

**计算机组成实验报告**

|  |  |
| --- | --- |
| 实验名称： | ALU、Regfile设计与有限状态机 |
| 姓 名： |  |
| 学 号： |  |
| 专 业： | 计算机科学与技术 |
| 课程名称： | 计算机组成 |
| 实验地点： | 东四-509 |
| 指导老师： |  |

2023 年 3 月 9 日

# 实验1—ALU、Regfile设计

# 实验目的和要求

1.1实验目的

1. 复习寄存器传输控制技术

2. 掌握CPU的核心组成：数据通路与控制器

3. 设计数据通路的功能部件

4. 进一步了解计算机系统的基本结构

5. 熟练掌握IP核的使用方法

1.2实验要求

熟悉SOC系统的原理，并掌握IP核集成设计CPU的方法，了解数据通路结构并实现ALU和 Register Files.

# 实验内容和原理

## 2.1 ALU设计

2.1.1 实验任务

设计实现数据通路部件ALU （采用原理图的设计方法）。

2.1.2 ALU基本功能分析

本实验ALU设计基于已实现五项基本运算的 ALU 模块（包括加法、减法、与操作、或操作与“set on less than”功能）；在此基础上扩展“异或”、“或非”与“位移”三项额外的功能。

输入输出介绍：

输入: 1. 32位数据A

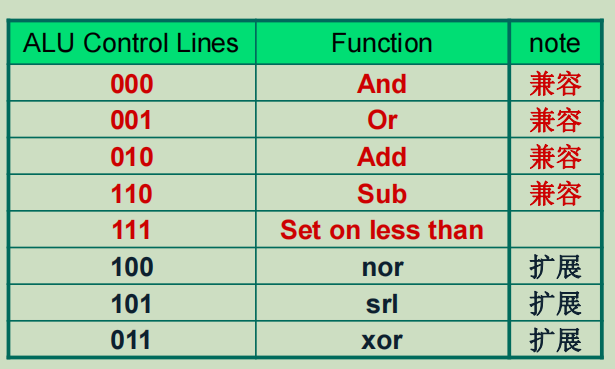
1. 32位数据B
2. 控制功能选择信号ALU\_operation

输出： 1.（经MUX挑选后的）result输出信号

2.是否发生溢出操作的信号overflow

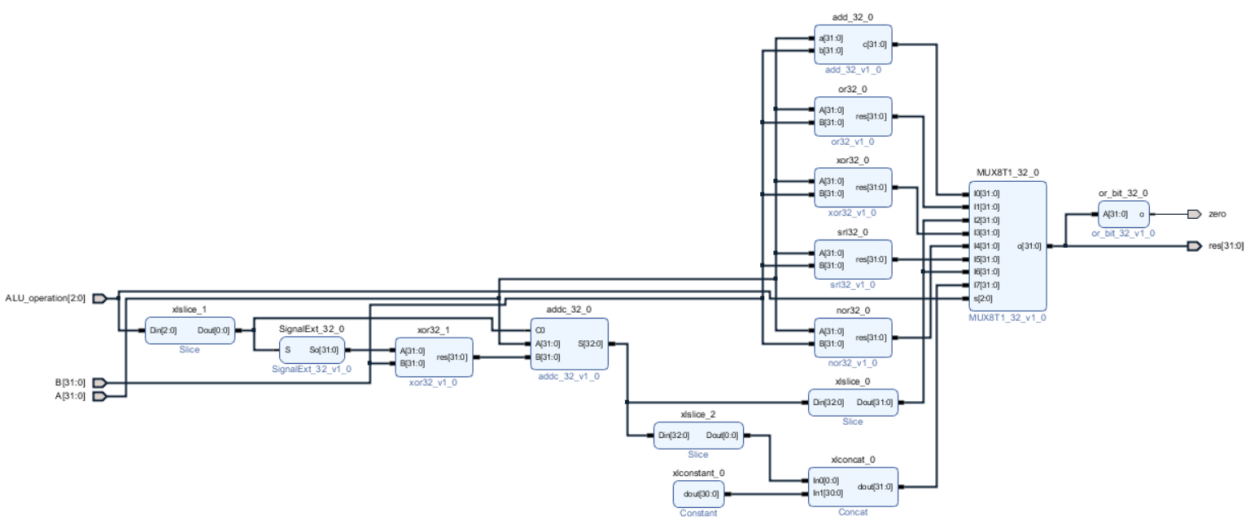
3.0输出（用于特殊表达）

模块功能对映控制信号选择表如下：



2.1.3 逻辑原理图输入设计ALU

逻辑原理设计图如下：



说明：图中使用Slice、Concat、Constant模块为基础模块，其余为自主设计模块。

主要模块及功能简介

SignalExt\_32\_v1\_0: 将输入符号位拓展至32位；

