数字 EDA 基础
上机实验报告

学　　院　　电子信息工程学院
班　　级　　130231 班
姓　　名　　何沃洲
学　　号　　13021264

2016 年 6 月

目 录

实验一 简单组合逻辑设计

1. 实验目的

（1）掌握基本组合逻辑电路的实现方法；
（2）初步了解两种基本组合逻辑电路的生成方法；
（3）学习测试模块的编写；
（4）通过综合和布局布线了解不同层次仿真的物理意义。

2. 实验内容

设计一个字节比较器：比较两个字节的大小，如 a[7:0]大于 b[7:0]，则输出高电平，否则输出低电平；并改写测试模型，使其能进行比较全面的测试。

以下为比较器和测试程序代码：

****
comparator.v
****

```
module comparator(out,in1,in2);
  input [7:0]in1,in2;
  output out;
 assign out=(in1>in2)? 1:0;
endmodule
```

****
测试文件 Test1.v
****

```
`timescale 1ns/1ns
module Test1;
  reg [7:0]a,b;
  reg clk;
  wire equal;
  initial
    begin
      a=0;
      b=0;
      clk=0;
    end
    always #100 clk=~clk;
    always @(posedge clk)
      begin
        a=$random%128;
        b=$random%128;
      end
    initial
      begin   #100000     $stop;
      end
  comparator
comparator(.in1(a),.in2(b),.out(equa
l));
endmodule
```

3. 设计思路

（1）字节比较器的实现

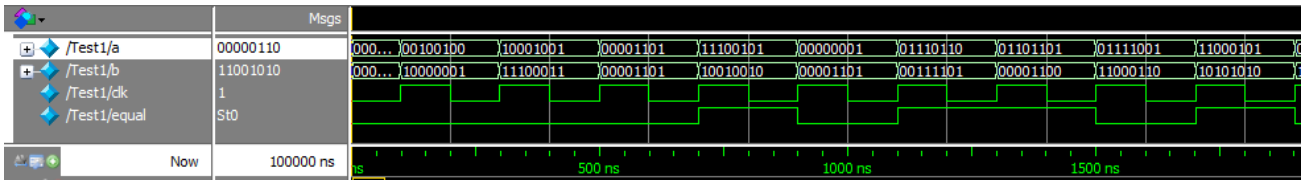
这个比较器应当有 3 个对外的端口，分别是：两个 8 位数据的输入端口，一个比较结果的输出端口。由于比较器的输出结果是随着输入变化立即变化的，因此这是一个组合逻辑的电路。主程序很容易实现。指导书中给出了一个可综合的数据比较器的示例程序，其中数据 a，b 均为 1bit。我们可以在模块中将 a、b 的定义改为字节型，即“input [7:0] a, b;”。

（2）测试程序的改进

书中所给的测试方法一原理很简单，但只对 a、b 等于 0 或 1 的情况进行了验证。测试模块要产生随机的两个 8 位输入数据，因此利用系统任务$random 来实现。通过一个时钟，每隔一段时间产生两个新的随机数，更充分地验证了 a 大于、小于、等于 b 时的输出情况。

4. 实验结果

波形：实验后得到的实验波形如下：

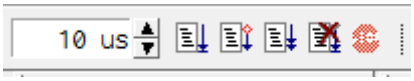


可以发现当 a>b,equ 输出高电平，反之输出了低电平，符合题目设计要求。

5. 思考题

在测试方法中，第二个 initial 有什么用？它与第一个 initial 块有什么关系？如果在第二个 initial 块中没有写#100000 或者$stop，仿真会如何进行？比较两种测试方法，哪一种方法更为全面？

1) 第二个 initial 块控制仿真何时停止；与第一个 initial 块同时开始互不影响；如果没有写#100000 则仿真开始时立即停止，没有写$stop 则仿真到达系统设置的时间才会停止；与第一种测试方法相比，这种方法覆盖到的数据更为广泛因而更为全面。在测试方法二中，第二个 initial 块用于设置仿真时长。
2) 它与第一个 initial 块为并行的关系。同一个测试文件中，可以有多个 initial 块，且均从 0 时刻，也就是模块运行开始时刻顺序运行，都只运行一次。
3) 如果在第二个 initial 块中，没有写#100000 $stop，仿真将会按测试界面设置的仿真时间进行仿真。（如下图）



4) 第二种测试方法更全面，因为第二种测试方法产生的 a 和 b 的值是随机的，可以测试多种情况，具有普遍性；而第一种测试方法只能按照测试文件中给定的数据进行测试，不具有普遍性。

6. 实验总结

本次实验是 Verilog 上机的第一个实验，内容比较简单，并且课本上已经给出了范例，只需要稍加修改便可以写出程序的代码。课本给出的示例代码其实有点冗余，是为了做示范需要，所以写了两个 initial 块，实际上代码可以都写在一个里面。

实验二 简单分频时序电路的设计

1. 实验目的

（1）掌握最基本时序电路的实现方法；
（2）学习时序电路测试模块的编写；
（3）学习综合和不同层次的仿真。

2. 实验内容

依然做 clk_in 的 2 分频 clk_out，要求输出时钟的相位与上面的 1/2 分频器的输出正好相反。编写测试模块给出仿真波形。

实验代码如下：

****
Freq_divider1.v
****

```
module
Freq_divider1(reset,clk_in,clk_out1,
clk_out2);
  input reset,clk_in;
  output clk_out1,clk_out2;
  reg clk_out1,clk_out2;

  always @(posedge clk_in)
    begin
      if (!reset)
      begin
        clk_out1=1;
        clk_out2=0;
      end
      else
      begin
        clk_out1=~clk_out1;
        clk_out2=~clk_out2;
      end
    end
endmodule
```

****
测试文件 Test2.v
****

```
`timescale 1ns/1ns
module Test1;
  `timescale 1ns/1ns
`define clk_cycle 100
module Test2;
  reg clk_in,reset;
  always  #`clk_cycle clk_in=~clk_in;
  initial
    begin
        clk_in=1;
        reset=0;
      #10 reset=0;
      #10 reset=1;
      #10000  $stop;
    end
    Freq_divider1
Freq_divider1(.reset(reset),.clk_in(
clk_in),.clk_out1(clk_out1),.clk_out
2(clk_out2));
  endmodule
```

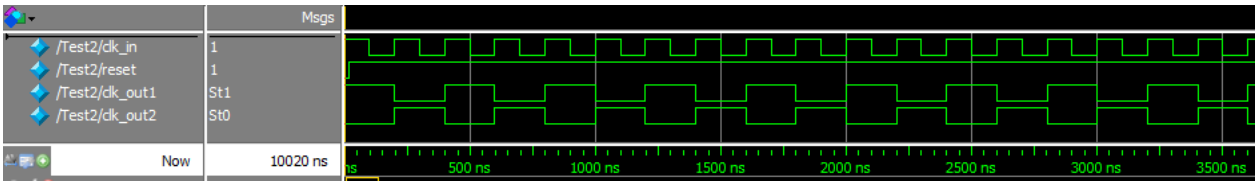
3. 设计思路

书中给出的 2 分频器实例中，在 reset=0 时将 clk_out 初始值置为零，而在 reset=1 后在 clk_in 上升沿对 clk_out 进行求反运算。

若要使相位相反，将 clk_out 初值设为 1 即可，这样在 reset=1 后在 clk_in 上升沿对 clk_out 进行求反运算，即可产生相位与示例相反的 2 分频输出。

4. 实验结果

波形：实验结束后波形如下



可以发现 out1，out2 频率为 clk 频率一半而且反相，符合实验要求。

5. 思考题

如果没有 reset 信号可否控制二分频信号的相位？只用 clk 时钟沿的触发如何直接产生 4、8、16 分频的信号？如何只用 clk 时钟沿的触发直接产生占空比不同的分频时钟：

（1）如果没有 reset 信号，可以通过在测试文件中的 initial 块中设置 clk_in 的初值为 1 来使输出反相。

（2）只用 clk 时钟沿的触发可以通过一个计数器来产生其他分频的时钟，并且通过调节计数器的值来产生不同占空比的分频时钟。这需要分为偶数分频和奇数分频，采用不同方法实现，代码如下：

任意偶数分频

```
module even(  input       clk,rst,
                  input[7:0]N,          // N 分频，N 为偶数
                  output reg       out);
reg[7:0]   cnt;
always @(posedge clk)
    begin
        if(!rst)
            begin
                out<=0;
                cnt<=0;
            end
        else if(cnt==N/2-1)
            begin
                out<=~out;
                cnt<=0;
            end
        else
            cnt<=cnt+1;
    end
    endmodule
```

测试文件 anydiv_tb.v:

```
`timescale 1ns/1ns
module    evendiv_tb;
wire out;
```

```
reg         clk,rst;
reg[7:0]   N;
initial
      begin
            N=50;
            rst=1;
            clk=0;
            #10000    $stop;
      end

always #10 clk=~clk;     // 50MHz clk
even div(clk,rst,N,out);
endmodule
```

任意奇数分频

```
module odd(clk_out,clk_p,clk_n,clk_in,rst);
output clk_out;
output clk_p,clk_n;
input clk_in,rst;
reg [2:0] cnt_p,cnt_n;
reg clk_p,clk_n;
parameter N=5;                    // N 分频，N 为奇数
always @ (posedge clk_in or negedge rst)
begin
      if(!rst)
            cnt_p <= 0;
      else if(cnt_p==N-1)
            cnt_p <=0;
      else
            cnt_p <= cnt_p + 1;
end
always @ (posedge clk_in or negedge rst)
begin
      if(!rst) clk_p <= 0;
      else if(cnt_p==(N-1)/2)
                  clk_p <= !clk_p;
          else if(cnt_p==N-1)
                  clk_p <= !clk_p;
end
always @ (negedge clk_in or negedge rst)
begin
          if(!rst)         cnt_n <= 0;
          else    if(cnt_n==N-1)       cnt_n <=0;
                    else cnt_n <= cnt_n + 1;
```

```
end
always @ (negedge clk_in or negedge rst)
begin
    if(!rst) clk_n <= 0;
    else if(cnt_n==(N-1)/2)
                clk_n <= !clk_n;
        else if(cnt_n==N-1)
                clk_n <= !clk_n;
end
assign clk_out = clk_p | clk_n;
endmodule
```

测试文件：

```
`timescale 1ns/1ns      //50MHz clk_in
module fp_odd_tb;
wire clk_out;
wire clk_p,clk_n;
reg clk_in;
reg rst;
initial
    begin
        rst=1;
        clk_in=0;
        #20 rst=0;
        #120 rst=1;
        #10000 $stop;
    end
always #10 clk_in=~clk_in;
odd odddiv(clk_out,clk_p,clk_n,clk_in,rst);
endmodule
```

产生占空比不同的分频

```
module ratio(  output    out
               input       clk,rst,
               input[7:0]    N);
               parameterhigh=10;        // (high/N)为占空比，此处为 10/50=1/5
               reg[7:0]   cnt;
always @(posedge clk)
    begin
        if(!rst)
            cnt<=0;
        else if(cnt==N/2-1)
            cnt<=0;
        else
```

```
                cnt<=cnt+1;
        end
        assign      out=(cnt<high/2)?1:0;
endmodule
```

