```
output[7:0] out;
reg[7:0]    out;
input[2:0] opcode;
input[7:0] a,b;              //操作数。

always@(opcode or a or b)    //电平敏感的always块
begin
      case(opcode)
          `plus:  out = a+b;   //加操作。
          `minus: out = a-b;  //减操作。
          `band:  out = a&b;   //求与。
          `bor:   out = a|b;   //求或。
          `unegate:  out=~a; //求反。
          default:   out=8'hx;//未收到指令时，输出任意态。
      endcase
end
endmodule
```

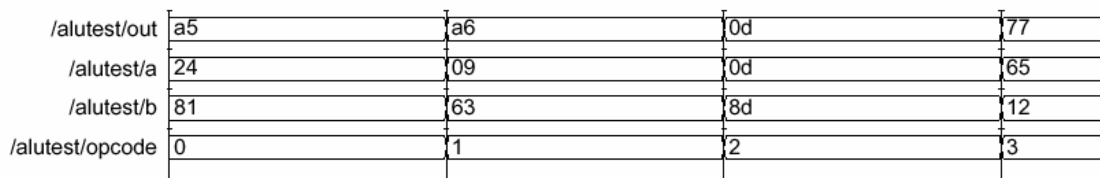
同一组合逻辑电路分别用always块和连续赋值语句assign描述时，代码的形式大相径庭，但是在always中适当运用default（在case结构中）和else（在if…else结构中），通常可以综合为纯组合逻辑，尽管被赋值的变量一定要定义为 reg 型。不过，如果不使用default或else对缺省项进行说明，则易生成意想不到的锁存器，这一点一定要加以注意。

指令译码器的测试模块源代码：

```
//------------- alu_Top.v -----------------
`timescale 1ns/1ns
`include  "./alu.v"
module    alutest;
   wire[7:0] out;
   reg[7:0]  a,b;
   reg[2:0]  opcode;
   parameter   times=5;
   initial
    begin
        a={$random}%256;  //Give a radom number blongs to [0,255] .
        b={$random}%256;  //Give a radom number blongs to [0,255].
        opcode=3'h0;
        repeat(times)
          begin
          #100      a={$random}%256;  //Give a radom number.
                    b={$random}%256;  //Give a radom number.
                    opcode=opcode+1;
          end
```

```
        #100  $stop;

    end

    alu     alu1(out,opcode,a,b);

endmodule
```

仿真波形（部分）：



练习：运用 always 块设计一个八路数据选择器。要求：每路输入数据与输出数据均为 4 位 2 进制数，当选择开关(至少 3 位)或输入数据发生变化时，输出数据也相应地变化。

练习六. 在 Verilog HDL 中使用函数

目的：掌握函数在模块设计中的使用。

与一般的程序设计语言一样，Veirlog HDL 也可使用函数以适应对不同变量采取同一运算的操作。Veirlog HDL 函数在综合时被理解成具有独立运算功能的电路，每调用一次函数相当于改变这部分电路的输入以得到相应的计算结果。

下例是函数调用的一个简单示范，采用同步时钟触发运算的执行，每个 clk 时钟周期都会执行一次运算。并且在测试模块中，通过调用系统任务$display 在时钟的下降沿显示每次计算的结果。

模块源代码：

```
module tryfunct(clk,n,result,reset);

  output[31:0] result;
  input[3:0]  n;
  input reset,clk;
  reg[31:0] result;

  always @(posedge clk)     //clk 的上沿触发同步运算。
    begin
         if(!reset)              //reset 为低时复位。
            result<=0;
        else
            begin
```

```
                result <= n * factorial(n)/((n*2)+1);
            end
    end

  function [31:0] factorial;      //函数定义。
    input  [3:0]  operand;
    reg    [3:0]  index;
    begin
      factorial = operand ? 1 : 0;
      for(index = 2; index <= operand; index = index + 1)
        factorial = index * factorial;
    end
  endfunction

endmodule
```

