Verilog RTL 代码新手上路教程

从编写基础模块开始

用绳量给我的地界

坐落在佳美之处

我的产业实在美好

杜伟韬

duweitao@cuc.edu.cn

广播电视数字化教育部工程研究中心

http://ecdav.cuc.edu.cn

2013年 6月 30日于 北京定福庄

献给广播学院的核桃林，还有我的老师们

目 录

0 修订记录和意见征求

2013年，6月末-7月初，从wiki版本整理修订成TEX版本。

2011年，5月末-2012年，7月，从word版本整理并修订为wiki版本

2010年，2月初，编写word格式版本

当前征求意见问题：本文中的参考代码是保留行号，还是去掉行号？

征集网页文档修订志愿者：本文档是从TEX生成的PDF，目前wiki网页版本内容落后于本文档，现面向数字化工程中心研究生征集志愿者，协助本文作者同步更新wiki版本的教程，有意向者请联系杜伟韬。

1 引言

数字电路设计是人类信息化浪潮的重要组成技术之一，正是由于数字集成电路设计与制造行业的进展，人类可以制造出愈加强大的处理器，再辅以EDA软件，以更加强大的设计辅助能力回馈给电路研发、制造行业，从而形成了促进生产力增长的正反馈链条，本文从若干常用的简单数字电路模块开始，给出其Verilog代码范例，这些常用模块在数字电路设计中的地位和乘法九九表在四则运算中的地位类似，通过熟练掌握这些模块的设计方法，包括修改并重用这些代码，在经过设计实践并具有了系统化的思维方式之后，学生可以设计出更为复杂的数字系统。

千里之行，始于足下，九层之台,起于垒土，本文中的电路模块将会是你日后设计的复杂数字系统中最基本的砖头与瓦片，除了熟练掌握这些最基本零部件的使用方法之外，另外建议阅读一些你在科研中遇到的专用芯片的数据手册，分析其中给出的电路结构图，并且请思考，如果要你在FPGA上面重现如此的电路，你该如何下手，另外，推荐阅读一些关于处理器体系结构方面的书籍，要知道，人类所设计的最复杂的数字电路，非处理器莫属，从处理器体系结构中，我们可以借鉴出非常多的设计思想，这是几代天才们研究、总结出的精华，它们将与其设计者的名字一起被载入教科书中，一代接着一代的传承。

2 技术综述

2.1 用FPGA/CPLD做些什么

对于没有接触过FPGA和EDA设计流程的读者，可以这样认为，“FPGA中包含了一块巨大的面包板，其中有大量的门电路和触发器被预先安置在上面，用户使用EDA工具来对这些电路进行连线，最终得到目标功能”。实际上，基于FPGA的电路设计流程是现代集成电路设计流程的前半部分，即专用集成电路ASIC和FPGA电路的设计流程存在部分的类似，两者的区别在于ASIC的设计者需要自行设计芯片内部的模块布局和绕线，而FPGA中的逻辑模块和布线资源都是预先放置好的。

因此，可以这样认为，FPGA可以完成绝大多数的数字电路能够完成的功能，从FPGA的用户群体来看，根据以往的统计，第一大应用是接口逻辑，其次是数字通信和信号处理，然后是电机控制和工业自动化。

2.2 Verilog 还是VHDL

Verilog和VHDL是EDA行业常用的两种语言，其各有千秋，相比之下，VHDL更加侧重逻辑，更加抽象，适用于描述复杂电路的行为，而Verilog则更加底层、更加具体，适合作为底层的描述语言。可以说，Verilog之于VHDL好比计算机行业中的C语言之于Pascal语言，本文推荐从Verilog开始入门EDA设计，因为其更加注重RTL（寄存器传输级）方面的概念，这样容易使得初学者建立起面向硬件的思维模式，作为学生，如果有机会，建议两者都学懂，因为从工业界来看，使用两种开发语言的团队都大有人在，所以为了提高就业几率，最好两者都搞懂。

2.3 电路原理图还是写代码

在使用EDA工具设计电路时，有两种常用方式，一种是用各种逻辑块拼图，另一种是写代码，作为EDA的初学者，电路拼图的方式简单易懂好上手，而代码的方式让人望而生畏，但是代码的方法其优势在于：*1*) 可以精确的描述电路，也可以批量的修改或生成一组信号。*2*) 有着良好的重用性，可以通过参数化的方式较为方便的修改信号位宽及常数定义。由于代码的方式没有拼图的方式来的直观，因此，通常在工程上，顶层的工程设计使用拼图的方式实现，而从第二层开始的模块通常使用代码实现。

2.4 需要培养怎样的能力

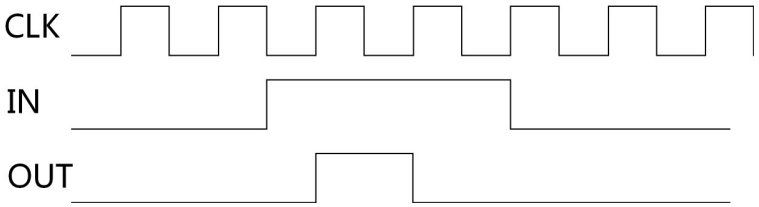
对于使用硬件描述语言（HDL）来设计集成电路逻辑而言，一个有经验的集成电路逻辑设计人员会拥有以下的能力来完成设计。

- 设计功能的需求分析：把需要完成的电路功能明确下来，定义出电路的顶层信号端口以及各个端口的时序。对于复杂的电路而言，需求分析的阶段显得尤为重要，因为复杂电路的修改成本会很高，一旦功能需求没有正确设定，则后面的设计和调试阶段所投入的大量时间成本均会被浪费。
- 设计的模块分割：把顶层的模块切分成为若干个小型的模块，同样，定义出各个小型模块的端口和时序，这一步骤是比较考验水平的，因为：*1*) 好的模块划分可以做到非常好的“设计正交性”以及各个模块之间的“去耦”。*2*) 例如当某一个子模块的内部设计出现错误时，如果模块划分做的好（这体现在各个子模块的接口和时序定义上）只需要修改出错的子模块内部即可，不需要改动接口定义和其它的子模块。*3*) 或者，当需要对整个电路进行功能升级的时候，只需要改动有关系的子模块即可。
- 实现各个子模块：*1*) 对于已经分割出来并且定义好功能和接口的子模块，其设计复杂度已经不是很高，对于有经验的人员可以非常熟练的编写其代码。*2*) 之所以能够做到这一点是由于设计者事先掌握了一系列的有代表性的“积木块”的代码编写方法。*3*) 在不同的设计项目中，这些“积木块”的功能尽管不是完全一样，但是却会非常的相似。*4*) 作者编写本教程的目的就是在于让初学者掌握一批有意义的“积木块”的编写方法，然后在其它的项目中熟练的使用这些“积木块”的改装版本。

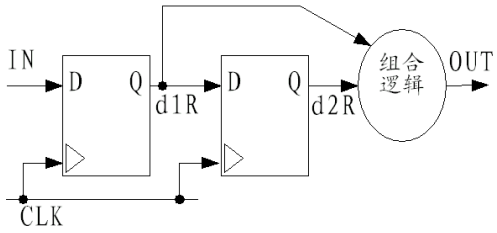
3 想清楚 RTL 结构的重要性

无论是VHDL还是Verilog，它们均属于“硬件描述语言”，即，它们是用来描述硬件电路的，换言之，开发者使用这些语言，在较高层次描述硬件的结构，然后由EDA工具转换成底层的描述，目前成熟的EDA工具都是将RTL（寄存器传输级）层次描述的硬件转换为在逻辑门层次描述的电路。所谓RTL的含义就是D触发器之间穿插着组合逻辑。对于组合逻辑，我们只需描述它输入和输出的关系表达式（用if—else和case语句）不必深究到底用怎样的逻辑门来实现，而对于D触发器，则是一定要做到心中有数，就是哪些变量会生成D触发器。

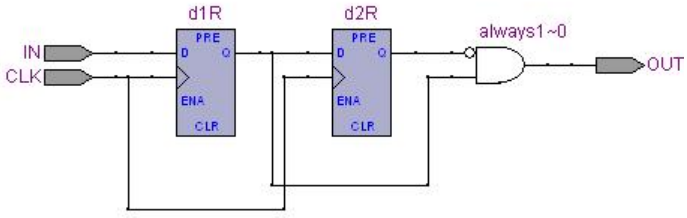
举例来说，一个在CLK时钟驱动下，对输入信号进行上跳沿捕获的电路，其接口时序如图1(a)所示，该电路用于探测输入信号的上跳并输出一个单时钟周期宽度的有效脉冲，该电路RTL结构如图1(b)所示，需要注意的是本处的示例电路只能用在单一时钟域中，它的Verilog代码如下所示，经过EDA工具进行编译之后，会提取出如图1(c)所示的RTL结构，我们可以观察EDA工具解析出的RTL是否和我们预想的一致，这是一种重要的验证手段。对于Quartus工具，请选择Tools - Netlist Viewer - RTL Viewer。



