上海交通大学
Shanghai Jiao Tong University

Verilog HDL 硬件描述语言（5）
高级设计话题

上海交通大学微电子学院
蒋剑飞

高级话题

- 存储器建模/文件读写
- 任务与函数/系统任务与函数
- 其它高级话题

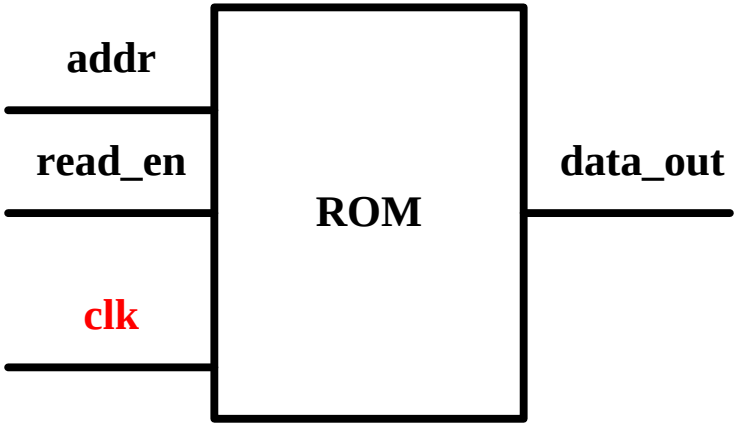
ROM的建模

- ROM是只读内存（Read-Only Memory）的简称，是一种只能读出事先所存数据的固态半导体存储器。其特性是一旦储存资料就无法再将之改变或删除。通常用在不需经常变更资料的电子或电脑系统中，资料并且不会因为电源关闭而消失。

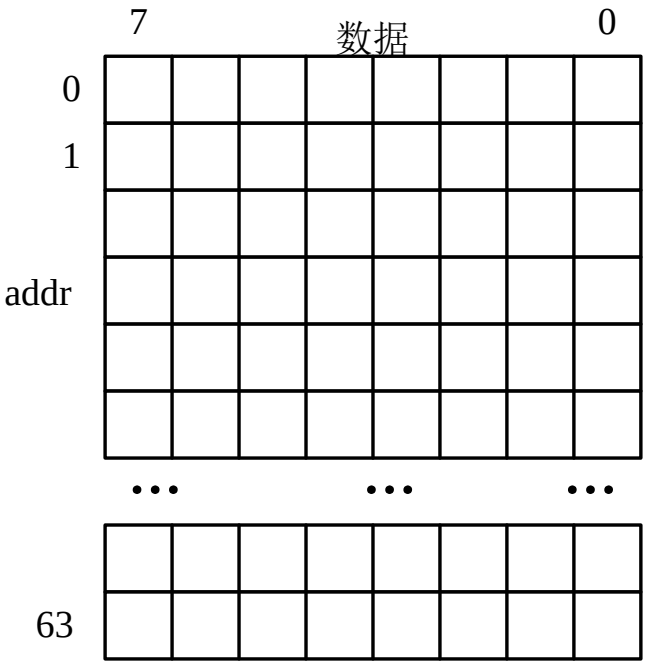
- ROM中一般存储启动程序、操作系统等固化软件。

ROM内容举例——TMS320C5402

ADDRESS RANGE	DESCRIPTION
F000h – F7FFh	Reserved
F800h – FBFFh	Bootloader
FC00h – FCFFh	μ-law expansion table
FD00h – FDFFh	A-law expansion table
FE00h – FEFFh	Sine look-up table
FF00h – FF7Fh	Reserved
FF80h – FFFFh	Interrupt vector table



