verilog 语言代码设计规范

目录

一、代码书写规范

1.1 模块说明书写规范

在开始子模块设计时，必须对子模块的基本信息给予说明。说明的位置一般在设计的开头，使用注释的形式用（/* */）说明该设计的作者、编写日期、版本号、在系统中的层次位置、基本功能描述等等。其形式如下所示：

```
/*************************************************************
*author   :abc                                                *
*date     :2xxx-xx-xx                                          *
*version  :1.0                                                *
*describe  :                                                  *
*************************************************************/
```

说明的内容要简洁清晰。使用/* */对将说明部分括起来是为了与普通注释相区别。

1.1 模块注释书写规范

注释对项目团队关于设计的交流至关重要，好的设计总是会在恰当的地方对语句或变量予以说明，没有注释的设计不是真正的工业级设计，通篇的注释同没有注释一样糟糕，会将代码淹没在无用的注释之中。这一节给出书写注释的规范。

如果设计中出现了一个新的变量，那么必须对这个新变量给予注释，对变量的注释应该放在变量的定义之前，注释应该说明变量的物理意义或作用。其形式如下：

```
……
//input port
//clear reset singal of system
//Ac control singal of accumulation
  input clear,Ac;
……
```

如果设计中的某块结构属于作者的创新或设计中很关键的部分，作者应该对这种结构的物理含义予以简要说明。注释在语句或结构的前一行开始写如：

```
……
//wallace tree multiplication
……
```

1.3 变量名称书写规范

verilog 语言规定了各种类型的标志符的格式，作为规范我们对用户自定义的各种变量的命名方法及书写格式加以约束。变量一般指模块（或设计）名、端口名、连线名、参照名、单元名以及内部寄存器名。首先变量名必须能表达实际的物理意义，如果需要几个单词来表示，那么单词之间用一个“_”分隔。变量名不宜过长，一般不要超过 16 个字符否则书写的效率会下降，因此变量名应该尽量使用单词的缩略写法，完整的含义应在注释中给予说明。我们规定常量参数一律使用大写字母表示，变量的名称一律用小写字母 a~z、数字 0~9 或下划线_表示，变量首字符一律用字母。模块名（或设计名）应该与文件名一致，一个文件只应包含一个模块。它是模块功能的缩略表达。端口名应与该端口实际的物理意义相一致。连线是对内部单元（实例）引脚间进行连接的物理线或是对组合逻辑输出端口、组合逻辑单元输出端进行赋值运算的输入线。连线名应该有确定的连接对象或是有确定的信号物理意义，所以针对单元引脚连接的连线它的名称应该表明所连接单元的名称，如 timing_alu 表示时序发生器发出的控制信号连接 ALU 部件的控制端。针对为输出端做赋值运算的输入连线它的名字应该表达相应的物理意义，绝不要使用通用名如：a,b,c 这样的名字。这类连线适当的名称如：add_a，add_b。参照名就是一个单元或实例参考的库标准单元或原始设计名，所以它的名字与库单元或模块名相同。单元与实例在 synopsys 的 DC 工具中是不加区分的，这里也等同看待，它的名字可以用参照名为头后加数字予以标识。如 alu1、acc1 等。如果设计的内部有中间级寄存器，那么寄存器以实际的物理意义进行命名，比如在乘法器中为分割关键路径引入的中间级寄存器可以命名为

pipeline1_out 等。

1.4 代码结构书写规范

好的代码结构可以清晰的看出设计的层次关系，进而使结构与设计者所要表达的逻辑意图一致，方便纠错和交流。代码最基本的结构有平行结构和层次结构，他们反映了代码的隶属关系，我们规定注释语句与语句块是一个层次的，这意味着对模块的注释行（以//开头）必须在一行的顶头开始写。其他语句的层次低于模块定义和注释，那么其他语句至少要向后缩退四个空格。语句块中的语句低于语句块的定义，例如：

```
always @(posedge clk)
begin
    out <= in;
end
//end always
```

语句块中的语句块和其他语句是同级的。语句块结束应该有注释说明结束的语句块的名称

例如：

```
always @(posedge clk1 negedge clear)
begin
    if(!clear)
    begin
        out <= 6'b000000;
    end
    else
    begin
        case(addr)
            `SFR_A:
            begin
                out <= in;
            end
……
        endcase
    end
end
```

二、使用 verilog 语言的语法范围

2.1 设计 RTL 代码的语法范围

verilog 语言是一种通用的 HDL（硬件描述语言），它的语法范围包括了用户各种设计层级的需求，虽然语言本身对这种层级并没有细致的区分，但因为设计者编写的代码最终要使用具体的综合工具综合成实际电路，所以语言层级的划分不可避免的具有针对某种工具的特点，我们这里的划分依据是根据 DesignCompile 综合工具来定的。在论述的过程中我们常使用左值和右值的称谓，一般我们将赋值运算符左侧的变量称作左值，右侧的变量称作右值，因此左值具有位置属性而右值具有数值属性。右值的任何地方禁止出现“x”，左值在任何情况下不可以为常量。在有些环境中也将左值称做写数据，右值称做读数据。在一个语句块中不可对同一个左值赋两次值，因为这样会引起数据冲突（三态门除外）。verilog 语言可以描述四个设计层级的语法范畴，依次是：系统行为级、模块寄存器传输级（RTL）、电路门级、晶体管开关级。层级之间的转换我们称之为综合。目前业界的流行的做法是前端设计将 RTL 级综合为门级网表。行为级到门级的直接综合还不成熟，另外，数字电路中的晶体管我们一般当作开关来对待，因此对它们具体的器件特性描述，并不是针对数字电路设计的 verilog 语言所能及的。

