上海交通大学
Shanghai Jiao Tong University

Verilog HDL 硬件描述语言（2）
组合逻辑与行为建模

上海交通大学微电子学院
蒋剑飞

组合逻辑的特点

- 组合逻辑在任何时刻，其输出由该时刻的输入决定，在组合逻辑中输入的改变经过一定的内部延时将传递到输出。

组合逻辑中的数据类型：线网？寄存器？

组合逻辑中的赋值：持续赋值

持续赋值与组合逻辑

- 持续赋值操作（assign）中，当表达式中的值改变时，立即更新目标值，与组合逻辑的特点符合，常常用来描述组合逻辑。

```
wire out,in1,in2;

assign out=in1 & in2;   //显式持续赋值

wire out2=in1 &in2;     //隐式持续赋值
```

```
wire [1:0] out,out2, in1,in2;

assign out=in1 & in2;

assign out2=in1 &in2;
```

常用组合电路的描述

反相器

```
module inverter (a,b);

input a;
output b;

wire a,b;

assign b=~a;

endmodule
```

定义线网类型

采用持续赋值

常用组合电路的描述

四输入与门

```
module and4 (a,b,c,d,y);

input a,b,c,d;
output y;
wire a,b,c,d,y,m1,m2;

assign y=m1 &m2;
assign m1=a&b;
assign m2=c&d;

endmodule
```

```
module and4 (a,b,c,d,y);

input a,b,c,d;
output y;
wire a,b,c,d,y,m1,m2;

assign m1=a&b;
assign m2=c&d;
assign y=m1 &m2;

endmodule
```