测试文件 ratio_tb.v:

```
`timescale 1ns/1ns
module    ratio_tb;
wire out;
reg       clk,rst;
reg[7:0]  N;
initial
        begin
            N=8'd50;
            rst=1;
            clk=0;
            #50 rst=0;
            #150 rst=1;
            #10000    $stop;
        end
always #10 clk=~clk;     // 50MHz clk
ratio div(out,clk,rst,N);
endmodule
```

6．实验总结

本次实验是 Verilog 上机的第二个实验，练习了常用的时序逻辑电路的设计——分频器。其中二分频的设计比较简单，任意分频的设计则不那么简单，也是很经典的问题。我通过上网查资料了解了奇偶分频的不同和任意占空比的产生方法，具体内容已经写到了实验报告中，弄清分频问题的确很有收获。

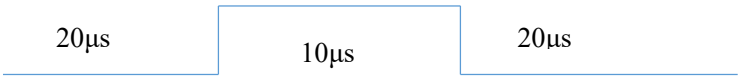
实验三 利用条件语句实现计数分频时序电路

1. 实验目的

（1）掌握条件语句在简单时序模块设计中的使用；
（2）学习在 Verilog 模块中应用计数器；
（3）学习测试模块的编写、综合和不同层次的仿真。

2. 实验内容

利用 10MHz 的时钟设计一个单周期形状的周期波形。单周期形状如下：



代码：

```
************************************
            Freq_divider2.v
************************************
module Freq_divider2(reset,clk,out);
  input reset, clk;
  reg [8:0] flag1;
  reg [8:0] flag2;
  output out;
  reg out;
  always @(posedge clk)
    begin
      if(!reset)
        begin
          flag1=0;
          flag2=0;
          out=0;
        end
      else
        begin
          if (out==1)
            begin
              if (flag1==100)
                begin
                  flag1=1;
                  out=~out;
                end
              else
                begin
                    flag1=flag1+1;
                end
            end
          else
            begin
              if (flag2==400)
                begin
                  flag2=1;
                  out=~out;
                end
              else
                begin
                    flag2=flag2+1;
                end
            end
        end
    end
  endmodule
************************************
          测试文件 Test3.v
************************************
`timescale 1ns/1ns
`define clk_halfcycle 50
module Test3;
  reg clk,reset;
  wire out;
  always  #`clk_halfcycle clk=~clk;
  initial
    begin
        clk=0;
        reset=0;
      #100 reset=1;
```

```
    #1000000  $stop;
end
Freq_divider2
```

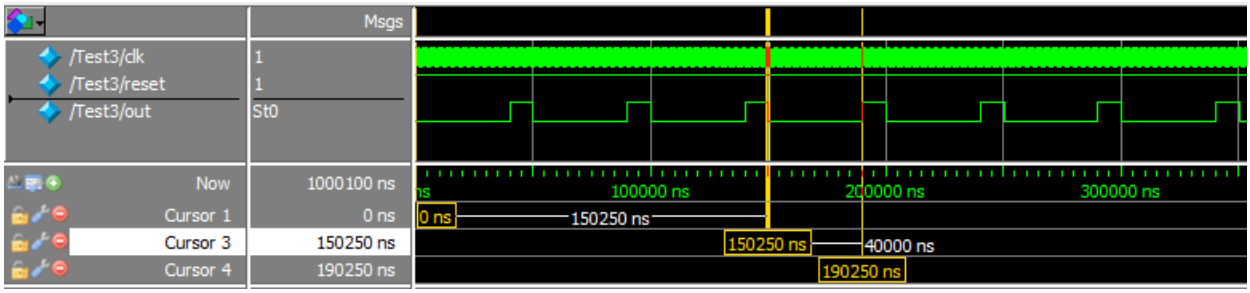
```
Freq_divider2(.reset(reset),.clk(clk
),.out(out));
  endmodule
```

3. 设计思路

书中给出的分频器是将 10M 时钟分频成 500K 输出，以计数的方式进行分频。10M 时钟周期为 0.1us，500K 时钟周期为 2us，因此每计数 20 次令输出翻转，即可形成 500K 时钟输出。而在题目要求中，输出波形为 40us 低电平接 10us 高电平，因此在复位后将输出初始化为低电平，先计数 400 次翻转，再计数 100 次翻转，如此循环，即可生成题目所要求的波形输出。

4. 实验结果

波形：仿真结束后获得的波形如下图所示



由图可以看出，所输出的波形单个周期的形状与题目要求相同。

5. 实验总结

本次实验是 Verilog 上机的第三个实验，一开始我以为是只输出一个周期，思考了好久没想出来方法，后来跟同学交流发现是要把这个作为周期波形输出，于是就简单多了，结合实验二中任意分频任意占空比的方法很容易就能完成。

实验四 阻塞赋值与非阻塞赋值的区别

1. 实验目的

（1）通过实验，掌握阻塞赋值与非阻塞赋值的概念以及区别；
（2）了解非阻塞和阻塞赋值的不同使用场合；
（3）学习测试模块的编写、综合和不同层次的仿真。

2. 实验内容

内容：在 blocking 模块中按如下两种写法，仿真与综合仿真的结果会有什么样的变化，做出仿真波形，分析综合结果。

1）always@(posedge clk)
begin
c=b;
b=a;
end

2) always@(posedge clk) b=a;
always@(posedge clk) c=b;

代码如下：

****
测试文件 Test4.v
****

```
`timescale 1ns/100ps
module Test4;
  wire [3:0] b1,c1,b2,c2;
  reg [3:0] a;
  reg  clk;
  initial
    begin
      clk=0;
      forever #50 clk=~clk;
    end
  initial
  begin
    a=4'h3;
     //$display("_________");
    #100 a=4'h7;
     //$display("_________");
    #100 a=4'hf;
     //$display("_________");
    #100 a=4'ha;
     //$display("_________");
    #100 a=4'h2;
     //$display("_________");
    #100
     //$display("_________");
    $stop;
  end
  blocking1 blocking1(clk,a,b1,c1);
  blocking2 blocking2(clk,a,b2,c2);
endmodule
```

****
blocking1.v & blocking2.v
****

```
module blocking1(clk, a,b,c);
  output [3:0] b,c;
  input [3:0] a;
  input clk;
  reg [3:0] b,c;
  always @(posedge clk)
    begin
      c = b;
      b = a;
 $display("Blocking1: a = %d,
 b = %d, c = %d.",a,b,c);
    end
  endmodule
```

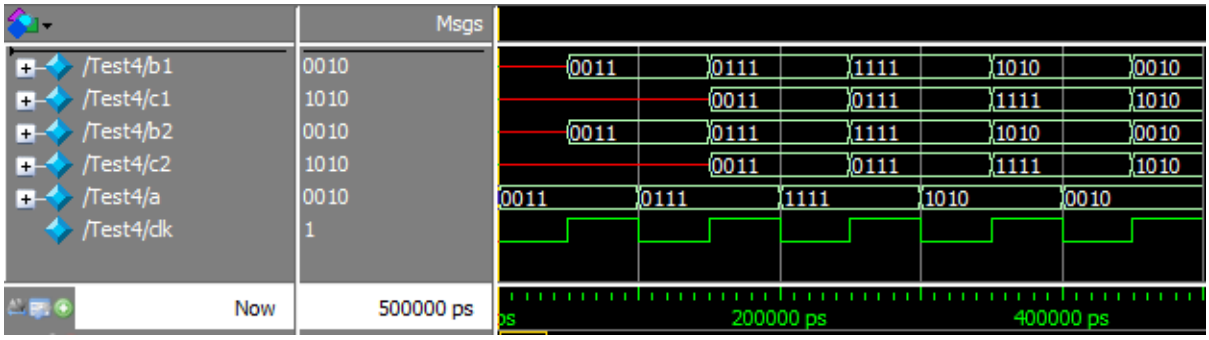
```
module blocking2(clk, a,b,c);
  output [3:0] b,c;
  input [3:0] a;
  input clk;
  reg [3:0] b,c;
  always @(posedge clk)
      c = b;
  always @(posedge clk)
   begin
         b = a;
      $display("Blocking2: a = %d, b
= %d, c = %d.",a,b,c);
   end
  endmodule
```

3. 设计思路

将书中所给 blocking 模块中的赋值部分改为题目所给的两种方式，进行仿真，观察输出波形是否有差异，以了解阻塞赋值法的不同赋值顺序及方式与非阻塞赋值方式相比是不是会产生不同的效果。

4. 实验结果

波形：结束仿真后，所获得的波形如下图所示



由此可见，当采用第一种方式时，第一个时钟上升沿，c 未被赋值，而采用第二种方式时，c 与 b 都在时钟的第一个上升沿被赋值。

5. 实验总结

两种阻塞赋值方式输出结果不同，第一种方式为真正的阻塞赋值，第二种在功能上实际上实现了非阻塞赋值。

在本次实验中，通过改变赋值模块中的赋值顺序和方式，并观察仿真波形，更深刻地认识到了阻塞赋值与非阻塞赋值方法的不同。在 always 模块中，阻塞赋值语句是顺序执行的，前一句没执行完不能执行后面一句，而非阻塞赋值语句是并发执行的。在第一种方式中，b 先赋给 c，a 再赋给 b；而第二种方式中，两个 always 块并行执行，a 赋给 b 与 b 赋给 c 的操作同时执行，因此两种方式的效果不同。

在时序逻辑电路中，通常使用非阻塞赋值，不仅可以防止竞争冒险现象，而且综合出来的及诶过正式我们想要的电路结构。而组合逻辑电路中则多用阻塞赋值，否则可能出现我们意想不到的综合结果。

实验五 用 always 块来实现较复杂的组合逻辑电路

1. 实验目的

（1）掌握用 always 实现较大组合逻辑电路的方法；
（2）进一步了解 assign 与 always 两种组合电路实现方法的区别和注意点；
（3）学习测试模块中随机数的产生和应用；
（4）学习综合不同层次的仿真，并比较结果。

2. 实验内容

运用 always 块设计一个 8 路数据选择器。

要求：每路输入数据与输出数据均为 4 位二进制数，当选择开关或输入数据发生变化时，输出数据也相应的变化。

代码：

****

select.v

****

```
module
select(a1,a2,a3,a4,a5,a6,a7,a8,out,
flag);
  input [3:0]
a1,a2,a3,a4,a5,a6,a7,a8;
  input [2:0] flag;
  output [3:0] out;
  reg [3:0] out;
    always @(flag)
    begin
      case(flag)
        3'b000: out=a1;
        3'b001: out=a2;
        3'b010: out=a3;
        3'b011: out=a4;
        3'b100: out=a5;
        3'b101: out=a6;
        3'b110: out=a7;
        3'b111: out=a8;
        default: out=4'bx;
      endcase
    end
endmodule
```