系统行为级描述一般作为系统设计的辅助手段或建立系统仿真环境的语言实现手段，在系统设计时，设计人员一般使用行为级描述来建立系统数据流的直观表述，以助选定设计方案，划分系统层次、确定模块接口等。这个级别是设计的最高抽象层，它不考虑设计的具体实现，只是确定实现的可行性，和估计实现的规模，因此系统设计时系统行为级描述不限制语言的种类，可以用 verilog 也可用 System C 甚至 C 语言，它们只是帮助系统设计人员最终得到系统各个模块的端口列表和层级子模块明确的功能定义。有些项目使用层级关系的原理图来说明系统结构，子模块使用黑盒来代替。一些小项目这个过程可以省略直接由编码人员来实现。行为级描述也可以建立系统的仿真环境，它也是实际工作中行为级最重要的作用。这时一般将被仿真的模块作为实例，仿真环境为实例提供各种输入向量，设计人员观察输出结果分析被测实例的功能正确与否，如有错误可以定

位错误以便修正。

HDL 的寄存器传输级（RTL）一般用于对电路子模块的描述，它一般由四种基本结构组成：寄存器、计数器、选择器、算术逻辑运算单元。其中寄存器对应 always 语句块，选择器对应 if…else…语句块，算术运算符加（+）、减（-）可以作为 RTL 代码直接综合，乘（*）、除（/）如果没有特殊的性能要求也可以受限使用，逻辑运算符与、或、非是 RTL 代码。计数器一般作为各种状态标志的产生器。我们规定 RTL 的抽象级别是逻辑抽象，所以它应该反应的是各种器件的逻辑行为和逻辑结构，而不是器件的物理组成（门级连接），另外它对应的是具体的物理实体，而不是理想的逻辑关系（没有实物对应的逻辑抽象和理想的信号时空顺序）。在逻辑抽象级我们一般将组合电路看作连线之间的算术逻辑赋值运算，将时序电路看作由寄存器存储组合逻辑产生结果的结构。所以 RTL 级的四种基本结构的组合变化足以构建各种器件千变万化的逻辑结构。我们规定对 wire 型变量的赋值使用连续赋值语句（assign），并且一律使用阻塞型赋值（=），组合逻辑中 always 块中的 reg 型变量一律使用阻塞型赋值，在时序逻辑中使用非阻塞型赋值（<=）。RTL 代码中要求不使用循环语句，所有涉及循环赋值的部分一律将循环打散采用显式赋值。代码中不要出现函数、任务的相关语句块，所有需要出现的地方使用 module 编写模块,再调用实例。没有实物对应的 initial 语句块以及具有理想信号时空顺序的语句，如显式指定延迟语句不能在 RTL 代码中出现。如果有实例调用那么实例的引脚连接采用端口对应禁止使用位置对应。因为综合工具没有与“x”对应的状态，所以设计中任何地方的值禁止出现“x”，如果对总线的某几位不加考虑，那么位选择时只出现要考虑的部分。对于重要的有含义的常量不要直接在设计中使用具体值，而应该用参数定义使常量参数化。

为了便于查找我们将综合工具不能综合的关键字列举如下：

- initial
- 循环语句：
  - repeat
  - forever
  - while
  - for 的非结构用法

- 数据类型
  - event
  - real
  - timeUDPs
- fork…join 块
- wait
- 过程连续赋值语句
  - assign 和 deassign
  - force 和 release
- 部分操作符
  - = = =
  - ！= =

以上的关键字，综合工具不支持，它们也不是我们设计 RTL 代码所必须的，我们规定设计的 RTL 代码一定不能含有上面列出的条目。

2.2 设计仿真代码的语法范围

在前面已经简单谈及了仿真代码的编写问题，阐明了仿真代码实际上是 HDL 行为级描述的一种应用。它是一种不考虑实际硬件的实现只为产生仿真向量的理想的逻辑抽象，它所产生的信号可以具有理想的时空顺序。它的逻辑范畴不受具体物理器件实现的限制。我们规定仿真代码中有关信号时间顺序的安排应放在 initial 语句块中，被测模块的实例引脚连接使用端口对应风格。仿真代码可以使用循环结构、可以使用函数、任务、用户定义原语。仿真代码产生的仿真向量要具有典型型，向量的覆盖面要大，对于逻辑边界一定要有对应的仿真向量，对于可能的数据路径也要找出相应的输入向量遍历它。仿真代码包括两个部分，分别是，虚拟器件建模或仿真环境建立。虚拟器件一般指 ROM、RAM 存储器件或没有用 RTL 代码实现的行为级抽象器件的建模。仿真环境一般包括各器件的实例调用、时钟的模型、和各信号的时空安排。为了对信号进行特定的时空安排我们可以在 verilog 语法范围内使用对信号的数值和出场顺序进行任意逻辑的处理，并且我们可以对信号的状态进行记录、观测、调整，仿真工具一般为我们内

置了不少系统调用，常用的调用我们列举如下：

1)$time //找到当前的仿真时间

2)$display, $monitor //显示和监视信号值的变化

3)$stop //暂停仿真

4)$finish //结束仿真

5)$readmemb //从外部文件向内部寄存器读数据

6)$dumpfile //打开记录数据变化的数据文件

7)$dumpvars; //选择需要记录的变量

8)$dumpflush; //把记录在数据文件中的资料转送到硬盘保存

系统调用的具体用法，和其他系统调用，请参见具体工具的使用手册。关于时钟的建模部分请参见 4.5 节，我们一般将信号的时空顺序和信号的初始化操作放在 initial 语句块中表达。

