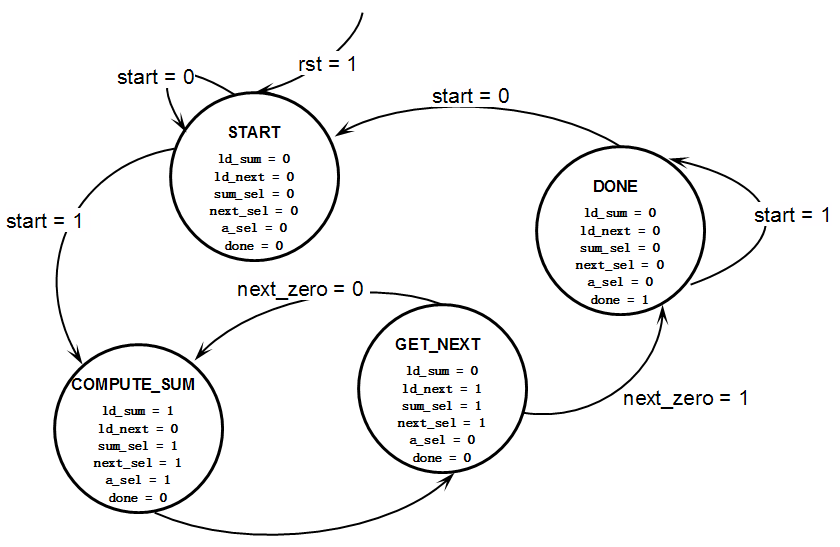
* 设计的有限状态机（FSM）能够正确综合；
* 编写有限状态机的仿真程序，完成有限状态机（FSM）的功能仿真，有限状态机功能仿真正确。

【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, State);

input clk,rst,start,next\_zero;

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

output reg [1:0] State;

parameter START=2'h0, COMPUTE\_SUM=2'h1, GET\_NEXT=2'h2, FINISHED=2'h3;

reg [1:0] NextState;

// CombLogic

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

case (State)

START: begin

LD\_SUM <= 1'b0;

LD\_NEXT <= 1'b0;

SUM\_SEL <= 1'b0;

NEXT\_SEL <= 1'b0;

A\_SEL <= 1'b0;

DONE <= 1'b0;

if (start == 2'b0)

NextState <= START;

else

NextState <= COMPUTE\_SUM;

end

COMPUTE\_SUM: begin

LD\_SUM <= 1'b1;

LD\_NEXT <= 1'b0;

SUM\_SEL <= 1'b1;

NEXT\_SEL <= 1'b1;

A\_SEL <= 1'b1;

DONE <= 1'b0;

NextState <= GET\_NEXT;

end

GET\_NEXT: begin

LD\_SUM <= 1'b0;

LD\_NEXT <= 1'b1;

SUM\_SEL <= 1'b1;

NEXT\_SEL <= 1'b1;

A\_SEL <= 1'b0;

DONE <= 1'b0;

if (next\_zero == 1'b0)

NextState <= COMPUTE\_SUM;

else

NextState <= FINISHED;

end

FINISHED: begin

LD\_SUM <= 1'b0;

LD\_NEXT <= 1'b0;

SUM\_SEL <= 1'b0;

NEXT\_SEL <= 1'b0;

A\_SEL <= 1'b1;

DONE <=1’b1;

if (start == 1'b0)

NextState <= START;

else

NextState <= FINISHED;

end

endcase

end

// StateReg

always @(posedge clk) begin

if (rst == 1'b1 )

State <= START;

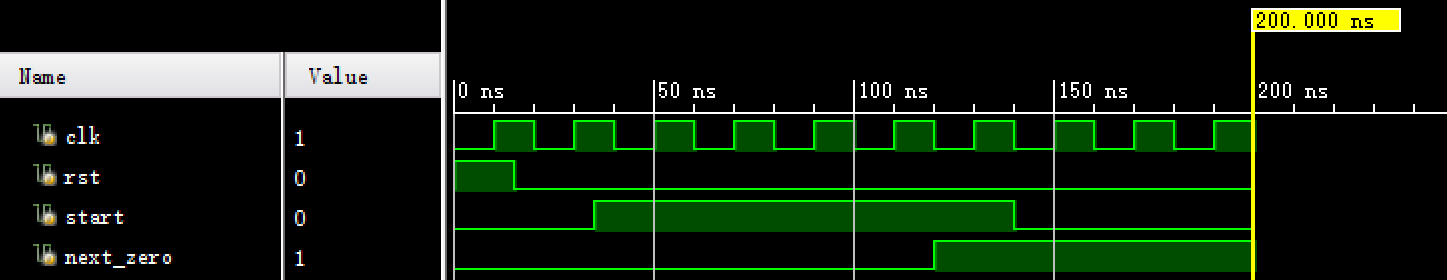
else

State <= NextState;

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;

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

initial begin

clk = 1'b0;

#200 $stop;

end

always

#10 clk = !clk;

initial begin

rst = 1'b1;

#15 rst = 1'b0;

end

initial begin

start = 1'b0;

#35 start = 1'b1;

#105 start = 1'b0;

end

initial begin

next\_zero = 1'b0;