存储器建模回顾



寄存器组可以用来建模ROM，RAM以及register file。

reg [msb:lsb] memory [upper1:lower1];

reg [7:0] memory1 [0:63]; //64个8位的存储器

reg mema [1:5]; //5个一位的存储器

reg [1:5] rega; //1个5位的寄存器

区别：对于存储器的访问须指定地址，不能实现同时的所有地址访问。

rega = 0; // Legal syntax

mema = 0; // illegal syntax

mema[1] = 0; //Assigns 0 to the first element of mema。

ROM建模举例

```
`timescale 1ns/10ps
module myrom(read_data,addr,read_en_);
input read_en_;
input [3:0] addr;
output [3:0] read_data;

reg [3:0] read_data;
reg [3:0] mem [0:15];

initial
  $readmemb("my_rom_data.txt",mem);

always @ (addr or read_en_ )
   if(!read_en_)
       read_data=mem[addr];
endmodule
```

my_rom_data.txt

```
0000
0101
1100
0011
1101
0010
0011
1111
1000
1001
1000
0001
1101
1010
0001
1101
```

testbench

```
`timescale 1ns/10ps
module test_myrom();
reg [3:0] addr;
reg read_en_;
wire [3:0] rd_data;
myrom i_rom(.addr(addr)
            ,.read_en_(read_en_)
            ,.read_data(rd_data));
initial
begin
    read_en_=1'b0;
    addr=4'd0;
#10 addr=4'd1;
#10 addr=4'd2;
#10 addr=4'd3;
#10 addr=4'd4;
#10 addr=4'd5;
#10 addr=4'd6;
end
```

my_rom_data.txt

```
0000
0101
1100
0011
1101
0010
0011
1111
1000
1001
1000
0001
1101
1010
0001
1101
```

用时钟同步的ROM模型

```
`timescale 1ns/10ps
module myrom(read_data,addr,read_en_,clk);
input read_en_,clk;
input [3:0] addr;
output [3:0] read_data;

reg [3:0] read_data;
reg [3:0] mem [0:15];

initial
  $readmemb(“my_rom_data.txt”,mem);

always @ (posedge clk)
   if(!read_en_)
       read_data=mem[addr];
endmodule
```

文件输入操作

- Verilog中有两个系统任务可以将数据文件读入寄存器组。一个读取二进制数据（$readmemb），另一个读取十六进制数据（$readmemh）。

- 语法

```
$readmemb (" file_name", <memory_name>);

$readmemh (" file_name", <memory_name>);
```

filename指定要调入的文件。
mem_name指定存储器名。

指定地址的文件读操作

- $readmemb

$readmemb ("file_name", <memory_name>, <start_addr>);

$readmemb ("file_name", <memory_name>, <start_addr>, <finish_addr>);

- $readmemh

$readmemh (" file_name", <memory_name>, <start_addr>);

$readmemh (" file_name", <memory_name>, <start_addr>, <finish_addr>);

start和finish决定存储器将被装载的地址。start为开始地址，finish为结束地址。如果不指定开始和结束地址，$readmem开始读入数据，从存储器的最低地址开始存放。

文件读操作举例（1）

```
`timescale 1ns/10ps
module myrom(read_data,addr,read_en_);
input read_en_;
input [3:0] addr;
output [3:0] read_data;

reg [3:0] read_data;
reg [3:0] mem [0:15];

initial
  $readmemb("my_rom_data.txt",mem,4'd2,4'd4);

always @ (addr or read_en_ or read_en_)
   if(!read_en_)
       read_data=mem[addr];
endmodule
```

mem
xxxx
xxxx
0000
0101
1100
xxxx
xxxx
xxxx
xxxx
xxxx
xxxx
xxxx
xxxx
xxxx
xxxx

my_rom_data.txt

0000
0101
1100
0011
1101
0010
0011
1111
1000
1001
1000
0001
1101
1010
0001
1101

文件格式说明

- 可以指定二进制（b）或十六进制（h）数
- 用下划线（_）提高可读性。
- 可以包含单行或多行注释。
- 可以用空格和换行区分存储器字。
- 可以给后面的值设定一个特定的地址，格式为：@(hex_address)
  - 十六进制地址的大小写不敏感。
  - 在@和数字之间不允许有空格。

文件读操作举例(2)