三、使用 verilog 语言的结构范围

3.1 系统设计文件的形式与使用方法

系统设计文件是编码设计人员的设计依据，一般由系统设计人员确定。它的内容至少应该包括子模块的功能定义，模块的端口定义、模块的时序要求、和项目的进度安排，无论项目开发采用增量式开发模式还是采用流水线开发模式，对于编码人员来说都是不需要关心的，因为增量式最小原形法与普通的流水线模式最大的不同是每次增量需要设计人员考虑的设计规模是不同的，一般增量式开发过程每次增量的设计目标都是在可以把握的范围之内的。需要设计的子模块功能在一次增量的过程中是不变的，当然不同的增量级别和迭代级别要考虑的功能和性能要求又是不同的。系统设计文件中对子模块的描述可以用层级式的原理图方式也可以用 verilog 语言配合文字给予说明的文本文件，编码设计人员拿到系统设计文件后，应该可以明确自己工作的职责，如明确需要设计的是什么功能模块、明确模块的输入输出端口是什么，模块与外部模块的耦合关系、是否本模块与其他模块有编写的顺序问题、模块需不需要再有层级关系以及模块必须交付的时间等。编码人员在充分理解了以上这些问题之后再开始具体的编码。

3.2 模块结构划分的标准

模块结构的划分不仅在系统级而且在编码级都必须仔细考虑，一般编码级的划分是对子模块的再次细分。因为模块的结构直接影响了综合后电路的质量，所以划分结构必然要考虑综合工具对各种结构的综合效率。verilog 语言可以用以下结构创建层级。一般 module 语句定义了新层，参照的实例在模块中创建了一个次层级，实例间具有层级的边界。算术运算符（+，-，…）暗指的电路能创建一个次层级。always 语句不创建层级关系。划分必须按一定的准则去做，我们将这些准则列举如下：

- 尽量不要将组合逻辑划分成一个独立的模块。因为 DC 必须保留端口的定义。逻辑优化不能穿越设计的边界。因此综合优化的效率可能会很低，电路的质量会很差。除非电路是纯组合逻辑的特殊情况，否则我们一般以寄存器为输出边界。
- 消除不必要的层级，避免粘连逻辑。我们将实例与实例通过门单元进行连接的逻辑叫做粘连逻辑。因为 DC 必须保留端口的定义，所以 DC 综合时不能将粘连逻辑吸收到任何模块内部进行优化，好的划分应该尽量消除不必要的层级，使优化充分发挥作用。
- 尽量平衡各逻辑块的尺寸和模块内部的数据传输路径长度。划分的太细不利于优化，但也不是划分的块越大越好，如果逻辑块太大，综合运行的时间可能非常长，模块内部的关键路径如果出现时序异常将很难修正，模块的维护和团队的交流将出现巨大障碍。
- 设计至少应该有三个层级。三个层级依次为：顶层、中间级、内核级。一般顶层只包含实例的调用，中间级是对一些可复用子模块的调用和相关处理，内核级一般是最底层，他包括设计的核心处理部件、项目可复用的模块等。
- 如果设计中有异步电路一般应该将异步电路单独设计，并且一个时钟对应一个模块，模块与模块之间的握手连接在异步电路的顶层处理。

按照以上准则得到的划分结构可能不止一个，设计人员必须在不同的划分中进行选择得到最佳的方案。

3.3 组合逻辑的代码风格

这一节和下一节将对具体器件用 verilog 语言实现的代码风格进行规范。所谓代码风格就是具体硬件使用 verilog 语言实现的算法。这里需要强调的是设计人员必须用硬件的思维来编写代码。

组合逻辑是可以认为是算术逻辑的赋值运算。一般情况下组合逻辑在连续赋值语句中实现，因此它的赋值对象不能是 reg 类型的变量，另外不能使用非阻塞型的赋值符号。由于与非、或非的物理实现要比与、或简单，因此考虑到速度与最终芯片的面积我们优先选择与非、或非的逻辑。如果必须用语句块实现组合逻辑电路那么，组合逻辑也可以放到 always 的语句块中，在 always 语句块中的左值必须是 reg 类型，组合逻辑的 always 语句块一般使用阻塞型赋值，如果一定要在电路中体现数据传输路径，那么在使用阻塞型赋值时必须考虑中间变量（既充当左值又充当右值的变量）的传输路径问题。在非阻塞赋值中，因为读变量的时间发生在变量值改变之前，所以每出现一个中间变量则传输路径会多一级，中间变量会被保留，因此中间变量也要在 always 事件列表中指明。但是使用阻塞型赋值时，读变量发生在变量值改变之后，所以中间变量常常是被前级变量替代了或与出现中间变量的位置直接与前级变量相连传输路径不再保留中间变量，为防止电路出现仿真结果在综合前后不一致的现象，在 always 语句块中使用阻塞型赋值时右值部分的变量和条件控制信号在 always 事件列表中要显式指明，但是中间变量不必在列表中出现。在任何情况下都不要在一个语句块中对同一个左值多次赋值。因为这样会因为竞争冒险而产生数据冲突。为了将代码的编写规范化我们将选择器、寄存器、计数器的结构分别给予规范。

选择器：

在 always 语句块中如果出现选择结构，那么它对应的是一个多路选择器，当我们使用 if…else 语句实现这种结构时，一般情况下我们推荐 if 和 else 要匹配。以下三种结构她们在逻辑上是等价的我们要求使用第一种。

```
always @(a or b or sel)
begin
    if (sel)
    begin
        out = a;
```

```
    end
    else
    begin
        out = b;
    end
end


always @(a or b or sel)
begin
    out = b;
    if (sel)
    begin
        out = a;
    end
end

always @(a or b or sel)
begin
    out =  (sel) ? a : b;
end
```

if 语句本身会引入电路的优先级结构，不同的编写风格会出现不同的优先级结果，我们分别给予解释。

```
always @(sel_a or sel_b or sel_c or
    a or b or c or d)
begin
    out = d;
    if (sel_a)
    begin
        out = a;
    end
    if (sel_b)
    begin
        out = b;
    end
    if (sel_c)
    begin
        out = c;
    end
end
```

