```
        $display("Undefined instruction");
        dumpstate;
        $finish;
    end
    endtask


    /* print the state of the 8085a */
    task dumpstate;
    begin
        $write( "\nDUMP OF 8085A REGISTERS\n",
            "acc=%h regb=%h regc=%h regd=%h rege=%h regh=%h regl=%h\n",
                acc, regb, regc, regd, rege, regh, regl,
            "cs=%h cz=%h cac=%h cp=%h cc=%h\n",
                cs, cz, cac, cp, cc,
            "pc=%h sp=%h addr=%h ir=%h data=%h\n",
                pc, sp, ir, addr, data,
            "intmask=%h aleff=%h s0ff=%h s1ff=%h hldaff=%h holdff=%h\n",
                intmask, aleff, s0ff, s1ff, hldaff, holdff,
            "intaff=%h trapff=%h trapi=%h inte=%h int=%h validint=%h\n",
                intaff, trapff, trapi, inte, int, validint,
            "haltff=%h resetff=%h clockff=%h sodff=%h\n",
                haltff, resetff, clockff, sodff,
            "read=%h write=%h iomff=%h acontrol=%h dcontrol=%h s=%h\n",
                read, write, iomff, acontrol, dcontrol, s,
            "clock=%h x2=%h sid=%h trap=%h rst7p5=%h rst6p5=%h rst5p5=%h\n",
                clock, x2, sid, trap, rst7p5, rst6p5, rst5p5,
            "intr=%h nreset=%h hold=%h ready=%h a=%h ad=%h\n\n",
                intr, nreset, hold, ready, a, ad,
            "instructions executed = %d\n\n", instruction);
    end
    endtask

endmodule /* of i85 */
```

上面两个例子是常用的微处理机 CPU 和外围芯片。在系统芯片的设计中，我们可以用虚拟模型来代替真实的器件对自己所设计的电路功能进行仿真，全面精确地验证自己所设计的部分是否正确。在 ASIC 的制造过程中我们可以利用现存的与之对应的门级结构的电路实体来实现电路的功能。这样就能用较快的速度把许多人的劳动成果集合在一起，把一个极其复杂的数字系统集成在一个很小的硅片上。

思考题：

1) 为什么要设计虚拟模块？
2) 虚拟模块有几种类型？

3) 为什么在 ASIC 设计中要尽量利用商业化的虚拟模块和 IP?

4) 为什么说编写完整精确的虚拟模块，编写者不但需要全面熟练地掌握 Verilog 语言，还需要有高度的责任性，并且需要有一个严格的质量保证体系来确保与工艺的电路的一致性？

第十章 设计练习进阶

前言：

在前面九章学习的基础上， 通过本章十个阶段的练习，一定能逐步掌握 Verilog HDL 设计的要点。我们可以先理解样板模块中每一条语句的作用，然后对样板模块进行综合前和综合后仿真，再独立完成每一阶段规定的练习。当十个阶段的练习做完后，便可以开始设计一些简单的逻辑电路和系统。很快我们就能过渡到设计相当复杂的数字逻辑系统。当然，复杂的数字逻辑系统的设计和验证，不但需要系统结构的知识和经验的积累，还需要了解更多的语法现象和掌握高级的 Verilog HDL 系统任务，以及与 C 语言模块接口的方法（即 PLI），这些已超出的本书的范围。有兴趣的同学可以阅读 Verilog 语法参考资料和有关文献，自己学习，我们将在下一本书中介绍 Verilog 较高级的用法。

练习一. 简单的组合逻辑设计

目的: 掌握基本组合逻辑电路的实现方法。

这是一个可综合的数据比较器，很容易看出它的功能是比较数据 a 与数据 b，如果两个数据相同,则给出结果 1,否则给出结果 0。在 Verilog HDL 中,描述组合逻辑时常使用 assign 结构。注意 equal=(a==b)?1:0,这是一种在组合逻辑实现分支判断时常使用的格式。

模块源代码：

```
//---------------- compare.v ------------------
module compare(equal,a,b);
input a,b;
output equal;
   assign  equal=(a==b)?1:0; //a 等于 b 时，equal 输出为 1；a 不等于 b 时，
                             //equal 输出为 0。
endmodule
```