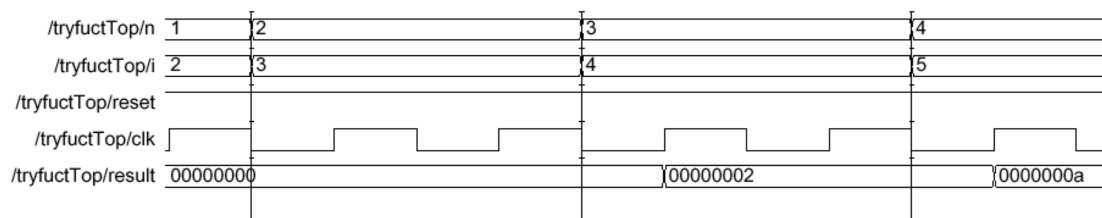
测试模块源代码：

```
`include "./step6.v"
`timescale 1ns/100ps
`define clk_cycle 50

module tryfuctTop;

reg[3:0] n,i;
reg reset,clk;

wire[31:0] result;

initial
  begin
    n=0;
    reset=1;
    clk=0;
    #100 reset=0;
    #100 reset=1;
    for(i=0;i<=15;i=i+1)
      begin
        #200 n=i;
      end
    #100 $stop;
  end

always #`clk_cycle clk=~clk;
```

```
tryfunct tryfunct(.clk(clk),.n(n),.result(result),.reset(reset));

endmodule
```

上例中函数 factorial(n)实际上就是阶乘运算。必须提醒大家注意的是，在实际的设计中，我们不希望设计中的运算过于复杂，以免在综合后带来不可预测的后果。经常的情况是，我们把复杂的运算分成几个步骤，分别在不同的时钟周期完成。

仿真波形(部分)：



练习：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、立方和阶乘的运算。编写测试模块，并给出仿真波形。

练习七. 在 Verilog HDL 中使用任务（task）

目的：掌握任务在结构化 Verilog HDL 设计中的应用。

仅有函数并不能完全满足 Veirlog HDL 中的运算需求。当我们希望能够将一些信号进行运算并输出多个结果时，采用函数结构就显得非常不方便，而任务结构在这方面的优势则十分突出。任务本身并不返回计算值，但是它通过类似 C 语言中形参与实参的数据交换，非常快捷地实现运算结果的调用。此外，我们还常常利用任务来帮助我们实现结构化的模块设计，将批量的操作以任务的形式独立出来，这样设计的目的通常一眼看过去就很明了。

下面是一个利用 task 和电平敏感的 always 块设计比较后重组信号的组合逻辑的实例。可以看到，利用 task 非常方便地实现了数据之间的交换，如果要用函数实现相同的功能是非常复杂的；另外，task 也避免了直接用一般语句来描述所引起的不易理解和综合时产生冗余逻辑等问题。

模块源代码：

```
//----------------- sort4.v ------------------
module sort4(ra,rb,rc,rd,a,b,c,d);
  output[3:0] ra,rb,rc,rd;
  input[3:0] a,b,c,d;
  reg[3:0] ra,rb,rc,rd;
  reg[3:0] va,vb,vc,vd;

  always @ (a or b or c or d)
    begin
```

```
      {va,vb,vc,vd}={a,b,c,d};
      sort2(va,vc);              //va 与 vc 互换。
      sort2(vb,vd);              //vb 与 vd 互换。
      sort2(va,vb);              //va 与 vb 互换。
      sort2(vc,vd);              //vc 与 vd 互换。
      sort2(vb,vc);              //vb 与 vc 互换。
      {ra,rb,rc,rd}={va,vb,vc,vd};
    end

  task sort2;
    inout[3:0] x,y;
    reg[3:0] tmp;
    if(x>y)
      begin
        tmp=x;       //x 与 y 变量的内容互换，要求顺序执行，所以采用阻塞赋值方式。
        x=y;
        y=tmp;
      end
  endtask

endmodule
```

值得注意的是 task 中的变量定义与模块中的变量定义不尽相同,它们并不受输入输出类型的限制。如此例,x 与 y 对于 task sort2 来说虽然是 inout 型,但实际上它们对应的是 always 块中变量，都是 reg 型变量。

