



第4章 运算符、赋值语句和结构说明语句

运算符及表达式

运算符按其功能可分为以下几类

- 1) 算术运算符(+, -, ×, /, %)
- 2) 赋值运算符(=, <=)
- 3) 关系运算符(>, <, >=, <=)
- 4) 逻辑运算符(&&, ||, !)
- 5) 条件运算符(?:)
- 6) 位运算符(~, |, ^, &, ^~)
- 7) 移位运算符(<<, >>)
- 8) 拼接运算符({ })
- 9) 其它

按其所带操作数的个数运算符可分为三种：

- 1) 单目运算符(unary operator):可以带一个操作数,操作数放在运算符的右边。
- 2) 双目运算符(binary operator):可以带二个操作数,操作数放在运算符的两边。
- 3) 三目运算符(ternary operator):可以带三个操作数,这三个操作数用三目运算符分隔开。

`clock = ~clock;` //~是一个单目取反运算符, clock是操作数。

`c=a|b;` //|是一个双目按位或运算符, a 和 b是操作数。

`r=s ? t : u;` //?: 是一个三目条件运算符, s, t, u是操作数。

基本的算术运算符

在Verilog HDL语言中，算术运算符又称为二进制运算符，共有下面几种：

- 1) + (加法运算符, 或正值运算符, 如 `rega+regb`, `+3`)
- 2) - (减法运算符, 或负值运算符, 如 `rega-3`, `-3`)
- 3) × (乘法运算符, 如 `rega*3`)
- 4) / (除法运算符, 如 `5/3`)
- 5) % (模运算符, 或称为求余运算符, 要求%两侧均为整型数据。如 `7%3` 的值为1)

在进行整数除法运算时，结果值要略去小数部分，只取整数部分。而进行取模运算时，结果值的符号位采用模运算式里第一个操作数的符号位。

模运算表达式	结果	说明
$10\%3$	1	余数为1
$11\%3$	2	余数为2
$12\%3$	0	余数为0即无余数
$-10\%3$	-1	结果取第一个操作数的符号位, 所以余数为-1
$11\%3$	2	结果取第一个操作数的符号位, 所以余数为2.

注意：在进行算术运算操作时，如果某一个操作数有不确定的值 x ，则整个结果也为不定值 x 。

位运算符

- 1) \sim //取反
- 2) $\&$ //按位与
- 3) $|$ //按位或
- 4) \wedge //按位异或
- 5) $\sim\sim$ //按位同或(异或非)

按位同或运算就是将两个操作数的相应位先进行异或运算再进行非运算

不同长度的数据进行位运算

两个长度不同的数据进行位运算时, 系统会自动的将两者按右端对齐. 位数少的操作数会在相应的高位用0填满, 以使两个操作数按位进行操作。

逻辑运算符

在Verilog HDL语言中存在三种逻辑运算符：

- 1) && 逻辑与
- 2) || 逻辑或
- 3) ! 逻辑非

逻辑运算符中“&&”和“||”的优先级别低于关系运算符，“!” 高于算术运算符。见下例：

$(a > b) \&\& (x > y)$ 可写成: $a > b \&\& x > y$

$(a == b) || (x == y)$ 可写成: $a == b || x == y$

$(!a) || (a > b)$ 可写成: $!a || a > b$

为了提高程序的可读性, 明确表达各运算符间的优先关系, 建议使用括号.

关系运算符

关系运算符共有以下四种：

$a < b$ a 小于 b

$a > b$ a 大于 b

$a \leq b$ a 小于或等于 b

$a \geq b$ a 大于或等于 b

在进行关系运算时，如果声明的关系是假的(false)，则返回值是0，如果声明的关系是真的(true)，则返回值是1，如果某个操作数的值不定，则关系是模糊的，返回值是不定值。

等式运算符

在Verilog HDL语言中存在四种等式运算符：

- 1) == (等于)
- 2) != (不等于)
- 3) === (等于)
- 4) !== (不等于)

这四个运算符都是二目运算符,它要求有两个操作数。“==”和“!=”又称为逻辑等式运算符。其结果由两个操作数的值决定。由于操作数中某些位可能是不定值x和高阻值z,结果可能为不定值x。

而“===”和“!==”运算符则不同,它在对操作数进行比较时对某些位的不定值x和高阻值z也进行比较,两个操作数必需完全一致,其结果才是1,否则为0。“===”和“!==”运算符常用于case表达式的判别,所以又称为“case等式运算符”。这四个等式运算符的优先级别是相同的。