#110 next\_zero = 1'b1;

end

endmodule

1. 仿真结果图：

A picture containing green, sitting, holding, screen

Description automatically generated

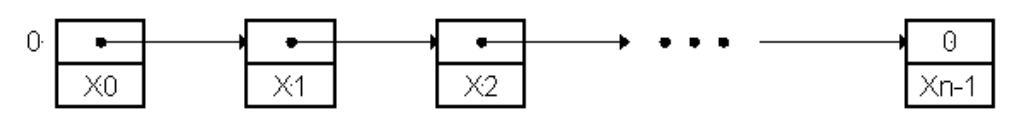
图4-2 有限状态机仿真结果图

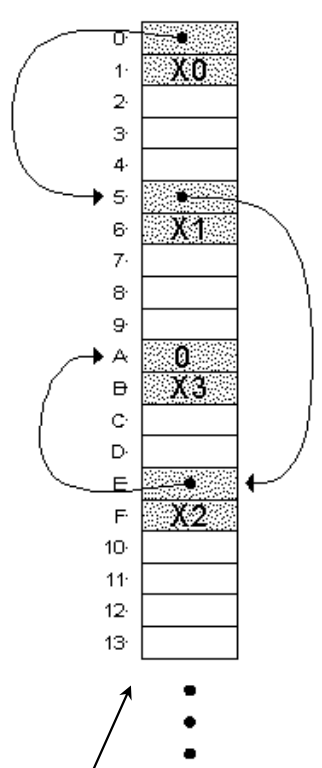
**3. 自动运算电路的设计（步骤3）**

【实验要求】

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

在存储器中存放的数据链表（第5页所示链表）其结构如下图7所示，链表的各个节点在存储器中不是连续存放，各节点的第一个地址存放下一个节点的地址，各节点的第二个地址中存放着要进行求和运算的数据，当下一个节点的地址为0时，表示到达链表的结尾，求和运算结束。

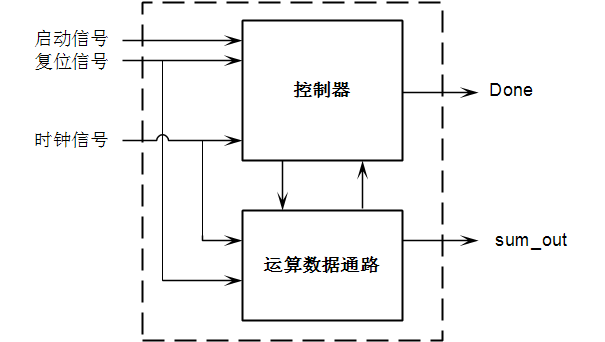




**图7 数据链表及其在存储器中的存放格式**

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

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



**图8 自动运算电路模块构成图**

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

00000003

00000002

00000000

00000007

00000004

00000000

00000000

0000000b

00000006

00000000

00000000

00000000

00000008

00000000

00000000

00000000

【实验提示】

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

该控制器模块的端口有：

输入端口：时钟clk，复位rst，启动求和start

输出端口: 求和结束DONE，求和结果sum\_out

**【实验填写】**

根据实验提示完成实验要求，包括：

1. 图8的Verilog程序代码

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

`timescale 1ns / 1ps

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

parameter WIDTH = 32;

parameter ADDR\_WIDTH = 32;

parameter SUM\_FILE = "sum\_init.mem";

input clk,rst,start;

output DONE;

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

wire [1:0] State;

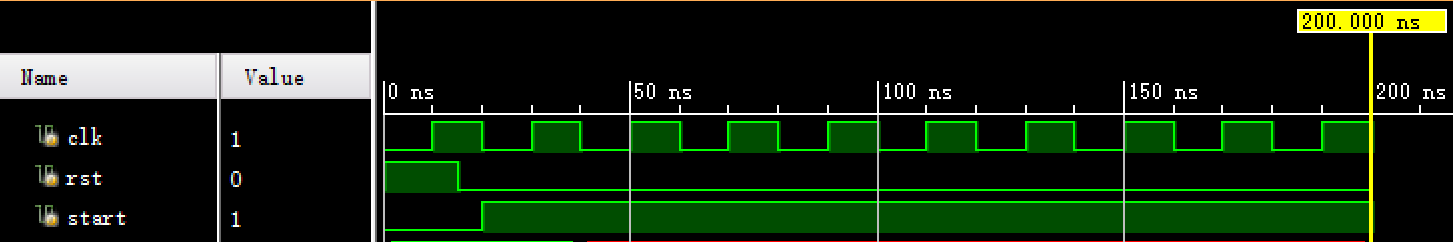
FSM myFSM(clk,rst,start, NEXT\_ZERO, LD\_SUM, LD\_NEXT, SUM\_SEL, NEXT\_SEL, A\_SEL, DONE, State);

datapath\_top #(WIDTH, ADDR\_WIDTH, SUM\_FILE) myDataPath(clk, rst, SUM\_SEL, NEXT\_SEL, A\_SEL, LD\_SUM, LD\_NEXT, NEXT\_ZERO, sum\_out);

assign next\_zero = NEXT\_ZERO ? 1 : 0 ;

endmodule

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 [7:0] sum\_out;

wire DONE;

auto\_add #(32, 32, "sum\_init.mem") myAdder(clk,rst,start,DONE,sum\_out);

initial begin

clk = 1'b0;

#200 $stop;

end

always begin

#10 clk = !clk;

end

initial begin

rst = 1'b1;

#15 rst = 1'b0;

end

initial begin

start = 1'b0;

#20 start = 1'b1;

end

endmodule

1. 仿真结果图：

A picture containing monitor, green, ball, screen

Description automatically generated

图4-3 自动加法器仿真结果图

## 遇到问题和解决方法

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

A picture containing monitor, indoor, green, playing

Description automatically generated

图4-4 错误的输出

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

A screenshot of a cell phone

Description automatically generated

图4-5 标结点检查结构描述

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

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



图4-6 rom波形图

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

A screen shot of a video game

Description automatically generated

图4-7 有关read\_addr的波形图

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

## 实验心得、意见和建议

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

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

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