数字系统设计

1．数字系统的设计过程

数字系统的设计可分为几个层次：系统级设计、电路级设计、芯片级设计和电路板级设计。

系统设计的主要任务是将设计要求转换为明确的、可实现的功能和技术指标，确定可行的技术方案，并在系统一级（顶层）进行功能和技术指标的描述。

通常通过对系统功能的模块划分进行系统功能和技术指标的分配，并确定各功能模块之间的接口关系。系统设计通常把系统功能逐步细分，运用框图与层次化的方法自顶向下进行设计，再对器件、电路等确定技术方案。

电路设计主要是确定实现系统功能的算法和电路形式，在电路级对系统的功能进行描述。

芯片设计则通过对芯片的设计、编程，实现电路设计所确定的算法和电路形式，即设计专门用途的集成电路芯片。

以往采用标准的 TTL、CMOS 电路系列及功能固定的专用集成电路进行设计时，用户只能根据系统设计的要求去选择器件，而不能定义或修改器件的逻辑功能。由于可编程逻辑器件可以由用户对其功能进行设计、对其引脚进行配置，这就可以采用基于芯片的方法进行系统设计，给设计者带来了很大方便。设计者可根据系统设计的功能模块划分，把功能模块放到芯片中进行设计，从而用单片或几片大规模可编程逻辑器件实现系统的主要功能。

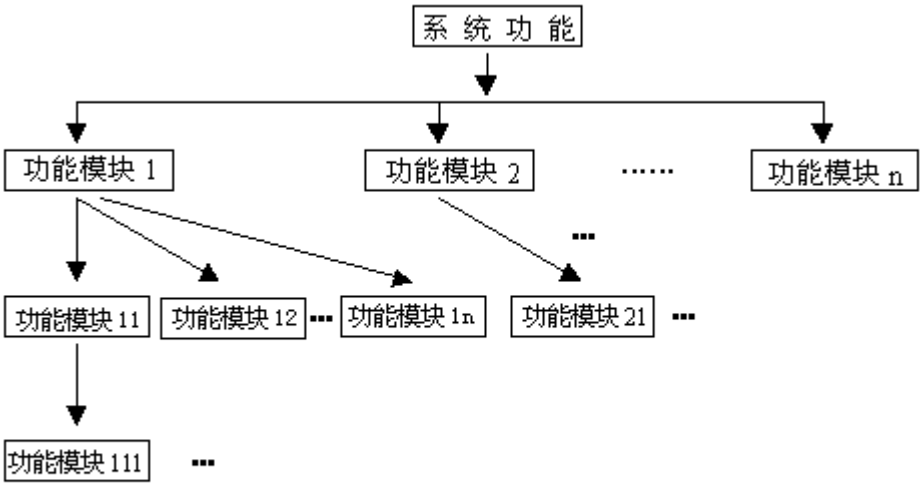
PCB 设计（电路板设计）是在芯片设计的基础上，通过对芯片和其它电路元件之间的连接，把各种元器件组合起来构成完整的电路系统；并依照电路性能、机械尺寸、工艺等要求，确定电路板的尺寸、形状，进行元器件的布局、布线。通常可借助 PCB 设计软件完成（如 Protel 等）。

数字系统设计还必须进行电路调试和系统调试。

可利用仪器仪表进行电路调试，检查器件、电路、PCB 设计是否有错误，测试电路板的功能是否达到设计要求。而系统调试则应检查各电路板间的接口是否符合设计规范，系统的整体功能、性能指标是否达到设计要求。在电路调试或系统调试中查出错误、发现问题时，应返回到器件设计、电路设计或 PCB 设计，修改出错的器件、电路或电路板设计；然后再进行调试，直至系统的功能、性能指标达到设计要求。

2．数字系统设计的基本方法

设计数字系统有多种方法，如自底向上设计法、自顶向下设计法、模块设计法等，对于采用高密度可编程逻辑器件进行设计的系统，通常使用自顶向下（即 TOP—DOWN）的设计方法，这也是基于芯片的系统设计的主要方法。这种方法在功能划分、任务分配、设计管理上为设计者提供了方便。



