第 5 章　Verilog HDL 程序结构

在传统的 Verilog HDL 书籍中，往往先讲述 Verilog HDL 语言的基本语法，直到最后才介绍 Verilog HDL 的程序结构，造成读者学习没有明确的目的性，不知道掌握的诸多语法如何使用，不宜掌握知识点。因此本书以 Verilog HDL 程序结构的说明和层次化设计方法为开启学习 Verilog HDL 语言的大门，让读者先从宏观上了解 Verilog HDL，再深入到语法细节，明白各条语句出现的背景和使用方法。这样不仅可以避免出现“只见树木，不见森林”的缺陷，还可以在后续学习中通过实例快速验证语法，真切感受 Verilog HDL 的硬件描述魅力。

5.1　程 序 模 块

Verilog HDL 程序是由模块构成的，每个模块的内容都嵌在 module 和 endmodule 两个语句之间。每个模块实现特定的功能，模块可以进行层次嵌套。每个模块都要进行端口定义，并说明输入/输出口，然后对模块的功能进行行为逻辑描述。

5.1.1　Verilog HDL 模块的概念

模块（module）是 Verilog HDL 最基本的概念，也是最常用的基本单元，用于描述某个设计的功能或结构以及相同通信的接口。模块的实际意义是代表硬件电路上的逻辑实体，每个模块都实现特定的功能。

例如一个实现了 2 输入加法器的模块就对应着一个 2 输入加法电路，同样可以被所有需要实现 2 输入加法的模块调用。由此可见，模块对应着的硬件电路之间是并行运行的，也是分层的，高层模块通过调用、连接低层模块的实例实现复杂的功能。如果要将所有的功能模块连接成一个完整系统，则需要一个模块将所有的子模块连接起来，这一模块被称为顶层模块（Top Module）。

在读者所熟悉的 C 语言中，一个.c 文件可以实现多个函数。与此类似，一个 Verilog HDL 文件（.v）也可以实现多个模块，但为了便于管理，一般建议一个.v 文件实现一个模块。需要注意的是，无论是面向综合的程序，还是面向仿真的程序，都需要以模块的形式给出，且模块的结构是一致的，只存在语句上的差别。

5.1.2　模块的基本结构

一个 Verilog HDL 模块的完整结构示例如下。

```
module
module_name (port_list)     //声明各种变量、信号
    reg                     //寄存器
    wire                    //线网
    parameter               //参数
    input                   //输入信号
    output                  //输出信号
    inout                   //输入/输出信号
    function                //函数
    task                    //任务
    ……
    //程序代码
    initial assignment
    always assignment
    module assignment
    gate assignment
    UDP assignment
    continous assignment
endmodule
```

声明部分用于定义不同的项，例如模块描述中使用的寄存器和参数。语句用于定义设计的功能和结构。声明部分可以分散于模块的任何地方，但是变量、寄存器、线网和参数等的声明必须在使用前出现。

在实际中，一个 Verilog HDL 模块并不需要具备所有的结构特征，基本的模块结构已经能够满足大多数设计。下面给出模块的基本结构。

```
module <模块名> (<端口列表>)
    <定义>
    <模块条目>
endmodule
```

其中，<模块名>是模块的唯一性标志符；<端口列表>定义了和其余模块进行通信连接的信号，根据数据流方向，可以分为输入、输出和双向端口 3 类；<定义>用来指定数据对象为寄存器型、存储器型、线型以及过程块；<模块条目>可以是 initial 结构、always 结构、连续赋值或模块实例。

下面给出一个简单的 Verilog HDL 模块，它实现了 3～8 线译码功能。

【例 5-1】 3～8 线译码器的 Verilog HDL 实现。

```
module decoder3to8(
    din, dout
    );
    input [2:0] din;
    output [7:0] dout;
    reg [7:0] dout;
    always @ (din) begin
     case(din)
        3'b000: dout <= 8'b0000_0001;
        3'b001: dout <= 8'b0000_0010;
        3'b010: dout <= 8'b0000_0100;
        3'b011: dout <= 8'b0000_1000;
        3'b100: dout <= 8'b0001_0000;
        3'b101: dout <= 8'b0010_0000;
        3'b110: dout <= 8'b0100_0000;
        3'b111: dout <= 8'b1000_0000;
```

```
    endcase
end
endmodule
```

5.1.3　端口声明

模块端口是指模块与外界交互信息的接口，包括以下 3 种类型。

（1）input：输入端口，模块从外界读取数据的接口，在模块内不可写。

（2）output：输出端口，模块往外界送出数据的接口，在模块内不可读。

（3）inout：输入/输出端口，也称双向端口，可读取数据也可以送出数据，数据可双向流动。

上述 3 类端口中，input 端口只能为线网型数据类型；output 端口可以为线网型，也可以为寄存器数据类型；而对于 inout 端口由于其具备输入端口特点，所以也只能声明为线网型数据类型（这里只给出端口的简要说明，各数据类型将在 6.2 节进行介绍）。

端口位宽由[M:N]定义，如果 M>N，则为降序排列，[M]位是有效数据的最高位，[N]是有效数据的最低位，其等效位宽为 M-N+1；如果 M<N，则为升序排列，[M]位是有效数据的最低位，[N]是有效数据的最高位，其等效位宽为 N-M+1。下面给出一些常用的端口表示实例。

```
output [15:0] crc_reg;
input [17 : 0] din;
input wr_en;
output [ 0: 17] dout;
```

5.2　Verilog HDL 的层次化设计

层次化设计的核心思想有两个：一是模块化，二是模块例化（也就是程序调用）。本节主要介绍如何在 Verilog HDL 程序中实现模块调用和系统的层次化设计，并给出在 ISE 中与图形化设计结合的方法。

5.2.1　Verilog HDL 层次化设计的表现形式

层次化设计，简单来讲就是在利用 Verilog HDL 语言来编写程序实现相应功能时，不需要把所有的程序写在一个模块中。如果将系统中所有功能都放在一个模块中，那么其错误的检查、功能验证和调试的难度及复杂度将是无法想象的。

层次化设计方法的基本思想是分模块、分层次地进行设计描述。描述系统总功能的设计为顶层设计，描述系统中较小单元的设计为底层设计。整个设计过程可理解为从硬件的顶层抽象描述向最底层结构描述的一系列转换过程，直到最后得到可实现的硬件单元描述为止。层次化设计所用的模块有两种：一是预先设计好的标准模块；二是由用户设计的具有特定应用功能的模块。

在基于 Verilog HDL 的层次化设计中，最明显的特征就是具备多个不同级别的 Verilog HDL 代码模块，除顶层模块外，每个模块完成一项较为独立的功能。

5.2.2 模块例化

Verilog HDL 的模块例化也称作程序调用，指将已存在的 Verilog HDL 模块作为当前设计的一个组件，设计人员将其视为黑盒子，直接向其输入即可得到相应的输出信号。通过程序例化，可在顶层模块中，将各底层元件用 Verilog HDL 语言连接起来；逐次封装，形成最终的顶层文件，以满足系统要求。

Verilog HDL 语言有 3 种模块调用方法：位置映射法、信号名映射法以及二者的混合映射法。

1．位置映射法

位置映射法严格按照模块定义的端口顺序来连接，不用注明原模块定义时规定的端口名，其语法为：

```
模块名 例化名 (连接端口 1 信号名, 连接端口 2 信号名, 连接端口 3 信号名,……);
```

下面给出一个位置映射法的例化实例。

【例 5-2】 利用位置映射法的 Verilog HDL 调用实例。

下面实现一个 4 输入的相等比较器，假设系统有 4 个输入端口，分别为 a0、a1、b0 和 b1，要求判断 a0 和 b0 是否相等，a1 和 b1 是否相等。

通过分析可以发现，a0、b0 和 a1、b1 的比较是相同操作，因此只要实现一个相等比较器，然后调用两次即可达到设计目的。这样在验证时只需要验证比较器这一个子模块即可，从而达到简化设计的目的。下面给出相等比较器的 Verilog HDL 代码。

```
module compare_core(
    result, a , b
    );
  input [7:0] a, b;
  output result;
  //判断两输入是否相等，相等输出 1，否则输出 0
  assign result = (a == b) ? 1 : 0;
endmodule
```

其次，在应用的顶层模块中两次调用比较器子模块，其代码如下。

```
module compare_app0(
    result0, a0 , b0,
    result1, a1 , b1
    );
  input [7:0] a0, b0, a1, b1;
  output result0, result1;
  //第一次调用比较器子模块，利用位置映射法
  compare_core inst_compare_core0(
  result0, a0,b0
  );
  //第二次调用比较器子模块，利用位置映射法
  compare_core inst_compare_core1(
  result1, a1,b1
  );
endmodule
```

顶层模块在 ISE 软件中综合后的 RTL 级结构图如图 5-1 所示，可以看出它两次调用了比较器子模块来达到设计目的。

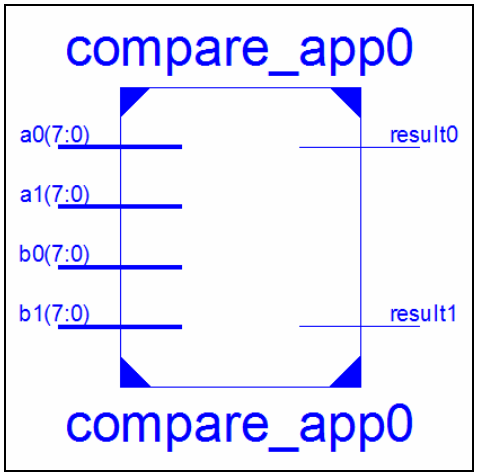

图 5-1　采用位置映射法的比较器应用模块 RTL 结构示意图

2．信号名映射法

信号名映射法，即利用“.”符号，表明原模块定义时的端口名，其语法为：

```
模块名 例化名
(.端口 1 信号名(连接端口 1 信号名),
.端口 2 信号名(连接端口 2 信号名),
.端口 3 信号名(连接端口 3 信号名),…);
```

显然，信号名映射法同时将信号名和被引用端口名列出来，不必严格遵守端口顺序，不仅降低了代码易错性，还提高了程序的可读性和可移植性。因此，在良好的代码中，严禁使用位置映射法，应全部采用信号名映射法。

【例 5-3】 将例 5-2 的模块调用通过信号名映射法实现。

```
module compare_app1(
    result0, a0 , b0,
    result1, a1 , b1
    );
  input [7:0] a0, b0, a1, b1;
  output result0, result1;
  //第一次调用比较器子模块，利用信号名映射法
  compare_core inst_compare_core0(
    .result(result0),
    .a(a0),
    .b(b0)
  );
  //第二次调用比较器子模块，利用信号名映射法
  compare_core inst_compare_core1(
    .a(a1),
    .b(b1),
    .result(result1)
    );
endmodule
```

上述程序在 ISE 中综合后的 RTL 级结构图如图 5-2 所示，可以看出它和图 5-1 是一致的，说明例 5-2 的代码和例 5-1 的代码是等效的。

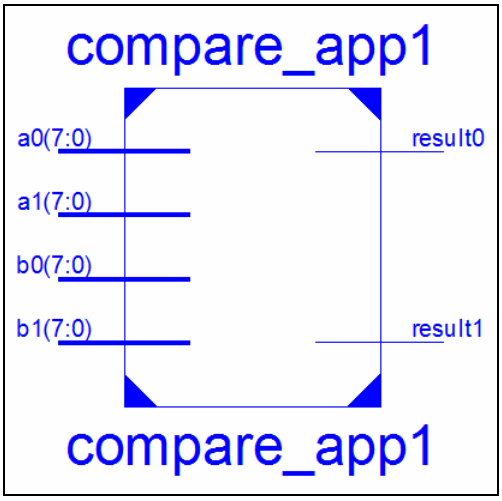

图 5-2　采用信号名映射法的比较器应用模块 RTL 结构示意图

3．特殊处理说明

例 5-1 和例 5-2 给出了基本完整的模块例化方法，但在实际中，还有大量的异常情况需要处理，包括部分输入、输出端口不用，某一端口位宽不匹配等异常情况。下面对这些特殊情况的应用进行说明。

1）悬空端口处理

在我们的例化实例中，被调用模块的某些管脚可能不需要使用，因此要在例化时进行特殊处理。首先需要说明的是，悬空例化模块端口只能在信号名映射法中完成，其方法有两种，下面以实例分别说明。

【例 5-4】 悬空端口处理。

在模块例化时，相应的端口映射采用空白处理。

```
DFF d1 (
  .Q(QS),
  .Qbar ( ),          //该管脚悬空
  .Data (D ) ,
  .Preset ( ),        //该管脚悬空
  .Clock (CK)
  );                  //信号名映射法
```

另一方法是直接在例化时，不调用该端口，其示例代码如下。

```
DFF d1 (
  .Q(QS),
  //.Qbar ( ),        //该管脚悬空
  .Data (D ) ,
  //.Preset ( ),      //该管脚悬空
  .Clock (CK)
  );                  //信号名映射法
```

需要说明的是：在模块例化时，如果将输入管脚悬空，则该管脚输入为高阻态 Z；如果将输出管脚悬空，则该输出管脚废弃不用。

2）不同端口位宽的处理

模块例化的另一大类异常是端口的位宽匹配问题，其处理原则为：当模块例化端口和被例化模块端口的位宽不同时，端口通过无符号数的右对齐截断方式进行匹配。下面通过一个实例来说明上述处理原则。

【例 5-5】 Verilog HDL 模块例化时端口位宽不匹配的处理实例。

被调用的子模块 Child 代码如下。

```
module Child (Pba, Ppy) ;
    input [5:0] Pba;
    output [2:0] Ppy;
    assign Ppy[2] = Pba[5] | Pba[4];
    assign Ppy[1] = Pba[3] && Pba[2];
    assign Ppy[0] = Pba[1] | Pba[0];
endmodule
```

顶层模块 Top 的代码如下。

```
module Top(Bdl, Mpr);
    input [1:2] Bdl;
    output [2:6] Mpr;
    //采用位置映射法例化模块 Child
    Child C1 (Bdl, Mpr) ;
endmodule
```

在 Child 模块的实例中，根据位宽不匹配的异常处理原则：Bdl[2]连接到 Pba[0]，Bdl[1]连接到 Pba[1]，余下的输入端口 Pba[5]、Pba[4]和 Pba[3]悬空，因此为高阻态 Z。与之相似，Mpr[6]连接到 Ppy[0]，Mpr[5]连接到 Ppy[1]，Mpr[4]连接到 Ppy[2]，如图 5-3 所示。

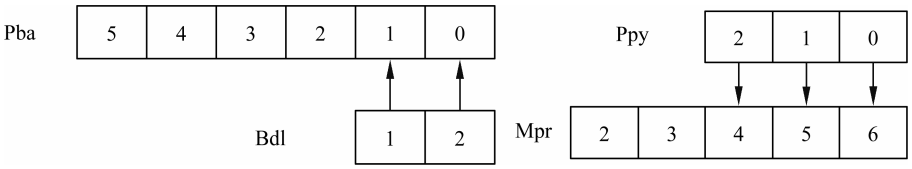

图 5-3　例 5-5 端口匹配示意图

在 EDA 软件操作中，上例实现究竟是怎样的呢？将上述代码添加到 ISE 中，将 Top.v 设置成顶层模块，综合后的 RTL 结构如图 5-4 所示。单击选中 Child 模块的输入、输出端口信号线时，可在 ECS 页面查阅相应的信号线名称。

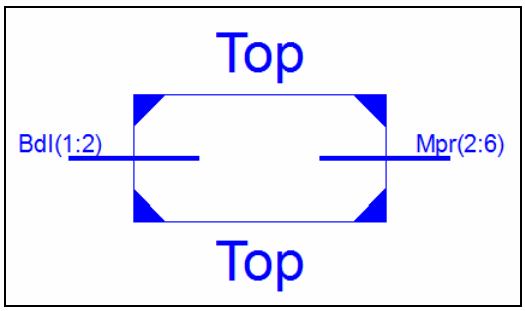

图 5-4　例 5-5 综合后的 RTL 级结构示意图

5.2.3 参数映射

参数映射的功能是实现参数化元件。所谓的“参数化元件”是指元件的某些参数是可调的，通过调整这些参数可实现结构类似而功能不同的电路。在应用中，很多电路都可采用参数映射的方法来达到统一设计，如计数器、分频器、不同位宽的加法器以及不同刷新频率的 VGA 视频接口驱动电路等。

1．参数定义

在 Verilog HDL 中用 parameter 来定义参数，即用 parameter 来定义一个标志符表示一个固定的参数。采用该类型可以提高程序的可读性和可维护性。parameter 型信号的定义格式如下。

```
parameter 参数名 1 = 数据名 1;
```

下面给出几个例子。

```
parameter s1 = 1;
parameter [3:0] S0=4'h0,
    S1=4'h1,
    S2=4'h2,
    S3=4'h3,
    S4=4'h4;
```

参数值的作用域为声明所在的整个.v 文件，其数值可以在编译时被改变。参数值可以使用参数定义语句改变或通过在模块初始化语句中定义来实现。

2．参数传递

参数传递是指在编译时对参数重新赋值而改变其值。传递的参数是子模块中定义的 parameter，其传递方法有下面两种。

1）使用“#”符号

在同一模块中使用“#”符号传递参数。参数赋值的顺序必须与原始模块中参数定义的顺序相同，并不一定要给所有的参数都赋予新值，但不允许跳过任何一个参数，即使是保持不变的值也要写在相应的位置上。其定义格式如下。

```
module_name #( parameter1, parameter2) inst_name( port_map);
module_name #( .parameter_name(para_value), .parameter_name(para_value))
inst_name (port map);
```

【例 5-6】 通过“#”字符实现一个模值可调的加 1 计数器。

顶层模块的代码如下。

```
module param_counter(
    clk_in, reset, cnt_out
    );
  input clk_in;
  input reset;
  output [15:0] cnt_out;
  //参数化调用，利用#符号将计数器的模值 10 传入被调用模块
  cnt #(10) inst_cnt(
```

```
  .clk_in(clk_in),
  .reset(reset),
  .cnt_out(cnt_out)
  );
  endmodule
```

被例化的参数化计数器代码如下。

```
module cnt(
    clk_in, reset, cnt_out
    );
    //定义参数化变量
    parameter [15:0]Cmax = 1024;
    input clk_in;
    input reset;
    output [15:0] cnt_out;
    reg [15:0] cnt_out;
    //完成模值可控的计数器
    always @(posedge clk_in) begin
    if(!reset)
      cnt_out <= 0;
    else
      if(cnt_out == Cmax)
        cnt_out <= 0;
      else
        cnt_out <= cnt_out + 1;
    end
endmodule
```

整个程序实现不同模值的计数器，从结构上分为两部分：一部分由计数器本身的功能实现，另一部分由元件定义和参数化调用部分实现。计数器的实现包含 parameter 语句，在元件定义和调用时，通过“#”符号来传递参数。

2）使用关键字 defparam

关键字 defparam 可以在上层模块直接修改下层模块的参数值，从而实现参数化调用，其语法格式如下。

```
defparam heirarchy_path.parameter_name = value;
```

这种方法与例化分开，参数需要由绝对路径来指定。参数传递时各个参数值的排列次序必须与被调用模块中各个参数的次序保持一致，并且参数值和参数的个数也必须相同。如果只希望对被调用模块内的个别参数进行更改，所有不需要更改的参数值也必须按对应参数的顺序在参数值列表中全部列出（原值复制）。使用 defparam 语句进行重新赋值时必须参照原参数的名字生成分级参数名。

【例 5-7】 通过 defparam 实现一个模值可调的加 1 计数器，计数器的最大值为 12，要求其功能和例 5-6 一致。

```
module param_counter(
    clk_in, reset, cnt_out
    );
  input clk_in;
  input reset;
  output [15:0] cnt_out;
  //调用计数器子模块
  cnt inst_cnt(
  .clk_in(clk_in),
```

```
  .reset(reset),
  .cnt_out(cnt_out)
  );
  //通过 defparam 参数指定例化模块的内部参数
  defparam inst_cnt.Cmax = 12;
endmodule
```

5.2.4 在 ISE 中通过图形化方式实现层次化设计

随着设计规模的增大，原理图输入法已不太适合单独应用于 FPGA 设计中，但由于其具备直观、清晰的特点，常和 HDL 设计混合使用，即通过 HDL 语言设计底层的复杂功能模块，来构建顶层模块，类似于利用原理图绘制软件设计整个系统。因此下面介绍通过原理图输入法建立顶层模块的方法。

1. 建立用户设计的图形化表示符号

只有将 HDL 模块转化成图形化符号才能在原理图输入法中调用，ISE 14.7 提供了上述转换功能。在工程中建立 HDL 模块，完成 HDL 仿真测试以及综合后，用鼠标选中该模块，在过程管理区单击 Design Utilities 项前面的“+”号，双击 Creat Schematic Symbol 选项，即可生成该模块的图形化符号，如图 5-5 所示。

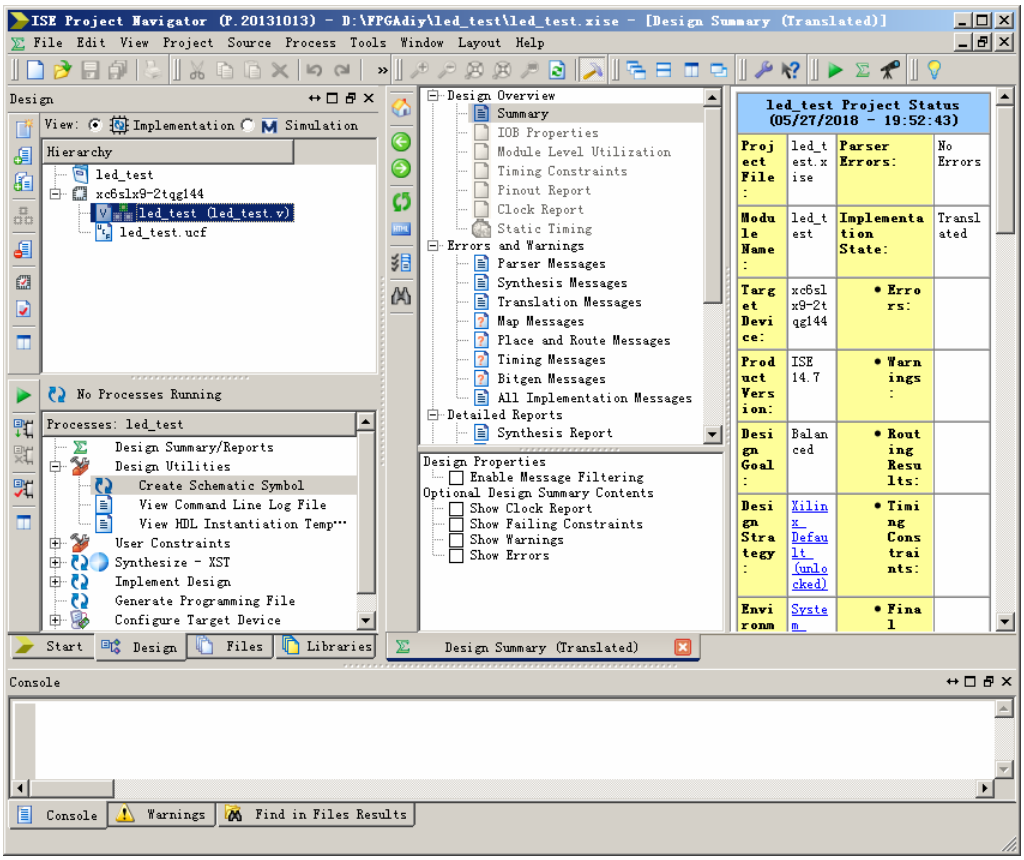

图 5-5　生成图形化符号界面

2．利用原理图法构建顶层模块

1）原理图设计法的输入界面

在当前工程中，新建 Schematic（原理图）类型的文件，然后在工程区的 Source 页面双击该文件，可打开原理图设计界面，如图 5-6 所示。

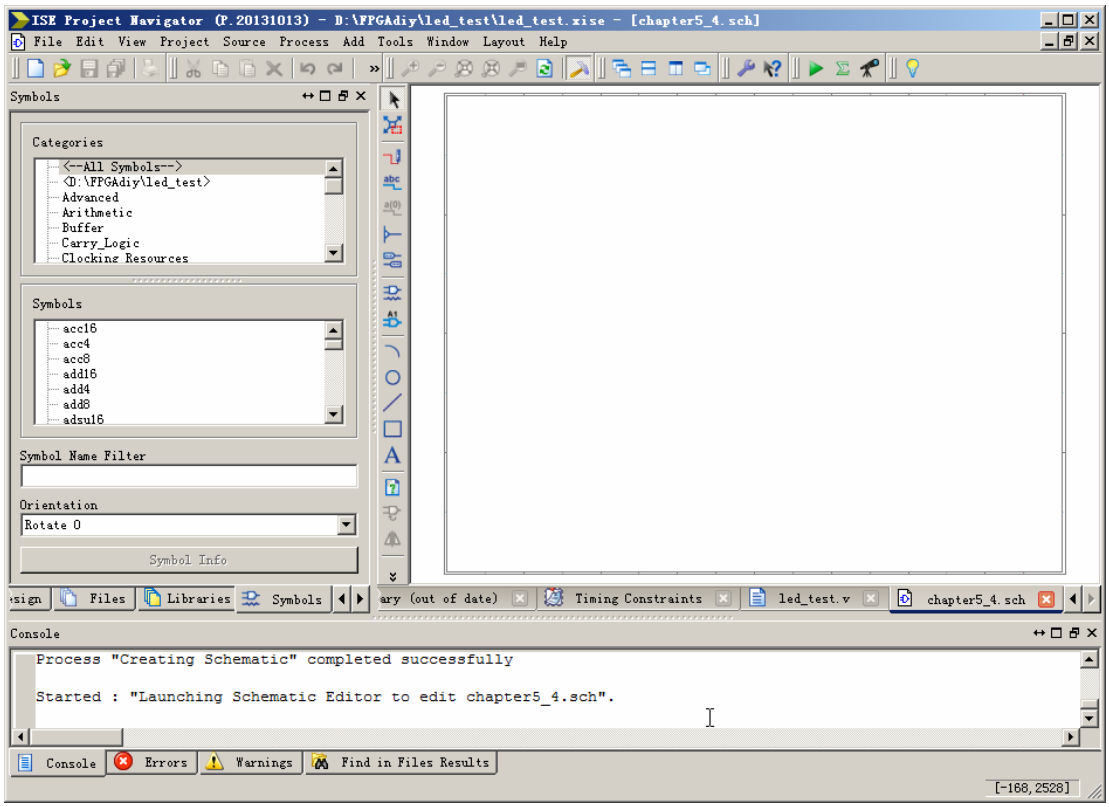

图 5-6　原理图设计界面

2）原理图设计法的基本操作

原理图设计的元件库中包含了固有的图形化组件和用户自定义的图形化模块组件，前者包含了数字电路中所有的基本单元和 Xilinx 系列 FPGA 中集成的硬核模块，如与门、非门、加法器、复用器、乘法器、块 RAM 和 PowerPC 处理器等。在原理图混合设计中，最常用且不可缺少的是 I/O 端口组件，因为用户自定义的图形化模块是不包括 I/O 逻辑的，因此需要添加元件库中的 I/O 单元，才能构成完整的电路。

❑ 添加用户自定义模块

在 Categories 栏选择当前工程路径条目，则 Symbols 栏会列出当前工程中用户自定义的所有图形化模块组件。单击目标组件，然后移动鼠标指针到设计区，会发现鼠标指针变成“+”形，且附带着图形化单元，在合适的位置单击，即可添加该组件。添加完成后，右击或按 Esc 键，可将鼠标指针形状恢复原状。

❑ 添加 I/O 单元

添加 I/O 单元的方法和添加用户自定义模块的方法是相似的，只是 I/O 模块需要通过单击工具栏的 图标得到，且 I/O 单元必须在其余功能元件添加之后才能添加。

添加的 I/O 单元会自动根据元件管脚的方向属性调整为输入或输出，同时也会自动调整位宽。在添加后，可通过双击 I/O 单元来修改管脚名称，编辑界面如图 5-7 所示。

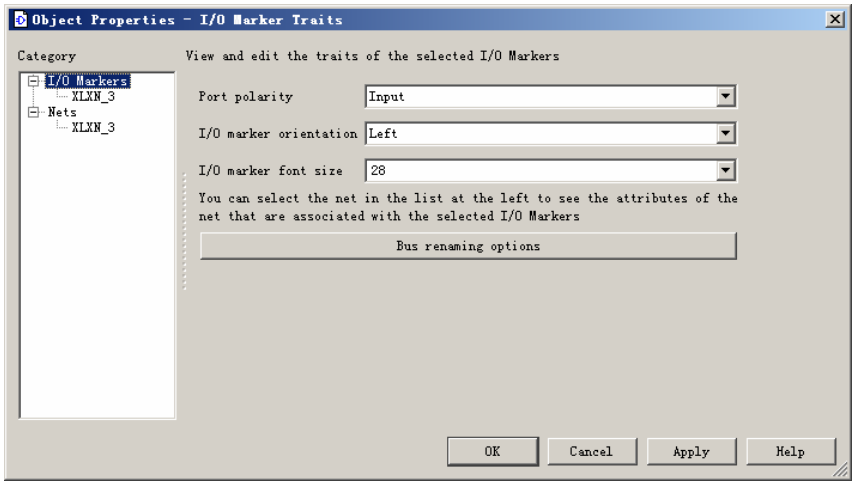

图 5-7　管脚编辑界面

3．应用实例

【例 5-8】 在原理图设计中调用例 5-6 的 16 加 1 计数器，形成完整设计。

（1）完成综合，并在过程管理区选择 Design Utilities 项的 Create Schematic Symbol 选项，生成计数器的图形化符号。

（2）在 ISE 工程管理区的 Source 页面，新建 Schematic 源文件，并命名为 mysch。

（3）双击 mysch，进入原理图编辑页面，在 Categories 栏选择 Counter，然后在 Symbols 选择 cb16ce，并将鼠标指针移动到原理图编辑区，单击左键添加 cb16ce。

（4）在工具栏单击 按钮，在 mycounter 模块的管脚上添加 6 个 I/O 单元。添加完毕后，双击 I/O 单元修改命令，如图 5-8 所示。

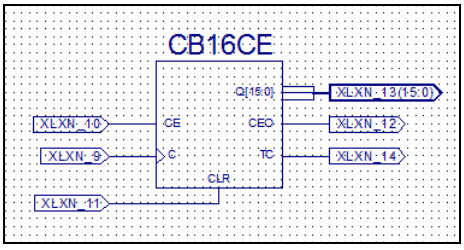

图 5-8　添加 I/O 单元

（5）保存原理图设计，关闭原理图编辑页面，返回工程管理区的 Source 页面。在 mysch 项上右击，选择 Set as Top Module 命令，将其设置为顶层模块。