****

测试文件 Test5.v

****

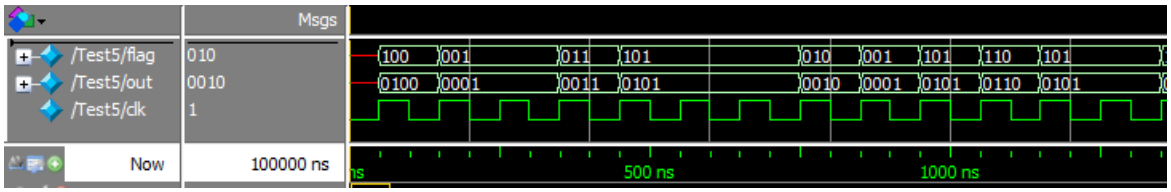
```
timescale 1ns/1ns
module Test5;
  reg[2:0] flag;
  wire[3:0] out;
  reg clk;
  initial
    clk=0;
  always #50 clk=~clk;
  always #100000  $stop;
  parameter
     a1=4'b0000,
     a2=4'b0001,
     a3=4'b0010,
     a4=4'b0011,
     a5=4'b0100,
     a6=4'b0101,
     a7=4'b0110,
     a8=4'b0111;
  always @(posedge clk)
    begin
      flag={$random}%8;
    end
 select select
(a1,a2,a3,a4,a5,a6,a7,a8,out,flag);
endmodule
```

3. 设计思路

将 flag 作为数据选择开关，当其取值为 0、1、2、3、4、5、6、7 时，分别使输出 out 等于对应的 a0、a1、a2、a4、a5、a6、a7。在测试程序中，利用系统任务$random，每一个时钟上升沿，产生随机数，生成四位二进制数输入到 flag，以检验数据选择器功能。

4. 实验结果

波形：仿真结束后，得到的波形如下图所示



由上图可以看出：每路输入数据与输出数据均为对应的二进制数，当选择开关或输入数据发生变化时，输出数据也相应的变化。符合题目要求。

5. 实验总结

本次实验是 Verilog 上机的第五个实验，是一个较为复杂的组合逻辑电路。这次我学习了数据选择器模块的编写，也进一步熟悉了 always 与 case 的结合使用以及系统任务$random 结合取余运算生成随机数的办法。

之前我曾设想简化代码，将输入像 C 语言那样做成数组形式，后来编译发现不通过，原因是端口不能为数组，这也让我增长了设计经验。

实验六 在 Verilog HDL 中使用函数

1. 实验目的

（1）了解函数的定义和在模块设计中的使用；
（2）了解函数的综合性问题；
（3）了解许多综合器不能综合复杂的算术运算

2. 实验内容

设计一个带控制端的逻辑运算电路，分别完成正整数的平方、立方和最大数为 5 的阶乘运算，要求可综合。编写测试模块。

代码：

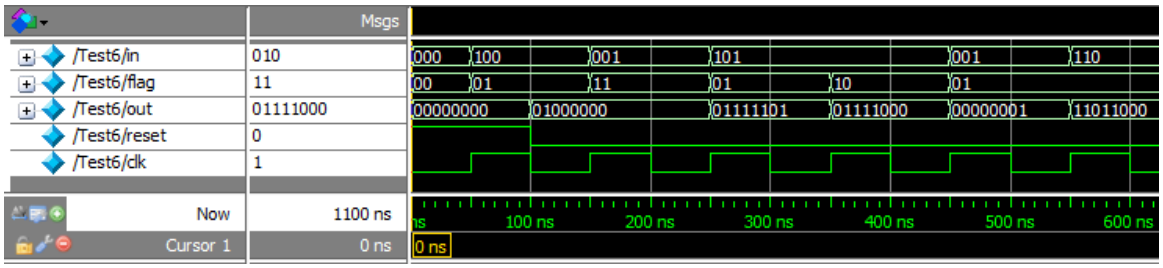
```
module
calculator(in,out,flag,reset);
  input [2:0] in;
  output [7:0] out;
  input [1:0] flag;
  input reset;
  reg [7:0] out;
  always @(*)
  if(!reset)
     begin
     case (flag)
       2'b00: out=square(in);
       2'b01: out=cube(in);
       2'b10: out=factorial(in);
       default: out=out;
     endcase
    end
  else  out=0;
    function [7:0] square;
      input [2:0] a;
      assign square=a*a;
    endfunction
    function [7:0] cube;
      input [2:0] b;
      assign cube=b*b*b;
    endfunction
    function [7:0] factorial;
      input [2:0] c;
      reg [2:0] index;
      begin
        factorial=c?1:0;
        for(index
=2;index<=c;index=index+1)
          factorial=index*factorial;
      end
    endfunction
endmodule
`timescale 1ns/1ns
module Test6;
  reg [2:0] in;
  reg [1:0] flag;
  wire[7:0] out;
  reg reset;
  reg clk;
  initial
    begin
      in=0;
      flag=0;
      clk=0;
      reset=1;
      #100 reset=0;
      #1000   $stop;
    end
  always #50 clk=~clk;
  always @ (posedge clk)
    begin
      in<={$random}%8;
      flag<={$random}%4;
    end
    calculator
calculator(in,out,flag,reset);
endmodule
```

3. 设计思路

1. 首先这个逻辑运算电路应当有 3 个对外的端口，分别是：一个数据输入端口；一个 2 位的功能控制端口；一个结果的输出端口。
2. 由于阶乘的最大数为 5，因此输入最大数据为 5，输出最大数据为 5！=120 设输入数据为 3 位，输出数据为 8 位。
3. 通过条件语句来选择不同的函数功能，每个函数功能分别编写来提高程序的可读性和可移植性。
4. 测试模块产生随机的输入数据以及控制信号，因此利用系统任务$random 来实现。通过一个时钟，每一个上升沿产生新的随机数，以检测逻辑运算电路在不同输入下的三种运算功能。

4. 实验结果

波形：仿真后得到的波形如下图所示：



由图可见，本实验完成了题目所要求的平方、立方、阶乘功能，符合实验要求。

5. 实验总结

本次实验是 Verilog 上机的第六个实验，是一个较为复杂的组合逻辑电路。其中运用了函数来实现一些模块化的功能，增强了程序的可读性和可移植性。在今后进行 Verilog 程序设计的时候，也应当多用这样的写法。

6. Verilog 可综合的总结

（1）所有综合工具都支持的结构：always，assign，begin，end，case，wire，tri，aupply0，supply1，reg，integer，default，for，function，and，nand，or，nor，xor，xnor，buf，not，bufif0，bufif1，notif0，notif1，if，inout，input，instantitation，module，negedge，posedge，operators，output，parameter

（2）所有综合工具都不支持的结构：time，defparam，$finish，fork，join，initial，delays，UDP，wait

（3）有些工具支持有些工具不支持的结构：casex，casez，wand，triand，wor，trior，real，disable，forever，arrays，memories，repeat，task，while

实验七 在 Verilog HDL 中使用任务

1. 实验目的

（1）掌握任务在 verilog 模块设计中的应用；

（2）学会在电平敏感列表的 always 中使用拼接操作、任务和阻塞赋值等语句，并生成复杂组合逻辑的高级方法。

2. 实验内容

用两种不同的方法设计一个功能相同的模块，该模块能完成四个 8 位 2 进制输入数据的冒泡排序。第一种，模仿上面的例子用纯组合逻辑实现，第二种假设 8 位数据是按照时钟节拍串行输入的，要求用时间触发任务的执行法，每个时钟周期完成一次数据交换的操作。比较两种方法的运行速度和所消耗资源的不同。

代码：实验所使用的代码如下

```
**********************************
            sort1.v & sort2.v
**********************************
module
sort1(a,b,c,d,ra,rb,rc,rd,reset);
  input[7:0] a,b,c,d;
  output[7:0] ra,rb,rc,rd;
  reg[7:0] ra,rb,rc,rd;
  reg[7:0] va,vb,vc,vd;
  input reset;
  always @ (a or b or c or d)
  if(!reset)
    begin
      {va,vb,vc,vd}={a,b,c,d};
      sort(va,vc);
      sort(vb,vd);
      sort(va,vb);
      sort(vc,vd);
      sort(vb,vc);
      {ra,rb,rc,rd}={va,vb,vc,vd};
    end
    else
    begin
    ra=0;rb=0;rc=0;rd=0;
    end
    task sort;
      inout[7:0] x,y;
      reg[7:0] tmp;
      if(x>y)
        begin
          tmp=x;
          x=y;
          y=tmp;
        end
    endtask
  endmodule

module
sort2(clk,reset,ra,rb,rc,rd,a);
output[7:0] ra,rb,rc,rd;
input[7:0] a;
input clk,reset;
reg[7:0] ra,rb,rc,rd;
reg[7:0] va,vb,vc,vd;
always @ (posedge clk)
 begin
    if(reset)
      begin
        va<=0;vb<=0;vc<=0;vd<=0;
      end
    else
      va<=a;
 end
always @ (posedge clk)
 begin
      sort(va,vc);
      sort(vb,vd);
      sort(va,vb);
      sort(vc,vd);
      sort(vb,vc);
```

```
    {ra,rb,rc,rd}={va,vb,vc,vd};
 end
task sort;
inout[7:0] x,y;
reg[7:0] tmp;
   if(x>y)
     begin
       tmp=x;
       x=y;
       y=tmp;
     end
endtask

endmodule
**********************************
        测试文件 Test71.v & Test72.v
**********************************
`timescale 1ns/1ns
module Test71;
  reg[7:0] a,b,c,d;
  wire[7:0] ra,rb,rc,rd;
  reg reset;

  initial
     begin
          a=0;b=0;c=0;d=0;
          reset=1;
          #100  reset=0;
     repeat(50)
     begin
       #100      a={$random}%256;
                 b={$random}%256;
                 c={$random}%256;
                 d={$random}%256;
     end
     #100 $stop;
   end
     sort1
sort1(.a(a),.b(b),.c(c),.d(d),.ra(ra
),.rb(rb),.rc(rc),.rd(rd),.reset(res
et));
  endmodule

`timescale 1ns/1ns
`define clk_cycle 50

module Test72;
  reg [7:0] a;
  wire [7:0] ra,rb,rc,rd;
  reg clk,reset;
  always #`clk_cycle clk= ~clk;
  initial
    begin
      a=0;
      clk=1;
      reset =1;
      #100  reset=0;
      repeat(50)
       begin
         #100 a={$random}%256;
       end
      #100  $stop;
    end
    sort2
sort2(clk,reset,ra,rb,rc,rd,a);
  endmodule
```

3. 设计思路

（1）第一种方法需要 8 个对外端口，分别为 4 个 8 位的输入端口和 4 个 8 位的输出端口。第二种方法需要 7 个对外端口，分别是 1 个 8 位数据输入端口，1 个时钟，1 个复位端口和 4 个 8 位输出端口。