测试模块源代码：

```
`timescale 1ns/100ps
`include "sort4.v"

module task_Top;
 reg[3:0] a,b,c,d;
 wire[3:0] ra,rb,rc,rd;

 initial
  begin
    a=0;b=0;c=0;d=0;
    repeat(5)
    begin
      #100  a ={$random}%15;
            b ={$random}%15;
            c ={$random}%15;
            d ={$random}%15;
    end
```

```
    #100 $stop;

sort4 sort4 (.a(a),.b(b),.c(c),.d(d), .ra(ra),.rb(rb),.rc(rc),.rd(rd));

endmodule
```

仿真波形（部分）：

/task_Top/a	0000	1000	1100	0110
/task_Top/b	0000	1100	0010	0100
/task_Top/c	0000	0111	0101	0011
/task_Top/d	0000	0010	0111	0010
/task_Top/ra	0000	0010		
/task_Top/rb	0000	0111	0101	0011
/task_Top/rc	0000	1000	0111	0100
/task_Top/rd	0000	1100		0110

练习：设计一个模块，通过任务完成 3 个 8 位 2 进制输入数据的冒泡排序。要求：时钟触发任务的执行，每个时钟周期完成一次数据交换的操作。

练习八. 利用有限状态机进行复杂时序逻辑的设计

目的：掌握利用有限状态机实现复杂时序逻辑的方法；

在数字电路中我们已经学习过通过建立有限状态机来进行数字逻辑的设计，而在 Verilog HDL 硬件描述语言中，这种设计方法得到进一步的发展。通过 Verilog HDL 提供的语句，我们可以直观地设计出适合更为复杂的时序逻辑的电路。关于有限状态机的设计方法在教材中已经作了较为详细的阐述，在此就不赘述了。

下例是一个简单的状态机设计，功能是检测一个 5 位二进制序列“10010”。考虑到序列重叠的可能，有限状态机共提供 8 个状态（包括初始状态 IDLE）。

模块源代码：

```
seqdet.v
module seqdet(x,z,clk,rst,state);
input  x,clk,rst;
output z;
output[2:0] state;
reg[2:0] state;
wire z;
```

```
parameter IDLE='d0,  A='d1,  B='d2,
                     C='d3,  D='d4,
                     E='d5,  F='d6,
                     G='d7;

assign  z = ( state==E && x==0 )? 1 : 0;     //当x=0时，状态已变为E，
                                            //状态为D时，x仍为1。因此
                      //输出为1的条件为（ state==E && x==0 ）。
always @(posedge clk)
   if(!rst)
          begin
          state <= IDLE;
          end
   else
          casex(state)
            IDLE : if(x==1)
                       begin
                          state <= A;
                       end
            A:      if(x==0)
                       begin
                          state <= B;
                       end
            B:      if(x==0)
                       begin
                          state <= C;
                       end
                    else
                       begin
                          state <= F;
                       end
            C:       if(x==1)
                       begin
                          state <= D;
                       end
                    else
                       begin
                          state <= G;
                       end
            D:       if(x==0)
                       begin
                          state <= E;
```

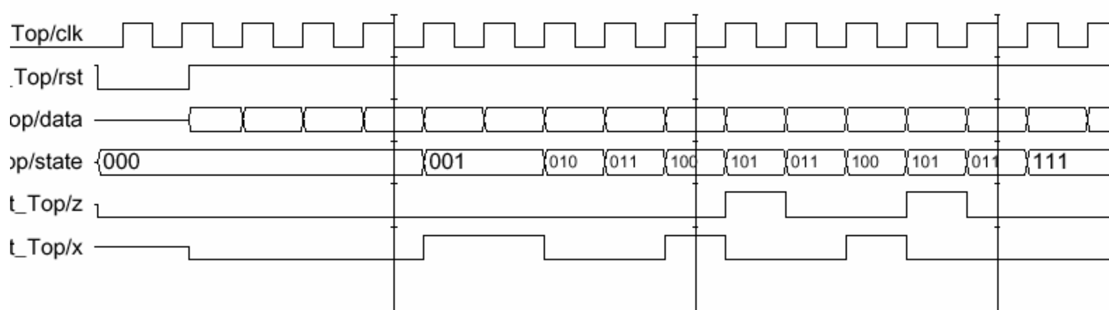
```
                    end
                 else
                    begin
                       state <= A;
                    end
           E:       if(x==0)
                    begin
                       state <= C;
                    end
                 else
                    begin
                       state <= A;
                    end
           F:       if(x==1)
                    begin
                       state <= A;
                    end
                 else
                    begin
                       state <= B;
                    end
           G:       if(x==1)
                    begin
                       state <= F;
                    end
          default:state=IDLE;      //缺省状态为初始状态。
          endcase
endmodule
```