Xor32\_v1\_0: 对输入32位数据进行“异或”操作；

Addc\_32\_v1\_0: 对输入32位数据进行加法或减法操作；

Or32\_v1\_0: 对输入32位数据进行“或”操作；

Srl32\_v1\_0: 对输入32位数据进行位移操作；

Nor32\_v1\_0: 对输入32位数据进行“或非”操作；

MUX8T1\_32\_v1\_0: 根据控制信号挑选32位输出；

Or\_bit\_32\_v1\_0: 对输入32位数据进行逐位“或”操作；

## 2.2 Regfile设计

2.2.1 实验任务

设计实现数据通路部件32\*32bit Register Files（采用硬件描述语言的设计方法）。

2.2.2 Regfile 基本功能介绍

Regfile为数字系统的功能组件之一；

端口说明：

二个读入端口

输入端口（寄存器输入选择信号）：

1. Rs1\_addr；

2. Rs2\_addr；

输出端口（寄存器数据贮存浏览）：

1.Rs1\_data；

2.Rs2\_data；

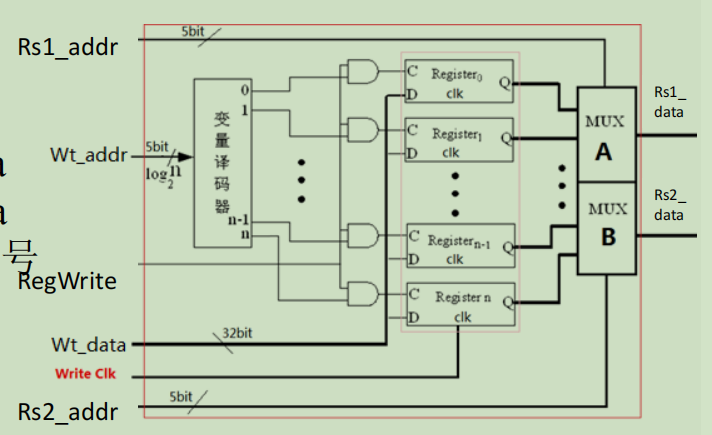
一个写端口

输入端口：

 1.Wt\_addr：写入寄存器选择信号；

2.Wt\_data：写入具体数据信号；

 3.RegWrite：写使能信号；

逻辑电路链接示意图如下：

2.2.3 Regfile封装

因为寄存器堆的设计带有clk和rst，所以直接封装会存在两个问题：端口警告与复位信号自动反向。

其中端口警告原因是clk端口属性未知，复位信号自动反向原因是系统默认是低电平而实验设计时高电平，需要进行属性约束。

因而，我们需要在clk的parameters下新建ASSOCIATED\_BUSIF，并在这个参数后面的value列输入定义的时钟信号的名字，此处为clk；而对于rst，我们需要在其端口编辑界面的Parameters属性下添加POLARITY，并将参数后面的value列输入属性ACTIVE\_HIGH.