（2）两种方法中，都要把输入的 4 个数据储存在内部的四个存储器中，对这四个存储器的数据进行冒泡排序，再将这四个储存器中的已经排序完成的数据输出。不同之处在于第一种方法直接将输入的数据储存起来。而第二种方法则每进来一个数据进行一次储存和一次排序。

（3）冒泡排序的算法可以通过比较和数据对换位置实现。其中数据对换位置的功能多次重复使用，可以由一个任务来完成。

4. 实验结果

波形：仿真结束后得到的波形如下图所示：

从第一组波形中可以看出，每次输入一组 a，b，c，d，都会输出一组 ra，rb，rc，rd。并且有 ra<rb<rc<rd。符合题目要求。

从第二组波形中可以看出，每次比较完成需要四个时钟周期来把四个数据全部输入。在每个时钟周期中都进行了一次比较，每次都把最大的数移到 rd 的位置上。在四个时钟周期完成后，有 ra<rb<rc<rd。符合题目要求。

5. 实验总结

本次实验是 Verilog 上机的第七个实验，分别完成了对并行输入数据和串行输入数据的冒泡排序以及并行输出。从效率上来说方法一的效率比较高，因为只需要进行一次比较，需要时间短。但是方法二完成了串行转并行的数据变换工作，比方法一的功能更强，并且由于采用了串行输入，其接口比第一种要少很多，节省了硬件资源。

实验八 利用有限状态机进行时序逻辑的设计

1. 实验目的

（1）掌握利用有限状态机实现一般时序逻辑分析的方法；
（2）掌握用 Verilog 编写可综合的有限状态机的标准模板；
（3）掌握用 Verilog 编写状态机模块的测试文件的一般方法

2. 实验内容

设计一个串行数据检测器。

要求：连续 4 个或以上为 1 时输出为 1，其他输入情况下为 0.编写测试模块对设计的模块进行各种层次的仿真，并观察波形，编写实验报告。

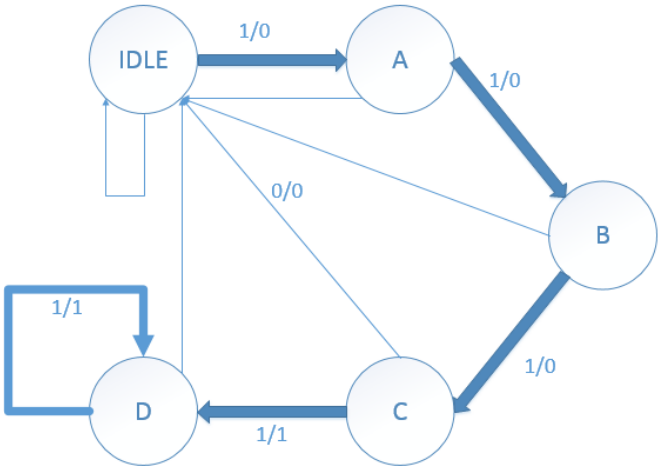
代码：

```
**********************************
                Detector.v
**********************************
module detector(in,out,clk,reset);
  input in,clk,reset;
  output out;
  reg[2:0] state;
  wire out;
  parameter IDLE='d0, A='d1, B='d2,
C='d3, D='d4;
  always @ (posedge clk)
    if(!reset)
       begin
         state=IDLE;
       end
    else
       casex(state)
         IDLE: if(in==1)
                  begin
                    state=A;
                  end
         A: if(in==1)
              begin
                state=B;
              end
            else
              begin
               state=IDLE;
              end
         B: if(in==1)
              begin
                state=C;
              end
            else
              begin
               state=IDLE;
              end
         C: if(in==1)
              begin
                state=D;
              end
            else
              begin
               state=IDLE;
              end
         D: if(in==1)
              begin
                state=D;
              end
            else
              begin
               state=IDLE;
              end
          default:state=IDLE;
       endcase

assign out=(state==D)? 1:0;
endmodule
```

```
**********************************
        测试文件 Test8.v
**********************************
`timescale 1ns/1ns
module Test8;
  reg clk,reset,in;
  wire out;
  always #10 clk= ~clk;
  always @ (posedge clk)
     in = {$random}%2;
  initial
      begin
        clk=0;
        reset=1;
        #10 reset=0;
        #100 reset=1;
        #1000 $stop;
      end
detector
detector(in,out,clk,reset);

      endmodule
```

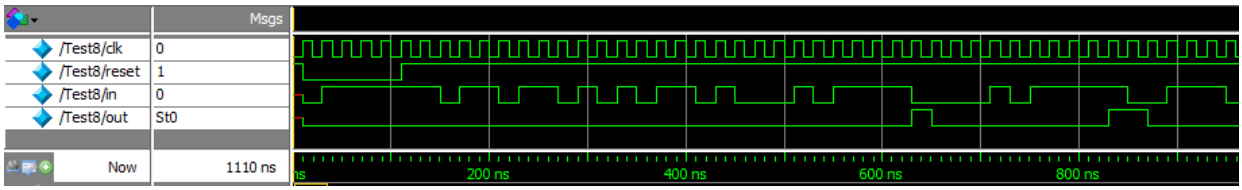
3. 设计思路

本题可以用一个有限状态机，每连续出现一个 1 状态跳转一次。当连续 4 个或 4 个以上 1 时，锁定在某个状态，并且输出 1，在这之后若出现 1，继续锁定在此状态，若出现 0，回到最初没有一个有效输入的状态。其状态转移图如下：



4. 实验结果

仿真得到的波形如图所示：



从波形中可以看出，当出现 4 个或 4 个以上连 1 的时候，输出 1，否则输出为 0，符合要求。

5. 实验总结

序列检测器是经典的状态机设计题目，掌握了序列检测器有助于更好地理解状态机的设计方法。更熟练地编写状态机，为以后设计大型系统打下基础。通过这次实验我想我基本掌握了小状态机的设计方法，感觉写 Verilog 程序熟练了很多。

实验九 利用状态机实现比较复杂的接口设计

1. 实验目的

（1）学习运用由状态机控制的逻辑开关，设计出一个比较复杂的接口逻辑；
（2）在复杂设计中使用任务（task）结构，以提高程序的可读性；
（3）加深对可综合模块的认识。

2. 实验内容

参考第二部分 15 章的例 15.2，编写可综合模块把符合 I²C 串行总线要求的地址和数据信号，在只有 sda 和 scl 两个信号的前提下转换为并行的数据和地址信号。

代码如下：

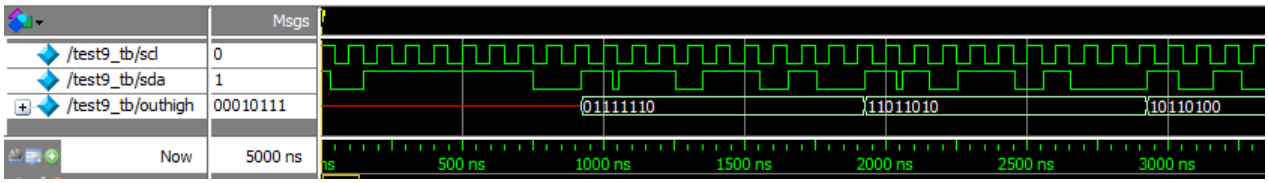
```
**********************************
            test9_out16hi.v
**********************************
module out16hi(scl,sda,outhigh);
  input scl,sda;
  output[15:0]outhigh;

  reg[8:0]mstate;
  reg[7:0]pdata,pdatabuf;
  reg[7:0]outhigh;
  reg StartFlag,EndFlag;

  always@(negedge sda)
    begin
      if(scl)
        begin
          StartFlag<=1;
          EndFlag<=0;
          $monitor($time,
"(1)startflag=1
endflag=?scl=,sda=",StartFlag,EndFla
g,scl,sda);
        end
      else if(EndFlag)
        begin
        StartFlag<=0;

$monitor($time ,"(2)startflag=
EndFlag=1
scl=,sda=",StartFlag,EndFlag,scl,sda
);
      end
    end
  always @(posedge sda)
    if(scl)
      begin
        EndFlag<=1;
        StartFlag<=0;

       $monitor($time,
"(3)startflag= EndFlag=1
scl=,sda=",StartFlag,EndFlag,scl,sda
);
        pdatabuf<=pdata;
      end
    else
      begin
        EndFlag<=0;
         $monitor($time,
"(4)startflag=,EndFlag=0
scl=,sda=",StartFlag,EndFlag,scl,sda
);
      end
  parameter ready=9'b0_0000_0000,
          sbit0=9'b0_0000_0001,
          sbit1=9'b0_0000_0010,
          sbit2=9'b0_0000_0100,
          sbit3=9'b0_0000_1000,
          sbit4=9'b0_0001_0000,
          sbit5=9'b0_0010_0000,
          sbit6=9'b0_0100_0000,
          sbit7=9'b0_1000_0000,
          sbit8=9'b1_0000_0000;
```

```
    always@(pdatabuf)
        begin
        outhigh<=pdatabuf;

      end

      always@(posedge scl)
        if(StartFlag)
        case(mstate)
          sbit0:begin
              mstate<=sbit1;
              pdata[7]<=sda;
              $display("I am in
sdabit0");

$monitor($time, ,"sda=",sda,StartFla
g,EndFlag);

$monitor($time, ,"pdata[7]=",pdata[7
]);
            end
            sbit1:begin
              mstate<=sbit2;
              pdata[6]<=sda;
              $display("I am in
sdabit1");

$monitor($time, ,"sda=",sda,StartFla
g,EndFlag);
            end
            sbit2:begin
              mstate<=sbit3;
              pdata[5]<=sda;
              $dispaly("I am in
sdabit2");

$monitor($time, ,"sda=",sda,StartFla
g,EndFlag);
            end
            sbit3:begin
              mstate<=sbit4;
              pdata[4]<=sda;
              $display("I am in
sdabit3");

$monitor($time, ,"sda=",sda,StartFla
g,EndFlag);
            end
            sbit4:begin
              mstate<=sbit5;
               pdata[3]<=sda;
              $display("I am in
sdabit4");

$monitor($time, ,"sda=",sda,StartFla
g,EndFlag);
            end
            sbit5:begin
              mstate<=sbit6;
               pdata[2]<=sda;
              $display("I am in
sdabit5");

$monitor($time, ,"sda=",sda);
            end
            sbit6:begin
              mstate<=sbit7;
               pdata[1]<=sda;
              $display("I am in
sdabit6");

$monitor($time, ,"sda=",sda,StartFla
g,EndFlag);
            end
            sbit7:begin
              mstate<=sbit8;
             pdata[0]<=sda;
              $display("I am in
sdabit7");

$monitor($time, ,"sda=",sda,StartFla
g,EndFlag);
           end
           sbit8:begin
             mstate<=sbit0;
              $display("I am in
sdatop");
```

```
$monitor($time, ,"sda=",sda,StartFlag,EndFlag);
          end
          default: mstate<=sbit0;
         endcase
       else mstate<=sbit0;
       endmodule

************************************
          测试文件 test9_tb.v
************************************
`timescale 1ns/1ns
`define halfperiod 50
module test9_tb;

  reg scl,sda;
  wire[7:0] outhigh;

  initial
    begin
      scl=1;
    end

    initial
    begin
       #(`halfperiod*100) $stop;
     end
    always#(`halfperiod) scl=~scl;
    initial
    forever
     begin
       #2 sda=1;
       #30 sda=0;
       #20 sda={$random}%2;  //1
       #50 sda=sda;
       #50 sda={$random}%2;   //2
       #50 sda=sda;
       #50 sda={$random}%2;   //3
       #50 sda=sda;
       #50 sda={$random}%2;   //4
       #50 sda=sda;
       #50 sda={$random}%2;  //5
       #50 sda=sda;
       #50 sda={$random}%2;   //6
       #50 sda=sda;
       #50 sda={$random}%2;  //7
       #50 sda=sda;
       #50 sda={$random}%2; //8
       #50 sda=sda;
       #50 sda=0;
       #70 sda=1; //
       #80 sda=1; //
    end

    out16hi
m2(.scl(scl),.sda(sda),.outhigh(outhigh));
    endmodule
```