(a) 边沿捕获电路接口时序



(b) 电路规划的RTL结构



(c) Quartus工具生成的RTL结构

图 1: RTL结构对照

以下是边沿捕获电路的代码，该代码中使用两个always 块描述了两部分电路，一个是移位寄存器的时序逻辑，另一个是用于判断信号上跳的组合逻辑，其中时序逻辑部分指定了需要有2个D触发器，以及它们之间的信号传递关系，EDA工具会按照该描述生成电路并且严格保证D触发器的数量和互联关系，而组合逻辑部分代码使用表达式描述了信号之间的关系，EDA工具在保证这个关系成立的前提下，自行生成逻辑门，这个过程类似于数字电路课本中在真值表上使用卡诺图来设计门电路。

```
1  // 边沿捕获电路代码
2  module top(
3    CLK       ,   // input clock
4    IN        ,   // input
```

```
  OUT       );  // output
input  CLK , IN;
output OUT     ;
// 电路中的D触发器输出端
reg  d1R, d2R;
// 组合逻辑输出信号，作为输出端口
reg  OUT;
//生成移位寄存的D触发器，时序逻辑对时钟上升沿敏感。
always @ (posedge CLK) begin
  d1R <= IN  ;
  d2R <= d1R ;
end
// 判断上跳沿的组合逻辑，新值为1，旧值为0，跳变发生。
always @ (d1R or d2R) begin
  if((d2R == 0)&&(d1R == 1))
    OUT = 1'b1;
  else
    OUT = 1'b0;
end
endmodule
// endmodule top
```

4 常用verilog模块设计范例

4.1 多路选择器

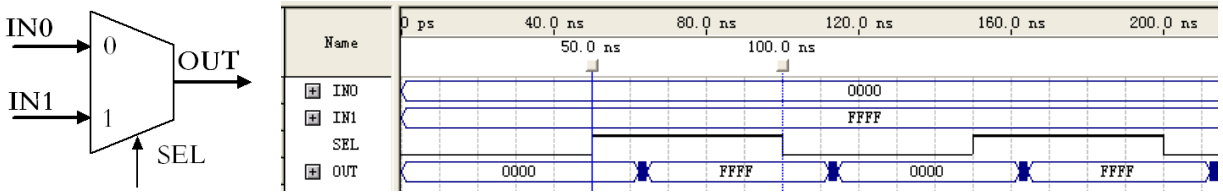
4.1.1 电路描述

该电路为纯组合逻辑，根据选通信号的值，把输入信号之一连接到输出信号上。电路应用时可能的变化：*1*) 数据信号的宽度，包括输入数据宽度和选通信号宽度。*2*) 选通逻辑的变化，控制信号为0或1时选通哪个通道。

该电路符号如图2(a)所示，电路经过Quartus编译后，时序仿真结果如图2(b)所示，从图中可以看到在SEL信号电平变化之后，OUT信号需要经过一段延时才变化，另外，在OUT信号的两个稳态数据之间，存在着小段的暂态过渡数据，这是因为多位信号之间存在着组合逻辑的竞争与冒险。

4.1.2 代码

```
// module  top, 选择器代码,
module top(
  IN0       ,   // input 1
  IN1       ,   // input 2
  SEL       ,   // select
```



(a) 电路符号　　(b) 波形仿真时序

图 2: 2输入选择器

```
  OUT          );  // out data
parameter WL = 16;          // 输入输出数据信号位宽
input [WL-1:0] IN0, IN1;    // 选择器的两个输入数据信号
input SEL;                  // 通道选通的控制信号
output[WL-1:0] OUT;         // 选择器的输入数据信号

reg    [WL-1:0] OUT;
// 生成组合逻辑的代码
always @ (IN0 or IN1 or SEL) begin
  if(SEL) // 为SEL1 选择输入1
    OUT = IN1;
  else    // 为SEL0 选择输入0
    OUT = IN0;
end
endmodule
// endmodule top
```

4.1.3 学生实验

(1) 做一个4选1的mux，并且进行波形仿真。

(2) 和2选1的mux对比电路资源开销。

(3) 修改信号的比特宽度和切换速率，观察输出信号中的竞争与冒险。

4.2 交叉开关

4.2.1 电路描述

对于复杂的信号切换，需要使用到交叉开关（crossbar）电路，这个电路类似一个交换机，每一个输出信号都可以被切换到任意一个输入信号上。该电路有以下特征：

1) 本质上是由多个MUX电路拼接构成。*2)* 通常用在复杂一些的信号选通的场合。*3)* 每一个输出端都有对应的选通信号。*4)* 用选通信号控制输出端选通到哪个输入端。*5)* 当输入和输出的端口增加时，该电路会消耗非常多的电路资源。

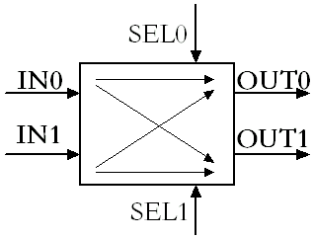


图 3: 2x2交叉开关矩阵

在使用该电路的时候可能会存在如下变化：*1*) 输入、输出信号的个数以及数据信号的宽度。*2*) 选通逻辑，即：输出端口序号、输入端口序号、选通信号数值之间的对应关系。

4.2.2 代码

```
// module  top, a 2x2 crossbar switch circuit
module top(
  IN0       ,   // input 1
  IN1       ,   // input 2
  SEL0      ,   // select the output0 source
  SEL1      ,   // select the output1 source
  OUT0      ,   // output data 0
  OUT1      );  // output data 1
parameter WL = 16;  // in, out data word length
input  [WL-1:0] IN0, IN1;
input SEL0, SEL1;
output[WL-1:0] OUT0, OUT1;

reg    [WL-1:0] OUT0, OUT1;
// get the OUT0 logic
always @ (IN0 or IN1 or SEL0) begin
  if(SEL0)
    OUT0 = IN1;
  else
    OUT0 = IN0;
end
// get the OUT1 logic
always @ (IN0 or IN1 or SEL1) begin
  if(SEL1)
    OUT1 = IN1;
  else
    OUT1 = IN0;
end
endmodule
// endmodule top
```

4.2.3 学生实验

(1) 编写一个4X4路交叉开关的RTL

(2) 该电路有4个输入、4个输出，每个输出配有2比特选择信号，用于将输出切换至任意一个输入信号

(3) 编译该电路，看RTL View。

(4) 比较2x2与4x4之间消耗资源的区别。通过对比资源，你有什么结论？

4.3 优先编码器

4.3.1 电路描述

所谓优先编码器是指对输入的多位比特信号进行检测，然后以发生置位的、优先级最高的那个比特的编号作为输出。例如，CPU的中断编码电路就是一个优先编码器，当多个中断事件发生时，中断编码器会以优先级最高的那个事件的编号作为中断向量。*1*) 纯粹的组合逻辑电路。*2*) 常用于输入信号状态检测。*3*) 优先的含义是，如果多个条件同时成立，则按照优先级高的条件输出。*4*) 使用if-else if 语句实现。

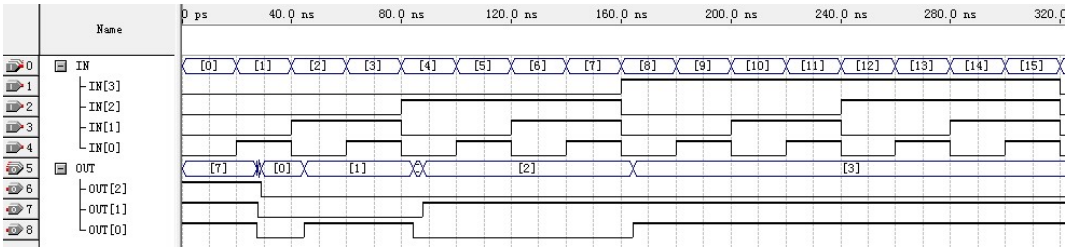


图 4: 优先编码器仿真时序

图4给出了一个4比特输入，3比特输出的优先编码器的时序仿真结果，4比特输入信号的优先级从信号0到4依次升高，输出信号的值表示输入信号中优先级最高的、发生置位（值为1）信号，另外，如果没有信号置位，则输出信号用全1的数值表示，因此，输出信号需要3比特的位宽。