===	0	1	x	z	==	0	1	x	z
0	1	0	0	0	0	1	0	x	x
1	0	1	0	0	1	0	1	x	x
x	0	0	1	0	x	x	x	x	x
z	0	0	0	1	z	x	x	x	x

```
module test_equal;  
reg A;
```

```
    initial
```

```
    begin
```

```
        A=1'bx;
```

```
        if(A==1'bx)
```

```
            $display("== two. A is x") ;
```

```
        if(A===1'bx)
```

```
            $display("=== three. A is x");
```

```
    end
```

```
endmodule
```


移位运算符

在Verilog HDL中有两种移位运算符：

<< (左移位运算符) 和 >> (右移位运算符)。其使用方法如下：

$a \gg n$ 或 $a \ll n$

a代表要进行移位的操作数，n代表要移几位。这两种移位运算都用0来填补移出的空位。

进行移位运算时应注意移位前后变量的位数

$$4' \text{ b}1001 \ll 1 = 5' \text{ b}10010;$$

$$4' \text{ b}1001 \ll 2 = 6' \text{ b}100100;$$

$$1 \ll 6 = 32' \text{ b}10000000;$$

$$4' \text{ b}1001 \gg 1 = 4' \text{ b}0100;$$

$$4' \text{ b}1001 \gg 4 = 4' \text{ b}0000;$$

```
module test_shift;  
  reg[3:0]  start,result;  
  reg[4:0]  result2;  
  initial  
  begin  
    start=4'b1001;  
    result=(start<<1);  
    result2=(start<<1);  
    start=(start<<1);  
  end  
endmodule
```

位拼接运算符

在Verilog HDL语言有一个特殊的运算符：位拼接运算符`{}`。用这个运算符可以把两个或多个信号的某些位拼接起来进行运算操作。

`{信号1的某几位, 信号2的某几位, ..., 信号n的某几位}`
即把某些信号的某些位详细地列出来，中间用逗号分开，最后用大括号括起来表示一个整体信号。

`{a, b[3:0], w, 3'b101}`

也可以写成为

`{a, b[3], b[2], b[1], b[0], w, 1'b1, 1'b0, 1'b1}`

在位拼接表达式中不允许存在没有指明位数的信号。这是因为在计算拼接信号的位宽的大小时必需知道其中每个信号的位宽。

缩减运算符(reduction operator)

缩减运算符是单目运算符,也有与或非运算。其与或非运算规则类似于位运算符的与或非运算规则,但其运算过程不同

```
reg [3:0] B;  
reg C;  
C = &B;
```

相当于:

```
C =( (B[0]&B[1]) & B[2] ) & B[3];
```

优先级

! ~
* / %
+ -
<< >>
< <= > >=
== != === !==
&
& ^ ~
|
&&
||
?:

最高优先级



最低优先级

关键词

在Verilog HDL中, 所有的关键词是事先定义好的确认符, 用来组织语言结构。关键词是用小写字母定义的, 因此在编写源程序时要注意关键词的书写, 以避免出错。下面是Verilog HDL中使用的关键词

always, and, assign, begin, buf, bufif0, bufif1, case, casex, casez, cmos, deassign, default, defparam, disable, edge, else, end, endcase, endmodule, endfunction, endprimitive, endspecify, endtable, endtask, event, for, force, forever, fork, function, highz0, highz1, if, initial, inout, input, integer, join, large, macromodule, medium, module, nand, negedge, nmos, nor, not, notif0, notif1, or, output, parameter, pmos, posedge, primitive, pull0, pull1, pullup, pulldown, rcmos, reg, releases, repeat, mmos, rpmos, rtran, rtranif0, rtranif1, scalared, small, specify, specparam, strength, strong0, strong1, supply0, supply1, table, task, time, tran, tranif0, tranif1, tri, tri0, tri1, triand, trior, trireg, vectored, wait, wand, weak0, weak1, while, wire, wor, xnor, xor

注意在编写Verilog HDL程序时, 变量的定义不要与这些关键词冲突.

赋值语句和块语句

赋值语句

在Verilog HDL语言中，信号有两种赋值方式：

1 非阻塞(Non_Blocking)赋值方式(如 `b <= a;`)

(1) 在语句块中，上面语句所赋的变量值不能立即就为下面的语句所用

(2) 块结束后才能完成这次赋值操作，而所赋的变量值是上一次赋值得到的

(3) 在编写可综合的时序逻辑模块时，这是最常用的赋值方法

2. 阻塞(Blocking)赋值方式(如 $b = a;$)

- (1) 赋值语句执行完后,才执行下一条语句。
- (2) b 的值在赋值语句执行完后立刻就改变的。
- (3) 在时序逻辑中使用时可能会产生意想不到的结果。

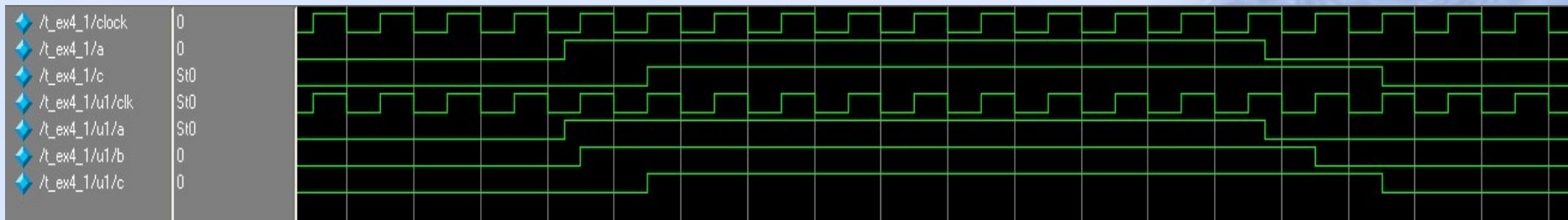
```
module ex4_1(c, a, clk);  
output c;  
input a, clk;  
reg c;  
  
reg b;  
  
always @(posedge clk)  
begin  
    b<=a;  
    c<=b;  
end  
endmodule
```

```
module t_ex4_1;  
  reg a, clock;  
  wire c;
```

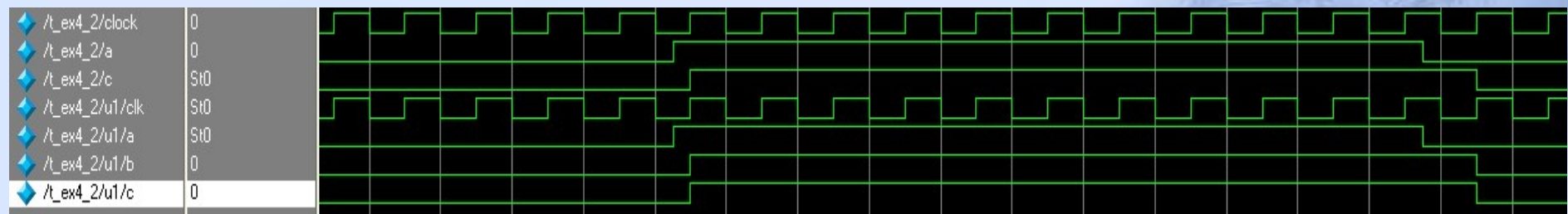
```
    initial  
    begin  
      clock=0;  
      a=0;  
      #205;  
      a=1;  
      #210;  
      a=0;  
    end
```

```
    always #10 clock=~clock;
```

```
    ex4_1 u1(.c(c),.a(a),.clk(clock));  
endmodule
```




```
module ex4_2(c, a, clk);  
output c;  
input a, clk;  
reg c;  
  
reg b;  
  
always @(posedge clk)  
begin  
    b=a;  
    c=b;  
end  
endmodule
```



```
module test_nonblock;  
reg[3:0]  a;
```

```
    initial
```

```
    begin
```

```
        a=0;
```

```
        a<=4' hc;
```

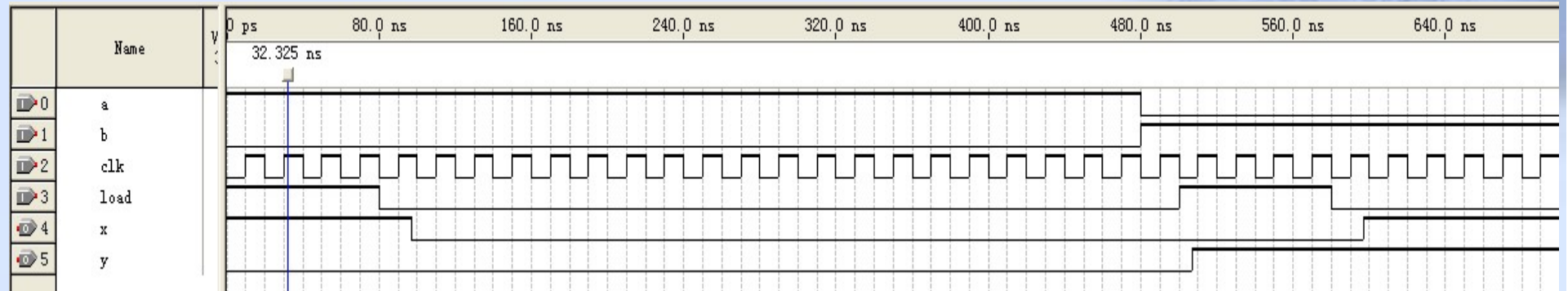
```
        a<=4' ha;
```

```
        a=4' h1;          //注意： a的最终结果是4' ha而不是4' h1
```

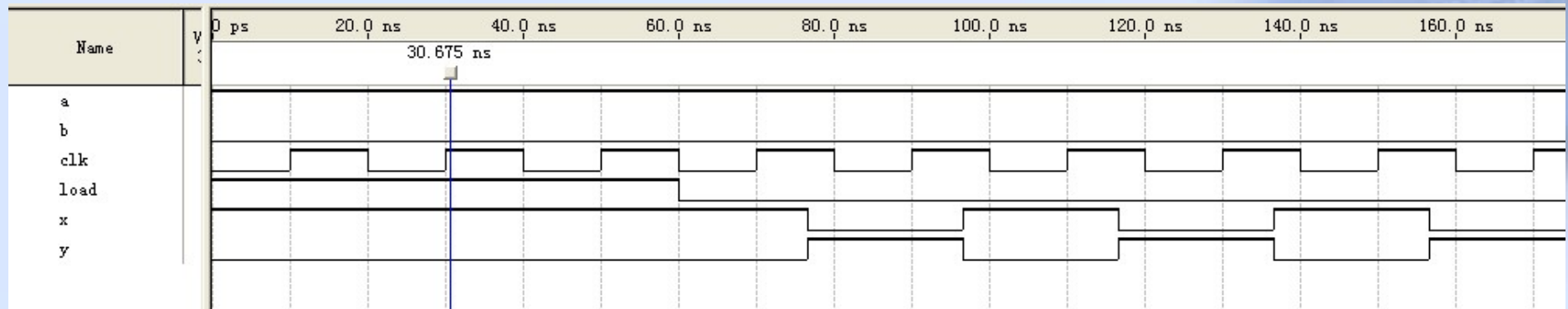
```
    end
```

```
endmodule
```

```
module jiaohuan_bianlian(x, y, clk, load, a, b);  
output  x, y;  
input  clk, load;  
input  a, b;  
reg x, y;  
  
    always @(posedge clk or posedge load)  
    begin  
        if (load)  
        begin  
            x=a;  
            y=b;  
        end  
        else  
        begin  
            x=y;  
            y=x;  
        end  
    end  
  
endmodule
```

```
module jiaohuan(x, y, clk, load, a, b) ;  
output x, y;  
input clk, load;  
input a, b;  
reg x, y;  
  
    always @(posedge clk or posedge load)  
    begin  
        if (load)  
        begin  
            x<=a;  
            y<=b;  
        end  
        else  
        begin  
            x<=y;  
            y<=x;  
        end  
    end  
endmodule
```



块语句

块语句通常用来将两条或多条语句组合在一起，使其在格式上看更象一条语句。

块语句有两种

`begin_end`语句 通常用来标识顺序执行的语句，用它来标识的块称为顺序块。

`fork_join`语句 通常用来标识并行执行的语句，用它来标识的块称为并行块。

顺序块

顺序块有以下特点：

- (1) 块内的语句是按顺序执行的，即只有上面一条语句执行后下面的语句才能执行。
- (2) 每条语句的延迟时间是相对于前一条语句的仿真时间而言的。
- (3) 直到最后一条语句执行完，程序流程控制才跳出该语句块。

顺序块的格式如下：

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

或

```
begin:块名  
块内声明语句  
    语句1;  
    语句2;  
    .....  
    语句n;  
end
```

其中块名即该块的名字，一个标识名。其作用后面再详细介绍。

块内声明语句可以是参数声明语句、reg型变量声明语句、integer型变量声明语句、real型变量声明语句。

```
module ex4_5;  
parameter d=50;  
reg[7:0] r;  
reg[3:0] a;  
event end_wave;  
  
    always @(end_wave)  
        a=4'hf;
```

```
initial
begin
    a=4'h2;
end

initial
begin
    #d r='h35;
    #d r='hE2;
    #d r='h00;
    #d r='hF7;
    #d ->end_wave;
end

endmodule
```