# 实验设备和环境

 实验设备：

1. 计算机（Intel Core i5以上，4GB内存以上）系统

2. Sword2.0/Sword4.0开发板

3. Xilinx VIVADO2017.4及以上开发工具

 材料：

无

# 实验实现方法、步骤与调试

4.1 ALU实验实现与仿真

4.1.1 ALU模块的verilog代码实现

具体实现代码如下：

module verilogALU(

input wire [31:0] A,B,

input wire [2:0] ALU\_operation,

output reg [31:0] res,

output reg zero

);

reg [31:0] a,b;

reg [32:0] c;

always @(\*)

begin

case(ALU\_operation)

0://and

begin res=A&B; end

1://or

begin res=A|B; end

2://add

begin res=A+B; end

6://sub

begin res=A-B; end

7://set on less than

begin

a={32{ALU\_operation[2]}};

b=a+B;

c=b+A+ALU\_operation[2];

res=c[32];

end

4://nor

begin res=~(A|B); end

5://srl

begin res=A>>1; end

3://xor

begin res=A^B; end

endcase

zero=~(|res);

end

endmodule

4.1.2 ALU模块仿真代码

具体仿真代码如下：

module ffffry;

reg [31:0]A;

reg [31:0]B;

reg [2:0] ALU\_operation;

wire [31:0] res;

wire zezo;

verilogALU m0(

.A(A),.B(B), .ALU\_operation(ALU\_operation),

.res(res), .zero(zero) );

initial begin

A=32'ha5a5a5a5;

B=32'h5a5a5a5a;

ALU\_operation=3'b111;#100;

ALU\_operation=3'b110; #100;

ALU\_operation=3'b101;#100;

ALU\_operation=3'b100; #100;

ALU\_operation=3'b011; #100;

ALU\_operation=3'b010; #100;

ALU\_operation=3'b001; #100;

ALU\_operation=3'b000; #100;

end

endmodule

4.2 Regfile的设计与仿真

4.2.1 Regfile的verilog代码实现

具体代码如下：

module regfile(

input clk,rst,regwrite,

input [4:0] rs1\_addr,rs2\_addr,wt\_addr,

input [31:0]wt\_data,

output [31:0] rs1\_data,rs2\_data

);

reg [31:0] register[1:31];

integer i;

assign rs1\_data=(rs1\_addr==0)?0:register[rs1\_addr];

assign rs2\_data=(rs2\_addr==0)?0:register[rs2\_addr];

always@(posedge clk or posedge rst)

begin

if(rst==1)

for(i=0;i<32;i=i+1) register[i]<=0;

else if((wt\_addr!=0)&&(regwrite==1))

register[wt\_addr]<=wt\_data;

end

Endmodule

4.2.2 Regfile的仿真代码

具体代码如下：

module regfiletext;

reg clk,rst,regwrite;

reg [4:0] rs1\_addr,rs2\_addr,wt\_addr;

reg [31:0]wt\_data;

wire [31:0] rs1\_data,rs2\_data;

regfile m0(.clk(clk),.rst(rst),.regwrite(regwrite),

.rs1\_addr(rs1\_addr),.rs2\_addr(rs2\_addr),.wt\_addr(wt\_addr),

.wt\_data(wt\_data),.rs1\_data(rs1\_data),.rs2\_data(rs2\_data) );

initial forever begin

clk=1;#5;

clk=0;#5;

end

initial begin

rst=1;

regwrite=0;

rs1\_addr=0;rs2\_addr=0;

wt\_addr=0; wt\_data=0;

#100;

rst=0;

regwrite=1;wt\_addr[4:0]=4'h5;wt\_data[31:0]=32'ha5a5a5a5;#100;

regwrite=1;wt\_addr[4:0]=4'ha;wt\_data[31:0]=32'h5a5a5a5a; #100;

regwrite=0;rs1\_addr[4:0]=4'h5; #100;

regwrite=0;rs2\_addr[4:0]=4'ha; #100;

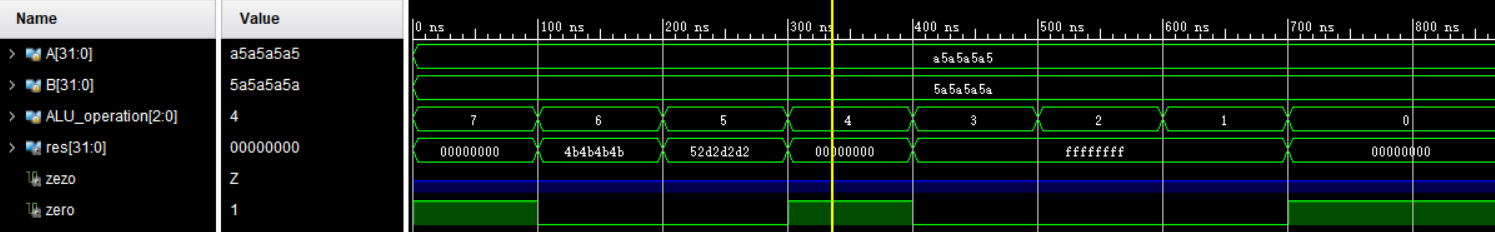
end

endmodule

# 实验结果与分析

5.1 ALU模块

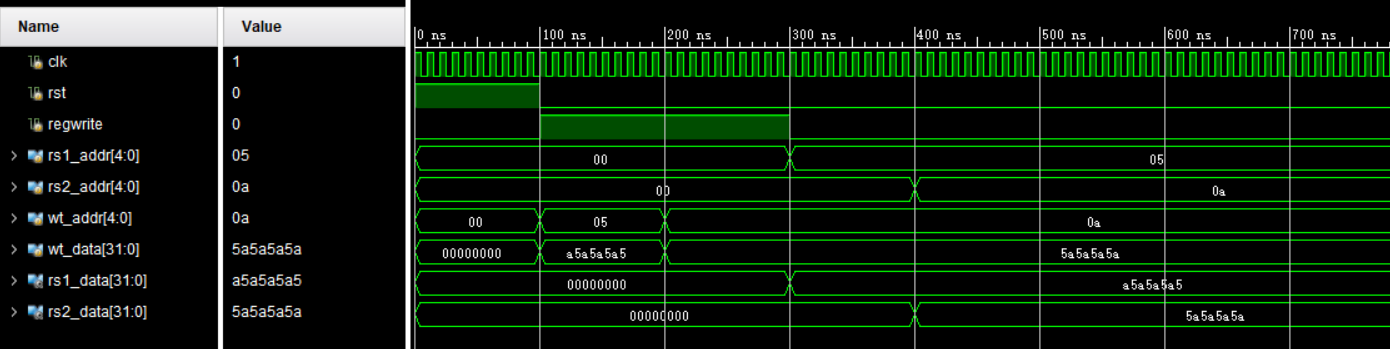
实验结果如下：



由上图可见，当我们输入的32位数据为 “5A5A5A5A”与“A5A5A5A5”时，当控制信号为“111”时，功能为“set on less than”,输出为32’h00000000时，zero信号为“1”；当控制信号为“111”时，功能为“set on less than”,输出为32’h00000000时，zero信号为“1”；当控制信号为“110”时，功能为减法,输出为32’h4b4b4b4b时，zero信号为“0”；当控制信号为“101”时，功能为向右位移,输出为32’h52d2d2d2时，zero信号为“0”；当控制信号为“100”时，功能为“或非”操作,输出为32’h00000000时，zero信号为“1”；当控制信号为“011”时，功能为“异或”操作,输出为32’hffffffff时，zero信号为“0”；当控制信号为“010”时，功能为加法,输出为32’hffffffff时，zero信号为“0”；当控制信号为“000”时，功能为“和”操作,输出为32’h00000000时，zero信号为“1”；均符合实验需求，因而证明了模块设计的正确性。