4.3.2 代码

```
1  // module top, 4 input priority encoder with zero input check
2  module top(
3    IN          ,   // input
4    OUT         );  // output
5  input [3:0] IN;
6  output[2:0] OUT;
7
8  reg   [2:0] OUT;
9  // get the OUT
10 always @ (IN) begin
```

```
    if(IN[3])          // 第一优先
      OUT = 3'b011;
    else if(IN[2])     // 第二优先
      OUT = 3'b010;
    else if(IN[1])     // 第三优先
      OUT = 3'b001;
    else if(IN[0])     // 第四优先
      OUT = 3'b000;
    else               // 无信号发生置位
      OUT = 3'b111; // 输出值可自定义，不和上面的输出值混淆即可
end
endmodule
```

4.3.3 学生实验

(1) 编写一个8输入的优先编码器，然后编译，看RTL View

(2) 和4输入的优先编码器对比电路资源

(3) 对电路进行时序仿真，调整输入信号的变换速率，观察组合逻辑的竞争与冒险情况

4.4 多路译码器

4.4.1 电路描述

数字电路课本中的3—8译码器就是多路译码器中的一种，这种电路的作用通常是用来作为器件或模块的选通信号，多路译码器有以下特征：*1*) 纯粹的组合逻辑电路。*2*) 使用case 语句实现，注意，一定要把case的情况写全。*3*) 如果case的情况没写全，则一定要加上default。

实际应用中，该电路可能会有以下变化：*1*) 编码信号的位宽，通常为输入N比特信号，输出为2^N比特信号。*2*) 编码的逻辑，即输入信号的编码格式和输出信号的置位格式（0有效或1有效）。

4.4.2 代码

```
// module top, 3-8 decoder
module top(
  IN          ,   // input
  OUT         );  // output

input  [2:0] IN;
output [7:0] OUT;

reg    [7:0] OUT;
// get the OUT
always @ (IN) begin
```

```
  case(IN)
    3'b000: OUT = 8'b0000_0001;
    3'b001: OUT = 8'b0000_0010;
    3'b010: OUT = 8'b0000_0100;
    3'b011: OUT = 8'b0000_1000;
    3'b100: OUT = 8'b0001_0000;
    3'b101: OUT = 8'b0010_0000;
    3'b110: OUT = 8'b0100_0000;
    3'b111: OUT = 8'b1000_0000;
    //   full case 不需要写default，否则一定要写
  endcase
end
endmodule
```

4.4.3 学生实验

(1) 编写一个4—16的译码器，编译后观察RTL View的电路结构。

(2) 和3—8译码器对比资源开销

(3) 进行波形仿真，设置不同速率的输入信号，观察输出信号的延迟以及竞争冒险。

4.5 无符号数加法器

4.5.1 电路描述

加法器是很常用的电路，根据输入数据格式的不同，可分为2种，包括：*1)* 无符号加法器，输入和输出数据都是无符号的整数。。*2)* 补码加法器，输入和输出数据都是2补码形式的有符号数。其中无符号加法器的用途广泛，常用于计数器累加、计算地址序号以及一些常规表达式的计算。数字电路课本中提到的使用多个1比特全加器级联成为多比特加法器的结构，称为行波进位（ripple-carry）加法器，这种结构的优点是电路资源开销较小，缺点是进位链延迟较大，不适合用于高速计算的场合，由此，诸多改进的加法器结构被提出，例如其中比较有名的超前进位（Carry Look Ahead）加法器，以及进位保留加法器（Carry Save Adder）。随着EDA工具的进展，当前的HDL代码编译工具已经可以根据电路的时序约束条件自动选择加法器的结构，而设计者仅需要通过一个“加号”来告知“此处需要使用加法器”。

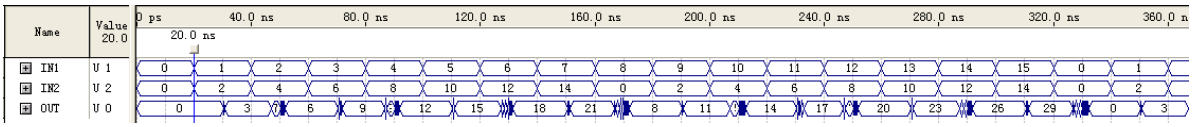


图 5: 无符号加法器仿真时序

图5为无符号加法器的仿真波形，其中IN1和IN2信号为4比特无符号数，OUT信号为5比特无符号数。经观察可发现加法器输入数据和对应结果在时间上存在延迟，这是组合逻辑门的延迟造成的（请估算一下延

迟时间长度）。另外可见，相邻的两个正确的输出结果之间存在“毛刺”，且“毛刺”的时间长度各不相同。请思考：

(1) 2比特的信号，00翻转成11，两个位能做到绝对的同时翻转么？对比一下，00翻转成11，和00翻转成10，哪个变化的多？哪个翻转过程中的过渡状态多？

(2) 注意输入和输出的信号位宽数。如果本例中，加法器的2个输入信号和1个输出信号宽度都是4比特，那么输出结果会一直正确么？在什么情况下会发生错误？

4.5.2 代码

注意，在Verilog语法中，没有声明为“有符号”格式的信号，其默认格式是无符号格式。

```
module top(
  IN1    ,
  IN2    ,
  OUT    );
input[3:0] IN1, IN2;
output[4:0] OUT;
reg[4:0] OUT;
always@(IN1 or IN2) begin // 生成组合逻辑的always 块
  OUT = IN1 + IN2;
end
endmodule
```

4.5.3 学生实验

(1) 把加法器的输出信号改成4比特位宽，编译，波形仿真。观察输出结果，说出输出和输入的对应关系，并指出什么时候输出会出错，出错的结果和理论上的正确是怎样的关系？

(2) 把加法器的输入信号改成8比特位宽，此时的输出信号应该是几个比特？编译，波形仿真。观察加法器的输出延迟，和4比特输入位宽的情况对比，哪个延迟大？为什么？

4.6 补码加法器

4.6.1 电路描述

补码加法器主要用于计算有符号数，即，使用二进制补码格式来表示的整数，从逻辑门电路的角度观察，补码加法器和无符号加法器的电路逻辑是不同的，但是用户可以通过使用”signed”关键字来描述加法器的输入数据格式为有符号数，则EDA工具检测到该信息后，会自动生成了补码加法器的电路逻辑。图6给出了一个补码加法器的仿真时序图，其中输入信号IN1和IN2均为4比特有符号数，输出信号OUT为5比特有符号数。

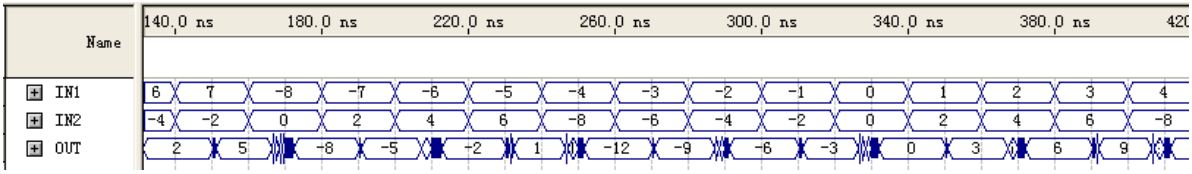


图 6: 补码加法器仿真时序

观察此波形图时，请留意以下几点：*1*) 补码加法器的输入数据和对应结果在时间上同样是有延迟的，这是组合逻辑的延迟。（请估算一下延迟长度）。*2*) 波形图上的数据是带负号的，这是通过选择信号一属性一Radix为有符号的十进制来观察的结果。

4.6.2 代码

```
module top(
  IN1    ,
  IN2    ,
  OUT    );
input signed [3:0] IN1, IN2;
output signed [4:0] OUT;
reg signed [4:0] OUT;
always@(IN1 or IN2) begin // 生成组合逻辑的always 块
  OUT = IN1 + IN2;
end
endmodule
```

4.6.3 学生实验

(1) 请思考，对于同样的二进制比特数据，我们可以用不同的Radix观察它，这说明了什么？

(2) 加法器的输入信号改成8比特位宽，此时的输出信号应该是几个比特？编译，波形仿真。观察加法器的输出延迟，对比4比特的延时情况。

(3) 使用有符号数观察加法器的输入数据，使用除了2补码之外，其它的Radix观察加法器的输出数据，你有什么结论？

4.7 流水线加法器

4.7.1 电路描述

所谓流水线，就是用一前一后两组D触发器把组合逻辑包起来。从之前的时序仿真图中可以看到，由于竞争冒险的存在，组合逻辑电路的输出信号中存在着不正确的暂态数据，如何避免这种暂态错误数据干扰后级电路，解决方法就是在组合逻辑电路的后级添加D触发器，只要在时钟的上升沿来到的时刻D触发器的输入是正确数据，则其输出数据就可以保证正确，则D触发器后级的组合逻辑的输入数据也能够保证正确。

图7给出了一个流水线加法器的RTL视图。 纯粹的加法器是由大量的组合逻辑门构成，如果加法器的位宽较

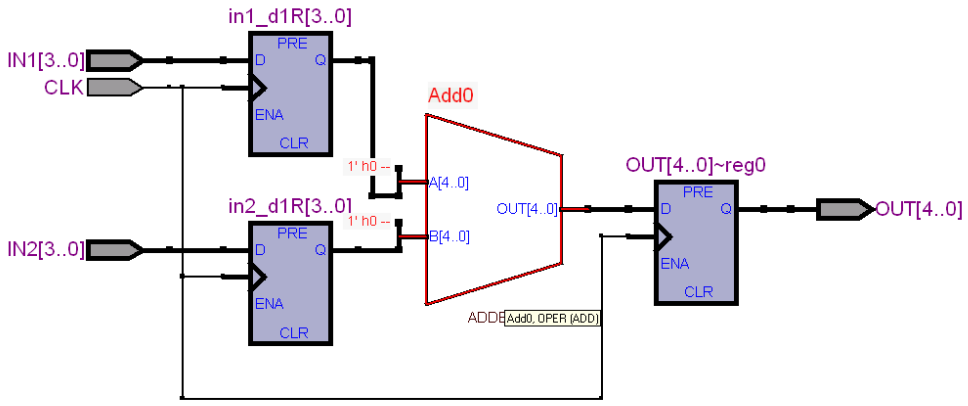


图 7: 无符号流水线加法器RTL视图

大，则这些组合逻辑的计算延迟较大，如果加法器电路的后级电路也是一个规模较大的组合逻辑，那么它们会和加法器电路合并成为一个更大的组合逻辑，从而带来更大的组合逻辑计算延迟。因此，使用D触发器将组合逻辑隔离起来，可以避免这种延迟的积累。

流水线技术可以让电路工作在更高的时钟频率，对于使用了流水线的电路，每一个D触发器都有其所容许的最小的建立与保持时间，所谓建立时间，是指当时钟上升沿来到之前，输入数据必须先稳定一段时间T_{su}，然后D触发器才能正确将其锁存，D触发器对于这个T_{su}的数值有下限要求，同样，所谓保持时间，是指当时钟上升沿来到之后，输入数据必须再稳定一段时间T_{hd}，D触发器对于这个T_{hd}的数值也有下限要求。

当两个D触发器之间的组合电路逻辑延迟变得更大的时候，会导致组合逻辑后级的D触发器输入延迟增大，从而导致建立时间裕度下降，因此需要增加电路的时钟周期，导致其只能工作在更低的时钟频率。为了让电路能够工作在更高的时钟频率，需要用D触发器来把大块的组合逻辑分割为小块，这就是流水线技术。（建议自行搜索关键字“静态时序分析”）

图8 给出了无符号流水线加法器的仿真结果，由图中可以见到，从信号数据输入到加法结果输出，存在2个时钟周期节拍的延迟，另外，在时钟上升沿来到的时刻，所有的数据均处于稳定状态，这保证了信号在各级时序和组合逻辑之间的正确传输。

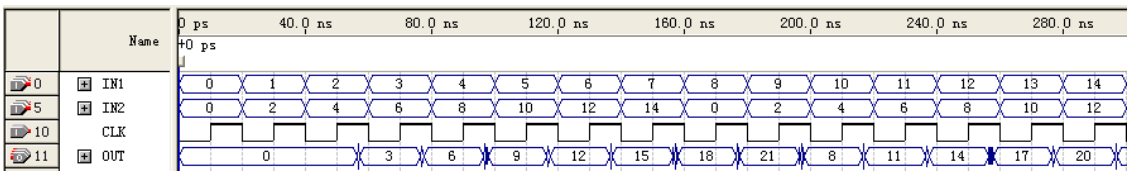


图 8: 无符号流水线加法器波形仿真结果

4.7.2 代码

```
module top(
  IN1    ,
  IN2    ,
  CLK    ,
  OUT    );
input  [3:0] IN1, IN2;
input CLK;
output  [4:0] OUT;
reg [3:0] in1_d1R, in2_d1R;
reg  [4:0] adder_out, OUT;
always@(posedge CLK) begin   // 生成D触发器的always块
  in1_d1R <= IN1;
  in2_d1R <= IN2;
  OUT     <= adder_out;
end
always@(in1_d1R or in2_d1R) begin // 生成组合逻辑的always 块
  adder_out = in1_d1R + in2_d1R;
end
endmodule
```

4.7.3 学生实验

(1) 不改变流水线的级数，把加法器的输入信号改成8比特位宽，编译，波形仿真，和不带流水线的情况对比一下，你有什么结论？

(2) 在8比特输入位宽的情况下，在输入上再添加一级流水线，观察编译和仿真的结果，你有什么结论？

(3) 加快输入信号的时钟频率和输入数据的速率，观察在什么频率下，计算结果会出错？

4.8 乘法器

4.8.1 电路描述

首先要强调的是，乘法器是一种奢侈品，会消耗大量的组合电路逻辑资源，一定要慎重使用。尤其是对于芯片内部没有硬件乘法器的FPGA芯片，需要更加慎重。为什么会消耗如此多的资源呢，请回想一下我们在草稿纸上手工计算多位数乘法的过程，我们需要做很多次的加法，用电路计算乘法也是类似的。

乘法器的代码和加法器类似，存在无符号乘法器和有符号补码乘法器，同样，选择电路结构这种细节工作是由EDA工具完成的，用户通过使用一个乘号“*”来告知工具“此处需要使用乘法器”。另外，乘法器也可以添加流水线以提高运行频率。

图9 给出了一个2输入4比特有符号1输出8比特的乘法器时序波形仿真结果，可以看到其中的组合逻辑延迟和毛刺远多于加法器，在使用乘法器时，注意以下几点：*1*) 对于一些简单常数乘法，可以用加法

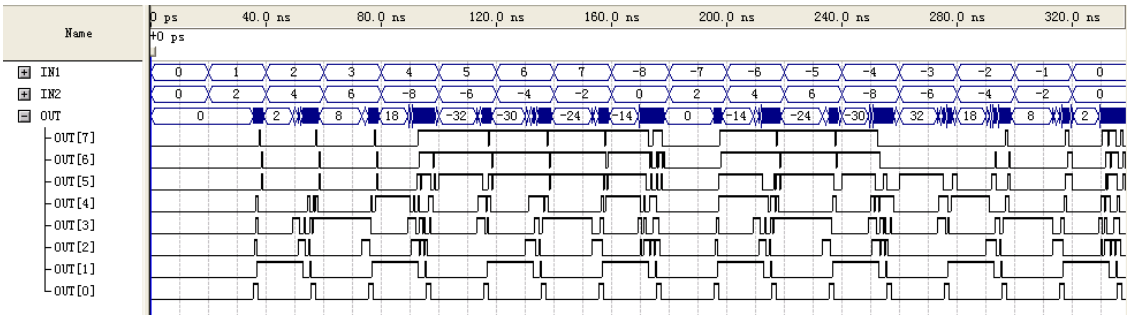


图 9: 有符号乘法器波形仿真结果

和移位来替代乘法，例如$A*3 = A*2 + A = (A << 1) + A$，最终消耗1个加法器和1个移位器。*2)* 对于补码乘法器，计算结果会扩展出1比特额外的符号位，从波形图中可以看到，最高2个比特总是一样的。*3)* 有些FPGA芯片内部有固化的硬件乘法器，对于这样的芯片，代码中的乘号“*”会变成芯片中的乘法器。*4)* 对于内部没有乘法器的FPGA芯片，代码中的乘号就会用逻辑单元拼接成加法器，再用加法器拼接成乘法器。