5.3　Verilog HDL 语言的描述形式

Verilog HDL 可以完成实际电路不同抽象级别的建模，具体而言有 3 种描述形式：如果从电路结构的角度来描述电路模块，称为结构描述形式；如果对线型变量进行操作，就是数据流描述形式；如果只从功能和行为的角度来描述一个实际电路，就成为行为级描述形式。

如前所述，电路具有 5 种不同模型（系统级、算法级、RTL 级、门级和开关级）。系统级、算法级、RTL 级属于行为描述；门级属于结构描述；开关级涉及模拟电路，在数字电路中一般不予考虑。

5.3.1　结构描述形式

Verilog HDL 中定义了 26 个门级关键字，实现了各类简单的门逻辑。结构化描述形式通过门级模块进行描述的方法，将 Verilog HDL 预先定义的基本单元实例嵌入到代码中，通过有机组合形成功能完备的设计实体。在实际工程中，简单的逻辑电路由少数逻辑门和开关组成，通过门元语可以直观地描述其结构，类似于传统的手工设计模式。

Verilog HDL 语言提供了 12 个门级原语，分为多输入门、多输出门以及三态门 3 大类，如表 5-1 所示。

表 5-1　门原语关键字说明列表

门 级 单 元		
多 输 入 门	多 输 出 门	三　态　门
and	buf	bufif0
nand	not	bufif1
or		notif0
nor		notif1
xor		
xnor		

结构描述的每一句话都是模块例化语句，门原语是 Verilog HDL 本身提供的功能模块。其最常用的调用格式为：

```
门类型 <实例名> (输出, 输入 1, 输入 2, ……, 输入 N)
```

例如：

```
nand na01(na_out, a, b, c );
```

表示一个名字为 na01 的与非门，输出为 na_out，输入为 a, b, c。

1．多输入门原语

1）and 门

and 门是二输入的与门。and 门的输入端口是对等的，无位置、优先级等区别，假设其

名称分别为 a、b，则其真值表如表 5-2 所示。

表 5-2　and门原语真值表

and（输出）		a（输入）			
		0	1	X	Z
b（输入）	0	0	0	0	0
	1	0	1	X	X
	X	0	X	X	X
	Z	0	X	X	X

2）nand 门

nand 门是二输入的与非门。nand 门的输入端口是对等的，无位置、优先级等区别，假设其名称分别为 a、b，则其真值表如表 5-3 所示。

表 5-3　nand 门原语真值表

nand（输出）		a（输入）			
		0	1	X	Z
b（输入）	0	1	1	1	1
	1	1	0	X	X
	X	1	X	X	X
	Z	1	X	X	X

3）or 门

or 门是二输入的或门。or 门的输入端口是对等的，无位置、优先级等区别，假设其名称分别为 a、b，则其真值表如表 5-4 所示。

表 5-4　or门原语真值表

or（输出）		a（输入）			
		0	1	X	Z
b（输入）	0	0	1	X	X
	1	1	1	1	1
	X	X	1	X	X
	Z	X	1	X	X

4）nor 门

nor 门是二输入的或非门。nor 门的输入端口是对等的，无位置、优先级等区别，假设其名称分别为 a、b，则其真值表如表 5-5 所示。

表 5-5　nor门原语真值表

nor（输出）		a（输入）			
		0	1	X	Z
b（输入）	0	1	0	X	X
	1	0	0	0	0
	X	X	0	X	X
	Z	X	0	X	X

5）xor 门

xor 门是二输入的异或门。xor 门的输入端口是对等的，无位置、优先级等区别，假设其名称分别为 a、b，则其真值表如表 5-6 所示。

表 5-6　xor门原语真值表

xor（输出）		a（输入）			
		0	**1**	**X**	**Z**
b（输入）	0	0	1	X	X
	1	1	0	X	X
	X	X	X	X	X
	Z	X	X	X	X

6）xnor 门

xnor 门是二输入的异或非门。xnor 门的输入端口是对等的，无位置、优先级等区别，假设其名称分别为 a、b，则其真值表如表 5-7 所示。

表 5-7　xnor门原语真值表

xnor（输出）		a（输入）			
		0	**1**	**X**	**Z**
b（输入）	0	1	0	X	X
	1	0	1	X	X
	X	X	X	X	X
	Z	X	X	X	X

2．多输出门原语

1）buf 门

buf 门是单输入的数据延迟门。buf 门的输入端口是对等的，无位置、优先级等区别，假设其名称分别为 a、b，则其真值表如表 5-8 所示。

表 5-8　buf门原语真值表

a（输入）	b（输出）
0	0
1	1
X	X
Z	Z

2）not 门

not 门是单输入的反相器。not 门的输入端口是对等的，无位置、优先级等区别，假设其名称分别为 a、b，则其真值表如表 5-9 所示。

表 5-9　not门原语真值表

a（输入）	b（输出）
0	1
1	0
X	X
Z	Z

3．三态门原语

1）bufif 0 门（或门）

bufif 0 门是单输入的三态门，控制端低有效。bufif 0 门的输入端口是对等的，无位置、优先级等区别，假设其名称分别为 a、b，则其真值表如表 5-10 所示。

表 5-10　bufif 0 门原语真值表

b（bufif 0 输出）		ctrl（控制端）			
		0	**1**	**X**	**Z**
a（输入）	0	0	Z	L	L
	1	1	Z	H	H
	X	X	Z	X	X
	Z	X	Z	X	X

2）bufif 1 门（或门）

bufif 1 门是单输入的三态门，控制端高有效，bufif 1 门的输入端口是对等的，无位置、优先级等区别，假设其名称分别为 a、b，则其真值表如表 5-11 所列。

表 5-11　bufif 1 门原语真值表

b（bufif 1 输出）		ctrl（控制器）			
		0	**1**	**X**	**Z**
a（输入）	0	Z	0	L	L
	1	Z	1	H	H
	X	Z	X	X	X
	Z	Z	X	X	X

3）notif 0 门（或门）

notif 0 门是单输入的三态反相器，控制端低有效，notif 0 门的输入端口是对等的，无位置、优先级等区别，假设其名称分别为 a、b，则其真值表如表 5-12 所示。