自顶向下设计法框图

自顶向下设计法通过功能分割手段将系统由上而下分层次、分模块进行设计和仿真。高层次设计主要进行功能和接口描述，定义模块的功能和接口，模块功能的进一步描述在下一层次说明，最底层的设计才涉及具体逻辑门和寄存器等实现方式的描述。这里的“模块”可以是芯片，也可以是电路板。高层次设计可以和具体的器件无关，可采用逻辑仿真手段检验设计是否正确。这种方法可用于系统的前期设计，在没有选定器件、做出电路系统之前，可用软件的仿真手段验证方案的可行性。在这基础上，选择器件、进行设计综合、定时仿真，做出具体的电路系统。

自顶向下设计法有很多优点，它作为一种模块化设计方法，对设计的描述由上而下、由粗略到详细，符合人们常规的思维习惯；而高层设计与器件无关，又便于将设计结果在各种集成电路或可编程器件之间移植；模块化的方式，也便于由多个设计者同时进行同一系统的设计。

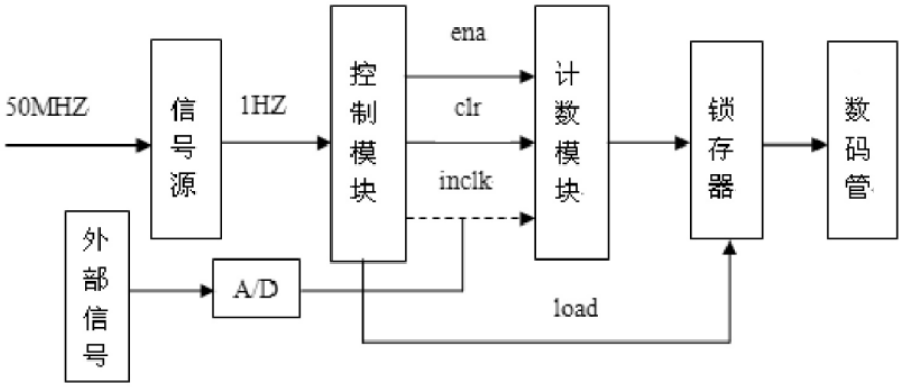
应用自顶向下设计法进行设计时，应注意做到逐层分解功能、分层次进行设计；而在各设计层次上，还应考虑相应的仿真验证问题。

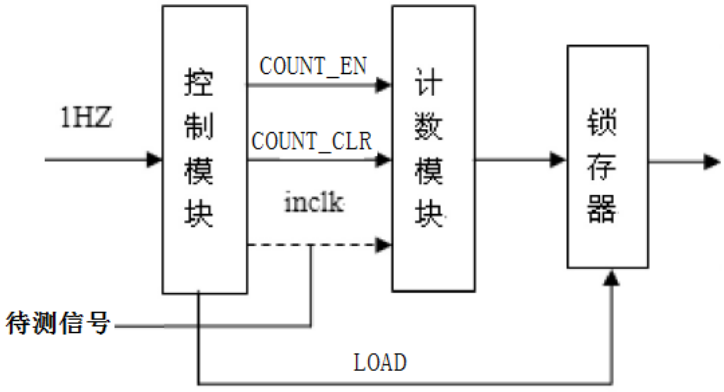
数字系统设计实例

4位简易数字频率计

一、功能与原理

采用一个标准的基准时钟,在单位时间(如1秒)里对被测信号的脉冲数进行计数,即为信号的频率。





(1). 控制模块

控制模块的作用是产生测频所需的各种控制信号。

标准输入时钟为1Hz，每两个时钟周期进行一次频率测量。

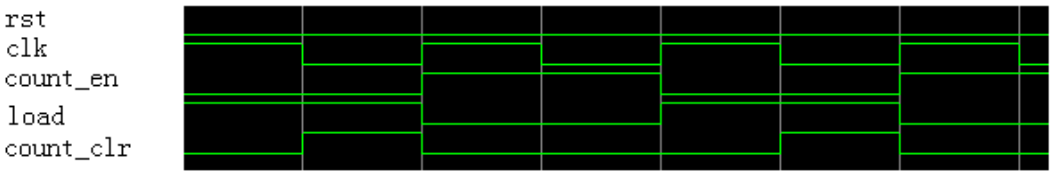
产生3个控制信号，分别为COUNT_EN、COUNT_CLR和LOAD。

COUNT_CLR：在每一次测量开始时对计数模块复位，清除上次测量结果；复位信号高电平有效，持续半个时钟周期。

COUNT_EN：为计数允许信号，在COUNT_EN信号上升沿时刻，计数模块开始对输入被测信号的脉冲进行计数，时间为一个时钟周期（1s），即为信号的频率。

LOAD：将被测信号频率的测量值存入寄存器，送数码管显示。

控制信号的时序关系:



(2). 寄存器模块

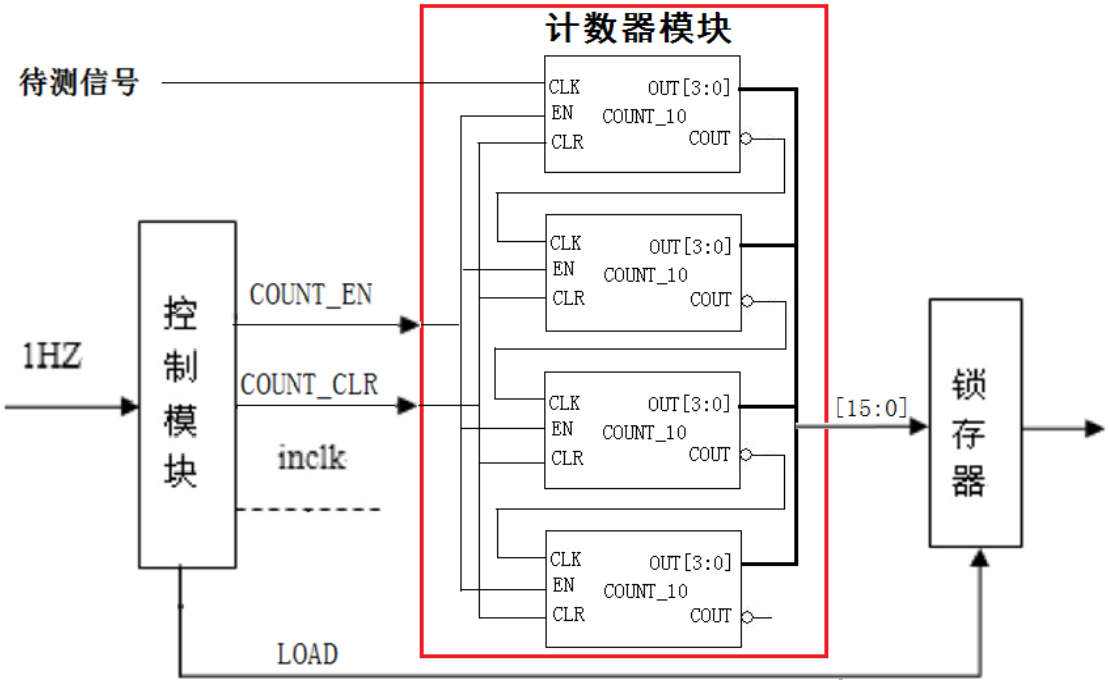
每次测频结束，在LOAD信号的上升沿将测量值入寄存器中。

(3). 计数模块

在单位时间内对待测信号的脉冲进行计数，该模块必须有计数允许、异步清零、时钟输入、数据输出等端口。

二、设计实现

四位频率计的顶层原理图



分别设计各底层模块，并设计顶层模块，以完成系统设计。

(1). 4 位数字频率计控制模块

```
module fre_ctrl(clk,rst,count_en,count_clr,load);
output count_en,count_clr,load;
input clk,rst;
reg count_en,load;

always @(posedge clk)
  begin
    if(rst) begin count_en=0; load=1; end
    else begin
         count_en=~count_en;

         load=~count_en; //load 信号的产生

         end
  end
```

```
assign count_clr=~clk&load; //count_clr 信号的产生
endmodule
```

(2). 4 位数字频率计计数子模块

```
module count10(out,cout,en,clr,clk);
output[3:0] out;
output cout;
input en,clr,clk;
reg[3:0] out;

always @(posedge clk or posedge clr)
  begin
    if (clr) out = 0; //异步清0
    else if(en)
         begin
          if(out==9) out=0;
          else out = out+1;
         end
  end
assign cout =((out==9)&en)?0:1; //产生进位信号
endmodule
```