4.8.2 代码

```
// 无符号乘法器
module top(
  IN1     ,
  IN2     ,
  OUT     );
 input [3:0] IN1, IN2;
 output [7:0] OUT;
 reg [7:0] OUT;
always@(IN1 or IN2) begin // 生成组合逻辑的always 块
  OUT = IN1 * IN2;
end
endmodule
//  有符号的2补码乘法器
module top(
  IN1     ,
  IN2     ,
  OUT     );
 input signed[3:0] IN1, IN2; //signed关键字
 output signed [7:0] OUT;
 reg signed[7:0] OUT;
always@(IN1 or IN2) begin // 生成组合逻辑的always 块
  OUT = IN1 * IN2;
end
endmodule
```

4.8.3 学生实验

(1) 改变乘法器的输入位宽为8比特，编译，波形仿真，观察信号毛刺的时间长度。

(2) 选一款没有硬件乘法器的FPGA芯片（例如Cyclone EP1C6）对比8比特的乘法器和加法器两者编译之后的资源开销(Logic Cell的数目）。

(3) 选一款有硬件乘法器的FPGA芯片（例如CycloneIII EP3C16）对比8比特的乘法器和加法器两者编译之后的资源开销(Logic Cell 和DSP Block的数目）。

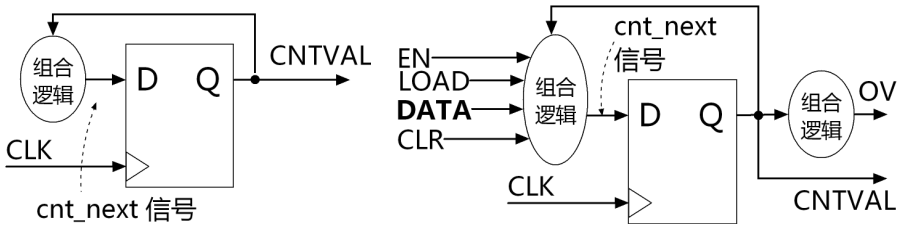
(4) 编写一个输入和输出都有D触发器的流水线乘法器代码，编译后波形仿真，观察组合逻辑延迟和毛刺的时间，和不带流水线的情况下对比。

4.9 计数器

4.9.1 电路描述

计数器是最为常用的时序电路之一，计数器在数字电路里面的作用，就如C程序中的for循环一样。在实际电路设计中，计数器的应用非常广泛，比如信号的同步、分频与延时，存储器读写地址的生成等多种场合。

图10(a)绘出了最简单的计数器，只有一个CLK信号和一个计数值Q信号，在实际应用中的计数器可能会出现较多的变化，例如图10(b)所示的带有各种端口，完成不同的逻辑功能的计数器，常用的功能有：*1*) 计数溢出功能，当计数到某个最大值MAX的时候，OV（OVerflow）信号输出1，否则输出0 *2*) 计数使能功能，当输入使能EN信号为1的周期，计数器的Q值会有更新的动作，否则保持不动*3*) 计数同步清零功能，当输入的CLR信号为1的周期，在下一个周期Q端清零。*4*) 计数同步加载功能，当输入的LOAD信号为1的周期，DATA信号被选通至触发器的D端，在下一个周期传递至Q端。



(a) 简单计数器　　(b) 多功能计数器

图 10: 计数器电路结构

以上的功能是有重叠的，根据实际应用不同，EN信号、CLR信号、LOAD信号的优先级可能会不同，例如某个电路需要在EN无效的时候D触发器彻底被锁死，清零信号和同步加载信号的值对D触发器没有影响，而另一个电路可能会要求清零信号的优先级更高。实际应用中，需要根据不同的需求用不同的代码来描述（主要是用if-else的嵌套，参考上文中的优先编码器例子）。由此可以见到使用代码来描述电路

的优势所在，即，一些细节的调整可以通过修改代码精确的完成。图11是一个带有计数使能（EN）、清零（CLR）、计数值加载（LOAD）功能的计数器仿真结果，其中EN、CLR、LOAD三者的优先级逐次降低，待加载数据为DATA信号，计数值为CNTVAL信号。

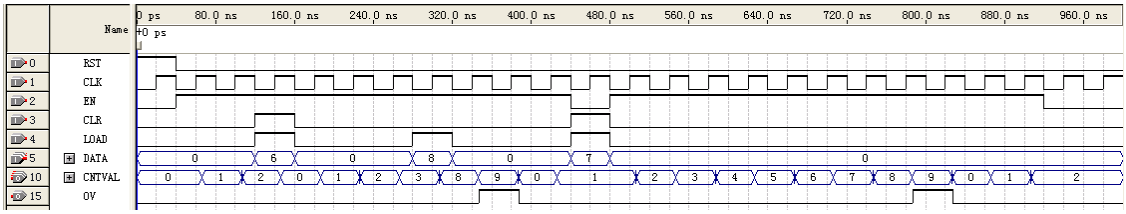


图 11: 多功能计数器波形仿真结果

4.9.2 代码

以下给出2份代码，分别是简单计数器和多功能计数器的HDL代码。

```
//简单计数器代码
module top(
  CLK    ,          // clock
  RST    ,          // reset
  OUTVAL);          // output counter value

parameter OUTWL = 4; // output wordlength
parameter CNTMAX = 11;  // counter max value

input             CLK, RST;
output signed [OUTWL-1:0] OUTVAL;
// DFF reg
reg   [OUTWL-1:0] cntR;
// combo logic
reg   [OUTWL-1:0] cntNext;

// update DFF
always @ (posedge CLK or posedge RST)
begin
  if(RST)  begin
    cntR  <= 0;
  end
  else    begin
    cntR <= cntNext;
  end
end
// get combo logic
always @ (cntR)  begin
```

```
29    if(cntR < CNTMAX)
30      cntNext = cntR + 1'b1;
31    else
32      cntNext = 0;
33  end
34  assign OUTVAL = cntR;
35  endmodule
36
37  // 多功能计数器代码
38  module top(
39    RST    , // 异步复位，高有效
40    CLK    , // 时钟，上升沿有效
41    EN     , // 输入的计数使能，高有效
42    CLR    , // 输入的清零信号，高有效
43    LOAD   , // 输入的数据加载使能信号，高有效
44    DATA   , // 输入的加载数据信号
45    CNTVAL, // 输出的计数值信号
46    OV     );// 计数溢出信号，计数值为最大值时该信号为1
47
48  input RST    , CLK    , EN     , CLR    , LOAD   ;
49  input [3:0] DATA ;
50  output [3:0] CNTVAL;
51  output OV;
52
53  reg [3:0] CNTVAL, cnt_next;
54  reg OV;
55  // 电路编译参数，最大计数值
56  parameter CNT_MAX_VAL = 9;
57
58  // 组合逻辑，计数使能最优先，清零第二优先，加载第三优先
59  always @(EN or CLR or LOAD or DATA or CNTVAL) begin
60    if(EN) begin     // 使能有效
61      if(CLR) begin // 清零有效
62        cnt_next = 0;
63      end
64      else begin  // 清零无效
65        if(LOAD) begin // 加载有效
66          cnt_next = DATA;
67        end
68        else begin      // 加载无效，正常计数
69          // 使能有效，清零和加载都无效，根据当前计数值计算下一值
70          if(CNTVAL < CNT_MAX_VAL) begin // 未计数到最大值，下一值加一
71            cnt_next = CNTVAL + 1'b1;
72          end
```

```
73          else begin // 计数到最大值，下一计数值清零
74            cnt_next = 0;
75          end
76        end // else LOAD
77      end  // else CLR
78    end // if EN
79    else begin  // 使能无效，计数值保持不动
80      cnt_next = CNTVAL;
81    end // else EN
82  end
83
84  // 时序逻辑更新下一时钟周期的计数值
85  // CNTVAL is a D-Flip-Flop
86  always @ (posedge CLK or posedge RST) begin
87    if (RST)
88      CNTVAL <= 0;
89    else
90      CNTVAL <= cnt_next;
91  end
92
93  // combo logic get OV
94  always @ (CNTVAL) begin
95    if (CNTVAL == CNT_MAX_VAL)
96      OV = 1;
97    else
98      OV = 0;
99  end
100 endmodule
```

4.9.3 学生实验

请完成以下设计实验，编译电路并且进行波形仿真。

(1) 观察图11，观察CNTVAL和LOAD、DATA、CLR、EN 等信号之间的逻辑关系。

(2) 设计一个带使能控制的计数器，在使能有效的时钟周期，计数器正常工作，每个时钟周期，计数值加1，从0到11然后再回到0，使能无效的时钟周期，计数器保持原值不动。

(3) 设计一个带使能和步进值加载的计数器，使能信号优先级最高，如果使能未开启，计数值保持不动。计数的最大值为10，计数的步进增量值可以加载，计数到了最大值的时候向0值回绕，例如，设当前计数值为9，计数步进值为3，则下一时刻的计数值为(9+3-10)=2。

