第3章 模块的结构、数据类型、变量和基本运算符号

3.1 模块的结构

Verilog结构位于module和endmodule声明语句之间，每个Verilog程序包括4个主要部分：

端口定义

I/O说明

内部信号声明

功能定义

模块端口定义格式

module 模块名（口1，口2，口3，…）

引用模块的两种连接方法

（1）在引用时严格按模块定义的端口顺序来连接，不用标明原模块定义时规定的端口名

（2）在引用时用“.”符号，标明定义时规定的端口名

不必严格按端口顺序对应

I/O说明的格式

输入口　　input[范围];

输出口　　output [范围];

输入/输出口　　inout [范围];

I/O说明也可以写在端口声明里。

```
module  module_name(input in_port1,input in_port2,
output out_port1,output  out_port2);
```

```
module test_width(b,a);
input[6:5] a;
output[3:2] b;

  assign b=a;
endmodule
```

内部信号说明

reg[范围] 变量1，变量2…;

wire[范围] 变量1，变量2…;

模块中实现逻辑功能的3种方法

(1)assign

```
assign    c=a&b;
```

(2)用实例元件

```
and      #2   u1(q,a,b);
```

(3)用always块

assign语句是描述组合逻辑最常用的方法之一。

always块既可用于描述时序逻辑，又可用于组合逻辑。

Verilog语言要点

(1)在Verilog模块中所有过程块（如initial块、always块）、连续赋值语句、实例引用都是并行的

(2)它们表示的是一种通过变量名互相连接的关系

(3)在同一模块中这三者出现的先后次序没有关系

(4)只有连续赋值语句assign和实例引用语句可以独立于过程块而存在于模块的功能定义部分

D触发器

```
module new_dff(q,clk,d);
input clk,d;
output q;
reg q;

  always @(posedge clk)
    q<=d;
endmodule
```

D触发器（带异步清除端）

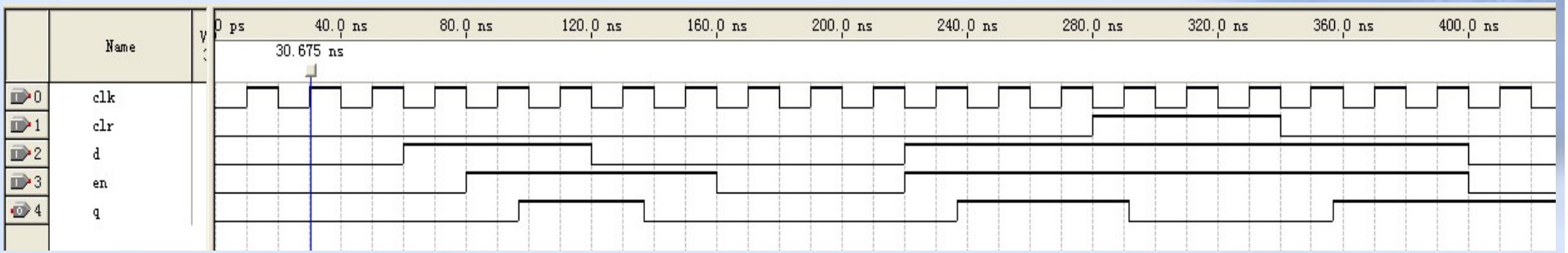
```
module new_dff2(q,clk,d,clr);
input clk,d,clr;
output q;
reg q;

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

D触发器（带异步清除端和使能端）

```
module new_dff3(q,clk,d,clr,en);
output q;
input clk,d,clr,en;
reg q;

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


Name
0 ps
40.0 ns
80.0 ns
120.0 ns
160.0 ns
200.0 ns
240.0 ns
280.0 ns
320.0 ns
360.0 ns
400.0 ns
30.675 ns
0
clk
1
clr
2
d
3
en
4
q

