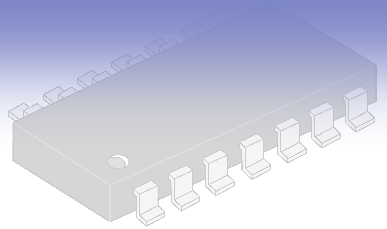
第 6 章

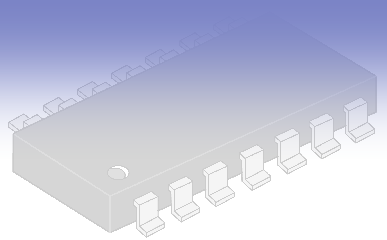
行为建模

内容



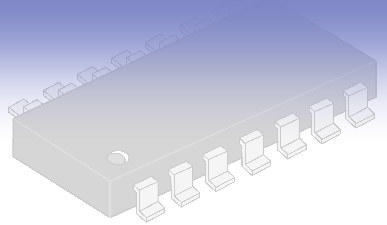
- 结构化过程语句
- 时序控制
- 顺序块和并行块
- 过程赋值语句
- 条件语句
- 多路分支语句
- 循环语句
- 生成块
- 举例

引子



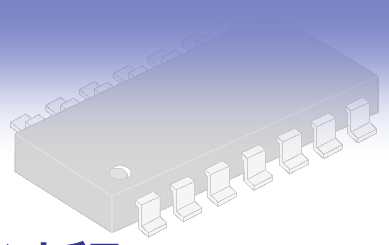
- 门级结构描述
  - 只适合规模较小的电路设计
  - 当电路规模很大、功能很复杂时，就不适合使用门级设计方法
- 数据流（data-flow）描述
  - 采用比门级描述更高的抽象层次
  - 描述操作、处理功能的实现
- 行为（behavioral）或算法描述
  - 数字系统（电路）的算法描述——不关心任何具体的物理实现
  - 算法（algorithm）
    - 信号处理运算、逻辑运算的硬件的实现方式

过程结构语句



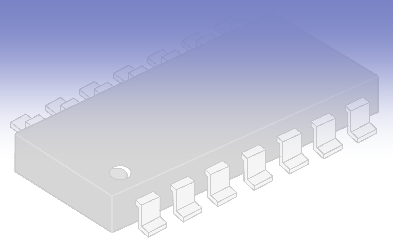
- Verilog有两种过程结构语句用于行为建模
  - initial
  - always
- 一个模块可以包含多个 initial 或 always 语句
  - 多个 initial 或 always 语句并发执行，执行顺序与其在模块的顺序无关
  - 仿真时，所有的 initial 和 always 语句都在 0 时刻开始执行
- initial 和 always 语句不能嵌套使用
  - 不能嵌套自身
  - 不能互相嵌套

过程结构语句



- 每一个 initial 和 always 语句代表一个独立的执行过程
  - 执行产生一个单独的控制流
- 如果一个initial 和 always 语句块中有多条语句
  - 使用关键字 begin … end 将多条语句组合成一个**顺序语句块**
  - 使用关键字 fork … join 将多条语句组合成一个**并行语句块**
  - begin … end 组成的顺序语句块中语句是顺序执行
  - fork … join组成的并行语句块中语句是并发执行
- 语句块可以嵌套
  - **顺序语句块**中可以包含并行语句块
  - **并行语句块**中可以包含顺序语句块
- initial 和 always 过程语句只能对寄存器类型变量进行赋值

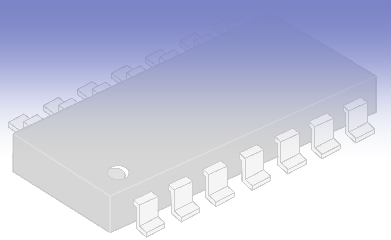
initial 语句



```
// initial 块
initial
begin
    …
    行为语句(behavioral statements)
    …
end
```