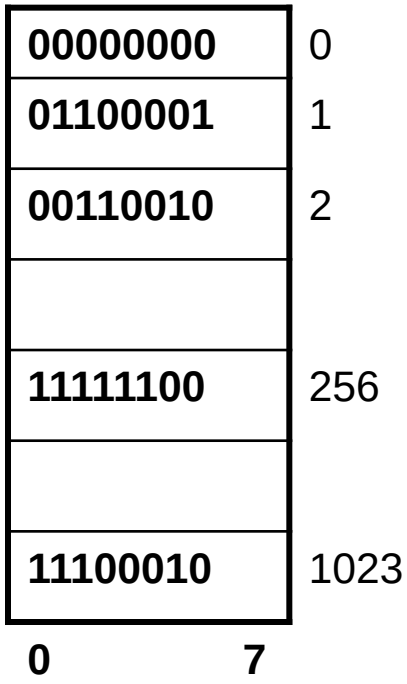
```
$readmemb("mem_file. txt", mema);
```

文本文件：**mem_file.txt**

```
0000_0000
0110_0001 0011_0010
// 地址3~255没有定义
@100 // hex
1111_1100
//地址257~1022没有定义
@3FF
1110_0010
```

声明的存储器组

reg [0:7] mema[0:1023]



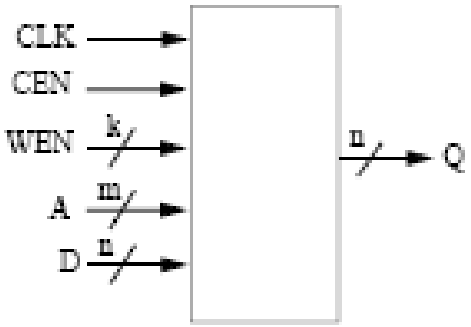

```
module readmem;
reg [0:7] mema [0:1023];
initial
$readmemb("mem_file.txt", mema);
endmodule
```

SRAM

- SRAM是英文Static RAM的缩写，它是一种具有静止存取功能的内存，不需要刷新电路即能保存它内部存储的数据。

- SRAM速度快，存储方便，但单位容量SRAM占用芯片的面积较大，一般作为快速存储设备如缓存等使用。

单口SRAM



Name	Type	Description
		Basic Pins
A[m-1:0]	Input	Addresses (A[0] = LSB)
D[n-1:0]	Input	Data inputs (D[0] = LSB)
CEN	Input	Chip Enable, active low
WEN [*]	Input	Write Enable, active low. *If word-write mask is enabled, this becomes a bus.
CLK	Input	Clock
Q[n-1:0]	Output	Data outputs (Q[0] = LSB)

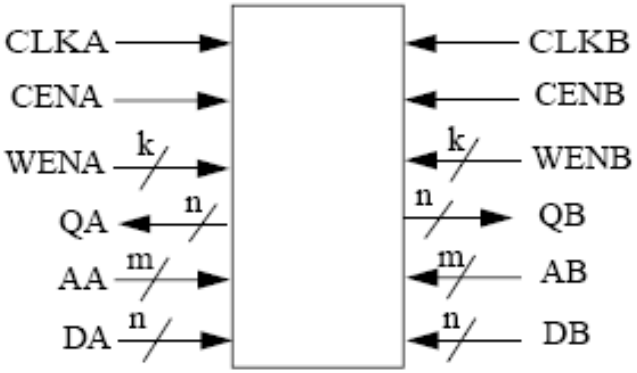
单口SRAM的Verilog模型

```
module single_port_sram(clk,cen,wen,addr,data_in,data_out);
input    clk,wen,cen;
input [15:0]      addr,data_in;
output [15:0] data_out;
reg [15:0]   data_out;
reg [15:0] s_p_sram [0:65535];

always @(posedge clk)
if(!wen && !cen)
   s_p_sram[addr]<=data_in;                 //write
else
   s_p_sram[addr]<=s_p_sram[addr];   //保持

always @(posedge clk)
if(!cen && wen)
   data_out<=s_p_sram[addr];                //read
else
   data_out<=16'bz;                                 //输出高阻
endmodule
```

双口SRAM



Name	Type	Description
Basic Pins		
AA[m-1:0]. AB[m-1:0]	Input	Addresses (AA[0] = LSB), (AB[0] = LSB)
DA[n-1:0], DB[n-1:0]	Input	Data inputs (DA[0] = LSB), (DB[0] = LSB)
CENA, CENB	Input	Chip Enables, active low
WENA[*], WENB[*]	Input	Write Enables, active low. *If word-write mask is enabled, this becomes a bus.
CLKA, CLKB	Input	Clocks
QA[n-1:0], QB[n-1:0]	Output	Data Outputs (QA[0] = LSB), (QB[0] = LSB)

