

组合逻辑电路设计 与Verilog基础

北京科技大学 计算机系
齐悦

目录

1

Verilog 建模方式

2

组合逻辑电路简介

3

常用组合逻辑模块的Verilog描述

4

组合逻辑电路可综合描述的常见问题

5

测试平台搭建

6

以加法器为例讲述关键路径的时延问题

Verilog概述

- **Verilog HDL与C语言**在很多地方非常相似。
- **C语言**的各函数之间是串行的，而**Verilog**的各个模块间是并行的
- 体会如何用Verilog描述电路

C语言	Verilog	功能	C语言	Verilog	功能
+	+	加	>=	>=	大于等于
-	-	减	<=	<=	小于等于
*	*	乘	==	==	等于
/	/	除	!=	!=	不等于
%	%	取模	~	~	取反
!	!	逻辑非	&	&	按位与
&&	&&	逻辑与			按位或
		逻辑或	^	^	按位异或
>	>	大于	<<	<<	左移
<	<	小于	>>	>>	右移

Verilog概述 - 初探

常用信号类型:
wire – 线网类型
reg – 寄存器类型

标识符

Module name

Module ports

Verilog中有三种端口类型:

input - 输入端口

output - 输出端口

inout - 双向端口

module Add_half (sum, c_out, a, b);

input

output

wire

a, b;

sum, c_out;

c_out_bar;

*Declaration of port
modes*

*Declaration of internal
signal*

xor (sum, a, b);

nand (c_out_bar, a, b);

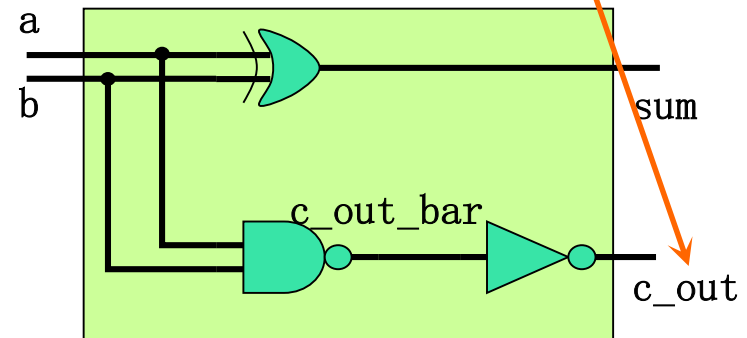
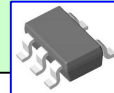
not (c_out, c_out_bar);

*Instantiation of primitive
gates*

endmodule

Verilog 关键字

端口等价于硬件
的引脚(pin)



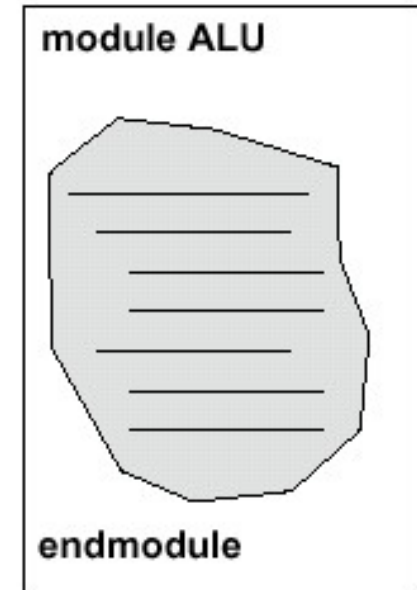
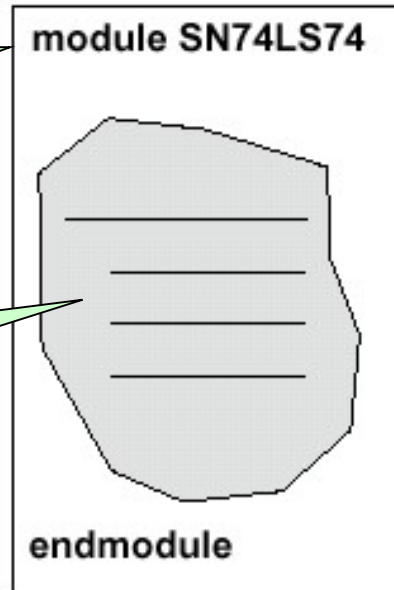
Verilog概述 - 初探

■ module模块

- ❑ 每一个模块的描述从关键词 *module* 开始，有一个名称（如 SN74LS74, DFF, ALU 等等），由关键词 *endmodule* 结束
- ❑ 物理块，如 IC 或 ASIC 单元
- ❑ 逻辑块，如一个 CPU 设计的 ALU 部分
- ❑ 整个系统

module 是层次化设计的基本构件

逻辑描述放在 **module** 内部



Verilog概述 - 初探

标识符与关键字

- 标识符就是用户为程序中的**Verilog** 对象所起的名字
- **a-z, A-Z, 0-9, _, \$**
- 标识符必须以字母或者下横线开头
- 标识符最长可以达到**1023**个字符
- **Verilog**语言是**大小写敏感**的
- 所有的**Verilog****关键字都是小写**

```
module adder(a,b,cin,s,cout);
    input a,b,cin;
    output s,cout;
    ...
endmodule
```

标识符

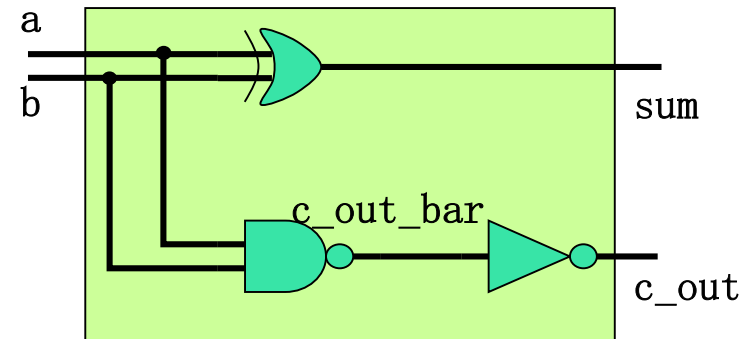
always	and	assign	begin	buf
bufif0	bufif1	case	casex	casez
cmos	deassign	default	defparam	disable
edge	else	end	endcase	endfunction
endmodule	endprimitive	endspecify	endtable	endtask
event	for	force	forever	fork
function	highz0	highz1	if	initial
inout	input	integer	join	large
macromodule	medium	module	nand	negedge
nmos	nor	not	notif0	notif1
pull0	pull1	pulldown	pullup	rcmos
reg	release	repeat	rmos	rpmos
rtran	rtranif0	rtranif1	scalared	small
specify	specparam	strong0	strong1	supply0
supply1	table	task	time	tran
tranif0	tranif1	tri	tri0	tri1
triand	trior	vectored	wait	wand
weak0	weak1	while	wire	wor
xnor	xor			

Verilog概述 - 建模方式

- Verilog 对电路建模方式有三类：
 - 结构化描述方式
 - 数据流描述方式
 - 行为描述方式
-

结构化描述方式

```
module Add_half ( sum, c_out, a, b );  
    input      a, b;  
    output     sum, c_out;  
    wire      c_out_bar;  
  
    xor (sum, a, b);  
    nand (c_out_bar, a, b);  
    not (c_out, c_out_bar);  
endmodule
```



□ wire (线网net)

- 表示元件之间的物理连接
- wire a,b;
- wire信号的缺省值是z

□ 基本门级元件

- Verilog中已有一些建立好的逻辑门和开关模型
- 逻辑门: and、nand、or...
- 缓冲器与非门: buf、not
- 三态门
- MOS开关
- ...

数据流描述方式

```
module Add_half ( sum, c_out, a, b );
```

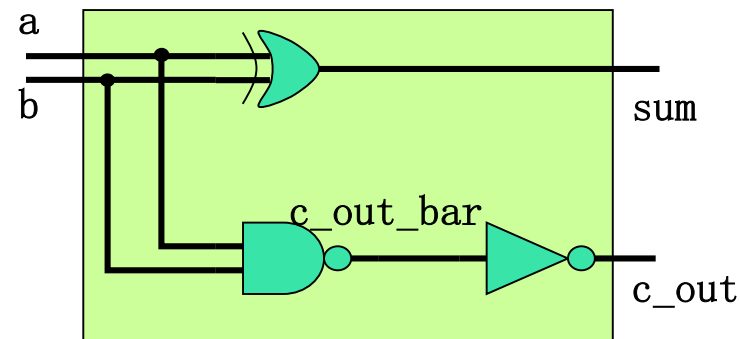
```
    input        a, b;
```

```
    output       sum, c_out;
```

```
    assign       sum = a ^ b;
```

```
    assign       c_out = a & b;
```

```
endmodule
```



□ assign

- 持续赋值语句
- 用来描述组合逻辑电路

□ 位运算符

- ~、&、|、^、^~
- 按位操作
- 两个操作数位数不同时，位数少的高位零扩展

行为描述方式

```
module Add_half ( sum, c_out, a, b );
```

```
    input      a, b;
```

```
    output     sum, c_out;
```

```
    reg        sum, c_out;
```

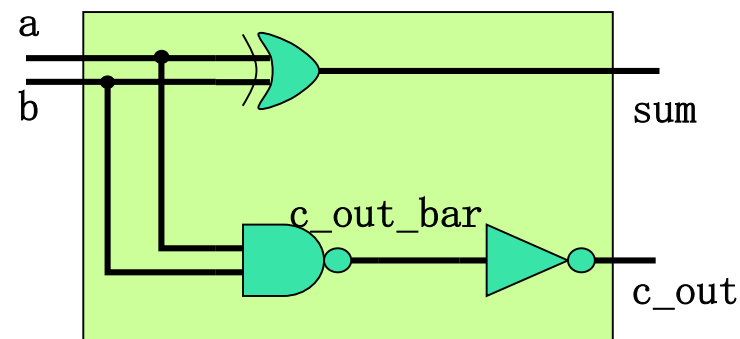
```
    always @ ( a or b ) begin
```

```
        sum = a ^ b;
```

```
        c_out = a & b;
```

```
    end
```

```
endmodule
```



□ reg (寄存器register)

- 过程块输出信号
- reg信号的缺省值是x

□ begin end顺序块

- 将多条语句组合在一起
- 块内语句顺序执行，上一条语句执行结束后下一条开始执行

□ wire与reg

- assign语句中被赋值的信号定义成wire
- 过程块中被赋值的信号定义成reg
- 除了应该定义成reg的都定义成wire

□ always过程块

- always块语句重复执行
- 敏感信号表
- 多个always块之间是并行执行的

行为描述方式

```
module Add_half ( sum, c_out, a, b );
```

```
    input      a, b;
```

```
    output    sum, c_out;
```

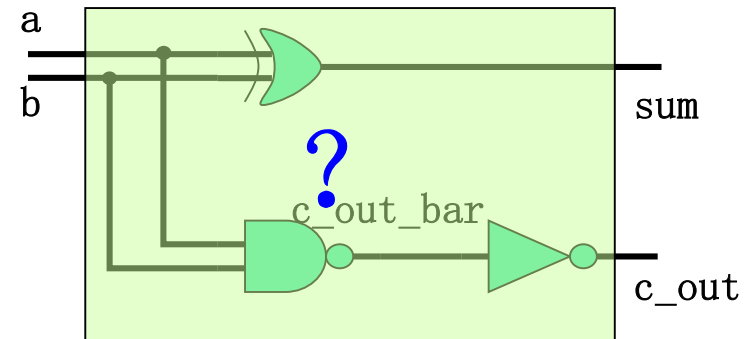
```
    reg       sum, c_out;
```

```
    always @ ( a or b ) begin
```

```
        {c_out, sum} <= a+b;
```

```
    end
```

```
endmodule
```



□ { } 位拼接运算符

- 用于位的重组和矢量构造
- 拼接时不限定操作数的数目。在操作符符号{ }中，用逗号将操作数分开
- {a,b,c,d}
- 可以在赋值号左边，也可以右边

目录

1

Verilog 建模方式

2

组合逻辑电路简介

3

常用组合逻辑模块的Verilog描述

4

组合逻辑电路可综合描述的常见问题

5

测试平台搭建

6

以加法器为例讲述关键路径的时延问题

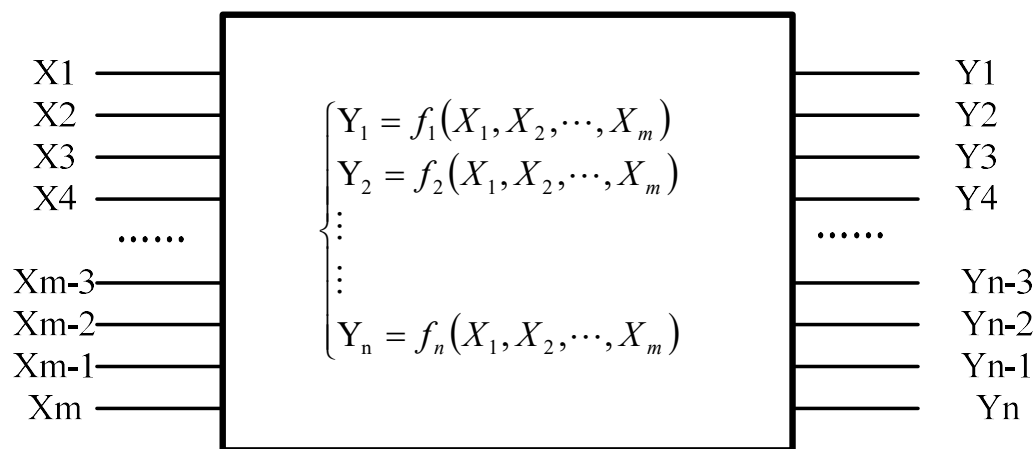
组合逻辑电路概述

- 数字电路按其完成逻辑功能的不同特点，划分为**组合逻辑电路**和**时序逻辑电路**两大类
- 复杂数字逻辑由组合逻辑电路和时序逻辑电路构成
- 常用的组合逻辑器件包括**编码器、译码器、多路选择器、比较器、加法器等**

**组合逻辑电路和时序逻辑电路
主要区别是什么？**

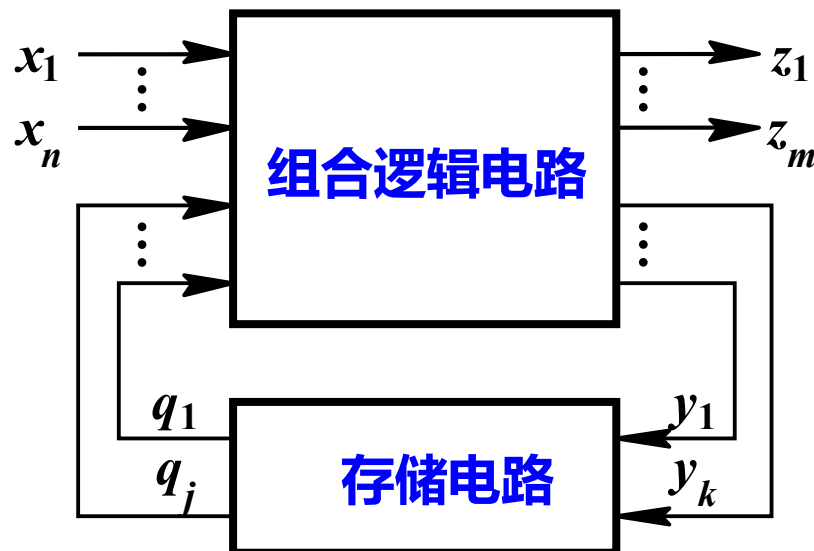
组合逻辑电路

- 从功能上，组合逻辑电路在任一时刻的输出仅和电路当前的输入有关
- 从内部结构上，组合电路都是单纯由逻辑单元组成（**不含存储单元，电路无记忆功能**）



时序电路

- 还和电路的内部状态有关
- 还有寄存器等存储单元，
电路内有反馈



逻辑关系:

$$\left\{ \begin{array}{ll} z_m = f_m(x_1, x_2, \dots, x_n, q_1^n, q_2^n, \dots, q_j^n) & \text{输出方程} \\ y_k = g_k(x_1, x_2, \dots, x_n, q_1^n, q_2^n, \dots, q_j^n) & \text{驱动方程} \\ q_j^{n+1} = h_j(y_1, y_2, \dots, y_k, q_1^n, q_2^n, \dots, q_j^n) & \text{状态方程} \end{array} \right.$$

Verilog如何描述组合逻辑

■ module中描述组合逻辑的语句

简单逻辑函数

□ 使用**assign**描述

□ 使用**always**描述

复杂组合逻辑

```
assign q = (a==1?) d : 0 ;
```

```
always @(a or d)
begin
    if (a==1) q = d ;
    else q = 0;
end
```


目录

1

Verilog 建模方式

2

组合逻辑电路简介

3

常用组合逻辑模块的Verilog描述

4

组合逻辑电路可综合描述的常见问题

5

测试平台搭建

6

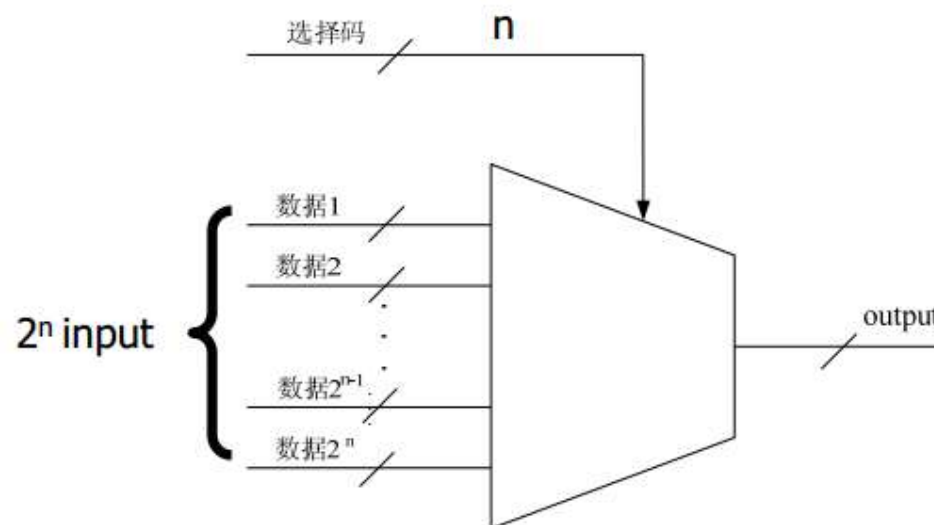
以加法器为例讲述关键路径的时延问题

多路选择器（数据选择器）

- 多路选择器是一个多输入单输出的组合逻辑电路
- 根据选择码，从多个输入数据流选取一个输出

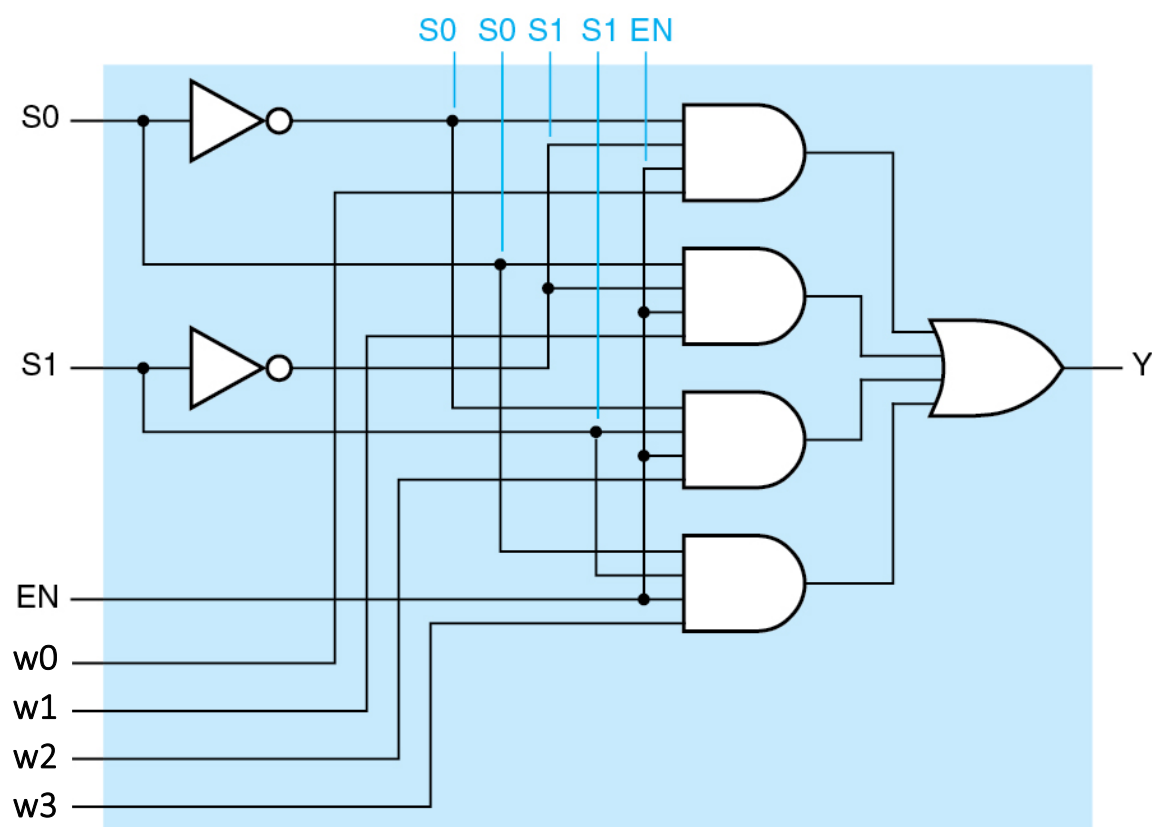
典型Verilog描述语句有以下三种

- assign语句 + 条件操作符
- if...else 及其嵌套语句
- case多分支语句



多路选择器—4选1

■ 电路结构



多路选择器—4选1

■ 数据流描述 - assign

```
module mux4to1 (w0, w1, w2, w3, S, f);
```

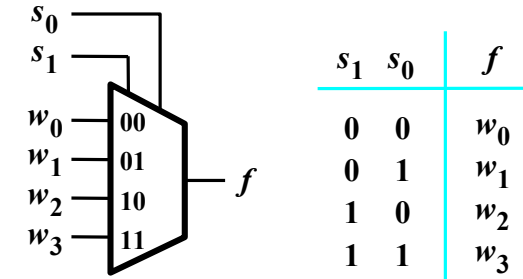
```
    input w0, w1, w2, w3;
```

```
    input [1:0] S;
```

```
    output f;
```

```
    assign f = S[1] ? (S[0] ? w3 : w2) : (S[0] ? w1 : w0);
```

```
endmodule
```



(a) Graphic symbol (b) Truth table

□ 多位宽端口

- 以 [MSB: LSB] 格式
- 如果 input [0:2] S, 则S[0]为高位

□ 条件运算符? :

- 三目运算符
- `result = condition ? true_value:false_value;`

■ 行为级描述 - if else 分支语句

```
module mux4to1 (W, S, f);
  input [0:3] W;
  input [1:0] S;
  output reg f;
```

```
always @(*)
```

```
  if (S == 0)
    f = W[0];
  else if (S == 1)
    f = W[1];
  else if (S == 2)
    f = W[2];
  else if (S == 3)
    f = W[3];
```

```
endmodule
```

```
module mux4to1 (W, S, f);
  input [0:3] W;
  input [1:0] S;
  output reg f;
```

```
always @(W, S) //always @(W or S)
```

```
  if (S == 2'b00)
    f = W[0];
  else if (S == 2'b01)
    f = W[1];
  else if (S == 2'b10)
    f = W[2];
  else if (S == 2'b11)
    f = W[3];
```

```
endmodule
```

s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

(b) Truth table

- if 语句
- 常量
- (*)
- 注释
- //
- /* ... */

if语句

if (表达式)

语句

else

语句

if (表达式)

语句

else if (表达式)

语句

else

语句

if (a > b)

res = 1;

else if (a < c)

begin

res = 2;

out = 4;

end

else

res = 3;

关系运算符

- >、<、>=、<=
- 其结果是1' b1、1' b0或1' bx

- 为确保正确关联，使用begin...end块语句指定其作用域
- 可以多层嵌套。在嵌套if序列中，else和前面最近的if相配对

数的表示方式

X可以用来定义十六进制数的4位二进制状态，八进制数的3位，二进制数的1位。Z的表示方法同X类似。

■ <size>'<base><value>

if (S == 2'b01)

- size: 以bit为单位
- base: b(二进制),o(八进制),d(十进制),h(16进制)
- value: 和进制相应的数值, x, z, ? (x,z不区分大小写)

■ 例

- 16 //默认32位十进制数, 32'd16
- 8'd16
- 8'h10
- 8'b0001_0010 //下划线“_”是为了增加可读性
- 32'bx0x1 //32位, 高位补x
- 2'b? //?、z、Z都表示高阻
- 10'b10 //左边添0补齐, 0000000010
- 3'b1001_0011 //与3'b011相等
- 5'h0FFF //与5'h1F相等

■ 行为级描述 - case 分支语句

```
module mux4to1 (W, S, f);
```

```
    input [0:3] W;
```

```
    input [1:0] S;
```

```
    output reg f;
```

```
    always @(W, S)
```

```
        case (S)
```

```
            0: f = W[0];
```

```
            1: f = W[1];
```

```
            2: f = W[2];
```

```
            3: f = W[3];
```

```
            default: f = 1'b0;
```

```
        endcase
```

```
    endmodule
```

s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

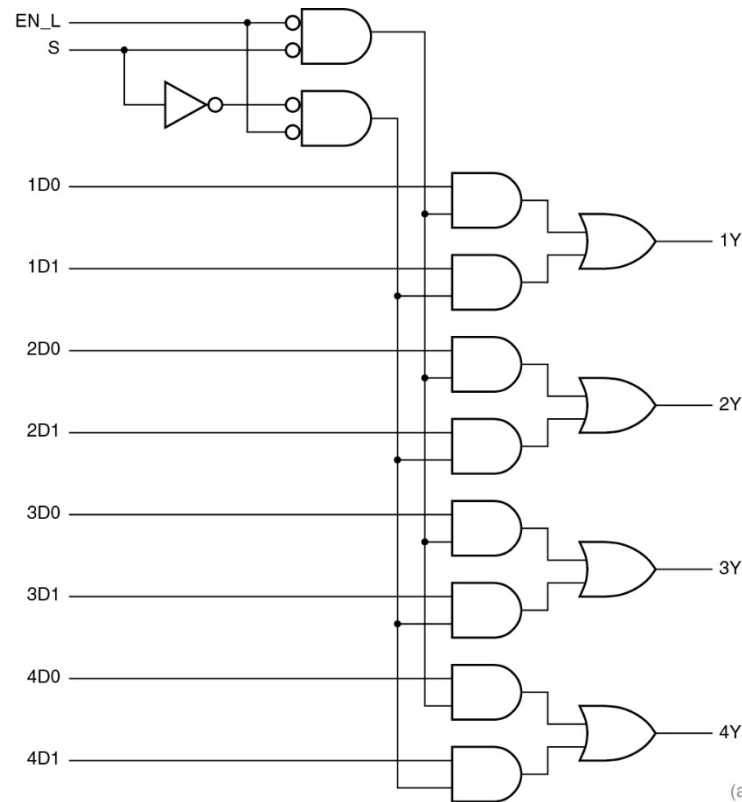
(b) Truth table

case 语句

- **default** 语句可选，在没有任何条件成立时执行
- 如果未说明 **default**，Verilog 不执行任何动作
- 多个 **default** 语句非法

■ 例 - 4输入8位多路选择器

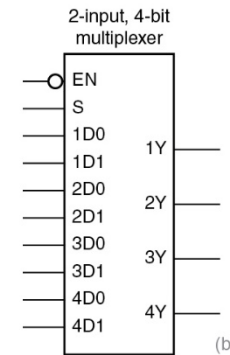
module mux8b_4to1 (W, S, f);



2输入4位

s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

(b) Truth table



endmodule

■ 例 - 4输入8位多路选择器

```
module mux4to1 (w0, w1, w2, w3, S, f);
```

```
    input [7:0] w0, w1, w2, w3;
```

```
    input [1:0] S;
```

```
    output reg[7:0] f;
```

```
    always @(*)
```

```
        case (S)
```

```
            0: f = w0;
```

```
            1: f = w1;
```

```
            2: f = w2;
```

```
            3: f = w3;
```

```
            default: f = 8'b0;
```

```
        endcase
```

```
endmodule
```

s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

(b) Truth table

□ 多位宽变量

- 以 [MSB: LSB] 格式
- reg [3: 0] v; // 4位寄存器
- reg [7: 0] m, n; // 两个8位寄存器
- wire [31: 0] databus, addrbus;
- wire [35: 4] abus, bbus;
- assign abus = databus;
- assign out[5:2] = v;
- assign out[7] = m[1];

■ 结构化描述例：用4选1搭建16选1选通器

```
module mux16to1 (W, S, f);
```

```
    input [0:15] W;
```

```
    input [3:0] S;
```

```
    output f;
```

```
    wire [0:3] M;
```

```
    mux4to1 Mux1 (W[0:3], S[1:0], M[0]);
```

```
    mux4to1 Mux2 (W[4:7], S[1:0], M[1]);
```

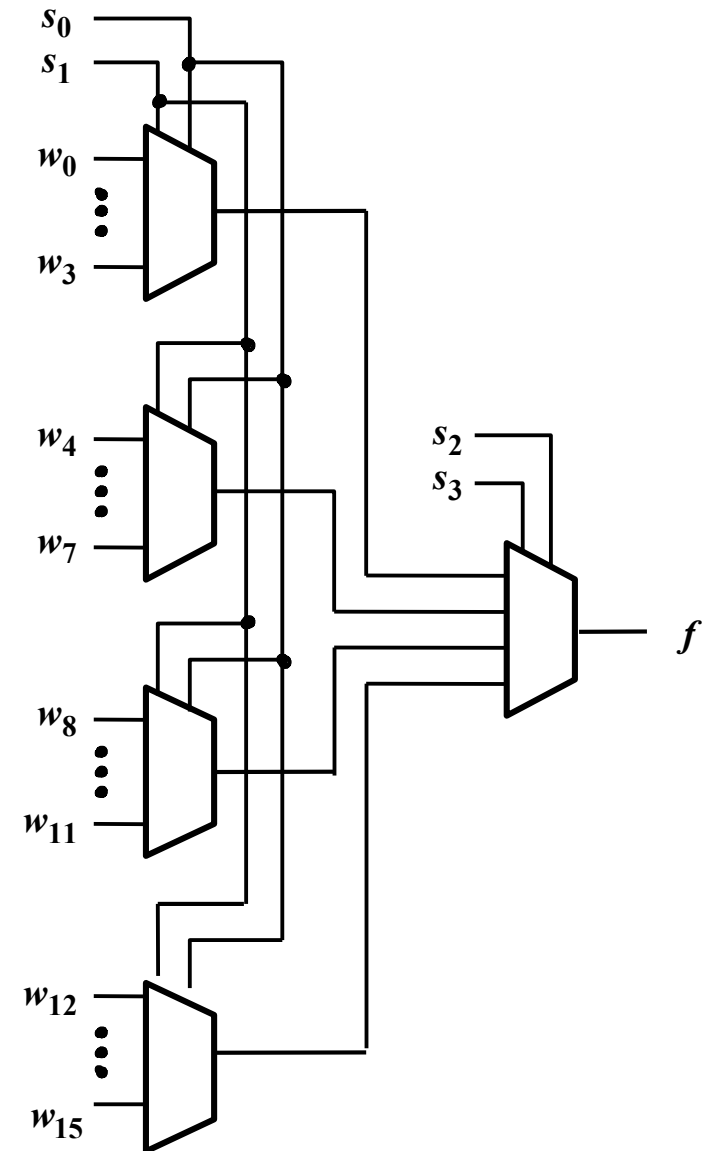
```
    mux4to1 Mux3 (W[8:11], S[1:0], M[2]);
```

```
    mux4to1 Mux4 (W[12:15], S[1:0], M[3]);
```

```
    mux4to1 Mux5 (M[0:3], S[3:2], f);
```

```
endmodule
```

模块实例化



元件例化

- 格式:

- 模块名 <实例名> (<端口列表>);

- 端口列表有两种表示方式

- 端口名字关联:

(. 端口名 (信号名1), . 端口名 (信号名2), ……)

- 位置关联: 隐式给出端口与信号之间的关系

(信号名1, 信号名2, ……)

例化的端口列表中信号的顺序要与该模块定义的端口列表中端口顺序严格一致

- 模块实例化与调用程序不同。每个实例都是模块的一个完全的拷贝，相互独立、并行

AND u1(a,b,and_out);

AND u1(.a(a), .b(b), .o(and_out));

多路选择器的Verilog编码风格

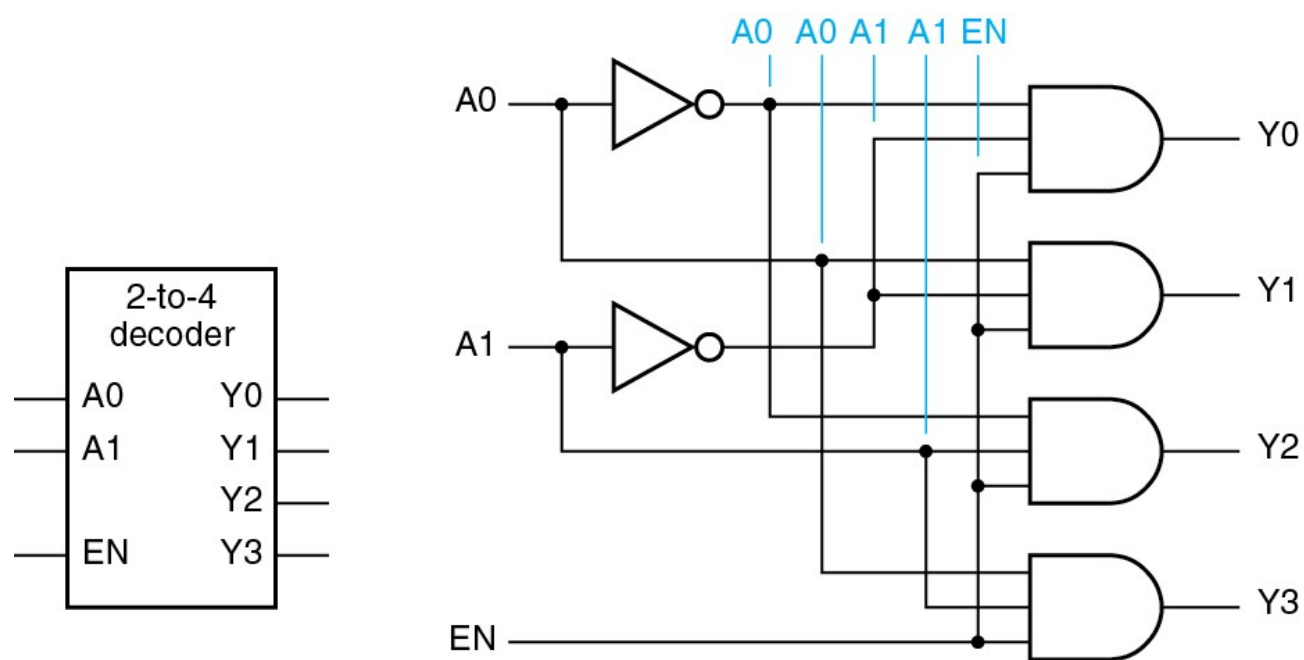
- 二选一建议使用**if else**
- 四选一以上建议使用**case**
- 多选一超过**8**输入时，建议拆分成多个小规模选通器

译码器/编码器

- 将一组形式的二进制数据转化为另一种形式的二进制数据
- 编码器将多位输入数据流编码成更短的码流，使得编码器的输出端口减少，具有 2^n （或小于）输入位的编码器可提供 n 位编码输出线
- 译码器将先前编码过的数据解码，是编码器的逆过程， n 位的输入可代表 2^n 种不同的信息

译码器/编码器

■ 电路结构



描述译码器/编码器的语句

- 通常，采用下列语句描述：
- **if...else**及其嵌套结构
- **case/casex/casez**结构

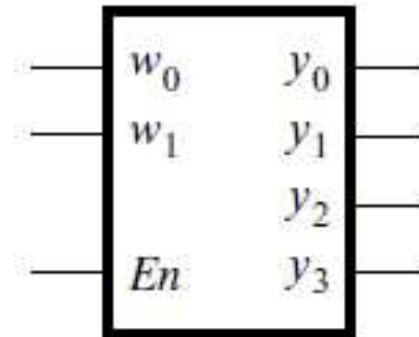
2-4译码器

```
module dec2to4 (W, En, Y);  
    input [1:0] W;  
    input En;  
    output reg [0:3] Y;
```

```
endmodule
```

En	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

(a) Truth table



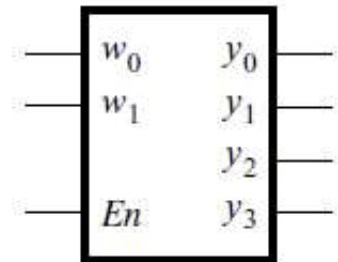
(b) Graphical symbol

■ 2-4译码器的另一种描述方式

```
module dec2to4 (W, En, Y);  
    input [1:0] W;  
    input En;  
    output reg [0:3] Y;  
  
    always @(W, En)  
    begin  
        if (En == 0)  
            Y = 4'b0000;  
        else  
            case (W)  
                0: Y = 4'b1000;  
                1: Y = 4'b0100;  
                2: Y = 4'b0010;  
                default: Y = 4'b0001;  
            endcase  
        end  
    end  
endmodule
```

En	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

(a) Truth table

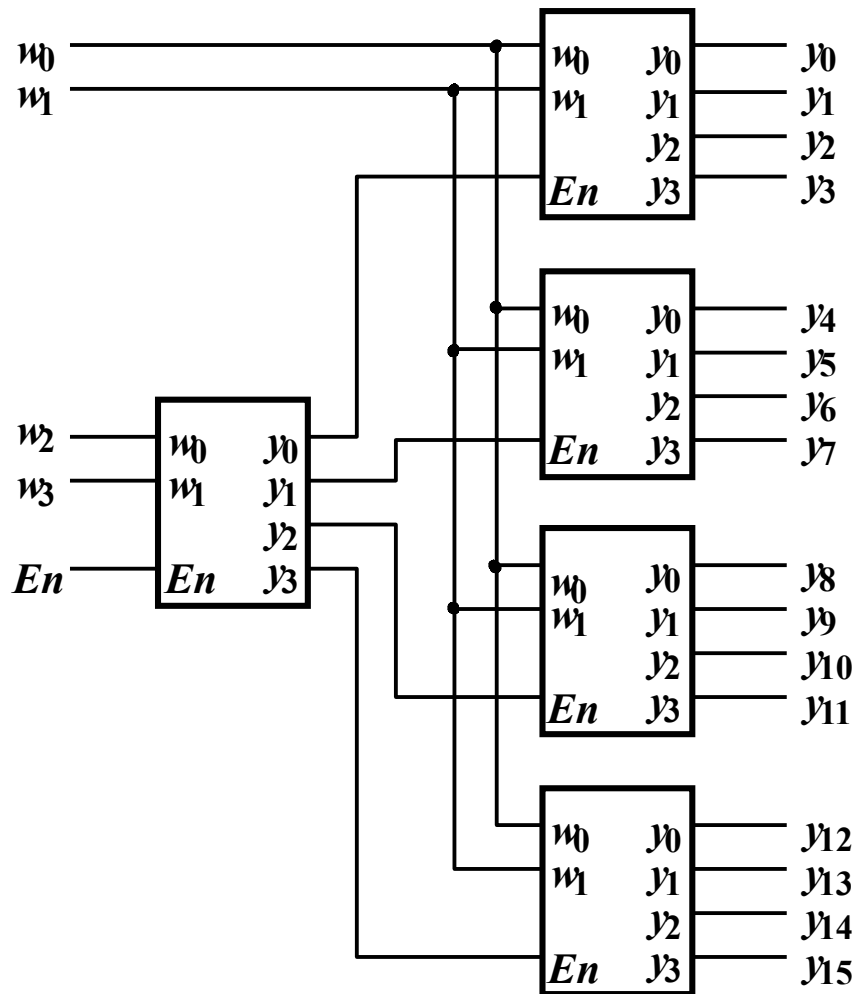


(b) Graphical symbol

□ 相等运算符

- 逻辑等: ==、!=
- case等: ===、!==

■ 结构化描述实现4-16译码器



```
module dec4to16 (W, En, Y);
```

```
    input [3:0] W;
```

```
    input En;
```

```
    output [0:15] Y;
```

```
    wire [0:3] M;
```

```
    dec2to4 Dec1 (W[3:2], En, M[0:3]);
```

```
    dec2to4 Dec2 (W[1:0], M[0], Y[0:3] );
```

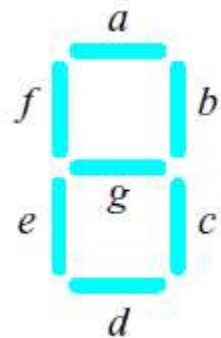
```
    dec2to4 Dec3 (W[1:0], M[1], Y[4:7] );
```

```
    dec2to4 Dec4 (W[1:0], M[2], Y[8:11]);
```

```
    dec2to4 Dec5 (W[1:0], M[3], Y[12:15] );
```

```
endmodule
```

代码转化器（显示译码器）



w_3	w_2	w_1	w_0	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	1	1	1	0	1	1	1
1	0	1	1	0	0	1	1	1	1	1
1	1	0	0	1	0	0	1	1	1	0
1	1	0	1	0	1	1	1	1	0	1
1	1	1	0	1	0	0	1	1	1	1
1	1	1	1	1	0	0	0	1	1	1

```

module seg7 (hex, leds);
    input [3:0] hex;
    output reg [1:7] leds;

```

```

    always @(hex)
        case (hex) //abcdefg
            0: leds = 7'b1111110;
            1: leds = 7'b0110000;
            2: leds = 7'b1101101;
            3: leds = 7'b1111001;
            4: leds = 7'b0110011;
            5: leds = 7'b1011011;
            6: leds = 7'b1011111;
            7: leds = 7'b1110000;
            8: leds = 7'b1111111;
            9: leds = 7'b1111011;
            10: leds = 7'b1110111;
            11: leds = 7'b0011111;
            12: leds = 7'b1001110;
            13: leds = 7'b0111101;
            14: leds = 7'b1001111;
            15: leds = 7'b1000111;
        endcase

```

```

endmodule

```

编码器

w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

□ 逻辑运算符

- !、&&、||
- 逻辑操作符的结果为一位1, 0或x

```
always @(W)
```

```
    case (W)
```

```
        4'b0001: Y = 2'b00;
```

```
        4'b0010: Y = 2'b01;
```

```
        4'b0100: Y = 2'b10;
```

```
        4'b1000: Y = 2'b11;
```

```
        default: Y = 2'b00;
```

```
    endcase
```

```
always @ (W)
```

```
begin
```

```
    if (W[3]) Y=2'b11;
```

```
    else if (W[2]) Y=2'b10;
```

```
    else if (W[1]) Y=2'b01;
```

```
    else Y=2'b00;
```

```
end
```

编码/译码电路的Verilog编码风格

- 通常使用**case**语句实现编码 / 译码模块
- 优先级编码器可以采用**if**语句实现
- 有时也可以用**for**循环结构实现，初学者不建议！！！！

设计举例：简单ALU（4位操作数）

Operation	Inputs	Outputs
	s_2 s_1 s_0	F
Clear	0 0 0	0 0 0 0
B - A	0 0 1	$B - A$
A - B	0 1 0	$A - B$
ADD	0 1 1	$A + B$
XOR	1 0 0	$A \text{ XOR } B$
OR	1 0 1	$A \text{ OR } B$
AND	1 1 0	$A \text{ AND } B$
Preset	1 1 1	1 1 1 1

确定输入信号和输出信号

指令译码、操作数A、B；运算结果

确定输入和输出的逻辑状态关系

用Verilog正确描述电路功能

```
module alu(F, S, A, B); // 74381 ALU
    output reg [3:0] F;
    input [2:0] S;
    input [3:0] A, B;
    always @(S, A, B)
        case (S)
            0: F = 4'b0000;
            1: F = B - A;
            2: F = A - B;
            3: F = A + B;
            4: F = A ^ B;
            5: F = A | B;
            6: F = A & B;
            default: F = 4'b1111;
        endcase
endmodule
```

每个操作可能表示一个子电路模块

设计举例：裁判表决器

- 设计一个举重裁判表决器，举重比赛有3个裁判，1个主裁判和2个副裁判，只有当两个或两个以上裁判判明成功，并且其中有主裁判时，表决器判决举重成功

确定输入信号和输出信号

input a,b,c; output y;

确定输入和输出的逻辑状态关系

用Verilog正确描述电路功能

输入			输出
A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

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


目录

1

Verilog 建模方式

2

组合逻辑电路简介

3

常用组合逻辑模块的Verilog描述

4

组合逻辑电路可综合描述的常见问题

5

测试平台搭建

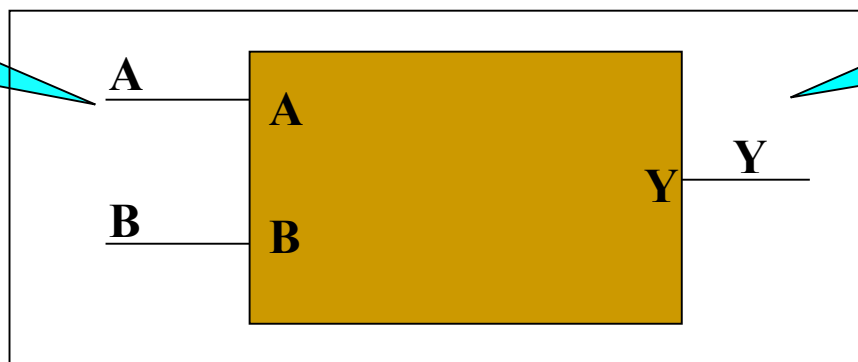
6

以加法器为例讲述关键路径的时延问题

选择正确的数据类型

输入端口可以由
net/register驱动，但
输入端口只能是**net**

双向端口输入/输出**只能**
是**net**类型



输出端口可以是
net/register类型，输
出端口只能驱动**net**

```
module top;  
  wire y;  
  reg a, b;  
  DUT u1 (y, a, b) ;  
  initial begin  
    a = 0; b = 0;  
    #5 a = 1;  
  end  
endmodule
```

在过程块中只能给
register类型赋值

```
module DUT (Y, A, B);  
  output Y;  
  input A, B;  
  wire Y, A, B;  
  and (Y, A, B) ;  
endmodule
```

若**Y, A, B**说明为
reg则会产生错误。

阻塞赋值和非阻塞赋值

过程赋值有两类：

```
module swap_vals;
  reg a, b, clk;
  initial begin
    a = 0; b = 1; clk = 0;
  end
  always #5 clk = ~clk;
  always @(posedge clk)
  begin
    a <= b; // 非阻塞过程赋值
    b <= a; // 交换a和b值
  end
endmodule
```

阻塞过程赋值执行完成后再执行顺序块内下一条语句。

非阻塞赋值不阻塞过程流，仿真器读入一条赋值语句并对它进行调度之后，即可以处理下一条赋值语句。

若过程块中的所有赋值**都是非阻塞**的，赋值按两步进行：

1. 仿真器计算所有右端表达式的值，保存结果，并进行调度在时序控制指定时间赋值。
2. 在经过相应的延迟后，仿真器通过将保存的值赋给左端表达式完成赋值。

□ #：延迟

- initial begin #10 rst=1; end
- 不可综合
- #10 延迟10个时间单位

□ initial

- 只执行一次
- Initial不可综合
- 主要用于激励文件

阻塞赋值和非阻塞赋值

```
always @(posedge clk) begin
    x = a ;
    y = x ;
    z = y ;
end
```

初始值a=0, x= 1, y=2, z=3

结果x= 0, y=0, z=0

```
always @(posedge clk) begin
    x = a;
    y = z ;
    z = x ;
end
```

初始值a=0, x= 1, y=2, z=3

结果x= 0, y=3, z=0

```
always @(posedge clk) begin
    x <= a ;
    y <= x ;
    z <= y ;
end
```

初始值a=0, x= 1, y=2, z=3

结果 x= 0, y=1, z=2

```
always @(posedge clk) begin
    x <= a ;
    y <= z ;
    z <= x ;
end
```

初始值a=0, x= 1, y=2, z=3

结果 x= 0, y=3, z=1

阻塞赋值实现组合逻辑电路

- 在描述组合逻辑电路的always块里用阻塞赋值
- 组合逻辑里的信号是单向按顺序经过一系列处理的过程

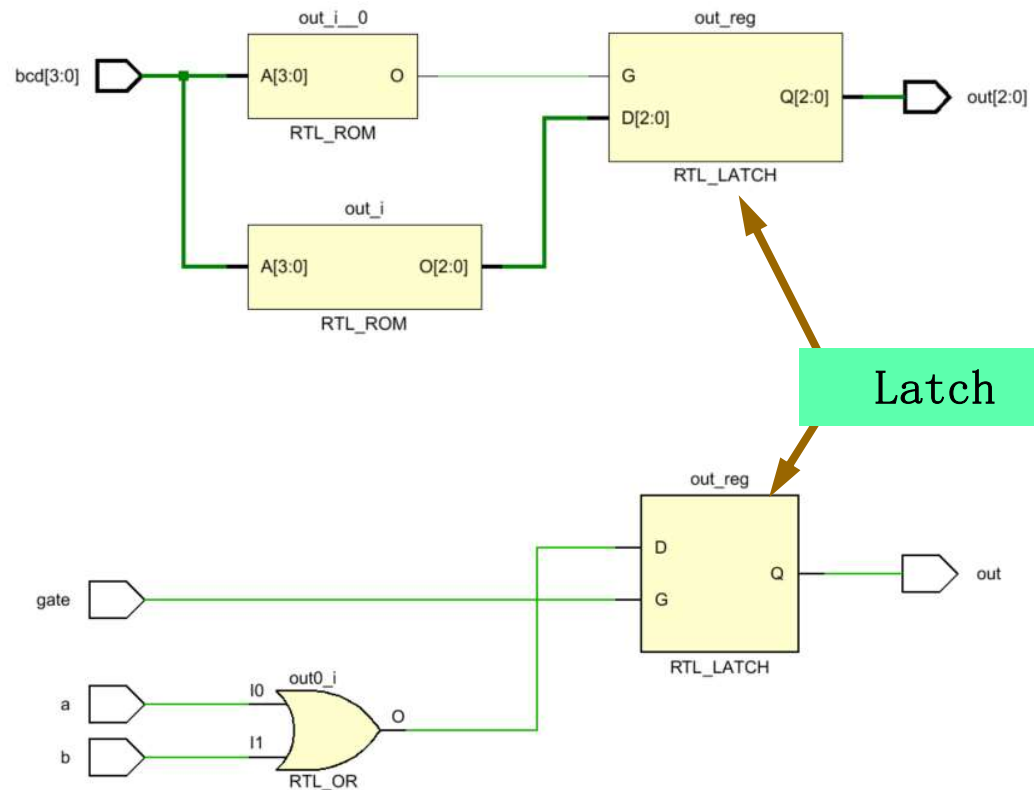
- 在同一个always/initial块里不要混用两种赋值语句
 - 同一个变量，既进行阻塞赋值，又进行非阻塞赋值，综合时会出错
 - 尽量不要在多个不同的always块中对同一变量赋值
-

避免Latch的产生

- 实现组合逻辑电路的always块中if和case语句的分支必须写全

```
always @(bcd) begin
  case ( bcd )
    4' d0 : out = 3' b001;
    4' d1 : out = 3' b010;
    4' d2 : out = 3' b011;
  endcase
end
```

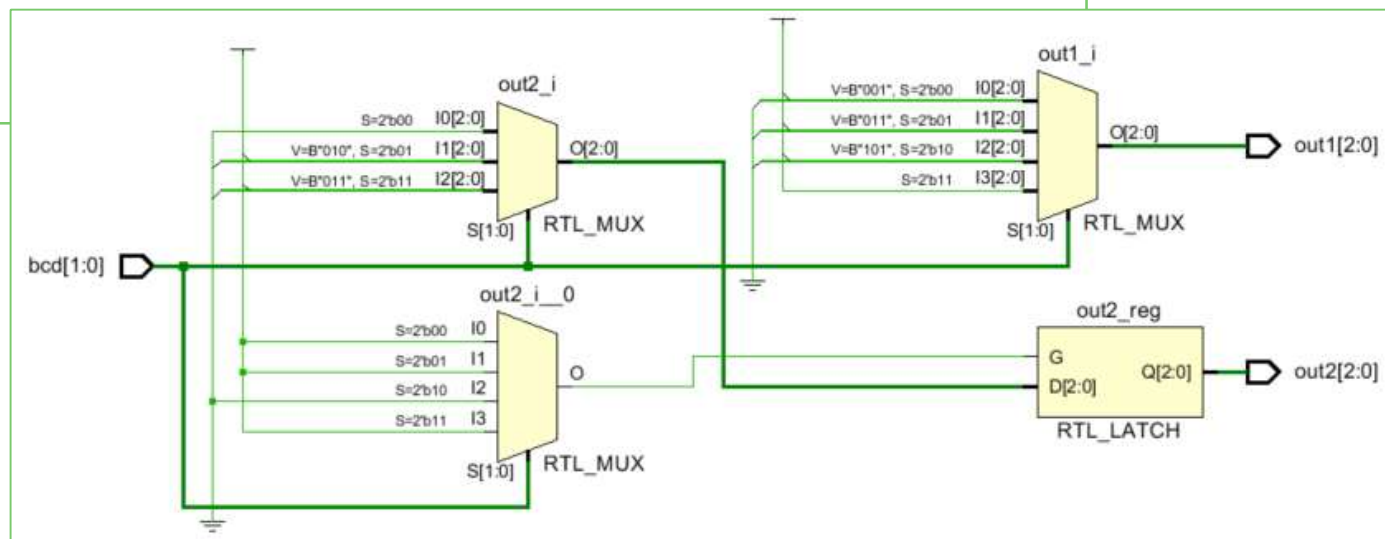
```
always @(a or b or gate )
begin
  if (gate)
    out = a | b;
end
```



避免Latch的产生

```
always @(bcd) begin
  case ( bcd )
    2' b00 : begin out1 = 3' b001; out2=2' b00; end
    2' b01 : begin out1 = 3' b011; out2=2' b10; end
    2' b10 : out1 = 3' b101;
    2' b11 : begin out1 = 3' b111; out2=2' b11; end
  endcase
end
```

没有改变out2
导致产生Latch



编译预处理

- 符号说明一个编译预处理
- 这些编译预处理使仿真编译器进行一些特殊的操作
- 编译预处理一直保持有效直到被覆盖或解除

宏定义- `define

- 提供一种简单的文本替换的功能

`define <macro_name> <macro_text>

在编译时<macro_text>替换<macro_name>。可提高描述的可读性。

```
`define not_delay #1
`define and_delay #2
`define or_delay #1
module MUX2_1 (out, a, b, sel);
output out;
input a, b, sel;
    not `not_delay not1( sel_, sel);
    and `and_delay and1( a1, a, sel_);
    and `and_delay and2( b1, b, sel);
    or `or_delay or1( out, a1, b1);
endmodule
```

定义not_delay

使用not_delay

文本包含-`include

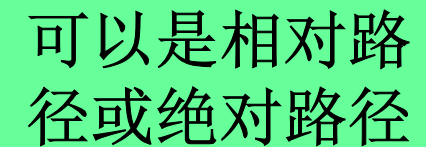
- 在当前内容中插入一个文件

格式: ``include "<file_name>"`

如 ``include "global.v"`

``include "parts/count.v"`

``include "../..../library/mux.v "`



可以是相对路径或绝对路径

- 说明

- `include`保存文件中的全局的或经常用到的一些定义
- 一个``include`命令只能指定一个被包含的文件
- 文件包含是可以嵌套的

目录

1

Verilog 建模方式

2

组合逻辑电路简介

3

常用组合逻辑模块的Verilog描述

4

组合逻辑电路可综合描述的常见问题

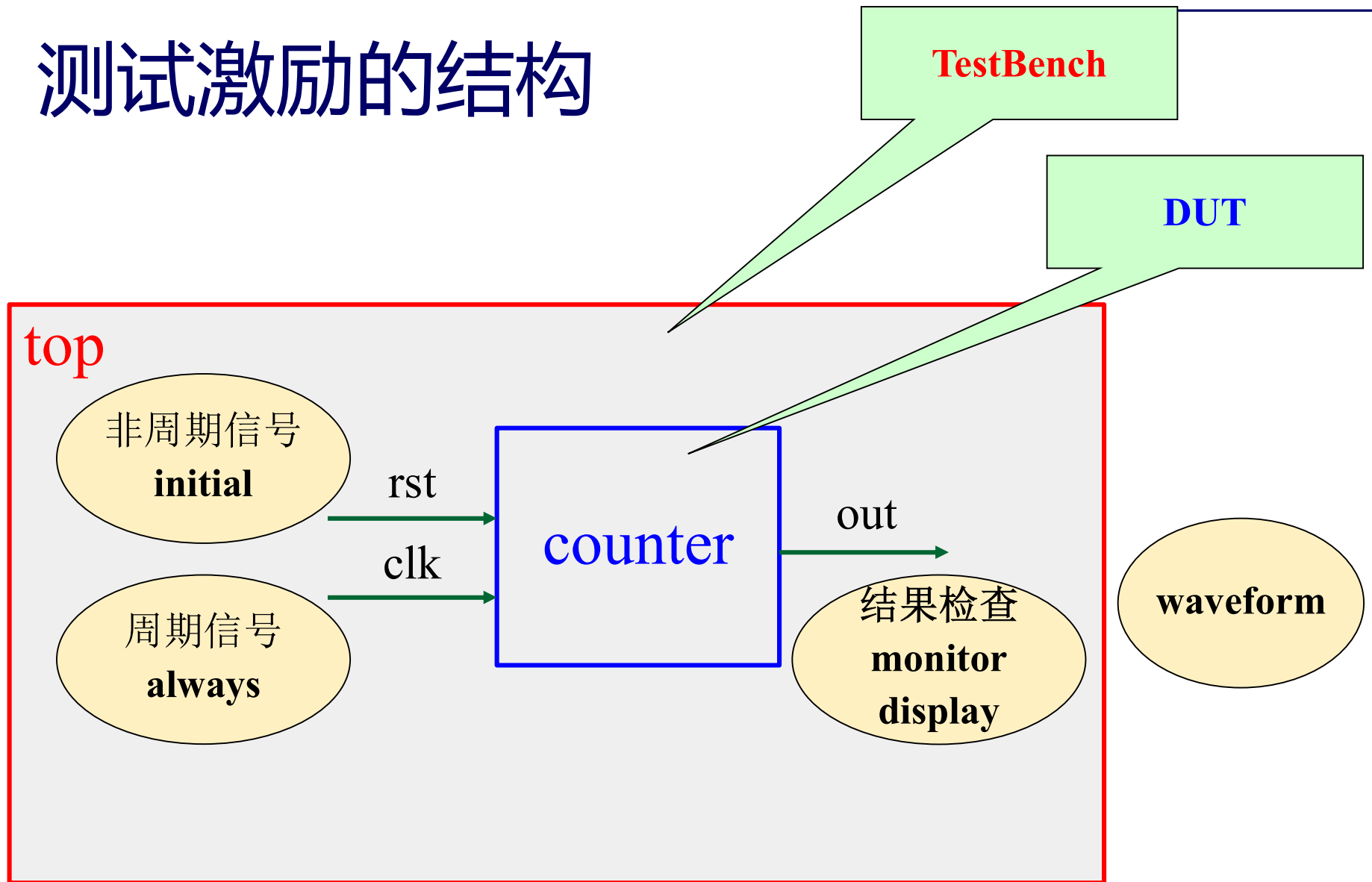
5

测试平台搭建

6

以加法器为例讲述关键路径的时延问题

测试激励的结构



被测模块DUT(divece under test)

```
module counter (out, clk, rst);  
    output reg [3:0] out;  
    input clk, rst;  
    wire clk, rst;  
  
    always @ (negedge rst or posedge clk) begin  
        if(!rst)  
            out<=0;  
        else  
            out<=out+1;  
        end  
    endmodule
```