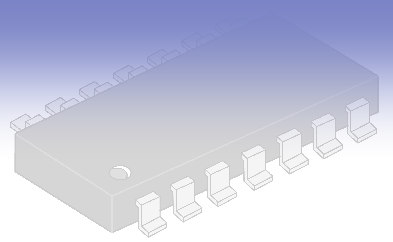
- 行为方式
  - initial块从仿真0时刻开始执行，在整个仿真过程中只执行一次
  - 若一个module中包含多个initial块
    - 多个initial块从0时刻开始并发执行
    - 每个initial块的执行都是各自独立的
  - 如果一个initial块中只有一条语句
    - 可以使用关键字 begin … end ，也可以省略

initial 语句的用途



- 初始化和信号赋值、信号监视、生成仿真波形
- 例1、
  - reg [1 : 0] y;
  - initial
  - y = 2;
    - // 初始化 y，在 0 时刻被赋值为 2
- 例2、
  - reg c;
  - initial
  - #2 c = ’b1;
    - // initial 在 0 时刻执行，使 c 在时刻 2 被赋值为 ‘b1

例、initial 语句

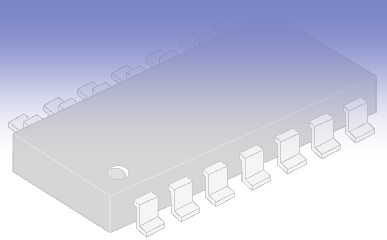


```
module stimulus;

reg x, y, a, b, m;

initial
    m = 1'b0;      // 只有一条语句，不需要使用 begin-end

initial
begin
    #5  a = 1'b1;      // 多条语句，使用 begin-end 形成语句块
    #25 b = 1'b0;
end

initial
begin
    #10 x = 1'b0;
    #25 y = 1'b1;
end

initial
    #50 $finish;

endmodule
```

- 用途
  - 初始化、信号监视、生成仿真波形

时间	所执行的语句
0	m = 1'b0;
5	a = 1'b1;
10	x = 1'b0;
30	b = 1'b0;
35	y = 1'b1;
50	$finish;

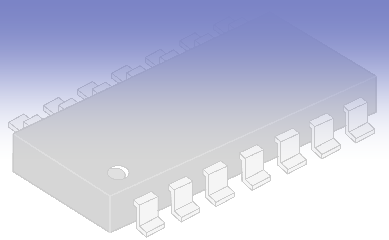
在变量声明时同时进行初始化



- 声明时赋初值

```
module ex01;

    // 定义时钟变量
    reg clock;

    // 设置时钟变量的初值为 0
    initial clock = 0;

    // 在时钟变量声明时，将其初始化
    // 只适用模块一级的变量声明
    reg clock = 0;



    always @(*) begin: BLOCK1
        reg temp;       // 不可初始化变量 temp

        temp = a;
    end
endmodule
```

always 语句

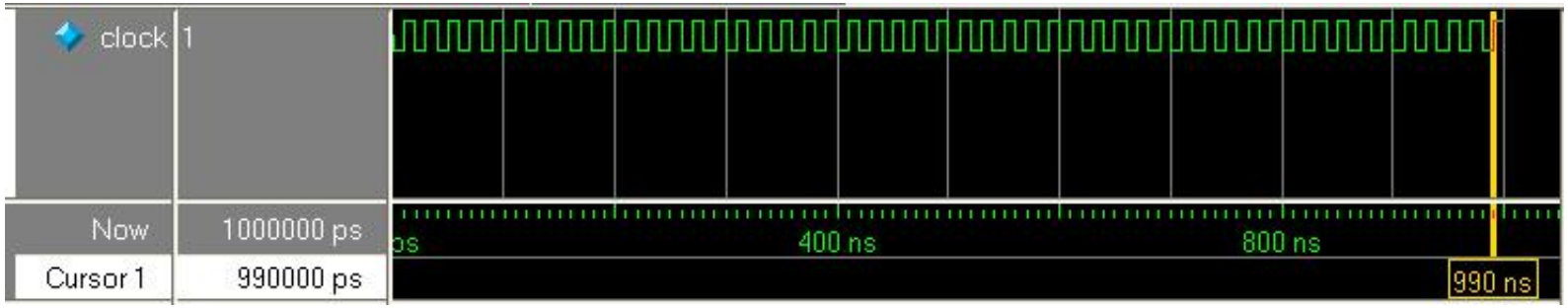


```
// always 结构块
always
begin
    …
    行为语句(behavioral statements)
    …
end
```

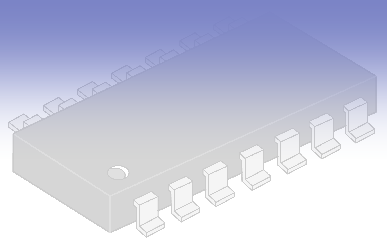
- always 语句块
  - **对于仿真而言**，从仿真0时刻开始执行语句块中的行为语句
  - 执行完最后一条语句后，从新开始执行第1条语句，无限循环下去，直到仿真结束
- always 语句块用于设计时序逻辑电路
  - **使用Verilog设计数字系统的主要描述方法**
  - 也用于对数字电路中一些反复执行的活动建模
    - 如，时钟信号发生器