3.2 数据类型及其常量和变量

4种逻辑值

0　　1　　z(高阻)　　x（不定值）

常量

在程序运行过程中，其值不能被改变的量称为常量。

3.2.1 常量

1 数字

（1）整数

二进制整数　　b或B

十进制整数　　d或D

十六进制整数　　h或H

八进制整数　　o或0

数字表达方式

<位宽>’<进制><数字>

4’b1110　　//4位二进制数

12’habc　//12位十六进制数

16’d255　//16位十进制数

’<进制><数字>

采用默认位宽，与仿真器和使用的计算机有关（最小为32位）

‘hc3　//32位16进制数

‘o21　//32位8进制数

〈数字〉

默认为十进制数

采用默认位宽，与仿真器和使用的计算机有关（最小为32位）

326//32位十进制数

（2）x和z值

一个x可以用来定义十六进制数的四位二进制数的状态，八进制数的三位，二进制数的一位。z的表示方式同x类似。z还有一种表达方式是可以写作?。在使用case表达式时建议使用这种写法，以提高程序的可读性。

4'b10x0 //位宽为4的二进制数从低位数起第二位为不定值
4'b101z //位宽为4的二进制数从低位数起第一位为高阻值
12'dz //位宽为12的十进制数其值为高阻值
12'd? //位宽为12的十进制数其值为高阻值
8'h4x //位宽为8的十六进制数其低四位值为不定值

（3）负数

一个数字可以被定义为负数,只需在位宽表达式前加一个减号,减号必须写在数字定义表达式的最前面。注意减号不可以放在位宽和进制之间也不可以放在进制和具体的数之间。见下例:

-8’d5 //这个表达式代表5的补数（用八位二进制数表示）

8‘d-5 //非法格式

（4）下划线(underscore_)

下划线可以用来分隔开数的表达以提高程序可读性。但不可以用在位宽和进制处,只能用在具体的数字之间。

16'b1010_1011_1111_1010 //合法格式

8'b_0011_1010 //非法格式

当常量不说明位数时，默认值是32位，每个字母用8位的ASCII值表示。

例：

10＝32’d10＝32’b1010

1=32’d1=32’b1

-1=-32’d1=32’hFFFFFFFF

‘BX=32’ BX=32’ BXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

“AB”=16’B01000001_01000010 //字符串AB，

//为十六进制数16’h4142

2 参数(Parameter)型

在Verilog HDL中用parameter来定义常量,即用parameter来定义一个标识符代表一个常量,称为符号常量,即标识符形式的常量,采用标识符代表一个常量可提高程序的可读性和可维护性。parameter型数据是一种常数型的数据，其说明格式如下：

parameter 参数名1＝表达式，参数名2＝表达式，…， 参数名n＝表达式;

parameter是参数型数据的确认符，确认符后跟着一个用逗号分隔开的赋值语句表。在每一个赋值语句的右边必须是一个常数表达式。也就是说，该表达式只能包含数字或先前已定义过的参数。

```
parameter msb=7; //定义参数msb为常量7
parameter e=25, f=29; //定义二个常数参数
parameter r=5.7; //声明r为一个实型参数
parameter byte_size=8, byte_msb=byte_size-1; //用常数表达式赋值
parameter average_delay = (r+f)/2; //用常数表达式赋值
```

参数型常数经常用于定义延迟时间和变量宽度。

在实例引用时可通过参数传递改变在被引用模块中已定义的参数

```
module two_delay(a,b,c,d);
output c,d;
input a,b;
reg c,d;
parameter delay1=2,delay2=2;

  always @(a)
    c<= #delay1 a;

  always @(b)
    d<=#delay2 b;

endmodule
```

```
module top_delay2(a1,b1,c1,d1);
input a1,b1;
output c1,d1;

  two_delay #(5,10)
  u1(.a(a1),.b(b1),.c(c1),.d(d1));

endmodule
```

```
module top_delay(a1,b1,c1,d1);
input a1,b1;
output c1,d1;

  two_delay #(.delay1(5),.delay2(10))
  u1(.a(a1),.b(b1),.c(c1),.d(d1));

endmodule
```

```
module top_delay3(a1,b1,c1,d1);
input a1,b1;
output c1,d1;

   two_delay #(.delay2(10))
   u1(.a(a1),.b(b1),.c(c1),.d(d1));

endmodule

module top_delay4(a1,b1,c1,d1);
input a1,b1;
output c1,d1;

   two_delay #(10)          //参数被传给了delay1
   u1(.a(a1),.b(b1),.c(c1),.d(d1));

endmodule
```