这种结构因为逻辑执行的先后问题，结果的产生发生在所有if语句块判断完成后，因此优先级是从下往上的。既最后完成判断的语句块它的路径离输出端是

最近的优先级也最高。

```
always @(sel_a or sel_b or sel_c or a or b or c or d)
begin
    if (sel_a)
    begin
        out = a;
    end
    else if (sel_b)
    begin
        out = b;
    end
    else if (sel_c)
    begin
        out = c;
    end
    else
    begin
        out = d;
    end
end
```

这种结构只要有一个条件发生匹配if 语句立即完成，跳至整个if结构的出口，因此它的优先级是从上往下的。我们要求使用第二种优先结构的写法。

当遇到不需要优先级的设计时我们一般选择case结构，这是一种在语法逻辑上没有优先级的结构，在case选择分支中的所有结构块他们在地位上是平行的。其结构如下所示。

```
always @(sel or a or b or c or d)
begin
    case (sel)
        2'b00 : out = a;
        2'b01 : out = b;
        2'b10 : out = c;
        2'b11 : out = d;
    endcase
end
```

使用case语句应该注意以下几个问题。首先，case语句的状态表达式不同于if语句的布尔表达式，case语句状态表达式中的包含的各种状态在它的状态项中都应该有明确的对应，如果状态项中的状态少于状态表达式包含的状态，那么综合后的电路可能会引入锁存，另一方面如果状态表达式中包含的状态少于状态项中的状态那么意味着状态表达式的一个值会开启多个状态项中的语句块这种

情况叫状态项重叠。发生状态项重叠的case 语句不再具有无优先级的特性，这使多个开启的状态项语句块依据编写的顺序具有从上而下的优先级，具有和if…else…语句一样的效用。其结构如下所示：

```
always @(a or b or c or d)
begin
    case (1'b1)
        sel_a : out = a;
        sel_b : out = b;
        sel_c : out = c;
        default : out = d;
    endcase
end
```

它和if…else… 结构是等价的。

```
if (sel_a)
begin
    out = a;
end
else if (sel_b)
begin
    out = b;
end
else if (sel_c)
begin
    out = c;
end
else
begin
    out = d;
end
```

因此为了使case 语句无优先级结构，并且不出现锁存，我们必须使状态项中的状态与状态表达式所能包含的状态完全一致。其中最后一个状态或几个状态可以不显式出现而在default分支中表达。具有这种结构的case语句我们称做全case语句，我们要求在描述没有优先级结构的多路选择器时使用全case语句。一般情况下我们不使用casex和casez这样的语句，禁止出现“?”、“x”这种不确定状态的表达，因为综合工具没有与不确定状态相对应的值。因此我们要求语句中的状态一定要明确只能为“0”“1”“z”（“z”只在三态门中出现）。

循环结构虽然具有简化编写工作量的优势，但是当使用阻塞赋值或非阻塞赋

值时会增加其逻辑传输级的复杂度，并且综合工具在综合以前会将循环部分打散再执行综合过程这样综合效率会降低，因此我们要求RTL文件不允许使用循环结构，其循环部分一律手工打散逐项输入。但是对行为级的描述我们并不加这种限制。需要说明的是，对于仿真代码中的循环体一定要注意循环的边界必须是可以算出的具体值，循环不能使用边沿触发的事件列表。

函数、任务结构也是我们在RTL代码中禁止使用的，因为综合时它只是将相关的结构放在调用的地方，把函数的输出作为右值进行赋值。这并不引入物理的层级边界，与逻辑上的函数是有层级边界的相矛盾，因此为了逻辑与物理的一致性我们取消对他们的使用。

当设计中需要双向总线，选择输出或异步设计的时候设计人员往往需要借助三态门来实现，三态门具有“0”、“1”、“z”三种状态，因为三态门中“z”状态与其他状态逻辑运算后结果不变，因此，三态门是电路描述中唯一可以在一个语法层次为同一个左值多次赋值的结构，综合工具不会检查这种情况下的数据冲突问题。为了不引起综合前后仿真结果的不同我们规定，三态门一律使用如下的连续赋值语句：

```
assign out = (sel_a) ? a : 1'bz;
```

并且遇到为同一个左值多次赋值的描述时使用如下结构：

```
assign out = (sel_a) ? a : 1'bz;
assign out = (!sel_b) ? b : 1'bz;
```

使用这种结构设计人员一定要小心。

设计中双向端口往往是不能避免的，双向端口具有既能输入又能输出的特点，要让双向端口不会发生数据冲突是设计人员的责任。常用的双向端口有两种基本的结构，一种是输入与输出是互斥的，另一种是输入常通，输出选通。这两种结构对应不同的描述，分别如下所示：

```
module bidi_module (d_port,inc,out_c,received,send);
    inout d_port;
    input inc,outc,send;
    output received;
    assign d_port = (outc)? send : 1'bz;
    assign received = (inc)? d_port : 1'bz;
```

```
endmodule;
module bidi_module (d_port,outc,received,send);
     inout d_port;
     input outc,send;
     output data_received;
     assign d_port = (outc) ? send : 1'bz;
     assign received = d_port;
endmodule;
```

选用结构时一定要注意描述上的差异，第二种结构的第二个连续赋值语句没三态门意味着d_port双向端口输入方向是常通的。三态门与双向端口的表达可以认为也是选择器的一种实际形式。

算术运算、关系运算与逻辑运算：

算术运算、关系运算与逻辑运算，包括加、减、乘、移位、大于、小于、等于、大于等于、小于等于、与、或、非、异或等以及它们的组合。这里需要说明的是算术运算的综合得宜于DC内部DesignWare库的支持，从IP的角度来看，算术运算属于IP复用。一个加号（+）或一个乘号即对应相应位数的一个加法器或乘法器实例调用。目前的综合工具不支持除法、求模以及幂运算，当然它们可以用其他手段来实现。如果系统要求高性能的加法器、乘法器我们要求手工实现，或选用高性能IP而不是直接调用DesignWare库中的IP。

因为往往多位的算术运算电路所产生的实际电路门数较多，如果设计时对电路的效率考虑不周，则会带入很多冗余结构，并使电路的传输路径非常长。我们要求设计人员使用加乘等规模较大的算术运算结构时，必须满足综合工具资源共享的要求。综合工具对代码优化后所得的资源共享结构并不能由编码设计人员自己决定，因此编码设计人员只需要将设计的代码满足综合工具优化共享的要求就可以了不需要显式在代码中固定结构。一般综合工具对满足以下三点要求的代码进行资源优化：第一，需要资源共享优化的代码必须在同一个module中。第二，需要资源共享优化的代码必须在同一个always块中，第三，需要资源共享优化的代码必须在一个if…else…或case语法结构（注意不是语句块）中。综合工具不会跨越语法结构进行资源优化。我们以加法运算举例如下：

```
module share(out,sel,a,b,c,d);
```

```
    input sel,a,b,c,d;
    output [1:0] out;
    reg [1:0]out
    always @(sel,a,b,c,d)
    begin
        if (sel)
        begin
            out = a + b;
        end
        else
        begin
            out = c + d;
        end
    end
endmodule
```

使用这种结构设计的选择加法运算电路，综合后并不会产生两个加法器，而是先通过一个选择器，选出需要加的信号，再通过一个加法器对信号完成加运算。人为的编写资源共享代码，并不会产生更好的效果。比如我们写如下代码：

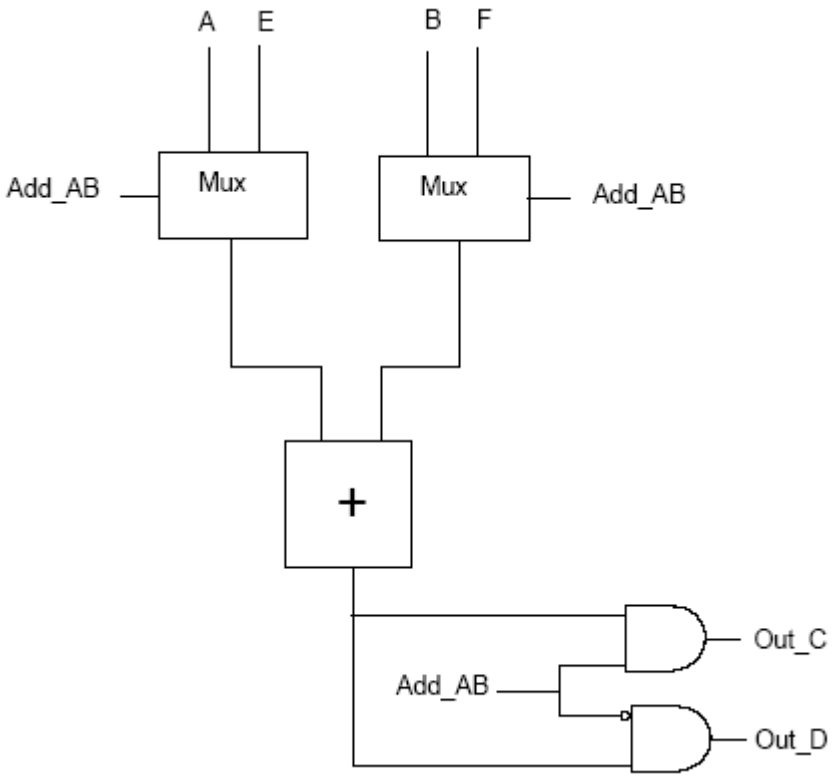
```
module share(out,sel,a,b,c,d);
    input sel,a,b,c,d;
    output [1:0] out;
    reg [1:0]out
    always @(sel,a,b,c,d)
    begin
        if (sel)
        begin
            mux_1 = a;
            mux_2 = b;
        end
        else
        begin
            mux_1 = c;
            mux_2 = d;
        end
        out = mux_1 + mux_2;
    end
endmodule
```

它的综合结果并不比上面的代码效率高，是否资源共享任然需要检查是否满足共享优化的条件。我们再看一个例子：

```
always @( add_ab or a or b or e or f)
begin
```

```
    out_c = 8'b0;
    out_d = 8'b0;
    if (add_ab)
    begin
        out_c = a + b;
    end
    else
    begin
        out_d = e + f;
    end
end
```

这个例子中两个加运算符的左值和右值均不相同，但它们在一个语法结构中，因此综合工具可以对他们进行资源共享。其电路结构如下所示：



另外，如果我们需要连加运算，我们必须考虑传输路径的长度，特别需要说明的是括号对传输路径的组织有很大影响。我们看看下面的语句：

```
data_out = data_a + data_b + data_c + data_d;
```

这条语句会形成三个串联的加法器结构。传输路径延迟无疑会是三个加法器延迟之和。但是通过人为组合，传输路径会改变，如下面的语句。

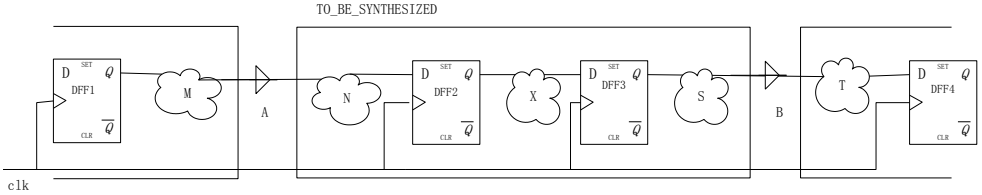
data_out = (data_a + data_b) + (data_c + data_d)；我们通过给综合工具显式指定数据传输方向，使得到的电路结构只包含两级加法器传输延迟。我们要求设计人员在写连加、连乘等会引入长传输路径的结构时必须用括号指定数据的传输方向。

3.4 时序逻辑的代码风格

上一节阐述了组合逻辑的代码风格，我们以此为基础进一步阐述时序逻辑的代码风格，我们知道在逻辑描述级别时序逻辑就是将组合逻辑的结果用寄存器再暂存的结构，这种结构与组合逻辑结构具有本质的不同，因为它可以将各个阶段的处理状态保存下来，对这些状态再次处理，而这样的工作对组合逻辑的电路来说是无法做到的。再处理这些中间状态需要各状态的数据都准备就绪，因此有了时序的概念。毫无疑问时序逻辑的核心器件时序器件，在一个时钟内对信号的处理由组合逻辑完成，所以时序逻辑的代码风格所规范的就是时序器件的产生算法。我们规定时序逻辑的always块一律使用非阻塞型赋值，很明显寄存器前的组合逻辑在always块作为是右值被表示，语句块中的右值一般是表达式或对其他变量的读数据操作。

寄存器：

我们先来阐述寄存器的代码风格。直观上我们可以把寄存器看作是触发器的阵列，它的基本单元是触发器，触发器是边沿触发的时序器件，我们规定触发器的时钟端采用上升沿触发，复位信号、置位信号采用下降沿触发。触发器不仅具有存储能力而且具有一定的逻辑运算能力，为了逻辑上的叙述方便我们将它们的逻辑运算能力划归入组合逻辑部分。构成如下结构：



图中我们看到触发器与触发器之间的部分都是组合逻辑。为了得到以上结构的实现我们要求寄存器一律以D触发器实现。触发器必须具有复位端（clear）、

时钟端（clk）。电路必须异步清零，并且复位信号的优先级高于时钟端，触发器的时钟信号不容许与其他普通信号发生逻辑运算。如果必要触发器还可以具有置位端（set），此时要求置位端的优先级高于复位端。这两种结构如下所示：

```
always @(posedge clk or negedge clear)
begin
    if (!clear)
    begin
        data_out <= 1'b0;
    end
    else
    begin
        data_out <= data_in;
    end
end

always @(posedge clk or negedge clear or negedge set)
begin
    if (!set)
    begin
        data_out <= 1'b1;
    end
    else if (!clear)
    begin
        data_out <= 1'b0;
    end
    else
    begin
        data_out <= data_in;
    end
end
```

电路在任何时候都禁止置位和复位信号同时有效。我们要求，对于时序电路的always模块它的事件敏感表只能包含边沿事件，不能将边沿事件和非边沿事件在一个事件敏感表中表示，并且事件敏感表中只能包含时钟事件（或可以当作时钟事件的控制事件）、复位和置位事件。

在低功耗设计中，设计者要尽量降低所有可能节省的功耗，在时序电路中时钟信号因为高稳定和高频率所消耗的功耗是很大的，所以有必要把不工作的局部电路的时钟关断，这就出现了门控时钟。对于寄存器来说门控时钟与普通时钟

并没有差别，所不一样的是，普通时钟是不允许时钟参与逻辑运算的，而门控时钟在一定的条件下可以与特殊的信号参与逻辑运算。它的描述如下所示：

```
assign gated_clk = clk & enable;
always @(posedge gated_clk or negedge clear)
begin
    if (!clear)
    begin
        out <= 8'b0;
    end
    else
    begin
        out <= out + 8'b1;
    end
end
```

门控时钟的时钟端命名必须予以标明（gated_clk），如果时序电路使用门控时钟那么对可测性设计是不利的，因为，只要门控时钟关断用户是没有办法控制它的，另外门控时钟也给静态时序分析带来了麻烦，因为STA需要用时钟来分组，如果时钟信号在某处与门连接那么它的传输路径将被打断，这时我们就得用其他的办法来实现。具体做法请参见《时序分析规范》，考虑到这些利与弊，我们要求除非系统方案选用低功耗设计，否则禁止使用门控时钟。如果在使用门控时钟的同时又要求可测性设计这时我们使用以下结构来实现：

```
module gated_clock (clock,clear,
                            test_mode, // Test Mode Select
                            out);
    assign gated_clock = clock & (Enable | Test_Mode);
    always @(posedge gated_clock or negedge clear)
    begin
        if (!clear)
        begin
            out <= 8'b0;
        end
        else
        begin
            out <= out + 8'b1;
        end
    end
endmodule
```

在这种情况下如果用户选用测试模式那么门控时钟的开关门将失去作用。具有可测性功能的门控时钟和门控时钟一样，我们都限制他们的使用。

逻辑电路的所有状态都应该在设计人员的控制之下或预料之中，但是不熟练的设计人员往往对电路的所有状态考虑不周使逻辑描述不能确定电路的所有状态，这对整个电路是很危险的，可能发生难以预料的问题，锁存就是这种情况下的罪魁祸首。其实在早期的有些电路中锁存是需要的，它具有电平触发的存储功能，电路结构简单占用面积小，但是现今的设计单元面积并不是最主要的考虑因素，电路的可靠性、可控性、易维护性甚至比面积更加重要，基于此我们要求电路避免锁存器，但是初学者往往不能分辨怎样的描述会产生锁存。我们将可能产生锁存的结构列举如下。

```
always @(enable or data)
begin
    if (enable)
    begin
        q = data;
    end
end
```

这个的结构会得到一个锁存，因为事件敏感表中的事件是非边沿触发事件，所以综合工具不会为语句块中的每个左值匹配一个寄存器，同时选择结构的分支不全意味着电路结构并没有给左值指定所有的情况下左值应该有的状态，这时DC会认为在其他情况下没有指定对应状态的左值会保持前一个状态不变，这具有存储能力，又是电平触发，因此形成了锁存。

```
always @(enable or a or b)
begin
    if (enable)
    begin
        out_1 = a;
    end
    else
    begin
        out_2 = b;
    end
end
```

这个结构会得到两个锁存。因为左值没有在各种情况下具有指定的值，如果要避免琐存那么out1、out2应该在if语句的每个分支中都有显式指定的状态。

注意下面的结构不会产生锁存。

```
always @(posedge clock)
begin
    if (enable)
    begin
        out = a;
    end
```

```
end
```

因为上面的结构使用的是边沿触发所以只会在触发器的数据端连接一个选择器。

```
input [3:0] in;
always @(in)
begin
    case (in)
        0 : out_1 = 1'b1;
        1,3 : out_2 = 1'b1;
        2,4,5,6,7 : out_3 = 1'b1;
        default : out_4 = 1'b1;
    endcase
end
```

上面的结构会产生锁存，因为状态项分支中的左值不全。那么不全的左值会被综合成锁存。要避免锁存可以使用下面的结构。

```
input [3:0] in;
always @(in)
begin
    out_1 = 1'b0;
    out_2 = 1'b0;
    out_3 = 1'b0;
    out_4 = 1'b0;
    case (Data_In)
        0 : out_1 = 1'b1;
        1,3 : out_2 = 1'b1;
        2,4,5,6,7 : out_3 = 1'b1;
        default : out_4 = 1'b1;
    endcase
end
```

这种结构会避免锁存。下面的结构同样可以避免。

```
input [3:0] in;
always @(in)
begin
    case (in)
        0 : begin
                out_1 = 1'b1;
                out_2 = 1'b0;
                out_3 = 1'b0;
                out_4 = 1'b0;
        end
        1,3 : //Analogous assignments
        2,4,5,6,7 : //Analogous assignments
```

```
        default : //Analogous assignments
    endcase
end
```

我们要求case语句必须写成上面这种不带锁存的形式。

以上的结构我们可以看出锁存是针对左值产生的,并且锁存是在一定条件下产生的，产生锁存的充要条件是：

第一、always过程块的事件是非边沿的。

第二、各种情况下对左值的赋值没有显式确定。

为了不引入锁存，我们必须避免两个条件同时成立。

计数器：

时序部件的另一个重要器件是计数器，实际上计数器是寄存器的一种特殊应用，但是因为计数器往往用于产生分频时钟、用于表示电路的状态、以及产生固定的延迟，所以它的地位非常特殊我们把它专门拿出来进行阐述。如果电路需要2分频、4分频、8分频…这样的时钟，除了用锁相环外，最简单的实现方式是写成象下面这样的结构，当然它也可以用来标记2、4、8这样2的指数倍个状态。

```
always @(posedge clk or negedge clear)
begin
    if(!clear)
    begin
        count <= 4'b0000;
    end
    else
    begin
        count <= count + 1;
    end
end
```

如果分频数不是2的指数倍而是象3、5、…11等这样的数，或需要标记的状态不是2的指数倍那么我们可以设计成下面的这种结构。

```
always @(posedge clk or negedge clear)
begin
    if(!clear)
    begin
         count <= 3'b000;
     end
     else if(count==3' b101)
     begin
```

```
        count <= 3'b000;
    end
    else
    begin
        count <= count + 1;
    end
end
```

这种结构可以描述电路从0到5记6个数，但是在用计数器记录电路状态时因为计数器的初始状态与其他时序部件都用clear复位信号清零，所以我们一般不用计数器的初态表示电路的状态。

如果计数器需要在控制信号的作用下计数，那么其结构如下所示。

```
always @(posedge clk or negedge clear)
begin
    if(!clear)
    begin
        count <= 3'b000;
    end
    else if(en_count)
    begin
        count <= count + 1;
    end
end
```

分频电路产生的时钟是电路中时序器件工作的参照，在对时钟相位偏移要求不高或分频级数不多的情况下我们可以采用以上结构，但是如果遇到时序要求非常严格或者时钟的相位差作为电路工作的判别依据如要求时钟相差90度等，这时以计数器作为分频电路的精度是远远不够的，我们必须考虑用精度更高的锁相环重新设计分频、倍频电路，否则可能产生严重问题。

通过以上规范的说明，我们建立了RTL代码编写的基本框架，我们要求设计人员编写的RTL文件的基本结构与书写格式必须满足以上要求。

3.5 仿真代码的代码风格

设计人员一般在完成RTL代码和顶层模块后要对所设计的模块进行仿真，这一节我们简要阐述编写仿真文件应该注意的事项。

在仿真代码的语法范围一节中我们阐述了编制仿真代码可用的关键字、系统调用以及逻辑范畴。这一节我们重点介绍仿真代码中的时钟产生、信号延迟、边界向量、和存储器建模的相关问题。

时钟是时序逻辑赖以存在的关键信号，它的地位无与伦比，仿真代码对时钟的产生有专门的模型。一般产生时钟的建模语句应该考虑仿真工具对时钟的抽象级别，如果时钟建模语句与仿真工具对时钟抽象级相一致，会加快仿真速度。考虑到一般工具对时钟的抽象级为行为级，我们也只规范行为级的时钟建模。