持续赋值语句是并发的，与其书写的顺序无关。

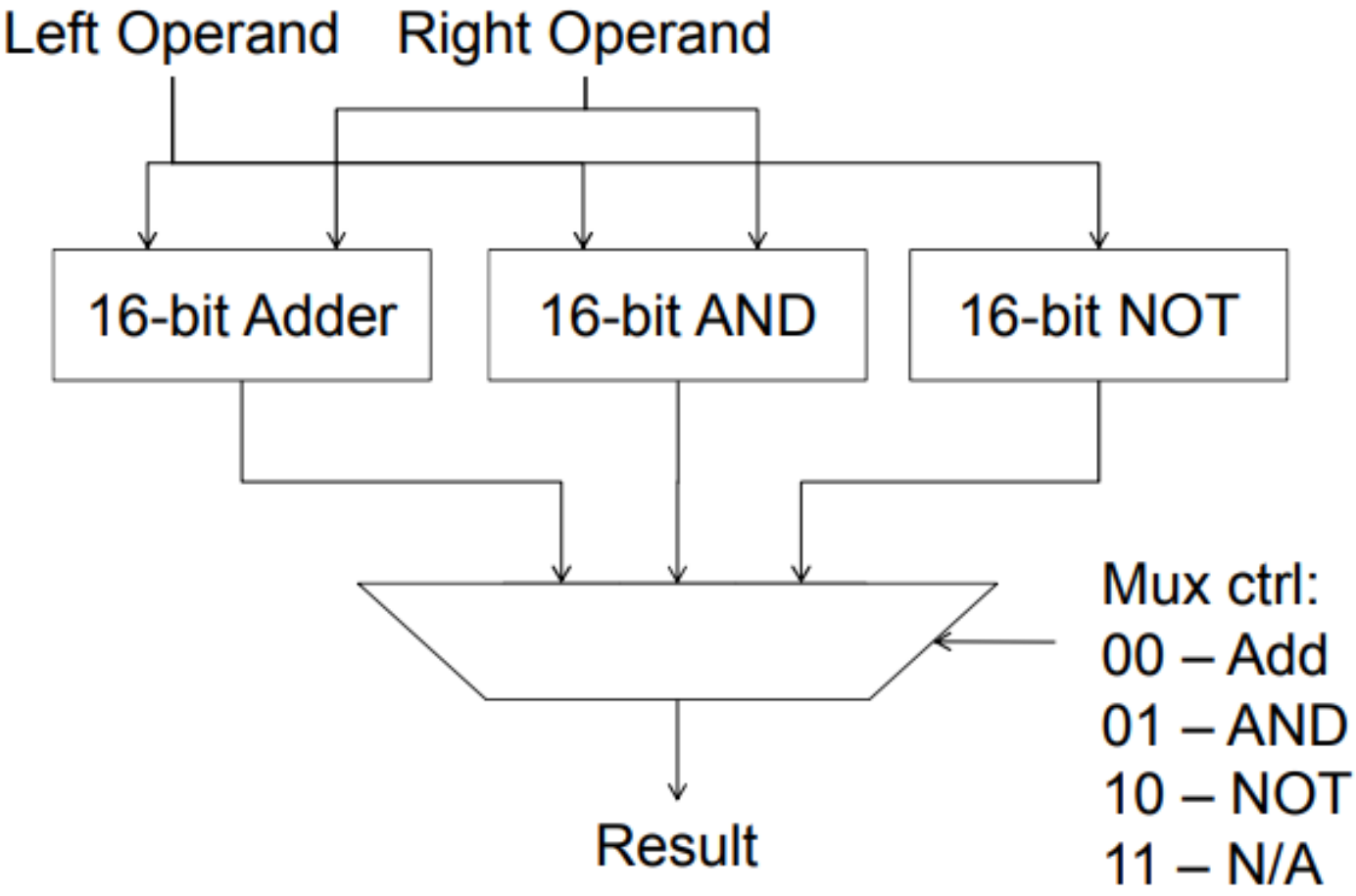
数据流模型

- 数据流模型主要用于组合逻辑的描述，采用assign语句进行赋值，通过布尔表达式能清楚地反映电路的功能结构与组合逻辑的数据特性。

```
module mux2 (f, a, b, sel );
output f;
input a,b,sel;

assign f = (a & ~sel) | (b & sel);
endmodule
```

```
module full_adder2
(a,b,cin,sum,cout);

input a,b,cin;
output sum,cout;

wire a,b,cin,sum,cout;

assign sum=a^b^cin;
assign cout=(a&cin) | (b&cin) | (a&b);

endmodule
```

复杂组合逻辑


Left Operand
Right Operand
16-bit Adder
16-bit AND
16-bit NOT
Result
Mux ctrl:
00 – Add
01 – AND
10 – NOT
11 – N/A

行为模型

- 行为模型是一个模块怎样工作的一种抽象，更加关心模块的工作行为，而不是其具体实现，属于比较高的抽象层次。

- 行为模型通过过程语句来描述。

Verilog中的代码组织

- initial
- always
- assign

```
module module_name (port_list);
input …
output …
wire…
reg…

initial
…

always
….

always
…

assign
…
assign
…
endmodule
```