5.1 Regfile模块

实验仿真结果如下：



由上图可见，将reset置1后，所有寄存器的存贮值归0；将reset置0后则可以开始正常操作；当regwrite信号置1时，写入有效，这时wt\_addr指向需要写入的寄存器，wt\_data存贮需要写入的数据；由图可得，我们向寄存器“05”写入“32’hA5A5A5A5”，向寄存器“0A”写入“32’h5A5A5A5A”。

在后续操作中，rs1\_addr指向需要读取的寄存器“05”，rs2\_addr指向需要读取的寄存器“0a”，并分别将结果放入rs1\_data与rs2\_data中，读出数据为“32’hA5A5A5A5”与“32’h5A5A5A5A”；与我们写入的数据相符，因而设计符合需求功能需求。

# 实验2—有限状态机

# 实验目的和要求

1.1实验目的

1. 复习有限状态机的基本概念

2. 掌握有限状态机的两种模型

3. 设计有限状态机解决实际问题

1.2实验要求

熟悉有限状态机的基本原理，掌握moore和mealy两种类型状态机，设计并实现状态机解决实际问题。

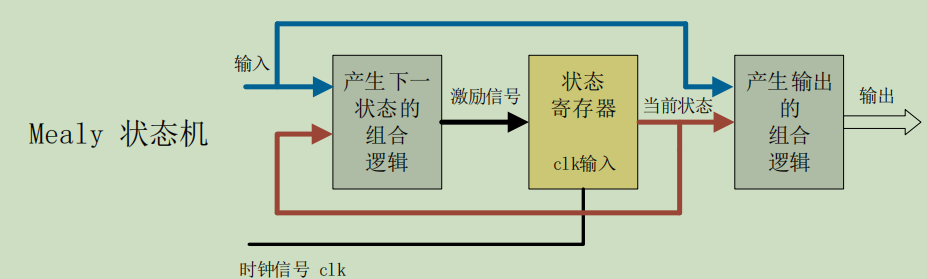
# 二、实验内容和原理

2.1状态机的基本概念

有限状态机（Finite State Machine，简称FSM）在有限个状态之间按一定规律转换的时序电路。

有限状态机通常是由寄存器组和组合逻辑组成时序电路，根据当前状态和输入信号可以控制下一个状态的跳转，有限状态机在电路中通常是作为控制模块，作为整个电路模块的核心而存在。