表 5-12　notif 0 门原语真值表

b（notif 0 输出）		ctrl（控制端）			
		0	**1**	**X**	**Z**
a（输入）	0	1	Z	H	H
	1	0	Z	L	L
	X	X	Z	X	X
	Z	X	Z	X	X

4）notif 1 门（或门）

notif 1 门是单输入的三态反相器，控制端高有效。notif 1 门的输入端口是对等的，无位置、优先级等区别，假设其名称分别为 a、b，则其真值表如表 5-13 所示。

表 5-13　notif 1 门原语真值表

b（notif 1 输出）		ctrl（控制端）			
		0	1	X	Z
a（输入）	0	Z	1	H	H
	1	Z	0	L	L
	X	Z	X	X	X
	Z	Z	X	X	X

基于门原语的设计，要求设计者首先将电路功能转化成逻辑组合，再搭建门原语来实现，是数字电路中最底层的设计手段。下面给出一个基于门原语的全加器设计实例。

【例 5-9】 利用 Verilog HDL 实现一个一位全加器。

```
module ADD(A, B, Cin, Sum, Cout);
    input A, B, Cin;
    output Sum, Cout;
    //声明变量
    wire S1, T1, T2, T3;
    //调用两个或非门
    xor X1 (S1, A, B),
    X2 (Sum, S1, Cin);
    //调用 3 个与门
    and A1 (T3, A, B),
    A2 (T2, B, Cin),
    A3 (T1, A, Cin);
    //调用一个或门
    or O1 (Cout, T1, T2, T3);
endmodule
```

在这一实例中，模块包含门的实例语句，也就是包含内置门 xor、and 和 or 的实例语句。图 5-9 为全加器的连接结构示意图。由于未指定顺序，门实例语句可以以任何顺序出现。

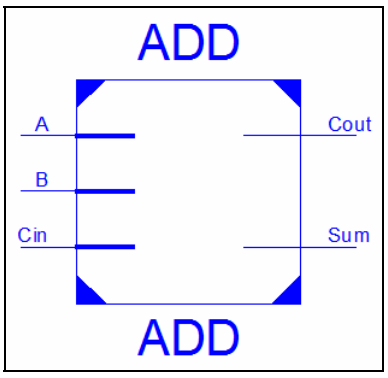

图 5-9　全加器的连接结构示意图

门级描述本质上也是一种结构网表，具备较高的设计性能（资源、速度性能）。读者在实际应用中的使用方法为：先使用门逻辑构成常用的触发器、选择器、加法器等模块，再利用已经设计的模块构成更高一层的模块，依次重复几次，便可以构成一些结构复杂的电路。其缺点是：不易管理，实现难度较大且需要一定的资源积累。

在 20 世纪 90 年代中期以前，可编程逻辑器件的资源很少，规模大都为目前主流器件

的千分之一到百分之一左右，再加上相关的设计都比较简单，因此结构化描述有着广泛的应用。

此后，特别是近十年，由于半导体器件规模和系统设计规模的飞速增长，这一传统且古老的设计方式已被彻底弃用。事实上，通过本书后续内容的学习，没有人还愿意书写或阅读类似于例 5-8 的代码。

5.3.2 行为描述形式

行为型描述主要包括语句/语句块、过程结构、时序控制、流控制等方面，是目前 Verilog HDL 中最重要的描述形式。

1．语句块

语句就是各条 Verilog HDL 代码。语句块指位于 begin…end/fork…join 块定义语句之间的一组行为语句，它将满足某一条件下的多条语句标记出来，类似于 C 语言中“{}”符号中的内容。

语句块可以有独立的名字，名字写在块定义语句的第一个关键字之后，即 begin 或 fork 之后，可以唯一地标识出某一语句块。如果有了块名字，则该语句块被称为一个有名块。在有名块内部可以定义内部寄存器变量，且可以使用 disable 中断语句中断。块名提供了唯一标识寄存器的一种方法，如例 5-9 所示。

【例 5-10】 语句块应用实例。

```
always @ (a or b )
begin : adder1 //adder1 为语句块声明语句
c = a + b;
end
```

以上语句定义了一个名为 adder1 的语句块，用于实现输入数据的相加。语句块按照界定不同分为以下两种。

1）begin…end（串行）

begin…end 块用来组合需要顺序执行的语句，因此被称为串行块。例如下面的语句：

```
reg[7:0] r;
begin //由一系列延迟产生的波形
    r = 8'h35; //语句 1
    r = 8'hE2; //语句 2
    r = 8'h00; //语句 3
    r = 8'hF7; //语句 4
end
```

其执行顺序是，首先执行语句 1，将 8'h35 赋给变量 r；再执行语句 2，将 8'hE2 再次赋给变量 r，覆盖语句 1 所赋的值；依此类推，最后，将 8'hF7 赋给变量 r，形成其最终的值。

串行块的执行特点如下。

- 串行块内的各条语句按它们在块内的语句逐次逐条顺序执行，当前一条执行完之后，才能执行下一条。例如，上例中语句 1～4 就是顺序执行的。
- 块内每一条语句中的延时控制都是相对于前一条语句结束时刻的延时控制。例如，

上例中语句 2 的时延为 2d。

- 在进行仿真时，整个语句块总的执行时间等于所有语句执行时间之和。如上例中语句块中总的执行时间为 4d。
- 在可综合语句中，begin…end 块内的语句在时序逻辑中本质上是并行执行的，和语句的书写顺序无关。读者可以从 EDA 设计的本质去简单理解，可综合 Verilog HDL 语句描述的是硬件电路，数字电路的各个硬件组成部分是并列工作的（PC 机的声卡和显卡就是同时工作的，用户可以同时听到声音并欣赏图像）。

2）fork…join（并行）

fork…join 用来组合需要并行执行的语句，被称为并行块。其应用示例如下。

```
parameter d = 50;
reg[7:0] r1, r2, r3, r4;
fork //由一系列延迟产生的波形
    r1 = ' h35; //语句 1
    r2 = ' hE2; //语句 2
    r3 = ' h00; //语句 3
    r4 = ' hF7; //语句 4
join
```