行为模型的基础——过程结构

- 两种过程结构：

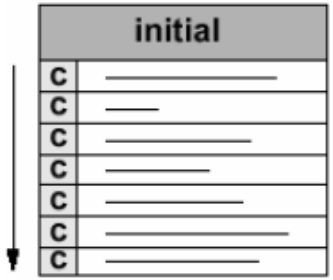
  initial结构（块）——单次执行

  always结构（块）——重复执行

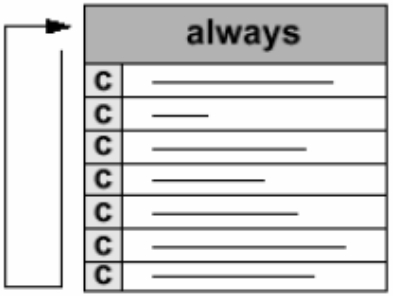
begin…end结构中语句顺序执行，只有一条语句时begin…end可以省略

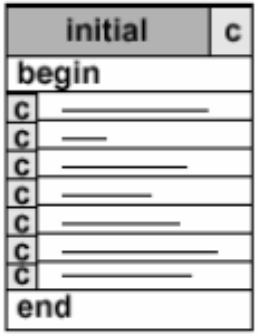
fork…join结构中语句的并行执行。

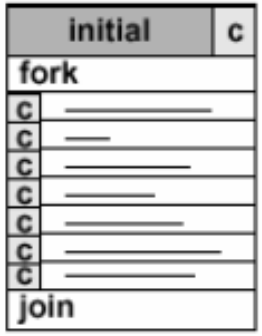
单次触发

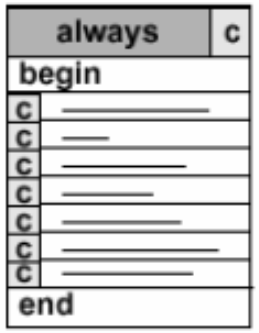


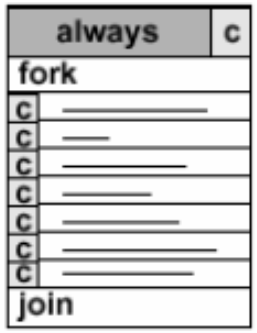
重复触发











过程块的语法介绍

- initial/always块的语法介绍（敏感列表...）

```
initial begin
a=1'b1;          //过程赋值
b=1'b0;
c=2'b1;
#2
a=1'b1;
b=1'b0;
c=2'b1;
end
```

```
module full_adder1( a, b, cin, sum,
cout);
input a, b, cin;
output sum, cout;
wire a, b, cin;
reg sum, cout;

always @(a or b or cin)      //敏感列表
//always @(*)
begin
 {cout, sum} = a + b+ cin;     //过程赋值
end

endmodule
```

过程赋值

- 在过程块中的赋值称为过程赋值，在过程赋值语句中表达式左边的信号必须是寄存器类型（如**reg**类型），表达式右边类型没有限制（表达式左边信号没有声明会怎样？）。

```
module full_adder (a,b,cin,sum,cout);

input a,b,cin;
output sum,cout;
wire a,b,cin;
reg sum,cout;

always @(a or b or cin)
begin
    sum=a^b^cin;
    cout=(a&cin) | (b&cin) | (a&b);
    middle=a^b;
end
endmodule
```

如果一个信号没有声明则缺省为**wire**类型。使用过程赋值语句给**wire**类型变量赋值会产生错误。

过程赋值与持续赋值比较

过程赋值	持续赋值
在always或者initial语句中出现	在一个模块内出现
执行与周围其他语句有关	与其它语句并行执行，在右端操作数的值发生变化时执行
驱动寄存器	驱动线网
使用”=“或者”<= “	使用 “= “赋值
无assign关键词（过程性持续赋值中除外）	有assign关键词

过程块的时序控制

- 过程块时序控制有两类：
  - 时延控制

```
always #5 clk=~clk;
```

```
initial begin
clk=0;
#5 clk=1;
end
```

  - 事件控制
    - 边沿触发事件控制@(<signal>)

```
always @(a or b or c)      //在信号发生翻转时执行
…
always @(posedge clk or negedge rst_n)   //可以指定翻转的沿
…
```

    - 电平敏感事件控制

```
always wait(a)
…
always wait(clk)
…
```

example

```
module test_process;
reg clk,a,b,c;
initial begin
clk=1'b0;
end
always #5 clk=!clk;
initial begin
wait (clk)
$display ("reached clk");
end
initial begin
//wait (posedge clk)// error
$display ("reached posedge of clk");
end
endmodule
```

```
module test_process;
reg clk,a,b,c;
initial begin
clk=1'b1;
end
//always #5 clk=!clk;
initial begin
wait (clk)
$display ("i have reached clk");
end
initial begin
@(clk)
$display ("Reacheed posedge of
clk");
end endmodule
```

```
VSIM 20> run
# i have reached clk
```

过程块的分支控制（if..else)

- If...else...控制

描述方式：
if (表达式)
begin
.....
end
else if（表达式）
begin
.....
end
else
begin
...
end

```
always @(a or b)
begin
if(a)
c=1'b1;
if(b)
c=1'b0;
end
```

要避免的写法

```
always @(a or b or c)
begin
if({a,b}==2'b10)
c=1'b1;
else if({a,b}==2'b01)
c=1'b0;
else if({a,b}==2'b11)
c=1'b0;
else
c=1'b1;
end
```

语法上和其他高级语言类似，编写代码时要养成良好的习惯：

考虑所有的分支情况；

尽量少地使用嵌套；

过程块的分支控制（case）

- case语句

```
case （操作数）
条件1：表达式1；
条件2：表达式2；
条件3：表达式3；
条件4：表达式4；
…
default: 表达式n
endcase
```

```
……
always @(a or b)
begin
case({a,b})
2’b10:c=1’b1;
2’b01:c=1’b0;
default:c=1’b1;
endcase
end
……
```

