数字集成电路设计与实现

第 1 章 绪论

1.1 数字集成电路的特点

数字电路通常是由简单的单元电路构成的规模庞大的系统，体现了“简单性”与“复杂性”的对立统一。基本的数字单元电路，如各种逻辑门电路和触发器、锁存器等，其电路结构比较简单，且实现的逻辑功能与其中晶体管尺寸无关。数字电路的性能指标相对较少，主要包括速度、功耗、面积三个方面，设计思路比较简单。但是，一个数字电路系统通常是非常复杂的，可能包含数百万个基本逻辑单元，其逻辑功能也需要有其它领域的知识才能理解。具有存储功能的数字逻辑单元，其输入信号和控制信号需要满足一定的时序关系才能正确实现逻辑功能。在达到一定规模后，各个单元电路的时序要求很难同时满足。制造工艺的进步，对数字电路性能提高作用显著。同样的设计，用特征尺寸更小的工艺实现，各方面都性能会有很大提高。因此，数字电路设计需要有较好的可移植性或重用性，以适应制造工艺的发展。数字电路的这些特点，决定了其设计技术的发展方向。

1.2 现代数字电路设计方法

在早期的集成电路设计中，数字电路与模拟电路的设计方法没有什么区别，都是**全定制设计。**全定制设计是一种晶体管级的设计，任何电路都要描述为由晶体管构成的电路网络。由于晶体管与版图之间具有明确的对应关系，这种设计方法的实现步骤相对较少，对 EDA 工具的依赖程度相对较低。在全定制设计问题中，设计者可以任意确定每个单元电路的结构和其中晶体管的尺寸，理论上讲，能够实现最优化的电路性能。由于具有较高的灵活性和设计自由度，全定制设计至今仍是模拟电路和规模较小的混合信号电路的设计方法。但是，对于规模庞大的数字电路来说，这种设计方法不仅设计工作量大，而且对电路的时序关系验证也十分困难，对于规模达到百万、千万晶体管的电路，完全采用全定制设计是不现实的。

现代数字集成电路设计方法来自对传统设计方法的总结和对计算机软件技术的引入。在对电路性能的要求没有达到工艺极限时，不需要对每个单元电路都进行特殊设计，可先设计出各种基本单元电路（cell），包括原理图和版图，形成

一个标准单元库，再利用库中的 cell 实现复杂的逻辑关系。这种方法使得自动化设计成为可能。即使在全定制设计中，对电路功能的描述也是层次化的，而不是直接描述为晶体管网络。有了标准单元库，一个电路就可以理解为由标准单元构成的网表，也就是说“描述级别”由晶体管级提升到了**“门级”。**复杂的电路在“门级”仍然难以看出逻辑功能，需要更高级别的描述。一个复杂的数字电路系统可以理解为由若干个具有典型逻辑功能的模块和一个控制电路组成的，常见的模块包括寄存器、计数器、算术和逻辑运算单元和存储器等。控制电路是一个有限状态机，在时钟的作用下，**状态机根据当前的状态和输入信号不断地进行状态转换，同时产生输出信号，控制各个逻辑模块工作。**这种级别的描述称为“架构级”，是最重要的设计级别。最高级别的描述是系统级，在这个级别，一般只定义系统的功能、外部接口和其中主要功能模块。最低级别的描述是版图，版图实际上就是一组几何图形，根据版图可以生成光刻版。一个设计实际上总是要从系统级开始考虑，然后是架构级、门级、晶体管级，最后是版图级。在传统的设计流程中，**可验证的设计描述是从晶体管级开始的**，用晶体管构成逻辑门，再由逻辑门构成功能模块，最后连接成系统。全部设计工作都要由设计者来完成，尽管也使用 EDA 工具，但这些工具只是代替了纸、笔和计算器，不能自动生成任何东西。

计算机技术的发展，使得从高级别的描述自动生成低级别的描述成为可能，这个过程与**从高级语言编写的程序生成机器码的过程相似**。用电路图描述复杂电路是很困难的，也很难被计算机理解，于是产生了硬件描述语言。硬件描述语言具有很强的描述能力，一段几百行的代码，有时可以代替几百张图纸。硬件描述语言诞生之初是为了保存设计或进行功能仿真，用于生成电路是后来的事。硬件描述语言也存在级别的概念，分为行为级、寄存器传输级（RTL 级）、门级、晶体管级等。版图的细节很难用语言描述，目前还不能自动生成，因此在所谓的自动化设计流程中，**标准单元库中的基本单元的版图还是手工设计的。**

在现代的 ASIC 设计方法中，标准单元库是由芯片制造厂提供的，设计者只需要用硬件描述语言写出对电路功能的描述，再用 EDA 工具的脚本语言，写出对电路性能的要求，大部分工作将由 EDA 工具实现。这种方法极大地提高了设计速度，也提高了设计的重用性。本文主要介绍这种设计方法，按业界习惯说法，

简称为 ASIC 设计方法。

ASIC 设计方法可以归纳为两部分工作，即设计和实现。设计指描述和验证电路功能，这部分工作需要由设计者完成。另一部分工作是实现的流程，在这部分工作中，设计者的任务是将对电路性能的要求，按 EDA 工具规定的形式，提供给 EDA 工具，用 EDA 工具自动生成基于 cell 的电路网表、版图和各种性能报告，最后确定设计的交付。**用 EDA 工具自动实现的设计工作称为“综合”**，从硬件描述语言**得到由 cell 网表的工作称为逻辑综合**，由 cell 网表自动生成版图的工作称为**物理综合**。设计过程的主要任务是给出**可综合的描述**，并验证其逻辑功能的正确性。

对于复杂的设计，电路功能需要通过多个级别的描述来完成。数字电路设计分为系统级、架构级、寄存器传输级（RTL）、门级和晶体管级等。所谓级别包括两方面含义，一是设计者对电路的认识，二是允许使用的描述方法。系统级设计的任务主要是定义电路的功能和外部特性，设计者只需要将电路为若干个抽象的功能模块，并将各个功能模块的逻辑功能定义清楚即可。架构级设计要具体一些，在这个级别，电路要描述成相互连接的若干个典型逻辑部件和控制其数据传输的**状态机**。典型逻辑部件包括计数器、寄存器、算术运算单元等，又称为**数据通道（Data path）**，状态机则是一个设计中具有特殊性的部分，它控制数据通道的工作。以上两种描述，描述方法没有什么限制，将问题说清楚即可。寄存器传输级的描述是 ASIC 设计中最重要的描述，必须使用硬件描述语言完成。所谓寄存器传输级描述是基于这样一种认识，即任何数字电路，无论功能如何，都是**由寄存器和寄存器之间的组合逻辑电路实现的，**寄存器用来保存数据，组合电路用于传输数据。**RTL 代码必须保证可综合性，只能使用硬件描述语言中的部分描述语句**。从 RTL 代码中**应可隐约看出电路结构，又不要写得过于具体**。细化到逻辑门和触发器的代码并不好，因为从 RTL 描述到 cell 的转换是逻辑综合工具的任务，人为 写到 cell 一级不仅降低了代码的可读性，也不利于优化。在 ASIC 设计中，门级和晶体管级电路是由 EDA 工具生成的，设计者的任务是给出功能正确的 RTL 级代码。验证 RTL 代码的正确性需要编写验证代码，这部分代码称为测试平台（Testbench）。Testbench 用于给 RTL 代码输入信号和判断输出的正确性，也要用硬件描述语言编写，但与 RTL 代码不同，这部分代码不需要生成

电路，对语言的使用没有限制。

1.3 硬件描述语言

硬件描述语言（HDL）是一种用于描述电路功能的语言，具有与计算机编程语言类似的语法，看起来与 C 语言等计算机编程语言很相似，但两者有本质区别。硬件描述语言编写的代码**是对电路的描述，而不是程序**。程序是指令的序列，是有执行次序的，而电路则不存在执行次序的概念。实际上，HDL 语言与 SPICE 语言属于同一类型，但抽象级别较高。在进行仿真分析，SPICE 更精确，但速度太慢，不适合用于大规模电路。HDL 代码的仿真速度要快得多，但只能反映逻辑功能和延迟，无法提供全面信息，也不如 SPICE 精确。

常用的硬件描述语言有 VHDL，Verilog，SystemC 等，这些硬件描述语言各有特点。每一种硬件描述语言都支持多个抽象级别的描述，支持级别有所不同。System C 主要用于系统级建模和功能验证。VHDL 和 Verilog 都支持 RTL 级描述，但 VHDL 侧重更高级别描述，语法复杂。Verilog 主要支持 RTL 和 RTL 以下级别，语法与 C 语言十分相似，应用最为广泛。

1.4 设计与实现流程

在 IC 设计领域，设计流程有两层含义，一是指一般意义的设计流程，强调的是设计步骤和各个步骤的关系，但不具体确定各个步骤的所使用的 EDA 工具。二是具体的设计流程，例如某个公司的设计的特有设计流程，这种流程对 EDA 工具也有明确要求。

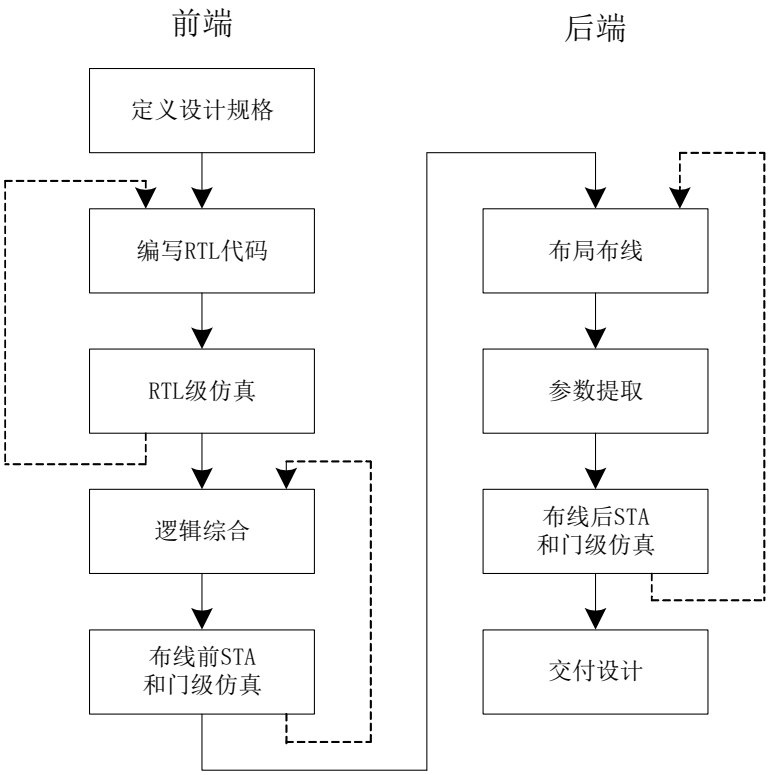


图 1-1 简化的设计流程

图 1-1 是一般意义的简化设计流程。设计流程并不是直线进行的，往往需要多次迭代，主要是解决时序收敛问题。本文将前端的前 3 个环节称为“设计”过程，而将以后的环节称为“实现”过程。

（1） 编写设计规格书

编写设计规格书（SPEC）是最重要设计环节，它实际上包含了系统级和架构级两部分设计内容。SPEC 中包括电路功能定义，芯片制造工艺和标准单元库选择，对最高时钟频率、输入输出时序、功耗、面积等性能方面的要求，以及对工作电压范围、环境温度、静电防护（ESD）、可测性等方面的要求等。此外，电路的基本架构一般也要在这个阶段确定。SPRC 就是技术文档，一般采用文字、框图、状态机、时序图等表示。大型的电路设计需要一个设计队伍来完成，在这个队伍中，每个人只负责一个环节或一个环节的一部分。SPEC 是协调整个设计队伍的依据。

（2）编写 RTL 代码

根据 Specification，用硬件描述语言编写可综合的代码。工具：文本编辑器，如 UltraEdit32，ActiveHDL 等。

（2） RTL 仿真

编写测试平台（TestBench），给出输入信号，验证 RTL 代码与 SPEC 的一致性。常用的工具有：Modelsim（Windows）、VCS（Unix，Linux）等。

（3） 逻辑综合

ASIC：利用综合工具将 RTL 代码转换为门级网表。门级网表中的基本元件包括触发器、逻辑门和缓冲器（Buffer）等称为 cell，由流片厂家（Foundry）以库的方式提供，门级网表就是 cell 和连接关系的集合。常用综合工具是 Synopsys 的 DC（Design Compiler）。

规模较大的设计需要考虑可测性设计（DFT，Design For Test）问题，低功耗设计需要进行功耗优化，如时钟门控、操作数隔离等，可使用 Synopsys 的 Power Compiler 进行功耗优化。

（4）布线前的静态时序分析（STA）和门级仿真。

静态时序分析是根据门延迟和设计约束，通过计算分析设计是否满足时序要求。主要工具是 Synopsys 的 PT（PrimeTime）。

门级仿真是用门级网表代替 RTL 代码，利用与 RTL 仿真相同的 Testbench，重新进行功能验证。主要使用 Synopsys 的 VCS。也可以使用 Modelsim。

布线前的静态时序分析和动态仿真中，cell 的延迟是比较基本准确的，但时钟通常使用理想时钟，连线延迟不考虑。因此，在这个阶段，只能说明设计是有可能实现的，允许有一些时序上的 violation 存在，需要在提取布线信息后解决。

（5） 布局布线

根据门级网表和流片厂家的库文件，利用工具自动完成电路的版图设计并进行设计规则和电气规则校验。可以使用 Synopsys 的 Astro 或 Cadence 的 Encounter 进行版图设计。

（6） 参数提取

参数提取（extract）就是从版图中提取实际的延迟信息和各种寄生参数，可采用 Synopsys 的 Star-rcxt 提取。

（7） 布线后的静态时序分析（STA）和门级仿真

把从版图中提取实际的延迟信息假如到网表中，重新进行 STA 和动态仿真，如果能够通过，则可交付流片（Tapeout）。否则需返回（6）进行修正，如仍无法解决，需返回（4）甚至返回（2）。

第 2 章 基本流程

2.1 概述

本章以一个简单例子对数字电路的 ASIC 设计流程进行展示，目的是使读者迅速了解设计过程的全貌，明确学习目标。

2.2 编写 RTL 代码

2.1.1 设计要求

本例要求设计一个 4 位二进制的计数器，输入输出引脚定义如图 2-1 所示。具体功能要求如下：

（1） 正常工作状态下按二进制方式计数，时钟（clk）上沿计数。

（2） 采用异步复位模式，复位信号 rst 高有效。

（3） 具有同步预置功能，预置 ld 输入为 1 时，时钟上沿后，计数器输出 q 等于预置输入 d。

（4） 计数可由使能端控制，使能 en 信号为 1 时，可计数，使能为 0 时，停止计数。

（5） 计数值达到 1111 时，进位输出信号 co 为 1。

主要性能要求为，在负载电容为 5PF，最高输入延迟为时钟频率的 20%时，最高时钟频率可达到 100MHz。

要求写出 Verilog 代码，验证功能的正确性，并用一个标准单元库进行逻辑综合和物理综合，最终得到该电路模块的版图。

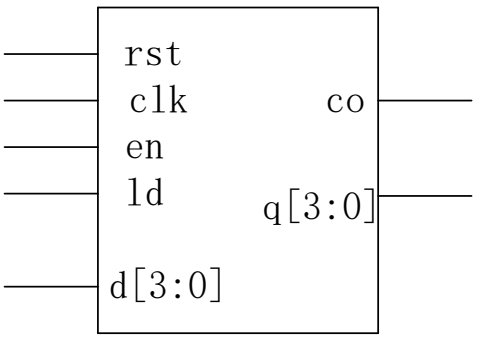


图 2-1 计数器

2.1.2 编写设计代码

根据设计要求，编写设计代码如下：

```
//------------------------------------------------------------------------------
//      Design unit : Example 1 for "Digital IC design and impletation"
//      File name     : counter.v
//      System          : Verilog 1993
//      Author          : XinXiaoNing
//      Revision       : Version 1.0 2/24/2005
//------------------------------------------------------------------------------
`timescale 1ns/1ps
module counter
(rst          ,
 clk          ,
 ld           ,
 en           ,
 d            ,
 q            ,
 co
 );
//------------------------------------------------------------------------
//                  data type definitions.
//------------------------------------------------------------------------
input rst;                          // system reset signal,active high
input clk;                          // system clock signal.
input ld;                           // signal for load "d" to "q".
input en;                           // count enable signal.
input[3:0] d;                       // data input to counter.
output[3:0] q;                      // counter outputs.
output co;                          // carray signal.
//------------------------------------------------------------------------
reg[3:0] q;
reg co;
```

```
//------------------------------------------------------------------------
always @(posedge clk or posedge rst)
begin
    if(rst == 1'b1)
        q <= 4'b0000;
    else if(ld == 1'b1)
        q <= d;
    else if(en == 1'b1)
        q <= q + 1'b1;
end
//------------------------------------------------------------------------
always @(q)
begin
    if(q == 4'b1111)
        co = 1'b1;
    else
        co = 1'b0;
end
//------------------------------ END ---------------------------------------
endmodule
```

这是一个典型的 RTL 代码，从这个例子中可以看出 Verilog 语言的基本语法结构，现对主要内容进行总结。

（1） Verilog 代码的基本单位是模块，以保留字 module 开始，以 endmodule 结束。

（2） 紧跟 module 的是模块名称，这里是 counter。

（3） 模块名称后的括号中的内容是输入、输出端口名称，以逗号分隔；右括号后要有“;”。

（4） 代码的主体分为两部分，即信号声明部分和信号赋值部分。

（5） 信号声明部分包括端口信号声明和内部信号声明。端口信号要声明其方向和位宽。方向分为 input、output 和 inout 等 3 种，位宽以[n：0]的格式声明。

（6） 内部信号分为 reg 和 wire 两种类型；**实际的寄存器信号必须用 reg 声明，但 reg 类型的信号并不一定对应或生成寄存器。**凡以 always 方式赋值的信号都要声明为 reg 类型。

（7） 时序电路部分，即包含寄存器，又时钟控制的电路，描述方式为

```
always @(posedge clk or posedge rst)
begin
    ;
end
```

其特征是使用了 posedge（上沿）或 negedge（下沿）关键字来描述信号变化的条件。其含义为，只有用 edge 声名的输入信号，发生规定方向的变化时，该赋值语句块内部的信号才可能发生变化。

（8） 组合逻辑部分的描述方法为

```
always @(q)
begin
    if(q == 4'b1111)
        co = 1'b1;
    else
        co = 1'b0;
end
```

特征是不使用 posedge 或 negedge，即任何时刻，当输入信号变化时，输出都有可能发生变化。

（9） 两个 always 语句之间没有执行次序问题，它们都是对电路的描述。

（10）// 后写是注释。

（11）`timescale 是定义时间单位和精度，这里定义基本单位为 1ns，分辨到 1ps，即在仿真时，时间可有 3 位小数。

此外，还需注意 Verilog 语言是区分大写和小写的，所有保留字都需要小写。以上是设计代码部分，要验证设计部分正确性，需要另外编写一段 testbench 代码，因为设计代码的输入信号必须从外部提供。学习 HDL 的关键在于**要始终考虑所描述的电路，“写的是代码，想的是电路”。**

2.1.3 编写验证代码

与设计代码不同，验证代码不需要综合出电路，对语言的使用没有限制。为了更方便地生成各种激励信号，验证代码可以写得抽象一些。在验证代码中，可以使用一些类似程序的编写方法，如使用任务、函数、循环和数据文件等。

在 Verilog 中，使用#表示时间的变化，例如#1 表示时间走过一个基本单位。仿真工具默认的时间单位是 1ns，可以使用`timescale 语句来修改。Testbench 也是一个模块，但没有输入输出端口。

```
//------------------------------------------------------------------------
//      Design unit : testbench of counter
//      File name     : counter_tb.v
//      System          : Verilog 1993
//      Author          : XinXiaoNing
//      Revision       : Version 1.0 2/24/2005
//------------------------------------------------------------------------
`timescale 1ns/1ps
module testbench();
//------------------------------------------------------------------------
//      Signals for mapping Testting module
//------------------------------------------------------------------------
reg rst,clk,ld,en;
reg [3:0] di;
wire co;
wire [3:0] qo;
//-------------------- Component under test --------------------------------
counter UUT
(.rst(rst)            ,
 .clk(clk)            ,
 .ld(ld)              ,
 .en(en)              ,
 .d(di)               ,
 .q(qo)               ,
 .co(co));
//------------------------------------------------------------------------
task half_pulse;
inout clock;
      #10 clock = ~ clock;
endtask
//------------------------------------------------------------------------
task pulse;
input[7:0] num;
integer i;
      for(i=0;i<num;i=i+1)
      begin
            half_pulse(clk);
            half_pulse(clk);
      end
endtask
//------------------------------------------------------------------------
```

```
//------------------------------------------------------------------------
initial
begin
    rst = 0;
    clk = 0;
    en = 0;
    ld = 0;
    di = 4'b0000;
end
//------------------------------------------------------------------------
always
begin
    rst = 1;
    pulse(2);
    rst = 0;
    en = 1;
    pulse(10);
    ld = 1;
    pulse(1);
    ld = 0;
    pulse(2);
    en =0;
    pulse(2);
    rst = 1;
    en = 1;
    pulse(10);
    $stop();
end
//------------------------------------------------------------------------
endmodule
```

在这段代码中，以下几个问题需要注意：

（1） 在 testbench 中例化设计代码时要求明确写出 testbench 中的信号与设计代码端口信号的连接关系。语法格式为

设计模块名 例化名 （.设计端口信号名（testbench 中定义的信号名），…）;

（2） 设计代码的输入信号，在 testbench 中要声明为 reg 类型，可用 initial 语句块或 always 语句块中赋值。设计代码的输出信号，在 testbench 中要声明为 wire 类型，不能赋值。

（3） 只有使用了#x，信号的变化才有时间间隔，否则都是同时发生的。

（4） $stop()是仿真工具提供的系统函数，不属于 verilog 语言。

以上仅是一个基本的验证代码，需要观察波形才能确定设计的正确性。完善的验证代码应加入判断语句，自动判断结果，并指出问题出现的位置。testbench 的编写方法属于验证技术，将在后续章节讨论。

2.1.4 基于 Modelsim 的 RTL 级仿真

Modelsim 的基本使用方法很简单，通过以下几个步骤就可看到输出波形。

（1） 新建一个 project。

（2） 将设计代码和 testbench 代码添加到 project；

（3） 编译所有文件。

（4） 执行仿真 simulate。

（5） 选择 testbench 模块。

（6） 添加信号波形。

（7） 运行仿真。

功能验证可能是设计流程中耗时最多的环节，对于复杂电路来说，全面验证其逻辑功能不仅需要严谨的工作态度，还应学习有关理论。

2.2 基本的逻辑综合方法

逻辑综合是将硬件描述语言编写的设计代码转换为标准单元库中的 cell 构成的网表的过程。目前，应用最广泛的逻辑综合工具是 Synopsys 的 Design Compiler，简称 DC。DC 需要在 Linux 环境下使用，应首先学习 Linux 操作系统的基本使用方法。

深入学习 DC 是后续章节的任务，这里先介绍一下基本使用方法，目的是验证设计代码的可综合性。RTL 代码只能使用 Verilog 语言的部分描述方法，仿真通过并不能证明可综合性，只有能 DC 的检查，该代码才是可用的。

使用 DC 之前，必须先建立一个工作目录，并写好一个环境设置文件，该文件名为.synopsys_dc.setup。在该文件中，要定义综合使用的标准单元库的名字和搜索路径。DC 在启动时，首先会读安装目录下的.synopsys_dc.setup，然后再读用户目录下的同名文件。如果用户设置与默认设置不同，则以用户设置为准。**因此，一定要先进入工作目录，再启动 DC。**这是 Linux 操作系统下的软件的共同

特点。以下是一个简单的环境设置文件。

```
set target_library {fsa0a_c_sc_tc.db}
set link_library {* fsa0a_c_sc_tc.db}
set symbol_library {fsa0a_c_sc.sdb}
set search_path " /usr/iclib/fsa0a_c/2004Q4v1.1/SC/FrontEnd/synopsys "
set WORK_DIR "/usr/dc09"
```

这个文件指定了库的名字和路径（要根据实际安装来写），并定义了一个环境变量 WORK_DIR，没有修改原环境文件的其它设置。使用 DC 需要编写脚本文件，其中内容主要是输入、输出文件的路径、电路的外部工作条件和时序约束等，详细内容将在后续章节介绍，以下仅介绍几个最基本的命令，使用这几个命令就可实现对设计代码的可综合性的检查。

（1） read_verilog

该命令用于读入 verilog 代码，基本格式为

read_verilog 路径和文件名

（2） elaborate

该命令用于将设计代码转换为统一的与工艺无关的内部电路，基本格式为

elaborate 模块名

注意是模块名，不是文件名。实际上，这个命令可以省略。

（3） current_design

指定当前需要综合的模块，格式为

current_design 模块名

注意是模块名，不是文件名。

（4） create_clock

用于确定时钟信号的名称、周期和波形等信息。例如，

```
create_clock -name "clk" -period 10 -waveform { 0 5 } [get_ports clk]
```

这个命令定义了一个名为 clk 的时钟信号，时钟周期为 10，时间单位在标准单元库中定义，一般为 ns。该时钟的占空比为 50%，信号来自 clk 端口。该命令用法比较复杂，将在后续章节继续介绍。

（5） set_input_delay

这个命令用来定义输入信号的延迟，没有该定义无法综合，详细用法以后

介绍。

（6） compile

执行优化和映射过程的命令，执行后可得到 cell 网表。

以下是一个最简单的脚本，**这个脚本主要用来检查代码是否存在不可综合的问题，**正式综合时还要加入更多约束。脚本中的其它命令将在后续章节介绍。

```
################################################################################
# 以下是设计代码的路径
read_verilog $DC_WORK_DIR/code/counter.v
# 指定需要综合的模块
current_design counter
link
# 每个输出需要驱动 4 个基本门
set_fanout_load 4 [all_outputs]
# 时序约束部分
# 时钟频率 100MHz。
create_clock -name "clk" -period 10 -waveform { 0 5 } [get_ports clk]
# 模块内部各寄存器时钟引脚处时钟沿的时间差，估计值。
set_clock_uncertainty -setup 0.1 clk
set_clock_uncertainty -hold 0.1 clk
#输入信号相对时钟的延迟
set_input_delay -max 2 -clock clk [all_inputs]
set_input_delay -min 0.1 -clock clk [all_inputs]
#暂不考虑时钟网络驱动能力问题
set_dont_touch_network [get_clocks clk]
# 实现综合，映射到 cell 库
compile
#生成时序报告
report_timing > $DC_WORK_DIR/rpt/setup_time.rpt
report_timing -delay min > $DC_WORK_DIR/rpt/hold_time.rpt
# 生成 Verilog 格式的 cell 网表文件
write -format verilog -hier -o $DC_WORK_DIR/nt/counter_gate.v
```

DC 有多种启动方式，以下只介绍一种。首先，使用 cd 命令进入工作目录，然后输入 design_vision，回车后将出现图 2-2 所示图形界面。

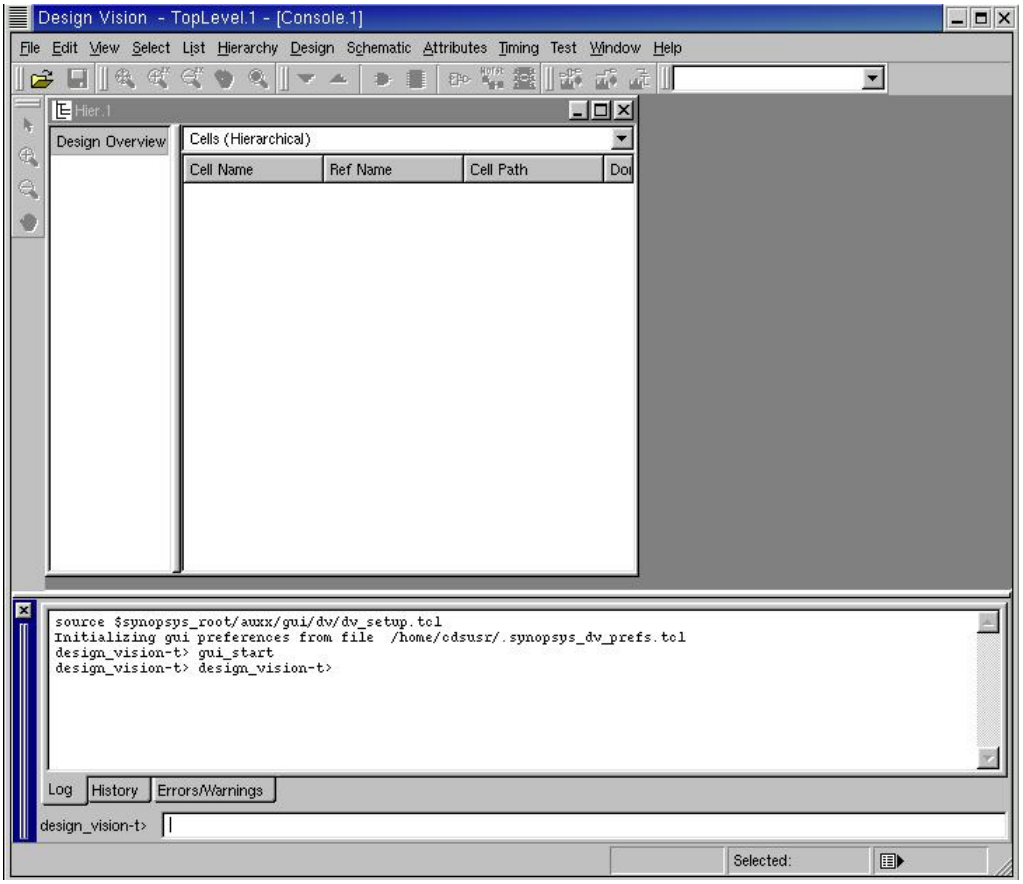


图 2-2 design_vision 的操作界面

利用菜单操作 File→Excute Script 后弹出以下窗口，找到脚本文件后，点“Open”开始执行逻辑综合。

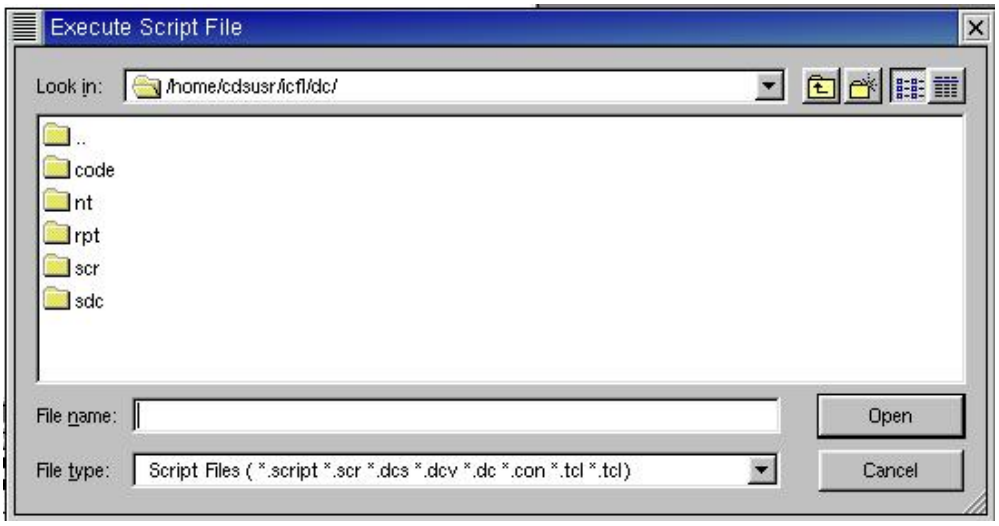


图 2-3 弹出窗口

如果脚本和设计代码无误，即可得到综合后的电路图、网表和时序分析报告等文件。

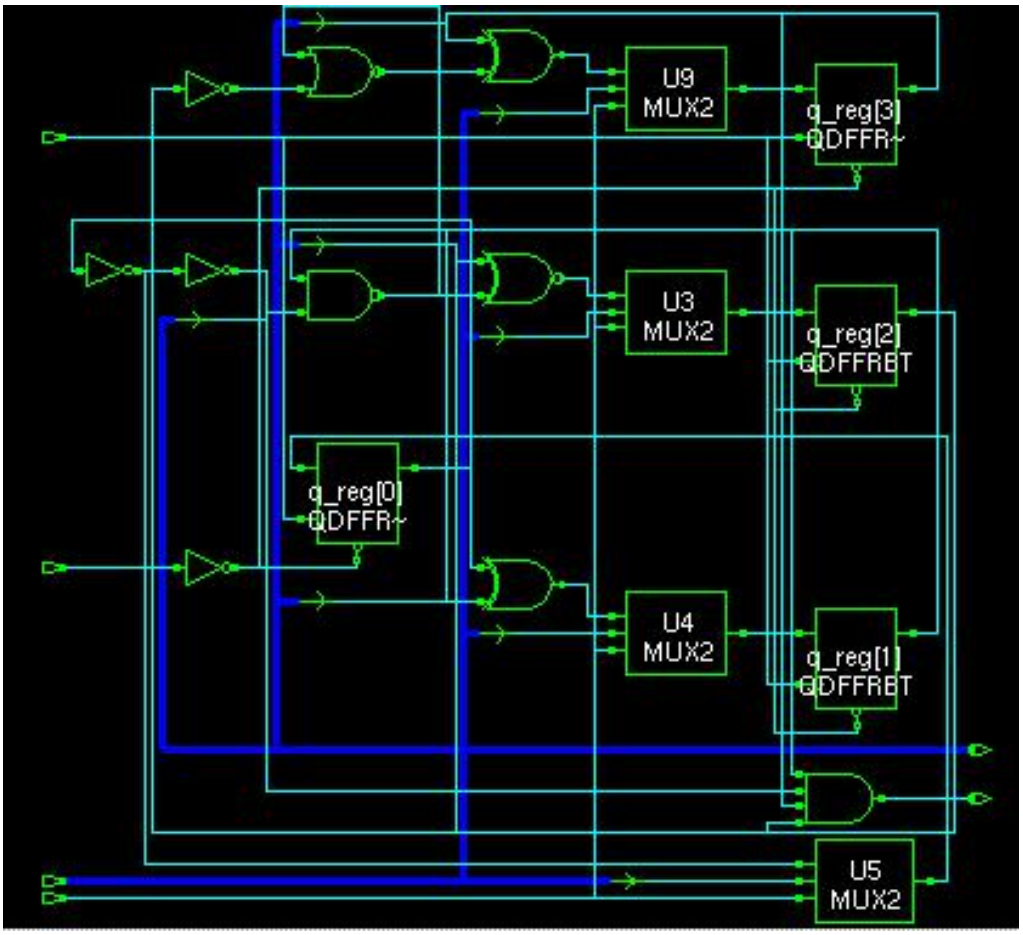


图 2-4 综合后的电路图

电路中的各个 cell，如逻辑门、多路选择器和触发器等都来自标准单元库。MUX 如果设计代码中有错误，将在窗口中出现错误或警告信息。电路图是由 cell 网表得到的，以下是 verilog 格式的网表，与标准单元库提供的各个 cell 的 verilog 文件放在一个 project 中，就进行“门级”仿真。DC 还可生成各种报告文件，其中最重要的是时序报告。

以下是网表文件。

```
module counter ( rst, clk, ld, en, d, q, co );
input  [3:0] d;
output [3:0] q;
input  rst, clk, ld, en;
output co;
    wire n2, n3, n4, n5, n6, n7, n8, n9, n10, n11, n12, n13, n14, n15, n16;
    MUX2 U3 ( .O(n15), .S(ld), .A(n7), .B(d[2]) );
    INV2 U4 ( .O(n8), .I(n4) );
    MUX2 U5 ( .O(n14), .S(ld), .A(n9), .B(d[1]) );
    MUX2 U6 ( .O(n16), .S(ld), .A(n6), .B(d[3]) );
    AN2B1P U7 ( .O(n3), .I1(q[2]), .B1(n4) );
    MUX2 U8 ( .O(n13), .S(ld), .A(n11), .B(d[0]) );
    DFFRBP \q_reg[1]  ( .Q(q[1]), .D(n14), .CK(clk), .RB(n12) );
    DFFRBP \q_reg[0]  ( .Q(q[0]), .QB(n2), .D(n13), .CK(clk), .RB(n12) );
    INV1 U9 ( .O(n10), .I(n5) );
    XOR2 U10 ( .O(n6), .I1(q[3]), .I2(n3) );
    DFFRBP \q_reg[3]  ( .Q(q[3]), .D(n16), .CK(clk), .RB(n12) );
    XOR2 U11 ( .O(n9), .I1(q[1]), .I2(n10) );
    XOR2 U12 ( .O(n7), .I1(q[2]), .I2(n8) );
    DFFRBT \q_reg[2]  ( .Q(q[2]), .D(n15), .CK(clk), .RB(n12) );
    XOR2 U13 ( .O(n11), .I1(en), .I2(q[0]) );
    AN4 U14 ( .O(co), .I1(q[1]), .I2(q[0]), .I3(q[3]), .I4(q[2]) );
    INV4 U15 ( .O(n12), .I(rst) );
    OR2B1P U16 ( .O(n5), .I1(n2), .B1(en) );
    OR2B1P U17 ( .O(n4), .I1(n5), .B1(q[1]) );
endmodule
```

以下是建立时间的报告。在建立时间报告中给出了一个“最坏”路径的延迟信息。最终结论是最后的一句，满足时序要求是“slack（MET）”，不满足则为“violated”。

Information: Updating design information... (UID-85)

Operating Conditions:
Wire Load Model Mode: enclosed

Startpoint: en (input port clocked by clk)
Endpoint: q_reg[3] (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max

Des/Clust/Port	Wire Load Model	Library
counter	enG5K	fsa0a_c_sc_tc

Point	Incr	Path
clock clk (rise edge)	0.00	0.00
clock network delay (ideal)	0.00	0.00
input external delay	2.00	2.00 f
en (in)	0.03	2.03 f
U16/O (OR2B1P)	0.19	2.21 r
U17/O (OR2B1P)	0.12	2.33 r
U7/O (AN2B1P)	0.15	2.49 f
U10/O (XOR2)	0.14	2.63 r
U6/O (MUX2)	0.13	2.76 r
q_reg[3]/D (DFFRBP)	0.00	2.76 r
data arrival time		2.76
clock clk (rise edge)	10.00	10.00
clock network delay (ideal)	0.00	10.00
clock uncertainty	-0.10	9.90
q_reg[3]/CK (DFFRBP)	0.00	9.90 r
library setup time	-0.09	9.81
data required time		9.81
data required time		9.81
data arrival time		-2.76
slack (MET)		7.05

以下是保持时间的报告，结论也是“MET”。

```
Operating Conditions:
Wire Load Model Mode: enclosed

  Startpoint: d[0] (input port clocked by clk)
  Endpoint: q_reg[0] (rising edge-triggered flip-flop clocked by clk)
  Path Group: clk
  Path Type: min

  Des/Clust/Port          Wire Load Model           Library
  ------------------------------------------------
  counter                 enG5K                     fsa0a_c_sc_tc

  Point                                             Incr          Path
  -----------------------------------------------------------
  clock clk (rise edge)                           0.00          0.00
  clock network delay (ideal)                     0.00          0.00
  input external delay                            0.10          0.10 f
  d[0] (in)                                       0.01          0.11 f
  U8/O (MUX2)                                     0.13          0.25 f
  q_reg[0]/D (DFFRBP)                             0.00          0.25 f
  data arrival time                                             0.25

  clock clk (rise edge)                           0.00          0.00
  clock network delay (ideal)                     0.00          0.00
  clock uncertainty                               0.10          0.10
  q_reg[0]/CK (DFFRBP)                            0.00          0.10 r
  library hold time                              -0.01          0.09
  data required time                                            0.09
  -----------------------------------------------------------
  data required time                                            0.09
  data arrival time                                            -0.25
  -----------------------------------------------------------
  slack (MET)                                                   0.16
```

从这个例子可以初步了解从代码编写到功能验证，再到逻辑综合的过程。在学习 RTL 级代码设计时，考虑代码的可综合性是必要的。逻辑综合工具比一般的仿真分析工具要严格得多。代码的可读性和综合结果是判断代码质量的依据。设计流程的其它环节涉及较多基础知识，将在后续章节介绍。

第 3 章 Verilog HDL 的 RTL 子集

Verilog 语言能够用于各种抽象级别的描述，但其中只有部分表示法可用于逻辑综合，有些表示法只能用于在仿真时编写测试平台，某些低级别的表示法，如门原语等，虽然能够被综合，但是在编写 RTL 代码时，不推荐使用。这里将能够被大多数逻辑综合工具接收的数据类型、信号类型和语句称为 Verilog 语言的 RTL 子集。

3.1 数据类型及常数、信号

3.1.1 常数

（1）数字

完整的数字表示法：

位宽’进制 数字。

例如： 2’b01，8’h1f，4’d12 等。注意，不论进制如何，这里的**“位宽”都是指转换为二进制后的位数**，例如 8’h1f = 8’b00011111。可以在数字中间加下划线以增加可读性。例如：16’b1101_1100_0010_1010;

（2） 信号的值

在 Verilog 语言中，一个数字信号可有 4 种取值，即 0，1，z， x。其中 z 表示高阻态，x 表示不定值。

（3）参数 parameter

parameter 的作用是用符号代替常量，类似 C 语言中的宏定义。例如：

parameter[2:0] DATA_WIDTH = 8；

parameter[3:0] IDLE_STATE = 4’b0000。

在编写代码时，使用参数代替常数是一种好习惯，可以提高代码的可读性。当同一常数多处使用时，修改也比较方便。

3.1.2 信号和变量

在 Verilog 语言本身并没有对信号和变量加以区分，但两者在概念上有本质的不同。信号对应于电路的输入、输出或连线。而变量则与电路没有对应关系（例如上一章中，testbench 中任务 pulse 中的 i 是变量，而不是信号）。通常在 RTL 代码中只有信号，而变量只出现在测试平台中。

信号和变量的命名规则相同，与 C 语言类似，**第一个字符只能是字母，**其

后可跟字母、数字或下划线。**Verilog 语言是区分大小写的，**在 RTL 代码中，推荐的写法是所有的信号都用小写字母加下划线表示。

RTL 代码中，信号只推荐使用两种类型，即 reg 型和 wire 型。**电路中的寄存器（触发器）的输出只能声明为 reg 型，但并非所有的 reg 型信号都是寄存器器的输出。**Verilog 语言中规定**所有用 always 语句赋值的信号都要声明为 reg 型。**wire 型信号主要用于描述模块之间的连线，也可用于描述组合逻辑关系。例如，在上一章的例子中，reg co，表示进位信号，它实际上是一个组合逻辑电路的输出。而 reg[3:0] q; 表示 4 位的逻辑信号，实际上是触发器的输出。也就是说，reg 型信号究竟是组合电路还是寄存器（触发器）的输出并不取决于声明，而要看以后的使用方法。

3.2 运算符

3.2.1 算术运算符

Verilog 的算术运算符与 C 语言相同，有以下几种：

＋，－，＊，/，%

其中只有 ＋，－ 是完全可综合的，＊ 虽然也能综合但需要慎重使用，后两种不要在 RTL 代码中使用。

3.2.2 逻辑运算符

&& || ！ 使用方法与 C 语言基本相同。

3.2.3 位运算符

~ , & , | , ^ 的定义与 C 语言相同。

3.2.4 关系运算符

< , <= , > , >=, == ,!=, 的定义与 C 语言相同。

在 Verilog 中还定义了全等运算符 === 和 !==，这里== 与 === 的区别是前者在操作数中**出现不定态 x 和高阻态 z 时，结果为不定值 x**，后者对 x 和 z 也进行比较，完全一致时为 1，否则为 0。例如：

a = 5’b11x01, b = 5’b11x01

使用 == 比较，结果为 x，使用 === 比较结果为 1。在 RTL 代码中，“===”和“！===”等较难理解的表示法，最好也不要使用。

3.2.5 移位运算

>> 表示右移，<< 表示左移，可综合。

3.2.6 位拼接

运算符 {}，用于将两个或多个“位宽”小的信号拼接为一个“位宽”较大的信号，例如，如果存在一个 8 位的信号“reg[7:0] a;”和两个 4 位信号“reg[3:0] b,c;”，且 b=4’b1101，c= 4’b1010，使用以下语句赋值

```
always @(b or c)
    a = {b, c};
```

结果为：

```
a = 8’b1101_1010;
```

位拼接实际上描述的是一种连接关系，这里 b 连接到 a 的高 4 位，c 连接到 a 的低 4 位。

3.2 基本语句

3.2.1 赋值语句

在 RTL 代码中，信号主要有以下两种赋值方式，即连续赋值和利用过程语句赋值。

（1） 连续赋值语句（Continous Assignments）。

连续赋值只能用于 wire 型信号，这里“连续”一词是从仿真意义上讲的，意思是无论输入信号是否变化，每个时间步长都对输出信号辅值。以下是使用方法。

```
wire a ,b, c;
assign a = b & c ;
```

连续赋值语句的优点是可以在一行代码中写出复杂的逻辑关系，但可读性不如使用过程语句，不推荐使用。

（2） 过程赋值语句（Procedural Assignments）

过程赋值语句就是在例子中出现的使用 always 语句块的赋值方法。reg 信号必须在 always 语句块中赋值。过程赋值又有两种具体写法，一种是非阻塞 non-blocking）式赋值，其特征是在 always 语句块中使用“<=”运算符，例如：

```
always @(posedge clk) begin
    if( rst == 1’b1)
```

```
q <= 4'b0000;
else if( en == 1'b1)
q <= 1 + 1'b1;
end
```

这种赋值方法主要用于时序电路的描述。另一种称为阻塞式（blocking）赋值，特征是在 always 语句块中“=”运算符。例如：

```
always @ ( a or b or c)
begin
    if(a == 1'b1)
        y = b + c;
    else
        y = 4'bz;
end
```

以上三种赋值语句有较大的区别。仿真时，连续赋值的表达式在每个时间单位都要计算。而过程赋值语句，只有敏感变量表中的信号发生变化时，才对 always 块中的表达式赋值。阻塞式赋值，仿真软件执行到该语句时，被赋值的信号值“立即”发生变化。“阻塞”了仿真程序对其它语句的处理。使用非阻塞式赋值时，仿真软件执行到该语句时，先将所有等号右边的值计算出来，但并不立即改变等号左边的信号的值，而是去考虑该时刻还有哪些信号需要赋值，在结束对该时刻所有信号的分析后，退出 always 块时才进行实际的赋值。因此，所谓“阻塞”指的是阻塞仿真软件对其它模块的计算。逻辑综合时，大多数情况下，阻塞式赋值和非阻塞式赋值在逻辑综合时没有什么区别，但推荐的用法是：**组合电路使用“=”赋值，时序电路使用“<=”赋值，**因为这种使用方法符合 EDA 软件设计者的认识，是综合效果最好的。此外，在一个 always 块中，**不允许混合使用两种赋值方式。**

3.2.2 条件语句

条件语句有 if-else 语句和 case 两种，都只能在 always 块中使用。if-else 用于描述具有优先级的电路。case 语句则描述无优先级关系的电路，一般说，在两种语句都可使用时，使用 case 的描述，生成的电路速度好快些。Case 语句的格

式如下：

```
module selector(sel,d,y);
input[1:0] sel;
input[3:0] d;
output y;
reg y;
always @(sel or d)
begin
        case (sel)
              2’b00 :    y = d[0];
              2’b01 :    y = d[1];
              2’b10 :    y = d[2];
              2’b11 :    y = d[3];
              default:    y= 0;
        endcase
end
endmodule
```

除了基本的 case 语句外，verilog 中还有 casex 和 casez 等描述方法，这些描述语句都比较容易出现问题，不推荐使用，就不再介绍了。

3.3 组合电路的描述方法

组合电路有以下几种描述方法：

（1） 使用 always 语句块

描述组合电路的 always 块的形式如下：

```
        always @(a or b or c or d)
        begin
              if(条件 1)
                    out =  表达式 1；
              else if (条件 2 )
                    out =表达式 2；
              …
```

```
        else
            out =表达式 n;
    end
```

需要注意的问题如下：

A．**所有表达式中用到的信号都要写在敏感变量表中**；

B．必须考虑所有条件下的输出，使用 if-else 法，**一定不能省略 else 后的表达式。**用 case 表示法时，如果没有用完全部输入组合，最后要有 default 条件下的输出，否则都将生成不希望的锁存器 latch。

（1） 使用连续赋值语句。

对于简单的组合逻辑关系可以使用 assign 赋值语句描述，以减少代码的长度。具体有以下几种：

A． 无条件的赋值语句

```
assign a = b || c || d;
```

B． 三目运算表示法

```
assign out = sel? in1:in2;
```

这种方法与以下表示法等价

```
always @(sel or in1 or in2)
begin
    if(sel == 1'b1)
        out = in1;
    else
        out = in2;
end
```

C． 隐含译码关系

```
assign inst_jp = {prg_reg[15:14] = = 2'b01};
```

这种逻辑关系等价与下面的描述相同（但 inst_jp 的声明不同），

```
always @(pro_reg)
begin
  if(pro_reg[15:14] == 2'b01)
```

```
        inst_jp = 1'b1;
    else
        inst_jp = 1'b0;
end
```

3.4 时序电路的描述方法

以触发器作为存储元件的时序电路只能使用 always 块描述，且必须使用 posedge 或 negedge 关键字描述时钟和异步复位信号。格式如下：

```
always @(posedge clk or posedge rst)
begin
    if(rst == 1'b1)
        out <=  常数;
    else begin
        其它情况下的输出;
        end
end
```

需要注意的问题：

（1） 在敏感变量表中，最多只能有两个使用 posedge 或 negedge 描述的信号，而且一个是时钟信号，一个是异步复位信号，如果采用同步复位，则只有一个时钟信号，否则不可综合；

（2） 表达式中出现的其它输入信号不必写在敏感变量表中。

（3） 可以只考虑所关心的变化（允许在 if 后不写无条件的 else）。

例如：

```
always @(posedge clk)
begin
    if(rst == 1'b1)
        q   <=   4'b0000;
    else if(en == 1'b1)
        q   <=   q + 1'b1;
end
```

没有必要写：else q <= q;

3.5 锁存器（latch）的描述方法

除非确实需要，一般不要使用锁存器。如果确实需要，可以使用如下方式描述：

```
always @(ale)
begin
    if(ale == 1'b1)
        addr = ad;
end
```

3.6 编写 RTL 代码必须遵守的规则

以下规则是在写 RTL 代码时必须遵守的，否则无法综合或综合的结果与预期的不一致。

规则 1：**不能在两个或多个 always 块中对同一信号赋植。**

在写 RTL 代码时，应将一个 always 块想象成一个电路单元，在 always 块中赋值的信号是这个电路单元的输出，它不能与其它电路的输出直接连接。这种错误在仿真工具中，并不出现错误提示，但综合和仿真结果很容易出错。

规则 2：**不能在一个 always 块中混合使用阻塞和非阻塞式赋值。**

这种描述方法，综合工具会报错。

规则 3：**在一个 always 块中最多只能使用两个 posedge（negedge）型的敏感信号。**

一个使用一个时钟信号和最多一个异步复位信号，否则不可综

合。

规则 4：**在 case 语句中，要包括所有输入组合或使用 default 指定未明确写出的其它组合的输出。**

否则会生成 latch。

规则 5：**对于组合电路，使用 if-else 语句时，应考虑所有可能出现的状态，避免生成 latch。**

3.7 风格及行规

- 信号名、端口名都用小写加下划线，常量名和用户定义类型用大写；

- 使用有意义的信号名，但不要长于 28 字符。
- 低电平有效信号，以下划线跟小写字母 n 表示，例如 rst_n,wr_n 等。
- 时钟信号以 clk 命名，多时钟用 clk1、clk2 等命名。高电平有效的复位信号以 rst 命名，低电平有效的复位信号以 rst_n 命名。
- 采用缩进提高可读性，使用四个空格，不要使用 TAB 键。
- 不要使用 Verilog 或 VHDL 保留字为端口、信号、函数、任务等命名。
- 组合电路使用阻塞式赋值，时序电路使用非阻塞式赋值。

3.8 其它设计原则

- 不要使用异步逻辑

  异步逻辑电路很难综合和验证，要尽可能避免。
- 尽量避免使用多时钟

  在大的设计中，使用分频时钟是难以避免的，但需要特殊处理。
- 内部不要使用三态电路（用 MUX 代替）；
- 顶层不要使用胶合逻辑

  设计代码最上层的模块中应只包含各个子模块的例化，相互之间以 wire 型信号相连接，不要再写逻辑关系了。
- 避免使用门控时钟；

  在时钟信号上加门电路控制是低功耗设计手段，但不要在 RTL 代码中，手工写“门控时钟”。需要时，用使能信号代替门控时钟，在综合时，可用设计工具自动转换。
- 避免过长的 if-else-if 链。
- 避免使用“门级”描述

  Verilog 中支持使用“门原语”描述逻辑关系，但在 RTL 代码中不要使用。
- 每个 always 语句块最好只处理一个信号

  一个 always 语句块中对多个信号赋值是可以综合的，但这种描述方法容易出出现问题。
- 尽量避免同时使用上升沿动作和下降沿动作的寄存器。

第 4 章 基本设计方法

复杂数字电路设计的难点在于对设计对象的理解。如果能够对设计对象的逻辑功能和输入输出时序有非常清晰的认识，用 HDL 语言描述其功能就是一个比较简单的任务。本章讨论复杂电路模块的分析代码编写思路，重点介绍代码编写时的思考方法。数字电路种类虽多，但大多由两部分构成，即数据通道部分和状态机部分。数据通道部分指有通用性的部分，状态机部分是整个电路中的控制部分，具有特殊性。

4.1 数据通道设计方法

数据通道包括简单的计数器、寄存器、算术运算器等，也包括一些基于特定算法的电路，如 FIFO、栈、以及数字滤波器和 FFT 运算等。这类电路的时序关系一般比较简单，关键是理解算法。以下以一个同步先进先出寄存器（FIFO）为例，介绍一下代码编写思路。

例 1：设计一个深度为 16，数据宽度为 8 的同步先进先出寄存器（FIFO）。

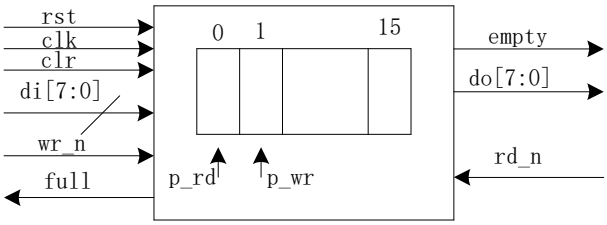


图 4-1 FIFO 的示意图

该问题的 RTL 代码的设计过程可以按以下步骤进行：

（1） 分析设计对象

FIFO 是用于 CPU 与其它数字部件或 CPU 与 CPU 之间缓冲处理速度的常用部件。在 FIFO 内部有一定数量的存储单元，可以用寄存器或 RAM 实现。简单地说，FIFO 的功能类似于软件技术中的队列，最先写入的数据最先可以读出，次写入的数据可以在第二次读操作时被读出，依次类推。如果数据写入与读出的操作是同一个时钟信号控制的，就称这种 FIFO 是同步的，否则为异步 FIFO。使用 FIFO 可以避免当 CPU 或其它数字处于“忙”状态时造成数据丢失。FIFO 的缓冲能力取决于存储单元的个数，称为 FIFO 的深度。一次可读写的数据的位数

称为 FIFO 的宽度。图 4-1 中的 fifo_full 和 fifo_empty 用来指示 FIFO 的状态。本例中考虑使用寄存器来实现存储单元，为实现 FIFO 的算法，除端口信号外，还需要两个内部信号来记忆应写入的存储单元的地址和应读出的存储单元的地址。

一些教课书中建议先画出电路结构，再编写代码。如果能够迅速画出电路结构当然好，但实际上，在对电路功能理解还不是很深时，画出好的电路结构图可能比编写代码更加困难。这里推荐的方法是**先编写一个行为级代码**，然后再逐渐修改为 RTL 级。

（2） 编写行为级代码

因此，在写 FIFO 的行为级代码时，第一步就是写出模块的框架和信号的声明，暂时不考虑其它细节问题。

```
module fifo16x8
( rst          ,
  clk          ,
  clr          ,
  rd_n         ,
  wr_n         ,
  di           ,
  do           ,
  full         ,
  empty
  );
input rst;                          // system reset signal,active high
input clk;                          // system clock signal.
input clr;                          // clear signal.
input rd_n;                         // FIFO read signal,active low.
input wr_n;                         // FIFO write signal,active low.
input[7:0] di;
output[7:0] do;
output full;
output empty;
reg[7:0] elem[15:0];
reg[3:0] p_wr;
reg[3:0] p_rd;
reg full;
reg empty;
reg[7:0] do;
//-------------------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------------------
endmodule
```

写完基本框架和信号定义后，就可以每个输出信号和内部信号的变化条件了。**可以从逻辑关系简单的信号开始，逐个进行描述。**

这里，我们首先考虑“p_wr”这个信号，根据 FIFO 的功能可以看出，该信号需要由时钟控制，因此需要用描述时序电路的方法进行描述，可以先写出如下框架：

```
always @(posedge clk or posedge rst)
begin
    if(rst == 1’b1)
        p_wr <= ?
    else if()
        p_wr <= ?
end
```

然后可以仔细分析在复位时，写指针 p_wr 需要取什么样的值（复位值其实可以任意选取，通常根据习惯取零），再考虑 p_wr 在其它情况下的值。这样，可以逐步完善该 always 块的内容。

```
always @(posedge clk or posedge rst)
begin
    if(rst == 1'b1)
        p_wr <= 4'b0000;
    else if(clr == 1'b1)
        p_wr <= 4'b0000;
    else if((wr_n == 1'b0) && (full == 1'b0))
        p_wr <= p_wr + 1'b1;
end
```

在这里，设计者需要做一个决定，当已经出现“满”标志时，如果有写操作，“写指针”是否要加 1，现有代码是按不再继续加 1 考虑的，事实上，继续加 1 也不算错，**但必须将处理办法写在说明书中，补充到 SPEC 中**。在本设计中，FIFO 深度为 16，p_wr 声明为 4 位，在 p_wr 等于 15 后再加一将自然回 0。

类似地，可以写出 p_rd 描述。实际上，在这个阶段只要仅可能将各种变化

关系写全就可以了，许多细节的问题可以在验证阶段修改。

```
always @(posedge clk or posedge rst)
begin
    if(rst == 1'b1)
        p_rd <= 4'b0000;
    else if(clr == 1'b1)
        p_rd <= 4'b0000;
    else if((rd_n == 1'b0) && (empty == 1'b0))
        p_rd <= p_rd + 1'b1;
end
```

这里设计者决定在已经出现“空”标志时，读操作不改变“读指针”。同样，这种决策也需要补充到 SPEC 中。

以下考虑存储单元 elem[15:0]的变化。存储单元实际上由 16 个寄存器组成，在本例中，可以不考虑复位后的值，因为 FIFO 总是在非空的状态下读出的值才有意义，如果由复位信号控制，会增加复位信号的负载。故可以描述如下：

```
always @(posedge clk)
begin
    if((wr_n == 1’b0) && (full == 1’b0))
        elem[p_wr] <= di;
end
```

对于“满”和“空”标志的处理有两种方法，也是需要设计者决策的。一种方法是将其理解为组合电路的输出，电路结构如图 4-2。

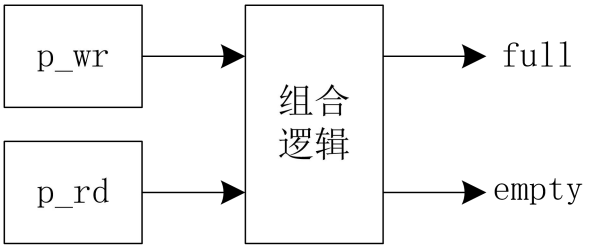


图 4-2 标志采用组合逻辑输出

另一种处理方法是采用寄存器输出，即“满”和“空”标志本身使用寄存器

记忆，从寄存器直接输出。电路结构如图 4-3 所示。

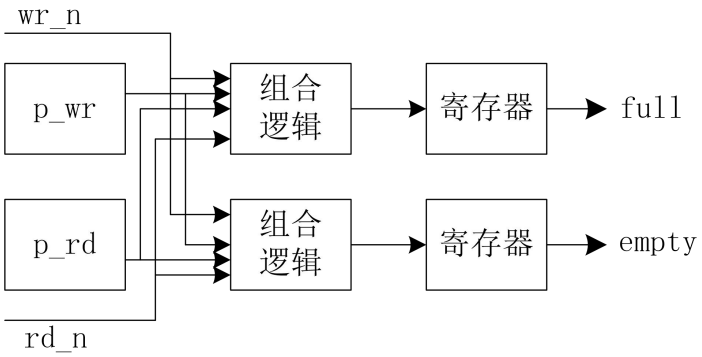


图 4-3 用寄存器直接输出

对第一种处理方法，设计者要描述的是“满”或“空”产生的条件，只与两个“指针”有关。在第 2 种处理方法中，设计者需要描述的是“即将”产生“满”或“空”状态的条件，不仅与两个“指针”有关，还与“读”、“写”输入有关。尽管第一种处理方法要简单得多，出于性能方面的考虑，仍推荐采用第 2 种方法。要想到当前电路模块的输出将是其它电路模块的输入，第一种方法的输出信号是有组合电路延迟的，而第 2 种方法则只有寄存器延迟。编写局部电路模块时，使所有输出信号都直接由寄存器输出是推荐的设计方法，对提高整体电路性能有利。

以下按寄存器输出方式考虑“满”、“空”信号的处理。empty 信号的逻辑关系比较复杂，可以分为两个 always 块来写。

定义一个中间信号 reg empty_m; 首先描述时序部分：

```
always @(posedge clk or posedge rst)
begin
    if(rst == 1’b1)
        empty <= 1’b1;
    else
        empty <= empty_m;
end
always @(rd_n or wr_n or empty or p_wr or p_rd or clr)
begin
    if((clr == 1'b1) || ((p_wr == p_rd + 1) && (rd_n == 1'b0) && (wr_n == 1'b1)))
```

```
        empty_m = 1'b1;
    else if((wr_n == 1'b0) && (rd_n == 1'b1))
        empty_m = 1'b0;
    else
        empty_m = empty;
end
```

将这两个模块写成一个是完全可以的，这里只是介绍一种思维方法。以后最好还是写成一个模块。例如对 full 信号可以描述如下：

```
always @(posedge clk or posedge rst)
begin
    if(rst == 1'b1)
        full <= 1'b0;
    else if(clr == 1'b1)
        full <= 1'b0;
    else if((p_wr == (p_rd + 4'b1111)) && (wr_n == 1'b0) && (rd_n == 1'b1))
        full <= 1'b1;
    else if((full == 1'b1) && (rd_n == 1'b0))
        full <= 1'b0;
end
```

对 do 信号也有两种处理方法，一种是使用一个寄存器缓冲，另一种直接输出，这里同样采用使用寄存器的处理方法。

```
always @(posedge clk or posedge rst)
begin
    if(rst == 1'b1)
        do <= 8'h00;
    else if(rd_n == 1'b0)
        do <= elem[p_rd];
end
```

将每个输出信号和中间信号描述完，初步的设计也就基本完成了，可以写下

面的 testbench 进行功能验证。

```
module testbench();
//------------------------------------------------------------------------
//       Signals for mapping Testting module
//------------------------------------------------------------------------
reg rst,clk,clr,wr_n,rd_n;
reg [7:0] di;
wire full_o,empty_o;
wire [7:0] do;
//-------------------- Component under test --------------------------------
fifo16x8 fifo
(.rst(rst)             ,
  .clk(clk)             ,
  .clr(clr)             ,
  .rd_n(rd_n)          ,
  .wr_n(wr_n)          ,
  .di(di)                ,
  .do(do)                ,
  .full(full_o)      ,
  .empty(empty_o));
//---------------------Variable for Simulation ------------------------------
reg[7:0] test_data[9:0];
//------------------------------------------------------------------------
task half_pulse;
inout clock;
begin
       #10 clock = ~ clock;
end
endtask
//------------------------------------------------------------------------
```

```
task pulse;
input[7:0] num;
integer i;
    for(i=0;i<num;i=i+1)
    begin
        half_pulse(clk);
        half_pulse(clk);
    end
endtask
//---------------------------------------------------------------------------
task write_byte;
input[7:0] data;
    begin
        di = data;
        wr_n = 1'b0;
        pulse(1);
        wr_n = 1'b1;
        pulse(1);
    end
endtask
//---------------------------------------------------------------------------
task read_byte;
output[7:0] data_buf;
    begin
        rd_n = 1'b0;
        pulse(1);
        data_buf = do;
        rd_n = 1'b1;
        pulse(1);
    end
```

```
endtask
//-------------------------------------------------------------------------
initial
begin
	rst = 0;
	clk = 0;
	clr = 0;
	rd_n = 1;
	wr_n = 1;
	di = 8'h00;
end
always
begin
	rst = 1;
	pulse(2);
	rst = 0;
	pulse(2);
	write_byte(8'h01);
	write_byte(8'h02);
	write_byte(8'h03);
	read_byte(test_data[0]);	// should be 01H
	if(test_data[0] != 8'h01) begin
		$display("error at step1");
		$stop();
		end
	read_byte(test_data[1]);	// should be 02H
	if(test_data[1] != 8'h02) begin
		$display("error at step2");
		$stop();
		end
```

```
write_byte(8'h11);
write_byte(8'h12);
write_byte(8'h13);                  // should FULL
read_byte(test_data[2]);      // should be 03H
if(test_data[2] != 8'h03) begin
        $display("error at step3");
        $stop();
        end
read_byte(test_data[3]);      // should be 11H
if(test_data[3] != 8'h11) begin
        $display("error at step4");
        $stop();
        end
read_byte(test_data[4]);      // should be 12H
if(test_data[4] != 8'h12) begin
        $display("error at step5");
        $stop();
        end
read_byte(test_data[5]);      // should be 13H
if(test_data[5] != 8'h13) begin
        $display("error at step6");
        $stop();
        end
clr = 1'b1;
pulse(1);
write_byte(8'h01);
read_byte(test_data[0]);      // should be 01H
if(test_data[0] != 8'h01) begin
        $display("error at step7");
        $stop();
```

```
            end
        pulse(2);
        $display("no error found");
        $stop();
end
//--------------------------------------------------------------------
endmodule
```

在这段验证代码中，使用了系统函数来自动判断设计的正确性，这是以后testbench 代码的推荐写法。尽管问题还不是很复杂，但已经可以看出，功能验证是困难的，FIFO 的操作有很多种情况，这段代码只验证了部分功能，覆盖全部可能的情况需要写很长的代码。功能验证不仅需要耐心细致，还需要掌握一定的理论。

从这个例子中，可以看出行为级描述的必要性。在编写代码之前，写好一个包含了对所有问题的解决方法的 SPEC 只是一种理想的情况，事实上，许多问题在编写代码之前不容易想到，只有在代码编写过程中才能反映出来。在实际的设计中，SPEC 也不可避免地要有修改和完善的过程。尽快写出行为级描述可以使SPEC 中的问题暴露出来，也使整个设计队伍对电路功能有统一的认识。所谓“行为级”代码并没有什么特殊写法，只是编写过程中对可综合性和代码质量的考虑要少一些，以实现功能为主要目标。

（3） 编写 RTL 代码

在这个例子中，已完成的“行为”级代码中，已经有许多部分是符合 RTL 代码要求的。人工判断的准则之一是能否从代码中看出电路结构。可以看出所谓“读”“写”指针其实都是计数器，“满”“空”标志部分的电路结构也比较明显，没有必要再修改了。代码中比较抽象的部分是存储单元部分，即 elem 数组。这部分电路结构还不清晰。为避免综合结构与设计思想不同，还应做进一步的描述。在考虑用寄存器实现时，所谓“数组”实际上是寄存器堆，其电路结构如图 4-4。

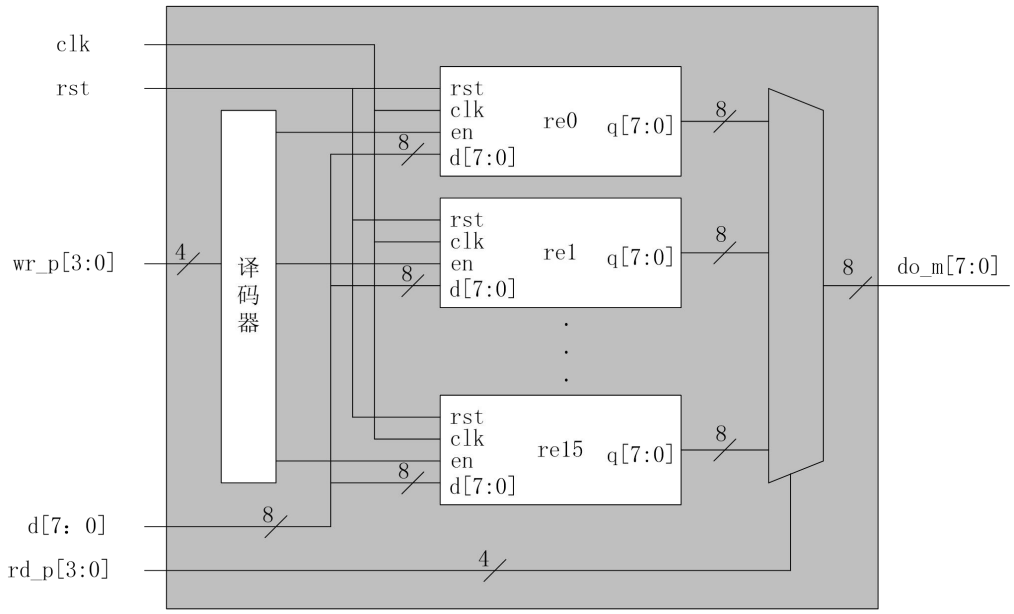


图 4-4 寄存器堆

其中每个 8 位寄存器的结构如图 4-5 所示。对于 RTL 级代码，只要描述到图 4-4 的级别就可以了，写到图 4-5 的级别就有些过细了。

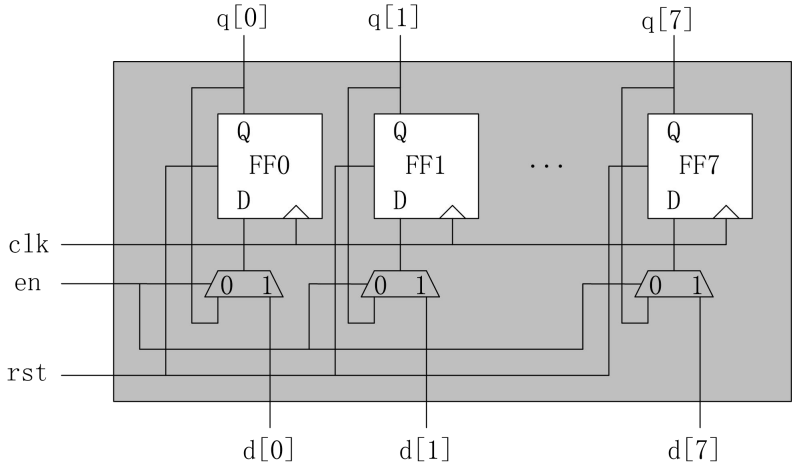


图 4-5 8 位寄存器

事实上，好的综合工具也可能正确综合数组，但为了不过分依赖工具，以下对“寄存器堆”进行 RTL 级描述，代替原有代码中的“数组”。

以下是寄存器堆部分的基本框架代码，具体的内容将填写在中间。这里没有使用复位信号对寄存器堆进行复位，以免使 rst 信号上负载过重。

```
//-----------------------------------------------------------------
// Design unit : rr16
// This is a RTL level description of the registers in FIFO.
//-----------------------------------------------------------------
module rr16(clk,aw,ar,d,wr_n,full,do_m);
input clk;
input[3:0] aw;
input[3:0] ar;
input[7:0] d;
input wr_n;
input full;
output[7:0] do_m;
//-----------------------------------------------------------------
reg[7:0] do_m;
reg[7:0] re[0:15];
//-----------------------------------------------------------------
// add code here
//-----------------------------------------------------------------
endmodule
```

原有的设计代码中应去掉对存储单元部分的声明和操作，在其中添加以下代码“调用”寄存器堆。

```
rr16 fifo_regs
(.clk(clk)       ,
 .aw(p_wr)       ,
 .ar(p_rd)       ,
 .d(di)          ,
 .wr_n(wr_n)     ,
 .full(full)     ,
 .do_m(do_m)
);
```

以下是对寄存器堆的写操作的代码。从这段代码已经可以明确看出电路的结构了。

```
//------------------------------------------------------------------
// No WRITE action when FIFO is full.
//------------------------------------------------------------------
always @(posedge clk)
begin
    if((wr_n == 1'b0) && (full == 1'b0)) begin
        case(aw)
            4'h0   : re[0]   <= d;
            4'h1   : re[1]   <= d;
            4'h2   : re[2]   <= d;
            4'h3   : re[3]   <= d;
            4'h4   : re[4]   <= d;
            4'h5   : re[5]   <= d;
            4'h6   : re[6]   <= d;
            4'h7   : re[7]   <= d;
            4'h8   : re[8]   <= d;
            4'h9   : re[9]   <= d;
            4'ha   : re[10] <= d;
            4'hb   : re[11] <= d;
            4'hc   : re[12] <= d;
            4'hd   : re[13] <= d;
            4'he   : re[14] <= d;
            4'hf   : re[15] <= d;
            endcase
        end
```

最后是对寄存器堆的读取操作部分，这种描述对应的是多路选择器。

```
//------------------------------------------------------------------
// Output do_m is combinational signal in this module.
//------------------------------------------------------------------
always @
(    ar or re[0] or re[1] or re[2] or re[3]
       or re[4] or re[5] or re[6] or re[7]
       or re[8] or re[9] or re[10] or re[11]
       or re[12] or re[13] or re[14] or re[15]
)
begin
      case(ar)
            4'h0    : do_m = re[0];
            4'h1    : do_m = re[1];
            4'h2    : do_m = re[2];
            4'h3    : do_m = re[3];
            4'h4    : do_m = re[4];
            4'h5    : do_m = re[5];
            4'h6    : do_m = re[6];
            4'h7    : do_m = re[7];
            4'h8    : do_m = re[8];
            4'h9    : do_m = re[9];
            4'ha    : do_m = re[10];
            4'hb    : do_m = re[11];
            4'hc    : do_m = re[12];
            4'hd    : do_m = re[13];
            4'he    : do_m = re[14];
            4'hf    : do_m = re[15];
            endcase
end
```

在这个设计中，设计是从 FIFO 的算法分析开始的，代码编写思路与程序设计有一定的相似之处，但编写过程中始终考虑的是电路问题。一个完整的设计除给出 RTL 代码、验证代码外，还要写出说明书。在说明书中要画出电路结构，说明对某些特殊问题的处理方法。

4.2 状态机设计方法

一般情况下，一个数字电路系统总是可以分解为数据通道（Datapath）和状态机两个部分。“数据通道”一词与字面意义不同，泛指较常用的数字部件，具有通用性。状态机则是对电路特殊性部分的描述。这里所说的“状态机”是一种描述电路功能的方法，与数字电路中的状态转换图类似，但要更抽象一些。以下以一个“密码”定时器为例，介绍状态机分析方法。

（1）电路功能描述

- 输入输出信号见图 4-6，只要求设计“密码定时器”模块。
- 上电复位后，定时器处于等待状态，输出控制信号“out”为 0，计时显示输出为 BCD 码“9”。
- 输入正确密码后开始倒计时，每秒计数值减 1，计时回 0 后，输出控制信号置 1。
- 按键事件信号（key_in）的每个脉冲代表一次数字输入事件，输入数字以 bcd 码形式给出。
- 密码为固定的 4 位数字，例如“2012”。
- 密码输入间隔时间或总的输入时间不限。输入数据的总位数也不限，只要 4 个连续的输入数据与密码相同即可。

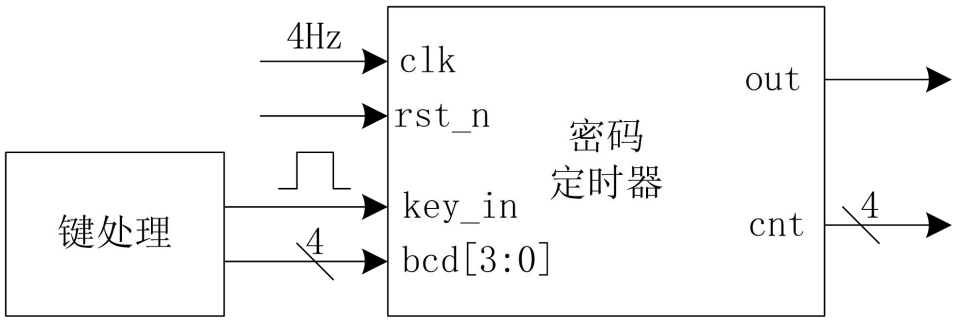


图 4-6 密码定时器

设计思路：可以想象电路中将包含分频电路、BCD 减法计数器等常用部件，这些属于“数据通道”部分，功能比较简单。问题的关键是如何描述密码的输入情况。用寄存器保存输入密码的方法可行，但耗用较多硬件资源。以下采用状态机

方法描述密码输入情况。

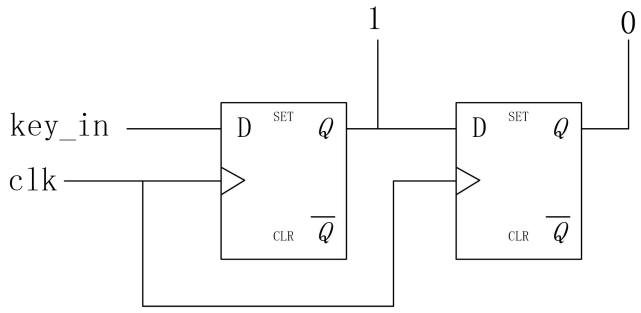


图 4-7 边沿检测

```
reg[1:0] key_in_edge;
always @(posedge clk)
begin
    if(rst_n == 1'b0)
        key_in_edge <= 2'b11;
    else begin
        key_in_edge[1] <= key_in_edge[0];
        key_in_edge[0] <= key_in;
        end
end
```

```
always @(posedge clk)
begin
    if(rst_n == 1'b0)
        div <= 2'b00;
    else
        div <= div + 1'b1;
end
//-------------------------------------------------------------
always @(posedge clk)
begin
    if(rst_n == 1'b0)
        cnt <= 4'd9;
    else if((div == 2'b11) && (state == PASS_4) && (cnt > 0))
        cnt <= cnt - 1'b1;
end
```

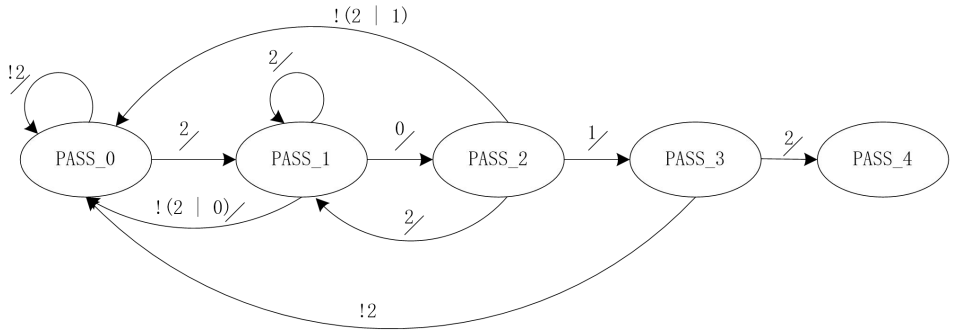


图 4-8 状态机

```
parameter[2:0] PASS_0 = 3'b00;
parameter[2:0] PASS_1 = 3'b01;
parameter[2:0] PASS_2 = 3'b10;
parameter[2:0] PASS_3 = 3'b11;
parameter[2:0] PASS_4 = 3'b11;
```

```
always @(posedge clk)
begin
    if(rst_n == 1'b0)
        state <= PASS_0;
    else
        state <= next_state;
end
```

```
always @(state or key_in_edge or bcd)
begin
    if(key_in_edge == 3'b01) begin
        case(state)
            PASS_0:begin
                    if(bcd == 3'd2)
                        next_state = PASS_1;
                    else
                        next_state = PASS_0;
                    end
            PASS_1:begin
                    if(bcd == 3'd0)
                        next_state = PASS_2;
                    else if(bcd == 3'd2)
                        next_state = PASS_1;
                    else
                        next_state = PASS_0;
                    end
            PASS_2:begin
                    if(bcd == 3'd1)
                        next_state = PASS_3;
                    else if(bcd == 3'd2)
                        next_state = PASS_1;
                    else
                        next_state = PASS_0;
                    end
            PASS_3:begin
                    if(bcd == 3'd2)
                        next_state = PASS_4;
                    else
                        next_state = PASS_0;
                    end
            default:    next_state = state;
            endcase
        end
    else
        next_state = state;
end
```

4.3 状态机描述练习

（1）报警器问题

（2）售货机问题

（3）通信协议问题

第 5 章 数字系统设计

5.1 引言

本章介绍一个数字芯片系统的系统分析、架构设计等前端设计过程，RTL 代码编写和验证工作是一个大作业，在后续章节中，完成该芯片的其它设计环节。

5.2 用户需求

许多单片机设计者反映，在应用系统设计时，经常需要精确测量两个物理量之差，例如温度差、压力差等。高分辨率 A/D 转换器通常采用 SPI 接口，使用单片机直接读取数据在时间上的同步。在单片机（MCU）系统中，处理器可能有多个比采样优先级更高的任务，分别读取两个采样通道的数据很难保证是同一时刻的测量值，有必要使用一种接口芯片来提高数据的同步性。

5.3 系统分析

5.3.1 用户需求分析

根据用户需求，考虑设计一种可提高同步精度的接口芯片，其应用示意图见图 6-1。图中的“SITF”有 3 个 SPI 接口，其中两个工作在“主”模式下，连接两个 A/D 转换器，一个工作在“从”模式下，连接 MCU。

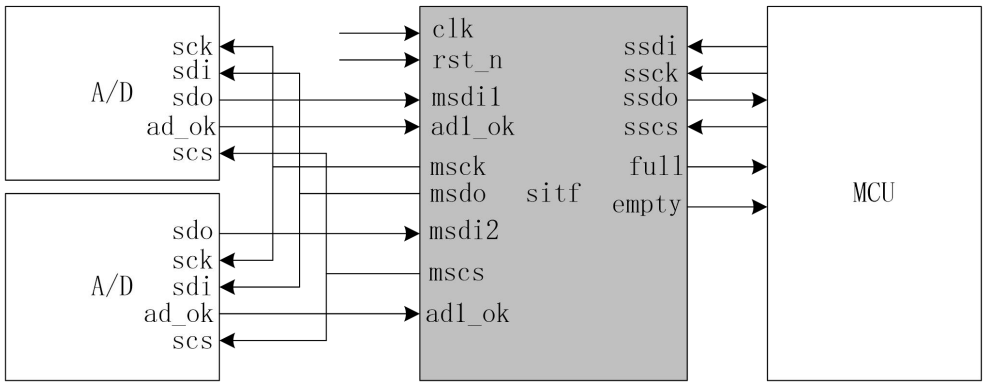


图 5-1 应用系统示意图

假设两个 A/D 转换器是同型号的芯片，可同时启动转换，待转换结束时同时读取采样值，分别保存到内部 FIFO 中，两个通道的采样值之差保存到另一个 FIFO 中。MCU 可分别读取 3 种数据，采样值在时间上的误差仅取决于两个 A/D 转换器的速度误差。

为提高芯片的通用性，应考虑以下问题：

（1） 应能连接多种具有 SPI 口的 A/D 转换器。

为简化问题，可假设系统中的两个 A/D 是同型号的，但固定具体型号。接口芯片应能支持多种分辨率和不同命令格式的 SPI 接口 A/D 转换器。

（2） 应有多种循环采样模式。

每个 A/D 转换器本身可能具有多个采样通道，应能支持使对应通道同步的多种采样循环模式。

5.3.2 相关知识

（1）SPI 接口

SPI 接口是 Motorola 首先提出的全双工三线同步串行外围接口，采用主从模式（Master Slave）架构；支持多 slave 模式应用，一般仅支持单 Master。

从概念上讲，SPI 口相当于将两个设备中的移位寄存器串联起来，通信就是将数据在两个 2 个移位寄存器中进行交换（见图 5-2）。时钟由 Master 控制，在时钟移位脉冲下，**数据按位传输，高位在前，低位在后**（MSB first）；SPI 接口有 2 根单向数据线，为全双工通信，目前应用中的数据速率可达几 Mbps 的水平。

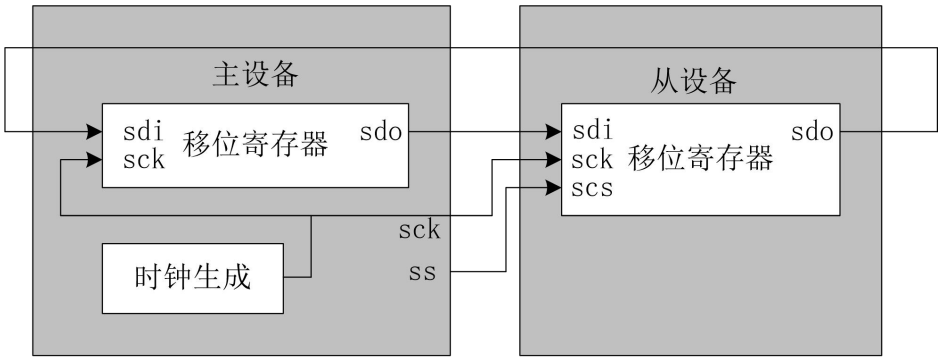


图 6-2 SPI 接口数据传输

SPI 接口共有 4 根信号线，分别是：设备选择线、时钟线、串行输出数据线、串行输入数据线。

设备选择线一般用 SS（Slave select）或 SCS 表示，用于选择激活某 Slave 设备，低有效，由 Master 驱动输出。只有当 SS-信号线为低电平时，对应 Slave 设备的 SPI 接口才处于工作状态。SPI 接口数据线是单向的，共有两根数据线，分别承担 Master 到 Slave、Slave 到 Master 的数据传输；但是不同厂家的数据线

命名有差别。Motorola 的经典命名是 MOSI（Master Out Slave In）和 MISO（Master In Slave Out）。这种命名主要用于主、从模式可组态的器件，作为主设备使用时，MOSI 是输出，而作为从设备使用时，MOSI 输入。MISO 线同理定义。因此在电路板上，Master 的 MOSI 引脚应与 Slave 的 MOSI 引脚连接在一起。双方的 MISO 也应该连在一起，而不是一方的 MOSI 连接另一方的 MISO。更多厂家是按照类似 SDI，SDO 的方式来命名，这是站在器件的角度根据数据流向来定义的。SDI 为串行数据输入，SDO 为串行数据输出。这种情况下，当 Master 与 Slave 连接时，就应该用一方的 SDO 连接另一个方的 SDI。

为了适用不同产品接口应用需要，SPI 接口定义了多种时序传输模式，并可通过设置接口单元寄存器中的相关控制位来选择。在 Motorola 的产品中，时序即是由两个控制位（极性控制、相位控制）来控制的。

时钟极性选择位（CPOL）用来选择在设备被使能激活后，还未进行数据传输时或两个字节数据传输间歇期间时，SCK 的空闲（Idle）电平。时钟相位选择位（CPHA）用来选择数据接收端设备的采样时刻。可能在 Idle to Active 的跳变沿，也可能在 Active to Idle 的跳变沿。在该采样时刻，线上数据必须已经稳定可靠，因此数据发送端设备应提前将数据移出到数据线上。图 5-3 是一种典型工作时序。

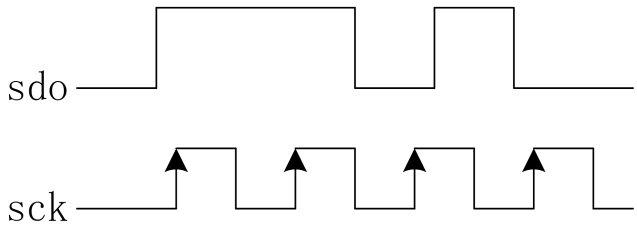
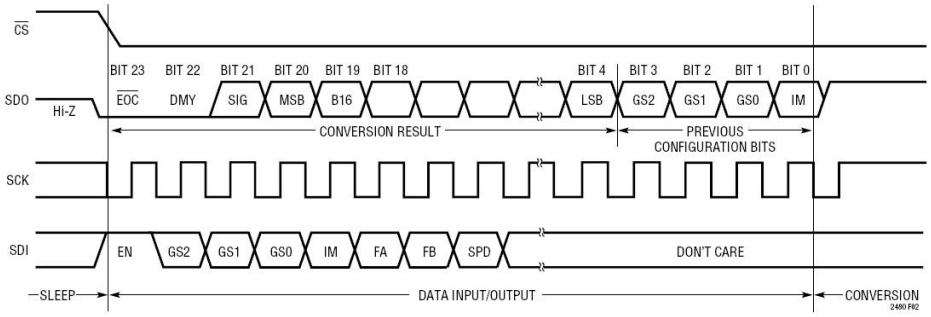


图 5-3 数据在时钟上沿前稳定的时序

关于 SPI 口的详细定义还需查找有关资料。常用的 A/D 转换器并不是标准 SPI 设备，一般只支持部分传输模式。

（2）常用 A/D 转换器命令格式

图 5-4 是 LTC2480 的传输时序，这种 A/D 转换器的 SCK 空闲电平为高电平，向 A/D 转换器发送数据时，应在 SCK=0 时，在 SDO（A/D 的 SDI）端准备数据，SCK 上沿后，数据被移入。



5-4 LT2480 的数据

这种 A/D 转换器具有 16 位分辨率，发送和接收数据共需要 24 个 SCK 时钟。发送结束后，SCK 应为高电平。A/D 开始转换后，其 SDO 端为高阻态，转换结束后，SDO 为低电平。

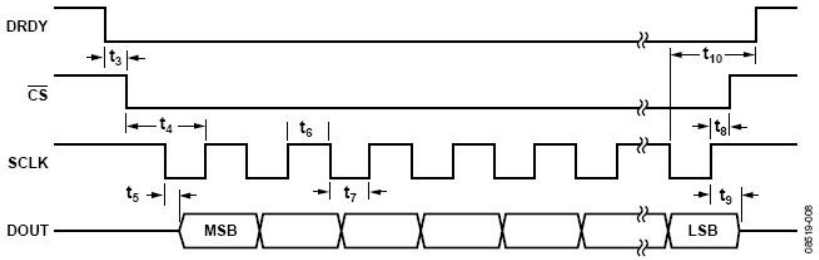


Figure 8. Read Cycle Timing Diagram

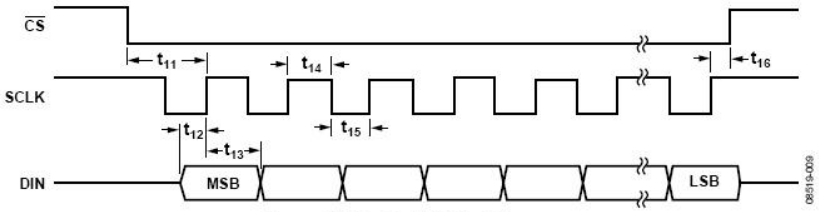


Figure 9. Write Cycle Timing Diagram

图 5-5 AD7715 的数据传输时序

图 5-5 是 AD7715 的时序，其基本格式于前一种相同，但高层协议很复杂，详见有关资料。

5.4 设计思路

5.4.1 基本设计思想

（1）采用同步单时钟模式设计

尽管从概念上讲，SPI 似乎是以 SCK 为时钟的，但按这种思路进行将导致

电路成为异步电路。异步设计很复杂，DC 不支持。按同步方式实现，需要一个频率较高的系统时钟，至少需要比最高的 SCK 时钟频率高一倍。数据同步可以用检测 SCK 的跳变的方式实现。

（3） 解决兼容性问题的思路

- 保留一种直通（BYPASS）模式，使 MCU 可直接控制 A/D，完成初始化工作。
- 定义多个（例如 3 个）命令寄存器，用于存放向 A/D 写入的各种操作命令，以便构成工作状态下的各种采样“循环模式”（见图 5-6）。
- 定义一个控制寄存器和一个发送长度寄存器，用于选择循环模式和发送长度。
- 定义一个控制寄存器，选择两次采样之间是否拉高 SS 线等问题。

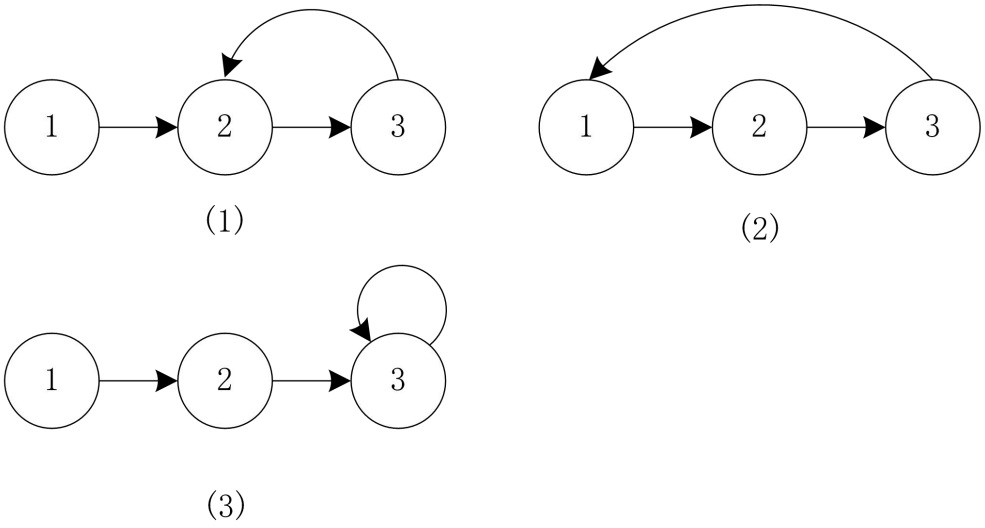


图 5-6 向 A/D 写入命令的模式

- 定义一个 24 位寄存器用于接收从 A/D 转换器读回的数据。
- 定义一个接收数据长度寄存器，使接口电路可以读不同长度的数据。
- 定义一个有效数据位置寄存器，使接口电路适应各种数据格式。

（4） 定义单片机的各种操作命令

在确定以上内容后，要确定 MCU 对接口芯片的操作方法。包括读、写各种寄存器，以及启动和复位 A/D 等操作。

命令	功能	编码
复位		
启动		
写寄存器 C0		
写寄存器 C1		
写寄存器 C2		
写寄存器 M0		

（5）确定芯片的工作状态

画出状态机。

（6）确定实现工艺等问题。

以上内容属于系统功能设计问题。完成之后应写出详细设计说明书。

6.5 芯片规格书

（1）端口定义

（2）内部资源

（3）主要工作状态

（4）主要性能

6.5 架构设计