例、使用 always 语句设计一个时钟发生器

```
module clock_gen (output reg clock);

// 在 0 时刻，初始化变量 clock
initial
        clock = 1'b0;

// 每半个周期翻转一次 clock，(time period = 20)
always
        #10 clock = ~clock;

initial
        #1000 $finish;

endmodule
```



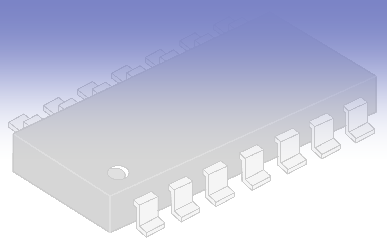
同时进行端口/数据声明和初始化



- 可以在声明端口/数据同时对其赋初始值
  - 一般用于仿真的目的
  - 在设计时，通常不需要对端口进行初始赋值

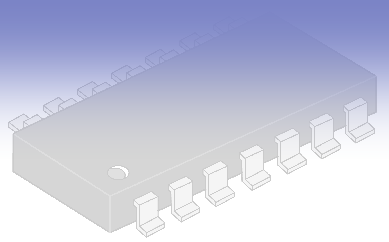
```
module adder(co, sum, a, b, ci);
    output reg co = 1'b0;              // 初始化 1 位输出变量 co
    output reg [7:0] sum = 8'b0;       // 初始化 8 位输出变量 sum
    input [7:0] a, b;
    input ci;

    always @*
        {co, sum} = a + b + ci;
endmodule
```

initial 和 always语句



- initial 语句块
  - initial块从仿真0时刻开始执行，在整个仿真过程中只执行一次
- initial 语句块用于测试模块的描述 —— 不可用于设计电路
- always 语句块
  - 从仿真0时刻开始顺序执行语句块中的行为语句
  - 执行完最后一条语句后，从新开始执行第1条语句，无限循环下去，直到仿真结束
- always 语句块用于设计各种逻辑电路
  - —— **建议**尽量避免用于测试模块中，如：testbench，top模块中
- 若一个module中包含多个initial/ always过程语句块
  - 各个 initial/always 块均从0时刻开始并发执行
  - 每个 initial/always 块都是独立执行

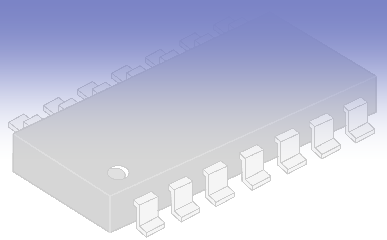
时序控制（1）



- 过程语句的时序控制有两种形式
  - ① 延时控制
  - ② 事件控制
- 延时控制
  - 常规延时控制：遇到该语句与开始执行该语句的时间间隔

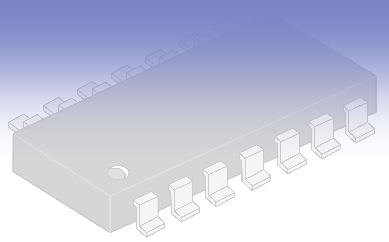
```
3  // 定义参数
4  parameter latency = 20;
5  parameter delta = 2;
6  // 定义寄存器类型变量
7  reg x, y, z, p, q;
8
9  initial
10 begin
11         x = 0; // 无延时控制
12         #10 y = 1; // 等待 10 个时间单位才执行 y = 1
13
14         #latency z = 0; // 延时 20 个时间单位
15         #(latency + delta) p = 1; // 使用表达式控制延时
16
17         #y x = x + 1; // 使用标识符表示的值控制延时
18
19         #(4:5:6) q = 0; // 最小、典型和最大延时值
20 end
```

时序控制（2）



- 延时控制——独立延时语句：
  - #DELAY;
  - 例9
    - parameter ON_DELAY = 3, OFF_DELAY = 5;
    - always
    - begin
    - # ON_DELAY;
    - refclk = 0;
    - # OFF_DELAY;
    - refclk = 1;
    - end
- 延时不必是常量，可以是表达式
  - 例10
    - # (period / 2);
    - clk = ~clk;
  - 或
    - #(period / 2) clk = ~clk;
- 如果延时表达式的值为 x 或 z，与 0 延时等效
- 如果延时表达式的值为负时，其二进制补码作为延时

时序控制（3）

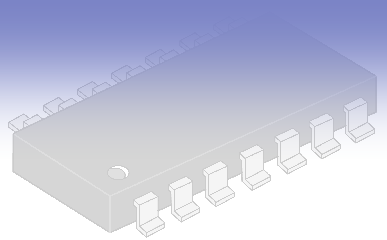


延时控制

- **内嵌赋值延时：**遇到该语句后，首先立即计算右侧表达式的值，推迟指定的时间之后，再将这个值赋给左边的变量

```
2  // 定义寄存器变量
3  reg x, y, z;
4
5  initial
6  begin
7          x = 0; z = 0;
8          y = #5 x + z; // 仿真时刻 0 得到 x 和 z 的值，计算 x + z，
9                        // 等待 5 个时间单位， 把计算结果赋给 y
10 end
11
12 // 等效的方法
13 initial
14 begin
15         x = 0; z = 0;
16         temp_xz = x + z;
17         #5 y = temp_xz; // 在当前时刻计算 x + z， 结果存储在临时变量中，
18                         // 即使在 0 到 5 个时间单位里，x 和 z 的值都发生变化，
19                         // 在第 5 个时间单位时刻， 赋给 y 的值仍为 0 时刻的计算结果
20 end
```

内嵌赋值控制



- 内嵌赋值延时和事件控制

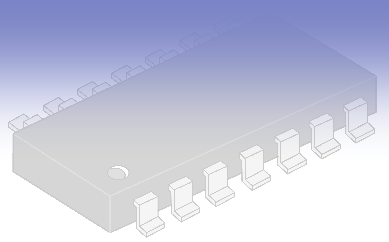
Intra-assignment timing control

With intra-assignment construct	Without intra-assignment construct

`a = #5 b;`	**begin**
	`  temp = b;`
	`  #5 a = temp;`
	end

`a = @(posedge clk) b;`	**begin**
	`  temp = b;`
	`  @(posedge clk) a = temp;`
	end

时序控制（4）



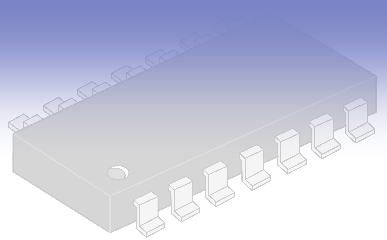
零延时

- 带零延时控制的语句在执行时刻相同的多条语句中最后执行。

```
 3 initial
 4 begin
 5      x = 0;
 6      y = 0;
 7 end
 8
 9 initial
10 begin
11      #0 x = 1; // 零延时控制
12      #0 y = 1;
13 end
```

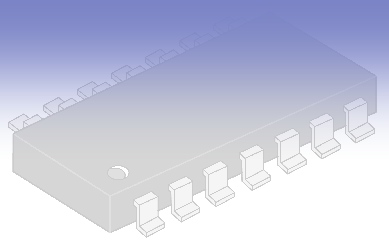
- 两个过程语句块中的 4 条赋值语句均在 0 时刻执行
- 具有零延时控制的语句最后执行
- 在 0 时刻结束时，x 和 y 的值都为 1

事件控制



- 事件
  - 寄存器类型或线网变量的值发生变化、发生正向跳变和负向跳变
- 4种事件控制方式
  - 常规事件控制 —— 两类
    - 边沿触发事件控制
    - 电平敏感事件控制
  - 敏感列表事件控制
  - 命名事件控制
- 语法
  - @ (事件) 过程语句
    - 带有事件控制的过程语句，必须等到指定的事件发生时，才能执行
    - 例、`@( ctrl ) a = b;`

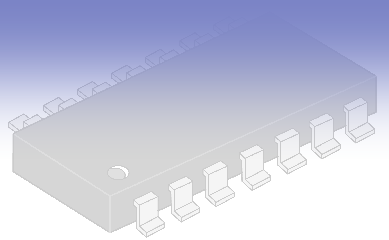
常规事件控制



- 例

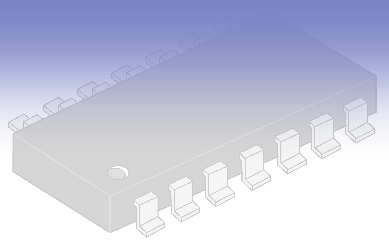
```
@(clock) q = d; // 当 clock 的值发生变化时执行
@(posedge clock) q = d; // 当 clock 正边沿到来时执行
@(negedge clock) q = d; // 当 clock 负边沿到来时执行
q = @(posedge clock) d; // 立即计算 d 的值，
                        // 当 clock 正边沿到来时赋给 q
```