3. 仿真波形

实验结束后仿真波形如下图所示：



由上图可以看出，scl 为高电平时，sda 由高变低，串行数据流开始；scl 为高电平时，sda 由低变高，串行数据流结束，并把接收数据并行输出到 outhigh，符合设计要求。

4. 实验总结

通过这次实验，我设计了简单的串并转换通信协议，进一步加深了状态机的理解与应用。

实验十 通过模块实例调用实现大型系统的设计

1. 实验目的

（1） 掌握和学习大型系统的嵌套和模块化的实例；
（2） 了解大型系统设计的得层次化、结构化解决办法的技术基础；
（3） 学习数据总线在模块实际中的应用和控制，掌握复杂接口模块设计的技术基础；
（4） 学习和掌握用工程概念来编写较为完整的测试模块，做到接近真实的完整测试。

2. 实验内容

通过申请CPU中断后取得数据进行处理,并把处理的结果通过同一条数据总线返回CPU的模块.具体的时序参数和 CPU 中断的响应时间和读写时序完全相同。

代码：

代码：实验所使用的代码如下：

```
**********************************
              ex10_P_S.v
**********************************
`define YES 1
`define NO 0

module
ex10_P_S(Dbit_out,link_S_out,data,nGet_AD_data,clk);

    input clk;
    input nGet_AD_data;
    input [7:0] data;
    output Dbit_out;
    output link_S_out;

    reg[3:0] state;
    reg[7:0] data_buf;
    reg link_S_out;
    reg d_buf;
    reg finish_flag;

    assign
Dbit_out=(link_S_out)?d_buf:0;

    always@(posedge clk or negedge
nGet_AD_data)
       if(!nGet_AD_data)
          begin
            finish_flag<=0;
            state<=9;
            link_S_out<=`NO;
            d_buf<=0;
            data_buf<=0;
          end
       else
          case(state)
9:  begin
          data_buf<=data;
          state<=10;
          link_S_out<=`NO;
        end
10: begin
          data_buf<=data;
          state<=0;
          link_S_out<=`NO;
        end
0:  begin
          link_S_out<=`YES;
          d_buf<=data_buf[7];
          state<=1;
        end
1:  begin
          d_buf<=data_buf[6];
          state<=2;
        end
2:  begin
          d_buf<=data_buf[5];
          state<=3;
        end
```

```
        3:  begin
            d_buf<=data_buf[4];
            state<=4;
          end
        4:  begin
            d_buf<=data_buf[3];
            state<=5;
          end
        5:  begin
            d_buf<=data_buf[2];
            state<=6;
          end
        6:  begin
            d_buf<=data_buf[1];
            state<=7;
          end
        7:  begin
            d_buf<=data_buf[0];
            state<=8;
          end
        8:  begin
            link_S_out<=`NO;
            state<=4'b1111;
            finish_flag<=1;
          end
        default:  begin
                  link_S_out<=`NO;
                  state<=4'b1111;
                end
        endcase

endmodule

************************************
            ex10_process.v
************************************
module
ex10_process(Dbit_out,link_S_out,Dbi
t_in);

    input Dbit_out;
    inout link_S_out;
    output Dbit_in;
    assign  Dbit_in=~Dbit_out;

endmodule

************************************
            ex10_S_P.v
************************************

`define YES 1
`define NO 0

module
ex10_S_P(data,Dbit_in,Dbit_ena,clk);

    output[7:0] data;
    input Dbit_in,Dbit_ena,clk;

    reg[7:0] data_buf;
    reg[3:0] state;
    reg p_out_link;

    assign
data=(p_out_link==`YES)?data_buf:8'b
z;

    always@(negedge clk)
        if(Dbit_ena)
            case(state)
                0:  begin
                    p_out_link<=`NO;

data_buf[7]<=Dbit_in;
                    state<=1;
                  end
                1:  begin

data_buf[6]<=Dbit_in;
                    state<=2;
                  end
                2:  begin

data_buf[5]<=Dbit_in;
                    state<=3;
                  end
```

```
            3:  begin

data_buf[4]<=Dbit_in;
                state<=4;
              end
            4:  begin

data_buf[3]<=Dbit_in;
                state<=5;
              end
            5:  begin

data_buf[2]<=Dbit_in;
                state<=6;
              end
            6:  begin

data_buf[1]<=Dbit_in;
                state<=7;
              end
            7:  begin

data_buf[0]<=Dbit_in;
                state<=8;
              end
            8:  begin
                p_out_link<=`YES;
                state<=4'b1111;
              end
            default: state<=0;
        endcase
      else
        begin
            p_out_link<=`YES;
            state<=0;
        end

endmodule
```

****

sys.v

****

```
module
sys(databus,use_p_in_bus,Dbit_out,Dbit_ena,nGet_AD_data,Dbit_in,clk);

    input nGet_AD_data;
    input use_p_in_bus;
    input clk;
    inout [7:0] databus;
    output Dbit_out;
    output Dbit_ena;
    output Dbit_in;

    wire clk;
    wire nGet_AD_data;
    wire Dbit_out;
    wire Dbit_ena;
    wire Dbit_in;
    wire[7:0] data;

    assign
databus=(!use_p_in_bus)?data:8'bzzzz_zzzz;

    ex10_P_S
m0( .Dbit_out(Dbit_out),.link_S_out(Dbit_ena),.data(databus),
                        .nGet_AD_data(nGet_AD_data),
                        .clk(clk));
    ex10_process
m(  .Dbit_out(Dbit_out),.link_S_out(Dbit_ena),.Dbit_in(Dbit_in));
    ex10_S_P
m1( .data(data),.Dbit_in(Dbit_in),.Dbit_ena(Dbit_ena),.clk(clk));

endmodule
```

****

test.v

****

```
`timescale 1ns/1ns
module ex10_tb;

    reg clk;
```

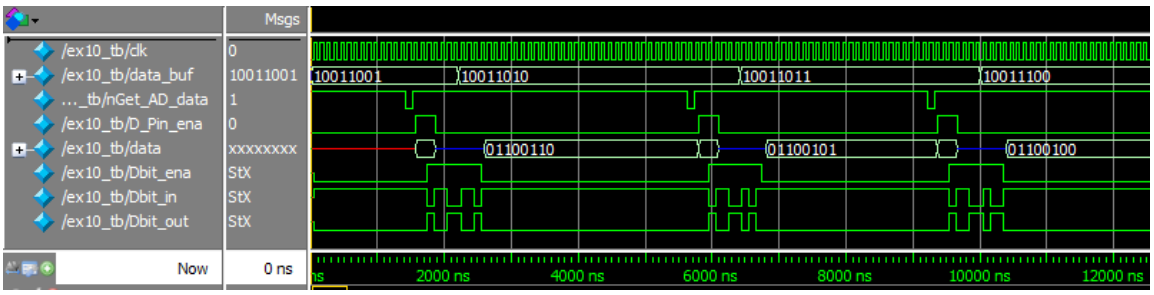
```
    reg[7:0] data_buf;
    reg nGet_AD_data,D_Pin_ena;
    wire[7:0] data;
    wire Dbit_ena,Dbit_in;

    assign
data=(D_Pin_ena)?data_buf:8'bz;

    initial
        begin
            clk=0;
            nGet_AD_data=1;
            data_buf=8'b1001_1001;
            D_Pin_ena=0;

        end

    initial
        begin
        repeat(10)
            begin
              #(100*14+{$random}%23)
nGet_AD_data=0;
              #(112+{$random}%12)
nGet_AD_data=1;
              #({$random}%50)
D_Pin_ena=1;

#(100*3+{$random}%5)D_Pin_ena=0;
              #333
data_buf=data_buf+1;

#(100*11+{$random}%1000);
            end
        end

    always #(50+$random%2) clk=~clk;

    sys
ms( .databus(data),.use_p_in_bus(D_P
in_ena),.Dbit_out(Dbit_out),.Dbit_en
a(Dbit_ena),.nGet_AD_data(nGet_AD_da
ta),.Dbit_in(Dbit_in),.clk(clk));

endmodule
```

3. 设计思路

把任务分为分为四个模块：
模块 ex10_P_S：在时钟节拍下实现并行字节数据向字节位流的转变，并产生相应字节位流的有效信号。
模块 ex10_S_P：实现字节位流向并行数据的转换
模块 ex10_process：模拟 CPU 对总线数据的运算，实现输入字节位流的处理，并输出字节位流，本实验中为取反。
模块 4：把三个独立的逻辑模块 ex10_P_S、ex10_process、ex10_S_P 合并到一个可综合的模块里，使两者的并行总线统一，配合有关信号，进行分时的输入输出。

4. 实验结果

仿真得到的波形时序如下图所示：



clk 为时钟信号，data_buf 是待处理的并行数据，nGet_AD_Data 是取并行数据控制信号线，D_Pin_ena 是并行总线用于输入数据的控制信号， Dbit_ena 是字节位流使能信号，Dbit_out 是未处理的串行字节位流，Dbit_in 是 CPU 取反处理后的数据，Data 是最终的并行输出。

由图可见，时序满足了通过一条数据总线 CPU 进行数据输入、处理、输出的设计要求。

5．实验总结

本次实验以 CPU 总线的数据转换为例，为我积累了设计可靠接收和发送接口的经验，受益匪浅。

实验十一 简单卷积器的设计

1. 实验目的

（1）学习和掌握高速计算逻辑状态机的控制基本方法；
（2）了解计算逻辑与存储器和 AD 模块的接口设计技术基础；
（3）进一步掌握数据总线在模块设计中的应用和控制。
（4）熟悉用工程概念来编写较完整的测试模块，做到接近真实的完整测试。