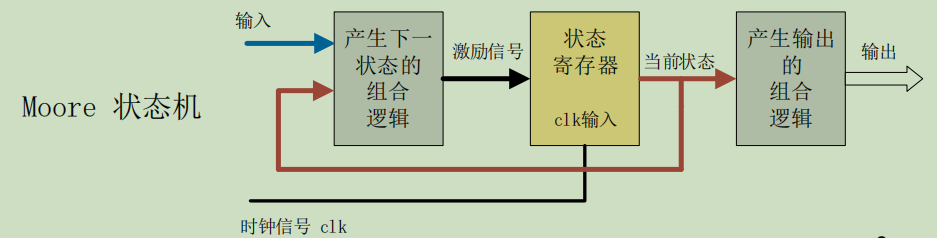
有限状态机主要分为两大类：

Mealy模型：其组合逻辑的输出不仅与当前状态有关，还与输入有关。

其示意图如下：

Moore模型：其组合逻辑的输出只与当前的状态有关。

其示意图如下：



另外，状态寄存器由一组触发器组成，用来记忆状态机当前所处的状态，状态的改变只发生在时钟的跳变沿。

状态是否改变、如何改变，取决于组合逻辑F的输出，F是当前状态和输入信号的函数。

状态机的输出是由输出组合逻辑G提供的，G也是当前状态和输入信号的函数。

2.2状态机的设计方法

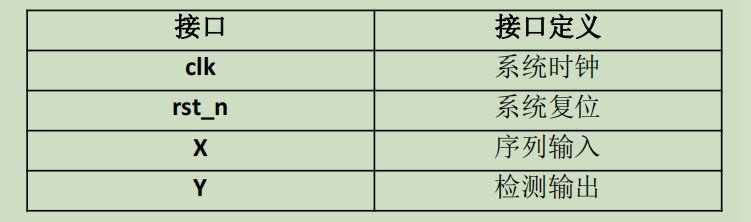
1.一段式描述（即状态跳转与输出信号都在同一个always块里面进行描述）。

2.二段式描述（即将输出信号,与状态跳转分开描述，便于设计代码管理）。

3.三段式描述（即将输出信号,与状态跳转分开描述，并且状态跳转用组合逻辑来控制）。

2.3状态机设计步骤

2.3.1系统架构和接口定义

示意图如下:

2.3.2状态定义和编码

状态机的编码方式主要包括：二进制码（Binary），格雷码（gray），独热码(one hot) 。

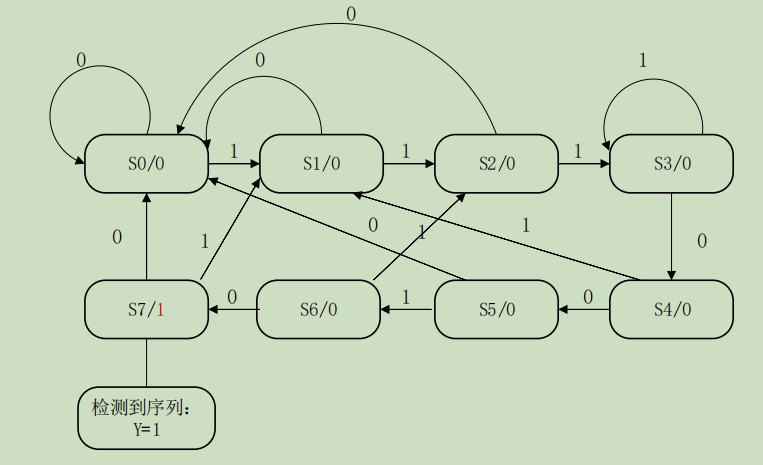
格雷码相对于二进制码而言，在状态跳转的时候，只有单比特翻转，它的功耗相对比较低。独热码相对于格雷码或者二进制码而言，它增加了两个寄存器来表示状态，但是它会更节省组合逻辑电路，因为它在比较状态的时候，只需要比较一个比特位，那么其电路的速度和可靠性就会增加。

四状态各编码方式编码举例如下：



2.3.3状态转换图

举例如下：



# 三、实验设备和环境

 实验设备：

1. 计算机（Intel Core i5以上，4GB内存以上）系统

2. Sword2.0/Sword4.0开发板

3. Xilinx VIVADO2017.4及以上开发工具

 材料：

无

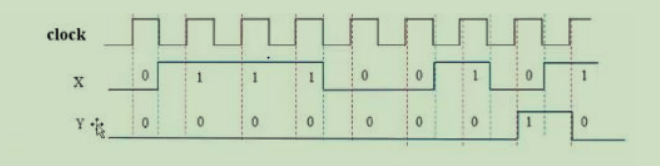
# 四、实验实现方法、步骤与调试

4.1实验具体内容

序列检测器，设计要求：用状态机设计序列检测器（1110010）。

设计功能：设计一个序列检测器，检测的序列为“1110010”；当输入信号X依次为“1110010”时，输出信号Y输出一个高电平，否则输出信号Y为低电平。

时序图：序列检测器是一种同步时序电路，它用于搜索，检测输入的二进制代码串中是否出现指定的代码序列，1110010序列检测原理图如下：



4.2 三段式设计代码如下

module threesement(

input wire clk,

input wire reset,

input wire in,

output wire out);

parameter[2:0]S0=3'b000,

S1=3'b001,

S2=3'b010,

S3=3'b011,

S4=3'b100,

S5=3'b101,

S6=3'b110,

S7=3'b111;

reg [2:0] current\_state;

reg [2:0] next\_state;

always@(posedge clk or negedge reset) begin

if(!reset)begin current\_state<=S0;end

else current\_state<= next\_state;

end

always@(\*)begin

case(current\_state)