依次对于分支项求值，第一个与条件表达式匹配的分支项被执行：

考虑所有的分支情况，分支不全时要使用default；

case/casez/casex的区别

- 在case结构中有三种不同类型的case，分别是case, casez, casex。
  - **case**语句进行逐位比较以求完全匹配（包括**x**和**z)**

    case treats 'z' & 'x' as it is
  - 在**casez**语句中，**?** 和**z** 被当作无关值。

    casez treats 'z' as don’t care
  - 在**casex**语句中，**?**，**z** 和**x** 被当作无关值。

    casex treats 'z' & 'x' as dont care

case/casez/casex的区别举例

```
case (sel)
00: mux_out = mux_in[0];
01: mux_out = mux_in[1];
1?: mux_out = mux_in[2];
default: mux_out = mux_in[3];
endcase
```

case treats 'z' & 'x' as it is

？可以用来代替z

When used in a number, the question-mark (?) character is a Verilog HDL alternative for the z character.

```
case (sel)
00: y = a;
01: y = b;
x0: y = c;
1x: y = d;
z0: y = e;
1?: y = f;
default: y = g;
endcase
```

sel	y	case item
00	a	00
11	g	default
xx	g	default
x0	c	x0
1z	f	1?
z1	g	default

casez

```
casez (sel)
00: y = a;
01: y = b;
x0: y = c;
1x: y = d;
z0: y = e;
1?: y = f;
default: y = g;
endcase
```

casez treats 'z' as don't care

sel	y	case item
00	a	00
11	f	1?
xx	g	default
x0	c	x0 (would have matched with z0(item 5) if item 3 is not present.)
1z	d	1x (would have matched with z0(item 5) & 1?(item 6) also.)
z1	b	01 (would have matched with 1?(item 6) also.)

casex

```
casex (sel)
00: y = a;
01: y = b;
x0: y = c;
1x: y = d;
z0: y = e;
1?: y = f;
default: y = g;
endcase
```

casex treats 'z' & 'x' as dont care

sel	y	case item
00	a	00
11	d	1x (would have matched with 1? also)
xx	a	00 (would have matched with 1? also)
x0	a	00 (would have matched with all items except 01)
1z	c	x0 (would have matched with all items except 00,01)
z1	b	01 (would have matched with 1x, 1? also)

Verilog主要建模层次比较

- 行为级模型
  - 高层次的抽象
  - 与硬件的关系最远，
  - 和人的想法最接近
- 数据流模型
  - 用布尔表达式描述的逻辑
- 门级模型
  - 内置基本门（或者用户定义原语UDP）实现的电路

二选一mux的两种模型

行为模型

```
module MUX_2_1 (a,b,sel,op);
input a,b,sel;
output op;
wire a,b,sel;
reg op;
always @(sel or a or b) begin
if(sel)
    op = b;
else
    op = a;
end
endmodule
```

数据流模型

```
module MUX_2_1 (a,b,sel,op);
input a,b,sel;

output op;

wire a,b,sel;
wire op;

assign op = sel?b:a;

endmodule
```

定义为reg类型的，不一定描述寄存器逻辑。

全加器的三种描述——行为模型

行为模型

```
module full_adder1( a, b, cin, sum, cout);
input a, b, cin;
output sum, cout;
wire a, b, cin;
reg sum, cout;
always @(a or b or cin)
 {cout, sum} = a + b+ cin;

endmodule
```

数据流模型

```
module full_adder2 (a,b,cin,sum,cout);

input a,b,cin;
output sum,cout;

wire a,b,cin,sum,cout;

assign sum=a^b^cin;
assign cout=(a&cin) | (b&cin) | (a&b);

endmodule
```

```
module full_adder3 (a,b,cin,sum,cout);

input a,b,cin;
output sum,cout;

wire a,b,cin,sum,cout;
wire p,m1,m2,m3;

xor (p,a,b);
xor (sum,p,cin);
and (m1,a,cin);
and (m2,b,cin);
and (m3,a,b);
or (cout,m1,m2,m3);

endmodule
```