并行块

并行块有以下四个特点：

- (1) 块内语句是同时执行的，即程序流程控制一进入到该并行块，块内语句则开始同时并行地执行。
- (2) 块内每条语句的延迟时间是相对于程序流程控制进入到块内时的仿真时间的。
- (3) 延迟时间是用来给赋值语句提供执行时序的。
- (4) 当按时间时序排序在最后的语句执行完后或一个disable语句执行时，程序流程控制跳出该程序块。

并行块的格式如下：

```
fork  
    语句1;  
    语句2;  
    .....  
    语句n;  
join
```

或

```
fork:块名  
块内声明语句  
    语句1;  
    语句2;  
    .....  
    语句n;  
join
```


块名即标识该块的一个名字，相当于一个标识符。

块内说明语句可以是参数说明语句、reg型变量声明语句、integer型变量声明语句、real型变量声明语句、time型变量声明语句、事件(event)说明语句。

```
module ex4_6;  
parameter d=50;  
reg[7:0] r;  
reg[3:0] a;  
event end_wave;  
  
    always @(end_wave)  
        a=4'hf;  
  
initial  
begin  
    a=4'h2;  
end
```

```
initial
fork
  #d r=' h35;
  #(2*d) r=' hE2;
  #(3*d) r=' h00;
  #(4*d) r=' hF7;
  #(5*d) ->end_wave;
join
endmodule
```

块名

在VerilogHDL语言中，可以给每个块取一个名字，只需将名字加在关键词begin或fork后面即可。使用块名的原因有以下几点。

- (1) 这样可以在块内定义局部变量，即只在块内使用的变量。
- (2) 这样可以允许块被其它语句调用，如disable语句。
- (3) 在Verilog语言里，所有的变量都是静态的，即所有的变量都只有一个唯一的存储地址，因此进入或跳出块并不影响存储在变量内的值。

基于以上原因，块名就提供了一个在任何仿真时刻确认变量值的方法。

```
module ex4_5_2;  
parameter d=50;  
//reg[7:0] r;  
reg[3:0] a;  
wire[7:0] r_out;  
event end_wave;
```

```
//assign r_out=ex4_5_2.block1.r;  
assign r_out=block1.r;
```