4.10 状态机

4.10.1 电路描述

有限状态机“FSM（Finite State Machine）”同样是数字电路设计中常用的模块，其在EDA设计中的地位等同于C语言中的If−else语句，状态机的应用非常广泛，主要用于接口控制器，协议转换和处理，以及处理器设计，因为从电路的观点来看，处理器本身就是一个复杂的状态机。状态机的设计要点包括：

(1) 把要解决的问题映射到状态空间，包括：*a*) 定义有哪些状态，这一步非常重要，好的状态定义可以极大的简化电路逻辑；*b*) 定义状态之间的跳转逻辑是怎样的；*c*) 定义输出信号和状态值之间的逻辑关系。

(2) 使用状态图或表格来描述状态机的跳转逻辑和输出逻辑，其中状态图的方式较为直观，但是绘制起来复杂通常用于汇报演示的文档中，表格的方式更加严谨精确，并且容易编制，实际的工程文档中，表格的方式使用的更多。

(3) 使用parameter定义状态，为每个状态命名以明确其表达的含义。

(4) 尽量使用三段式的标准写法便于代码的维护和修改。

参考电路

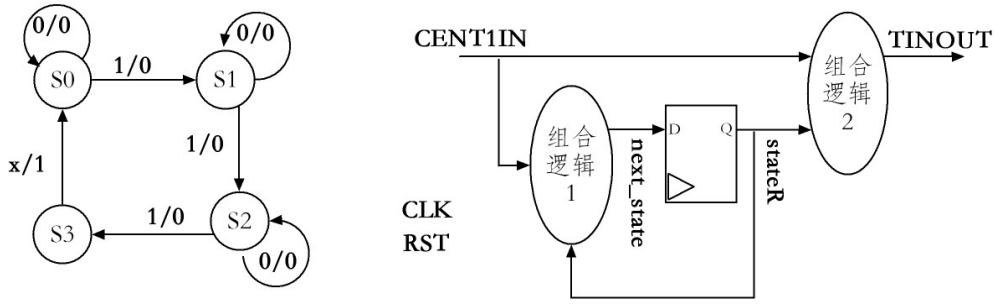


图 12: 可乐售卖机的状态迁移和RTL结构图

表 1: 状态跳转逻辑表

当前状态	输入	次态
ST_0_CENT	CENT1IN == 0	ST_0_CENT
ST_0_CENT	CENT1IN == 1	ST_1_CENT
ST_1_CENT	CENT1IN == 0	ST_1_CENT
ST_1_CENT	CENT1IN == 1	ST_2_CENT
ST_2_CENT	CENT1IN == 0	ST_2_CENT
ST_2_CENT	CENT1IN == 1	ST_3_CENT
ST_3_CENT	Donot care	ST_0_CENT

本例中，我们要实现一个可乐售卖机的状态机电路，假设可乐3分钱1罐，机器只接受1分钱的硬币，当投入第三个硬币的时候，投出一罐可乐，它的状态转移逻辑和电路RTL逻辑如图12所示，请注意以下要

表 2: 状态输出逻辑表

当前状态	输出（TINOUT）
ST_0_CENT	0
ST_1_CENT	0
ST_2_CENT	0
ST_3_CENT	1

点：*1)* 从电路的RTL结构上来看，状态机和计数器比较相似，但是它们的功能不同。*2)* 具体的状态跳转和输出逻辑如表1和表2所示。*3)* 当设计状态机时，强烈建议先把这两个表格画出来再开始写代码，这样编写代码的时候逻辑较为清晰。*4)* 从表2中可见，输出信号仅和状态值有关系，这是因为本例中定义了一个“3硬币”状态，一旦电路进入此状态后，则会在当前时钟周期输出1罐可乐，并且在下一时钟周期进入到“0硬币”状态。*5)* 如果采用不同的状态定义策略，比如最多仅定义“2硬币”的状态，则相应的状态转移表和输出逻辑表也需进行相应调整。

4.10.2 代码

此处展示的是三段式状态机代码，所谓三段式是指代码中包括：*1)* 一段组合逻辑的always块，用于根据当前状态和输入值计算次状态。*2)* 一段组合逻辑的always块，用于根据当前状态和输入值计算输出值。*3)* 一段时序逻辑的always块，用于更新状态寄存器（D触发器）。

```
1  // 三段式状态机代码
2  module top(
3    CLK         ,   // clock
4    RST         ,   // reset
5    CENT1IN     ,   // input 1 cent coin
6    TINOUT      );  // output 1 tin cola
7
8  input  CLK         ;
9  input  RST         ;
10 input  CENT1IN     ;
11 output TINOUT      ;
12
13 parameter ST_0_CENT = 0;
14 parameter ST_1_CENT = 1;
15 parameter ST_2_CENT = 2;
16 parameter ST_3_CENT = 3;
17
18 reg [2-1:0]stateR         ;
19 reg [2-1:0]next_state     ;
20 reg         TINOUT        ;
21
22 // 组合逻辑，计算次状态
23 always @ (CENT1IN or stateR) begin
```

```
24    case (stateR)
25      ST_0_CENT :begin if(CENT1IN) next_state = ST_1_CENT ; else next_state = ST_0_CENT; end
26      ST_1_CENT :begin if(CENT1IN) next_state = ST_2_CENT ; else next_state = ST_1_CENT; end
27      ST_2_CENT :begin if(CENT1IN) next_state = ST_3_CENT ; else next_state = ST_2_CENT; end
28      ST_3_CENT :begin next_state = ST_0_CENT; end
29    endcase
30  end
31
32  // 组合逻辑，计算输出
33  always @ (stateR) begin
34    if(stateR == ST_3_CENT)
35      TINOUT = 1'b1;
36    else
37      TINOUT = 1'b0;
38  end
39
40  // 时序逻辑，状态寄存器更新
41  always @ (posedge CLK or posedge RST)begin
42    if(RST)
43      stateR <= ST_0_CENT;
44    else
45      stateR <= next_state;
46  end
47  endmodule
```

对于代码编写的规范的状态机，EDA工具可以检测出来，例如Quartus中，编译代码之后，可以在Tools－Netlist Viewers－State Machine Viewer 里面看到你写的状态机的状态转移图和表达式。这对你调试电路非常有帮助。图13 是经过Quartus编译检测出的状态迁移图。

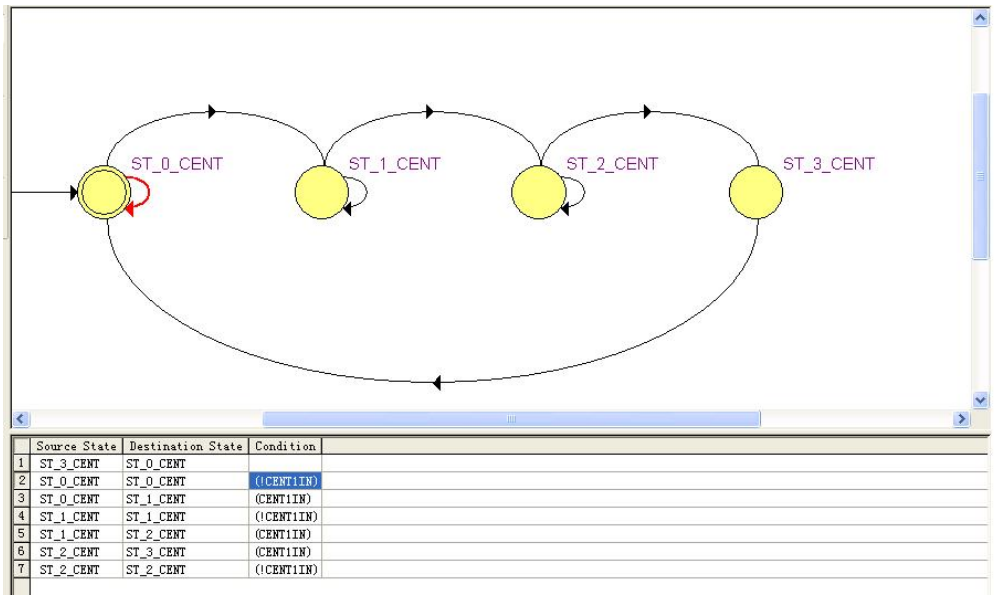
4.10.3 学生实验

设计一个用于识别2进制序列“1011”的状态机，包括：

(1) 电路每个时钟周期输入1比特数据，当捕获到1011的时钟周期，电路输出1，否则输出0

(2) 使用序列101011010作为输出的测试序列

(3) 扩展要求：电路添加输入使能端口，只有输入使能EN为1的时钟周期，才从输入的数据端口向内部获取1比特序列数据。



	Source State	Destination State	Condition
1	ST_3_CENT	ST_0_CENT	
2	ST_0_CENT	ST_0_CENT	(!CENT1IN)
3	ST_0_CENT	ST_1_CENT	(CENT1IN)
4	ST_1_CENT	ST_1_CENT	(!CENT1IN)
5	ST_1_CENT	ST_2_CENT	(CENT1IN)
6	ST_2_CENT	ST_3_CENT	(CENT1IN)
7	ST_2_CENT	ST_2_CENT	(!CENT1IN)