门级模型

四选一mux的两种模型

行为模型

```
module MUX_4_1 (a,b,c,d,sel,op);
input a,b,c,d;
input [1:0] sel;
output op;
wire a,b,c,d;
wire [1:0] sel;
reg op;
always @(sel or a or b or c or d)
begin
case(sel)
2'b00: op = a;
2'b01: op = b;
2'b10: op = c;
2'b11: op = d;
endcase
end
endmodule
```

数据流模型

```
module MUX_4_1 (a,b,c,d,sel,op);

input a,b,c,d;
input [1:0] sel;

output op;

wire a,b,c,d,op;

wire [1:0] sel;

assign op =(sel[1]==1)
            ?(sel[0]==1?d:c)
            :(sel[0]==1?b:a);


endmodule
```

二四译码器的两种模型

行为模型

```
module dec_2_4 (a,b,d1,d2,d3,d4);
input a,b;
output d1,d2,d3,d4;
wire a,b;
reg d1,d2,d3,d4;
always @(a or b)
case({a,b})
2'b00: begin d1=0; d2=1;
             d3=1; d4=1;
end
2'b01: begin d1=1; d2=0;
             d3=1; d4=1;
end
2'b10: begin d1=1; d2=1;
             d3=0; d4=1;
end
2'b11: begin d1=1; d2=1;
             d3=1; d4=0;
end
endcase
endmodule
```

数据流模型

```
module dec_2_4 (a,b,d1,d2,d3,d4);
input a,b;
output d1,d2,d3,d4;
wire a,b,d1,d2,d3,d4;

assign d1=a  |  b;
assign d2=a  | !b;
assign d3=!a | b;
assign d4=!a | !b;

endmodule
```

循环结构语句

- Verilog定义了四种循环结构
  - ***repeat***：将一块语句循环执行确定次数。
    repeat (次数表达式） **<**语句**>**
  - **while**：在条件表达式为真时一直循环执行
    while (条件表达式） **<**语句**>**
  - **forever**：重复执行直到仿真结束
    forever <语句**>**
  - **for**：在执行过程中对变量进行计算和判断，在条件满足时执行
    for(赋初值；条件表达式；计算） **<**语句**>**

repeat

- repeat：将一块语句循环执行确定次数。

repeat (次数表达式) 语句

```
// Parameterizable shift and add multiplier
module multiplier( result, op_a, op_b);

input [7:0] op_a, op_b;
output [15:0] result;
reg [15:0] shift_opa, result;
reg [7:0] shift_opb;
always @( op_a or op_b) begin
result = 0;
shift_opa = op_a; // 扩展至16位
shift_opb = op_b;
repeat (8) begin
if (shift_opb[0]) result = result + shift_opa;
shift_opa = shift_opa << 1; // Shift left
shift_opb = shift_opb >> 1; // Shift right
end   //end repeat
end   //end always
endmodule
```

while

- ***while***：只要表达式为真(不为0)，则重复执行一条语句(或语句块)

```
module test();
reg [7: 0] tempreg;
reg [3: 0] count;
initial
begin
count = 0;
tempreg=7;
while (tempreg)     // 统计tempreg中1 的个数
begin
if (tempreg[0]) count = count + 1;
tempreg = tempreg >> 1; // Shift right
end
$display ("%b", count);
end
endmodule
```

forever

- ***forever***：一直执行到仿真结束

forever应该是过程块中最后一条语句。其后的语句将永远不会执行。forever语句不可综合，通常用于test bench描述。

```
...
reg clk;
initial
begin
clk = 0;
forever
begin
#10 clk = 1;
#10 clk = 0;
end
end
...
```

这种行为描述方式可以非常灵活地描述时钟，可以控制时钟的开始时间及周期占空比。

for

- ***for***：只要条件为真就一直执行条件表达式若是简单的与0比较通常处理得更快一些。

```
// X检测
for (index = 0; index < size; index = index + 1)
if (val[ index] === 1'bx)
$display (" found an X");

// 存储器初始化; “!= 0”仿真效率高
for (i = size; i != 0; i = i - 1)
memory[ i- 1] = 0;

// 阶乘序列
factorial = 1;
for (j = num; j != 0; j = j - 1)
factorial = factorial * j;
```

内容总结

- 组合逻辑建模
- initial/always
- if.../else..　case
- repeat/while/for/forever

Thank you !