双口SRAM举例

```
module dual_p_sram(clka,cena,wena,qa,aa,da,
                                clkb,cenb,wenb,qb,ab,db);
input    clka,cena,wena,clkb,cenb,wenb;
input [15:0]      aa,da,ab,db;
output [15:0] qa,qb;
reg [15:0] sram [8192:0];
reg  [15:0] qa,qb;
always @(posedge clka)          //porta read
if(!cena && wena)
   qa<=sram[aa]; else …
 always @(posedge clka)         //porta write
 if(!cena && !wena)
   sram[aa]<=da;  else…
 always @(posedge clkb)          //portb read
if(!cenb && wenb)
   qb<=sram[ab];  else …
 always @(posedge clkb)         //portb  write
 if(!cenb && !wenb)
   sram[ab]<=db;   else
   endmodule
```

其它常用的存储器

- SDRAM：同步动态随机存储器，同步是指Memory工作需要同步时钟，功能与时序复杂。
- DDRAM：Double Data Rate SDRAM （DDR SDRAM）双沿工作，时序尤其复杂。
- FLASH：(nand flash，nor flash）接口有多个标准，有复杂的，也有类似SRAM的接口。

文件输出

- $monitor,$display等系统任务可以将结果输出到标准输出设备，相似的系统任务（$fmonitor, $fdisplay)可以将结果输出到文件中。

- $fopen打开一个文件并返回一个多通道描述符（MCD）。
  - MCD是与文件唯一对应的32位无符号整数。
  - 如果文件不能打开并进行写操作，MCD将等于0。
  - 如果文件成功打开，MCD中的一位将被置位。

- 以$f开始的显示系统任务将输出写入与MCD相对应的文件中。

文件输出举例

```
. . .
integer MCD1;

…
MCD1 = $fopen("<name_of_file>");
$fdisplay( MCD1, P1, P2, .., Pn);
$fwrite( MCD1, P1, P2, .., Pn);
$fstrobe( MCD1, P1, P2, .., Pn);
$fmonitor( MCD1, P1, P2, .., Pn);
$fclose( MCD1);
```

```
`timescale 1ns/100ps
module inv_tb;
reg   a;
wire   b;
not inv(b,a);
initial   a=0;
always #5 a=~a;
integer file_handler;
initial
file_handler=$fopen("data_out.txt");
initial
$monitor("%t,%b,%b",$time,a,b);
initial
$fmonitor(file_handler,"%t,%b,%b",
$time,a,b);
//$fclose(file_handler);
endmodule
```

可以执行写文件的系统任务

- $fdisplay,$fdisplayb, $fdisplayh,$fdisplayo
- $fwrite, $fwriteb, $fwriteh, $fwriteo
- $fstrobe, $fstrobeb, $fstrobeh, $fstrobeo
- $fmonitor, $fmonitorb, $fmonitorh, $fmonitoro

任务(task)与函数(function)

- 通过把代码分成小的模块或者使用任务和函数，可把电路功能分成许多较小的、易于管理的部分，从而提高代码的可读性、可维护性和可重用性。

- 任务：一般用于编写测试模块，或者行为描述的模块。

- 函数：一般用于计算，或者用来代替组合逻辑。

Verilog中的任务

- 任务定义

```
task task_identifier ;
{ task_item_declaration
input …
output …
inout …
数据类型定义
 }
statement_or_null
功能描述
endtask
```

端口与数据类型的声明语法与module相同，但不能声明线网类型，port声明并非必要，task可以不包含任何port声明。

- 任务调用（必须在initial或always过程块中调用）

```
task_identifier ( expression { , expression } )
```