并行块的执行特点为：

- 语句块内各条语句是各自独立同时开始执行的，各条语句的起始执行时间等于程序流程进入该语句块的时间。如上例中语句 2 并不需要等语句 1 执行完才开始执行，它与语句 1 是同时开始的。
- 块内每一条语句中的延时控制都是相对于程序流程进入该语句块的时间而言的。
- 在进行仿真时，整个语句块总的执行时间等于执行时间最长的那条语句所需要的执行时间。需要说明的是：begin…end 块是可综合语句，其串行执行的特点是从语法结构上讲的。在实际电路中，各条语句之间并不全是串行的，这一点是 Verilog HDL 设计思想的难点之一。fork…join 块是不可综合的，更多地应用于仿真代码中。

2．过程结构

过程结构采用下面 4 种过程模块来实现，具有强的通用型和有效性。

- initial 模块；
- always 模块；
- task（任务）模块；
- function（函数）模块。

一个程序可以有多个 initial 模块、always 模块、task 模块和 function 模块。initial 模块和 always 模块都是并行执行的，区别在于 initial 模块只执行一次，而 always 模块则是不断重复地运行。initial 模块是不可综合的，常用于仿真代码的变量初始化中；always 模块则是可综合的。下面给出 initial 模块和 always 模块的说明。

1）initial 模块

在进行仿真时，一个 initial 模块从模拟 0 时刻开始执行，且在仿真过程中只执行一次，在执行完一次后，该 initial 就被挂起，不再执行。如果仿真中有两个 initial 模块，则同时从 0 时刻开始并行执行。

initial 模块是面向仿真的，是不可综合的，通常被用来描述测试模块的初始化、监视、波形生成等功能。其格式为：

```
initial begin/fork
    块内变量说明
    时序控制 1 行为语句 1;
    ……
    时序控制 n 行为语句 n;
end/join
```

其中，begin…end 块定义语句中的语句是串行执行的，而 fork…join 块语句中的语句定义是并行执行的。当块内只有一条语句且不需要定义局部变量时，可以省略 begin…end/fork…join。

【例 5-11】 下面给出一个 initial 模块的实例。

```
initial begin
//初始化输入向量
    clk = 0;
    ar = 0;
    ai = 0;
    br = 0;
    bi = 0;
    //等待 100 个仿真单位，全局 reset 信号有效
    //其中#为延迟控制语句
    #100;
    ar = 20;
    ai = 10;
    br = 10;
    bi = 10;
end
```

2）always 模块

和 initial 模块不同，always 模块是一直重复执行的，并且可被综合。always 过程块由 always 过程语句和语句块组成，其格式为：

```
always @ (敏感事件列表) begin/fork
    块内变量说明
    时序控制 1 行为语句 1;
    ……
    时序控制 n 行为语句 n;
end/join
```

其中，begin…end/fork…join 的使用方法和 initial 模块中的一样。敏感事件列表是可选项，但在实际工程中却很常用，而且是比较容易出错的地方。敏感事件表的目的就是触发 always 模块的运行，而 initial 后面是不允许有敏感事件表的。

敏感事件表由一个或多个事件表达式构成，事件表达式就是模块启动的条件。当存在多个事件表达式时，要使用关键词 or 将多个触发条件结合起来。Verilog HDL 的语法规定：对于这些表达式所代表的多个触发条件，只要有一个成立，就可以启动块内语句的执行。例如，在语句

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

```
end
```

中，always 过程块的多个事件表达式所代表的触发条件是：只要 a、b、c 信号的电平有任意一个发生变化，begin…end 语句就会被触发。

always 模块主要是对硬件功能的行为进行描述，可以实现锁存器和触发器等基本数字处理单元，也可以用来实现各类大规模设计。

【例 5-12】 always 模块的应用示例。

```
module and3(f, a, b, c);
    input a, b, c;
    output f;
    reg f;
    always @(a or b )begin
    f = a & b & c;
    end
endmodule
```

3．时序控制

Verilog HDL 提供了两种显式时序控制的类型，一是延迟控制，通过表达式定义开始遇到这一语句和真正执行这一语句之间的延迟时间；二是事件控制，通过表达式完成控制，只有当某一事件发生时才允许语句继续向下执行。

一般来讲，延时控制语句是不可综合的，常用于仿真，而事件控制是可综合的，通过 always 语句来实现，分为电平触发和信号跳变沿触发两类。

4．流控制

流控制描述一般采用 assign 连续赋值语句实现，主要用于完成简单的组合逻辑功能。连续赋值语句右边所有的变量受持续监控，只要这些变量有一个发生变化，整个表达式将被重新赋值给左端。其语法格式如下：

```
assign L_s = R_s;
```

【例 5-13】 一个利用数据流描述的移位器。

```
module mlshift2(a, b);
    input a;
    output b;
    //数据流描述语句，使用 assign 关键字
    assign b = a<<2;
endmodule
```

在上述模块中，只要 a 的值发生变化，b 就会被重新赋值，所赋值为 a 左移两位后的值。

5.3.3　混合设计模式

在 Verilog HDL 模块中，结构描述、行为描述可以自由混合。也就是说，模块描述中可以包括实例化的门、模块实例化语句、连续赋值语句以及行为描述语句，它们之间可以相互包含。always 语句和 initial 语句（切记只有寄存器类型数据才可以在这两个模块中赋

值）可用于驱动门和开关，而来自于门或连续赋值语句（只能驱动线网型）的输出能够反过来用于触发 always 语句和 initial 语句。

【例 5-14】 与非门混合设计。

```
module hunhe_demo(
    A, B, C
    );
        input A, B;
        output C;
        //定义中间变量
        wire T;
        //调用结构化与门
        and A1 (T, A, B);
        //通过数据流形式对与门输出求反，得到最终的与非门结果
        assign C = ~T;
endmodule
```

5.4 思考与练习

1．概念题

（1）一个完整的 Verilog HDL 程序包含哪些部分，其中哪些部分是必须的？
（2）解释 Verilog HDL 模块（module）的概念，指明其与传统软件函数的区别。
（3）在 Verilog HDL 模块中，端口可以分为哪几类？各有什么特点？
（4）如何通过 Verilog HDL 语言完成模块例化，有哪几种方法？
（5）有哪几种方法可以实现 Verilog HDL 子模块的参数化配置？
（6）在 ISE 中，如何生成一个模块的图形化表示符号？
（7）Verilog HDL 有哪几种描述形式？

2．操作题

（1）通过“#”字符实现一个模值可调的加 1 计数器，计数器最大值为 12。
（2）通过 defparam 实现一个模值可调的加 1 计数器，计数器的最大值为 6。