图 13: 可乐售卖机的状态迁移和RTL结构图

4.11 移位寄存器

4.11.1 电路描述

串行数据和并行数据之间的相互转换是在接口设计中很常见的功能，一般而言，数据在FPGA内部都是并行传递的，当通过串行接口协议（例如SPI，I2C，I2S等）把数据从FPGA内部传送到一个外部芯片（例如一片EEPROM存储器或是一片音频DAC）时就需要用到并串转换。反之，当数据从一个串行接口的芯片进入FPGA的时候，需要使用到串并转换电路，无论是串并或是并串转换，其核心的电路都是移位寄存器，区别在于寄存器的加载方式不同。

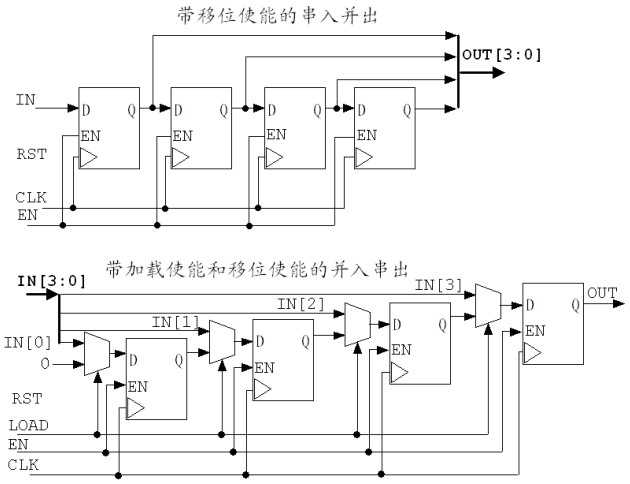


图 14: 移位寄存器RTL结构图

图14展示了串入并出和并入串出两种结构的移位寄存器，其中，串入并出结构的寄存器在使能信号的控制下，每个D触发器逐个从前级搬运信号，同时全部D触发器的输出信号作为并行信号输出。并入串出结构的移位寄存器通过通过LOAD信号控制加载，一旦加载有效，全部的D触发器从输入的IN信号分别加载各自对应比特的数据。另外，在不进行加载的周期，使能信号EN控制D触发器从前级搬运数据进行串行移位。

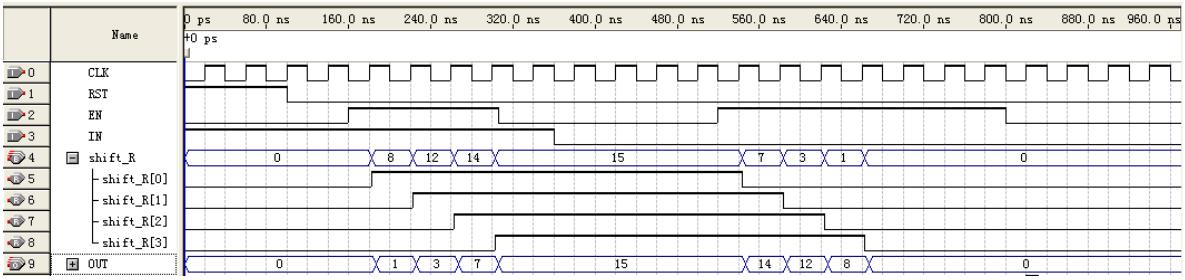


图 15: 串入并出移位寄存器仿真时序波形

图15给出了一个4比特，并入串出的移位寄存器仿真时序，在使能信号EN为高的周期，数据从高位串行向低位移动，在使能无效的周期，D触发器值保持不动。

4.11.2 代码

```
// 串入并出移位寄存器
module top(
  RST    ,    // 异步复位，高有效
  CLK    ,    // 时钟，上升沿有效
  EN     ,    // 输入数据串行移位使能
  IN     ,    // 输入串行数据
  OUT    );   // 并行输出数据

input RST, CLK, EN;
input IN;
output [3:0] OUT;
reg [3:0] shift_R;

assign OUT[3:0] = shift_R[3:0];
// 时序逻辑根据输入使能进行串行移位
// shift_R 会被编译为触发器D
always @ (posedge CLK or posedge RST) begin
  if(RST)
    shift_R[3:0] <= 0;
  else
    if(EN) begin // 串行移位的使能有效
      shift_R[3:1] <= shift_R[2:0];
```

```
23        shift_R[0]    <= IN;
24      end
25      else begin // 使能无效保持不动
26        shift_R[3:0] <= shift_R[3:0];
27      end
28  end // always
29  endmodule
```

4.11.3 学生实验

设计一个如本节“电路描述”部分的“带加载使能和移位使能的并入串出”的移位寄存器，其中加载信号的优先级最高，在加载信号有效的时钟周期，全部D触发器并行加载外部送入的并行数据，再加载信号无效，移位使能信号有效的周期，D触发器进行串行移位，从最高位向最低位串行移动，最高位补零，当所有使能信号都无效的周期，D触发器保持不动。

4.12 同步双口RAM

4.12.1 电路描述

现代的FPGA中，通常会固化一些RAM块，这些RAM块被称作双端口同步RAM，即RAM有两个端口，每个端口都有一套地址、数据、时钟信号，并且可以独立对RAM的内容进行读写，所谓同步是指每个端口的读写都需要有同步时钟，相应端口的地址，数据都需要用这个时钟进行同步。在实际应用时，通常要避免访问冲突，即一个端口写、一个端口读，并且要注意避免在同一个时刻从两个端口对同一个地址内容既写又读。这种双口RAM的两个端口的读写时钟可以是异步的，即写入时钟和读出时钟可以相互独立，没有特定的依赖关系，所以可以使用这种RAM在不同的时钟域之间传输数据或是构建一个FIFO模块（也是用于跨越时钟域传输数据），需要格外注意的是，在读写速率不一致的应用场景，需要预留足够的存储深度以防止RAM的上溢（写满）或下溢（读漏）。FPGA中的RAM块，通常可以根据用户的要求有一定的形变能力，以Altera芯片中的M4K RAM块为例，该RAM块的容量固定为4K比特，但是地址线宽度和数据线宽度在一定范围内可调节，比如可以配置为1bit数据4096个地址，2比特数据2048个地址，直至32比特数据128个地址。

在使用这些RAM块的时候，有两种方式，一种是使用Quartus工具提供的MegaWizard IP例化工具，该方法较为直观，具体请自行搜索。一种是使用Verilog代码描述RAM行为，由Quartus工具推测生成RAM（详情参考Quartus Handbook 的Implementing Inferred RAM部分），这种方法重用性较好，并且可以方便的在综合编译之前直接进行行为仿真，请参考后续代码部分。

4.12.2 代码

本例参考代码提供一个地址宽度，数据宽度均参数化的双口RAM，参数的缺省值设定为0，这种方法要求在例化该模块时强行指定参数值，其意义是防止例化模块时因为忘记指定例化参数，从而导致使用了不正确的缺省参数值。

```
module dpram(
  WE      ,   // write enable
  WCLK    ,   // write clock
  RCLK    ,   // read clock
  WA      ,   // write address
  RA      ,   // read address
  WD      ,   // write data
  RD      );  // read data
// external set param
parameter DATAWL    = 0;
parameter ADDRWL    = 0;
parameter C2Q       = 0;
  input                     WE, WCLK, RCLK;
  input  [ADDRWL-1:0]       WA, RA;
  input  [DATAWL-1:0]       WD;
  output [DATAWL-1:0]       RD;
  reg [DATAWL-1:0] RD;
  reg [DATAWL-1:0] mem [(1<<ADDRWL)-1:0];
  always @ (posedge WCLK) begin
    if(WE)
      mem[WA] <= #C2Q WD;
  end
  always @ (posedge RCLK) begin
    RD <= #C2Q mem[RA];
  end
endmodule   // module dpram()
```