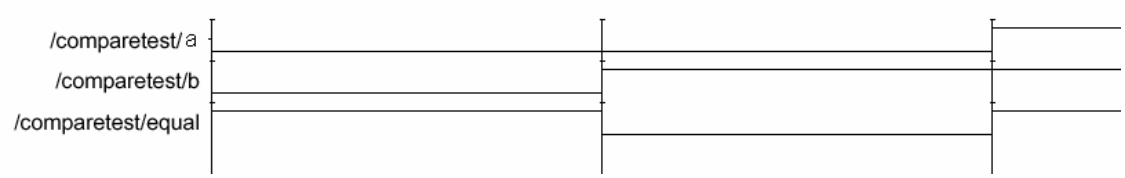
测试模块用于检测模块设计得正确与否，它给出模块的输入信号，观察模块的内部信号和输出信号，如果发现结果与预期的有所偏差，则要对设计模块进行修改。

测试模块源代码：

```
`timescale 1ns/1ns       //定义时间单位。
`include  ".\compare.v" //包含模块文件。在有的仿真调试环境中并不需要此语句。
                        //而需要从调试环境的菜单中键入有关模块文件的路径和名称
module  comparetest;
   reg a,b;
   wire equal;
   initial               //initial 常用于仿真时信号的给出。
```

```
    begin
      a=0;
      b=0;
    #100   a=0; b=1;
    #100   a=1; b=1;
    #100   a=1; b=0;
    #100   $stop;       //系统任务，暂停仿真以便观察仿真波形。
    end

   compare  compare1(.equal(equal),.a(a),.b(b));    //调用模块。

endmodule
```

仿真波形（部分）：



练习：
设计一个字节（8 位）比较器。
要求：比较两个字节的大小，如 a[7:0]大于 b[7:0]输出高电平,否则输出低电平,改写测试模型,使其能进行比较全面的测试 。

练习二. 简单时序逻辑电路的设计

目的：掌握基本时序逻辑电路的实现。

在 Verilog HDL 中，相对于组合逻辑电路，时序逻辑电路也有规定的表述方式。在可综合的 Verilog HDL 模型，我们通常使用 always 块和 @(posedge clk)或 @(negedge clk)的结构来表述时序逻辑。下面是一个 1/2 分频器的可综合模型。

```
// half_clk.v:

   module half_clk(reset,clk_in,clk_out);
   input clk_in,reset;
   output clk_out;
   reg clk_out;

   always @(posedge clk_in)
```

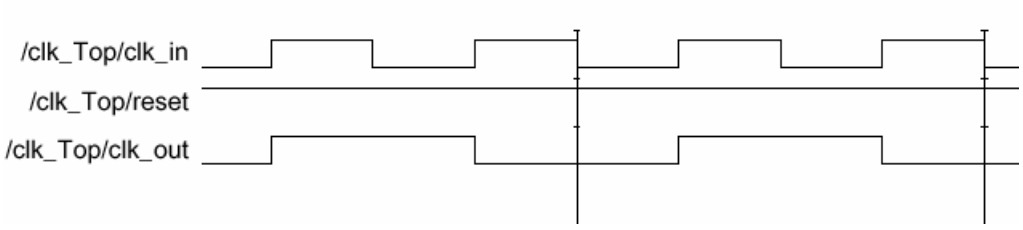
```
    begin
      if(!reset)  clk_out=0;
      else      clk_out=~clk_out;
    end
  endmodule
```

在 always 块中，被赋值的信号都必须定义为 reg 型，这是由时序逻辑电路的特点所决定的。对于 reg 型数据，如果未对它进行赋值，仿真工具会认为它是不定态。为了能正确地观察到仿真结果，在可综合风格的模块中我们通常定义一个复位信号 reset，当 reset 为低电平时，对电路中的寄存器进行复位。