测试模块源代码：

```
//------------------ seqdet.v -------------------
`timescale 1ns/1ns
`include "./seqdet.v"
module seqdet_Top;
  reg clk,rst;
  reg[23:0] data;
  wire[2:0] state;
  wire z,x;
  assign x=data[23];
  always  #10 clk = ~clk;
  always @(posedge clk)
         data={data[22:0],data[23]};

  initial
```

```
    begin
      clk=0;
      rst=1;
      #2 rst=0;
      #30 rst=1;
      data ='b1100_1001_0000_1001_0100;
      #500 $stop;
    end

  seqdet  m(x,z,clk,rst,state);

endmodule
```

仿真波形：



练习：设计一个串行数据检测器。要求是：连续 4 个或 4 个以上的 1 时输出为 1，其他输入情况下为 0。编写测试模块并给出仿真波形。

练习九. 利用状态机的嵌套实现层次结构化设计

目的： 1.运用主状态机与子状态机产生层次化的逻辑设计；
2.在结构化设计中灵活使用任务（task）结构。

在上一节，我们学习了如何使用状态机的实例。实际上，单个有限状态机控制整个逻辑电路的运转在实际设计中是不多见，往往是状态机套用状态机，从而形成树状的控制核心。这一点也与我们提倡的层次化、结构化的自顶而下的设计方法相符，下面我们就将提供一个这样的示例以供大家学习。

该例是一个简化的 EPROM 的串行写入器。事实上，它是一个 EPROM 读写器设计中实现写功能的部分经删节得到的，去除了 EPROM 的启动、结束和 EPROM 控制字的写入等功能，只具备这样一个雏形。工作的步骤是： 1.地址的串行写入； 2.数据的串行写入；3.给信号源应答，信号源给出下一个操作对象； 4.结束写操作。通过移位令并行数据得以一位一位输出。

模块源代码：

```
module writing(reset,clk,address,data,sda,ack);
  input reset,clk;
  input[7:0] data,address;

  output sda,ack;  //sda 负责串行数据输出;
                   //ack 是一个对象操作完毕后，模块给出的应答信号。
  reg link_write;  //link_write 决定何时输出。
  reg[3:0] state;  //主状态机的状态字。
  reg[4:0] sh8out_state;  //从状态机的状态字。
  reg[7:0] sh8out_buf;    //输入数据缓冲。
  reg finish_F;           //用以判断是否处理完一个操作对象。
  reg ack;

  parameter
    idle=0,addr_write=1,data_write=2,stop_ack=3;
  parameter
    bit0=1,bit1=2,bit2=3,bit3=4,bit4=5,bit5=6,bit6=7,bit7=8;

  assign   sda = link_write? sh8out_buf[7] : 1'bz;

  always @(posedge clk)
    begin
    if(!reset)               //复位。
      begin
         link_write<= 0;
         state    <= idle;
         finish_F <= 0;
         sh8out_state<=idle;
               ack<= 0;
         sh8out_buf<=0;
      end
    else
      case(state)

      idle:
        begin
        link_write  <= 0;
           state    <= idle;
           finish_F <= 0;
           sh8out_state<=idle;
                 ack<= 0;
           sh8out_buf<=address;
          state    <= addr_write;
        end
```