defparam 可以在一个模块中改变另外一个模块的参数。

```
module block(b,a);
input a;
output b;
reg b;

parameter delay=0;

  always @(a)
    b<=#delay a;

endmodule
```

```
module block_up(a,b,c,d);
input a,b;
output c,d;
wire temp1,temp2;

  assign temp1=a&b;
  assign temp2=a;

  block u1(.b(c),.a(temp1));
  block u2(.b(d),.a(temp2));

endmodule
```

```
module test_block_up2;
  reg a_in,b_in;
  wire c_out,d_out;
  defparam
    test_block_up2.test_u.u1.delay=5,
    test_block_up2.test_u.u2.delay=10;

  block_up test_u(.a(a_in),.b(b_in),.c(c_out),.d(d_out));
  initial
  begin
    a_in=0;
    b_in=0;
    #20 a_in=1;
    #100 b_in=1;
    #100 a_in=0;
  end
endmodule
```

```
module Annotate;
  defparam
    test_block_up.test_u.u1.delay=5,
    test_block_up.test_u.u2.delay=10;
endmodule
```

```
module test_block_up;
  reg a_in,b_in;
  wire c_out,d_out;

  Annotate u123();

  block_up test_u(.a(a_in),.b(b_in),.c(c_out),.d(d_out));

  initial
  begin
    a_in=0;
    b_in=0;
    #20 a_in=1;
    #100 b_in=1;
    #100 a_in=0;
  end
endmodule
```

3.2.2 变量

1 网络型（线网，net ）

表示硬件单元之间的连接。

net不是关键字，代表了一组数据类型，包括wire,wand,wor,tri,triand,trior,trireg等。

线网的默认值为z(trireg例外，默认值为x)

本课程主要讨论wire

wire型数据常用来表示用于以assign关键字指定的组合逻辑信号。

Verilog程序模块中输入输出信号类型缺省时自动定义为wire型。

wire型信号可以用作任何方程式的输入，也可以用作“assign”语句或实例元件的输出。

格式：

```
wire [n-1:0] 数据名1,数据名2,…数据名i;
        //共有i条总线，每条总线内有n条线路

wire [n:1] 数据名1,数据名2,…数据名i;


wire a; //定义了一个一位的wire型数据

wire [7:0] b; //定义了一个八位的wire型数据

wire [4:1] c, d; //定义了二个四位的wire型数据
```

2寄存器型

通常表示一个存储数据的空间。

reg是最常用的寄存器型数据。

reg型变量并不严格对应于电路上的存储单元。

Verilog还支持integer,real和time寄存器数据类型。

reg的默认初始值是x。

格式:

```
reg [n-1:0] 数据名1,数据名2,… 数据名i;

reg [n:1] 数据名1,数据名2,… 数据名i;
```

```
reg rega; //定义了一个一位的名为rega的reg型数据

reg [3:0] regb; //定义了一个四位的名为regb的reg型数据

reg [4:1] regc, regd;
                //定义了两个四位的名为regc和regd的reg型数据
```

3 memory型

Verilog HDL通过对reg型变量建立数组来对存储器建模，可以描述RAM型存储器，ROM存储器和reg文件。数组中的每一个单元通过一个数组索引进行寻址。memory型数据是通过扩展reg型数据的地址范围来生成的。

格式：

reg [n-1:0] 存储器名[m-1:0];

或 reg [n-1:0] 存储器名[m:1];

reg [7:0] mema[255：0];

在同一个数据类型声明语句里，可以同时定义存储器型数据和reg型数据。

```
parameter wordsize=16, //定义二个参数。
memsize=256;
reg [wordsize-1:0] mem[memsize-1:0],writereg, readreg;
```

尽管memory型数据和reg型数据的定义格式很相似，但要注意其不同之处。如一个由n个1位寄存器构成的存储器组是不同于一个n位的寄存器的。

```
reg [n-1:0] rega; //一个n位的寄存器
reg mema [n-1:0]; //一个由n个1位寄存器构成的存储器组
```

一个n位的寄存器可以在一条赋值语句里进行赋值，而一个完整的存储器则不行。

```
rega =0; //合法赋值语句
mema =0; //非法赋值语句
```

如果想对memory中的存储单元进行读写操作，必须指定该单元在存储器中的地址。

```
mema[3]=0; //给memory中的第3个存储单元赋值为0。
```

进行寻址的地址索引可以是表达式，这样就可以对存储器中的不同单元进行操作。表达式的值可以取决于电路中其它的寄存器的值。