边沿触发事件控制（1）



- 例、
  - @ (posedge clock)
  - state0 = state1;
- posedge （正沿） —— 正沿是下面变化的一种
  - 0 → x
  - 0 → z
  - 0 → 1
  - x → 1
  - z → 1
- negedge（负沿） —— 负沿是下面变化的一种
  - 1 → x
  - 1 → z
  - 1 → 0
  - x → 0
  - z → 0

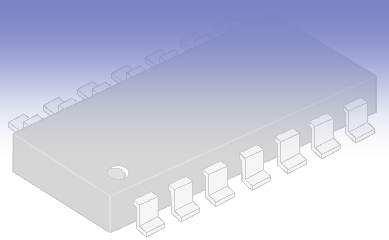
边沿触发事件控制（2）



- 例、使用系统任务（$time）获得时钟正脉冲的宽度

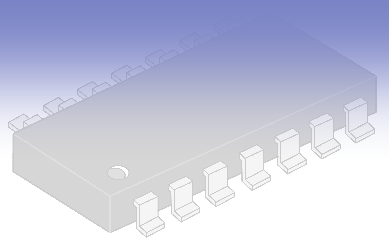
```
// 使用时间寄存器类型，定义时间变量
time  riseEdge, onDelay;

initial
    begin
        @  (posedge clk);
            riseEdge = $time; // 使用 $time 返回当前仿真时间
        @  (negedge clk);
            onDelay = $time – riseEdge; // 得到时钟信号的正脉冲宽度
            $display ( “The on-period of clock is %t.”, onDelay );
    end
```

电平敏感事件控制



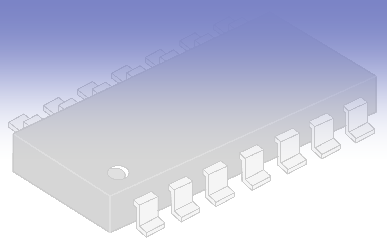
- 例1、
  - @ id;　// 等到指定的 id 上事件发生，再执行后面的语句
- 例2、
  - @ flag　y = a;
  - 或写成：**@flag;**
  - **y = a;**
  - // 当 flag 上发生事件，就执行赋值语句，如：原来是低电平（0），现在变成高电平（1），反之亦然

敏感事件控制



- 使用 wait 控制时序
  - ——等待电平敏感条件为真
  - wait 语句后面的语句块必须等待某个条件为真时才能执行
- 例、
  - **always**
  - **wait** (**count_enable**) #20 count = count + 1
  - 执行过程：
    - 如果 count_enable 为 0，不执行后面的语句 —— 只能为真
    - 仿真程序会停到下来，直到其为真，如：1 （高电平）
- @ 与 wait 比较
  - **wait** （**condition**)
    - 等待某个条件为真
  - @（**event**) —— **event** 可以是 0 变到 1、x 或 z，等等
    - 等待某个事件发生

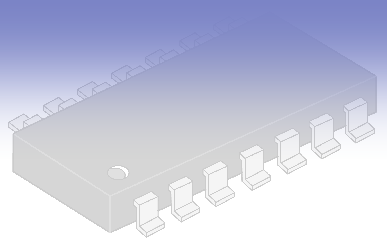
命名事件控制



- 命名事件的使用
  - 声明一个事件（类型变量）：**event　事件标识符，...**
  - 触发事件（->）
  - 执行事件控制的过程语句

```
// 定义一个事件：received_data
event received_data;
// 在 clock 正边沿到来时刻进行检测
always @(posedge clock)
begin
    if(last_data_packet)   // 如果是最后一个数据帧
        -> received_data; // 触发事件 received_data
end

always @(received_data) // 等待事件 received_data 发生
    // 当事件发生时，存储数据
    data_buf = {data_pkt[0], data_pkt[1], data_pkt[2], data_pkt[3]};
```

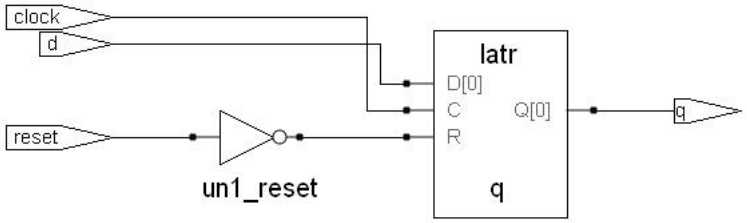
or 事件控制——敏感变量列表

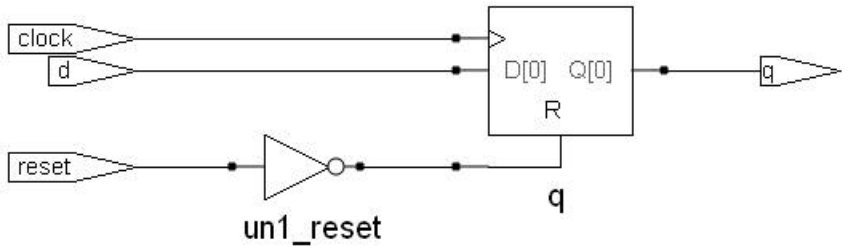


- 多个事件中的任一个发生变化，触发语句块的执行
- 敏感变量列表
  - 由 **or** 连接的多个信号变量
  - **or** 只是事件控制表达式中说明有不同事件关键字，并非逻辑或
- 例1
  - @(posedge clear or negedge reset)
  - q = 0;
- 例2
  - @ (ctrl_A or ctrl_b)
  - bus = ‘bz;

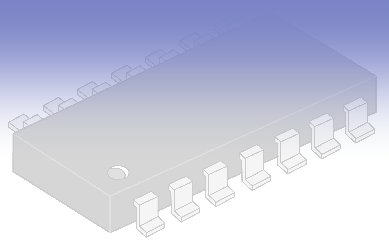
在敏感列表中用逗号，代替 or

```
// 有异步复位的电平敏感锁存器（latch）
always @( reset, clock, d)  // 等待 reset 或 clock 或 d 发生变化
begin
    if (reset)  // 如果 reset 已变成高电平，复位，即置 q 为 0.
        q = 1'b0;
    else if(clock)  // 如果 clock 已变成高电平，锁存输入信号 d
        q = d;
end

// 异步复位边沿触发 D 触发器（flipflop）
// 时钟信号 clk 正边沿到来时触发 D 触发器，reset 负边沿触发复位
always @(posedge clk, negedge reset)
    if(!reset)
        q <=0;
    else
        q <=d
```





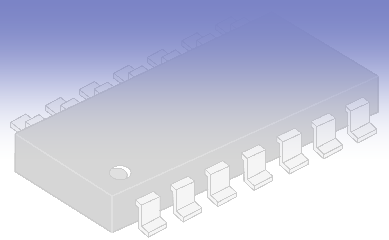
敏感列表的简化书写



- 使用 @* 和 @(*)
  - 表示对其后语句块中所有输入变量的变化都是敏感的
  - 避免敏感变量（事件控制）列表中含有多个信号变量的书写繁琐和遗漏

```
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

// 简化的替代方式
always @(*)
begin
    out1 = a ? b+c : d+e;
    out2 = f ? g+h : p+m;
end
```

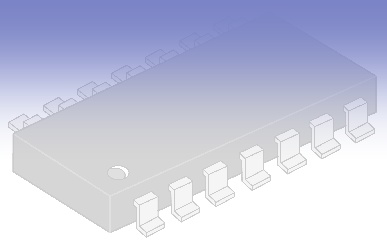
语句块



- 将两条或多条语句组合成相当于一条语句，两种
  - 顺序语句块（begin … end）
    - 语句块中的语句按给定顺序执行
  - 并行语句块（fork … join）
    - 语句块中的语句并行执行
- 语句块可以有标识符，但标识符是可选的
  - 如果有标识符
    - **在语句块中可以声明寄存器变量**
    - 语句块可以用其标识符引用，如：使用禁止语句停止语句块的执行

```
initial
begin : break_block
    i = 0;
    forever begin
        if (i==a)
            disable break_block;
        #1 i = i + 1;
    end
end
```

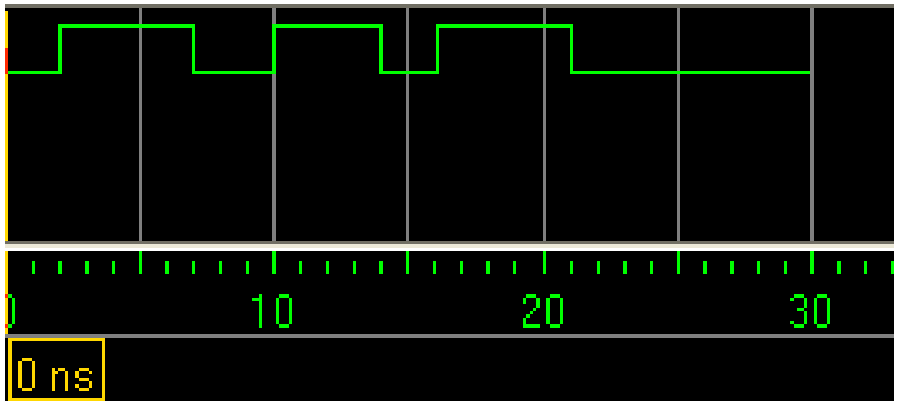
顺序语句块



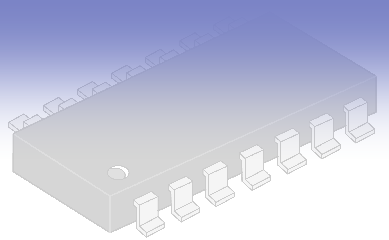
- 行为特性
  - 语句块中的语句按给定顺序执行
  - 语句的延时总是相对于前面的语句的完成时间
- 语法
  - **begin** [:block_id(declarations)]
  - 过程语句;
  - 过程语句;
  - …
  - 过程语句;
  - **end**
- 语句块可以有自己的名字 —— 称之为命名块

例、顺序语句块（1）

- 例、产生波形
  - begin
  - #2 stream = 1;
  - #5 stream = 0;
  - #3 stream = 1;
  - #4 stream = 0;
  - #2 stream = 1;
  - #5 stream = 0;
  - end
- 语句的延时总是相对于前面的语句的完成时间

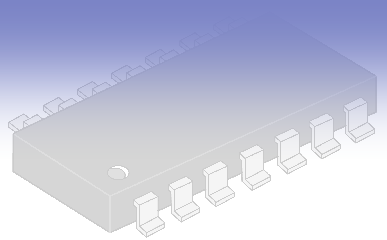


例、顺序语句块（2）



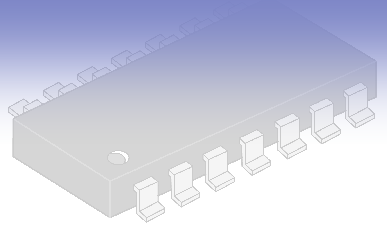
- 例、
  - begin
  - a = b | c; // 首先执行，执行完成后
  - @ (negedge clk) // 等到 clk 出现负跳沿时，再执行赋值
  - f = &a;
  - end
- 例、
  - begin: SEG // 顺序语句块带有标识 SEG，有一个局部寄存器说明
  - **reg [3:0] k**;
  - k = p & q;
  - f = & k;
  - end

并行语句块

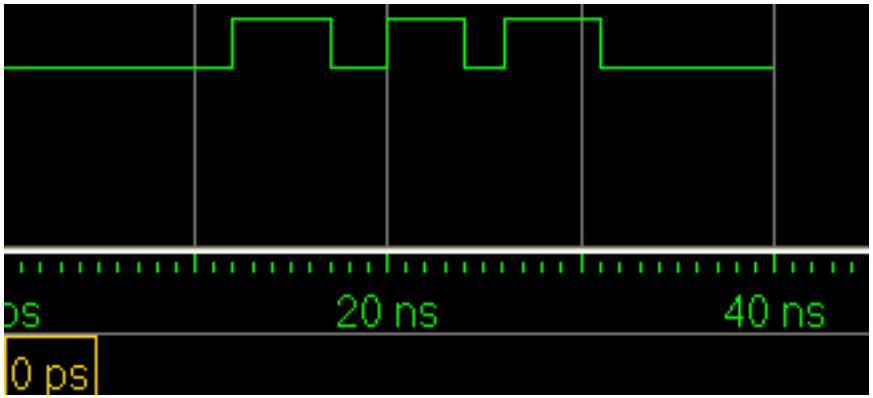


- 并行语句块的行为特性
  - 并行块中的语句并发执行
  - 各语句的延时均以语句块的开始时间为基准
    - 顺序语句块——语句的延时总是相对于前面的语句的完成时间
  - 各语句中的延时和事件控制决定语句的执行顺序
- 语法
  - **fork** [: block_id(declarations)]
  - 过程语句;
  - 过程语句;
  - …
  - 过程语句;
  - **join**