如用本行也是正确的

```
always @(end_wave)  
    a=4'hf;
```



```
initial
begin
    a=4'h2;
end
```

```
initial
begin :block1
reg[7:0] r;
    #d r='h35;
    #d r='hE2;
    #d r='h00;
    #d r='hF7;
    #d ->end_wave;
end
```

```
endmodule
```

```
//注意reg型变量r在两处有定义
module ex4_5_3;
parameter d=50;
reg[7:0] r;//此处对r进行了定义
wire[7:0] r_out;
event end_wave;
```

```
    //assign r_out=ex4_5_2.block1.r;  如用本行也是正确的
assign r_out=block1.r;
```

```
    always @(end_wave)
        r=8'h5;
```

```
    initial
    begin
        r=8'h32;
    end
```

```
initial
begin :block1
reg[7:0] r; //此处对r进行了定义
    #d r='h35;
    #d r='hE2;
    #d r='h00;
    #d r='hF7;
    #d ->end_wave;
end

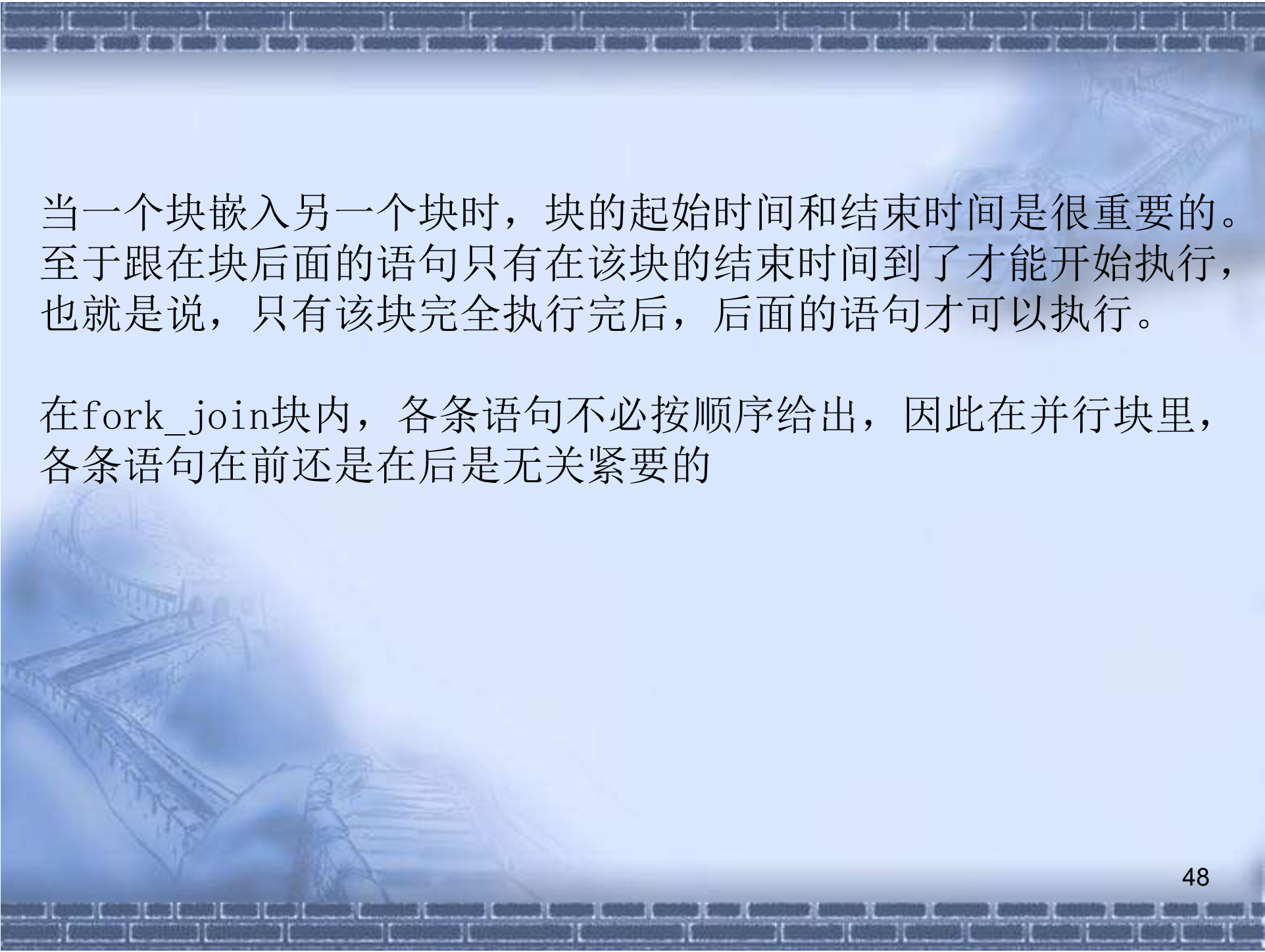
endmodule
```

起始时间和结束时间

在并行块和顺序块中都有一个起始时间和结束时间的概念。

对于顺序块，起始时间就是第一条语句开始被执行的时间，结束时间就是最后一条语句执行完的时间。

而对于并行块来说，起始时间对于块内所有的语句是相同的，即程序流程控制进入该块的时间，其结束时间是按时间排序在最后的语句执行完的时间。



当一个块嵌入另一个块时，块的起始时间和结束时间是很重要的。至于跟在块后面的语句只有在该块的结束时间到了才能开始执行，也就是说，只有该块完全执行完后，后面的语句才可以执行。

在fork_join块内，各条语句不必按顺序给出，因此在并行块里，各条语句在前还是在后是无关紧要的


```
module ex4_7;  
parameter d=50;  
reg[7:0] r;  
reg[3:0] a;  
event end_wave;  
  
    always @(end_wave)  
        a=4'hf;  
  
    initial  
    begin  
        a=4'h2;  
    end
```

```
initial
fork
  #(5*d) ->end_wave;
  #(4*d) r=' hF7;
  #(3*d) r=' h00;
  #(2*d) r=' hE2;
  #d r=' h35;
join
endmodule
```

```
module lx5_24;
reg x, y, a, b, p, m;
  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
    $display("the end time is %t", $time);
  end
endmodule
```