2. 实验内容

模范示范的 AD 虚拟模块，编写 AD 变换器全功能的虚拟模块，编写完整的静态 RAM 虚拟模块，根据两种器件的时序设计符合工程要求的卷积器。

代码如下：

```
************************************
                con3ad.v
************************************
`timescale 1ns/100ps

module  con3ad( indata, outdata,
address ,CLK, reset, start,
nconvst1, nconvst2, nconvst3,
             nbusy1, nbusy2, nbusy3,
wr, enout1, enout2);

input   indata,
        CLK,
        reset,
        start,
        nbusy1,
        nbusy2,
        nbusy3;
output  outdata,
        address,
        nconvst1,
        nconvst2,
        nconvst3,
        wr,
        enout1,
        enout2;
wire[7:0]   indata;
wire        CLK,
           reset,
           start,
           nbusy1,
           nbusy2,
           nbusy3;
reg[7:0]    outdata;
reg[10:0]   address;
reg     nconvst1,
        nconvst2,
        nconvst3,
        wr,
        enout1,
        enout2;
reg[6:0]    state;
reg[5:0]    i;
reg[1:0]    j;
reg[11:0]   counter;
reg[23:0]   line;
reg[15:0]   result;
reg         high;
reg         k;
reg         EOC1, EOC2, EOC3;

parameter   h1 = 1 ,h2 = 2,h3 = 3;
parameter   IDLE = 7'b0000001,
READ_PRE = 7'b0000010,
           READ = 7'b0000100,  CALCU
= 7'b0001000,
           WRREADY = 7'b0010000, WR =
7'b0100000,
           WREND = 7'b1000000;

always  @(posedge CLK)
    begin
```

```
        if(! reset)
           begin
              state   <= IDLE;
              counter <= 12'b0;
              wr      <= 1;
              enout1  <= 11;
              enout2  <= 11;
              outdata <= 8'bz;
              address <= 11'bz;
              line    <= 24'b0;
              result  <= 16'b0;
              high    <= 0;
           end // end of "if
        else
           begin
              case(state)
              IDLE:   if( start)
                        begin
                           counter
<= 0;
                           state   <=
READ_PRE;
                        end
                      else
                        state <=
IDLE;
              READ_PRE:   if(EOC1 ||
EOC2 || EOC3)
                           state <=
READ;
                        else
                           state <=
READ_PRE;
              READ:   begin
                        high    <= 0;
                        enout2  <= 1;
                        wr <= 1;
                        if(j==1)
                           begin

if(EOC1)

begin

line    <= {line[15:0],indata};

state   <= CALCU;
                                    end
                                 else
state <= READ_PRE;
                              end
                           else if(j ==
2 && counter!=0)
                              begin

if(EOC2)

begin

line    <= {line[15:0], indata};

state   <= CALCU;
                                    end
                                 else
state   <= READ_PRE;
                              end
                       else if(j == 3 && counter != 0)
                              begin

if(EOC3)

begin

line    <= {line[15:0], indata};

state   <= CALCU;
                                    end
                                 else
state   <= READ_PRE;
                              end
                           else    state
<= READ;
                     end
              CALCU:  begin
                           result  <=
line[7:0] * h1 + line[15:8] * h2 +
line[23:16] * h3;
                           state   <=
```

```
WRREADY;
                    end
                WRREADY:begin
                        wr <= 1;
                        address <=
counter;
                        if(k == 1)
state <= WR;
                        else
state <= WRREADY;
                    end
                WR:    begin
                        if(! high)
enout1 <= 0;
                        else
enout2 <= 0;
                        wr  <= 0 ;
                        if( ! high)
outdata <= result[7:0];
                        else
outdata <= result[15:8];
                        if(k==1)
state   <= WREND;
                        else
state   <= WR;
                    end
                WREND:  begin
                        wr      <= 1;
                        enout1  <= 1;
                        enout2  <= 1;
                        if(k==1)
                            if(!
high)
                                begin

high <= 1;

state <= WRREADY;
                                end
                            else
                                begin

counter <= counter+1;
if(counter[11] && counter[0])

state <= IDLE;

else    state <= READ_PRE;
                                    end
                            else    state
<= WREND;
                        end
                default:    state <=
IDLE;
                endcase // end of the
case
            end // end of "else"
end     // end of "always"

always  @(posedge CLK)
    begin
        if(! reset) i<=0;
        else
            begin
                if(i== 44)  i <= 0;
                else        i <= i+1;
            end
    end

always  @(posedge CLK)
    begin
        if(i==4)    j <= 2;
        else if(i==10)  j <= 0;
        else if(i==19)  j <= 3;
        else if(i==25)  j <= 0;
        else if(i==34)  j <= 1;
        else if(i==40)  j <= 0;
    end

always  @(posedge CLK)
    begin
        if(state == WRREADY ||
state== WR || state== WREND)
            if(k==1)    k <= 0;
            else        k <= 1;
        else k <= 0;
    end
```

```
always  @(posedge CLK)
    begin
        if(!reset) nconvst1 <= 1;
        else if(i== 0)  nconvst1 <= 0;
        else if(i== 3)  nconvst1 <= 1;
    end

always  @(posedge CLK)
    begin
        if(!reset) nconvst2 <= 1;
        else if(i== 15) nconvst2 <=
0;
        else if(i== 18) nconvst2 <=
1;
    end

always  @(posedge CLK)
    begin
        if(!reset)  nconvst3 <= 1;
        else if(i==30)  nconvst3 <= 0;
        else if(i==33)  nconvst3 <= 1;
    end

always  @(negedge CLK)
    begin
        EOC1 <= nbusy1;
        EOC2 <= nbusy2;
        EOC3 <= nbusy3;
    end

endmodule
```

****

测试文件 tb_con3ad.v

****

```
`timescale 1 ns/100 ps

module testcon3ad;
wire    wr,
        enin,
        enout1,
        enout2;
wire[10:0]  address;
reg clk,
    reset,
    start,
    rd;
wire    nbusy1,
        nbusy2,
        nbusy3;

wire    nconvst1,
        nconvst2,
        nconvst3;
wire[7:0]   indata;
wire[7:0]   outdata;

parameter HALF_PERIOD = 15;

initial
    begin
        clk = 1;
        forever #HALF_PERIOD
clk=~clk;
    end

initial
    begin
        reset = 1;
        #110    reset = 0;
        #140    reset = 1;
    end

initial
    begin
        start = 0;
        rd = 1;
        #420    start = 1;
        #120    start = 0;
        #107600 start = 1;
        #150    start = 0;
    end

assign  enin = 1;

con3ad
con3ad( .indata(indata),    .outdata(
```

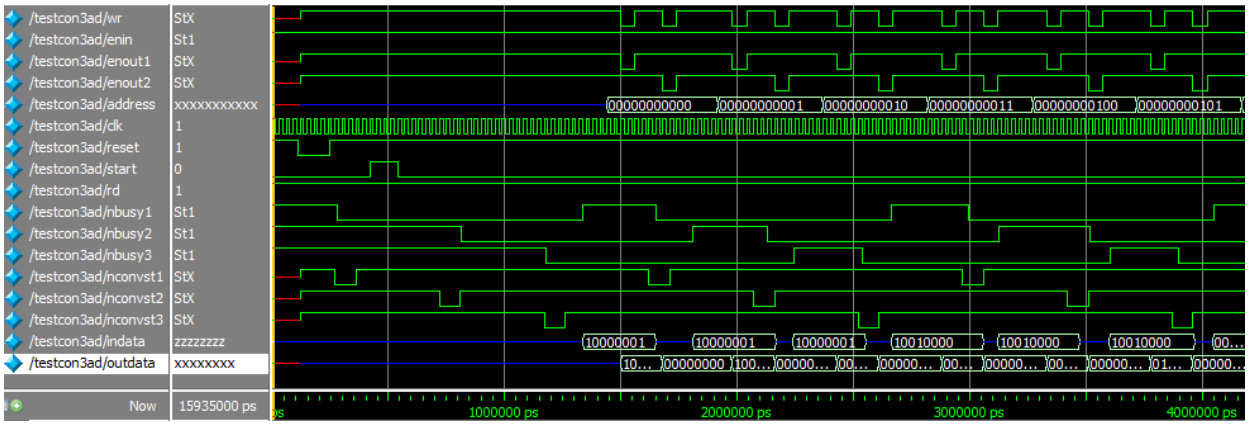
```
outdata),  .address(address),
              .CLK(clk),          .res
et(reset),      .start(start),
              .nconvst1(nconvst1),.nc
onvst2(nconvst2),.nconvst3(nconvst3)
,
              .nbusy1(nbusy1),    .nb
usy2(nbusy2),    .nbusy3(nbusy3),
              .wr(wr),            .enou
t1(enout1),    .enout2(enout2));

sram
ramlow( .Address(address),.Data(outd
ata),.SRW(wr),.SRG(rd),.SRE(enout1))
;


adc
ad_1(.nconvst(nconvst1),.nbusy(nbusy
1),.data(indata));
adc
ad_2(.nconvst(nconvst2),.nbusy(nbusy
2),.data(indata));
adc
ad_3(.nconvst(nconvst3),.nbusy(nbusy
3),.data(indata));

endmodule
```

3. 设计思路

首先控制 A/D 变换器进行变换，从 A/D 变换器得到变换后的数字序列，然后对数字序列进行卷积，把结果存入 RAM 中。

4. 实验结果

仿真得到的波形时序如图所示：



enout1、enout2 分别是存储卷积低、高字节结果 RAM 的片选信号，nconvst 给出 A/D 转换器的控制信号，start 高脉冲表示一次卷积运算的开始，nbusy 表示 A/D 转换器的忙/闲工作状态，结果由 outdata 存储。

时序符合题目的设计要求。

5. 实验总结

通过这次实验我掌握了通过阅读器件手册、了解器件原理之后进行它们的行为建模，对我理解 A/D 转换的过程有极大的帮助。

实验十二　利用 SRAM 设计一个 LIFO

1. 实验目的

（1）　学习和掌握存取堆栈管理的状态机设计的基本方法；
（2）　了解并掌握用储存器构成 LIFO 的接口设计的基本技术；
（3）　用工程概念来编写完整的测试模块，达到完整测试覆盖。