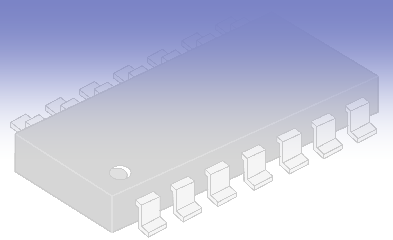
并行语句块举例



- 例
  - fork
    - # 0 stream = 0;
    - #12 stream = 1;
    - #17 stream = 0;
    - #20 stream = 1;
    - #24 stream = 0;
    - #26 stream = 1;
    - #31 stream = 0;
  - join



嵌套块



- 顺序块和并行块可以互相嵌套混合使用
- 例：

```
initial
begin
    pm_enable = 1'b0;
    #1 pm_enable = 1'b1;
end

initial
begin : seq_a
        #4 pm_write = 6'd5;
        fork :  par_a  // 语句块的开始时间?
             #6 pm_select = 4'd7;
             begin : seq_b
                   wdog_rst = pm_enable;
                   #2 wdog_intr = wdog_rst;
             end
             # 2 frc_sel = 4'd3;
             # 4 pm_iter = 4'd2;
             # 8 itop = 4'd4;
        join
        # 8 pm_lock = 1'b1;
        # 2 pcell_id = 6'd52;
        # 6 $stop;
end // 语句块的结束时间?
```

顺序语句块和并行语句块的混合

```
# t= 0, pm_enable=0, wdog_rst=x,wdog_intr=x, pm_lock=x, pm_write= x, pcell_id= x, pm_select= x, pm_iter= x, frc_sel= x, itop= x
# t= 1, pm_enable=1, wdog_rst=x,wdog_intr=x, pm_lock=x, pm_write= x, pcell_id= x, pm_select= x, pm_iter= x, frc_sel= x, itop= x
# t= 4, pm_enable=1, wdog_rst=1,wdog_intr=x, pm_lock=x, pm_write= 5, pcell_id= x, pm_select= x, pm_iter= x, frc_sel= x, itop= x
# t= 6, pm_enable=1, wdog_rst=1,wdog_intr=1, pm_lock=x, pm_write= 5, pcell_id= x, pm_select= x, pm_iter= x, frc_sel= 3, itop= x
# t= 8, pm_enable=1, wdog_rst=1,wdog_intr=1, pm_lock=x, pm_write= 5, pcell_id= x, pm_select= x, pm_iter= 2, frc_sel= 3, itop= x
# t=10, pm_enable=1, wdog_rst=1,wdog_intr=1, pm_lock=x, pm_write= 5, pcell_id= x, pm_select= 7, pm_iter= 2, frc_sel= 3, itop= x
# t=12, pm_enable=1, wdog_rst=1,wdog_intr=1, pm_lock=x, pm_write= 5, pcell_id= x, pm_select= 7, pm_iter= 2, frc_sel= 3, itop= 4
# t=20, pm_enable=1, wdog_rst=1,wdog_intr=1, pm_lock=1, pm_write= 5, pcell_id= x, pm_select= 7, pm_iter= 2, frc_sel= 3, itop= 4
# t=22, pm_enable=1, wdog_rst=1,wdog_intr=1, pm_lock=1, pm_write= 5, pcell_id=52, pm_select= 7, pm_iter= 2, frc_sel= 3, itop= 4
```

```
initial
begin
    pm_enable = 1'b0;
    #1 pm_enable = 1'b1;
end

initial
begin : seq_a
        #4 pm_write = 6'd5;
        fork :  par_a  // 语句块的开始时间?
             #6 pm_select = 4'd7;
             begin : seq_b
                    wdog_rst = pm_enable;
                    #2 wdog_intr = wdog_rst;
             end
             # 2 frc_sel = 4'd3;
             # 4 pm_iter = 4'd2;
             # 8 itop = 4'd4;
        join
        # 8 pm_lock = 1'b1;
        # 2 pcell_id = 6'd52;
        # 6 $stop;
end // 语句块的结束时间?
```