4.12.3 学生实验

(1) 选一个拥有片内RAM的FPGA型号，编译代码，看看综合结果（资源消耗报告，RTL View和Tech View）

(2) 在编译器的综合选项中，寻找“自动RAM识别”的开关，关闭这个开关，看看综合结果变成什么。

(3) 选一个片内没有RAM的CPLD型号，编译代码，看看综合结果。

(4) 选一个拥有片内RAM的FPGA型号，改变双口RAM例化时的参数（包括地址和数据位宽，尝试改的很大和很小）编译代码，看看综合结果。

你得到什么结论？

4.13 同步ROM

4.13.1 电路描述

在Altera的FPGA中的ROM是使用RAM块实现的，因为FPGA在上电时可以对其中的内容进行初始化，因此可以让一块同步RAM表现出与同步ROM完全一致的电路行为。同样在使用这些ROM时，有两种方式，MegaWizard IP例化工具和Verilog代码的行为模型。

本例给出了一个同步的ROM参考代码，输入信号为时钟、地址、输出信号为数据（注意，ROM和RAM不能有复位信号）请注意：在描述ROM时，最好把所有的地址case写全，比如7比特的地址，就从0写到127，即使你只使用其中的一部分（比如100个地址）实际电路中也会消耗掉2的幂个地址，因为RAM块物理地址个数都是2的幂。把地址写全有助于编译工具进行工作，否则对于一些大型的ROM（比如几千个地址），地址不写全编译工具有可能会死机。通常，大型的rom代码是自动生成的，你可以使用一个C语言程序或者MATLAB程序自动生成这些地址和数据的对应代码（别忘记fprintf函数）。

4.13.2 代码

```
1  // module top, a synchronized rom
2  module top(
3    CLK    ,          // clock
4    RA     ,          // read address
5    RD     );         // read data
6  input           CLK;
7  input   [6   :0] RA;
8  output  [12  :0] RD;
9  reg     [12  :0] RD;
10 always @ (posedge CLK)
11   case(RA)
12      7 'd 0      : RD = #1 13'd 0      ; // 0x0
13      7 'd 1      : RD = #1 13'd 101    ; // 0x65
14      7 'd 2      : RD = #1 13'd 201    ; // 0xC9
15      7 'd 3      : RD = #1 13'd 301    ; // 0x12D
16      //    ...  ...  ...
17      7 'd 123    : RD = #1 13'd 8176   ; // 0x1FF0
18      7 'd 124    : RD = #1 13'd 8181   ; // 0x1FF5
19      7 'd 125    : RD = #1 13'd 8185   ; // 0x1FF9
20      7 'd 126    : RD = #1 13'd 8189   ; // 0x1FFD
21      7 'd 127    : RD = #1 13'd 8190   ; // 0x1FFE
22   endcase
23 endmodule
```

4.13.3 学生实验

(1) 选一个拥有片内RAM的FPGA型号，编译代码，观察综合结果（资源消耗报告，RTL View和Tech View）

(2) 在编译器的综合选项中，寻找“自动ROM识别”的开关，关闭这个开关，观察综合结果变成什么。

(3) 选一个片内没有RAM的CPLD型号，编译代码，观察综合结果。

(4) 自己设计一个C代码或是Matlab代码，该代码能够实现以下功能：指定ROM的数据宽度和地址宽度，自动生成一个ROM的代码，ROM中的内容是一个存放在C或者Matlab数组中的无符号整数。

4.14 寄存器组和多重例化

4.14.1 电路描述

本例是比较深入的应用，对于重复例化结构相似的模块，以及设计结构上带有深度流水线的电路时，可以使用verilog提供的for循环和generate语句，使用这些语法特性的时候需要注意：

1) for和generate的每一次循环，都会生成电路，所以如果使用不慎，会生成庞大的电路。*2*) 一定要把verilog的for循环和C语言中的for循环区别开，这不是软件代码，而是电路，这里的描述是一种空间上的并行展开。

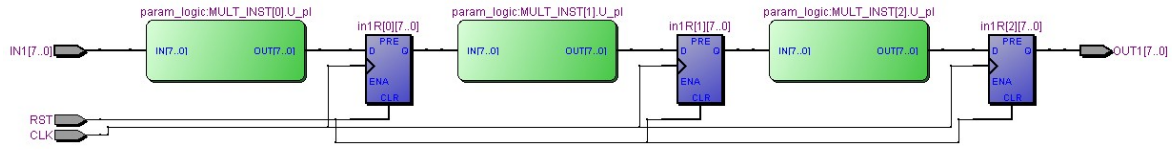


图 16: 一个使用generate和for语句生成的电路RTL视图

4.14.2 代码

```
// module top, example for, 'for' and 'generate'
module top(
  CLK    ,
  RST    ,
  IN1    ,
  OUT1   );
parameter DWL = 8;
parameter reg_len = 3;
input CLK;
input RST;
input   [DWL-1:0] IN1;
```

```
12  output [DWL-1:0] OUT1;
13
14  reg  [DWL-1:0] in1R[reg_len-1:0];
15  wire [DWL-1:0] in1R_inW[reg_len-1:0];
16  integer idx;
17  always @ (posedge CLK or posedge RST) begin
18    if(RST) begin
19      for(idx = 0; idx < reg_len; idx = idx +1) begin
20        in1R[idx] <= 0;
21      end
22    end
23    else begin
24      for(idx = 0; idx < reg_len; idx = idx +1) begin
25        in1R[idx] <= in1R_inW[idx];
26      end
27    end
28  end
29
30  genvar i_g;
31  generate
32  for (i_g =0; i_g < reg_len; i_g=i_g+1) begin: MULT_INST
33    wire [DWL-1:0] logic_inW;
34    if(i_g == 0) begin
35      assign logic_inW = IN1;
36    end
37    else begin
38      assign logic_inW = in1R[i_g-1];
39    end
40    param_logic U_pl(
41      .IN     (logic_inW      ),
42      .OUT    (in1R_inW[i_g] ));
43    defparam U_pl.DWL      = DWL;
44    defparam U_pl.INST_P1 = i_g;
45  end
46  endgenerate
47  assign OUT1 = in1R[reg_len-1];
48  endmodule
49
50  module param_logic(
51    IN     ,
52    OUT    );
53  parameter DWL      = 0;
54  parameter INST_P1 = 0;
55  input  [DWL-1:0] IN   ;
```

```
56 output [DWL-1:0] OUT ;
57 assign OUT = IN + INST_P1;
58 endmodule
```

4.14.3 学生实验

如果你需要设计CIC滤波器，建议先手工编写CIC滤波器代码，然后可以尝试使用for和generate再次描述该电路，注意观察电路的RTL view 并通过仿真验证电路运行的正确性。

5 后记

作者水平有限，编此拙文，仅为抛砖引玉，希望这些粗浅的文字不至于误导读者，更希望能对您步入电子学的有趣世界提供些许帮助，同时，作为一名在高校的研究院所工作的教师，作者也衷心的希望喜爱电子技术和数字通信、信号处理的本科同学报考我们数字化工程中心的研究生，这里拥有积极向上的研究氛围并且同时注重算法理论研究和工程实验样机研制两方面的工作，因此这里的研究生在电子学、通信与信号处理学科的各方面能力和素质均能得到良好的训练，从而毕业时拥有良好的竞争实力和就业、深造去向，详情请参考实验室主页上“毕业去向”部分。

这里是：中国传媒大学 广播电视数字化教育部工程研究中心，在校内简称数字化工程中心，英文缩写是ECDAV，实验室的网址是 http://ecdav.cuc.edu.cn 。我们是一个教育部直属的工程研究中心，请参考 http://baike.baidu.com/view/2391206.htm 中的第53号单位。

长久以来，我们致力于数字无线广播领域的研究，参与了相关领域多项国家标准的方案设计、样机研制、外场实验以及标准的起草，同时，我们也致力于专用领域的远程无线数据通信研究，这是因为数字广播中所采用的技术非常适合应用在数百公里大尺度距离上的无中继数据传输，比如在突发的地质灾害现场，能够进行远程通联的设备往往只有短波电台和卫星，其它需要中继基站和路由的设备很多都不能发挥作用。

正因如此，在个人移动通信和无线互联网如此繁荣和吸引眼球的今天，数字广播仍然在无线通信领域拥有属于它的重要位置，同时，我们也知道，无论是南海的堡礁或是北疆的哨所，亦或是深山村落里的油灯下，还是灾区简易房的褥褓里，祖国，需要我们，而我们，需要你。