S0:if (in==0) next\_state=S0;

else next\_state=S1;

S1:if (in==0) next\_state=S0;

else next\_state=S2;

S2:if (in==0) next\_state=S0;

else next\_state=S3;

S3:if (in==0) next\_state=S4;

else next\_state=S3;

S4:if (in==0) next\_state=S5;

else next\_state=S1;

S5:if (in==0) next\_state=S0;

else next\_state=S6;

S6:if (in==0) next\_state=S7;

else next\_state=S2;

S7:if (in==0) next\_state=S0;

else next\_state=S1;

endcase

end

assign out=(current\_state==S7)?1:0;

endmodule

三段式可以在组合逻辑后再增加一级寄存器来实现时序逻辑输出：

1、可以有效地滤去组合逻辑输出的毛刺。

2、可以有效地进行时序计算与约束。

3、另外对于总线形式的输出信号来说，容易使总线数据对齐，从而减小总线数据间的偏移，减小接收端数据采样出错的频率。

4.3 三段式设计有限状态机仿真

具体代码如下：

module threesementsim;

wire out;

reg clk;

reg reset;

reg in;

threesement m0(.clk(clk),.reset(reset), .in(in), .out(out));

initial forever begin

clk=1;#10;

clk=0;#10;

end

initial begin

clk=0; reset=0; #15;

reset=1;

in=0;#20;

in=1;#20;

in=1;#20;

in=1;#20;

in=0;#20;

in=0;#20;

in=1;#20;

in=0;#20;

in=1;#20;

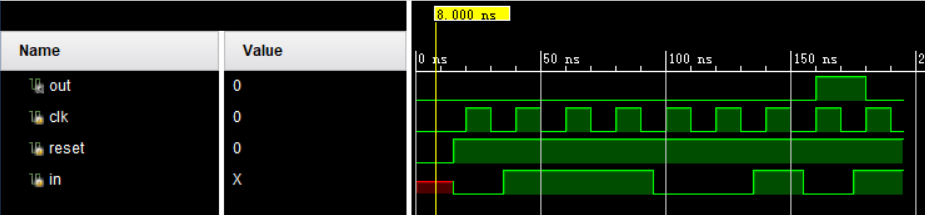
$finish;

end

endmodule

# 五、实验结果与分析

三段式设计有限状态机仿真效果如下：



由上图可见，当reset信号置“0”时，有限机状态被重置，因而输出0；当reset置为“1”时，状态机开始工作；随着信号“1110010”的输入，在序列完整输入前，状态机一直保持0的输出；序列一旦完整输入，状态机立马输出“1”，且随着下一时刻时钟上升沿的到来，状态机的状态被改变或重置，从而输出“0”。

**实验讨论与新得**

第一次写计组的实验报告，若有不当，还望赵老师和助教哥哥斧正。

总体而言，lab1还是一如既往的方便，可能也是有源代码的加成，使实验实现本身并不复杂；在实验的过程中，我也渐渐熟悉了vivado的使用，和ise相比有使用便捷之处（特别是连线部分，还是很自动的QAQ），但是规范也是多了不少，不得不熟悉以遵守。

但在收获之外，我也还有不少疑问，想请赵老师和助教哥哥解惑：第一，如何实现ALU的循环移位功能，虽然在实验的实现要求中，我们只要求了右移一位的操作，但在课外探究循环移位的过程总，却发现实现并不那么简单，有不少奇奇怪怪的报错或时间运行问题；第二，在ALU模块的仿真过程中，我的“zero”变量输出了两次（即有两条输出刻画线），一条遵循着输出规律，符合每一时刻的预期输出效果，而另一条却一直维持蓝色，且不上下移动，很想知道这个情况出现的原因或存在着什么实现问题。