2. 实验内容

模仿书中例题的设计思路和方法，设计一个利用同类静态 RAM 的堆栈，即实现最大可达 1024 字节的 LIFO（Last In First Out）堆栈。

代码如下：

```
************************************
                LIFO.v
************************************
`define SRAM_SIZE 8
`timescale 1ns/1ns

module LIFO(
          in_data,
          out_data,
          liford,
          lifowr,
          nfull,
          nempty,
          address,
          sram_data,
          rd,
          wr,
          clk,
          rst);

input        liford, lifowr, clk,
rst;

input[7:0]    in_data;
output[7:0]   out_data;


reg[7:0]      in_data_buf,
             out_data_buf;

output        nfull, nempty;
reg           nfull, nempty;

output        rd, wr;

inout[7:0]    sram_data;

output[10:0]  address;
reg[10:0]     address;

reg[10:0]     lifo_wp,
             lifo_rp;

reg[10:0]     lifo_wp_next,
             lifo_rp_next;

reg           near_full, near_empty;

reg[3:0]      state;

parameter     idle       = 'b0000,
             read_ready  = 'b0100,
             read        = 'b0101,
             read_over   = 'b0111,
             write_ready = 'b1000,
             write       = 'b1001,
             write_over  = 'b1011;

always @(posedge clk or negedge rst)
  if (~rst)
    state <= idle;
  else
    case(state)
      idle:
```

```
            if (lifowr==0 && nfull)
              state<=write_ready;
            else if(liford==0 &&
nempty)
              state<=read_ready;
            else
              state<=idle;

        read_ready:
              state <= read;

        read:
            if (liford == 1)
              state <= read_over;
            else
              state <= read;

        read_over:
              state <= idle;

        write_ready:
              state <= write;

        write:
            if (lifowr == 1)
              state <= write_over;
            else
              state <= write;

        write_over:
              state <= idle;

        default: state<=idle;
   endcase

assign rd = ~state[2];
assign wr = (state == write) ?
lifowr : 1'b1;

always @(posedge clk)
   if (~lifowr)
      in_data_buf <= in_data;

assign sram_data = (state[3]) ?
                   in_data_buf :
8'hzz;


always @(state or liford or lifowr
or lifo_wp or lifo_rp)
   if (state[2] || ~liford)
      address = lifo_rp;
   else if (state[3] || ~lifowr)
      address = lifo_wp;
   else
      address = 'bz;

assign out_data = (state[2]) ?
                  sram_data : 8'bz;

always @(posedge clk)
   if (state == read)
      out_data_buf <= sram_data;

always @(posedge clk or negedge rst)
   if (~rst)
   begin
      lifo_rp <= -1;
      lifo_rp_next <= -2;
   end
   else if (state == read_over)
   begin
      lifo_wp = lifo_rp;
      lifo_rp = lifo_rp_next;
      lifo_rp_next = lifo_rp_next -
1;
      lifo_wp_next = lifo_wp_next -
1;
   end

always @(posedge clk or negedge rst)
   if (~rst)
   begin
      lifo_wp <= 0;
      lifo_wp_next <= 1;
   end
   else if (state == write_over)
   begin
```

```
      lifo_rp = lifo_wp;
      lifo_wp = lifo_wp_next;
      lifo_rp_next = lifo_rp_next +
1;
      lifo_wp_next = lifo_wp_next +
1;
   end

always @(posedge clk or negedge rst)
   if (~rst)
      near_empty <= 1'b0;
   else if (lifo_rp == 0)
      near_empty <= 1'b1;
   else
      near_empty <= 1'b0;

always @(posedge clk or negedge rst)
   if (~rst)
      nempty <= 1'b0;
   else if (near_empty && state ==
read)
      nempty <= 1'b0;
   else if (state == write)
      nempty <= 1'b1;

always @(posedge clk or negedge rst)
   if (~rst)
      near_full <= 1'b0;
   else if (lifo_wp == `SRAM_SIZE -
1)
      near_full <= 1'b1;
   else
      near_full <= 1'b0;

always @(posedge clk or negedge rst)
   if (~rst)
      nfull <= 1'b1;
   else if (near_full && state ==
write)
      nfull <= 1'b0;
   else if (state == read)
      nfull <= 1'b1;

endmodule
```

测试文件 Test12.v

```
`define lifo_SIZE 8
`include "sram.v"
`timescale 1ns/1ns

module Test12;

reg [7:0]     in_data;
reg           liford,lifowr;

wire[7:0]     out_data;
wire          nfull, nempty;

reg           clk,rst;

wire[7:0]     sram_data;
wire[10:0]    address;
wire          rd,wr;

reg [7:0]
data_buf[`lifo_SIZE:0];
integer index;

initial  clk=0;
always   #25 clk=~clk;

initial
  begin
     liford=1;
     lifowr=1;
     rst=1;
     #40 rst=0;
     #42 rst=1;

     if (nempty)
$display($time,"Error: lifo be
empty, nempty should be low.\n");

     index = 0;
     repeat(`lifo_SIZE) begin
        data_buf[index]=$random;
        write_lifo(data_buf[index]);
```

```
        index = index + 1;
      end

      if (nfull)
$display($time,"Error: lifo full,
nfull should be low.\n");
      repeat(2) write_lifo($random);
      #200

      index=0;

read_lifo_compare(data_buf[index]);
      if (~nfull)
$display($time,"Error: lifo not
full, nfull should be high.\n");

      repeat(`lifo_SIZE-1) begin
        index = index + 1;

read_lifo_compare(data_buf[index]);
      end

      if (nempty)
$display($time,"Error: lifo be
empty, nempty should be low.\n");

      repeat(2)
read_lifo_compare(8'bx);

      reset_lifo;

      repeat(`lifo_SIZE*2)
      begin
        data_buf[0] = $random;
        write_lifo(data_buf[0]);

read_lifo_compare(data_buf[0]);
      end

      reset_lifo;
      read_lifo_compare(8'bx);
      write_lifo(data_buf[0]);
      read_lifo_compare(data_buf[0]);
      $stop;
  end

LIFO  lifo(
            .in_data(in_data),.out_
data(out_data),
            .liford(liford),.lifowr
(lifowr),
            .nfull(nfull),.nempty(n
empty),
            .address(address),.sram
_data(sram_data),
            .rd(rd),.wr(wr),
            .clk(clk),.rst(rst)
      );

sram m1( .Address(address),
        .Data(sram_data),
        .SRG(rd),
        .SRE(1'b0),
        .SRW(wr));

task write_lifo;
input [7:0] data;
begin
        in_data=data;
   #50   lifowr=0;
   #200  lifowr=1;
   #50;
end
endtask

task read_lifo_compare;
input [7:0] data;
begin
   #50   liford=0;
   #200  liford=1;
        if (out_data != data)
          $display($time,"Error:
Data retrieved (%h) not match the
one stored (%h). \n",
                  out_data, data);

   #50;
```

```
end
endtask

task reset_lifo;
begin
   #40 rst=0;
   #40 rst=1;
end
endtask

endmodule
```

3. 设计思路

（1） 本题需要在前面例题的基础之上进行编写。首先是要引用练习十一的附录 2 当中 SRAM 的模型。

（2） 数据传输部分仅仅涉及 SRAM 与外部的通信，因此这部分内容可以直接将例题中的代码引用过来。

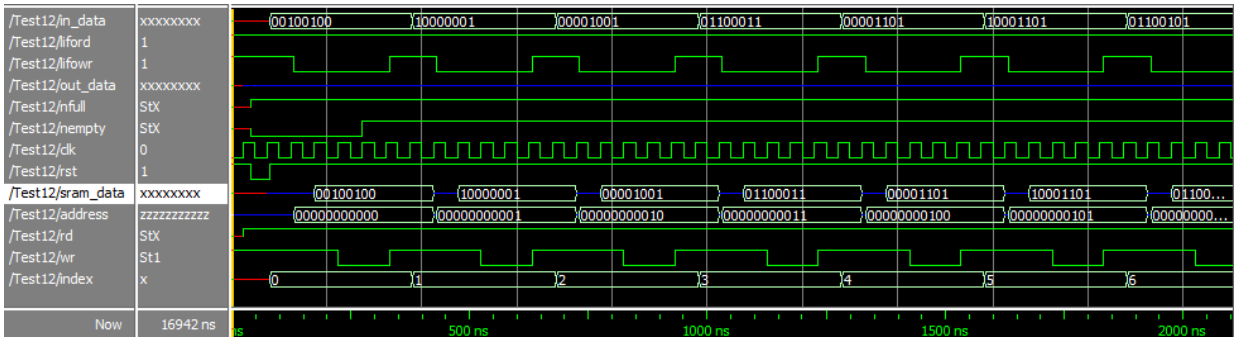
（3） 需要进行修改的仅仅是如何产生对 SRAM 的地址这一部分。类似于原例题中的思路，我们用一个数组来对这个问题建立模型。通过四个指针的移动来完成地址的改变。其重要注意与例题中不同的是，lifo_rp_next 和 lifo_wp_next 的移动方向，随着输入指令 liford 和 lifowr 的不同而不同。

（4） 由于编写的 LIFO 是从例题中的 FIFO 修改得到的，其对外接口都一样，因此仅需要直接调用例题中的测试文件即可完成测试。

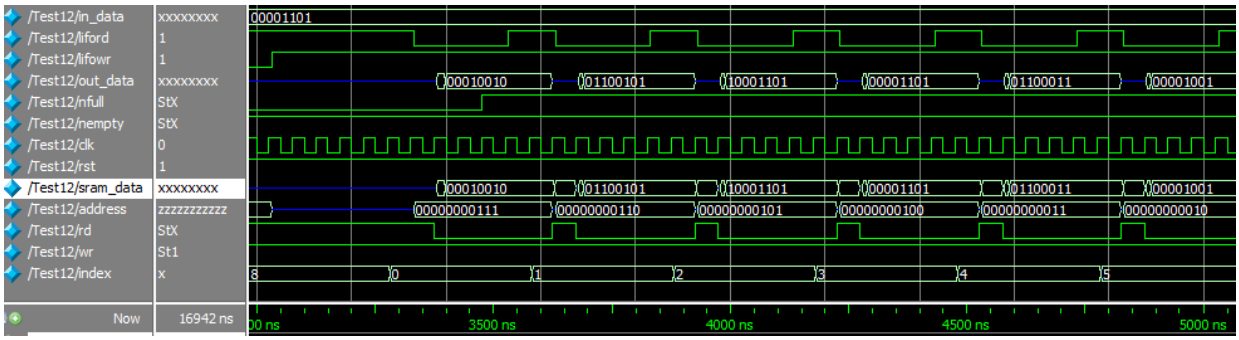
4. 实验结果

仿真得到的波形主要有以下三个部分：

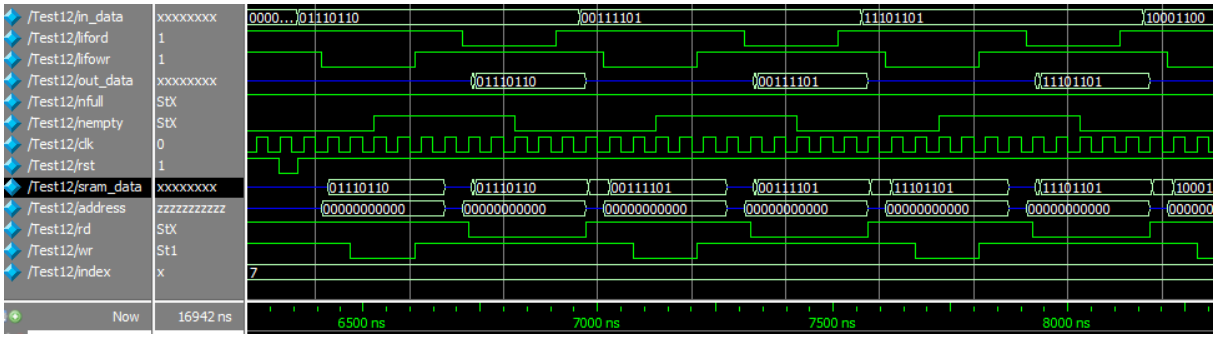
【图 1】



【图 2】



【图 3】



图一是测试模块中的连续写部分，可以看出，随着每个写时钟的到来，SRAM 中从地址 0 开始依次写入数据。当写地址大于 7 时，输入的数据不再被写入 SRAM 中。