- 测试模块可以通过模块名及端口说明使用被测模块。实例化时不需要知道其实现细节。这正是自上而下设计方法的一个重要特点
-

测试模块(tb.v)

```
`timescale 10ns/1ns
module top;
    // Data type declaration

    // Instantiate modules
    counter cnt (dout, clk, rst);
    // Apply stimulus

    // Display results

endmodule
```

由于`top`已经是最顶层模块，
不会被其它模块实例化。因此
不需要有端口信号

测试模块 – 激励+输出

```
module top;  
  // Data type declaration  
  reg clk, rst;  
  wire [3:0] dout;  
  // counter instance  
  counter cnt (dout, clk, rst);  
  // Apply stimulus  
  initial begin  
    clk = 0;  
    rst = 1;  
    #15 rst = 0;  
    #10 rst = 1;  
    #55 $finish;  
  end
```

只有always/initial过程的输出才需要定义为reg

端口信号的缺省类型是wire, 可以省略

```
always  
  #10 clk = ~clk;  
// Display results  
initial begin  
  $monitor($time,, "%b %b %b",  
    rst, clk, dout);  
end  
endmodule
```

系统任务和函数

- **\$<identifier>**
- **\$符号指示这是系统任务或函数，如：**
 - 停止仿真\$stop
 - 结束仿真\$finish
 - 返回当前仿真时间\$time
 - 监视信号值\$monitor，若参数列表中的参数值**发生变化**，则在时间单位末显示参数值
 - 显示信号值\$display，在当前时间显示参数值

\$monitor(\$time, o, in1, in2);

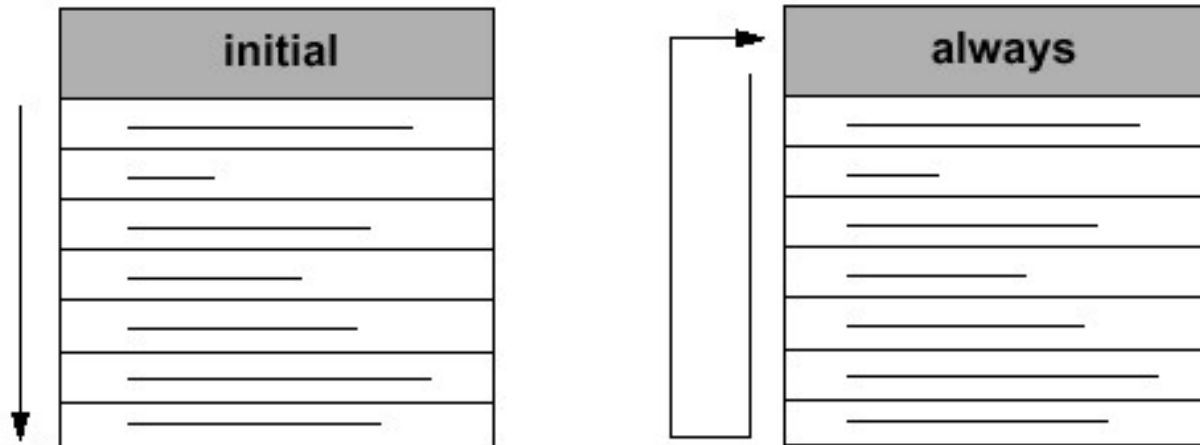
\$monitor(\$time, , out, , a, , b, , sel);

\$monitor(\$time, "%b %h %d %o", sig1, sig2, sig3, sig4);

\$monitor(\$time, "a = %b, b = %h", a, b);

测试模块 – 过程块

- 过程语句有两种
 - **initial** : 只执行一次
 - 赋初值
 - 产生激励信号
 - 检查输出结果
 - **always** : 重复执行
 - 产生周期性激励信号



initial赋初值

```
reg [3:0] out ;  
  
always @(negedge rst  
        or posedge clk)  
begin  
    if ( !rst )  
        out <= 0;  
    else  
        out <= out + 1;  
end
```

```
reg [3:0] out ;  
  
initial out = 0;  
  
always @(posedge clk)  
begin  
    out <= out + 1;  
end
```

- 如果不给信号赋初值？
- 为了仿真，怎么给初值？
 - 初值的各种情况、状态机的各种情况

产生激励信号

■ 周期信号（如时钟）

```
always  
  #10 clk = ~clk;
```

```
initial begin  
  clk = 0;  
  forever  
    #10 clk = ~clk;  
end
```

□ 循环控制语句

- for
- while
- forever
- repeat

■ 复位信号

```
initial begin  
  rst = 1;  
  #15 rst = 0;  
  #10 rst = 1;  
  #55 $finish;  
end
```

```
initial fork  
  rst = 1;  
  #15 rst = 0;  
  #25 rst = 1;  
  #80 $finish;  
join
```

```
initial begin  
  rst <= 1;  
  rst <= #15 0;  
  rst <= #25 1;  
  #80 $finish;  
end
```

□ 并行语句块

- fork join
- 不可综合

检查输出信号

```
`timescale 1ns/1ns
```

```
module top;
```

```
reg in;
```

```
wire out;
```

```
assign #1 out=~in;
```

```
initial begin
```

```
  $monitor($time,,  
    "out=%b in=%b",out,in);
```

```
end
```

```
initial begin
```

```
  in = 0;
```

```
  #10 in = 1;
```

```
  #10 in = 0;
```

```
end
```

```
endmodule
```

```
0 out=x in=0
```

```
1 out=1 in=0
```

```
10 out=1 in=1
```

```
11 out=0 in=1
```

```
20 out=0 in=0
```

```
21 out=1 in=0
```

```
`timescale 1ns/1ns
```

```
module top;
```

```
reg in;
```

```
wire out;
```

```
assign #1 out=~in;
```

```
initial begin
```

```
  $display($time,,  
    "out=%b in=%b",out,in);
```

```
end
```

```
initial begin
```

```
  in = 0;
```

```
  #10 in = 1;
```

```
  #10 in = 0;
```

```
end
```

```
endmodule
```

```
0 out=x in=x
```

时间尺度-timescale

- **`timescale** 说明时间单位及精度

格式: **`timescale** <time_unit> / <time_precision>

如: **`timescale 1 ns / 100 ps**

time_unit: 延时或时间的测量单位

time_precision: 延时值超出精度要先舍入后使用

- **`timescale** 必须在模块之前出现

```
`timescale 1 ns / 10 ps
```

```
// All time units are in multiples of 1 nanosecond
```

```
module MUX2_1 (out, a, b, sel);
```

```
output out;
```

```
input a, b, sel;
```

```
not #1 not1( sel_, sel);
```

```
and #2 and1( a1, a, sel_);
```

```
and #2 and2( b1, b, sel);
```

```
or #1 or1( out, a1, b1);
```

```
endmodule
```

时间尺度

- **time_precision**不能大于**time_unit**
 - **time_precision**和**time_unit**的表示方法: **integer unit_string**
 - **integer**: 可以是1, 10, 100
 - **unit_string**: 可以是s(second), ms(millisecond), us(microsecond), ns(nanosecond), ps(picosecond), fs(femtosecond)
 - 以上**integer**和**unit_string**可任意组合
 - 尽可能地使精度与时间单位接近, 只要满足设计的实际需要就行
 - **precision**是仿真器的仿真时间步长
 - 若**time_unit**与**precision_unit**差别很大将严重影响仿真速度。
 - 如说明一个`**timescale 1s / 1ps**, 则仿真器在1秒内要扫描其事件序列 10^{12} 次; 而`**timescale 1s/100ms**则只需扫描10次。
-

时间尺度

- 所有**timescale**中的最小值决定仿真时的最小时间单位。因为仿真器必须对整个设计进行精确仿真

```
`timescale 10ns/ 1ns
module1 (...);
    #1.23      // 12ns
    ...
endmodule

`timescale 100ns/ 1ns
module2 (...);
    #1.23      // 123ns
    ...
endmodule

`timescale 1ps/ 100fs
module3 (...);
    #1.23      // 1.23ps
    ...
endmodule
```

目录

1

Verilog 建模方式

2

组合逻辑电路简介

3

常用组合逻辑模块的Verilog描述

4

组合逻辑电路可综合描述的常见问题

5

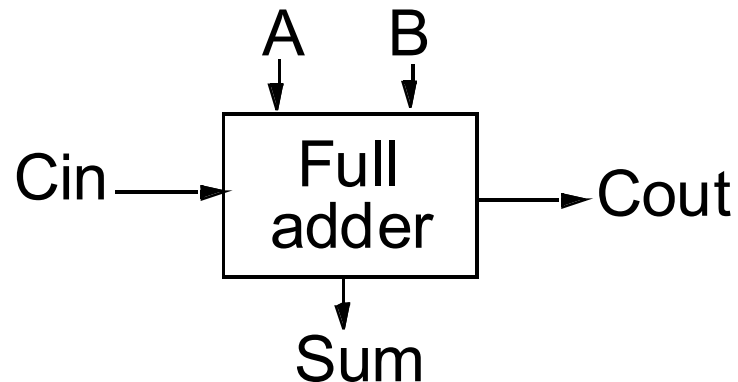
测试平台搭建

6

以加法器为例讲述关键路径的时延问题

组合逻辑应用—加法器设计

加法器常常是限制速度的部件。加法器的优化可在逻辑级和电路级进行

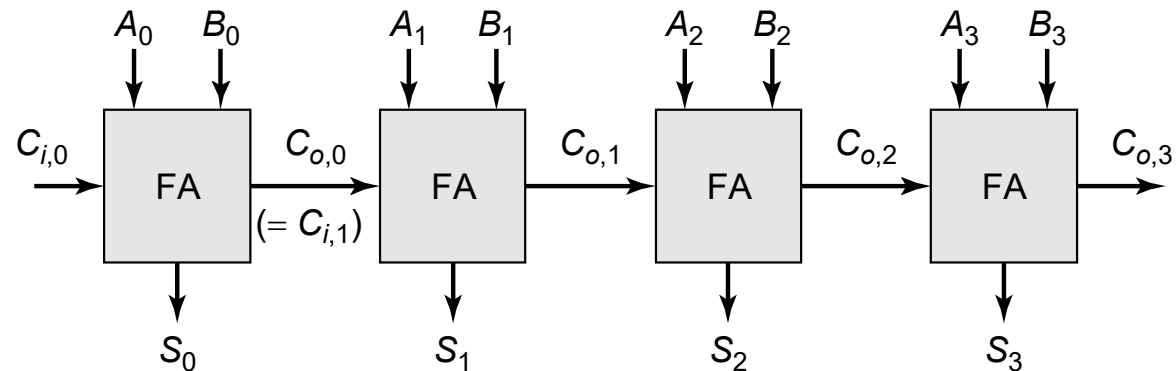


$$\begin{aligned} S &= A \oplus B \oplus C_i \\ &= A\bar{B}\bar{C}_i + \bar{A}B\bar{C}_i + \bar{A}\bar{B}C_i + ABC_i \\ C_o &= AB + BC_i + AC_i \end{aligned}$$