- 占空比为1：1的时钟建模。

```
reg clk;
always begin
    #period/2  clk=1;
    #period/2  clk=0;
end
```

注意下面的写法会使仿真速度降低：

```
always #5 clk = ~clk;
```

这是因为每一个周期仿真工具会多进行一步非运算。

- 开头带有延迟的时钟建模。

```
 reg clk;
 initial
 begin
     clk=0;
     #(delay)
     forever
     begin
         #(period/2) clk=1;
         #(period/2) clk=0;
     end
 end 占空比不为 1:1 的周期性时钟建模。
reg clk;
always begin
    #4  clk=1;
    #8  clk=0;
end
```

- 时钟周期规律性变化的时钟建模。对这种时钟我们一般将延迟表示为一个差分方程，根据前面的时钟周期来确定现在的时钟周期。

```
integer period, cnt;
```

```
reg clk;
initial
begin
    clk = 0;
    period = 4;
……
end
initial
begin
    forever
    begin
        period = period*2;
        for(cnt = 0;cnt<3;cnt = cnt + 1)
        begin
            #(period/2) clk = 1;
            #(period/2) clk = 0;
        end
    end
end
```

上面这个例子可以创建一个周期变化的时钟，它的初始周期为4ns，以后每三个周期,它的时钟周期变宽一倍。用这种时钟对网表仿真时可以估计电路速度。

时钟的建模不止以上这几种，对时钟建模时，一定要把握准确、简单、直接的原则。如果时钟没有初态，要在initial语句块中指定。

对信号指定的具体的延迟是行为级代码的特征之一，它当然不会对应具体的物理实体，但在仿真时，指定的延迟给设计人员带来了巨大的便利性，它可以给信号安排恰当的时空顺序，测试期望的数据传输路径。延迟一般用#delay来表示，当然delay可以是一个复杂的表达式。在过程赋值语句中它表示经过delay延迟后将右值赋给左值。例如：#5 out = in；在单元（特指门单元）调用中，它表示经过delay延迟将输入传输到输出：例如：and #2 and2(out,a,b)；在实

例调用中它表示参数的传递的延迟，例如：

```
adder #2 adder1(.out(out),.add_a(a),.add_b(b));
```

仿真为了达到高的覆盖率，必须要仔细选择边界向量，边界向量指器件可能输入的数值边界、器件可能存储量的容量边界以及关键路径对应的最大延迟边界等。在边界内的其他数值我们只需选择有限的代表信号就可以了,但边界向量要一一仿真。输入的数值边界往往与时序是密不可分的，我们要仔细的安排它们的出场顺序。我们一般将它们放在initial语句块中。结构如下所示：

```
//将测试模块作为一个给被测模块输送信号和从被测模块接受信号的虚拟器件。
module test_adder(a,b,out);
//测试模块的输入连接被测模块的输出，从被测模块接受输出的数据。
    input out;
//测试模块的输出连接被测模块的输入，给被测模块提供输入向量
    output a,b;
    reg a, b;
    wire out;
//引用 adder 实例，使用端口对应模式。
  adder adder1(.out(out),.add_a(a),.add_b(b));
//加入激励信号
    initial
    begin
        a=0;
        b=1;
        #10 b=0;
        #10 a=1;
        #10 $stop;
```

end 如果我们不仅要用图形界面观察仿真波形，还要记录不同时刻仿真的状态数据，我们可以使用相应的系统调用。例如：

```
initial
begin
    $monitor ($time,“out=%b a=%b”,out,a,b);
end
```

其中$time返回当前的仿真时刻。其他的系统调用与此类似。

仿真代码有时还要包括虚拟器件的实现，实际工作中因为存储器一般不由综合工具综合生成，它使用全定制或IP来设计，但仿真时它们又必不可少，因此我们需要对ROM进行虚拟器件描述。

```
`timescale 1ns/1ns
module rom(read_data,addr,read_en_);
```

```
    input read_en_;
    input [3:0] addr;
    output [3:0] read_data;
    reg [3:0] read_data;
    reg [3:0] mem [0:15];
    initial
    begin
        $readmemb(“my_rom_data”,mem);
    end
    always @ (negedge read_en)
    begin
        read_data=mem[addr];
    end
endmodule
```

外部文件的数据可能为：

```
    0000
    0101
    1100
    ……
```

在有些电路设计中，ROM 的存储功能常用组合逻辑来代替，这时我们可以参考以下结构：

```
module rom(read_data,addr,read_en_);
    input read_en_;
    input [3:0] addr;
    output [3:0] read_data;
    wire [3:0] mem [0:3];
    reg [3:0] read_data;

    assign mem[4’b0000] = 4’b0001;
    assign mem[4’b0001] = 4’b0011;
    ……
    always @ (negedge read_en)
    begin
        read_data=mem[addr];
    end
endmodule
```

这种结构是可以综合的，当然它不会综合出我们常见的 ROM 结构来。

对 RAM 的虚拟模型可以描述如下：

```
`timescale 1ns/1ns
module mem(data,addr,read,write);
    inout [3:0] data;
    inout [3:0] addr;
    input read, write;
    reg [3:0] memory [0:15]; //4 bits, 16 words
```

```
//从存储器读出到总线上
assign data=(!read)? memory[addr]:4’bz;
//从总线写入存储器
always @ (negedge write)
begin
    memory[addr]=data;
end
```

endmodule 我们这里给出 ROM、RAM 的参考写法，实际工作中，这些存储器件可能还有其他要求，如：

- 同步读写。
- 多次读，同时进行一次写。
- 多次同步读写，同时提供一些方法保证一致性。

等等，具体设计时应根据具体情况灵活运用。