图二是测试模块中的连续读部分，可以看出，随着每个读时钟的到来，SRAM 中从地址 7 开始依次读出数据。当读地址小于 0 时，不再从 SRAM 中读出数据。

综合图一和图二可以看到，第一个读出的数据与第八个写入的数据相同，第二个读出的数据与第七个写入的数据相同，符合 LIFO 的要求。

图三是测试模块中写后读部分，可以看出，每次写入数据后，SRAM 中会存入数据。下一个时钟周期里，再从 SRAM 中输出这个数据。

综上所述，所编写的代码完成了要求的功能。

5．实验总结

本次实验是 Verilog 最后一个上机实验，内容比较复杂，繁琐。虽然书中的例题已经解决了大部分问题，只需要改少量代码即能完成任务，在编写代码的过程中还是遇到了一些问题。

首先是读代码耗费了大量的时间。我用了一个多小时的时间读了例题中 FIFO 的主程序代码和测试文件代码，初步搞懂了程序的算法和设计思路。但是对于一些细节问题并没有特别深入的理解，仅仅是把计算 LIFO 读写指针这一部分研究的比较透彻，以便能够改动这部分代码。

接着是对 LIFO 的模型抽象过程中遇到了问题。由于 LIFO 与 FIFO 的规则不同，其指针的移动方向也不相同。而且判断堆栈的空和满的方式也不一样。

在修改代码的过程中，我们从例题出发，尝试改变了代码的结构，结果得到了不正确的结果。通过对于结果出现问题原因的分析，进一步理解了原例题中代码结构设定的方式。并且通过一个下午的调试，终于完成了这个工程。

附录

以下是实验十一和十二用到的两个模型 ADC 和 SRAM：

```
//////////////////////////////////////////////////////////////////////////////////
                                    adc.v
//////////////////////////////////////////////////////////////////////////////////
`timescale 100 ps/100 ps

module  adc (nconvst, nbusy, data);
input   nconvst;
output  nbusy;
output  data;
reg[7:0]    databuf,i;
reg     nbusy;
wire[7:0]   data;
reg[7:0]    data_mem[0:255];
reg     link_bus;
integer tconv,
        t5,
        t8,
        t9,
        t12;
integer wideth1,
        wideth2,
        wideth;

always  @(negedge nconvst)
    begin
        tconv   = 9500+{$random} %500;
        t5      = {$random} %1000;
        t8      = 200;
        t9      = 100+{$random} %900;
        t12     = 2500;
    end

initial
    begin
        $readmemh("adc.data",data_mem);
        i = 0;
        nbusy = 1;
        link_bus = 0;
    end

assign  data = link_bus ? databuf : 8'bzz;
```

```
always @(negedge nconvst)
    fork
        #t5 nbusy =0;
        @(posedge nconvst)
            begin
                #tconv  nbusy= 1;
            end
    join

always @(negedge nconvst)
    begin
        @(posedge nconvst)
            begin
                #(tconv-t8) databuf = data_mem[i];
            end

        if(wideth <10000 && wideth>500)
            begin
                if(i== 255) i=0;
                else       i=i+1;
            end
        else i = i;
    end


always @(negedge nconvst)
    fork
        #t9 link_bus= 1'b0;          @(posedge nconvst)
            begin
                #(tconv-18) link_bus=1'b1;
            end
    join

always @(posedge nbusy)
    begin
        #t12;
        if(!nconvst)
            begin
                $display("Warning!  SHA Acquisition Time is too short! ") ;
            end
        //else $display(" SHA Acquisition Time is enough! ");
    end
```

```
always @(negedge nconvst)
    begin
        wideth= $time;
        @(posedge nconvst)  wideth= $time-wideth;
        if(wideth <= 500 || wideth > 10000)
            begin
                $display ("nCONVST Pulse Width = %d",wideth);
                $display ("Warning! nCONVST Pulse Width is too narrow or too
wide!");
            // $stop;
            end
    end

endmodule
```

****

sram.v

****

```
`timescale 1ns/1ns
/*************************************************************************
***
 *   File Name          : sram.v                                        *
 *   Function           : 2K*8bit Asynchronous CMOS Static RAM           *

*************************************************************************
*/
/*************************************************************************
**
 * Module Name          : sram                                          *
 * Description          : 2K*8bit Asynchronous CMOS Static RAM           *
 * Reference               : HM-65162 reference book                     *

*************************************************************************
*/
/*************************************************************************
***
 * sram is a Verilog HDL model for HM-65162,2K*8bit Asynchronous CMOS Static
*
 * RAM. It is used in simulation to substitute the real RAM to verify whether
 * the writing or reading of the RAM is OK. This module is a behavioral model
*
 * for simulation only, not synthesizable. It's writing and reading function
*
 * are verified.                                                          *
```

```
 ***************************
*******************************************/
//---------------------- sram.v ----------------------------------------
module sram(Address, Data, SRG, SRE, SRW);
input [10:0]   Address;

input         SRG, // Output enable
          SRE, // Chip enable
          SRW; // Write enable

inout [7:0]    Data;    // Bus

wire  [10:0]   Addr = Address;
reg   [7:0]    RdData;
reg   [7:0]    SramMem [0:'h7ff];
reg           RdSramDly, RdFlip;
wire  [7:0]    FlpData, Data;
reg    WR_flag;  //To judge the signals according to the specification of HM-
65162
integer       i;

wire          RdSram = ~SRG & ~SRE;
wire          WrSram = ~SRW & ~SRE;
reg   [10:0]   DelayAddr;
reg   [7:0]    DelayData;
reg       WrSramDly;

integer file;

assign FlpData = (RdFlip) ? ~RdData : RdData;
assign Data    = (RdSramDly) ? FlpData: 'hz;

/******************parameters of read circle***************************/
                //²ÎÊýÐò°Å¡¢×î´ó»ò×îÐ¡¡¢²ÎÊý°¬Òå
parameter TAVQV=90,   //2         (max)    Address access time
        TELQV=90,   //3         (max)   Chip enable access time
        TELQX=5,    //4         (min)   Chip enable output enable    time
        TGLQV=65,   //5         (max)   Output enable access time
        TGLQX=5,    //6         (min)   Output inable output enable time
        TEHQZ=50,   //7         (max)   Chip enable output disable time
        TGHQZ=40,   //8         (max)   Output enable output disable time
        TAVQX=5;    //9         (min)   Output hold from address change

/******************parameters of write circle*************************/
```

```
parameter TAVWL=10,   //12       (min)  Address setup  time,
          TWLWH=55,   //13        (min)  Chip enable pulse setup time,
                                       //write enable pluse width,
          TWHAX=15,   //14       (min10)    Write    enable read setup time,
                                       //¶ÁÉÏÉýÑØºóµØÖ·±£Áô¼¼ä
          TWLQZ=50,   //16       (max)  Write enable output disable time
          TDVWH=30,   //17       (min)    Data setup time
          TWHDX=20,   //18       (min15)  Data hold time
          TWHQX=20,   //19       (min0)   Write    enable output enable time,0
          TWLEH=55,   //20       (min)   Write enable pulse setup time
          TDVEH=30,   //21       (min)   Chip enable data setup time
          TAVWH=70;   //22       (min65)  Address valid    to end of write

initial
  begin
      file=$fopen("ramlow.txt");
      if(!file)
    begin
        $display("Could not open the file.");
        $stop;
    end
  end

initial
  begin
    for(i=0 ; i<'h7ff ; i=i+1)
      SramMem[i] = i;
  //   $monitor($time,,"DelayAddr=%h,DelayData=%h",DelayAddr,DelayData);
  end

initial  RdSramDly = 0;
initial  WR_flag=1;

/*************************READ CIRCLE*****************************/
always @(posedge RdSram)  #TGLQX  RdSramDly = RdSram;
always @(posedge SRW)     #TWHQX  RdSramDly = RdSram;

always @(Addr)
      begin
     #TAVQX;
     RdFlip = 1;
     #(TGLQV - TAVQX);        //address access time
     if (RdSram)     RdFlip = 0;
```

```
      end

always @(posedge RdSram)
      begin
     RdFlip = 1;
     #TAVQV;   // Output enable access time
     if (RdSram) RdFlip = 0;
      end

always @(Addr)      #TAVQX RdFlip = 1;

always @(posedge SRG)    #TEHQZ RdSramDly  = RdSram;
always @(posedge SRE)    #TGHQZ RdSramDly  = RdSram;
always @(negedge SRW)    #TWLQZ RdSramDly  = 0;

always @(negedge WrSramDly or posedge RdSramDly)   RdData = SramMem[Addr];

/*************************WRITE CIRCLE*************************/
always @(Addr)    #TAVWL DelayAddr  = Addr; //Address setup
always @(Data)    #TDVWH DelayData  = Data; //Data setup
always @(WrSram)  #5 WrSramDly =WrSram;
always @(Addr or Data or WrSram)  WR_flag=1;

always @(negedge SRW )
       begin
          #TWLWH;      //Write enable pulse width
          if (SRW)
           begin
               WR_flag=0;
               $display("ERROR! Can't write!
                         Write enable time (W) is too short!");
           end
       end

always @(negedge SRW )
       begin
          #TWLEH;      //Write enable pulse setup time
          if (SRE)
           begin
               WR_flag=0;
               $display("ERROR! Can't write! Write enable
                           pulse setup time (E) is too short!");
           end
       end
```

```
always @(posedge SRW )
      begin
          #TWHAX;       //Write enable read setup time
          if(DelayAddr !== Addr)
            begin
                WR_flag=0;
                $display("ERROR! Can't write!
                        Write enable read setup   time is too short!");
            end
      end

always @(Data)
   if (WrSram)
    begin
        #TDVEH;        //Chip enable data setup time
        if (SRE)
          begin
              WR_flag=0;
              $display("ERROR! Can't write!
                     Chip enable Data setup time is too short!");
          end
    end

always @(Data)
  if (WrSram)
    begin
        #TDVEH;
        if (SRW)
          begin
              WR_flag=0;
              $display("ERROR! Can't write!
                     Chip enable Data setup time is too short!");
          end
    end

always @(posedge SRW )
  begin
      #TWHDX;        //Data hold time
      if(DelayData !== Data)
       $display("Warning!  Data hold time is too short!");
  end

always @(DelayAddr or DelayData or WrSramDly)
```

```
  if (WrSram &&WR_flag)
    begin
        if(!Addr[5])
          begin
              #15 SramMem[Addr]=Data;
             // $display("mem[%h]=%h",Addr,Data);
              $fwrite(file,"mem[%h]=%h    ",Addr,Data);
              if(Addr[0]&&Addr[1])  $fwrite(file,"\n");
          end
        else
          begin
              $fclose(file);
              $display("Please check the txt.");
              //$stop;
          end
    end

endmodule
```