(3). 频率计锁存器模块

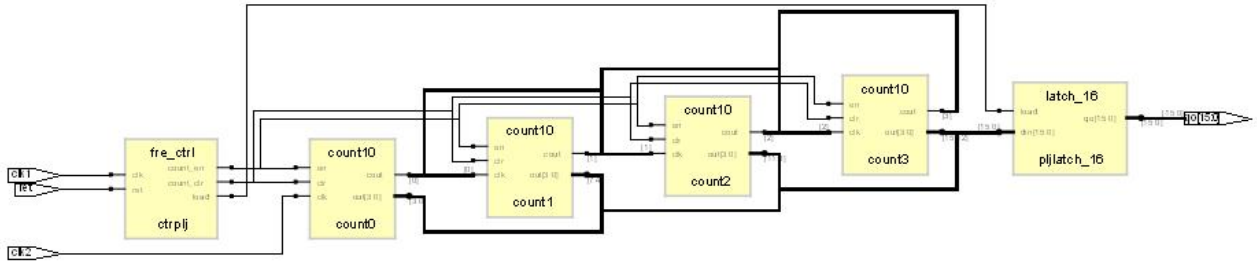
```
module latch_16(qo,din,load);
output[15:0] qo;
input[15:0] din;
input load;
reg[15:0] qo;
always @(posedge load)
   begin qo=din; end
endmodule
```

(4). 频率计顶层模块

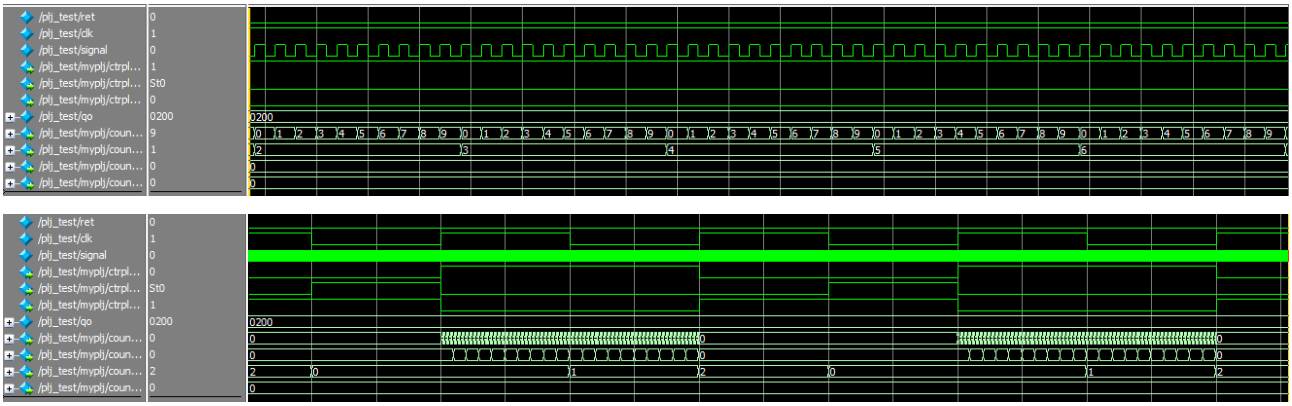
```
module plj (clk,ret,signal,qo);
output [15:0]   qo;
input   clk,ret,signal;
wire   [15:0]   qo;

wire count_en,load;
wire[15:0] count_out;
wire[3:0] count_cout;
wire count_clr;
```

```
    fre_ctrl ctrplj(clk,ret,count_en,count_clr,load);
    count10 count0(count_out[3:0],count_cout[0],count_en,count_clr,signal);
    count10 count1(count_out[7:4],count_cout[1],count_en,count_clr,count_cout[0]);
    count10 count2(count_out[11:8],count_cout[2],count_en,count_clr,count_cout[1]);
    count10 count3(count_out[15:12],count_cout[3],count_en,count_clr,count_cout[2]);
    latch_16 pljlatch_16(qo,count_out,load);
endmodule
```

在 **synplify** 上综合，得到相应的电路图:

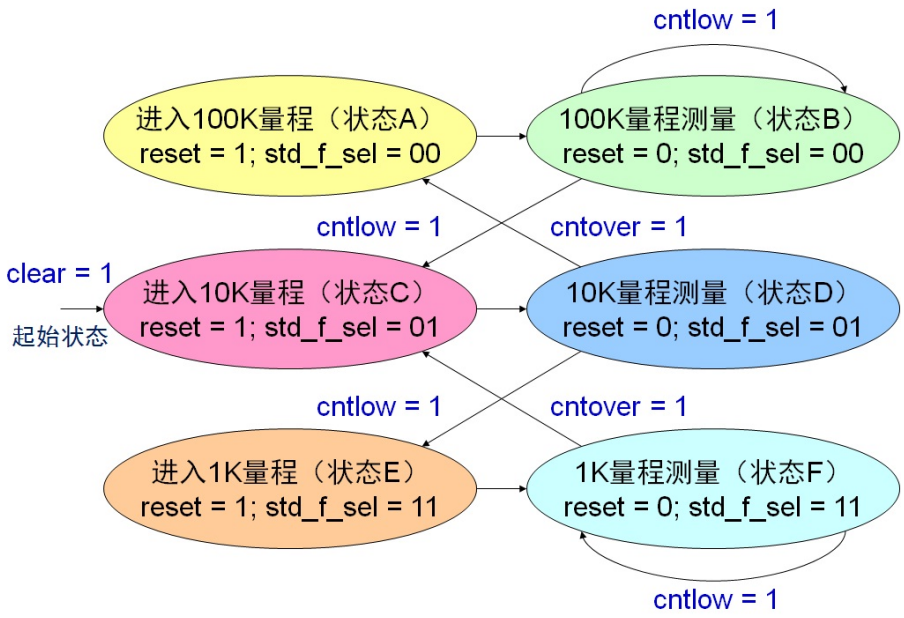


在 **modelsim** 上仿真，可得到相应波形:



自动转换量程频率计控制器

设计一个自动转换量程的频率计控制器，用状态机实现。
下图是该频率计控制器的状态转换图：




```
/*信号定义：
clk：  输入时钟；
clear：  为整个频率计的异步复位信号；
reset：  用来在量程转换开始时复位计数器；
std_f_sel：  用来选择标准时基；
cntover：  代表超量程；
cntlow：  代表欠量程。
状态 A，B，C，D，E，F 采用独热码编码 */
module control(std_f_sel,reset,clk,clear,cntover,cntlow);
output[1:0] std_f_sel;
output reset;
input clk,clear,cntover,cntlow;
reg[1:0] std_f_sel;
reg reset;
reg[5:0] present,next; //用于保存当前状态和次态的中间变量
parameter start_f100k=6'b000001, //状态A 编码，采用独热码编码
          f100k_cnt=6'b000010, //状态B
          start_f10k=6'b000100, //状态C
          f10k_cnt=6'b001000, //状态D
          start_flk=6'b010000, //状态E
          flk_cnt=6'b100000; //状态F

always @(posedge clk or posedge clear)
  begin
```

```
    if(clear) present<=start_f10k; //start_f10k 为起始状态
    else present<=next;
  end
always @(present or cntover or cntlow)
  begin
    case(present) //用case 语句描述状态转换
    start_f100k: next<=f100k_cnt;
    f100k_cnt:
      begin
        if(cntlow) next<=start_f10k;
        else next<=f100k_cnt;
      end
    start_f10k: next<=f10k_cnt;
    f10k_cnt:
      begin
        if(cntlow) next<=start_f1k;
        else if(cntover) next<=start_f100k;
          else next<=f10k_cnt;
      end
    start_f1k: next<=f1k_cnt;
    f1k_cnt:
      begin
        if(cntover) next<=start_f10k;
        else next<=f1k_cnt;
      end
    default:next<=start_f10k; //缺省状态为起始状态
  endcase
end
always @(present) //该进程产生各状态下的输出
  begin
    case(present)
    start_f100k: begin reset=1; std_f_sel=2'b00; end
    f100k_cnt: begin reset=0; std_f_sel=2'b00; end
    start_f10k: begin reset=1; std_f_sel=2'b01; end
    f10k_cnt: begin reset=0; std_f_sel=2'b01; end
    start_f1k: begin reset=1; std_f_sel=2'b11; end
    f1k_cnt: begin reset=0; std_f_sel=2'b11; end
    default: begin reset=1; std_f_sel=2'b01; end
    endcase
  end
endmodule
```

在 **synplify** 上综合，得到相应的电路图: