

```

//本模块生成 N 位的加法器
module adder ( co , sum , a0 , al , ci );
  //参数声明,本参数可以重新定义
  parameter N = 4;           //默认的总线位宽为 4

  //端口声明
  output [ N-1 : 0 ] sum ;
  output co ;
  input [ N-1 : 0 ] a0 , al ;
  input ci ;

  //根据总线的位宽,调用(实例引用)相应的加法器
  //参数 N 在调用(实例引用)时可以重新定义,调用(实例引用)
  //不同位宽的加法器是根据不同的 N 来决定的
  generate
    case ( N )
      //当 N=1 或 2 时分别选用位宽为 1 位或 2 位的加法器
      1 : adder_1bit adder1 ( co , sum , a0 , al , ci ); // 1 位的加法器
      2 : adder_2bit adder2 ( co , sum , a0 , al , ci ); // 2 位的加法器
      //默认的情况下选用位宽为 N 位的超前进位加法器
      default : adder_cla # ( N ) adder3 ( co , sum , a0 , al , ci );
    endcase
  endgenerate //生成块的结束

endmodule

```

5.8 举 例

本节中将使用下面两个例子说明前面各节中讨论的各种行为级结构的使用方法,并使用行为级语句对其进行描述并仿真。

5.8.1 四选一多路选择器

下面使用行为级的 case 语句来实现四选一多路选择器。这种风格的行为级描述可以通过综合工具转换为逻辑网表,不会对四选一多路选择器的电路实现和仿真结果造成影响。

[例 5.18] 行为级描述的四选一多路选择器。

```

//四选一多路器,其端口列表完全根据输入/输出图编写
module mux4_to_1 (out, i0, i1, i2, i3, s0);

  //根据输入/输出图的端口声明
  output out;
  input i0, i1, i2, i3;

```

```

input s1, s0;
//输出端口被声明为寄存器类型变量
reg out;

//若输入信号改变,则重新计算输出信号 out
//造成输出信号 out 重新计算的所有输入信号必须写入 always @(...)的电平敏感列表
always @(s1 or s0 or i0 or i1 or i2 or i3)
begin
    case ({s1, s0})
        2'b00: out = i0;
        2'b01: out = i1;
        2'b10: out = i2;
        2'b11: out = i3;
        default: out = 1'bx;
    endcase
end
endmodule

```

5.8.2 四位计数器

下面将用行为级对一个四位脉动进位计数器进行描述。在数据流级或门级,可以根据硬件实现方式将其设计成脉动进位、同步计数等。但是在行为级,可从一个更加抽象的角度来考虑问题的,并不关心具体的硬件实现方法,而只是对它的功能进行说明。计数器的行为级设计如[例 5.19]所示。从这个例子可以看到,行为级描述与逻辑结构描述相比是非常简洁的。如果输入信号的值不包括 x 和 z 的话,则使用行为级的描述代替逻辑结构描述不会对计数器的仿真结果造成影响。

[例 5.19] 四位计数器的行为级描述。

```

//四位二进制计数器
module counter ( Q , clock, clear );
    //输入/输出端口
    output [3 : 0] Q;
    input clock, clear;
    //输出变量 Q 被定义为寄存器类型
    reg [3 : 0] Q;

    always @ (posedge clear or negedge clock)
    begin
        if (clear)
            Q <= 4'd0;      //为了能生成诸如触发器一类的时序逻辑,建议使用非阻塞赋值
        else

```

```

Q <= Q + 1 ; //Q 是一个四位的寄存器,计数超过 15 后又会归零,因此模 16 没有必要
end

endmodule

```

小 结

在本章中学习了 Verilog 语法中几种条件语句、循环语句、块语句和生成语句的写法。有些语句和 C 语言很类似,所以比较容易理解。但也有一些语句则完全不同,应该注意到在 Verilog 语言中这些语句表示的不是一个直接的计算过程,而表示的是逻辑电路硬件的行为。因此,语句细微的差别其含义有很大的不同;通过综合生成的、对应的硬件也有很大的变化。必须认真理解这些细节才能够设计出符合要求的逻辑。所以,应格外要注意:if else 语句的 else 是不是你设计中想要的行为;在 case 语句中也要注意,如果条件都不符合究竟如何处理;在条件语句中是否存在无关的位,这些细节的考虑会使设计出的电路更加简洁,所以准确地理解 casex 和 casez 与 case 有什么不同也是很重要的。另外,for 循环变量的增加也与 C 不同,不能用简化的十写法。行为级的描述可根据电路实现的算法进行,不必包含硬件实现方面的细节。行为级设计一般用于设计初期,使用它来对各种与设计相关的折衷进行评估。在许多方面,行为建模与 C 语言编程很类似。

结构化的过程块,即 initial 和 always 块,构成了行为级建模的基础。其他所有的行为级语句只能出现在这两种块之中。initial 块只执行一次,而 always 块不断地反复执行,直到仿真结束。

行为级建模中的过程赋值用于对寄存器类型的变量赋值。阻塞赋值必须按照顺序执行,前面语句完成赋值之后才能执行后面的阻塞赋值;而非阻塞赋值将产生赋值调度,同时执行其后面的语句。

Verilog 中控制时序和语句执行顺序的 3 种方式是基于延迟的时序控制、基于事件的时序控制和电平敏感的时序控制。基于延迟的时序控制包括 3 种形式:常规延迟、0 延迟和内嵌延迟。基于事件的时序控制则包括常规事件、命名事件和 OR(或)事件。Wait 语句用于对电平敏感的时序控制。

行为级条件语句使用关键词 if_else 来表示。如果条件分支比较多,那么使用 case 语句更加方便。casex 语句和 casez 语句是 case 语句的特殊形式。

Verilog 中的 4 种循环语句分别用关键字 while、for、repeat 和 forever 表示。

顺序块和并行块使用两种类型的块语句。顺序块使用关键字 begin-end,而并行块使用关键字 fork-join 来表示。块可以具有名字,并且可以嵌套使用。如果块具有名字,那么可以在设计中的任何地方对其禁用。命名块能够通过层次名进行引用。

生成语句可以在仿真开始前的详细设计阶段动态地生成 Verilog 代码,这促进了参数化建模。当需要对矢量的多个位进行重复操作、模块实例的重复引用或根据参数的定义确定是否包括某一段代码的时候,使用生成语句是非常方便的。生成语句有 3 种类型是:循环生成语句、条件生成语句和 case 生成语句。

思 考 题

1. 为什么建议在编写 Verilog 模块程序时,如果用到 if 语句,建议大家把配套的 else 情况也考虑在内?
2. 用 if(条件 1)语句; elseif(条件 2)语句; elseif(条件 3)语句;... else 语句和用 case endcase 表示不同条件下的多个分支是完全相同的,还是有什么不同?
3. 如果 case 语句的分支条件没有覆盖所有可能的组合条件,定义了 default 项和没有定义 default 项有什么不同?
4. 仔细阐述 case、casex 和 casez 之间的不同。
5. forever 语句如果运行了,在它下面的语句能否运行? 它位于 begin-end 块和位于 fork join 块有什么不同?
6. forever 语句 repeat 语句能否独立于过程块而存在,即能否不在 initial 或 always 块中使用?
7. 用 for 循环为存储器许多单元赋值时是否需要时间? 为什么如果不定义时间延迟,它可以不需要时间就把不管多大的储存器赋值完毕?
8. for 循环是否可以表示可以综合的组合逻辑? 请举例说明。
9. 在编写测试模块时用什么方法可以使 for 循环按照时钟的节拍运行? 请比较图 5.3 所示程序段。

| | |
|--|---|
| <pre>always @(posedge clk) begin for (i=0; i<=1024; i=i+1) mem[i] = i; end</pre> <p>这样写能不能按照时钟节拍来对 mem[i]赋值? 右边框内的程序呢?</p> | <pre>initial begin for (i=0; i<=1024; i=i+1) begin mem[i] = i; @(posedge clk) end end</pre> |
|--|---|

图 5.3 两种程序段

10. 声明一个名为 oscillate 的寄存器变量并将它初始化为 0,使其每 30 个时间单位进行一次取反操作。不要使用 always 语句(提示:使用 forever 循环)。
11. 设计一个周期为 40 个时间单位的时钟信号,其占空比为 25%。使用 always 和 initial 块进行设计,将其在仿真 0 时刻的值初始化为 0。
12. 给定下面含有阻塞过程赋值语句的 initial 块,每条语句在什么仿真时刻开始执行? a、b、c 和 d 在仿真过程中的中间值和仿真结束时的值是什么?

```
initial
begin
  a = 1'b0 ;
  b = #10 1'b0 ;
  c = #5 1'b0 ;
  d = #20 { a , b , c } ;
end
```

13. 在第 12 题中,如果 initial 块中包括的是非阻塞过程赋值语句,那么各个问题的答案是什么?
14. 下面例子中 d 的最终值是什么?

```
initial
```

```

begin
    b = 1'b1 ; c = 1'b0 ;
    #10 b = 1'b0 ;
end
initial
begin
    d = #25 ( b | c ) ;
end

```

15. 使用带有同步清零端的 D 触发器(清零端高电平有效,在时钟下降沿执行清零操作)设计一个下降沿触发的 D 触发器,只能使用行为语句(提示:D 触发器的输出 q 应当声明为寄存器变量)。使用设计出的 D 触发器输出一个周期为 10 个时间单位的时钟信号。

16. 使用带有异步清零端的 D 触发器设计第 15 题中要求的 D 触发器(在清零端变为高电平后立即执行清零操作,无须等待下一个时钟下降沿),并对这个 D 触发器进行测试。

17. 使用 wait 语句设计一个电平敏感的锁存器,该锁存器的输入信号为 d 和 clock,输出为 q,其功能是当 clock=1 时 $q=d$ 。

18. 使用条件语句设计[例 5.17]中的四选一多路选择器,外部端口必须保持不变。

19. 使用 case 语句设计八功能的算术运算单元(ALU),其输入信号 a 和 b 均为 4 位,还有功能选择信号 select 为 3 位,输出信号为 out(5 位)。算术运算单元 ALU 所执行的操作与 Select 信号有关,具体关系如表 5.1 所列(忽略输出结果中的上溢和下溢的位)。

表 5.1 select 信号的功能

| select 信号 | 功 能 |
|-----------|----------------------|
| 3'b 000 | $out = a$ |
| 3'b 001 | $out = a + b$ |
| 3'b 010 | $out = a - b$ |
| 3'b 011 | $out = a/b$ |
| 3'b 100 | $out = a \% b$ (余数) |
| 3'b 101 | $out = a << 1$ |
| 3'b 110 | $out = a >> 1$ |
| 3'b 111 | $out = a > b$ (大小比较) |

20. 使用 while 循环设计一个时钟信号发生器。其时钟信号的初值为 0,周期为 10 个时间单位。
21. 使用 for 循环对一个长度为 1 024(地址从 0~1 023)、位宽为 4 的寄存器类型数组 cache_var 进行初始化,把所有单元都设置为 0。
22. 使用 forever 循环设计一个时钟信号,周期为 10,占空比为 40%,初值为 0。
23. 使用 repeat 将语句 $a=a+1$ 延迟 20 个时钟上升沿之后再执行。
24. 下面是一个内嵌顺序块和并行块的块语句。该块的执行结束时间是多少?事件的顺序是怎样的?每条语句的仿真结束时间是多少?

```

initial
begin
    x = 1'b0 ;

```

```
#5 y = 1'b1 ;
fork
#20 a = x ;
#15 b = y ;
join
#40 x = 1'b1 ;
fork
#10 p = x ;
begin
#10 a = y ;
#30 b = x ;
end
#5 m = y ;
join
end
```

25. 用 forever 循环语句、命名块(named block)和禁用(disabling of)命名块来设计一个八位计数器。这个计数器从 count = 5 开始计数,到 count = 67 结束计数。每个时钟正跳变沿计数器加一,时钟的周期为 10。计数器的计数只用了一次循环,然后就被禁用了(提示:使用 disable 语句)。

第6章 结构语句、系统任务、 函数语句和显示系统任务

概 述

在本章中将学习 Verilog 语法中两种结构语句以及如何定义和使用任务与函数,还有几个常用系统任务的用法。除了函数之外,这些语句在 C 语言中从来都没有定义过。特别需要注意的是,函数虽然在 C 语言中有过定义,但与 Verilog 的函数定义则完全不同。我们一定要注意这些语句所代表的物理意义,有意识地把结构语句、任务、函数与虚拟测试信号的生成或硬件电路结构联系起来,只有通过物理意义的深入理解,才能在设计中准确地应用。

6.1 结构说明语句

Verilog 语言中的任何过程模块都从属于以下 4 种结构的说明语句:

- (1) initial 说明语句;
- (2) always 说明语句;
- (3) task 说明语句;
- (4) function 说明语句。

一个程序模块可以有多个 initial 和 always 过程块。每个 initial 和 always 说明语句在仿真的一开始同时立即开始执行。initial 语句只执行一次,而 always 语句则是不断地重复活动着,直到仿真过程结束。但 always 语句后跟着的过程块是否运行,则要看它的触发条件是否满足,如满足则运行过程块一次,再次满足则再运行一次,直至仿真过程结束。

在一个模块中,使用 initial 和 always 语句的次数是不受限制的,它们都是同时开始运行的。

task 和 function 语句可以在程序模块中的一处或多处调用,其具体使用方法以后再详细地加以介绍。在“Verilog 数字设计基础”部分里,只对 initial 和 always 语句深入加以介绍。

6.1.1 initial 语句

initial 语句的格式如下:

```
initial
begin
    语句 1;
    语句 2;
    .....
    语句 n;
end
```

举例说明：

[例 6.1] 用 initial 块对存储器变量赋初始值。

```
initial
begin
    areg = 0;           // 初始化寄存器 areg
    for ( index = 0; index < size; index = index + 1 )
        memory[index]=0; // 初始化一个 memory
    end
```

在这个例子中用 initial 语句在仿真开始时对各变量进行初始化，注意这个初始化过程不需要任何仿真时间，即在 0 ns 时间内，便可以完成存储器的初始化工作。

[例 6.2] 用 initial 语句来生成激励波形。

```
initial
begin
    inputs = 'b000000; // 初始时刻为 0
    #10 inputs = 'b011001; (' 是英文输入法中的单引号)
    #10 inputs = 'b011011;
    #10 inputs = 'b011000;
    #10 inputs = 'b001000;
end
```

从这个例子中，可以看到 initial 语句的另一用途，即用 initial 语句来生成激励波形作为电路的测试仿真信号。

注意：一个模块中可以有多个 initial 块，它们都是并行运行的。initial 块常用于测试文件和虚拟模块的编写，用来产生仿真测试信号和设置信号记录等仿真环境。

6.1.2 always 语句

always 语句在仿真过程中是不断活动着的。但 always 语句后跟着的过程块是否执行，则要看它的触发条件是否满足，如满足则运行过程块一次；如不断满足，则不断地循环执行。其声明格式如下：

always <时序控制> <语句>

always 语句由于其不断活动的特性，只有和一定的时序控制结合在一起才有用。如果一个 always 语句没有时序控制，则这个 always 语句将会使仿真器产生死锁，见[例 6.3]。

[例 6.3] always areg = ~areg;

这个 always 语句将会生成一个 0 延迟的无限循环跳变过程，这时会发生仿真死锁。

但如果加上时序控制，则这个 always 语句将变为一条非常有用的描述语句，见[例 6.4]。

[例 6.4] always #half_period areg = ~areg;

这个例子生成了一个周期为：period(=2 * half_period) 的无限延续的信号波形。常用这种方法来描述时钟信号，并作为激励信号来测试所设计的电路。

[例 6.5]

```
reg [7:0] counter;
```

```

reg tick;
always @(posedge areg)
begin
    tick = ~tick;
    counter = counter + 1;
end

```

这个例子中,每当 areg 信号的上升沿出现时把 tick 信号反相,并且把 counter 增加 1。这种时间控制是 always 语句最常用的。

always 的时间控制可以是沿触发也可以是电平触发的,可以单个信号也可以多个信号,中间需要用关键字 or 连接,如图 6.1 所示。

```

always @(posedge clock or posedge reset)
begin
    :
end

```

```

always @ ( a or b or c )
begin
    :
end

```

沿触发的 always 块常常描述时序行为,如有限状态机。如果符合可综合风格要求,则可通过综合工具自动地将其转换为表示寄存器组和门级组合逻辑的结构,而该结构应具有时序所要求的行为;而电平触发的 always 块常常用来描述组合逻辑的行为。如果符合可综合风格要求,可通过综合工具自动将其转换为表示组合逻辑的门级逻辑结构或带锁存器的组合逻辑结构,而该结构应具有所要求的行为。一个模块中可以有多个 always 块,它们都是并行运行的。如果这些 always 块是可以综合的,则表示的是某种结构;如果不可综合,则是电路结构的行为,因此,多个 always 块并没有前后之分。

1. always 块的 OR 事件控制

有时,多个信号或者事件中任意一个发生的变化都能够触发语句或语句块的执行。在 Verilog 语言中,可以使用“或”表达式来表示这种情况。由关键词“or”连接的多个事件名或者信号名组成的列表称为敏感列表。关键词“or”被用来表示这种关系,或者使用“,”来代替。如 [例 6.6] 所示。

[例 6.6] OR 事件控制(敏感列表)。

```

//有异步复位的电平敏感锁存器
always @ ( reset or clock or d )
    //等待复位信号 reset 或时钟信号 clock,或输入信号 d 的改变
begin
    if ( reset )           //若 reset 信号为高,把 q 置零
        q = 1'b0 ;

```

由两个沿触发的 always 只要其中一个沿出现,就立即执行一次过程块

由多个电平触发的 always 块,只要 a、b、c 中任何一个发生变化,从高到低或从低到高都会执行一次过程块

图 6.1 always 的沿和电平触发

```

else if ( clock )      //若 clock 信号为高,锁存输入信号 d
    q = d ;
end

```

Verilog1364-2001 版本的语法中,对于原来的规定作了补充:关键词“or”也可以使用“,”来代替。[例 6.7]给出了使用逗号的例子。使用“,”来代替关键词“or”也适用于跳变沿敏感的触发器。

[例 6.7] 使用逗号的敏感列表。

```

//有异步复位的电平敏感锁存器
always @ ( reset , clock , d )
//等待复位信号 reset 或时钟信号 clock,或输入信号 d 的改变

begin
    if ( reset )          //若 reset 信号为高,把 q 置零
        q = 1'b0 ;
    else if ( clock )     //若 clock 信号为高,锁存输入信号 d
        q = d ;
end

//用 reset 异步下降沿复位,clock 正跳变沿触发的 D 寄存器
always @ ( posedge clk , negedge reset ) //注意:使用逗号来代替关键字 or
if ( ! reset )
    q <= 0 ;
else
    q <= d ;

```

如果组合逻辑块语句的输入变量很多,那么编写敏感列表会很烦琐并且容易出错。针对这种情况,Verilog 提供另外两个特殊的符号:@* 和@(*),它们都表示对其后面语句块中所有输入变量的变化是敏感的^[1]。[例 6.8]说明了如何用这两个符号表示组合逻辑的敏感列表。

[例 6.8] @* 操作符的使用。

```

//用 or 操作符的组合逻辑块
//编写敏感列表很烦琐并且容易漏掉一个输入
always @ ( a or b or c or d or e or f or g or h or p or m )
begin
    out1 = a? (b+c) : (d+e);
    out2 = f? (g+h) : (p+m);
end

//不用上述方法,用符号 @(*) 来代替,可以把所有输入变量都自动包括进敏感列表
always @ ( * )

```

[1] 读者可以阅读 Verilog1364-2001 版标准中关于这两个符号的使用细节及其限制。

```

begin
    out1 = a? (b+c) : (d+e);
    out2 = f? (g+h) : (p+m);
end

```

2. 电平敏感时序控制

前面所讨论的事件控制都需要等待信号值的变化或者事件的触发,使用符号@和后面的敏感列表来表示。Verilog 同时也允许使用另外一种形式表示的电平敏感时序控制(即后面的语句和语句块需要等待某个条件为真才能执行)。Verilog 语言用关键字 wait 来表示等待电平敏感的条件为真。

```

always
    wait (count_enable)
        # 20 count = count + 1;

```

在上面的例子中,仿真器连续监视 count_enable 的值,若其值为 0,则不执行后面的语句,仿真会停顿下来;如果其值为 1,则在 20 个时间单位之后执行这条语句。如果 count_enable 始终为 1,那么 count 将每过 20 个时间单位加 1。

6.2 task 和 function 说明语句

task 和 function 说明语句分别用来定义任务和函数,利用任务和函数可以把一个很大的程序模块分解成许多较小的任务和函数便于理解和调试。输入、输出和总线信号的值可以传入、传出任务和函数。任务和函数往往还是大的程序模块中在不同地点多次用到的相同的程序段。学会使用 task 和 function 语句可以简化程序的结构,使程序明白易懂,是编写较大型模块的基本功。

6.2.1 task 和 function 说明语句的不同点

任务和函数有些不同,主要的不同有以下 4 点:

- (1) 函数只能与主模块共用同一个仿真时间单位,而任务可以定义自己的仿真时间单位。
- (2) 函数不能启动任务,而任务能启动其他任务和函数。
- (3) 函数至少要有一个输入变量,而任务可以没有或有多个任何类型的变量。
- (4) 函数返回一个值,而任务则不返回值。

函数的目的是通过返回一个值来响应输入信号的值。任务却能支持多种目的,能计算多个结果值,这些结果值只能通过被调用的任务的输出或总线端口送出。Verilog HDL 模块使用函数时是把它当作表达式中的操作符,这个操作的结果值就是这个函数的返回值。下面可用例子来说明:

例如,定义一任务或函数对一个 16 位的字进行操作让高字节与低字节互换,把它变为另一个字(假定这个任务或函数名为: switch_bytes)。

任务返回的新字是通过输出端口的变量,因此 16 位字节互换任务的调用源码是:

```
switch_bytes(old_word,new_word);
```

任务 switch_bytes 把输入 old_word 字的高、低字节互换放入 new_word 端口输出。而函

数返回的新字是通过函数本身的返回值。因此 16 位字节互换函数的调用源码是：

```
new_word = switch_bytes(old_word);
```

下面分两节分别介绍任务和函数语句的要点。

6.2.2 task 说明语句

如果传给任务的变量值和任务完成后接收结果的变量已定义,就可以用一条语句启动任务,任务完成以后控制就传回启动过程。如任务内部有定时控制,则启动的时间可以与控制返回的时间不同。任务可以启动其他的任务,其他任务又可以启动别的任务,可以启动的任务数是没有限制的。不管有多少任务启动,只有当所有的启动任务完成以后,控制才能返回。

(1) 任务的定义 定义任务的语法如下:

```
task <任务名>;  
  <端口及数据类型声明语句>  
  <语句 1>  
  <语句 2>  
  :  
  <语句 n>  
endtask
```

这些声明语句的语法与模块定义中的对应声明语句的语法是一致的。

(2) 任务的调用及变量的传递 启动任务并传递输入、输出变量的声明语句的语法如下:
任务的调用:

```
<任务名>(端口 1, 端口 2, …, 端口 n);
```

下面的例子说明怎样定义任务和调用任务:

任务定义:

```
task my_task;  
  input a, b;  
  inout c;  
  output d, e;  
  :  
  <语句>          //执行任务工作相应的语句  
  :  
  c = fool;        //赋初始值  
  d = foo2;        //对任务的输出变量赋值  
  e = foo3;  
endtask
```

任务调用: my_task(v,w,x,y,z);

任务调用变量(v,w,x,y,z)和任务定义的 I/O 变量(a,b,c,d,e)之间是一一对应的。当任务启动时,由 v,w 和 x 传入的变量赋给了 a,b 和 c,而当任务完成后的输出又通过 c,d 和 e 赋给了 x,y 和 z。下面通过一个具体的例子来说明怎样在模块的设计中使用任务,使程序容易读懂。

[例 6.9] 描述红绿黄交通灯行为的 Verilog 模块, 其中使用了任务。

```

module traffic_lights;
    reg clock, red, amber, green;
    parameter on = 1, off=0, red_tics=350,
              amber_tics=30, green_tics=200;
    //交通灯初始化
    initial red=off;
    initial amber=off;
    initial green=off;
    //交通灯控制时序
    always
        begin
            red=on;           //开红灯
            light(red,red_tics); //调用等待任务
            green=on;         //开绿灯
            light(green,green_tics); //等待
            amber=on;         //开黄灯
            light(amber,amber_tics); //等待
        end
    //定义交通灯开启时间的任务
    task light;
        output color;
        input[31:0] tics;
        begin
            repeat(tics)
                @(posedge clock); //等待 tics 个时钟的上升沿
                color=off;          //关灯
        end
    endtask
    //产生时钟脉冲的 always 块
    always
        begin
            # 100 clock=0;
            # 100 clock=1;
        end
    endmodule

```

这个例子描述了一个简单的交通灯的时序控制, 并且该交通灯有它自己的时钟产生器。在这里, 该模块只是一个行为模块, 不能综合成电路网表。

6.2.3 function 说明语句

函数的目的是返回一个用于表达式的值。

(1) 定义函数的语法:

```
function <返回值的类型或范围> (函数名);
    <端口说明语句>
    <变量类型说明语句>
    begin
        <语句>
        .....
    end
endfunction
```

在这里,<返回值的类型或范围>这一项是可选项,如默认则返回值为一位寄存器类型数据。下面用例子说明:

```
function [7:0] getbyte;
    input [15:0] address;
    begin
        <说明语句>           //从地址字中提取低字节的程序
        getbyte = result_expression; //把结果赋予函数的返回字节
    end
endfunction
```

(2) 从函数返回的值: 函数的定义蕴含声明了与函数同名的、函数内部的寄存器。如在函数的声明语句中<返回值的类型或范围>为默认,则这个寄存器是一位的;否则是与函数定义中<返回值的类型或范围>一致的寄存器。函数的定义把函数返回值所赋值寄存器的名称初始化为与函数同名的内部变量。下面的例子说明了这个概念:getbyte被赋予的值就是函数的返回值。

(3) 函数的调用: 函数的调用是通过将函数作为表达式中的操作数来实现的。其调用格式如下:

<函数名> (<表达式>, ... <表达式>)

其中函数名作为确认符。下面的例子中,两次调用函数 getbyte,把两次调用产生的值进行位拼接运算,以生成一个字。

word = control? {getbyte(msbyte),getbyte(lsbyte)} : 0;

(4) 函数的使用规则: 与任务相比较函数的使用有较多的约束,下面给出的是函数的使用规则:

- ① 函数的定义不能包含有任何的时间控制语句,即任何用#、@、或 wait 来标识的语句。
- ② 函数不能启动任务。
- ③ 定义函数时至少要有一个输入参量。
- ④ 在函数的定义中必须有一条赋值语句给函数中的一个内部变量赋以函数的结果值,该内部变量具有和函数名相同的名字。

(5) 举例说明: 下面的例子中定义了一个可进行阶乘运算的名为 factorial 的函数。该函数返回一个 32 位的寄存器类型的值,还可向后调用自身,并且打印出部分结果值。

[例 6.10] 阶乘函数的定义和调用。

```

module tryfact;
    //函数的定义
    function[31:0]factorial;
        input[3:0]operand;
        reg[3:0]index;
        begin
            factorial = 1; //0 的阶乘为 1, 1 的阶乘也为 1
            for(index=2; index<=operand; index=index+1)
                factorial = index * factorial;
        end
    endfunction
    //函数的测试
    reg[31:0]result;
    reg[3:0]n;
    initial
    begin
        result=1;
        for(n=2;n<=9;n=n+1)
        begin
            $display("Partial result n= %d result= %d", n, result);
            result = n * factorial(n)/((n*2)+1);
        end
        $display("Final result= %d",result);
    end
endmodule //模块结束

```

前面已经介绍了足够的语句类型可以编写一些完整的模块。接着将举许多实际的例子介绍函数的使用。这些例子都给出了完整的模块描述,因此可以对它们进行仿真测试和结果检验。通过学习和练习能逐步掌握利用 Verilog HDL 设计数字系统的方法和技术。

6.2.4 函数的使用举例

在本节中将讨论两个例子。第一个例子是奇偶校验位计算器,它的返回值是一个一位的值;第二个例子对能左/右移位的 32 位寄存器建模,它的返回值是移位后的 32 位值。

1. 奇偶校验位的计算

这个函数的功能是计算 32 位地址值的偶校验位,并且返回校验位的值。在这个例子中假设采用了偶校验。[例 6.11]给出了函数 calcParity 的定义和调用。

[例 6.11] 偶校验位的计算。

```

//定义一个模块,其中包含能计算偶校验位的函数(calcParity)
module parity;
    reg [31:0] addr;
    reg parity;

```

```

initial
begin
    addr = 32'h3456_789a;
    #10 addr = 32'hc4c6_78ff;
    #10 addr = 32'hff56_ff9a;
    #10 addr = 32'h3faa_aaaa;
end

//每当地址值发生变化,计算新的偶校验位
always @(addr)
begin
    parity = calc_parity(addr); //第一次启动校验位计算函数 calc_parity
    $display("Parity calculated = %b", calc_parity(addr));
    //第二次启动校验位计算函数 calc_parity
end

//定义偶校验计算函数
function calc_parity;
input [31:0] address;
begin
    //适当地设置输出值,使用隐含的内部寄存器 calc_parity
    calc_parity = ~address; //返回所有地址位的异或值
end
endfunction

endmodule

```

在函数第一次被调用时,它的返回值被用来设置寄存器变量 parity;在第二次被调用时,它的返回值直接使用系统任务 \$ display 进行显示。从这点可以看出,函数的返回值被用在函数调用的地方。声明函数变量的另一种方法是使用 C 风格的描述。[例 6.12] 给出了使用 C 风格进行变量声明的 calc_parity 定义。

[例 6.12] 使用 C 风格进行变量声明的函数定义。

```

//定义偶校验位计算函数,该函数采用 ANSI C 风格的变量声明
function calc_parity(input [31:0] address);
begin
    //适当地设置输出值,使用隐含的内部寄存器 calc_parity
    calc_parity = ~address; //返回所有地址位的异或值
end
endfunction

```

2. 左/右移位寄存器

为了说明如何声明函数的输出范围可考虑一个具有移位功能的函数。该函数根据控制信号的不同将一个 32 位数每次左移或者右移一位。[例 6.13] 定义了这个函数。

[例 6.13] 左/右移位寄存器。

```

//定义一个包含移位函数的模块
module shifter;

//左/右移位寄存器
`define LEFT_SHIFT      1'b0
`define RIGHT_SHIFT     1'b1

reg [31:0] addr, left_addr, right_addr;
reg control;

//每当新地址出现时就计算右移位和左移位的值
always @(addr)
begin
    //调用下面定义的具有左右移位功能的函数
    left_addr = shift(addr, `LEFT_SHIFT);
    right_addr = shift(addr, `RIGHT_SHIFT);
end

//定义移位函数,其输出是一个 32 位的值
function [31:0] shift;
    input [31:0] address;
    input control;
    begin
        //根据控制信号适当地设置输出值
        shift = (control == `LEFT_SHIFT) ? (address << 1) : (address >> 1);
    end
endfunction

endmodule

```

6.2.5 自动(递归)函数

Verilog 中的函数是不能够进行递归调用的。设计模块中若某函数在两个不同的地方被同时并发调用,由于这两个调用同时对同一块地址空间进行操作,那么计算结果将是不确定的。

若在函数声明时使用了关键字 automatic,那么该函数将成为自动的或可递归的,即仿真器为每一次函数调用动态地分配新的地址空间,每一个函数调用对各自的地址空间进行操作。因此,自动函数中声明的局部变量不能通过层次名进行访问。而自动函数本身可以通过层次名进行调用。

[例 6.14]说明如何定义自动函数,来完成阶乘运算。

[例 6.14] 递归(自动)函数。

```
//用函数的递归调用定义阶乘计算
```

```
module top;
```

```
:
```

```
//定义自动(递归)函数
```

```
function automatic integer factorial;
```

```

input [ 31 : 0 ] oper ;
integer i ;
begin
    if ( operand >= 2 )
        factorial = factorial ( oper - 1 ) * oper ; //递归调用
    else
        factorial = 1 ;
end
endfunction

//调用该函数
integer result ;
initial
begin
    result = factorial ( 4 ) ; //调用 4 的阶乘
    $display ( " Factorial of 4 is % 0d ", result ) ; //显示 24
end
.....
.....
endmodule

```

6.2.6 常量函数

常量函数实际上是一个带有某些限制的常规 Verilog 函数。这种函数能够用来引用复杂的值，因而可用来代替常量。

在[例 6.15]中声明了一个常量函数，它可以用来自计算模块中地址总线的宽度。

[例 6.15] 常量函数。

```

module ram ( ..... );
parameter RAM_DEPTH = 256 ;
input [ clog2( RAM_DEPTH ) - 1 : 0 ] addr_bus ; //计算出的常量值
:
:
:
// 
function integer clogb2 ( input integer depth ) ;
begin
    for ( clogb2 = 0 ; depth > 0 ; clogb2 = clogb2 + 1 )
        depth = depth >> 1 ;
    end
endfunction
:
:
:
```

```
endmodule
```

6.2.7 带符号函数

带符号函数的返回值可以作为带符号数进行运算, [例 6.16] 给出了带符号函数的例子。

[例 6.16] 带符号函数。

```
module top ;
;
//
//
function signed [ 63 : 0 ] compute_signed ( input [ 63 : 0 ] vector ) ;
;
;
;
endfunction

//
if ( compute_signed ( vector ) < -3 )
begin
;
end
;
endmodule
```

6.3 关于使用任务和函数的小结

在前述中已对 Verilog 行为建模中使用的任务和函数进行了讨论, 可概括为以下特点:

- (1) 任务和函数都是用来对设计中多处使用的公共代码进行定义; 使用任务和函数可以将模块分割成许多个可独立管理的子单元, 增强了模块的可读性和可维护性; 它们和 C 语言中的子程序起相同的作用。
- (2) 任务可以具有任意多个输入、输入/输出(inout)和输出变量; 在任务中可以使用延迟、事件和时序控制结构, 在任务中可以调用其他的任务和函数。
- (3) 可重入任务使用关键字 automatic 进行定义, 它的每一次调用都对不同的地址空间进行操作。因此在被多次并发调用时, 它仍然可以获得正确的结果。
- (4) 函数只能有一个返回值, 并且至少要有一个输入变量; 在函数中不能使用延迟、事件和时序控制结构, 但可以调用其他函数, 不能调用任务。
- (5) 当声明函数时, Verilog 仿真器都会隐含地声明一个同名的寄存器变量, 函数的返回值通过这个寄存器传递回调用处。
- (6) 递归函数使用关键字 automatic 进行定义, 递归函数的每一次调用都拥有不同的地址空间。因此对这种函数的递归调用和并发调用可以得到正确的结果。
- (7) 任务和函数都包含在设计层次之中, 可以通过层次名对它们进行调用。

6.4 常用的系统任务

本节中将讨论 Verilog 语言中一些常用的系统任务及各自适用不同的场合。还将讨论用于文件输出、显示层次、选通显示(strobing)、存储器初始化和值变转储的系统任务^[3]。

6.4.1 \$ display 和 \$ write 任务

格式：

\$ display (p1,p2,…,pn);

\$ write (p1,p2,…,pn);

这两个函数和系统任务的作用是用来输出信息，即将参数 p2 到 pn 按参数 p1 给定的格式输出。参数 p1 通常称为“格式控制”，参数 p2 至 pn 通常称为“输出表列”。这两个任务的作用基本相同。\$ display 自动地在输出后进行换行，\$ write 则不是这样。如果想在一行里输出多个信息，可以使用 \$ write。在 \$ display 和 \$ write 中，其输出格式控制是用双引号括起来的字符串，它包括以下两种信息：

(1) 格式说明，由 "%" 和格式字符组成。它的作用是将输出的数据转换成指定的格式输出。格式说明总是由 "%" 字符开始的。对于不同类型的数据用不同的格式输出。表 6.1 中给出了常用的几种输出格式。

表 6.1 输出格式及说明

| 输出格式 | 说 明 |
|---------|-------------------------------------|
| %h 或 %H | 以十六进制数的形式输出 |
| %d 或 %D | 以十进制数的形式输出 |
| %o 或 %O | 以八进制数的形式输出 |
| %b 或 %B | 以二进制数的形式输出 |
| %c 或 %C | 以 ASCII 码字符的形式输出 |
| %v 或 %V | 输出网络型数据信号强度 |
| %m 或 %M | 输出等级层次的名字 |
| %s 或 %S | 以字符串的形式输出 |
| %t 或 %T | 以当前的时间格式输出 |
| %e 或 %E | 以指数的形式输出实型数 |
| %f 或 %F | 以十进制数的形式输出实型数 |
| %g 或 %G | 以指数或十进制数的形式输出实型数 无论何种格式都以较短的结果输出 |

(2) 普通字符，即需要原样输出的字符。其中一些特殊的字符可以通过表 6.2 中的转换序列来输出。表中的字符形式用于格式字符串参数中，用来显示特殊的字符。

[3] 其他系统任务，例如用作符号转换的 \$ signed 和 \$ unsigned 在本书中不讨论。详细的细节请参考“IEEE 标准 Verilog 硬件描述语言”文档。

表 6.2 换码序列及功能

| 换码序列 | 功 能 |
|------|-----------------|
| \n | 换 行 |
| \t | 横向跳格(即跳到下一个输出区) |
| \\\ | 反斜杠字符\ |
| \" | 双引号字符" |
| \o | 1~3位八进制数代表的字符 |
| %% | 百分符号% |

在 \$display 和 \$write 的参数列表中,其“输出表列”是需要输出一些数据,可以是表达式。下面举几个例子说明一下。

[例 6.17]

```
module disp;
initial
begin
    $display("\\\\t%\\n\\\"123");
end
endmodule
```

输出结果为

```
\%
"S
```

从这个例子中可以看到一些特殊字符的输出形式(八进制数 123 就是字符 S)。

[例 6.18]

```
module disp;
reg[31:0] rval;
pulldown(pd);
initial
begin
rval=101;
    $display("rval=%h hex %d decimal", rval, rval);
    $display("rval=%o octal %b binary", rval, rval);
    $display("rval has %c ascii character value", rval);
    $display("pd strength value is %v", pd);
    $display("current scope is %m");
    $display("%s is ascii value for 101",101);
    $display("simulation time is %t", $time);
end
endmodule
```

其输出结果为:

rval=00000065 hex 101 decimal

rval=00000000145 octal 00000000000000000000000000001100101 binary

rval has e ascii character value

pd strength value is StX

current scope is disp

e is ascii value for 101

simulation time is 0

输出数据的显示宽度：在 \$display 中，输出列表中数据的显示宽度是自动按照输出格式进行调整的。这样在显示输出数据时，在经过格式转换以后，总是用表达式的最大可能值所占的位数来显示表达式的当前值。在用十进制数格式输出时，输出结果前面的 0 值用空格来代替。对于其他进制，输出结果前面的 0 仍然显示出来。例如对于一个值的位宽为 12 位的表达式，如按照十六进制数输出，则输出结果占 3 个字符的位置；如按照十进制数输出，则输出结果占 4 个字符的位置。这是因为这个表达式的最大可能值为 FFF(十六进制)、4 095(十进制)。可以通过在 % 和表示进制的字符中间插入一个 0 自动调整显示输出数据宽度的方式。见下例：

```
$display("d=%0h a=%0h",data,addr);
```

这样在显示输出数据时，在经过格式转换以后，总是用最少的位数来显示表达式的当前值。下面举例说明：

〔例 6.19〕

```
module printval;  
    reg[11:0] r1;  
begin  
    initial  
        r1 = $random;  
        $display("r1 = %d", r1);  
    end  
endmodule
```

如果输出列表中表达式的值包含有不确定的值或高阻值，其结果输出遵循以下规则：

(1) 在输出格式为十进制的情况下:

- ① 如果表达式值的所有位均为不定值,则输出结果为小写的 x。
 - ② 如果表达式值的所有位均为高阻值,则输出结果为小写的 z。
 - ③ 如果表达式值的部分位为不定值,则输出结果为大写的 X。
 - ④ 如果表达式值的部分位为高阻值,则输出结果为大写的 Z。

(2) 在输出格式为十六进制和八进制的情况下:

 - ① 每 4 位二进制数为一组代表一位十六进制数,每 3 位二进制数为一组代表一位八进

制数。

② 如果表达式值相对应的某进制数的所有位均为不定值,则该位进制数的输出结果为小写的 x。

③ 如果表达式值相对应的某进制数的所有位均为高阻值,则该位进制数的输出结果为小写的 z。

④ 如果表达式值相对应的某进制数的部分位为不定值,则该位进制数输出结果为大写的 X。

⑤ 如果表达式值相对应的某进制数的部分位为高阻值,则该位进制数输出结果为大写的 Z。

对于二进制输出格式,表达式值的每一位的输出结果为 0、1、x、z。下面举例说明:

语句输出结果:

| | |
|---|---------------|
| \$ display("%d", 1'bx); | 输出结果为:x |
| \$ display("%h", 14'bx0_1010); | 输出结果为:xxXa |
| \$ display("%h %o", 12'b001x_xx10_1x01, 12'b001_xxx_101_x01); | 输出结果为:XXX1x5X |

注意:因为 \$ write 在输出时不换行,要注意它的使用。可以在 \$ write 中加入换行符\n,以确保明确的输出显示格式。

6.4.2 文件输出

Verilog 的结果通常输出到标准输出和文件 verilog.log 中。可以将 Verilog 的输出重新定向到选择的文件。

1. 打开文件

文件可以用系统任务 \$ fopen 打开。

用法: \$ fopen("<文件名>");^[3]

用法: <文件句柄> = \$ fopen("<文件名>");

任务 \$ fopen 返回一个被称为多通道描述符 (multichannel descriptor)^[4] 的 32 位值。多通道描述符中只有一位被设置成 1。标准输出有一个多通道描述符,其最低位(第 0 位)被设置成 1。标准输出也称为通道 0。标准输出一直是开放的。以后对 \$ fopen 的每一次调用打开一个新的通道,并且返回一个设置了第 1 位、第 2 位等,直到 32 位描述符的第 30 位。第 31 位是保留位。通道号与多通道描述符中被设置为 1 的位相对应。[例 6.20]说明了文件描述符的使用方法。

[例 6.20] 文件描述符。

```
//多通道描述符
integer handle1, handle2, handle3; //整型数为 32 位
//标准输出是打开的; descriptor = 32'h0000_0001 (第 0 位置 1)
```

[3] “IEEE 标准 Verilog 硬件描述语言”文档提供了 \$ fopen 的其他功能。本书提到的 \$ fopen 语法对大多数应用是足够的。然而,如果需要其他的功能,可参考“IEEE 标准 Verilog 硬件描述语言”文档。

[4] “IEEE 标准 Verilog 硬件描述语言”文档提供了使用单通道(single-channel)文件描述符最多可以打开 2³⁰个文件的方法。详细的细节可参考该文档。

```

initial
begin
    handle1 = $fopen("file1.out"); //handle1 = 32'h0000_0002 (bit 1 set 1)
    handle2 = $fopen("file2.out"); //handle2 = 32'h0000_0004 (bit 2 set 1)
    handle3 = $fopen("file3.out"); //handle3 = 32'h0000_0008 (bit 3 set 1)
end

```

多通道描述符的优点在于可以有选择地同时写多个文件。下面将详细解释这一点。

2. 写文件

系统任务 \$fdisplay、\$fmonitor、\$fwrite 和 \$fstrobe 都用于写文件^[5]。

注意：这些任务在语法上与常规系统任务 \$display、\$monitor 等类似，但是它们提供了额外的写文件功能。

下面将只考虑 \$fdisplay 和 \$fmonitor 任务。

用法：
 $\$fdisplay(<\text{文件描述符}>, p1, p2, \dots, pn);$

$\$fmonitor(<\text{文件描述符}>, p1, p2, \dots, pn);$

$p1, p2, \dots, pn$ 可以是变量、信号名或者带引号的字符串。文件描述符是一个多通道描述符，它可以是一个文件句柄或者多个文件句柄按位的组合。Verilog 会把输出写到与文件描述符中值为 1 的位相关联的所有文件中。下面将使用 [例 6.20] 中定义的文件描述符来解释 \$fdisplay 和 \$fmonitor 任务的使用。

```

//所有的句柄已经在[例 6.18]中定义
//写到文件中去
integer desc1, desc2, desc3; //3个文件的描述符
initial
begin
    desc1 = handle1 | 1; //按位或; desc1 = 32'h0000_0003
    $fdisplay(desc1, "Display 1"); //写到文件 file1.out 和标准输出 stdout

    desc2 = handle2 | handle1; //desc2 = 32'h0000_0006
    $fdisplay(desc2, "Display 2"); //写到文件 file1.out 和 file2.out

    desc3 = handle3; //desc3 = 32'h0000_0008
    $fdisplay(desc3, "Display 3"); //只写到文件 file3.out
end

```

3. 关闭文件

文件可以用系统任务 \$fclose 来关闭。

[5] “IEEE 标准 Verilog 硬件描述语言”文档提供了许多用于文件输出的其他功能。本书提到的文件输出系统任务对于大多数数字电路设计者是足够用的。然而，如果需要使用文件输出的其他功能，可参考“IEEE 标准 Verilog 硬件描述语言”文档。IEEE 标准 Verilog 硬件描述语言也提供了读文件的系统任务。这些系统任务包括 \$fgetc、\$ungetc、\$fscanf、\$sscanf、\$fread、\$ftell、\$fseek、\$rewind 和 \$fflush。然而，大多数数字电路设计者不经常需要这些功能。因此，本书没有涉及。如果需要使用这些读文件的功能，可参考“IEEE 标准 Verilog 硬件描述语言”文档。

用法: \$fclose(<文件描述符>);

```
//关闭文件
fclose(handle1);
```

文件一旦被关闭就不能再写入。多通道描述符中的相应位被设置为0,下一次\$ fopen 的调用可以重用这一位。

6.4.3 显示层次

通过任何显示任务,比如\$display、\$write、\$monitor 或者 \$strobe 任务中的%m 选项的方式可以显示任何级别的层次,这是非常有用的选项。例如,当一个模块的多个实例执行同一段 Verilog 代码时,%m 选项会区分哪个模块实例在输出。显示任务中的%m 选项无需参数,参见[例 6.21]。

[例 6.21] 显示层次。

```
//显示层次信息
module M;
initial $display("Displaying in %m");
endmodule

//调用模块 M
module top;
M m1();
M m2();
M m3();
endmodule
```

仿真输出如下所示:

Displaying in top.m1

Displaying in top.m2

Displaying in top.m3

这一特征可以显示全层次路径名,包括模块实例、任务、函数和命名块。

6.4.4 选通显示

选通显示(Strobing)由关键字为 \$strobe 的系统任务完成。这个任务与 \$display 任务除了一点小差异外,其他非常相似。如果许多其他语句与 \$display 任务在同一个时间单位执行,那么这些语句与 \$display 任务的执行顺序是不确定的。如果使用 \$strobe,该语句总是在同时刻的其他赋值语句执行完成之后才执行。因此,\$strobe 提供了一种同步机制,它可以确保所有在同一时钟沿赋值的其他语句在执行完毕之后才显示数据,参见[例 6.22]。

[例 6.22] 选通显示。

```
//选通显示
always @ (posedge clock)
begin
    a = b ;
    c = d ;
end

always @ (posedge clock)
    $strobe ("Displaying a = %b, c = %b", a, c); //显示正跳变沿时刻的值
```

在[例 6.22]中,时钟上升沿的值在语句 `a = b` 和 `c = d` 执行完之后才显示。如果使用 `$display`,`$display` 可能在语句 `a = b` 和 `c = d` 之前执行,结果显示不同的值。

6.4.5 值变转储文件

值变转储文件(VCD)是一个 ASCII 文件,它包含仿真时间、范围与信号的定义以及仿真运行过程中信号值的变化等信息。设计中的所有信号或者选定的信号集合在仿真过程中都可以被写入 VCD 文件。后处理工具可以把 VCD 文件作为输入并把层次信息、信号值和信号波形显示出来。现在有许多商业后处理工具以及集成到仿真器中的工具可供使用。对于大规模设计的仿真,设计者可以把选定的信号转储到 VCD 文件中,并使用后处理工具去调试、分析和验证仿真输出结果。在调试过程中 VCD 文件的使用流程如图 6.2 所示。

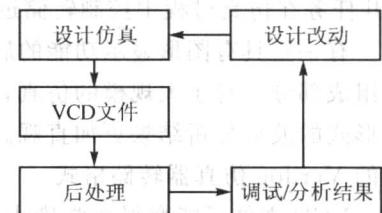


图 6.2 用仿真产生的 VCD 文件进行分析和查错

Verilog 提供了系统任务来选择要转储的模块实例或者模块实例信号(`$dumpvars`),选择 VCD 文件的名称(`$dumpfile`),选择转储过程的起点和终点(`$dumpon`,`$dumpoff`),选择生成检测点(`$dumpall`),每个任务的使用方法如[例 6.23]所示。

[例 6.23] VCD 文件系统任务。

```
//指定 VCD 文件名。若不指定 VCD 文件,则由仿真器指定一默认文件名
initial
    $dumpfile (" myfile.dmp"); //仿真信息转储到 myfile.dmp 文件
//转储模块中的信号
initial
    $dumpvars ; //没有指定变量范围,把设计中全部信号都转储
initial
    $dumpvars ( 1, top ) ; //转储模块实例 top 中的信号
//数 1 表示层次的等级,只转储 top 下第一层信号
//即转储 top 模块中的变量,而不转储在 top 中调用
//模块中的变量
initial
```

```

$ dumpvars (2, top.m1) ;      //转储 top.m1 模块下两层的信号
initial
$ dumpvars (0, top.m1) ;      //数 0 表示转储 top.m1 模块下面各个层的所有信号

//启动和停止转储过程
initial
begin
    $ dumpon ;                //启动转储过程
    #100000 $ dumpoff ;       //过了 100 000 个仿真时间单位后,停止转储过程
end

//生成一个检查点,转储所有 VCD 变量的现行值
initial
$ dumpall ;

```

\$ dumpfile 和 \$ dumpvars 任务通常在仿真开始时指定, \$ dumpon、\$ dumpoff 和 \$ dumpall 任务在仿真过程中控制转储过程^[6]。

有一些具有图形显示功能的后处理工具可供商业上的应用,是目前仿真和调试过程的重要组成部分。对于大规模的仿真,设计者难以分析 \$ display 和 \$ monitor 语句的输出。从图形形式的波形分析结果更加直观。VCD 之外的其他格式也已经出现,但是 VCD 仍然是最流行的 Verilog 仿真器转储格式。

VCD 文件可能变得非常庞大(对大规模设计而言,VCD 文件的大小可能达到数百兆字节),因而只能选择那些需要检查的信号进行转储,注意到这一点是很重要的。

6.5 其他系统函数和任务

Verilog HDL 语言中还有以下一些常用的系统函数和任务:

```

$ bitstoreal, $ rtoi, $ setup, $ finish, $ skew, $ hold,
$ setuphold, $ itor, $ period, $ time, $ printtimescale,
$ timefoemat, $ realtime, $ width, $ realtobits, $ recovery,

```

在 Verilog HDL 语言中每个系统函数和任务前面都用一个标识符 \$ 来加以确认,这些系统函数和任务提供了非常强大的功能。有兴趣的同学可以参阅附录:Verilog 语言参考手册。

小结

在本章中我们学习了 Verilog 语法中两种最重要的结构语句 initial 和 always;还学习了如何定义和使用任务与函数,及几个常用系统任务:\$ display、\$ write、\$ strobe、\$ fopen、

^[6] 其他任务,例如 \$ dumpports、\$ dumpportsoff、\$ dumpportson、\$ dumpportsall、\$ dumpportslimit 和 \$ dumpportsflush 等细节可参考“IEEE 标准 Verilog 硬件描述语言”文档。

\$fclose、\$fdisplay、\$fmonitor 等的用法。

需要牢记的是：

- (1) 一个程序模块可以有多个 initial 和 always 过程块。
- (2) 每个 initial 和 always 说明语句在仿真的一开始便同时立即开始运行。
- (3) initial 语句在模块中只执行一次。
- (4) always 语句则是不断地活动着，直到仿真过程结束。
- (5) always 语句后跟着的过程块是否运行，则要看它的触发条件是否满足，如满足则运行过程块一次，再次满足则再运行一次，循环往复直至仿真过程结束。
- (6) always 的时间控制可以是沿触发也可以是电平触发的，可以单个信号也可以多个信号，中间需要用关键字 or 连接。
- (7) 沿触发的 always 块常常描述时序行为，如有限状态机。
- (8) 而电平触发的 always 块常常用来描述组合逻辑的行为。
- (9) \$display 和 \$write 与 C 语言的对应语句的格式控制很类似；但有些不同需要注意，\$strobe 显示变量的时刻比 \$display 确定。
- (10) 任务和函数在以后还要详细讲解，目前不必花费太多的时间。
- (11) 文件的读写与 C 语言很类似，可以参考本章的例子学习使用。编写复杂系统的测试模块，掌握文件的读写是非常有必要的。

思考题

1. 怎样理解 initial 语句只执行一次的概念？
2. 在 initial 语句引导的过程块中是否可以有循环语句？如果可以，是否与思考题 1. 互相矛盾？
3. 怎样理解由 always 语句引导的过程块是不断活动的？
4. 不断活动与不断执行有什么不同？
5. 怎样理解沿触发和电平触发的不同？
6. 是不是可以说沿触发是有间隔的，在一定的时间区间里只需要注意有限的点，而电平触发却需要注意无穷多个点？
7. 沿触发的 always 块和电平触发的 always 块各表示什么类型的逻辑电路的行为？为什么？
8. 简单叙述任务与函数的不同点。
9. 简单叙述 \$display、\$write 和 \$strobe 的不同点。
10. 简单叙述 Verilog1364-2001 版语法规定的电平敏感列表的简化写法。
11. 如何在 Verilog 测试模块中，利用文件的读写产生预定格式的信号，并记录有测试价值的信号？

第 7 章 调试用系统任务和常用编译预处理语句

概 述

在本章中将学习 Verilog 语法中几种常用于调试和查错的系统任务以及编写实用模块时常用的编译预处理语句。其中有许多系统任务 C 语言中是没有的,有些虽然类似,但存在很大的不同;编译预处理语句与 C 语言中的类似,但也有所不同,需要注意。在学习中我们要注意这些语句含义、使用的场合和在程序模块中的位置,有意识地把这些系统任务与测试模块的编写联系来。只有深入理解了有关语法的实质,才能在设计中准确地应用。

7.1 系统任务 \$ monitor

格式:

```
$ monitor (p1, p2, ..., pn);
$ monitor;
$ monitoron;
$ monitoroff;
```

任务 \$ monitor 提供了监控和输出参数列表中的表达式或变量值的功能。其参数列表中输出控制格式字符串和输出表列的规则和 \$ display 中的一样。当启动一个带有一个或多个参数的 \$ monitor 任务时,仿真器则建立一个处理机制,使得每当参数列表中变量或表达式的值发生变化时,整个参数列表中变量或表达式的值都将输出显示。如果同一时刻,两个或多个参数的值发生变化,则在该时刻只输出显示一次。但在 \$ monitor 中,参数可以是 \$ time 系统函数。这样参数列表中变量或表达式的值同时发生变化的时刻可以通过标明同一时刻的多行输出来显示。如:

```
$ monitor ( $ time, , "rxo=%b txo=%b", rxo, txo );
```

在 \$ display 中也可以这样使用。注意在上面的语句中,“,”代表一个空参数。空参数在输出时显示为空格。

\$ monitoron 和 \$ monitoroff 任务的作用是通过打开和关闭监控标志来控制监控任务 \$ monitor 的启动和停止,这样使得程序员可以很容易地控制 \$ monitor 何时发生。其中 \$ monitoroff 任务用于关闭监控标志,停止监控任务 \$ monitor, \$ monitoron 则用于打开监控标志,启动监控任务 \$ monitor。通常在通过调用 \$ monitoron 来启动 \$ monitor 时,不管 \$ monitor 参数列表中的值是否发生变化,总是立刻输出显示当前时刻参数列表中的值,这用于在监控的初始时刻设定初始比较值。在默认情况下,控制标志在仿真的起始时刻就已经打开了。在多模块调试的情况下,许多模块中都调用了 \$ monitor,因为任何时刻只能有一个

\$monitor 起作用,因此需配合 \$monitoron 与 \$monitoroff 使用,把需要监视的模块用 \$monitoron 打开,在监视完毕后及时用 \$monitoroff 关闭,以便把 \$monitor 让给其他模块使用。\$monitor 与 \$display 的不同处还在于 \$monitor 往往在 initial 块中调用,只要不调用 \$monitoroff, \$monitor 便不间断地对所设定的信号进行监视。

7.2 时间度量系统函数 \$time

在 Verilog HDL 中有两种类型的时间系统函数: \$time 和 \$realtime。用这两个时间系统函数可以得到当前的仿真时刻。

1. 系统函数 \$time

\$time 可以返回一个 64 位的整数来表示的当前仿真时刻值。该时刻是以模块的仿真时间尺度为基准的。下面举例说明:

[例 7.1]

```
'timescale 10 ns/1 ns
module test;
    reg set;
    parameter p=1.6;
    initial
        begin
            $monitor($time,, "set=", set);
            # p set=0;
            # p set=1;
        end
    endmodule
```

输出结果为

```
0 set=x
2 set=0
3 set=1
```

在这个例子中,模块 test 想在时间为 16 ns 时设置寄存器 set 为 0,在时间为 32 ns 时设置寄存器 set 为 1。但是由 \$time 记录的 set 变化时刻却和预想的不一样。这是由下面两个原因引起的:

(1) \$time 显示时刻受时间尺度比例的影响。在 [例 7.1] 中,时间尺度是 10 ns,因为 \$time 输出的时刻总是时间尺度的倍数,这样将 16 ns 和 32 ns 输出为 1.6 和 3.2。

(2) 因为 \$time 总是输出整数,所以,在将经过尺度比例变换的数字输出时,要先进行取整。在 [例 7.1] 中,1.6 和 3.2 经取整后为 2 和 3 输出。

注意:时间的精确度并不影响数字的取整。

2. \$realtime 系统函数

\$realtime 和 \$time 的作用是一样的,只是 \$realtime 返回的时间数字是一个实型数,该数字也是以时间尺度为基准的。下面举例说明:

```
[例 7.2] 请写出以下代码的输出结果。该代码中使用了 $realtime 任务。
timescale 10 ns/1 ns
module test;
    reg set;
    parameter p=1.55;
    initial begin
        $monitor($realtime,, "set=%f",set);
        #p set=0;
        #p set=1;
    end
endmodule
```

输出结果为：

```
0 set=x
1.6 set=0
3.2 set=1
```

从[例 7.2]可以看出，\$ realtime 将仿真时刻经过尺度变换以后即输出，无须进行取整操作。所以 \$ realtime 返回的时刻是实型数。

7.3 系统任务 \$ finish

格式：

```
$ finish;
$ finish(n);
```

系统任务 \$ finish 的作用是退出仿真器，返回主操作系统，也就是结束仿真过程。任务 \$ finish 可以带参数，根据参数的值输出不同的特征信息。如果不带参数，默认 \$ finish 的参数值为 1。下面给出了对于不同的参数值，系统输出的特征信息：

0 不输出任何信息；

1 输出当前仿真时刻和位置；

2 输出当前仿真时刻、位置和在仿真过程中所用 memory 及 CPU 时间的统计。

7.4 系统任务 \$ stop

格式：

```
$ stop;
$ stop(n);
```

\$ stop 任务的作用是把 EDA 工具(例如仿真器)置成暂停模式，在仿真环境下给出一个交互式的命令提示符，将控制权交给用户。这个任务可以带有参数表达式。根据参数值(0,1 或 2)的不同，输出不同的信息。参数值越大，输出的信息越多。

7.5 系统任务 \$ readmemb 和 \$ readmemh

在 Verilog HDL 程序中有两个系统任务 \$ readmemb 和 \$ readmemh，并用来从文件中读取数据到存储器中。这两个系统任务可以在仿真的任何时刻被执行使用，其使用格式共有以下 6 种：

- (1) \$ readmemb("<数据文件名>", <存储器名>);
- (2) \$ readmemb("<数据文件名>", <存储器名>, <起始地址>);
- (3) \$ readmemb("<数据文件名>", <存储器名>, <起始地址>, <结束地址>);
- (4) \$ readmemh("<数据文件名>", <存储器名>);
- (5) \$ readmemh("<数据文件名>", <存储器名>, <起始地址>);
- (6) \$ readmemh("<数据文件名>", <存储器名>, <起始地址>, <结束地址>);

在这两个系统任务中，被读取的数据文件的内容只能包含：空白位置(空格、换行、制表格(tab)和 form - feeds)，注释行(//形式的和/*...*/形式的都允许)、二进制或十六进制的数字。数字中不能包含位宽说明和格式说明，对于 \$ readmemb 系统任务，每个数字必须是二进制数字，对于 \$ readmemh 系统任务，每个数字必须是十六进制数字。数字中不定值 x 或 X，高阻值 z 或 Z，和下画线(_)的使用方法及代表的意义与一般 Verilog HDL 程序中的用法及意义是一样的。另外，数字必须用空白位置或注释行来分隔开。

在下面的讨论中，地址一词指对存储器(memory)建模的数组的寻址指针。当数据文件被读取时，每一个被读取的数字都被存放到地址连续的存储器单元中去。存储器单元的存放地址范围由系统任务声明语句中的起始地址和结束地址来说明，每个数据的存放地址在数据文件中进行说明。当地址出现在数据文件中，其格式为字符“@”后跟上十六进制数。如：

@hh...h

对于这个十六进制的地址数中，允许大写和小写的数字。在字符“@”和数字之间不允许存在空白位置。可以在数据文件里出现多个地址。当系统任务遇到一个地址说明时，系统任务将该地址后的数据存放到存储器中相应的地址单元中去。[例 7.3]说明了怎样初始化存储器。

[例 7.3] 初始化存储器。

```
module test;
    reg [7:0] memory [0:7]; //声明有 8 个 8 位的存储单元
    integer i;
    initial
    begin
        //读取存储器文件 init.dat 到存储器中的给定地址
        $ readmemb("init.dat", memory);
        //显示初始化后的存储器内容
        for(i=0; i < 8; i = i + 1)
            $ display("Memory [%d] = %b", i, memory[i]);
    end
endmodule
```

```
    end
```

```
endmodule
```

文件 init.dat 包含初始化数据。用@<地址>在数据文件中指定地址，地址以十六进制数说明。数据用空格符分隔。数据可以包含 x 或者 z。未初始化的位置默认值为 x。名为 init.dat 的样本文件内容如下所示。

```
@002
11111111 01010101
00000000 10101010

@006
1111zzzz 00001111
```

当仿真测试模块时，将得到下面的输出：

```
Memory [0] = xxxxxxxx
Memory [1] = xxxxxxxx
Memory [2] = 11111111
Memory [3] = 01010101
Memory [4] = 00000000
Memory [5] = 10101010
Memory [6] = 1111zzzz
Memory [7] = 00001111
```

对于上面 6 种系统任务格式，需补充说明以下 5 点：

(1) 如果系统任务声明语句中和数据文件里都没有进行地址说明，则默认的存放起始地址为该存储器定义语句中的起始地址。数据文件里的数据被连续存放到该存储器中，直到该存储器单元存满为止或数据文件里的数据存完。

(2) 如果系统任务中说明了存放的起始地址，没有说明存放的结束地址，则数据从起始地址开始存放，存放到该存储器定义语句中的结束地址为止。

(3) 如果在系统任务声明语句中，起始地址和结束地址都进行了说明，则数据文件里的数据按该起始地址开始存放到存储器单元中，直到该结束地址，而不考虑该存储器的定义语句中的起始地址和结束地址。

(4) 如果地址信息在系统任务和数据文件里都进行了说明，那么数据文件里的地址必须在系统任务中地址参数声明的范围之内。否则将提示错误信息，并且装载数据到存储器中的操作被中断。

(5) 如果数据文件里的数据个数和系统任务中起始地址及结束地址暗示的数据个数不同的话，也要提示错误信息。

下面举例说明：先定义一个有 256 个地址的字节存储器 mem：

```
reg[7:0] mem[1:256];
```

以下给出的系统任务以各自不同的方式装载数据到存储器 mem 中：

```
initial $ readmemh("mem.data",mem);
initial $ readmemh("mem.data",mem,16);
initial $ readmemh("mem.data",mem,128,1);
```

第一条语句在仿真时刻为 0 时,将装载数据到以地址是 1 的存储器单元为起始存放单元的存储器中去。第二条语句将装载数据到以单元地址是 16 的存储器单元为起始存放单元的存储器中去,一直到地址是 256 的单元为止。第三条语句将从地址是 128 的单元开始装载数据,一直到地址为 1 的单元。在第三种情况中,当装载完毕,系统要检查在数据文件里是否有 128 个数据,如果没有,系统将提示错误信息。

7.6 系统任务 \$ random

这个系统函数提供了一个产生随机数的手段。当函数被调用时返回一个 32 位的随机数。它是一个带符号的整形数。

\$ random 一般的用法是: \$ random % b ,其中 b>0。它给出了一个范围在($-b+1$):($b-1$)中的随机数。下面给出一个产生随机数的例子:

```
reg[23:0] rand;
rand = $ random % 60;
```

上面的例子给出了一个范围在 -59~59 之间的随机数,下面的例子通过位并接操作产生一个值在 0~59 之间的数。

```
reg[23:0] rand;
rand = { $ random } % 60;
```

利用这个系统函数可以产生随机脉冲序列或宽度随机的脉冲序列,以用于电路的测试。[例 7.4]中的 Verilog HDL 模块可以产生宽度随机的随机脉冲序列的测试信号源,在电路模块的设计仿真时非常有用。同学们可以根据测试的需要,模仿[例 7.4],灵活使用 \$ random 系统函数编制出与实际情况类似的随机脉冲序列。

[例 7.4]

```
'timescale 1ns/1ns
module random_pulse( dout );
    output [9:0] dout;
    reg [9:0] dout;
    integer delay1,delay2,k;
    initial begin
        #10 dout=0;
        for (k=0; k< 100; k=k+1)
            begin
                delay1 = 20 * ( { $ random } % 6 );
                // delay1 在 0~100 ns 间变化
                delay2 = 20 * ( 1 + { $ random } % 3 );
                #delay1
                if (k < 99) begin
                    #delay2
                    if (dout == 0) begin
                        dout = 1;
                    end
                    else begin
                        dout = 0;
                    end
                end
            end
    end
endmodule
```

```

// delay2 在 20~60 ns 间变化
# delay1  dout = 1 << ({ $random } %10);
//dout 的 0~9 位中随机出现 1, 并出现的时间在 0~100 ns 间变化
# delay2  dout = 0;
//脉冲的宽度在在 20~60 ns 间变化
end
end
endmodule

```

7.7 编译预处理

Verilog HDL 语言和 C 语言一样也提供了编译预处理的功能。“编译预处理”是 Verilog HDL 编译系统的一个组成部分。Verilog HDL 语言允许在程序中使用几种特殊的命令(它们不是一般的语句)。Verilog HDL 编译系统通常先对这些特殊的命令进行“预处理”,然后将预处理的结果和源程序一起在进行通常的编译处理。

在 Verilog HDL 语言中,为了和一般的语句相区别,这些预处理命令以符号“`”开头(位于主键盘左上角,其对应的上键盘字符为“~”。注意这个符号是不同于单引号“!”的)。这些预处理命令的有效作用范围为定义命令之后到本文件结束或到其他命令定义替代该命令之处。Verilog HDL 提供了以下预编译命令:

```

`accelerate, `autoexpand_vectors, `celldefine, `default_nettype, `define, `else, `endcell-
define, `endif, `endprotect, `endprotected, `expand_vectors, `ifdef, `include, `noaccelerate,
`noexpand_vectors, `noremove_gatenames, `noremove_netnames, `nounconnected_drive,
`protect, `protecte, `remove_gatenames, `remove_netnames, `reset, `timescale, `unconnected_
_drive

```

在这一小节里只对最常用的 `define、`include、`timescale 进行介绍,其余的可查阅参考 Verilog 硬件描述手册。

7.7.1 宏定义 `define

用一个指定的标识符(即名字)来代表一个字符串,它的一般形式为:

`define 标识符(宏名)字符串(宏内容)

如:

`define signal string

它的作用是指定用标识符 signal 来代替 string 这个字符串,在编译预处理时,把程序中在该命令以后所有的 signal 都替换成 string。这种方法使用户能以一个简单的名字代替一个长的字符串,也可以用一个有含义的名字来代替没有含义的数字和符号。因此,把这个标识符(名字)称为“宏名”,在编译预处理时将宏名替换成字符串的过程称为“宏展开”。`define 是宏定义命令。

[例 7.5]

```

`define WORDSIZE 8
module

```

```
reg[1:WORDSIZE] data; //相当于定义 reg[1:8] data;
```

关于宏定义的 8 点说明：

(1) 宏名可以用大写字母表示,也可以用小写字母表示。建议使用大写字母,以与变量名相区别。

(2) 'define 命令可以出现在模块定义里面,也可以出现在模块定义外面。宏名的有效范围为定义命令之后到原文件结束。通常,'define 命令写在模块定义的外面,作为程序的一部分,在此程序内有效。

(3) 在引用已定义的宏名时,必须在宏名的前面加上符号“\”,表示该名字是一个经过宏定义的名字。

(4) 使用宏名代替一个字符串,可以减少程序中重复书写某些字符串的工作量。而且记住一个宏名要比记住一个无规律的字符串容易,这样在读程序时能立即知道它的含义,当需要改变某一个变量时,可以只改变 'define 命令行,一改全改。如[例 7.5]中,先定义 WORDSIZE 代表常量 8,这时寄存器 data 是一个 8 位寄存器。如果需要改变寄存器的大小,只需把该命令行改为:'define WORDSIZE 16。这样寄存器 data 则变为一个 16 位的寄存器。由此可见使用宏定义,可以提高程序的可移植性和可读性。

(5) 宏定义是用宏名代替一个字符串,也就是做简单的置换,不做语法检查。预处理时照样代入,不管含义是否正确,只有在编译已被宏展开后的源程序时才报错。

(6) 宏定义不是 Verilog HDL 语句,不必在行末加分号。如果加了分号会连分号一起进行置换,见[例 7.6]。

[例 7.6]

```
module test;
    reg a, b, c, d, e, out;
    'define expression a+b+c+d;
    assign out = \expression + e;
    :
endmodule
```

经过宏展开以后,该语句为

```
assign out = a+b+c+d; + e;
```

显然出现语法错误。

(7) 在进行宏定义时,可以引用已定义的宏名,可以层层置换,见[例 7.7]。

[例 7.7]

```
module test;
    reg a, b, c;
    wire out;
    'define aa a + b
    'define cc c + 'aa
    assign out = \cc;
endmodule
```

这样经过宏展开以后, assign 语句为

```
assign out = c + a + b;
```

(8) 宏名和宏内容必须在同一行中进行声明。如果在宏内容中包含有注释行, 注释行不会作为被置换的内容, 见[例 7.8]。

[例 7.8]

```
module test;
    assign out = c + a + b;
endmodule
```

经过宏展开以后, 该语句为

```
nand #5 g121(q21,n10,n11);
```

宏内容可以是空格, 在这种情况下, 宏内容被定义为空的。当引用这个宏名时, 不会有内容被置换。

注意: 组成宏内容的字符串不能够被以下的语句记号分隔开的:

- 注释行;
- 数字;
- 字符串;
- 确认符;
- 关键词;
- 双目和三目字符运算符。

如下面的宏定义声明和引用是非法的:

```
'define first_half "start of string
$display ('first_half end of string');
```

在使用宏定义时要注意以下情况:

(1) 对于某些 EDA 软件, 在编写源程序时, 如使用和预处理命令名相同的宏名会发生冲突, 因此建议不要使用和预处理命令名相同的宏名。

(2) 宏名可以是普通的标识符(变量名)。例如 signal_name 和 `signal_name 的意义是不同的。但是这样容易引起混淆, 建议不要这样使用。

7.7.2 “文件包含”处理 `include

所谓“文件包含”处理是一个源文件可以将另外一个源文件的全部内容包含进来, 即将另外的文件包含到本文件之中。Verilog HDL 语言提供了 `include 命令用来实现“文件包含”的操作。其一般形式为:

```
`include"文件名"
```

图 7.1 表示“文件包含”的含意。图 7.1(a)为文件 File1.v, 它有一个 `include "File2.v" 命令, 然后还有其他的内容(以 A 表示)。图 7.1(b)为另一个文件 File2.v, 文件的内容以 B 表示。在编译预处理时, 要对 `include 命令进行“文件包含”预处理: 将 File2.v 的全部内容复制

插入到 `include "File2.v"` 命令出现的地方, 即 File2.v 被包含到 File1.v 中, 得到图 7.1(c) 所示的结果。在接着往下进行的编译中, 将“包含”以后的 File1.v 作为一个源文件单位进行编译。

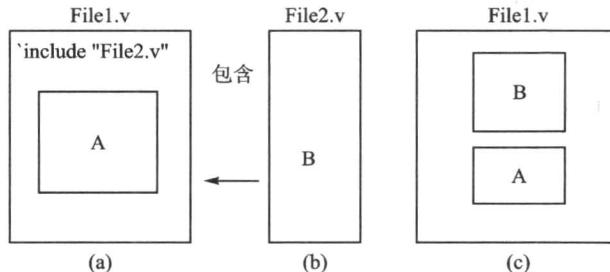


图 7.1 “文件包含”示意图

“文件包含”命令是很有用的, 可以节省程序设计人员的重复劳动; 可以将一些常用的宏定义命令或任务(task)组成一个文件, 然后用`include`命令将这些宏定义包含到自己所写的源文件中, 相当于工业上的标准元件拿来使用。另外, 在编写 Verilog HDL 源文件时, 一个源文件可能经常要用到另外几个源文件中的模块, 遇到这种情况即可用`include`命令将所需模块的源文件包含进来, 见[例 7.9]。

[例 7.9] 文件的包含。

(1) 文件 aaa.v

```

module aaa(a,b,out);
    input a, b;
    output out;
    wire out;
    assign out = a ^ b;
endmodule

```

(2) 文件 bbb.v

```

`include "aaa.v"
module bbb(c,d,e,out);
    input c,d,e;
    output out;
    wire out_a;
    wire out;
    aaa aaa(.a(c),.b(d),.out(out_a));
    assign out = e & out_a;
endmodule

```

在[例 7.9]中, 文件 bbb.v 用到了文件 aaa.v 中的模块 aaa 的实例器件, 通过“文件包含”处理来调用。模块 aaa 实际上是作为模块 bbb 的子模块来被调用的。在经过编译预处理后, 文件 bbb.v 实际相当于下面的程序文件 bbb.v:

```

module aaa(a,b,out);

```

```

input a, b;
output out;
wire out;

assign out = a ^ b;
endmodule

module bbb( c, d, e, out);
input c, d, e;
output out;
wire out_a;
wire out;

aaa aaa (.a(c), .b(d), .out(out_a));
assign out= e & out_a;
endmodule

```

关于“文件包含”处理的 5 点说明：

- (1) 一个`include 命令只能指定一个被包含的文件,如果要包含 n 个文件,要用 n 个`include 命令。注意下面的写法是非法的,如 `include"aaa.v" "bbb.v"。
- (2) `include 命令可以出现在 Verilog HDL 源程序的任何地方,被包含文件名可以是相对路径名,也可以是绝对路径名。例如:`include "parts/count.v"。
- (3) 可以将多个`include 命令写在一行,在`include 命令行,可以出现空格和注释行。例如下面的写法是合法的:

```
`include "fileB" `include "fileC" //including fileB and fileC
```

- (4) 如果文件 1 包含文件 2,而文件 2 要用到文件 3 的内容,则可以在文件 1 用两个`include 命令分别包含文件 2 和文件 3,而且文件 3 应出现在文件 2 之前。例如在下面的例子中,即在 file1.v 中定义:

```

`include"file3.v"
`include"file2.v"

module test(a,b,out);
input[1:'size2] a, b;
output[1:'size2] out;
wire[1:'size2] out;
assign out= a + b;
endmodule

```

file2.v 的内容为:

```
'define size2 `size1+1
```

file3.v 的内容为:

```
'define size1 4
```

:

这样, file1.v 和 file2.v 都可以用到 file3.v 的内容。在 file2.v 中不必再用 'include "file3.v" 了。

(5) 在一个被包含文件中又可以包含另一个被包含文件, 即文件包含是可以嵌套的。例如上面的问题也可以这样处理, 如图 7.2 所示。

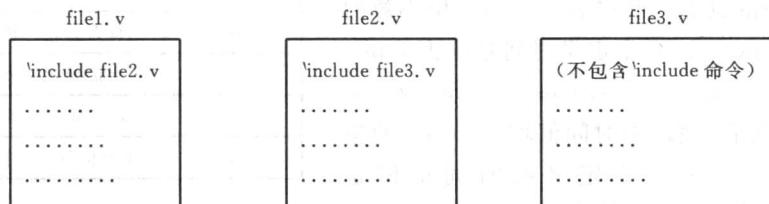


图 7.2 文件的嵌套包含(一)

它的作用和图 7.3 的作用是相同的。

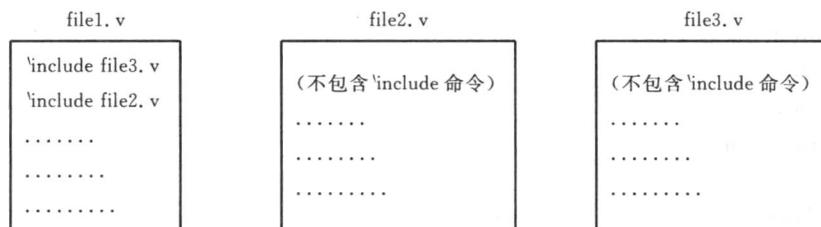


图 7.3 文件的嵌套包含(二)

许多 Verilog 编译器支持多模块编译, 也就是说只要把需要用 'include 包含的所有文件都放置在一个项目中, 建立存放编译结果的库, 用模块名就可以把所有有关的模块联系在一起, 此时在程序模块中就不必使用 'include 编译预处理指令。

7.7.3 时间尺度 'timescale

'timescale 命令用来说明跟在该命令后的模块的时间单位和时间精度。使用 'timescale 命令可以在同一个设计里包含采用了不同的时间单位的模块。例如, 一个设计中包含了两个模块, 其中一个模块的时间延迟单位为纳秒(ns), 另一个模块的时间延迟单位为皮秒(ps)。EDA 工具仍然可以对这个设计进行仿真测试。

'timescale 命令的格式如下:

`timescale<时间单位>/<时间精度>

在这条命令中, 时间单位参量是用来定义模块中仿真时间和延迟时间的基准单位的。时间精度参量是用来声明该模块的仿真时间的精确程度的, 该参量被用来对延迟时间值进行取整操作(仿真前), 因此该参量又可以被称为取整精度。如果在同一个程序设计里, 存在多个 'timescale 命令, 则用最小的时间精度值来决定仿真的时间单位。另外时间精度至少要和时间单位一样精确, 时间精度值不能大于时间单位值。

在 'timescale 命令中, 用于说明时间单位和时间精度参量值的数字必须是整数, 其有效数字为 1, 10, 100, 单位为秒(s)、毫秒(ms)、微秒(μs)、纳秒(ns)、皮秒(ps)、飞秒(fs)。这几种单位的意义说明如表 7.1 所列。

下面举例说明 `timescale 命令的用法。

[例 7.10] `timescale 1ns/1ps; 在这个命令之后,模块中所有的时间值都表示是 1 ns 的整数倍。这是因为在 `timescale 命令中,定义了时间单位是 1 ns。模块中的延迟时间可表达为带 3 位小数的实型数,因为 `timescale 命令定义时间精度为 1 ps。

[例 7.11] `timescale 10 μs/100 ns; 在 `timescale 命令定义后,模块中时间值均为 10 μs 的整数倍。因为 `timescale 命令定义的时间单位是 10 μs。延迟时间的最小分辨度为十分之一微秒(100 ns),即延迟时间可表达为带一位小数的实型数。

[例 7.12]

```
'timescale 10 ns/1 ns
module test;
reg set;
parameter d=1.55;
initial
begin
#d set=0;
#d set=1;
end
endmodule
```

在这个例子中,`timescale 命令定义了模块 test 的时间单位为 10 ns、时间精度为 1 ns。因此,在模块 test 中,所有的时间值应为 10 ns 的整数倍,且以 1 ns 为时间精度。这样经过取整操作,存在参数 d 中的延迟时间实际是 16 ns(即 1.6×10 ns)。这意味着在仿真时刻为 16 ns 时寄存器 set 被赋值 0;在仿真时刻为 32 ns 时寄存器 set 被赋值 1。仿真时刻值是按照以下的步骤来计算的。

- (1) 根据时间精度,参数 d 值被从 1.55 取整为 1.6。
- (2) 因为时间单位是 10 ns,时间精度是 1 ns,所以延迟时间 #d 作为时间单位的整数倍为 16 ns。

(3) EDA 工具预定在仿真时刻为 16 ns 的时候给寄存器 set 赋值 0(即语句 #d set=0; 执行时刻),在仿真时刻为 32 ns 的时候给寄存器 set 赋值 1(即语句 #d set=1; 执行时刻)。

注意:如果在同一个设计里,多个模块中用到的时间单位不同,需要用到以下的时间结构:

- (1) 用 `timescale 命令来声明本模块中所用到的时间单位和时间精度。
- (2) 用系统任务 \$ printtimescale 来输出显示一个模块的时间单位和时间精度。
- (3) 用系统函数 \$ time 和 \$ realtime 及 %t 格式声明来输出显示 EDA 工具记录的时间信息。

7.7.4 条件编译命令 `ifdef、`else、`endif

一般情况下,Verilog HDL 源程序中所有的行都将参加编译。但是有时希望对其中的一

表 7.1 常用时间单位与定义之对应

| 时间单位 | 定 义 |
|------|------------------------|
| s | 秒(1 s) |
| ms | 千分之一秒(10^{-3} s) |
| μs | 百万分之一秒(10^{-6} s) |
| ns | 十亿分之一秒(10^{-9} s) |
| ps | 万亿分之一秒(10^{-12} s) |
| fs | 千万亿分之一秒(10^{-15} s) |

部分内容只有在满足条件时才进行编译,也就是对一部分内容指定编译的条件,这就是“条件编译”。有时,希望当满足条件时对一组语句进行编译,而当条件不满足时则编译另一部分。

条件编译命令有以下几种形式:

(1) `ifdef 宏名 (标识符)

程序段 1

`else

程序段 2

`endif

它的作用是当宏名已经被定义过(用`define 命令定义),则对程序段 1 进行编译,程序段 2 将被忽略;否则编译程序段 2,程序段 1 被忽略。其中`else 部分可以没有,即:

(2) `ifdef 宏名 (标识符)

程序段 1

`endif

这里的“宏名”是一个 Verilog HDL 的标识符,“程序段”可以是 Verilog HDL 语句组,也可以是命令行。这些命令可以出现在源程序的任何地方。

注意:被忽略掉不进行编译的程序段部分也要符合 Verilog HDL 程序的语法规则。

通常在 Verilog HDL 程序中用到`ifdef、`else、`endif 编译命令的情况有以下几种:

(1) 选择一个模块的不同代表部分。

(2) 选择不同的时序或结构信息。

(3) 对不同的 EDA 工具,选择不同的激励。

最常用的情况是:Verilog 代码中的一部分可能适用于某个编译环境,但不适用于另一个环境。如设计者不想为两个环境创建两个不同版本的 Verilog 设计,还有一种方法就是所谓的条件编译,即设计者在代码中指定其中某一部分只有在设置了特定的标志后,这一段代码才能被编译。

设计者也可能希望在程序的运行中,只有当设置了某个标志后,才能执行 Verilog 设计的某些部分,这就是所谓的条件执行。

条件编译可以用编译指令`ifdef, `ifndef, `else, `elsif 和`endif 实现。[例 7.13]中包含一段能进行条件编译的 Verilog 源代码。

[例 7.13] 条件编译。

```
//条件编译
//例 7.13.1
`ifdef TEST          //若设置 TEST 标志,则编译 test 模块
module test;
    initial
        $display("Module %m compiled");
endmodule
`else                //在默认情况下,则编译 stimulus 模块
module stimulus;
    initial
        $display("Module %m compiled");
```

```

    endmodule // 'endif 语句的结束

// [例 7.13. 2]
module top;

bus_master b1(); //无条件地调用模块
`ifndef ADD_B2
    bus_master b2(); //若定义了 ADD_B2 文本宏标志,则有条件地调用 b2
`ifndef ADD_B3
    bus_master b3(); //若定义 ADD_B3 文本宏标志,则有条件地调用 b3
`else
    bus_master b4(); //在默认情况下,则有条件地调用 b4
`endif
`ifndef IGNORE_B5
    bus_master b5(); //若没有定义 IGNORE_B5 文本宏标志,则有条件地调用 b5
endmodule

```

'ifdef 和 `ifndef 指令可以出现在设计的任何地方。设计者可以有条件地编译语句、模块、语句块、声明和其他编译指令。`else 指令是可选的。一个 `else 指令最多可以匹配一个 `ifdef 或者 `ifndef。一个 `ifdef 或者 `ifndef 可以匹配任意数量的 `elsif 命令。`ifdef 或 `ifndef 总是用相应的 `endif 来结束。

Verilog 文件中,条件编译标志可以用 `define 语句设置。在上例中,可以通过在编译时用 `define 语句定义文本宏 TEST 和 ADD_B2 的方式定义标志。如果没有设置条件编译标志,那么 Verilog 编译器会简单地跳过该部分。`ifdef 语句中不允许使用布尔表达式,例如使用 TEST && ADD_B2 来表示编译条件是不允许的。

7.7.5 条件执行

条件执行标志允许设计者在运行时控制语句执行的流程。所有语句都被编译,但是有条件地执行它们。条件执行标志仅能用于行为语句,系统任务关键字 \$ test \$ plusargs 用于条件执行。

参阅[例 7.14]例子,用 \$ test \$ plusargs 描述条件执行。

[例 7.14] 带 \$ test \$ plusargs 的条件执行。

```

//条件执行
module test;
reg a, b, c;
initial
begin
    a = 1'b1; b = 1'b0; c = 1'b1;
    if ($ test$ plusargs ("DISPLAY_VAR"))

```

```

$ display("Display = %b ", {a,b,c}); //只有当标志设置时才能显示
else
    $ display ("No Display");           //其他情况下不显示
end
endmodule

```

仅当在运行时设置了标志 DISPLAY_VAR 时才显示变量。可以指定 +DISPLAY_VAR 选项在程序运行时设置标志。

可以使用系统任务关键字 \$ value \$ plusargs 来进一步控制条件执行。该系统任务用于测试调用选项的参数值。如果没有找到匹配的调用选项，那么 \$ value \$ plusargs 返回 0；如果找到匹配的选项，那么 \$ value \$ plusargs 返回非 0 值。[例 7.15] 给出了 \$ value \$ plusargs 的示例。

[例 7.15] 带 \$ value \$ plusargs 的条件执行。

```

//用系统任务 $ value $ plusargs 的条件执行
module test ;
reg [8 * 128 - 1 : 0 ] test_string ;
integer clk_period ;
...
...
initial
begin
if ( $ value $ plusargs(" test name = %s", test_string))
    $ readmemh (test_string, vectors) ; //读取测试向量
else
    //否则显示错误信息
    $ display (" Test name option not specified") ;
if ( $ value $ plusargs(" clk_t = %d", clk_period))
    forever # (clk_period/2) clk = ~clk ; //设置时钟
else
    //否则显示错误信息
    $ display (" Clock period option name not specified") ;
end
//例如要启动上述选项,需要使用带 +testname=test1.vec +clk_t = 10
// Test name = "test1.vec" 和 clk_period = 10 的命令行来启动仿真器
endmodule

```

小结

在本章中学习了 Verilog 语法中几种最常用的调试和查错系统任务以及编写实用模块时常用的编译预处理语句。这些语句是非常有用的，可以说是调试模块所必须的。只有掌握它

们才能够设计有用的逻辑电路系统。需要注意的有以下几点：

- (1) 在多模块调试的情况下，\$monitor 需配合 \$monitoron 与 \$monitoroff 使用。
- (2) \$monitor 与 \$display 的不同处在于 \$monitor 是连续监视数据的变化，因而往往只要在测试模块的 initial 块中调用一次就可以监视被测试模块的所有感兴趣的信号，不需要、也不能在 always 过程块中调用 \$monitor。
- (3) \$time 常用在 \$monitor 中，用来做时间标记。
- (4) \$stop 和 \$finish 常用在测试模块的 initial 模块中，配合时间延迟来控制仿真的持续时间。
- (5) \$random 在编写测试程序是非常有用的，可以用来产生边沿不稳定的波形，和随机出现的脉冲。正确地使用它能有效地发现实际设计中存在的问题。
- (6) \$readmem 在编写测试程序也是非常有用的，可以用来生成给定的复杂数据流。复杂数据可以用 C 语言产生，存在文件中。用 \$readmem 取出存入存储器，再按节拍输出，这在验证算法逻辑电路时特别有用。
- (7) 在用 `timescale 时需要注意的是，当多个带不同 `timescale 定义的模块包含在一起时只有最后一个才起作用。所以属于一个项目，但 `timescale 定义不同的多个模块最好分开编译，不要包含在一起编译，以免把时间单位搞混。
- (8) 宏定义字符串引用时，不要忘记要用 “`” 引导。这与 C 语言不同，C 语言直接引用就行，但 Verilog 必须用 “`” 引导。
- (9) include 等编译预处理也必须用 “`” 引导，而不是与 C 语言一样用 “#” 引导或不需要引导符。
- (10) 合理地使用条件编译和条件执行预处理可以使测试程序适应不同的编译环境，也可以把不同的测试过程编写到一个统一的测试程序中去，可以简化测试的过程，对于复杂设计的验证模块的编写很有实用价值。

在学习中要注意这些语句含义、使用的场合和在程序模块中的位置，有意识地把这些系统任务与测试模块的编写联系起来。只有深入理解了有关语法的实质，才能在设计中准确地应用。

思 考 题

1. 为什么在多模块调试的情况下 \$monitor 需要配合 \$monitoron 和 \$monitoroff 来工作？
2. 请用 \$random 配合求模运算编写：
 - (1) 用于测试的跳变沿抖动为周期 1/10 的时钟波形。
 - (2) 随机出现的脉宽随机的窄脉冲。
3. Verilog 的编译预处理与 C 语言的编译预处理有什么不同？
4. 请仔细阐述 `timescale 编译预处理的作用？
5. 不同 `timescale 定义的多模块仿真测试时需要注意什么？
6. 为什么说系统任务 \$readmem 可以用来产生用于算法验证的极其复杂的测试用数据流？
7. 为什么说熟练地使用条件编译命令可以使源代码有更大的灵活性，可以适用于不同的实现对象，如不同工艺的 ASIC 或速度规模不同的 FPGA 或 CPLD，从而为软核的商品化创造条件？

第8章 语 法 概 念 总 复 习 练 习

概 述

在本章中希望大家通过独立完成 28 个练习题,把前面 7 章中提到的 Verilog 基本语法复习巩固一下。掌握语法只靠阅读理解是远远不够的,必须通过大量的练习才能掌握。从做题出现的错误中,发现自己理解的不足,从而得到正确的概念。本章中有些题可以帮助理解模块的构造;有些题解释了端口和矢量定义的含义;有些题可知道什么地方应该用什么变量类型;有些题是为了帮助理解两种不同的赋值操作而专门设计的;有些题是帮助理解 case 和 casex 有什么不同;……。总而言之这些题都是经过精心考虑的,希望认真地独立思考,分析为什么标准答案是这样的。如果你的答案都与标准答案一致,说明你已经掌握了基本语法的要点。这对于以后的学习是非常重要的。下面就是自我测试题。

(1) 图 8.1 为一个填空练习,将所给各个选项根据以下电路图,填入程序中的适当位置。

```
assign module; ~| &. input output  
inputs outputs endmodule  
A,B,C,D  
AOI (A,B,C,D,F)
```

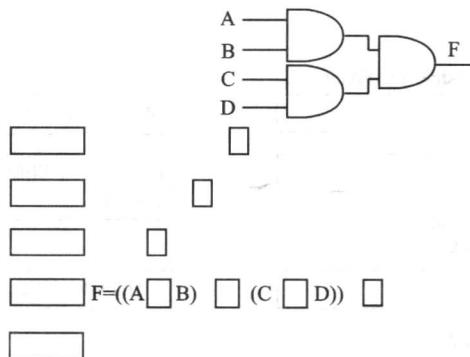


图 8.1 练习题电路图

标准答案:

```
module AOI(A,B,C,D,F);  
input A,B,C,D;  
output F;  
assign F = ((A&B)&(C&D));  
endmodule
```

(2) 在这一题中,将做有关层次电路的练习(见图 8.2)。通过这个练习,将加深对模块间

调用时,引脚间连接的理解。假设已有全加器模块 FullAdder,若有一个顶层模块调用此全加器,连接线分别为 W4,W5,W3,W2 和 W1。请在调用时正确地填入 I/O 的对应信号。

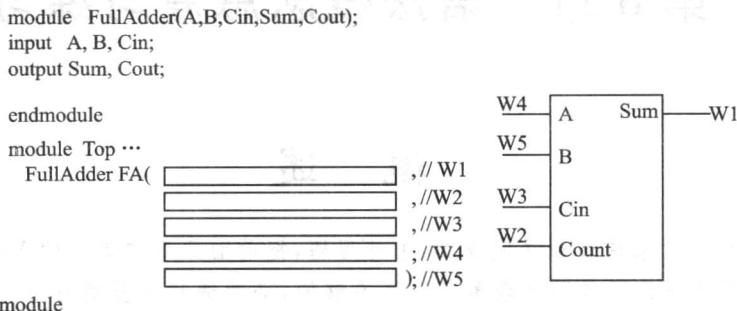


图 8.2 层次电路的练习

标准答案:

```

module Top ...
  FullAdderFA(.Sum(W1), //W1
               .Cout(W2), //W2
               .Cin(W3), //W3
               .A(W4), //W4
               .B(W5));
endmodule

```

(3) 本题是一个测试模块,没有输入、输出端口,请将相应项填入合适的位置,如图 8.3 所示。

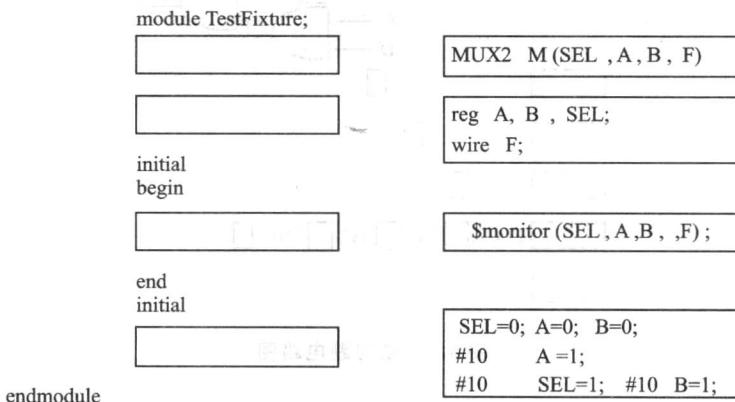


图 8.3 测试模块

标准答案:

```

module TestFixture;
  reg A, B, SEL;
  wire F;
  MUX2M(SEL, A, B, F);
  initial

```

```

begin
    SEL=0; A=0; B=0;
    #10 A=1;
    #10 SEL=1; #10 B=1;
end
initial
    $ monitor(SEL,A,B,,F);
endmodule

```

(4) 指出下面几个信号的最高位和最低位。

reg [1:0] SEL; input [0:2] IP; wire [16:23] A;

标准答案：

MSB:SEL[1] MSB:IP[0] MSB:A[16]

LSB:SEL[0] LSB:IP[2] LSB:A[23]

(5) P,Q,R 都是 4bit 的输入矢量,下面哪一种表达形式是正确的。

1) input P[3:0],Q,R;

2) input P,Q,R[3:0];

3) input P[3:0],Q[3:0],R[3:0];

4) input [3:0] P,[3:0]Q,[0:3]R;

5) input [3:0] P,Q,R。

标准答案:5)。

(6) 请将下面选项中的正确答案填入空的方括号中:

1) (0:2) 2) (P;0) 3) (Op1:Op2) 4) (7:7) 5) (2:0) 6) (7:0)

reg [7:0] A;

reg [2:0] Sum, Op1, Op2;

reg P, OneBit;

initial

begin

Sum=Op1+Op2;

P=1;

A[]=Sum;

:

end

标准答案:5)。

(7) 请根据以下两条语句,从选项中找出正确答案。

1) reg [7:0] A;

A=2'hFF;

① 8'b0000_0011 ② 8'h03 ③ 8'b1111_1111 ④ 8'b11111111

标准答案:①和②。

2) reg [7:0] B;

B=8'bZ0;
 ① 8'0000_00Z0 ② 8'bZZZZ_0000 ③ 8'b0000_ZZZ0 ④ 8'bZZZZ_ZZZ0

标准答案:④。

(8) 请指出下面几条语句中变量的类型。

1) assign A=B;

2) always #1

 Count=C+1;

标准答案:

A(wire) B(wire/reg) Count(reg) C(wire/reg)

(9) 指出下面模块中 Cin,Cout,C3,C5 的类型。

```
module FADD(A,B,Cin,Sum,Cout);
    input A, B, Cin;
    output Sum, Cout;
    :
endmodule
module Test;
    :
FADD(C1,C2,C3,C4,C5);
    :
endmodule
```

标准答案:

Cin(wire) Cout(wire/reg) C3(wire/reg) C5(wire)

(10) 在下一个程序段中,当 ADDRESS 的值等于 5'b0X000 时,问 casex 执行完后 A 和 B 的值是多少。

```
A=0;
B=0;
casex(ADDRESS)
5'b00???: A=1;
5'b01???: B=1;
5'b10? 00, 5'b11? 00:
begin
    A=1;
    B=1;
end
endcase
```

标准答案: A=1 and B=0

(11) 事件 A 分别在 10,20,30 发生,而 B 一直保持 X 状态,问在 50 时 Count 的值是多少。

reg [7:0] Count;

```

initial
  Count=0;
always
begin
  @(A) Count=Count+1;
  @(B) Count=Count+1;
end

```

标准答案:Count=1。这是因为当 A 第一次发生时,Count 的值由 0 变为 1,然后事件控制@(B) 阻挡了进程。

(12) 在下列程序中 initial 块执行完后,I,J,A,B 的值会是多少?

```

reg [2:0] A;
reg [3:0] B;
integer I, J;
initial
begin
  I=0;
  A=0;
  I=I-1;
  J=I;
  A=A-1;
  B=A;
  J=J+1;
  B=B+1;
end

```

标准答案:

| | |
|------|--|
| I=-1 | (整数可为负数) |
| J=0 | |
| A=7 | (A 为 reg 型,为非负数,又因为 A 为 3 位即为 111) |
| B=8 | (在 B=A 时,B=0111,然后 B=B+1,所以 B=4'b1000) |

(13) 在下列程序中,当 V 的值发生变化且为 -1 时,执行完 always 块后 Count 的值应是多少?

```

reg[7:0]V;
reg[2:0]Count;

always @(V)
begin
  Count=0;
  while(~V[Count])
    Count=Count+1;
end

```

标准答案:Count=0;

(14) 在下列程序中,循环执行完后,V 的值是多少?

```
reg [3:0] A;
reg V, W;
integer K;
:
A=4'b1010;
for(K=2;K>=0;K=K-1)
begin
    V=V^A[k];
    W=A[K]^A[K+1];
end
```

标准答案:V 的值是它进入循环体前值的取反。因为 V 的值与 0,1,0 进行了异或,与 1 的异或改变了 V 的值。

(15) 在下列程序中,给出了几种硬件实现,问以下的模块被综合后可能是哪一种?

```
always @(posedge Clock)
if(A)
    C=B;
```

- 1) 不能综合。
- 2) 一个上升沿触发器和一个多路器。
- 3) 一个输入是 A,B,Clock 的三输入与门。
- 4) 一个透明锁存器。
- 5) 一个带 clock 有始能引脚的上升沿触发器。

标准答案:2) 和 5)。

(16) 在下列程序中,always 状态将描述一个带异步 Nreset 和 Nset 输入端的上升沿触发器,则空括号内应填入什么,可从以下 5 种答案中选择。

```
always @()
if(!Nreset)
    Q<=0;
else if(!Nset)
    Q<=1;
else
    Q<=D;
```

- 1) negedge Nset or posedge Clock
- 2) posedge Clock
- 3) negedge Nreset or posedge Clock
- 4) negedge Nreset or negedge Nset or posedge Clock
- 5) negedge Nreset or negedge Nset

标准答案:4)。

(17) 下面给出了几种硬件实现,问以下的模块被综合后可能是哪一种?

- 1) 带异步复位端的触发器;
- 2) 不能综合或与预先设想的不一致;
- 3) 组合逻辑;
- 4) 带逻辑的透明锁存器;
- 5) 带同步复位端的触发器。

① always @ (posedge Clock)

```
begin
    A <= B;
    if(C)
        A <= 1'b0;
end
```

标准答案:5)。

② always @ (A or B)

```
case(A)
    1'b0: F = B;
    1'b1: G = B;
endcase
```

标准答案:2)。

③ always @ (posedge A or posedge B)

```
if(A)
    C <= 1'b0;
else
    C <= D;
```

标准答案:1)。

④ always @ (posedge Clk or negedge Rst)

```
if(Rst)
    A <= 1'b0;
else
    A <= B;
```

标准答案:2),产生了异步逻辑。

(18) 在下列程序中,模块被综合后将产生几个触发器?

```
always @ (posedge Clk)
begin: Blk
    reg B, C;
    C = B;
    D <= C;
    B = A;
```

```
end
```

- 1) 2个寄存器 B 和 D
- 2) 2个寄存器 B 和 C
- 3) 3个寄存器 B, C 和 D
- 4) 1个寄存器 D
- 5) 2个寄存器 C 和 D

标准答案:2)。

- (19) 在下列程序中,各条语句的顺序是错误的,请根据图 8.4 所示电路图调整好它们的顺序。

```
Output=FF3
reg FF1, FF2, FF3;
    FF2=FF1;
always @ (posedge Clock)
end
    FF3=FF2;
begin
```

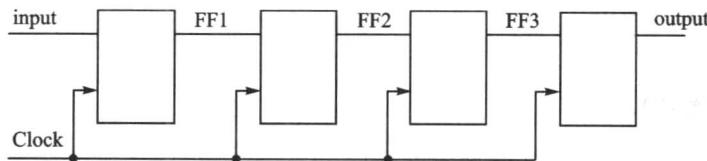


图 8.4 4 位移位寄存器电路

标准答案: 1),2)方框内。

| | |
|--|--|
| 1) <pre>reg FF1, FF2, FF3; always @(posedge Clock) begin FF1 <= Input; FF2 <= FF1; FF3 <= FF2; Output <= FF3; end</pre> | 2) <pre>reg FF1, FF2, FF3; always @(posedge Clock) begin Output = FF3; FF3 = FF2; FF2 = FF1; FF1 = Input; end</pre> |
|--|--|

- (20) 根据 SEL 列与 OP 列的对应关系,在模块的空括号中填入相应的值。

```
SEL:OP
000:1
001:3      casex(SEL)
010:1      3'b( ): OP=3;
011:3      3'b( ): OP=1;
100:0      3'b( ): OP=0;
```

```
101;3      endcase
110;0
111;3
```

标准答案：

```
casex(SEL)
3'bXX1:    OP=3;
3'b0X0:    OP=1;
3'b1X0:    OP=0;
endcase
```

(21) 在以下表达式中选出正确的。

- 1) $4'b1010 \& 4'b1101 = 1'b1$
- 2) $\sim 4'b1100 = 1'b1$
- 3) $! 4'b1011 || ! 4'b0000 = 1'b1$
- 4) $\& 4'b1101 = 1'b1$
- 5) $1b'0 || 1b'1 = 1'b1$
- 6) $4'b1011 \&& 4'b0100 = 4'b1111$
- 7) $4'b0101 << 1 = 5'b01011$
- 8) $! 4'b0010 \text{ is } 1'b0$
- 9) $4'b0001 || 4'b0000 = 1'b1$

标准答案：3), 5), 8) 和 9)。

(22) 在下列程序的括号中填入 display 的正确值。

```
integerI;
reg[3:0] A;
reg[7:0] B;
initial
begin
I=-1; A=I; B=A;
$display("%b",B);(      )
A=A/2;
$display("%b",A);(      )
B=A+14
$diaplay("%d",B);(      )
A=A+14;
$display("%d",A);(      )
A=-2;I=A/2;
$display("%d",I);(      )
end
```

标准答案：

```
I=-1;A=I;B=A;
```

```

$ display("%b", B);(00000111)
A=A/2;
$ display("%b", A);(0111)
B=A+14
$ display("%d", B);(21)
A=A+14;
$ display("%d", A);(5)           //A为4位,所以21被截为5
A=-2;I=A/2;
$ display("%d", I);(7)           //A=-2,则是1110

```

(23) 请问{1,0}与下面哪一个值相等。

- 1) 2'b01
- 2) 2'b10
- 3) 2'b00
- 4) 64'H00000000000002
- 5) 64'H0000000100000000

标准答案:5)。位拼接运算符必须指明位数,若不指明则隐含着为32位的二进制数[即整数]。

(24) 根据下题给出的程序,确定应将哪一个选项填入尖括号内。如图8.5所示为选项填空练习。

- 1) defs.Reset
- 2) "defs.v".Reset
- 3) M.Reset
- 4) Reset

| |
|---|
| <pre> module defs; parameter Reset=8'b10100101; endmodule (file defs.v) </pre> |
| <pre> module M; : if (OP==<>) Bus=0; (file M.v) endmodule </pre> |

图8.5 选项填空练习

① 标准答案:1)。模块间调用时,若引用其他模块定义的参数,要加上其他模块名,作为这个参数的前缀。

```

module M
'include "defs.v"
:
if(OP==<defs.Reset>)
Bus=0;
endmodule

```

② 标准答案:4)

```

parameter Reset=8'b10100101; (File defs.v)
module M
'include "defs.v"
:

```

```
if(OP==<Reset>)
Bus=0;
endmodule
```

(25) 如果调用 Pipe 时,想把 Depth 的值变为 8,问程序中的空括号内应填入何值?

```
Module Pipe(IP,OP)
parameter Option=1;
parameter Depth=1;
:
endmodule
Pipe( ) P1(IP1,OP1);
```

标准答案:#(1,8)。其中 1 对应参数 Option,8 对应参数 Depth。

(26) 若想使 P1 中的 Depth 的值变为 16,则应向空括号中填入哪个选项?

```
module Pipe (IP ,OP);
parameter Option =1;
parameter Depth = 1;
:
endmodule

module
Pipe P1(IP1 ,OP1);
( );
endmodule
```

- 1) defparam P1.Depth=16;
- 2) parameter P1.Depth=16;
- 3) parameter Pipe.Depth=16;
- 4) defparam Pipe.Depth=16;

标准答案:1)。用后缀改变引用模块的参数要用 defparam 及用本模块名作为引用参数的前缀,如 p1.Depth。

(27) 如果想在 Test 的 monitor 语句中观察 Count 的值,则在空括号中应填入什么?

```
Module Test
Top T();
initial
$monitor( )
endmodule
```

```
module Top;
Block B1();
Block B2();
endmodule
```

```

module Block;
Counter C();
endmodule

module Counter;
reg [3:0] Count;
:
endmodule

```

标准答案:T.B1.C.Countor,T.B2.C.Count。

(28) 图 8.6 所示方框中用 initial 块给 reg[7:0]V 赋值,请指明每种情况下 V 的 8 位数都是什么值?

该题说明在数的表示时,已标明字宽的数若用 XZ 表示某些位,只有在最左边的 X 或 Z 具有扩展性。

```

reg [7:0]V
initial
begin
    V=8'b0;
    V=8'b1;
    V=8'bX;
    V=8'BZX;
    V=B'BXXZZ;
    V=8'b1x;
end

```

标准答案:

```

8'b00000000
8'b00000001
8'bXXXXXXXX
8'bZZZZZZZX
8'BXXXXXXZ
8'b0000001X

```

图 8.6 initial 块赋值及答案

小结

通过以上 28 个练习,若能不看标准答案做对 90% 以上的题,说明你在阅读前面 7 章时是非常认真的,你的记忆和理解能力也是相当好的。若只能做对 50% 或更少,也没有关系。只要通过阅读前 7 章,能明白为什么标准答案是对的就可以了。以后还要多做几遍练习,尽力做到不用查阅前 7 章就能正确地知道标准答案。能做到这样,就说明你已经掌握了基本语法,就可以阅读《Verilog 数字系统设计教程》第 9 章以后的例题,然后模仿,逐步开始编写自己的 Verilog 模块。按照本书为你所设计的步骤,很快能设计出一些非常实用的数字逻辑电路。

第二部分 设计和验证部分

在前 8 章里我们学习了 Verilog 硬件描述语言的发展历史、主要用途、基本概念和基本语法。在本部分(第 9 章~第 18 章)里将通过许多简单和容易理解的例子分成 10 章由浅入深地讲解：

- (1) 不同抽象级别的 Verilog 模型及其作用；
- (2) 如何编写和验证简单的纯组合逻辑模块；
- (3) 如何编写和验证简单的时序逻辑模块；
- (4) 可综合模块的标准风格和注意事项；
- (5) 如何对简单电路模块进行功能的全面测试；
- (6) 复杂的数字系统是如何构成的；
- (7) 怎样根据系统需求,把组合逻辑和时序逻辑配合起来设计复杂的数字系统模块；
- (8) 怎样完整地验证所做的设计,以保证设计的正确性。

在阅读课本的基础上,可以通过在计算机上自己动手做一遍课本上的实验练习示例,再结合思考题进行改进设计,并验证改进后的设计是否达到了要求,来达到学习的目的。只有通过艰苦的练习才能够掌握设计的诀窍。

第 9 章 Verilog HDL 模型的不同抽象级别

概 述

由第一部分中可知,Verilog 模型可以是实际电路中不同级别的抽象。所谓不同的抽象级别,实际上是指同一个物理电路,可以在不同的层次上用 Verilog 语言来描述它。如果只从行为和功能的角度来描述某一电路模块,就称为行为模块;如果从电路结构的角度来描述该电路模块,就称为结构模块。抽象的级别和它们对应的模块类型常可以分为以下 5 种,即 Verilog 语法支持数字电路系统的 5 种不同描述方法:

- (1) 系统级(system);
- (2) 算法级(algorithmic);
- (3) RTL 级(Register-Transfer-Level);
- (4) 门级(gate-level);
- (5) 开关级(switch-level)。

系统级、算法级和 RTL 级是属于行为级的,门级是属于结构级的。在本章的各节中,将通过许多实际的 Verilog HDL 模块的设计来了解不同抽象级别模块的可综合性的问题。对于数字系统的逻辑设计工程师而言,熟练地掌握门级、RTL 级、算法级、系统级是非常重要的。而对于电路基本部件(如与或非门、缓冲器、驱动器等)库的设计者而言,则需要掌握用户自定义源语元件(UDP)和开关级的描述。在本设计教程的第二部分由于篇幅有限,只对 UDP 做了简单的介绍。有关 UDP 的编写要点将在高级教程里做较深入的讲解。由于开关级的描述涉及模拟电路的许多知识,在本教材中略去了这方面的内容。

一个复杂电路的完整 Verilog HDL 模型是由若干个 Verilog HDL 模块构成的,每一个模块又可以由若干个子模块构成。这些模块可以分别用不同抽象级别的 Verilog HDL 模块描述,在一个模块中也可以有多种级别的描述。利用 Verilog HDL 语言提供的这种结构,就能以这种清晰层次结构来描述极其复杂的大型设计。

9.1 门级结构描述

一个逻辑电路是由许多逻辑门和开关所组成,因此用基本逻辑门的模型来描述逻辑电路结构是最直观的。Verilog HDL 提供了一些描述门类型的关键字,可以用于门级结构建模。

9.1.1 与非门、或门和反向器及其说明语法

Verilog HDL 中有关门类型的关键字共有 26 之个,在本教材中只介绍最基本的 8 个。有关其他的门类型关键字,读者可以通过翻阅:Verilog 语言参考手册,在设计的实践中逐步掌握。下面列出了 8 个基本的门类型(GATETYPE)关键字和它们所表示门的类型:

and——与门；

nand——与非门；

nor——或非门；

or——或门；

xor——异或门；

xnor——异或非门；

buf——缓冲器；

not——非门。

门与开关的说明语法可以用标准的声明语句格式和一个简单的实例引用加以说明。门声明语句的格式如下：

〈门的类型〉[〈驱动能力〉〈延时〉]〈门实例 1〉[,〈门实例 2〉,...,〈门实例 n〉];

门的类型是门声明语句所必须的，它可以是 Verilog HDL 语法规定的 26 种门类型中的任意一种。驱动能力和延时是可选项，可根据不同的情况选不同的值或不选。门实例 1 是在本模块中引用的第一个这种类型的门，而门实例 n 是引用的第 n 个这种类型的门。有关驱动能力的选项在以后的章节里再详细加以介绍。下面用一个具体的例子来说明门类型的引用：

```
nand #10 nd1(a,data,clock,clear);
```

该例说明在模块中使用了一个名为 nd1 的与非门(nand)，输入为 data、clock 和 clear，输出为 a，输出与输入的延时为 10 个单位时间。

Verilog 语法支持的基本逻辑部件其行为是由该基本逻辑部件的原语(primitive)提供的，原语部件和本章中 9.3 节介绍的 UDP 没有本质上的差别。Verilog 编译或解释器能正确地处理有关原语部件的语法现象。

9.1.2 用门级结构描述 D 触发器

[例 9.1]是用 Verilog HDL 语言描述的 D 型主从触发器模块，通过这个例子，可以学习门级结构建模的基本方法。

【例 9.1】 用基本逻辑单元组成触发器(文件名为 flop.v)。

```
module      flop(data,clock,clear,q,qb);
    input       data,clock,clear;
    output      q,qb;

    nand #10  nd1(a,data,clock,clear),           //注意结束时用逗号,最后才用分号
              nd2(b,ndata,clock),                 //表示 nd1~nd8 都是 nand(与非门)
              nd4(d,c,b,clear),
              nd5(e,c,nclock),
              nd6(f,d,nclock),
              nd8(qb,q,f,clear);

    nand #9   nd3(c,a,d),
              nd7(q,e,qb);

    not #10   iv1(ndata,data),
              iv2(nclock,clock);
```

```
endmodule
```

在这个 Verilog HDL 结构描述的模块中, flop 定义了模块名, 设计上层模块时可以用模块名(flop)调用这个模块; module, input, output, endmodule 等都是关键字; nand 表示与非门。在 Verilog 语法中有这两个基本逻辑单元的行为, 它们由原语(primitive)描述, 这是一种类似真值表的描述(见 9.3 节 用户定义的原语 UDP); #10 表示 10 个单位时间的延时; nd1, nd2, ..., nd8, iv1, iv2 分别为图 9.1 中的各个基本部件的输出和输入信号。

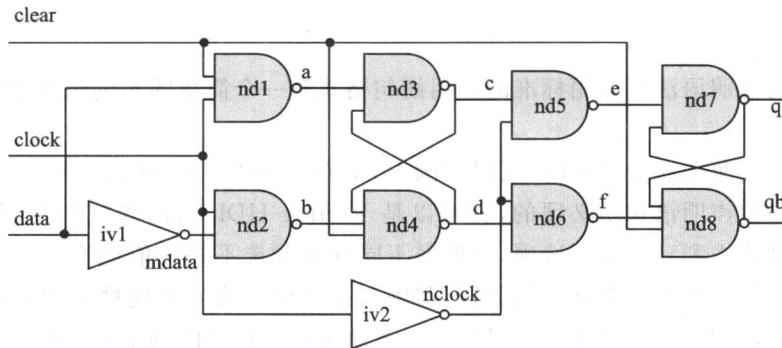


图 9.1 D 型主从触发器的电路结构图

9.1.3 由已经设计成的模块构成更高一层的模块

如果已经编制了一个模块, 如 9.1.2 节中的 flop, 可以在另外的模块中引用这个模块。引用的方法与门类型实例引用非常类似, 只需在前面写上已编的模块名, 紧跟着写上引用的实例名, 按顺序写上实例的端口名即可, 也可以用已编模块的端口名按对应的原则逐一填入, 见下面的两条语句:

- (1) flop flop_d(d1, clk, clrb, q, qn);
- (2) flop flop_d (.clock(clk), .q(q), .clear(clrb), .qb(qn), .data(d1));

这两条语句都表示实例 flop_d 引用了已编模块 flop。也可以将 flop 实例化为 flop_d。从上面两条语句可以看出实例引用时, flop_d 的端口信号与 flop 的端口对应有两种不同的表示方法。模块的端口名可以按序排列, 也可以不必按序排列。如果模块的端口名按序排列, 只需按序列出实例的端口名(见语句 1)。如果模块的端口名不按序排列, 则实例的端口信号和被引用模块的端口信号必须一一列出(见语句(2))。

[例 9.2] 引用了 9.1.2 节中已设计的模块 flop, 用它构成一个 4 位寄存器。

【例 9.2】 用触发器组成带清零端的 4 位寄存器(文件名为 hardreg.v)。

```
'include      " flop.v"
module       hardreg(d,clk,clrb,q);
  input        clk,clrb;
  input[3:0]   d;
  output[3:0]  q;

  flop        f1(d[0],clk,clrb,q[0],), //注意结束时用逗号,最后才用分号
              f2(d[1],clk,clrb,q[1],), //表示 f1~f4 都是 flop
```

```

module hardreg(d,clk,clrb,q);
    input clk,clrb;
    input[3:0] d;
    output[3:0] q;
    reg [3:0] qb;
    f1(d[0],clk,clrb,q[0]);
    f2(d[1],clk,clrb,q[1]);
    f3(d[2],clk,clrb,q[2],);
    f4(d[3],clk,clrb,q[3],);
endmodule

```

在上面这个结构描述的模块中,hardreg 定义了模块名;f1,f2,f3,f4 分别为图 9.2 中的各个基本部件,而其后面括号中的参数分别为图 9.2 中各基本部件的输入、输出信号。请注意当 f1~f4 实例引用模块 flop 时,由于不需要 flop 端口中的 qb 口,故在引用时把它省去,但逗号仍需要保留。

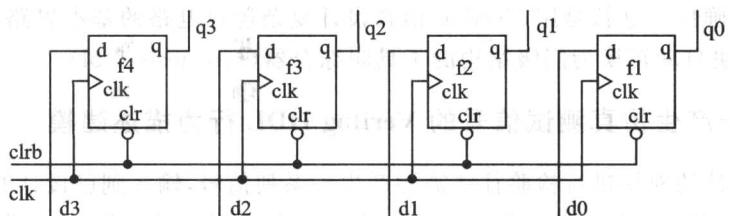


图 9.2 4 位寄存器电路结构图

显而易见,通过 Verilog 语言中的模块实例引用,可以构成任何复杂结构的电路。这种以结构方式所建立的 Verilog 模型不仅是可以仿真的,而且也是可以综合的,其本质是表示电路的具体结构。也可以说,这种 Verilog 文件也是一种结构网表。这就是以电路结构为基础建模的基本思路。

9.2 Verilog HDL 的行为描述建模

对于[例 9.1]和[例 9.2]两个例子,还可以用比较抽象 Verilog 描述方法来建立模型,如[例 9.3]所示。

【例 9.3】 用行为描述的方法来描述带清零端的 4 位寄存器(文件名为 hardreg.v)。

```

module hardreg(d,clk,clrb,q);
    input clk,clrb;
    input[3:0] d;
    output[3:0] q;
    reg [3:0] qb;

    always @ (posedge clk or posedge clrb)
    begin
        if (clrb)
            q <= 0;
        else
            q <= d;
    end
endmodule

```

[例 9.3]的行为和[例 9.2]的功能是完全一致的,实际上它们是同一物理电路的两种不同的表示方法。如果有一种工具能自动地把[例 9.3]的描述转换为[例 9.2],再细化为由 4 个[例 9.1]组成的结构,这样就把比较抽象的行为描述具体化为门级电路的描述。而门级描述表示的是电路结构,它是电路布线的依据。设计的目的就是产生行为和功能准确的电路结构。电路结构看起来相当复杂,难以理解,而行为的描述比较直观。我们可以用比较直观的行为描述来开始设计过程,通过 Verilog 语言的仿真测试验证其正确后,就完成了第一步设计工作。然后用一种工具把行为模块自动转化为门级结构,再次经过 Verilog 语言的仿真测试验证其正确后,便完成了前端的逻辑设计。接下去可以进行后端制造的准备工作,这样做大大提高了设计的效率和准确性。这就是用 Verilog 语言设计复杂逻辑电路的基本思路。这种能把行为级的 Verilog 模块自动转换为门级结构的工具叫综合器(synthesis tool)。

9.2.1 仅用于产生仿真测试信号的 Verilog HDL 行为描述建模

为了对已设计的模块进行检验往往需要产生一系列信号,输入到已设计的模块,并检查已设计模块的输出,看它们是否符合设计要求。这就要求编写测试模块,也称做测试文件(英文名为 test-bench 或 test-fix ture),常用带.tf 扩展名的文件来描述测试模块,也可以仍旧用带.V 扩展名的文件来描述测试模块。

下面的 Verilog HDL 行为描述模型用于产生时钟信号,以验证电路功能。其输出的仿真信号共有 2 个,分别是时钟 clk 和复位信号 reset。初始状态时,clk 置为低电平,reset 为高电平。reset 信号输出一个复位信号之后,维持在高电平。这一功能可利用下面的语句来实现:

```
initial
begin
    reset=1;           //初始状态
    clk=0;
    #3 reset=0;
    #5 reset=1;
end
```

以后每隔 5 个时间单位,时钟就翻转一次,这一功能可利用下面的语句来实现:

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

从而该模块所产生的时钟周期为 10 个时间单位。

完整的源程序如下:

```
module gen_clk ( clk, reset);
    output clk;
    output reset;
    reg clk, reset;

initial
begin
    reset = 1;           //initial state
    clk=0;
```

```

#3 reset = 0;
#5 reset = 1;
end
always #5 clk = ~clk;

endmodule

```

用这种方法所建立的模型主要用于产生仿真时测试下一级电路所需的信号,如下一级电路有输出反馈到上一级电路,并对上一级电路有影响时,也可以在这个模型中再加入输入信号,用于接收下一级电路的反馈信号。可以利用这个反馈信号再在这个模块中编制相应的输出信号,这样就比用简单的波形描述信号能更好地仿真实际电路。

再举一个简单的例子,即编制 9.1.3 节中完成的设计(即 hardreg 模块)的测试文件。这个测试文件不仅要包括时钟信号(clock)、数据(data[3:0])、清零信号(clearb)的变化,还须引用 4 位寄存器(hardreg)模块,以观测各种组合信号输入到该 4 位寄存器(hardreg)模块后,它的输出(q[3:0])的变化。这个测试文件完整的源程序如下:

[例 9.4] 对 4 位带清零端的寄存器进行全面的测试(文件名为 hardreg-top.v):

```

`include "flop.v"
`include "hardreg.v"           //仿真时需要包含文件"hardreg.v" 和 "flop.v"
/* * * * 如果仿真环境可以把有关的文件安排在一个项目中,只要底层模块经过编译,并记录在编译
的库中,可以不用包含文件。 * */
module hardreg_top;           //顶层模块,没有输入和输出的端口
    reg clock, clearb;         //为产生测试用的时钟和清零信号需要寄存器
    reg [3:0] data;             //为产生测试用数据需要用寄存器
    wire [3:0] qout;            //为观察输出信号需要从模块实例端口中引出线

#define stim #100 data=4'b0 //宏定义 stim 可使源程序简洁
event end_first_pass;          //定义事件 end_first_pass
    hardreg#(4) reg_4bit(.d(data), .clk(clock), .clr(clearb), .q(qout));

```

把本模块中产生的测试信号 data,clock,clearb 输入实例 reg_4bit 以观察输出信号 qout。实例 reg_4bit 实际上是已经设计好的模块 hardreg。实例引用的 hardreg 模块,根据包含文件的不同,可以是表示行为的模块,也可以是表示结构的模块。

```

***** initial
begin
    clock = 0;
    clearb = 1;
end
always #50 clock = ~clock;
always @(end_first_pass)
    clearb = ~clearb;

```

```

always @(posedge clock)
    $ display("@ time %0d clearb= %b data= %d qout= %d", $ time, clearb, data, qout);
// ****
类似于 C 语言的 printf 语句, 可打印不同时刻的信号值
***** /
initial
begin
repeat(4)          //重复 4 次产生下面的 data 变化
begin
data=4'b0000;
`stim 0001;
// ****
宏定义 stim 引用, 等同于 #100 data=4'b0001;。注意引用时要用'符号。
***** /
`stim 0010;
`stim 0011;
`stim 0100;
`stim 0101;
⋮
`stim 1110;
`stim 1111;
# 200 -> end_first_pass;
end
// ****
延迟 200 个单位时间, 触发事件 end_first_pass
***** /
$ finish;           //结束仿真
end
endmodule

```

在上面的例子中, 大家看到了一个前面未见过的语法现象: event。它用来定义一个事件, 以便在后面的操作中触发这一事件。它的触发方式是:

time(触发的时刻) ->(事件名)

上面简单地介绍了利用 Verilog HDL 门级结构建模, 来设计复杂数字电路的最基本的思路。而实用的电路设计往往并没有那么简单, 常需要利用多种方法来建立电路模型, 既利用电路图输入的方法又利用 Verilog HDL 各种建模的方法, 发挥各自在不同类型电路描述中的长处。而且要在层次管理工具的协调下把各个既独立又互相联系的模块组织成复杂的大型数字电路, 只有这样才能有效地设计出高质量的数字电路来。

9.2.2 Verilog HDL 建模在 Top-Down 设计中的作用和行为建模的可综合性问题

Verilog HDL 行为描述建模不仅可用于产生仿真测试信号对已设计的模块进行检测, 也

常常用于复杂数字逻辑系统的顶层设计,也就是说,通过行为建模把一个复杂的系统分解成可操作的若干模块,每个模块之间的逻辑关系通过行为模块的仿真加以验证。虽然这些子系统在设计的这一阶段还不都是电路逻辑,也未必能用综合器把它们直接转换成电路逻辑,但还是能把一个大的系统合理地分解为若干个较小的子系统。然后,每个子系统再用可综合风格的 Verilog HDL 模块(门级结构或 RTL 级、算法级、系统级的模块)或电路图输入的模块加以描述。当然这种描述可以分很多个层次来进行,但最终的目的是要设计出具体的电路来。所以,在任何系统的设计过程中接近底层的模块往往都是门级结构或 RTL 级的 Verilog HDL 模块,或电路图输入的模块。

由于 Verilog HDL 行为描述用于综合的历史还只有 10 多年,可综合风格的 VHDL 和 Verilog HDL 的语法只是它们各自语法的一个子集。又由于 HDL 的可综合性研究近年来发展很快,可综合子集的国际标准目前尚未最后形成^[1],因此各厂商的综合器所支持的可综合 HDL 子集也略有不同。本教材中有关可综合风格的 Verilog HDL 的内容,只着重介绍门级逻辑结构、RTL 级和部分算法级的描述,而系统级(数据流级)的综合由于还不太成熟,暂不作介绍。

所谓逻辑综合就其实质而言是设计流程中的一个阶段,在这一阶段中将较高级抽象层次的描述自动地转换成较低层次描述。就现在达到的水平而言,所谓逻辑综合就是通过综合器把 HDL 程序转换成标准的门级结构网表,而并非真实具体的门级电路。而真实具体的电路还需要利用 ASIC 和 FPGA 制造厂商的布局布线工具,根据综合后生成的标准的门级结构网表来产生。为了能转换成标准的门级结构网表,HDl 程序的编写必须符合特定综合器所要求的风格^[1]。由于门级结构、RTL 级的 HDL 程序的综合是很成熟的技术,所有的综合器都支持这两个级别 HDL 程序的综合,因而是本书综合方面介绍的重点。

9.3 用户定义的原语

用户定义的原语是从英语 User Defined Primitives 直接翻译过来的,简称 UDP。利用 UDP 用户可以定义自己设计的基本逻辑元件的功能。也就是说,可以利用 UDP 来定义自己特色的用于仿真的基本逻辑元件模块并建立相应的原语库。这样,就可以与调用 Verilog HDL 基本逻辑元件同样的方法来调用原语库中相应的元件模块,并进行仿真。由于 UDP 是用查表的方法来确定其输出的,用仿真器进行仿真时,对它的处理速度较对一般用户编写的模块快得多。与一般的用户模块比较,UDP 更为基本,它只能描述简单的能用真值表表示的组合或时序逻辑。UDP 模块的结构与一般模块类似,只是不用 module 而改用 primitive 关键词开始,不用 endmodule 而改用 endprimitive 关键词结束。在 Verilog 的语法中,还规定了 UDP 的形式定义和必须遵守的几个要点,与一般模块有不同之处,将在下面加以介绍。

1. 定义 UDP 的语法

```
primitive 元件名(输出端口名,输入端口名 1,输入端口名 2,……)
    output 输出端口名;
    input  输入端口名 1,输入端口名 2,……;
    reg   输出端口名;
```

[1] 参阅参考资料[1]。

```

    initial begin
        //输出端口寄存器或时序逻辑内部寄存器赋初值(0,1,或 X);
    end
    table
        //输入 1   输入 2   输入 3   .....   : 输出
        逻辑值   逻辑值   逻辑值   .....   : 逻辑值 ;
        逻辑值   逻辑值   逻辑值   .....   : 逻辑值 ;
        逻辑值   逻辑值   逻辑值   .....   : 逻辑值 ;
        .....           .....           .....   : ..... ;
    endtable
endprimitive

```

2. 注意点

- (1) UDP 只能有一个输出端,而且必定是端口说明列表的第一项。
- (2) UDP 可以有多个输入端,最多允许有 10 个输入端。
- (3) UDP 所有端口变量必须是标量,也就是必须是 1 位的。
- (4) 在 UDP 的真值表项中,只允许出现 0,1,X 的 3 种逻辑值,高阻值状态 Z 是不允许出现的。
- (5) 只有输出端才可以被定义为寄存器类型变量。
- (6) initial 语句用于为时序电路内部寄存器赋初值,只允许赋 0,1,X 的 3 种逻辑值,默认值为 X。

对于数字系统的设计人员来说,只要了解 UDP 的作用就可以了;而对微电子行业中的基本逻辑元器件设计工程师,必须深入了解 UDP 的描述,才能把所设计的基本逻辑元件,通过 EDA 工具呈现给系统设计工程师。有关 UDP 的详细编写方法和要点,将在高级教程里讲解。有兴趣的读者可以参阅本书的语法篇:“Verilog 软件描述语言参考手册”中有关 UDP 的语法和使用说明。

小结

在本章中介绍了 Verilog HDL 模块的不同抽象级别。一个复杂数字系统的设计往往是由若干个模块构成的,每一个模块又可以由若干个子模块构成。这些模块可以是由电路图描述的模块,也可以是由 Verilog HDL 描述的模块,各 Verilog HDL 模块可以是不同级别的描述。同一个物理电路也可以用不同级别的 Verilog HDL 模块来描述。利用 Verilog HDL 语言结构所提供的这种功能不仅可以用来描述,也可以用来验证极其复杂的大型数字系统的总体设计,把一个大型设计分解成若干个可以操作的模块,分别用不同的方法加以实现。目前,用门级和 RTL 级抽象描述的 Verilog HDL 模块可以用综合器自动转换成标准的逻辑网表(即 EDIF 文件或电子设计接口文件);用算法级描述的 Verilog HDL 模块,只有算术运算的离散步骤,如加法和乘法,综合器能把它转换成标准的逻辑网表;而不能综合连续的复杂运算过程,而用系统级描述的模块,目前尚未有综合器能把它转换成标准的逻辑网表,往往只用于系统仿真,即编写测试信号对已经设计的电路部分进行全面的测试和验证。逻辑网表可以用多种方法表示,EDIF 是常用的一种。不同形式的网表文件表示的是同一个物理意义,即电路结

构。这种结构也可以用门级 Verilog 语言来表示，我们把它称为 Verilog 网表（Verilog Netlist）。它与本章中[例 9.1]和[例 9.2]的描述在实质上是完全一致的。

思 考 题

1. Verilog HDL 的模型共有哪几种类型(级别)？
2. 每种类型的 Verilog HDL 各有什么特点？主要用于什么场合？
3. 不可综合成为电路的 Verilog 模块有什么用处？
4. 为什么说 Verilog HDL 的语言结构可以支持构成任意复杂的数字逻辑系统？
5. 什么是综合？是否任何符合语法的 Verilog HDL 程序都可以综合？
6. 综合后生成的是不是真实的电路？若不是，还需要哪些步骤才能真正变为具体的电路？
7. 为什么综合以后还可以用 Verilog 进行仿真？
8. 同一物理电路的行为模块仿真验证与结构模块的仿真验证在意义上有什么不同？
9. 为什么说前端逻辑设计必须包括结构仿真验证，只有行为验证是远远不够的？
10. 什么是 Top-Down 设计方法？通过什么手段来验证系统分块的合理性。
11. 编写两路每路为 1 位信号的二选一多路器的行为模块，再编写它的结构模块。然后编写测试模块分别对这两个模块进行测试，观测仿真运行的结果，编写实验报告。
12. 如果让你编写两路每路为 8 位信号的二选一多路器的结构模块是不是感觉麻烦？编写行为模块是不是很方便？
13. 用什么方法可以把行为模块转换为结构模块？用你掌握的综合器，把 10 题和 11 题从行为模块转换为 Verilog 网表，仔细阅读自动生成的网表文件。
14. 编写测试模块分别对行为的和自动生成的 Verilog 网表进行测试，比较仿真结果细微的不同，分析为什么不同。

第 10 章 如何编写和验证简单的纯组合逻辑模块

概 述

数字逻辑系统的设计是一个非常细致、严密和费时间的复杂过程,设计人员必须具有极其认真负责的工作态度、敏捷的头脑、顽强的毅力和细致踏实的作风。设计过程中的每一个小模块都需要极其认真地编写尽可能详细的说明,并进行严格完整的测试,以防止可能出现的错误。只有这样才能够保证由这些部件模块组成的系统能够顺利地通过检错、测试;只有具有完整说明文档的模块才能得到良好的维护和改进。

每个部件模块的设计工作包括 3 个部分:1) 电路模块的设计;2) 测试模块的设计;3) 设计文档的编写和整理。测试模块的设计和文档编写是比电路模块设计更为重要的设计环节。测试是否严密和完整决定了系统设计的成败,设计文档的完整和准确也是系统设计成败的关键,缺少完整的设计说明文件,就不能维持设计工作的连续性,为今后的调试和维护带来困难。

从学习过的数字电路基础可知,组合电路逻辑在数字系统中起着基本组件的作用。也可以说,如果不了解组合逻辑的构成,就不可能对数字逻辑系统有任何了解。采用 Verilog 或 VHDL 高层次设计方法,也是以基本逻辑电路知识为基础的。如果没有基本的逻辑电路知识,即使对 Verilog 或 VHDL 的语法了如指掌,也不可能设计出结构合理的复杂系统。对于组合逻辑部件(如多路器、比较器、加法器、乘法器、双向三态门和总线等)电路结构和性能的深入了解,是设计复杂数字逻辑系统的基础。所以,应该认真地复习一下它们的结构和逻辑表达式,并用可综合的 Verilog 模块来表示。

10.1 加法器

在数字电路课程里已学习过一位的加法电路,即全加器。它的真值表(见表 10.1)很容易写出,电路结构也很简单,仅由几个与门和非门组成。

表中的 X_i, Y_i 表示两个加数, S_i 表示和, C_{i-1} 表示来自低位的进位, C_i 表示向高位的进位。从真值表很容易写出如下逻辑表达式

$$C_i = X_i Y_i + Y_i C_{i-1} + X_i C_{i-1}$$

$$S_i = X_i \bar{C}_i + Y_i \bar{C}_i + C_{i-1} \bar{C}_i + X_i Y_i C_{i-1}$$

全加器和 S_i 的表达式也可以表示为

$$S_i = P_i \oplus C_i \quad \text{其中 } P_i = X_i \oplus Y_i \quad (10.1)$$

$$C_i = P_i \cdot C_{i-1} + G_i \quad \text{其中 } G_i = X_i \cdot Y_i \quad (10.2)$$

式(10.2)就是进位递推公式。参考清华大学出版社

表 10.1 一位全加器的真值表

| X_i | Y_i | C_{i-1} | S_i | C_i |
|-------|-------|-----------|-------|-------|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

出版的、刘宝琴老师编写的《数字电路与系统》，可以很容易地写出超前进位形成电路的逻辑，在这里不再详细介绍。

在数字信号处理的快速运算电路中常常用到多位数字量的加法运算，这时需要用到并行加法器。并行加法器比串行加法器快得多，电路结构也不太复杂，它的原理很容易理解。现在普遍采用的是 Carry - Look - Ahead - Adder 加法电路（也称超前进位加法器），只是在几个全加器的基础上增加了一个超前进位形成逻辑，以减少由于逐位进位信号的传递所造成的延迟。图 10.1 表示了一个 4 位二进制超前进位加法电路。

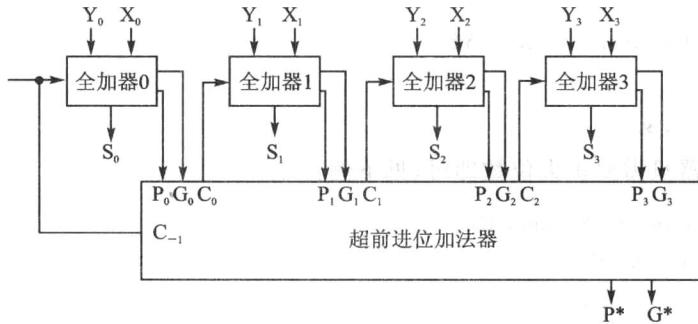


图 10.1 由 4 个一位全加器组成的超前进位 4 位加法器

同理，16 位的二进制超前进位加法电路可用 4 个四位二进制超前进位加法电路再加上超前进位形成逻辑来构成，如图 10.2 所示。依次类推可以设计出 32 位和 64 位的加法电路。

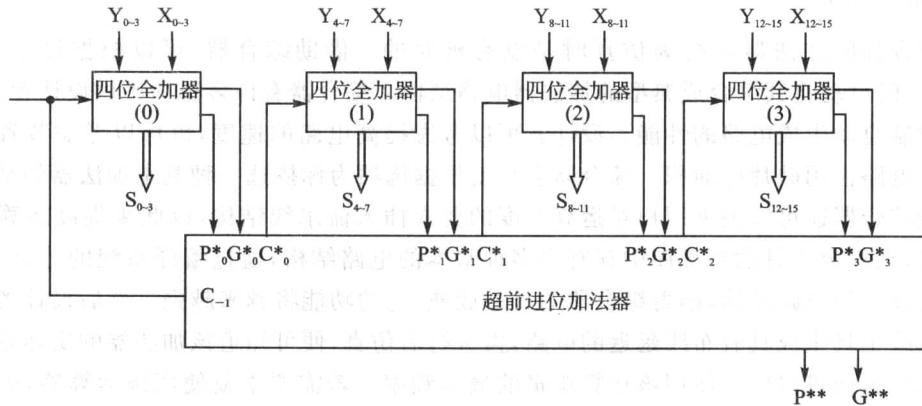


图 10.2 由 4 个四位全加器组成的超前进位 16 位加法器

在实现算法时（如卷积运算和快速富里叶变换），常常用到加法运算。由于多位并行加法器是由多层组合逻辑构成，加上超前进位形成逻辑虽然减少了延迟，但还是有多级门和布线的延迟，而且随着位数的增加延迟还会积累。由于加法器的延迟，使加法器的使用频率受到限制，这是指计算的节拍（即时钟）必须要大于运算电路的延迟，只有在输出稳定后才能输入新的数进行下一次运算。如果设计的是 32 位或 64 位的加法器，延迟就会更大。为了加快计算的节拍，可以在运算电路的组合逻辑层中加入多个寄存器组来暂存中间结果。也就是采用数字逻辑设计中常用的流水线（pipe-line）办法，来提高运算速度，以便更有效地利用该运算电路。在本节的后面还要较详细地介绍流水线结构的概念和设计方法。也可以根据情况增加运算器

的个数,以提高计算的并行度。

用 Verilog HDL 来描述加法器是相当容易的,只需要把运算表达式写出就可以了,见下例:

```
module add_4( X, Y, sum, C );
    input [3 : 0] X, Y;
    output [3 : 0] sum;
    output C;

    assign {C, Sum} = X + Y;

endmodule
```

而 16 位加法器只需要扩大位数即可,见下例:

```
module add_16( X, Y, sum, C );
    input [15 : 0] X, Y;
    output [15 : 0] sum;
    output C;

    assign {C, Sum} = X + Y;

endmodule
```

这样设计的加法器在行为仿真时是没有延时的。借助综合器,可以根据以上 Verilog HDL 源代码自动将其综合成典型的加法器电路结构。综合器有许多选项可供设计者选择,以便用来控制自动生成电路的性能。设计者可以考虑提高电路的速度,也可以考虑节省电路元件以减少电路占用硅片的面积。综合器会自动根据选项为你挑选一种基本加法器的结构。有的高性能综合器还可以根据用户对运算速度的要求插入流水线结构,以此来提高运算器的性能。可见,在综合工具的资源库中存有许多种基本的电路结构,通过编译系统的分析,自动为设计者选择一种电路结构,随着综合器的日益成熟,它的功能将越来越强。然后设计者还需通过布局布线工具生成具有布线延迟的电路,再进行后仿真,便可知道该加法器的实际延时。根据实际的延迟便可以确定使用该运算逻辑的最高频率。若需要重复使用该运算器,则需要在控制数据流动的状态机中为其安排必要的时序。

在 FPGA 的库中或某工艺的 ASIC 库中,都有参数化的加法器供设计者选用。设计者也可以通过编写代码或使用图形工具配置并引用库中的参数化加法器实例,来完成加法器电路的设计。综合器之所以能实现加法器电路也是因为库中已经存在着可配置的参数化加法器的电路结构和相应行为模型的缘故。通过综合器和仿真器的编译系统将直观的加法操作符号语句自动地与相应的电路结构和行为模型匹配而实现的。

10.2 乘法器

在数字信号处理中经常需要进行乘法运算,乘法器的设计对运算的速度有很大的影响。

本节讨论两个二进制正数的乘法电路和运算时间延迟问题,以及怎样用 Verilog HDL 模型来表示乘法运算。还将讨论当用综合工具生成乘法运算电路时,怎样来控制运算的时间延迟。

设两个 n 位二进制正数 X 和 Y,即

$$X : X_{n-1} \cdots X_1 X_0$$

$$Y : Y_{n-1} \cdots Y_1 Y_0$$

则 X 和 Y 的乘积 Z 有 $2n$ 位,并且式中的 $Y_i X_j$ 称为部分积,记为 P_i 。显然,两个一位二进制数相乘遵循如下规则

$$0 \times 0 = 0; \quad 0 \times 1 = 0; \quad 1 \times 0 = 0; \quad 1 \times 1 = 1$$

因此 $Y_i X_j$ 可用一个与门实现,记 $P_{i,j} = Y_i X_j$ 。

例:两个 4 位二进制数 X 和 Y 相乘。

| 被乘数: | | X_3 | X_2 | X_1 | X_0 | | | |
|---------------|-------|-----------|-----------|-----------|-----------|-------|-------|-------|
| \times 乘 数: | | Y_3 | Y_2 | Y_1 | Y_0 | | | |
| | | $Y_0 X_3$ | $Y_0 X_2$ | $Y_0 X_1$ | $Y_0 X_0$ | | | |
| | | $Y_1 X_3$ | $Y_1 X_2$ | $Y_1 X_1$ | $Y_1 X_0$ | | | |
| | | $Y_2 X_3$ | $Y_2 X_2$ | $Y_2 X_1$ | $Y_2 X_0$ | | | |
| | | $Y_3 X_3$ | $Y_3 X_2$ | $Y_3 X_1$ | $Y_3 X_0$ | | | |
| 乘 积: | Z_7 | Z_6 | Z_5 | Z_4 | Z_3 | Z_2 | Z_1 | Z_0 |

快速乘法器常采用网格形式的叠带阵列结构,图 10.3 示出两个 4 位二进制数相乘的结构图。图中每一个乘法单元 MU 的逻辑如图 10.4 所示,即每一个 MU 由一个与门和一个全加器构成。事实上,图 10.3 中第一行的每个 MU 可用一个与门实现,每一行最右边一个 MU 中的全加器可用半加器实现。图 10.3 实现乘法的最长延时为 1 个与门的传输延时加上 8 个全加器的传输延时。假设每个全加器产生的传输延时和与产生进位的传输延时相同,并且均相当 4 个与门的传输延时,则图 10.3 逐位进位并行乘法器的最长延时为 $1 + 8 \times 4 = 33$ 个门的传输延时。

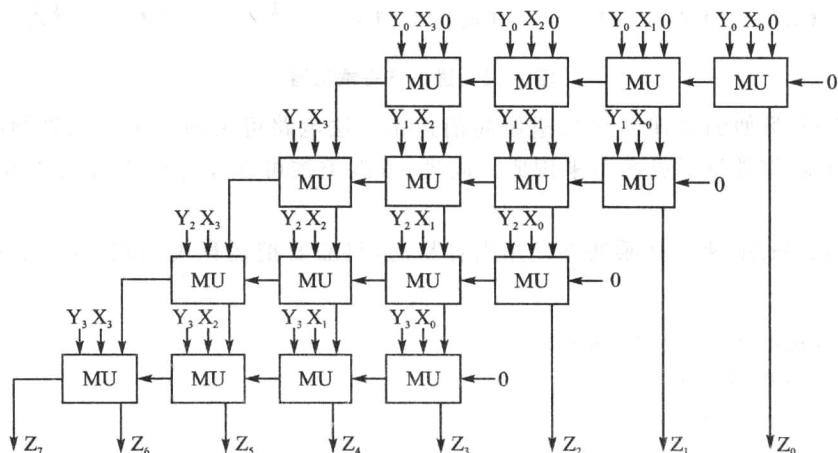


图 10.3 逐位进位并行乘法器

为了提高乘法运算速度可以改为图 10.5 所示的进位节省乘法器(Carry-Save Multiplier)。图中用了一个 3 位的超前进位加法器, 9 个图 10.4 所示的乘法单元, 7 个与门。显然, 图 10.5 中第 2 行的乘法单元中全加器可改为半加器。图 10.5 执行一次乘法运算的最长延时为 1 个与门的传输延时加上 3 个全加器的传输延时, 再加上三位超前进位加法器的传输延时。设三位超前进位加法器的传输延时为 5 个门的传输延时, 则最长延时为 $1+3\times 4+1\times 5=18$ 的传输延时。节省乘法运算时间的关键在于每个乘法单元的进位输出向下斜送到下一行, 故有进位节省乘法器之称。

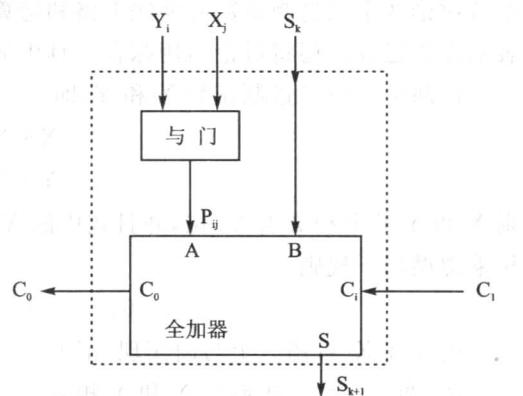


图 10.4 乘法单元 (MU)

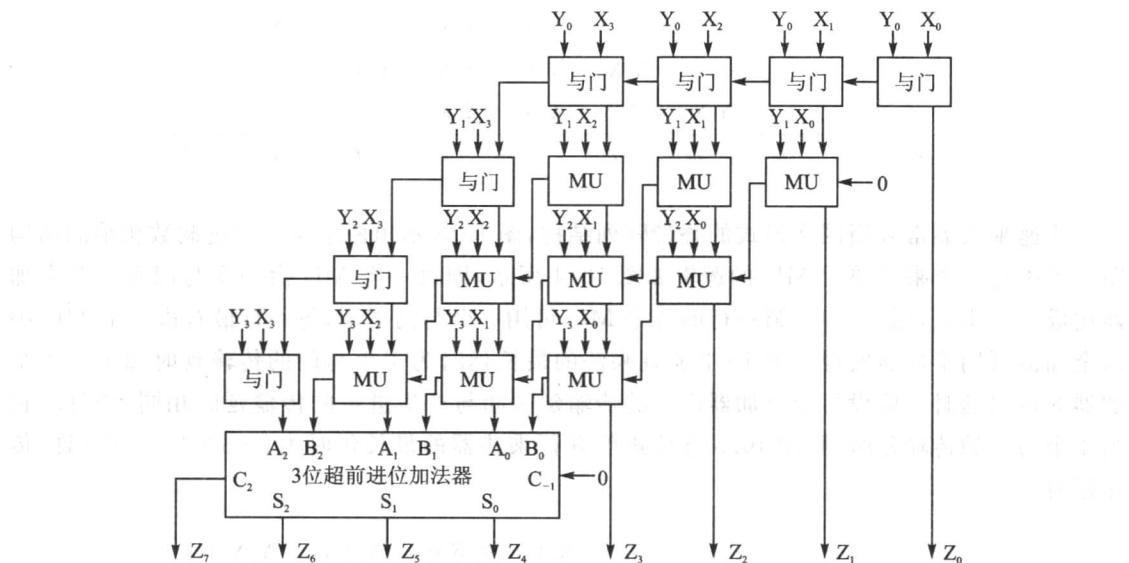


图 10.5 进位节省乘法器

根据加法器类似的道理, 8 位二进制超前进位乘法电路可用两个 4 位二进制超前进位乘法电路再加上超前进位形成逻辑来构成。同理, 依次类推可以设计出 16 位、32 位和 64 位的乘法电路。

用 Verilog HDL 来描述乘法器是相当容易的, 只需要把运算表达式写出就可以了, 见下列程序:

```
module mult_4( X, Y, Product);
  input [3 : 0] X, Y;
  output [7 : 0] Product;

  assign Product = X * Y;
```

```
endmodule
```

而 8 位乘法器只需要扩大位数即可, 见下列程序:

```
module mult_8( X, Y, Product);
    input [7 : 0] X, Y;
    output [15 : 0] Product;
    assign Product=X * Y;
endmodule
```

这样设计的乘法器在行为仿真时是没有延时的。借助综合器, 可以根据以上 Verilog HDL 源代码自动将其综合成典型的乘法器电路结构。综合器有许多选项可供设计者选择, 以便用来控制自动生成电路的性能。设计者可以考虑提高速度, 也可以考虑节省电路元件以减少电路占用硅片的面积。综合器会自动根据选项和约束文件为你挑选一种基本乘法器的结构。有的高性能综合器还可以根据用户对运算速度的要求插入流水线结构, 来提高运算器的性能。随着综合工具的发展, 其资源库中将存有越来越多种类的基本电路结构, 通过编译系统的分析, 自动为设计者选择一种更符合设计者要求的电路结构部件。然后设计者通过布局布线工具生成具有布线延迟的电路, 再进行布线后仿真, 便可精确地知道该乘法器的实际延时。根据实际的延迟便可以确定使用该运算逻辑的最高频率。若需要重复使用该运算器, 便可以根据此数据在控制数据流动的状态机中为其安排必要的时序。所以, 借助于硬件描述语言和综合工具大大加快了计算逻辑电路设计的过程。

在 FPGA 的库中或某工艺的 ASIC 库中, 都有参数化的乘法器供设计者选用。设计者可以通过编写代码或使用图形工具配置并引用库中的参数化乘法器实例, 来完成乘法器电路的设计。综合器之所以能实现乘法器电路也是因为库中已经存在着可配置的参数化乘法器的电路结构和相应行为模型的缘故。通过综合器和仿真器的编译系统将直观的乘法操作符号语句自动地与相应的电路结构和行为模型匹配而实现的。

10.3 比较器

数值大小比较逻辑在计算逻辑中是常用的一种逻辑电路, 一位二进制数的比较是它的基础。表 10.2 列出了一位二进制数比较电路的真值表。

表 10.2 一位二进制数比较电路的真值表

| X | Y | $(X > Y)$ | $(X \geq Y)$ | $(X = Y)$ | $(X \leq Y)$ | $(X < Y)$ | $(X \neq Y)$ |
|---|---|-----------|--------------|-----------|--------------|-----------|--------------|
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 |

从真值表很容易写出一位二进制数比较电路的布尔表达式

$$\begin{aligned}(X > Y) &= X \cdot (\sim Y) \\ (X < Y) &= (\sim X) \cdot Y \\ (X = Y) &= (\sim X) \cdot (\sim Y) + X \cdot Y\end{aligned}$$

便容易画出逻辑图。

位数较多的二进制数比较电路比较复杂,以前常用74LS85型四位数字比较器来构成位数较多的二进制数比较电路,如8位、16位、24位、32位的比较器。读者可以参考清华大学出版社刘宝琴老师编写的“数字电路与系统”中有关多位并行比较器设计的章节,在这里不再详细介绍。

用Verilog HDL来设计比较电路是很容易的。下面是一个位数可以由用户定义的比较电路模块。

```
module compare_n ( X, Y, XGY, XSY, XEY );
    input [width-1:0] X, Y;
    output XGY, XSY, XEY;
    reg XGY, XSY, XEY;
    parameter width=8;

    always @ ( X or Y ) //每当 X 或 Y 变化时
        begin
            if(X==Y)
                XEY=1; //设置 X 等于 Y 的信号为 1
            else XEY=0;

            if (X>Y)
                XGY=1; //设置 X 大于 Y 的信号为 1
            else XGY=0;

            if (X<Y)
                XSY=1; //设置 X 小于 Y 的信号为 1
            else XSY=0;
        end
    endmodule
```

综合工具能自动把以上原代码综合成一个8位比较器。如果在实例引用时分别改变width值为16和32,综合工具就能自动把以上原代码分别综合成数据宽度为16位和32位的比较器。

10.4 多路器

多路选择器(Multiplexer)简称多路器,它是一个多输入、单输出的组合逻辑电路,在数字系统中有着广泛的应用。它可以根据地址码(选择码)的不同,从多个输入数据流中选取一个,让其输出到公共的输出端。在算法电路的实现中多路器常用来根据地址码(选择码)来调度数

据。可以很容易地写出一个有两位地址码,可以从四组输入信号线中选出一组通过公共输出端输出的功能表,四选一功能如表 10.3 所列。

表 10.3 四选一功能输出

| 地址 1 | 地址 0 | 输入 1 | 输入 2 | 输入 3 | 输入 4 | 输出 |
|------|------|------|------|------|------|------|
| 0 | 0 | 1 | 0 | 0 | 0 | 输入 1 |
| 0 | 1 | 0 | 1 | 0 | 0 | 输入 2 |
| 1 | 0 | 0 | 0 | 1 | 0 | 输入 3 |
| 1 | 1 | 0 | 0 | 0 | 1 | 输入 4 |

可以很容易地写出它的布尔表达式,也很容易画出逻辑图。但是当地址码比较长,比如有 12 位长,而且每组输入信号位数较宽(如位宽为 8)信号组的数目又较多时,再加上又需多路选择使能控制信号时,其逻辑电路的基本单元需要量是较大的,如画出逻辑图来就显得很复杂,电路具体化后不易于理解(读者可以参考阎石老师主编的“数字电子技术基础”教材,复习多路选择器的概念)。

用 Verilog HDL 来设计多路选择器电路是很容易的。下面是带使能控制信号(nCS)的数据位宽可由用户定义的(8 位)八路数据通道选择器模块的程序。

```

module Mux_8( addr,in1, in2, in3, in4, in5, in6, in7, in8, Mout, nCS);
  input [2:0] addr;
  input [width-1:0] in1, in2, in3, in4, in5, in6, in7, in8;
  input nCS;
  output [width-1:0] Mout;
  reg[width-1:0] Mout;
  parameter width = 8;

  always @ (addr or in1 or in2 or in3 or in4 or in5 or in6 or in7 or in8 or nCS)
  begin
    if (! nCS)                                //nCS 低电平使多路选择器工作
      case(addr)
        3'b000: Mout = in1;
        3'b001: Mout = in2;
        3'b010: Mout = in3;
        3'b011: Mout = in4;
        3'b100: Mout = in5;
        3'b101: Mout = in6;
        3'b110: Mout = in7;
        3'b111: Mout = in8;
      endcase
    else                                         //nCS 高电平关闭多路选择器
      Mout = 0;
  end
endmodule

```

综合工具能自动把以上源代码综合成一个数据位宽为 8 的八路选一数据多路器。如果在实例引用时分别改变参数 width 值为 16 和 32, 综合工具就能自动把以上源代码分别综合成数据宽度为 16 位和 32 位的八选一数据多路器。

10.5 总线和总线操作

总线是运算部件之间数据流通的公共通道。在硬线逻辑构成的运算电路中只要电路的规模允许, 可以比较自由地来确定总线的位宽, 因此可以大大提高数据流通的速度。适当的总线位宽, 配合适当并行度的运算逻辑和步骤就能显著地提高专用信号处理逻辑电路的运算能力。各运算部件和数据寄存器组可以通过带控制端的三态门与总线的连接。通过对控制端电平的控制来确定在某一时间片段内, 总线归哪两个或哪几个部件使用(任何时间片段只能有一个部件发送, 但可以有一个或几个接收)。用 Verilog 描述总线和总线操作是非常简单的。

图 10.6 是三态数据总线的开关逻辑图。下面是一个简单的与总线有接口的模块是如何对总线进行操作的例子:

```
module SampleOfBus( DataBus, link_bus, write );
    inout [11:0] DataBus;           //12 位宽的总线双向端口
    input link_bus;                //向总线输出数据的控制电平
    reg [11:0] outsigs;           //模块内 12 位宽的数据寄存器
    reg[13:0]insigs;              //模块内 14 位宽的数据寄存器

    assign DataBus = (link_bus) ? outsigs : 12 'h zzz ;
    //当 link_bus 为高电平时通过总线把储存在 outsigs 的计算结果输出
```

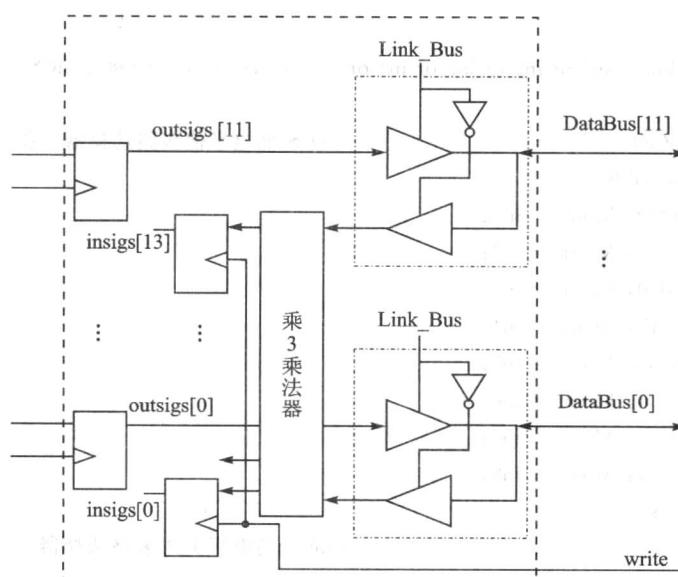


图 10.6 三态数据总线的开关逻辑图

```

always @ (posedge write)
begin
    insigs <= DataBus * 3;
end
endmodule

```

通过以上例子可知,为使这个总线连接模块能正常工作的最重要的因素是与其他模块的配合,如:何时提供 write 信号?此时 DataBus 上的数据是否已正确提供?何时提供 Link_Bus 电平?输出的数据是否能被有效地利用?控制信号的相互配合由同步状态机控制的开关阵列控制。在后面几章里将详细介绍如何用 Verilog HDL 来设计复杂的同步状态机,并产生精确同步的开关控制信号来控制数据的正确流动。

10.6 流水线

1. 流水线设计技术

流水线(pipe-line)的设计方法已经在高性能的、需要经常进行大规模运算的系统中得到广泛的应用,如 CPU(中央处理器)等。目前流行的 CPU,如 intel 的奔腾处理器在指令的读取和执行周期中充分地运用了流水线技术以提高它们的性能。高性能的 DSP(数字信号处理)系统也在它的构件(building-block functions)中使用了流水线设计技术。通过加法器和乘法器等一些基本模块,本节讨论了有关流水线的一些基本概念,并对采用两种不同的设计方法:纯组合逻辑设计和流水线设计方法时,在性能和逻辑资源的利用等方面的不同进行了比较和权衡。

2. 流水线设计的概念

所谓流水线设计实际上是把规模较大、层次较多的组合逻辑电路分为几个级,在每一级插入寄存器组并暂存中间数据。K 级的流水线就是从组合逻辑的输入到输出恰好有 K 个寄存器组(分为 K 级,每一级都有一个寄存器组),上一级的输出是下一级的输入而又无反馈的电路。

图 10.7 表示了如何把组合逻辑设计转换为相同组合逻辑功能的流水线设计。这个组合逻辑包括两级:第一级的延迟是 T₁ 和 T₃ 两个延迟中的最大值;第二级的延迟等于 T₂ 的延迟。为了通过这个组合逻辑得到稳定的计算结果输出,需要等待的传播延迟为 [max(T₁, T₃) + T₂] 个时间单位。在从输入到输出的每一级插入寄存器后,流水线设计的第一级寄存器所具有的总的延迟为 T₁ 与 T₃ 时延中的最大值加上寄存器的 T_{eo}(触发时间)。同样,第二级寄存器延迟为 T₂ 的时延加上 T_{eo}。采用流水线设计为取得稳定的输出总体计算周期为

$$\max(\max(T_1, T_3) + T_{eo}, (T_2 + T_{eo}))$$

流水线设计需要两个时钟周期来获取第一个计算结果,而只需要一个时钟周期来获取随后的计算结果。开始时用来获取第一个计算结果的两个时钟周期被称为采用流水线设计的首次延迟(latency)。对于 CPLD 来说,器件的延迟如 T₁、T₂ 和 T₃ 相对于触发器的 T_{eo} 要长得多,并且寄存器的建立时间 T_{su} 也要比器件的延迟快得多。只有在上述关于硬件时延的假设为真的情况下,流水线设计才能获得比同功能的组合逻辑设计更高的性能。

采用流水线设计的优势在于它能提高吞吐量(throughput)。假设 T₁、T₂ 和 T₃ 具有同样的传递延迟 T_{pd}，对于组合逻辑设计而言，总的延迟为 2 * T_{pd}；对于流水线设计来说，计算周期为 (T_{pd} + T_{co})。前面提及的首次延迟(latency)的概念实际上就是将(从输入到输出)最长的路径进行初始化所需要的时间总量；吞吐延迟则是执行一次重复性操作所需要的时间总量。在组合逻辑设计中，首次延迟和吞吐延迟同为 2 * T_{pd}。与之相比，在流水线设计中，首次延迟是 2 * (T_{pd} + T_{co})，而吞吐延迟是 T_{pd} + T_{co}。如果 CPLD 硬件能提供快速的 T_{co}，则流水线设计相对于同样功能的组合逻辑设计能提供更大的吞吐量。

典型的富含寄存器资源的 CPLD 器件(如 Lattice 的 ispLSI 8840)的 T_{pd} 为 8.5 ns, T_{co} 为 6 ns。

流水线设计在性能上的提高是以消耗较多的寄存器资源为代价的。对于非常简单的用于数据传输的组合逻辑设计，例如上述例子，可将它们转换成流水线设计且可能只需增加很少的寄存器单元。随着组合逻辑变得复杂，为了保证中间的计算结果都在同一时钟周期内得到，必须在各级之间加入更多的寄存器。如果需要在 CPLD 中实现复杂的流水线设计，以获取更优良的性能，如具有丰富寄存器资源的 CPLD 结构及可预测的延迟两大特点的 FPGA，是一个很有吸引力的选择。

3. 流水线加法器(或乘法器)与组合逻辑加法器(或乘法器)的比较

采用流水线技术可以在相同的半导体工艺的前提下通过电路结构的改进来大幅度地提高重复多次使用的复杂组合逻辑计算电路的吞吐量。下面是一个 n 位全加器的例子，如图 10.8 所示。为实现该加法功能需要 3 级电路：

- (1) 加法器输入的数据产生器和传送器；
- (2) 数据产生器和传送器的超前进位部分；
- (3) 数据产生、传送功能和超前进位三者求和电路。

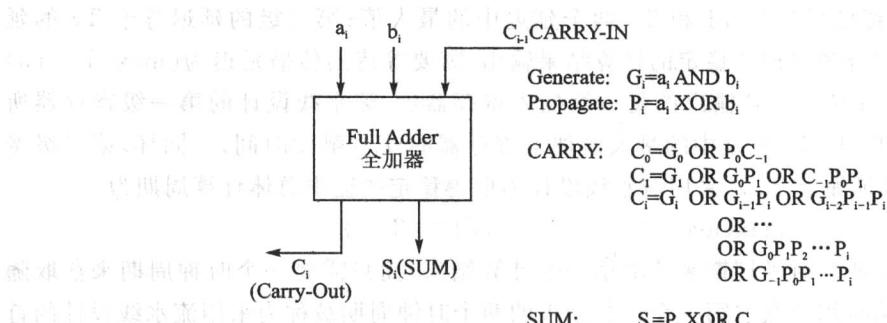


图 10.8 n 位全加器的方程式

在 n 位组合逻辑全加器中插入三层寄存器或寄存器组，将它转变为 n 位流水线全加器，如

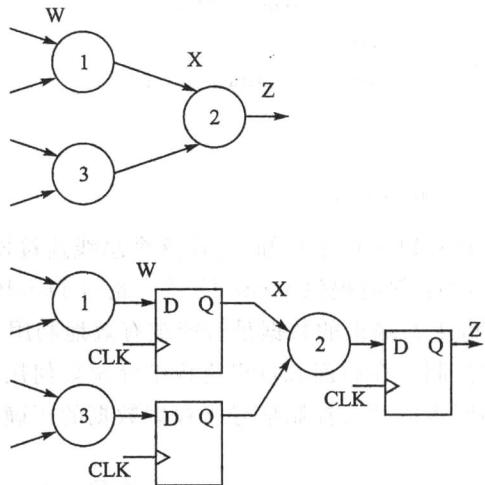


图 10.7 组合逻辑设计转化为流水线设计

图 10.9(a)所示。由于进位 C_{-1} 既是第一级逻辑的输入, 又是第二级逻辑输入, 因此将 C_{-1} 进位改为流水线结构时需要使用两级寄存器。同样地, 发生器输出在作为求和单元的输入之前, 也要多次插入寄存器。作为求和单元的输出, 进位 C_{out} 要达到同一流水线的级别也需要插入两层寄存器。

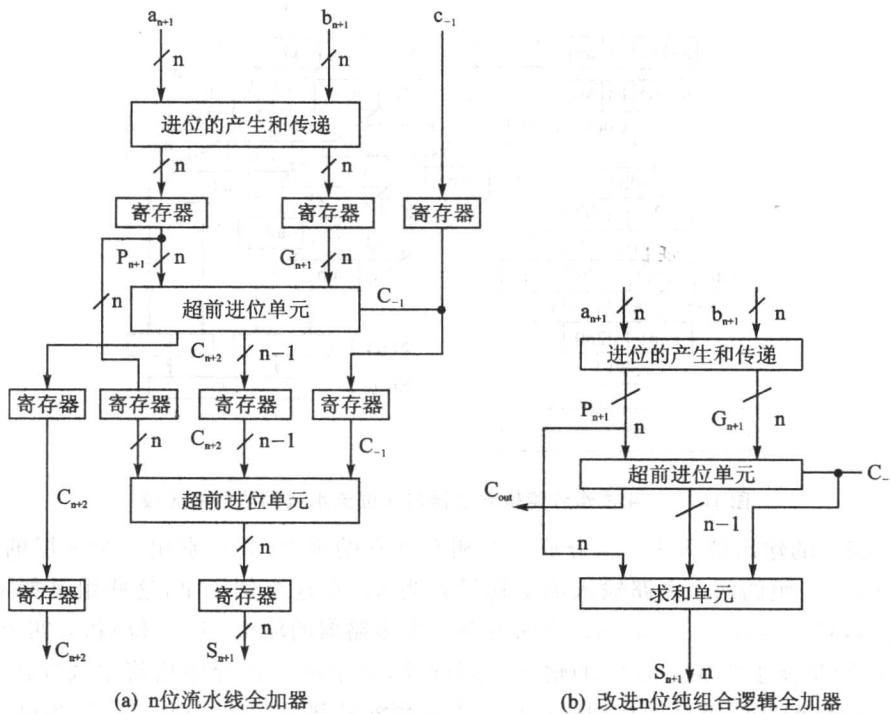


图 10.9 n 位纯组合逻辑全加器

若用拥有 840 个宏单元和 312 个有寄存能力的 I/O 单元(Lattice ispLSI8840)分别来实现 16 位组合逻辑全加器和 16 位流水线全加器并比较它们的运行速度, 对于 16 位组合逻辑全加器, 共用了 34 个宏单元。执行一次计算需经过 3 个 GLB 层, 每次计算总延迟为 45.6 ns, 而 16 位流水线全加器共用了 81 个宏单元。执行一次计算只需经过 1 个 GLB 层, 每次计算总延迟为 15.10 ns(但第一次计算需要多用 3 个时钟周期), 吞吐量约增加了 3 倍。

4. 流水线乘法器与组合逻辑乘法器的比较

首先, 若采用一个 4×4 乘法器的例子来说明部分积乘法器的基本概念; 然后, 通过一个复杂得多的 6×10 乘法器来比较流水线乘法器和组合逻辑乘法器这两个不同设计方法的实现在性能上有何差异。

如图 10.10 所示, 4×4 乘法器可以被分解为部分积的矢量和(或称加权和), 比如说是 16 个 1×1 乘法器输出的矢量和。这里并没有直接在 4×4 乘法器的每一级都插入寄存器以达到改为流水线结构的目的, 而是将其分割为 1×4 乘法器来产生所有的部分积矢量。这样分割的结果是形成了两级的流水线设计, 相对 1×1 乘法器的组合具有更短的首次延迟, 而吞吐延迟相同。每一级的流水线求和用图 10.9(a)所示的流水线加法器来实现。

可以用一个类似图 10.10 中的 4×4 或更为复杂的 6×10 流水线乘法器来比较流水线乘

法器与非流水线乘法器之间在性能上的差异。如图 10.11 所示,该 $6 * 10$ 流水线乘法器采用 6 个 10 位乘法器来实现 $1 * 10$ 乘法—— $a_0 * b[9:0]$, $a_1 * b[9:0]$, $a_2 * b[9:0]$, $a_3 * b[9:0]$, $a_4 * b[9:0]$, $a_5 * b[9:0]$ 。由于 a_i 非 0 即 1,那么 $1 * 10$ 乘法器的结果是 $b[9:0]$ 或 0,这表示下一级的两个输入不是 $b[9:0]$ 就是 0。

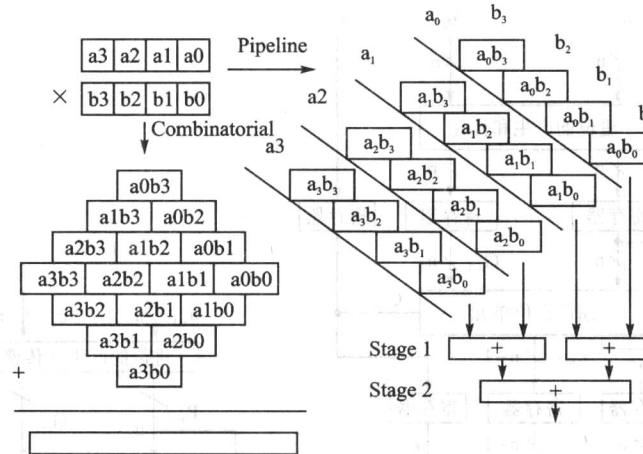


图 10.10 4 位组合逻辑乘法器与 4 位流水线乘法器的比较

6 个多路器的输出被两两一组分成三个相互独立的组合,并分别用一个 3 层的流水线加法器加起来,每一组的两个多路输入的下标号差为 3。在这个例子里,这些组是如下组织的: $[a_5, a_2], [a_4, a_1], [a_3, a_0]$ 。 $[a_5, a_2]$ 意味着第一个多路器的输出 M(10 位)和第四个多路器的输出 N(10 位)是流水线加法器 O 的输入。同样地,其余的两组分别用流水线加法器 P 和 Q 加在一起。这样的两两组合能在加的过程中去除额外的部分积项。以 $[a_5, a_2]$ 为例,其等式一般表示为

$$G(j, 0) = \{000, M(i, 0)\} \text{ and } \{N(i, 0), 000\}$$

$$P(j, 0) = \{000, M(i, 0)\} \text{ xor } \{N(i, 0), 000\} \quad (0 <= i <= 9, 0 <= j <= 12)$$

$$C_j = G_j \text{ or } G_{j-1}P_j \text{ or } G_{j-2}P_{j-1}P_j \quad (0 <= j <= 12)$$

or ... or $G_0P_1P_2P_3 \dots P_j$

$$S_k = P_k \text{ xor } C_{k-1} \quad (0 <= k <= 13)$$

由于 M 与 N 的间隔为 3,M 的高三位和 N 的低三位必定是 0。因此,M 和 N 完成与操作后, G_0, G_1, G_2 和 G_{10}, G_{11}, G_{12} 必定为 0。进一步地说,因为存在这样一些结果为 0 的发生器,进位的计算就可以得到简化。既然进位计算得到了简化,那么求和运算也就自然得到了简化。同样地,流水线加法器 P, Q 的输入间隔也是 3。流水线加法器 T 和 S 的输入之间的间隔分别为 1 和 2。由于加法器 T 是一个三层流水线加法器,所以在 Q 和 S 之间也插入了三层寄存器组从而达到与 T 相同的流水线级别。

这里依然使用 Lattice 的 ispLSI8840 来比较 $6 * 10$ 乘法器的状态,可分别用组合逻辑和流水线实现的。组合逻辑的 $6 * 10$ 乘法器在用 HDL 实现时消耗了 14 个 GLB 中的 93 个宏单元。执行一次运算需要经过 5 个 GLB 层,具有的最大传递延迟为 73.5 ns。相应的是,流水线设计的 $6 * 10$ 乘法器在用 HDL 实现时消耗了 22 个 GLB 中的 360 个宏单元。执行一次运算只需经过一个 GLB 层,计算周期只要 15.30 ns,比组合逻辑的实现快 4 倍有余。该设计的相

应首次延迟是9个时钟周期。图10.11为 6×10 流水线乘法器原理图。

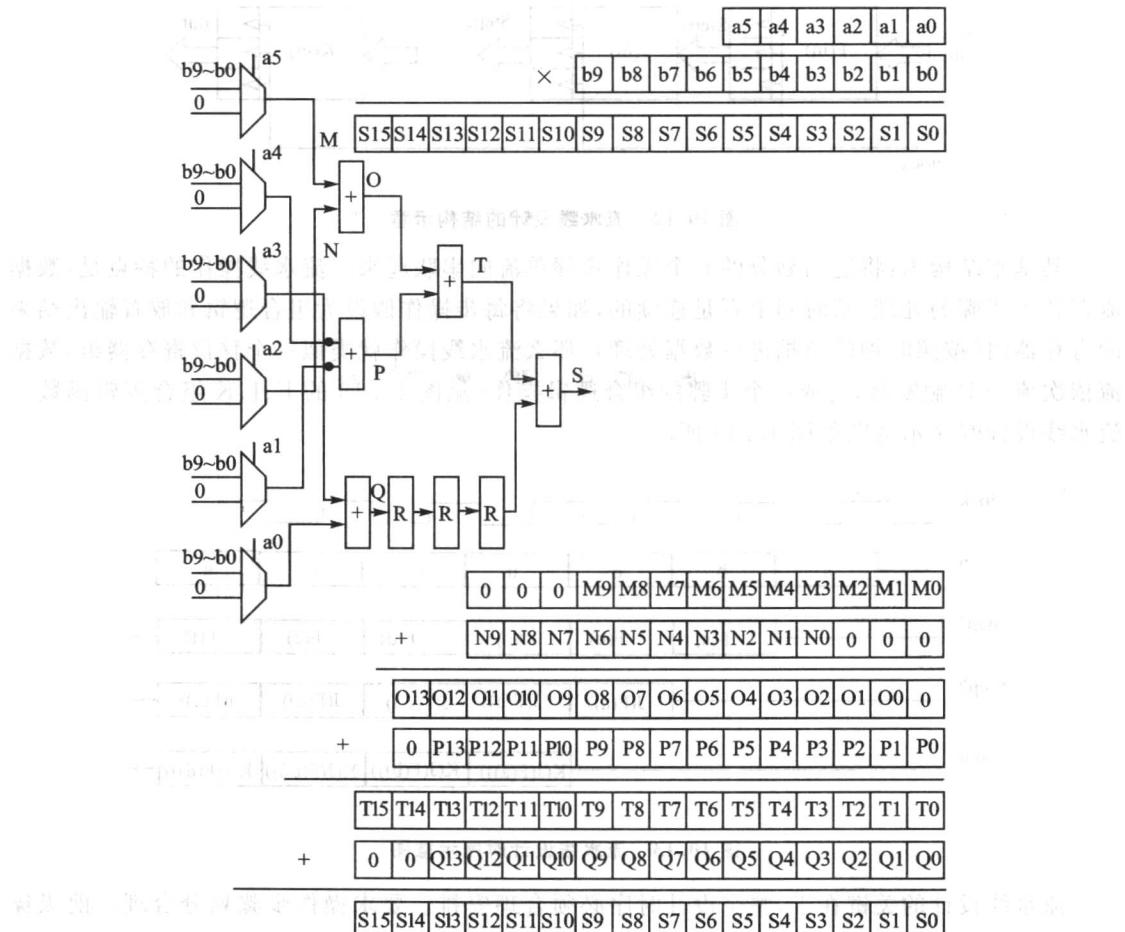


图 10.11 6×10 流水线乘法器

5. 基本的流水线操作

流水线处理是提高组合逻辑设计的处理速度和吞吐量的最常用手段。如果某个组合逻辑设计的处理流程可以分为若干个步骤，而且整个数据处理过程是“单流向”的，即没有反馈或者迭代运算，前一个步骤的输出是下一个步骤的输入，则可以考虑采用流水线设计方法提高系统的数据处理频率，即吞吐量。把组合逻辑分成延迟时间基本相等的小块，如图10.12中的1, 2, 3, ..., n个块，每块完成一定的组合逻辑功能(F, J, \dots, K)都用寄存器暂时保存组合逻辑输出的数据值，以此作为下一级组合逻辑块的输入，每个块都有寄存器暂时保存组合逻辑输出值，只要小块的组合逻辑的延迟小于时钟周期，整个组合逻辑的输入值每个时钟就可以变化一次，不会由于组合逻辑的延迟引起输出值的错误。若没有这些寄存器来暂时保存局部组合逻辑的输出值，则为了保证整体组合逻辑的输出正确，输入端信号的变化周期必须大于整体逻辑的延迟时间。数据处理的吞吐量受到限制。采用了流水线方法，虽然第一次输出有较长的延迟，但过了若干个周期后，每个时钟周期可以输出值一次，数据处理的频率，即吞吐量大大增加了。

流水线设计的结构示意图如图 10.12 所示。

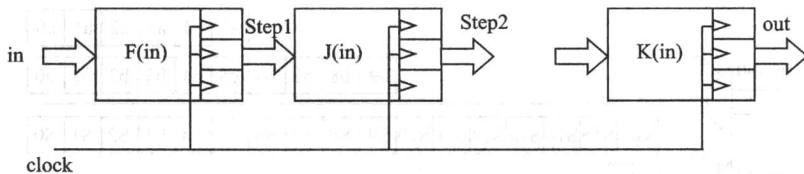


图 10.12 流水线设计的结构示意

其基本结构为：将适当划分的 n 个操作步骤单流向串联起来。流水线操作的特点是，数据流在各个步骤的处理，从时间上看是连续的，如果将每步操作假设为组合逻辑和放置输出结果的寄存器组（按照时钟的节拍进行数据处理），那么流水线操作就类似一个移位寄存器组，数据流依次流经 D 触发器，完成每个步骤的组合逻辑操作（见图 10.13 的 F、J、K 组合逻辑函数）。流水线设计时序示意图如图 10.13 所示。

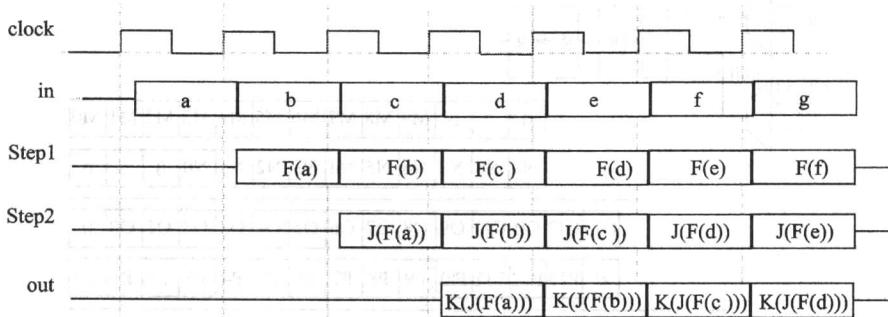


图 10.13 流水线设计时序示意图

流水线设计的关键在于，整个设计时序必须合理安排。要求操作步骤划分合理。前级操作时间最好与后级的操作时间比较接近，在这种情况下，前级的输出可以直接作为后级的输入。上面的例子就反映了这种最简单的情况。如果前级操作时间小于后级操作时间，而设计的吞吐量要求又非常高，则必须通过复制逻辑，将数据流分流，或者在前级对数据采用存储、后处理方式，否则会造成后级数据溢出。

在现代数字系统设计中经常使用到流水线处理的方法，如 WCDMA 中的 RAKE 接收机、搜索器、前导捕获，图像处理器和高速标量中央处理单元等。流水线处理方式之所以能提高时钟频率，是因为复制了处理模块，它是面积换取速度思想的又一种具体体现。

小结

改为流水线结构是提高组合逻辑吞吐量从而增强计算性能的一个重要办法。为获取高性所付出的代价是要使用更多的寄存器。要实现这样大规模的运算部件，只含少量寄存器资源的普通 PLD 器件是无法办到的，必须使用拥有大量寄存器资源的 CPLD 或 FPGA 器件或设计专用的 ASIC。当用 Verilog 语言描述流水线结构的运算部件时，要使用结构描述，才能够真正综合成设计者想要的流水线结构。简单的运算符表达式只有在综合库中存有相应的流

水线结构的宏库部件时,才能综合成流水线结构从而显著地提高运算速度。从这一意义上来说,深入了解和掌握电路的结构是进行高水平 HDL 设计的基础。

思 考 题

1. 写出 8 位加法器和 8 位乘法器的逻辑表达式,比较用超前进位逻辑和不用超前进位逻辑的延迟。
2. 为什么用算术操作符号表示的加法器和乘法器能通过综合器转变成逻辑电路?除了用算术操作符的表达式实现加法器和乘法器外,是否可以直接引用可配置的参数化实例来实现算术操作电路?
3. 提高复杂运算组合逻辑运算速度有哪些办法?
4. 如何用 Verilog HDL 模块来描述总线的操作?为什么总线的操作必须有严格的时序控制?
5. 详细解释为什么采用流水线的办法可以显著提高层次多的复杂组合逻辑的运算速度。

本章将介绍复杂数字系统的基本概念、设计方法和实现技术。通过学习本章，读者将能够理解复杂数字系统的构成原理，掌握设计方法，并能够应用所学知识解决实际问题。

第11章 复杂数字系统的构成

概 述

数字逻辑的门类千变万化,但就其本质而言,只有组合逻辑和时序逻辑两大类。它们在复杂数字系统的设计中,各自承担自己的责任。一般情况下,组合逻辑可以用来完成简单的逻辑功能,如多路器、与、或、非逻辑运算、加法和乘法等算术运算。而时序逻辑则可以用来产生与运算过程有关的(按时间节拍)多个控制信号序列。在用可综合的硬件描述语言设计的复杂运算逻辑系统中,往往用同步状态机来产生与时钟节拍密切相关(同步)的多个控制信号序列,用它来控制多路器或数据通道的开启/关闭,来使有限的组合逻辑运算器资源得到充分的运行,并寄存有意义的运算结果,或把它们传送到指定的地方,例如存储单元或者有关组件的输入/输出端口。用可综合的 Verilog HDL 来设计这样的复杂计算逻辑必须遵循一定的规定,才能使通过综合工具自动生成的电路结构比较合理,彻底消除冒险和竞争现象,即使发现控制逻辑时序存在问题,也比较容易解决。下面我们就介绍一些具体的方法和规定,只要按照这些办法和规定去做,就能使设计工作比较顺利地完成,即使设计非常复杂也有办法逐步加以解决。

11.1 运算部件和数据流动的控制逻辑

11.1.1 数字逻辑电路的种类

(1) 组合逻辑:输出只是当前输入逻辑电平的函数(有延时),与电路的原始状态无关的逻辑电路。也就是说,当输入信号中的任何一个发生变化时,输出都有可能会根据其变化而变化,但与电路目前所处的状态没有任何关系(即逻辑电路没有记忆部件)。

(2) 时序逻辑:输出不只是当前输入的逻辑电平的函数,还与电路目前所处的状态有关的逻辑电路(即逻辑电路有记忆部件)。

同步有限状态机是同步时序逻辑的基础。所谓同步有限状态机是电路状态的变化只能在同一时钟跳变沿时刻发生的逻辑电路。而状态是否发生变化还要看输入条件,如输入条件满足,当时钟跳变沿到来时刻,则进入下一状态;否则即使时钟不断跳变,电路系统仍停留在原来的状态。利用同步有限状态机可以设计出极其复杂灵活的数字逻辑电路系统,产生各种有严格时序和条件要求的控制信号波形,有序地控制计算逻辑中数据的流动。

11.1.2 数字逻辑电路的构成

(1) 组合逻辑:组合逻辑是由与、或、非门组成的网络。常用的组合电路有多路器、数据通路开关、加法器、乘法器等。

(2) 时序逻辑:时序逻辑是由多个触发器和多个组合逻辑块组成的网络。常用的有计数