任务

- 任务一般在调用它的模块内部定义。也可以在单独的文件中定义，这时需要在调用它的模块文件中通过编译指导`include包括进来。
- 任务可以用来建模时序逻辑，也可以用来建模组合逻辑。
- 模块的端口可以使任意数目（包括0）。
- 其中可以包含时间控制（如：# delays, @, wait)。
- 任务可以接受和驱动全局变量（module范围内），当任务中使用局部变量时，局部变量在任务的最后需要传递给输出端口才能输出。
- 任务中定义的变量属于任务的局部变量。任务中定义的端口顺序决定了任务在被调用时全局变量的位置顺序。
- 任务可以调用任务或者函数。
- 任务必须在过程块中被调用。

task应用举例（1）

```
task my_task;
input a, b;
inout c;
output d, e;
begin
// statements that perform the work of the task
. . .
c = foo1;
d = foo2;
e = foo3;
end
endtask
```

```
my_task (v, w, x, y, z);
```

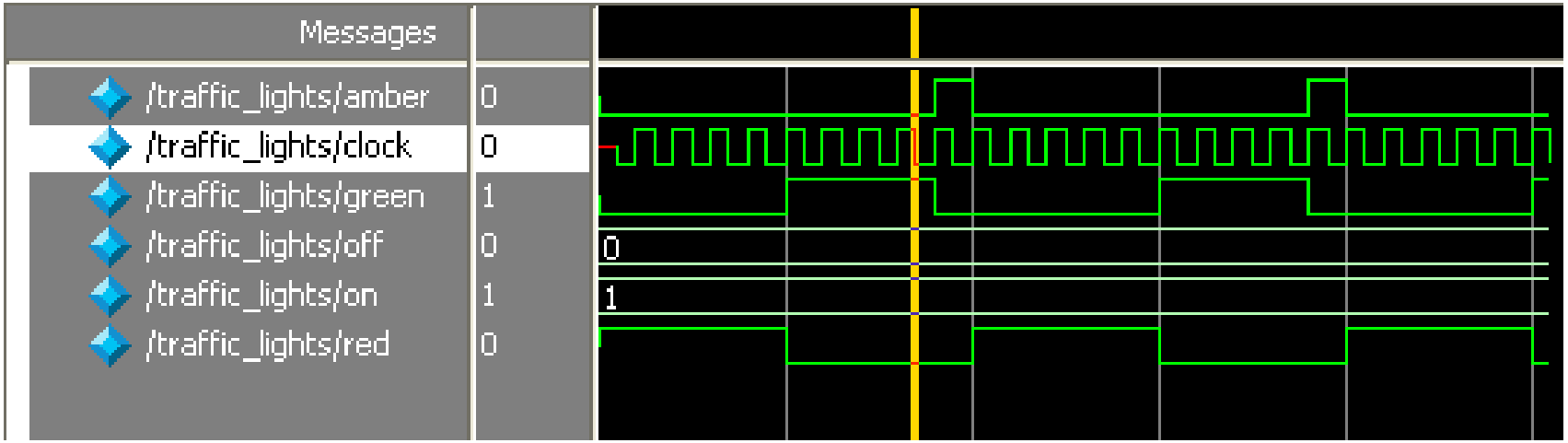
a = v;	x = c;
b = w;	y = d;
c = x;	z = e;

任务使用举例（2）

```
module traffic_lights;
reg clock, red, amber, green;
parameter on = 1, off = 0,
red_tics = 5,
amber_tics = 1, green_tics = 4;
// initialize colors.
initial red = off;
initial amber = off;
initial green = off;
always begin  // sequence to control the
lights.
red = on;                 // turn red light on
light(red, red_tics);  // and wait.
green = on;              // turn green light on
light(green, green_tics); // and wait.
amber = on;              // turn amber light on
light(amber, amber_tics); // and wait.
end
```

```
// task to wait for 'tics' positive
edge clocks
// before turning 'color' light off.
task light;
output color;
input [31:0] tics;
begin
repeat (tics) @ (posedge clock);
color = off;      // turn light off.
end
endtask
always begin // waveform for the
clock.
#100 clock = 0;
#100 clock = 1;
end
endmodule // traffic_lights.
```

仿真波形




```
repeat (tics) @ (posedge clock);
color = off;      // turn light off.
```

函数定义

```
function [ 宽度与类型 ] function_identifier ;
function_item_declaration { function_item_declaration }
statement
endfunction
```

```
function [7:0] getbyte;
input [15:0] address;
begin
getbyte = result_expression;
end
endfunction
```

