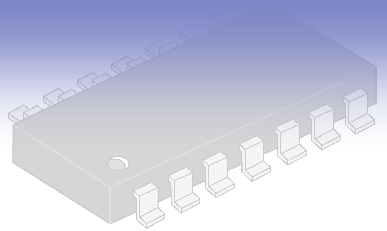
第 7 章

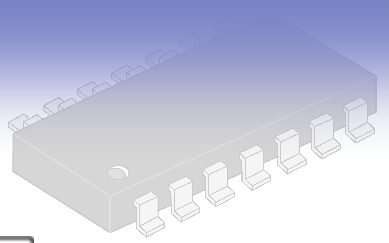
任务和函数

内容



- 任务和函数的区别
- 任务
- 函数

Verilog 模块的结构



module 模块名(标识符) (端口列表/端口声明列表);

端口声明;
参数声明;

wire, reg 及其它类型变量声明;

assign 数据流语句;

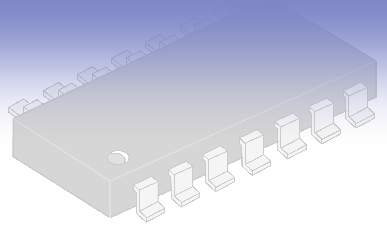
initial 块和 always块
所有行为语句均在这些块中;

低层模块实例语句;

任务和函数;

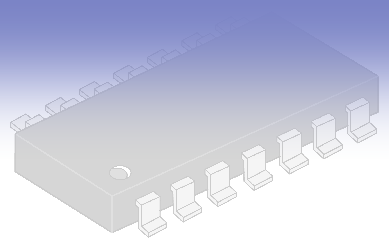
endmodule // 模块结束语句

任务/函数



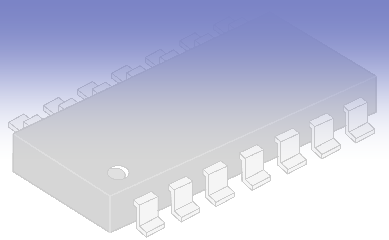
- 语句块➔任务/函数
- 使用任务/函数
  - 将一个大的模块中的多次用到的语句块和程序段设计成任务/函数
  - 将一个很大的模块分解成多个较小的任务和函数
  - 便于模块的理解、调试
- 任务和函数和命名块一样，可以通过层次名访问

任务和函数的区别



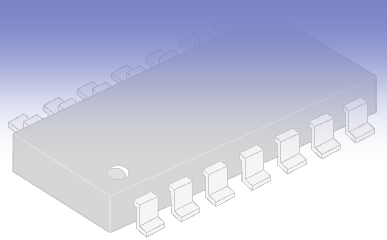
- 函数
  - 能调用另外一个函数，但不能调用另外一个任务
  - 总是在仿真时刻0就开始执行
  - 不能包含任何延时、事件控制或时序控制声明语句
  - 至少有一个输入变量
  - 不能有输出（output）和双向（inout）变量
  - **函数只能返回一个值**
- 任务
  - 可以调用另一个任务或函数
  - 可以在非0仿真时刻执行
  - 可以包含任何延时、事件控制或时序控制声明语句
  - 可以没有或有多个（input）、输出（output）和双向（inout）变量
  - 不返回任何值
  - 通过输出（output）和双向（inout）变量传递多个值

任务和函数的使用



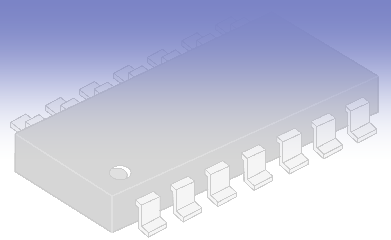
- 任务和函数必须在模块内进行定义
- 其作用范围仅局限于定义它们的模块
- 任务
  - 代替一段普通的、可能多次调用的代码块
- 函数
  - 代替纯组合逻辑代码，完成各种运算和转换

任务



- 类似一个过程
  - 将共同的代码段写成任务，可以在设计的不同位置调用
- 定义
  - **task** 任务名标识符; // 要有一个任务标识符
  - parameter_declaration;
  - input_declaration;
  - output_declaration;
  - inout_declaration;
  - register_declaration;
  - event_declaration;
  - statement;
  - **endtask**
  - 任务定义以关键字 task 开始，以 endtask 结束
  - 可以有一或多个输入/输出参数
  - 可以带时序、时间控制
  - 输出参数的值直到任务退出时，才传给调用参数
- 调用
  - 任务名 [(参数1,参数2,参数3, ... ,参数n)];
  - 调用语句中的参数表必须与定义中的输入/输出参数顺序相匹配
  - 任务调用语句是过程语句，在 always 和 initial 中使用
  - 任务调用中的**输出/双向（输入输出）参数**必须是寄存器类型

例、简单任务

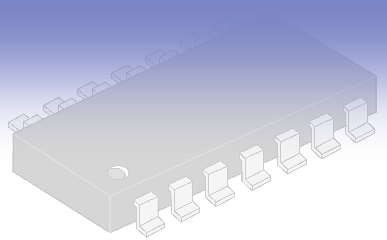


- 任务定义
  - **task** first_task;
  - **parameter** size = 4;
  - **input integer** a;
  - **inout** [size-1:0] b;
  - **output** c;
  - **reg** [size-1:0] d;
  - **event** e;
  - **begin**
  - d = b;
  - c = |d;
  - b = ~b;
  - **if** (!a) -> e;
  - **end**
  - **endtask**

- 任务调用
  - **integer** x;
  - **reg** a, b, y;
  - **reg** [3:0] z;
  - **reg** [7:0] w;
  - first_task(x, z, y);
  - first_task(x, w[7:4], w[1]);
  - first_task(1, {a, b, w[3], x[0]}, y);

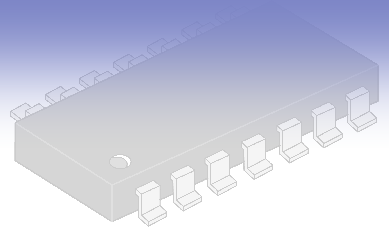
- 说明
  - 调用参数
    - 第 1 个变量是整数表达式
    - 第 2 个变量是 4-bit 寄存器类型变量
    - 第 3 个变量是 1-bit 寄存器类型变量
  - 调用参数的顺序为在 task 定义中输入/输出的出现顺序

自动（可重入）任务



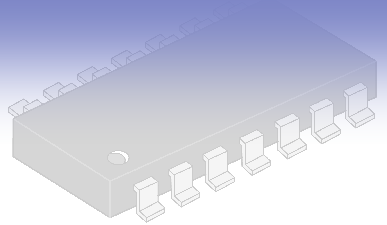
- 任务中声明的变量的地址空间
  - 在任务中声明的变量的地址空间是静态分配的
  - 如果一个在模块中的两个地方被同时调用，则两次调用任务将对同一块地址空间进行操作
- 使用自动任务避免这个问题
  - 通过使用关键字 automatic 声明自动任务
  - 每次调用都动态分配存储空间，每个任务调用都对各自独立的地址空间进行操作

应用任务举例



```
module top;
reg [15:0] cd_xor, ef_xor;
reg [15:0] c, d, e, f;
...
task automatic bitwise_xor;
output [15:0] ab_xor;
input [15:0] a, b;
begin
    #delay ab_and = a & b;
    ab_or = a | b;
    ab_xor = a ^ b;
end
endtask
...
always @(posedge clk)
    bitwise_xor(ef_xor, e, f);
...
always @(posedge clk2)
    bitwise_xor(cd_xor, c, d);
...
endmodule
```

C风格任务定义



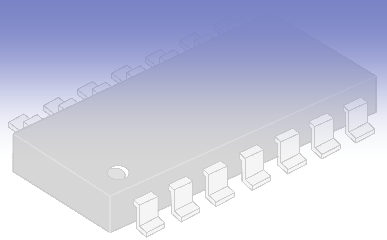
```
// 使用 ANSI C 风格的变量声明进行任务定义
task bitwise_op(output [15:0] ab_and, ab_or, ab_xor,
                input [15:0] a, b);
begin
    #delay ab_and = a & b;
    ab_or = a | b;
    ab_xor = a ^ b;
end
endtask
```

使用任务设计模块

- 例、

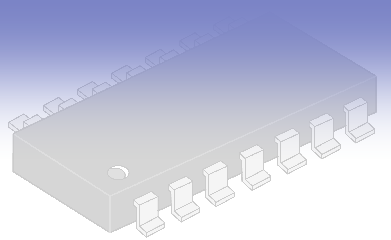
```
module adder #(parameter N=4) (
        output reg cout,
        output reg [N-1:0] sum,
        input [N-1:0] a, b,
        input cin,
        input clk, reset );

    always @(posedge clk) begin
        if (reset == 1'b1)
            {cout, sum} <= 0;
        else
            add_value(cout, sum, a, b, cin);
    end

    task add_value( output cout,
                    output [N-1:0] sum,
                    input [N-1:0] a, b,
                    input cin );
        begin
            {cout, sum} <= a + b + cin;   // <= ?
        end
    endtask
endmodule
```

函数



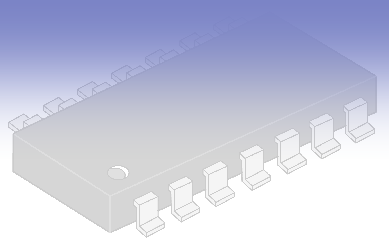
- 也是可以在模块不同位置执行的共同代码
- 定义
  - **function** [返回值的类型和范围] 函数名;
  - parameter_declaration;
  - input_declaration;
  - register_declaration;
  - event_declaration;
  - 过程语句;
  - **endfunction**
  - 需要指定返回值的类型和范围，缺省为 1 位二进制数
  - **蕴涵声明了与函数同名的、函数内部寄存器**
  - 必须在函数的定义中对此寄存器进行赋值
- 调用
  - **函数名** (参数1,参数2,参数3, ... ,参数n);
    - 将函数作为表达式中的操作数使用
    - 不能包含时序控制语句
    - 至少要有一个 input 变量
    - 不能有 output 或 inout 变量
    - 只能在一个模块内部声明

例、一些简单函数



- 例1、 返回16 bit 值
  - **function** [15:0] negation;
  - **input** [15:0] a;
  - **negation** = ~a;
  - **endfunction**
- 例2、返回实数值
  - **function** real multiply;
  - **input** a, b;
  - **real** a, b;
  - **multiply** = ((1.2 * a) * (b * 0.17)) * 5.1;
  - **endfunction**
- 例3、函数的调用
  - **real** a;
  - **wire** [15:0] b;
  - **wire** c, d, e, f;
  - **assign** b = **negation** ({4{c, d, e, f}});
  - **initial**
  - **begin**
  - a = **multiply**(1.5, a);
  - **$display**(“b=%b ~b=%b”, b, negation(b));
  - **end**

例、奇偶校验计算

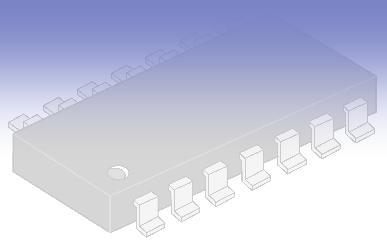


```
// 定义一个模块，其中包含计算奇偶校验位函数（calc_parity）
module parity;
...
reg [31:0] addr;
reg parity;

// 每当地址值发生变化，计算新的奇偶校验位
always @(addr)
begin
        parity = calc_parity(addr); // 第一次引用 Calc_parity
        // 第二次引用 calc_parity
        $display("Parity calculated = %b", calc_parity(addr));
end
...
// 定义计算奇偶校验函数
function calc_parity;
    input [31:0] address;
    // 使用隐含的内部寄存器 calc_parity.
    calc_parity = ^address; // 返回所有位的异或值（异或规约运算）
endfunction
...
endmodule
```

- C风格函数定义

```
// 采用 ANSI C 风格定义计算奇偶校验函数
function calc_parity (input [31:0] address);
    // 使用隐含的内部寄存器 calc_parity.
     calc_parity = ^address; // 返回所有位的异或值（异或规约运算）
endfunction
```

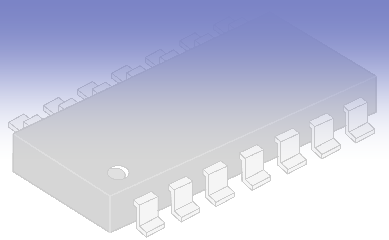
自动（递归）函数



- 函数中局部变量是静态分配的
  - 每次调用都对对同一块地址空间进行操作
- 使用自动函数
  - 通过使用关键字 automatic 声明自动函数
  - 每次调用都动态分配属于这次调用的存储空间
- 计算阶乘

```
// 用函数递归调用执行阶乘运算
module top;
...
// 定义函数
function automatic [31:0] factorial( input [31:0] operand );
    factorial = (operand >= 2) ? factorial (operand -1) * operand : 1 ;
endfunction

// 调用函数
reg [31:0] result;
initial
begin
     result = factorial(4); // 调用4的阶乘
     $display("Factorial of 4 is %0d", result); // 显示 24
end
...
endmodule
```

常数函数（constant function）



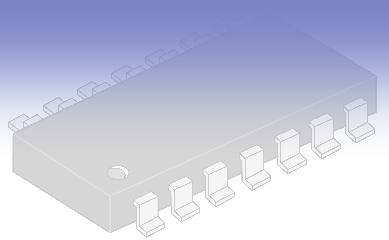
- 实现函数：$\log_2(N)$

```
module decoderlogN #(parameter N=8) (
    output reg [N-1:0] y,
    input [clog2(N)-1:0] a );    // 2^n = N -->  log2(N)==n

    function integer clog2(input integer n);
    begin
        clog2 = 0;
        n--;
        while ( n>0 ) begin
            clog2 = clog2 + 1;
            n = n>>1;
        end
    end
    endfunction

    always @*
        y = 1'b1 << a;
endmodule
```

A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

3-8译码器（3-to-8 Decoder）真值表

系统任务/函数



- 显示任务 —— 信息显示和输出
  - 显示和输出
  - 监控
- 显示和输出
  - 语法格式
    - 任务名 (格式说明1，输出变量列表1，... ，格式说明n，输出变量列表n);
  - 任务名
    - $display, $displayb, $displayh, $displayo
      - 输出后自动换行
    - $write, $writeb, $writeh, $writeo,
      - 没有自动换行
  - 没有格式说明的情况下，默认格式
    - 无后缀为十进制
    - 以后缀标志的格式输出

常用输出格式

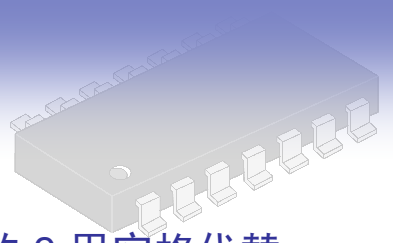
常用输出格式

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	输出线网型数据的信号强度
%m 或 %M	输出层次名
%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	以科学表示法输出实数
%f 或 %F	以十进制的形式输出实数
%g 或 %G	以指数/十进制的形式输出实数，无论何种格式均以较短的结果输出

显示特殊字符

输出格式	说　明
\n	换行
\t	制表符（tab）
\\	反斜线
\"	双引号
\o	1～3 位八进制数表示的字符
%%	百分号%

例、显示输出

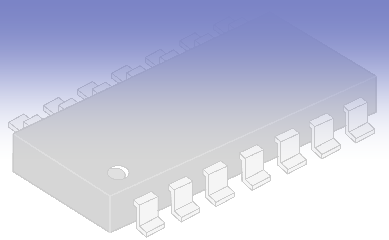


- 格式转换后：用表达式最大可能的值所占的位数显示当前值，十进制前的 0 用空格代替，其它左边补 0

```
`timescale 1ns/1ns
module display_test;
    reg [15 : 0] rval = 87;
    reg pin_a = 1'b1;
    initial
    begin
        $display("rval = %h(hex) = %d(decimal)", rval, rval);
        #2 $display("rval = %O(octal) = %B(binary)", rval, rval);
        #2 $display("rval has %c ASCII character value", rval);
        #2 $display("pin_a strength value is %v", pin_a);
        #2 $display("current scope is %m");
        #2 $display("%s is ASCII value for 87", 87);
        #2 $display("simulation time is %4t", $time);
        $write("\n");
        $write("\\\t%% <----> ");
        $write("\"\115\"\n");
    end
endmodule
```

```
# rval = 0057(hex) =    87(decimal)
# rval = 000127(octal) = 0000000001010111(binary)
# rval has W ASCII character value
# pin_a strength value is St1
# current scope is display_test
#    W is ASCII value for 87
# simulation time is   12
#
# \     % <----> "M"
```

$display 的显示输出



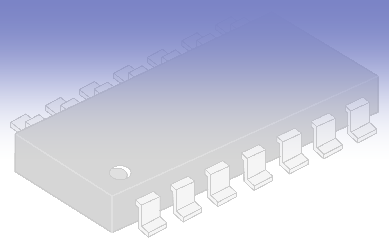
- 显示宽度
  - 自动按照输出格式进行调整
  - 用表达式的最大可能值所占的位数显示当前值
  - 在 % 与表示进制的字符之间加 0，可调整显示输出数据的宽度

```
1 module adjust_w;
2     reg [11 : 0] ival;
3     initial
4     begin
5         ival = 10;
6         $display("Printing with maximum size = %dd <==> %hh", ival, ival);
7         $display("Printing with maximum size = %0dd <==> %0hh", ival, ival);
8     end
9 endmodule
```

模块运行后的输出结果：

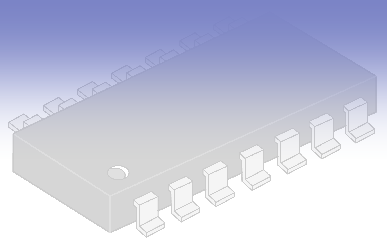
```
# Printing with maximum size =  10d <==> 00ah
# Printing with maximum size = 10d <==> ah
```

监控



- 监控指定的参数
  - 只要参数表中的变量和表达式的值发生变化，则显示整个参数表的值
- 语法格式
  - $monitor(格式说明1，变量列表1，... ，格式说明n，变量列表n);
  - $monitor;
  - $monitoron;
  - $monitoroff;
- 说明
  - $monitoron、$monitoroff 用于打开（启动）和关闭监控任务
  - 默认的情况下，在仿真的的起始时刻，监控任务就已启动
  - 在多模块的调试的情况下，任何时刻只能有一个 $monitor 起作用
    - 需要对特定模块打开和关闭监控任务
  - 在$monitor 中的格式说明与 $display 的相同

仿真时间系统函数



- 得到当前仿真时间
  - $time
    - 返回64位整数表示的当前仿真时刻
    - 显示时刻的单位由时间尺度决定
    - 总是输出整数
  - $realtime
    - 返回一个实数型的时间数值
    - 显示时刻的单位由时间尺度决定

例、使用系统任务和函数 $monitor 和 $time

```
1 `timescale 1ns / 1ns
2 `include "dflipflop.v"
3 module dff_trial;
4     reg p_d, p_clk, p_clrn, p_ena;
5     reg c = 1'b0;
6     initial
7     begin
8         #3 p_ena = 1;
9         #5 p_clrn = 1;
10    end
11    initial
12    begin
13        p_clk = 0;
14        forever
15        begin
16            p_clk = #6  1;
17            p_clk = #4  0;
18            c =  ~c;
19            p_d = c;
20
21        end
22    end
23    initial
24    begin
25        $monitor ("At  %0tns,    D = %d, Clk = %d,",$time,p_d, p_clk,
26                  " and Q is %b, Clr = %d, Ena = %d", p_q, p_clrn, p_ena);
27    end
28
29    wire p_q;
30    dflipflop dff_instane( .q_out(p_q), .d(p_d), .clk(p_clk), .clrn(p_clrn), .ena(p_ena) );
31 endmodule
```

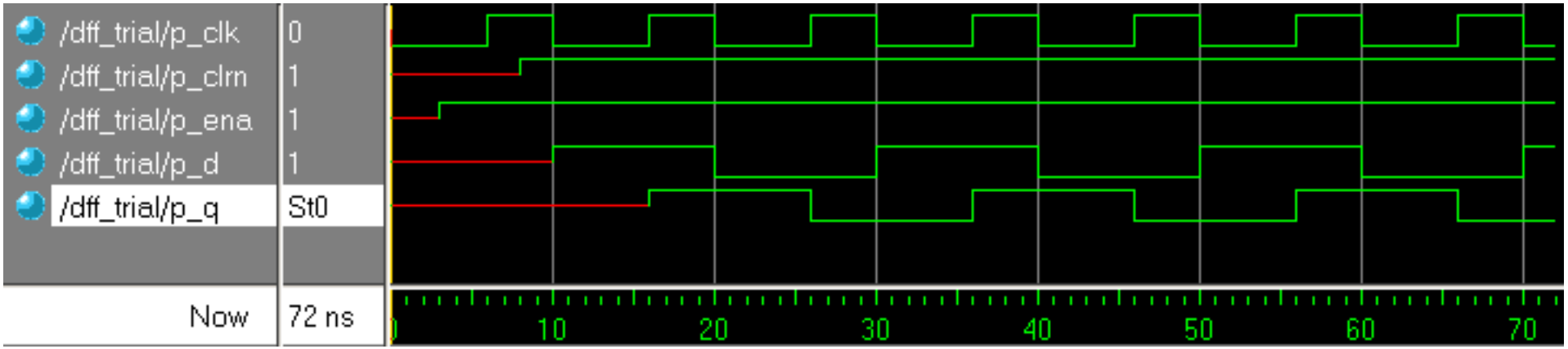
```
1 module dflipflop ( q_out, d, clk, clrn, ena);
2     output q_out;
3     input d, clk, clrn, ena;
4     reg q;
5     always @ (posedge clk, negedge clrn)
6     begin
7         if ( clrn == 1'b0)
8         begin
9             q <= 'b0;
10        end
11        else if ( clk == 1'b1)
12        begin
13            if ( ena == 1'b1)
14                q <= d;
15            else
16                q <= q;
17        end
18    end
19
20    assign q_out = q;
21 endmodule
```

使用系统任务和函数模块的仿真结果

系统任务 $monitor 的输出：

```
# At 0ns,   D = x, Clk = 0, and Q is x, Clr = x, Ena = x
# At 3ns,   D = x, Clk = 0, and Q is x, Clr = x, Ena = 1
# At 6ns,   D = x, Clk = 1, and Q is x, Clr = x, Ena = 1
# At 8ns,   D = x, Clk = 1, and Q is x, Clr = 1, Ena = 1
# At 10ns,  D = 1, Clk = 0, and Q is x, Clr = 1, Ena = 1
# At 16ns,  D = 1, Clk = 1, and Q is 1, Clr = 1, Ena = 1
# At 20ns,  D = 0, Clk = 0, and Q is 1, Clr = 1, Ena = 1
# At 26ns,  D = 0, Clk = 1, and Q is 0, Clr = 1, Ena = 1
# At 30ns,  D = 1, Clk = 0, and Q is 0, Clr = 1, Ena = 1
# At 36ns,  D = 1, Clk = 1, and Q is 1, Clr = 1, Ena = 1
# At 40ns,  D = 0, Clk = 0, and Q is 1, Clr = 1, Ena = 1
# At 46ns,  D = 0, Clk = 1, and Q is 0, Clr = 1, Ena = 1
# At 50ns,  D = 1, Clk = 0, and Q is 0, Clr = 1, Ena = 1
# At 56ns,  D = 1, Clk = 1, and Q is 1, Clr = 1, Ena = 1
# At 60ns,  D = 0, Clk = 0, and Q is 1, Clr = 1, Ena = 1
# At 66ns,  D = 0, Clk = 1, and Q is 0, Clr = 1, Ena = 1
# At 70ns,  D = 1, Clk = 0, and Q is 0, Clr = 1, Ena = 1
```

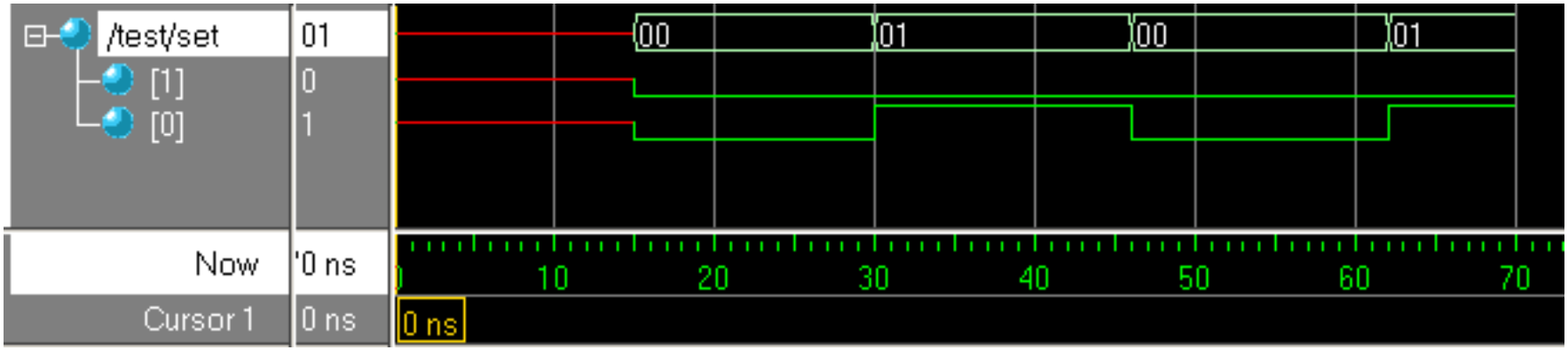
对应的仿真波形输出：



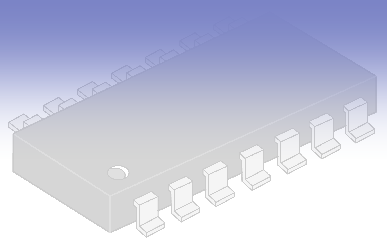
例、使用系统任务 $realtime

```
1 `timescale 10ns / 1ns
2 module test;
3     reg [1 : 0] set;
4     parameter DELAY0 = 1.525;
5     parameter DELAY1 = 1.55;
6     initial
7     begin
8         $monitor("At %0tns,", $realtime, " set = ", set);
9         #DELAY0 set = 1'b00;
10        #DELAY0 set = 1'b01;
11        #DELAY1 set = 1'b10;
12        #DELAY1 set = 1'b11;
13    end
14 endmodule
```

```
# At 0ns, set = x
# At 15ns, set = 0
# At 30ns, set = 1
# At 46ns, set = 0
# At 62ns, set = 1
```



文件输出任务

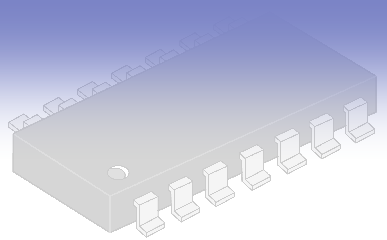


- 文件的打开和关闭
- 系统函数 $fopen 用于打开一个文件
  - 用法
    - integer 文件指针 = $fopen(file_name);
    - 系统函数 $fopen 返回一个关于文件的无符号整数文件指针
- 系统函数 $fclose 关闭一个打开的文件
  - 用法
    - $fclose (文件指针);
- 输出到文件
  - 有与显示/监控系统任务相对应的向文件输出系统任务
  - $fdisplay(文件指针，...);
  - $write(文件指针，...);
  - $monitor(文件指针，...);
    - 所有这些任务的第一个参数是文件指针

例、使用文件输出

```
1 `timescale 10ns / 1ns
2 module file_test;
3     reg [1 : 0] set;
4     parameter DELAY0 = 1.525;
5     parameter DELAY1 = 1.55;
6
7     integer file;
8
9     initial
10    begin
11
12        file = $fopen("trial.txt");
13
14        $fmonitor(file, "At %0tns,", $realtime, " set = ", set);
15
16        #DELAY0 set = 1'b00;
17        #DELAY0 set = 1'b01;
18        #DELAY1 set = 1'b10;
19        #DELAY1 set = 1'b11;
20
21        $fclose( file );
22    end
23 endmodule
```

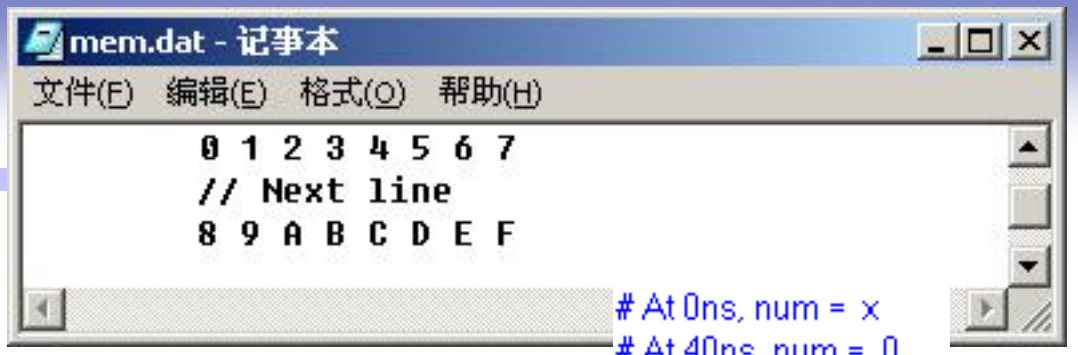
文件输入任务



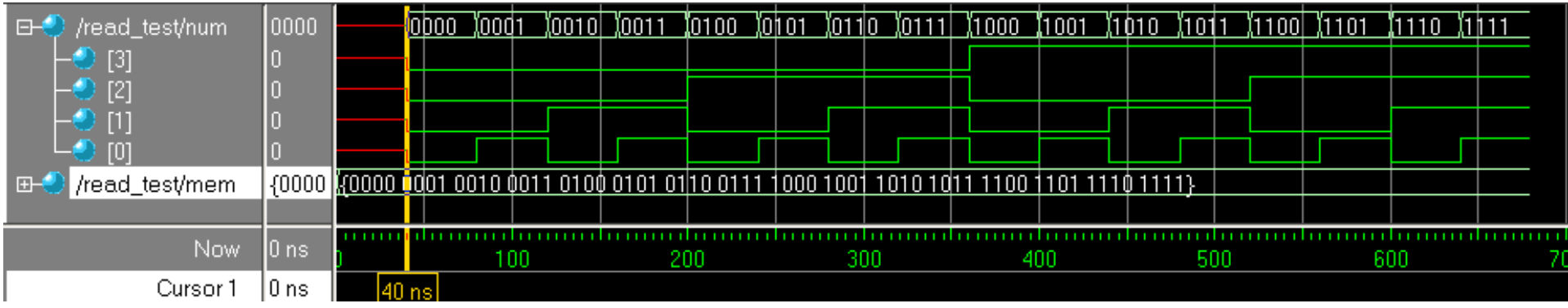
- 从文件中读取数据到存储器中
- 语法
  - $readmemb(“数据文件名”, memory_name, [起始地址, [结束地址]]
  - $readmemh(“数据文件名”, memory_name, [起始地址, [结束地址]]
- 数据文件
  - 可以包含
    - 空白（空格、换行、制表（tab））
    - 注释行
    - 二进制数
    - 十六进制数
  - 每个数据用空白字符（或注释行分开）
  - 数据文件中地址的说明方法：@ 后跟十六进制数表示
    - @hhhhhhhh

例、从文件输入数据

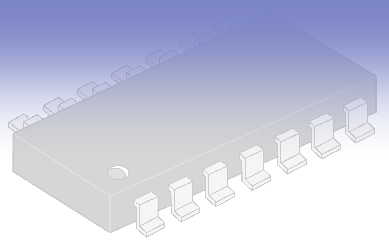
```
1 `timescale 10ns / 1ns
2
3 module read_test;
4
5     parameter BITS = 4;
6     parameter UNITS = 16;
7
8     reg [BITS -1 : 0] mem [0 : UNITS - 1];
9
10    integer index = 0;
11
12    reg [BITS -1 : 0] num;
13
14    initial
15    begin
16
17        $readmemh("mem.dat", mem);
18
19        $monitor("At %0tns,", $realtime, " num = ", num);
20
21        for ( index = 0; index < UNITS; index = index + 1)
22            #4   num = mem[index];
23    end
24 endmodule
```




```
# At 0ns, num = x
# At 40ns, num = 0
# At 80ns, num = 1
# At 120ns, num = 2
# At 160ns, num = 3
# At 200ns, num = 4
# At 240ns, num = 5
# At 280ns, num = 6
# At 320ns, num = 7
# At 360ns, num = 8
# At 400ns, num = 9
# At 440ns, num = 10
# At 480ns, num = 11
# At 520ns, num = 12
# At 560ns, num = 13
# At 600ns, num = 14
# At 640ns, num = 15
```

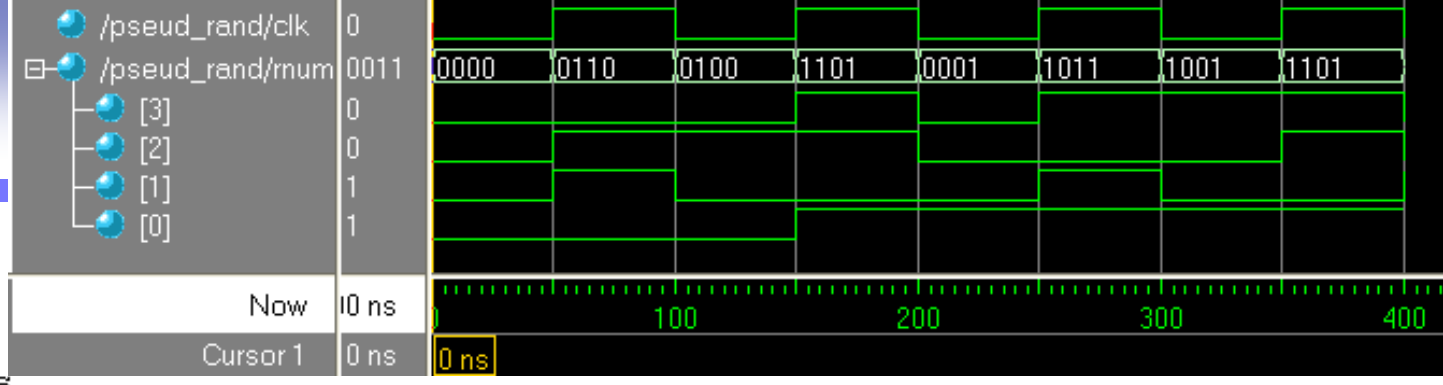


$random 函数



- 产生随机数
- 语法
  - $random [(seed)]
  - 根据种子变量(seed)的取值，返回一个32位有符号随机数
  - 种子变量控制函数的返回值
  - 种子变量的类型
    - 整数
    - 寄存器
    - 时间
  - 未指定种子时，使用缺省种子产生随机数

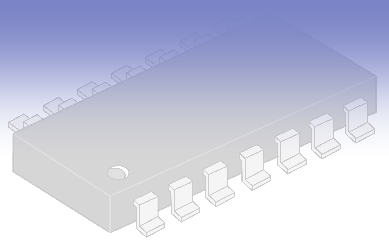
例、随机数




```
1 `timescale 1ns / 1ns
2
3 module pseud_rand;
4
5     reg [3 : 0 ] rnum;
6     reg clk;
7
8     integer seed = 10;
9
10    initial
11    begin
12        clk = 1'b0;
13        forever #50  clk = !clk;
14    end
15
16    always @(posedge clk)
17    begin
18        rnum = { $random(seed) } % 16;
19        $monitor ( "At %0tns, the rising edge of clock,    rnum = %b", $time, rnum );
20    end
21
22    always @(negedge clk)
23    begin
24        rnum = { $random(seed) } % 16;
25        $monitor ( "At %0tns, the falling edge of clock,    rnum = %b", $time, rnum );
26    end
27
28 endmodule
```

```
# At 0ns, the falling edge of clock,   rnum = 0000
# At 50ns, the rising edge of clock,   rnum = 0110
# At 100ns, the falling edge of clock,   rnum = 0100
# At 150ns, the rising edge of clock,   rnum = 1101
# At 200ns, the falling edge of clock,   rnum = 0001
# At 250ns, the rising edge of clock,   rnum = 1011
# At 300ns, the falling edge of clock,   rnum = 1001
# At 350ns, the rising edge of clock,   rnum = 1101
```

编译指令

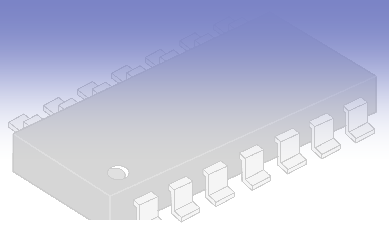


- 时间尺度
  - `timescale <时间单位> / <时间精度>
  - 时间单位
    - 定义延迟的时间单位
  - 时间精度
    - 定义仿真时间的精确程度
  - 时间精度不能大于时间单位
- 编译指令后面不加分号（；）

常用时间单位

时间单位	定义
s	秒（1s）
ms	10^{-3}s
μs	10^{-6}s
ns	10^{-9}s
ps	10^{-12}s
fs	10^{-15}s

例、时间单位和时间精度

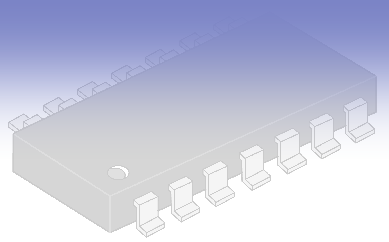


```
 1 `timescale 10ns / 1ns
 2 module test;
 3     reg [1 : 0] set;
 4     parameter DELAY0 = 1.525;
 5     parameter DELAY1 = 1.55;
 6     initial
 7     begin
 8         $monitor("At %0tns,", $realtime, " set = ", set);
 9         #DELAY0 set = 1'b00;
10         #DELAY0 set = 1'b01;
11         #DELAY1 set = 1'b10;
12         #DELAY1 set = 1'b11;
13     end
14 endmodule
```

```
# At 0ns, set = x
# At 15ns, set = 0
# At 30ns, set = 1
# At 46ns, set = 0
# At 62ns, set = 1
```

- `timescale 定义了本模块的
  - 时间单位为：10ns
  - 时间精度为：1ns
    - 所有延时的时间单位为 10ns，且精度取 1ns
- 仿真时刻的计算
  - 1.525×10 = 15.25，由于精度为 1ns，取整后为 15ns
  - 1.55×10 = 15.5，同样，取整后，为16ns

显示一个模块的时间单位和精度



- 使用系统任务
  - $printtimescale [(层次路径名)] ;
  - 说明
    - 没有使用变量调用时，显示当前模块的时间单位和精度
    - 如果指定层次路径名调用时，显示指定模块的时间单位和精度

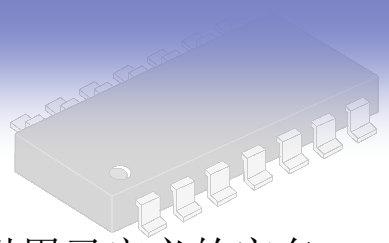
```
1 `include "m_and.v"
2
3 `timescale 100 ns / 10ns
4
5 module measure_time;
6     wire py, pa, pb;
7     m_and  m_instance(py, pa, pb);
8     initial
9     begin
10        $printtimescale;
11
12        $printtimescale(measure_time.m_instance);
13    end
14 endmodule
```

```
1 `timescale 1 ns / 100 ps
2
3 module m_and (y, a, b);
4   output y;
5   input a, b;
6   assign y = a & b;
7 endmodule
```

仿真运行输出结果：

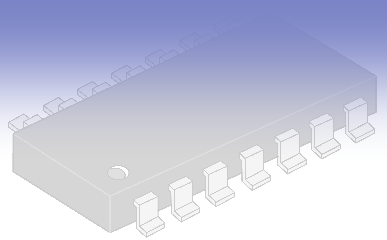
```
VSIM 5> run
# Time scale of (measure_time) is 100ns / 10ns
# Time scale of (measure_time.m_instance) is 1ns / 100ps
```

宏定义



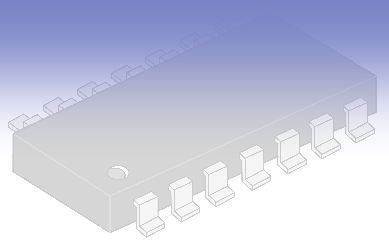
- 用一个指定的宏名（标识符）代表一个字符串（宏内容）
- 用法
  - `define　宏名　字符串
    - 编译时，把程序中在该命令后的所有的“宏标识符”都替换成相应的字符串
  - 例
    - `define　WORDSIZE　16
    - module　m_name;
    - 　　reg [`WORDSIZE – 1 : 0]　data;
    - 　　...
    - endmodule
  - 在模块定义内部和外部均可使用编译指令`define
  - 宏名的有效范围到源文件结束
  - 引用宏名时，必须在宏名之前加上符号“`”
  - 在宏定义时，可以引用已定义的宏名
  - 例
    - module macro_test;
    - 　reg　a, b, c;
    - 　wire　y_out;
    - `define　ab　a & b
    - `define　abc　`ab & c
    - 　assign　y_out = `abc;
    - endmodule
  - assign 语句展开后，为：
    - assign　y_out = a & b& c;

条件编译命令



- 对编译内容进行控制
  - 有选择地对源程序文件中的部分内容进行编译
  - 否则，对整个源程序文件中的全部进行编译
- 用法
  - `ifdef　宏名
  - 　　程序段 1
  - [`else
  - 　　程序段 2]
  - `endif
- 应用
  - 选择一个模块的不同代码
    - 如，选择不同的激励

文件包含



- 用于将另一个源文件的全部内容包含进来
  - `include “文件名”
  - 文件名
    - 被嵌入文件的路径名

```
1 `include "m_and.v"
2
3 `timescale 100 ns / 10ns
4
5 module measure_time;
6     wire py, pa, pb;
7     m_and  m_instance(py, pa, pb);
8     initial
9     begin
10        $printtimescale;
11
12        $printtimescale(measure_time.m_instance);
13    end
14 endmodule
```

```
1 `timescale 1 ns / 100 ps
2
3 module m_and (y, a, b);
4   output y;
5   input a, b;
6   assign y = a & b;
7 endmodule
```