测试模块的源代码：

```
//-------------------  clk_Top.v ------------------------------

`timescale 1ns/100ps
`define clk_cycle 50

module clk_Top.v
reg clk,reset;
wire clk_out;

always  #`clk_cycle  clk = ~clk;

initial
  begin
        clk = 0;
        reset = 1;
    #100 reset = 0;
    #100 reset = 1;
    #10000 $stop;
  end

half_clk half_clk(.reset(reset),.clk(clk_in),.clk_out(clk_out));

endmodule
```

仿真波形：



练习：依然作 clk_in 的二分频 clk_out，要求输出与上例的输出正好反相。编写测试模块，给出仿真波形。

练习三. 利用条件语句实现较复杂的时序逻辑电路

目的：掌握条件语句在 Verilog HDL 中的使用。

与常用的高级程序语言一样,为了描述较为复杂的时序关系，Verilog HDL 提供了条件语句供分支判断时使用。在可综合风格的 Verilog HDL 模型中常用的条件语句有 if...else 和 case...endcase 两种结构，用法和 C 程序语言中类似。两者相较，if...else 用于不很复杂的分支关系，实际编写可综合风格的模块、特别是用状态机构成的模块时，更常用的是 case...endcase 风格的代码。这一节我们给的是有关 if...else 的范例，有关 case...endcase 结构的代码已后会经常用到。

下面给出的范例也是一个可综合风格的分频器，是将 10M 的时钟分频为 500K 的时钟。基本原理与 1/2 分频器是一样的，但是需要定义一个计数器，以便准确获得 1/20 分频

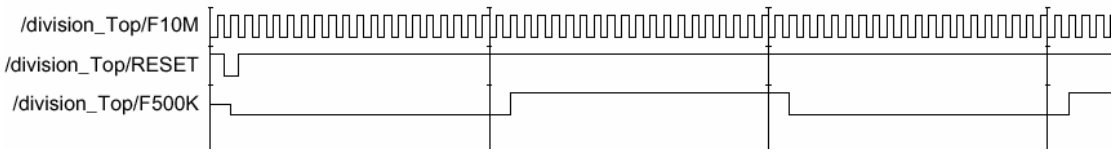
模块源代码：

```
// ---------------- fdivision.v --------------------------------
module fdivision(RESET,F10M,F500K);
    input F10M,RESET;
    output F500K;
    reg F500K;
    reg [7:0]j;
      always @(posedge F10M)
        if(!RESET)          //低电平复位。
          begin
            F500K <= 0;
            j <= 0;
          end
        else
          begin
            if(j==19)       //对计数器进行判断，以确定 F500K 信号是否反转。
              begin
                j <= 0;
                F500K <= ~F500K;
              end
            else
              j <= j+1;
          end
    endmodule
```

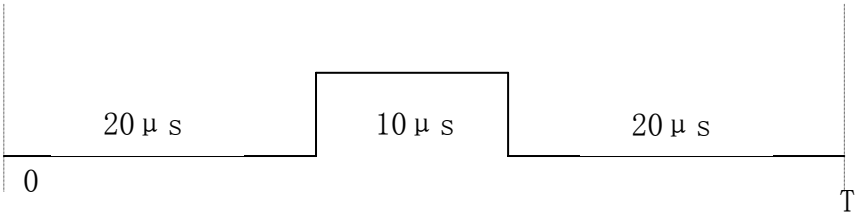
测试模块源代码：

```
//---------------  fdivision_Top.v ------------------------

`timescale 1ns/100ps
`define clk_cycle 50

module division_Top;

reg F10M,RESET;

wire  F500K_clk;

always #`clk_cycle   F10M_clk  =  ~ F10M_clk;

initial
  begin
    RESET=1;
    F10M=0;
    #100 RESET=0;
    #100 RESET=1;
    #10000 $stop;
  end

fdivision fdivision (.RESET(RESET),.F10M(F10M),.F500K(F500K_clk));

endmodule
```