函数语法

- 一般在调用它的模块内部定义。也可以在单独的文件中定义，这时需要在调用它的模块文件中通过编译指导`include包括进来。
- 宽度与类型可以不定义，这时**函数使用缺省的值：1bit reg类型**，函数必须有至少一个的输入，但不能含有任何输出。
- 函数定义不能包含任何定时控制语句。
- 函数中定义的变量属于函数的局部变量。函数中定义的端口顺序决定了函数在被调用时全局变量的位置顺序。
- 一个函数只能返回一个值，该值的变量名与函数同名，数据类型默认为reg类型。
- 函数可以接受和驱动全局变量（module范围内），当函数中使用局部变量时，局部变量的输出需要传递给返回值。
- 函数不能调用任务，但任务可以调用函数。

函数调用

- 函数在调用时传递给函数的变量顺序与函数输入端口的声明顺序相同。

- 函数可以在持续赋值语句或者过程中调用。

函数应用举例（1）

```
module foo(loo);
input [7:0] loo;

wire [7:0] goo = zero_count(loo);  //在持续赋值中调用函数

function [3:0] zero_count;          //函数定义
input [7:0] in_bus;
integer i;
begin
zero_count = 0;
for (i=0; i<8; i= i+1)
if (!in_bus[i])
zero_count = zero_count +1;
end
endfunction

endmodule
```

函数应用举例（2）

```
module test_function();
parameter MAX_BITS =8;
reg [MAX_BITS-1:0] d;
reg clk;
function [MAX_BITS-1:0] reverse_bits;
input [7:0] data;
integer K;
for(K=0; K< MAX_BITS; K=K+1)
reverse_bits[MAX_BITS-(K+1)] = data[K];
endfunction
always @ (posedge clk)
    d<= reverse_bits(d);
initial begin
    clk=0;
    d=8'b10101010;
    $monitor("%b",d);
end
always #5 clk=!clk;
initial
endmodule
```

```
# 10101010
# 01010101
# 10101010
# 01010101
# 10101010
# 01010101
# 10101010
# 01010101
```

函数应用举例(3)

```
module tryfact;
function [31:0] factorial; // define the function
input [3:0] operand;
reg [3:0] i;
begin
factorial = 1;
for (i = 2; i <= operand; i = i + 1)
factorial = i * factorial;
end
endfunction
// test the function
integer result,n;
initial begin
for (n = 0; n <= 7; n = n+1) begin
result = factorial(n);
$display("%0d factorial=%0d", n, result);
end
end
endmodule // tryfact
```

```
# 0 factorial=1
# 1 factorial=1
# 2 factorial=2
# 3 factorial=6
# 4 factorial=24
# 5 factorial=120
# 6 factorial=720
# 7 factorial=5040
```

函数与任务回顾

- 任务（task)
  - 通常用于调试，或对硬件进行行为描述
  - 可以包含时序控制（#延迟，@, wait)
  - 可以有 input，output，和inout参数
  - 可以调用其他任务或函数
- 函数
  - 通常用于计算，或描述组合逻辑
  - 不能包含任何延迟，函数仿真时间为0
  - 只含有input参数并由函数名返回一个结果
  - 可以调用其他函数，但不能调用任务

任务与函数特点

- 任务和函数必须在module内调用
- 在任务和函数中不能声明线网类型
- 所有输入/输出都是局部寄存器
- 任务/函数执行完成后才返回结果。例如，若任务/函数中有forever语句，则永远不会返回结果

系统任务与函数

- Verilog读取当前仿真时间的系统函数

  $time

  $stime

  $realtime

- Verilog支持文本输出的系统任务

  $display

  $strobe

  $write

  $monitor

仿真时间函数

- $time,$realtime,和$stime函数返回当前仿真时间，这些函数的返回值使用调用模块中`timescale定义的时间单位。

$time返回一个64位整数时间值。

$stime返回一个32位整数时间值。

$realtime返回一个实数时间值。

$stime函数返回一个32位整数时间值。对大于$2^{32}$的时间，返回模$2^{32}$的值。使用它可以节省显示及打印空间。