仿真波形：



练习：利用 10M 的时钟，设计一个单周期形状如下的周期波形。



练习四. 设计时序逻辑时采用阻塞赋值与非阻塞赋值的区别

目的：1.明确掌握阻塞赋值与非阻塞赋值的概念和区别；
2.了解阻塞赋值的使用情况。

阻塞赋值与非阻塞赋值，在教材中我们已经了解了它们之间在语法上的区别以及综合后所得到的电路结构上的区别。在 always 块中，阻塞赋值可以理解为赋值语句是顺序执行的，而非阻塞赋值可以理解为赋值语句是并发执行的。实际的时序逻辑设计中，一般的情况下非阻塞赋值语句被更多地使用，有时为了在同一周期实现相互关联的操作，也使用了阻塞赋值语句。(注意：在实现组合逻辑的 assign 结构中，无一例外地都必须采用阻塞赋值语句。

下例通过分别采用阻塞赋值语句和非阻塞赋值语句的两个看上去非常相似的两个模块 blocking.v 和 non_blocking.v 来阐明两者之间的区别。

模块源代码：

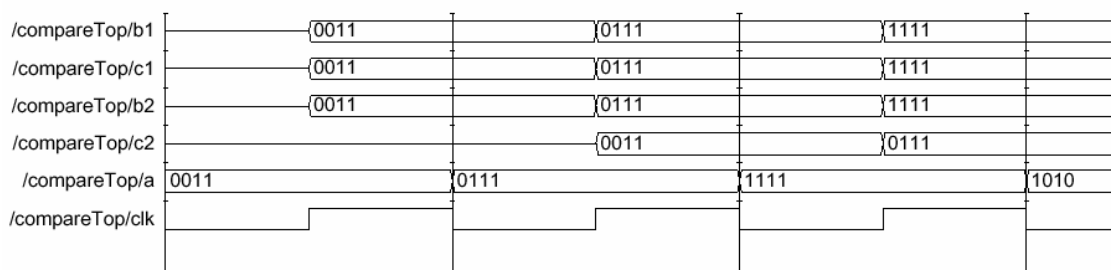
```
// -------------   blocking.v ----------------

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


//-------------  non_blocking.v --------------------
module non_blocking(clk,a,b,c);

  output [3:0] b,c;
  input  [3:0] a;
  input        clk;
  reg    [3:0] b,c;

  always @(posedge clk)
  begin
    b <= a;
    c <= b;
    $display("Non_Blocking: a = %d, b = %d, c = %d.",a,b,c);
  end
```

```
endmodule
```

测试模块源代码：

```
//-------------  compareTop.v ------------------------------

`timescale 1ns/100ps
`include "./blocking.v"
`include "./non_blocking.v"

module compareTop;

  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
  non_blocking  non_blocking(clk,a,b2,c2);
  blocking      blocking(clk,a,b1,c1);

endmodule
```

仿真波形（部分）：



思考：在 blocking 模块中按如下写法，仿真与综合的结果会有什么样的变化？作出仿真波形，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;
```

练习五. 用 always 块实现较复杂的组合逻辑电路

目的: 1.掌握用 always 实现组合逻辑电路的方法;
2.了解 assign 与 always 两种组合逻辑电路实现方法之间的区别。

仅使用 assign 结构来实现组合逻辑电路,在设计中会发现很多地方会显得冗长且效率低下。而适当地采用 always 来设计组合逻辑，往往会更具实效。已进行的范例和练习中，我们仅在实现时序逻辑电路时使用 always 块。从现在开始，我们对它的看法要稍稍改变。

下面是一个简单的指令译码电路的设计示例。该电路通过对指令的判断，对输入数据执行相应的操作，包括加、减、与、或和求反，并且无论是指令作用的数据还是指令本身发生变化，结果都要作出及时的反应。显然，这是一个较为复杂的组合逻辑电路，如果采用 assign 语句，表达起来非常复杂。示例中使用了电平敏感的 always 块，所谓电平敏感的触发条件是指在@后的括号内电平列表中的任何一个电平发生变化，(与时序逻辑不同，它在@后的括号内没有沿敏感关键词，如 posedge 或 negedge）就能触发 always 块的动作，并且运用了 case 结构来进行分支判断，不但设计思想得到直观的体现，而且代码看起来非常整齐、便于理解。

```
//---------------  alu.v ----------------------------
`define plus    3'd0
`define minus   3'd1
`define band    3'd2
`define bor     3'd3
`define unegate 3'd4

module  alu(out,opcode,a,b);
```