设计示例

用一位全加器组成四位全加器

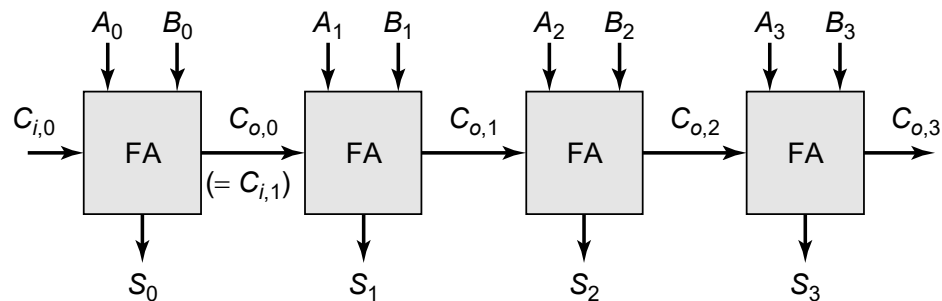
```
module FullAdder (A, B, Ci, S, Co);  
    input  A, B, Ci;  
    output S, Co;  
  
    assign S = A ^ B ^ Ci;  
    assign Co = (A & B) | (A & Ci) | (B & Ci);  
endmodule
```



设计示例（续）

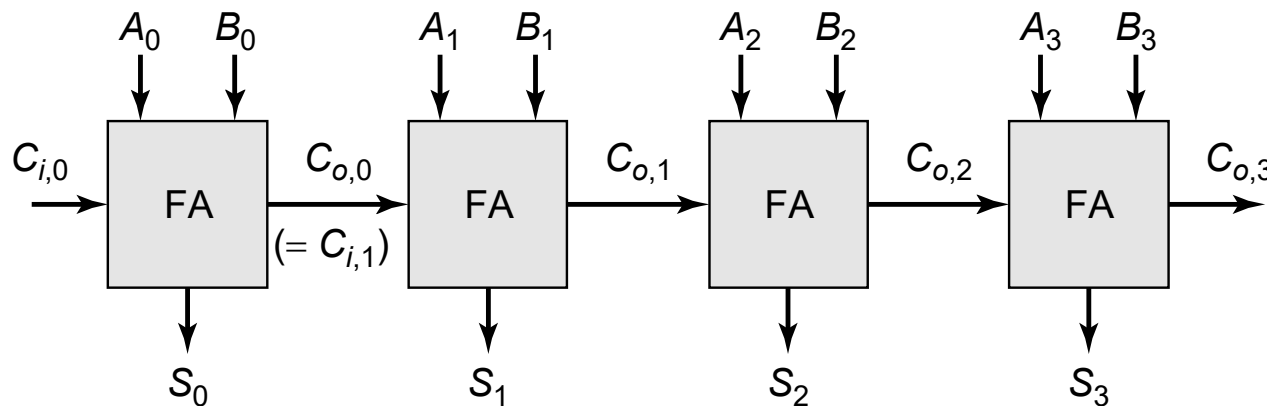
用一位全加器组成四位全加器

```
module ADDER4BIT ( Ai, Bi, S, OVF);  
    input    [3:0] Ai, Bi;  
    output   [3:0] S;  
    output   OVF;  
    wire     [2:0] CY;  
    FullAdder U0 (Ai[0], Bi[0], 0, S[0], CY[0]);  
    FullAdder U1 (Ai[1], Bi[1], CY[0], S[1], CY[1]);  
    FullAdder U2 (Ai[2], Bi[2], CY[1], S[2], CY[2]);  
    FullAdder U3 (Ai[3], Bi[3], CY[2], S[3], OVF);  
endmodule
```



逐位进位（Ripple-Carry）加法器

- (1) 结构：由N个一位加法器串联而成，第i级的Carry-out用来产生第i+1级的SUM和Carry
- (2) 特点：结构直观简单，但因高位运算必须等低位进位来到后才能进行，故运行速度慢



最坏情况下关键路径的延时：

$$t_{adder} = (N-1)t_{carry} + t_{sum}$$

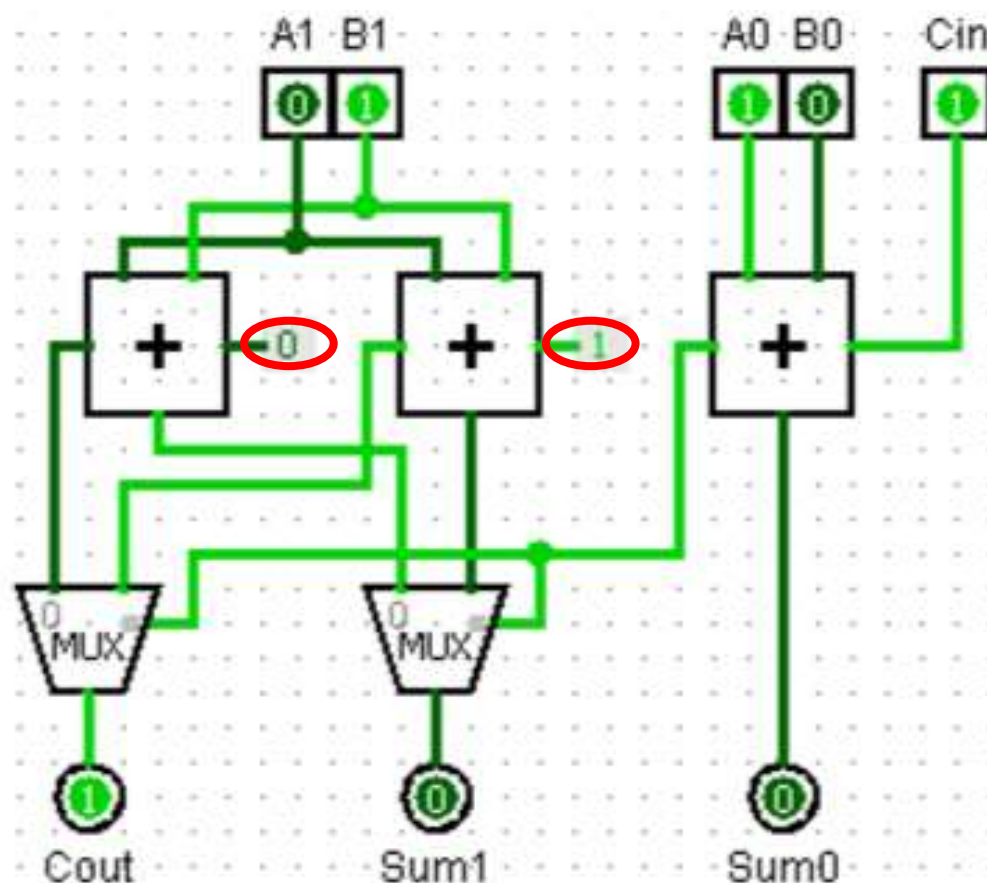
可见运算的延迟是由于进位的延迟。进位的解决是核心问题。

进位选择（Carry-Select）加法器

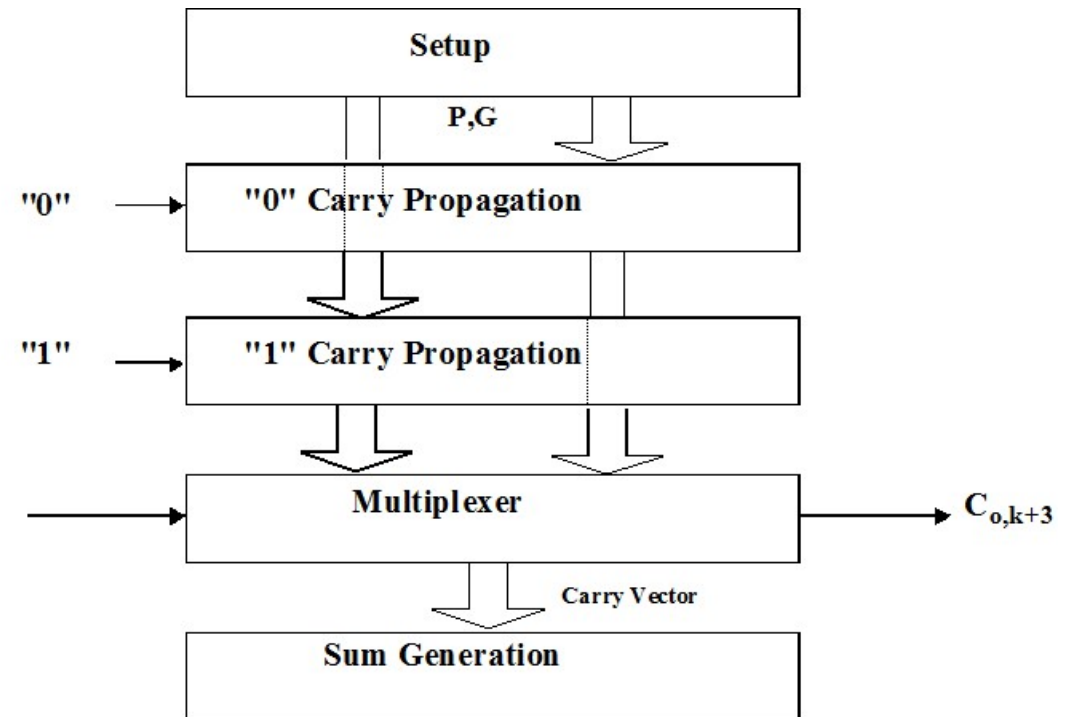
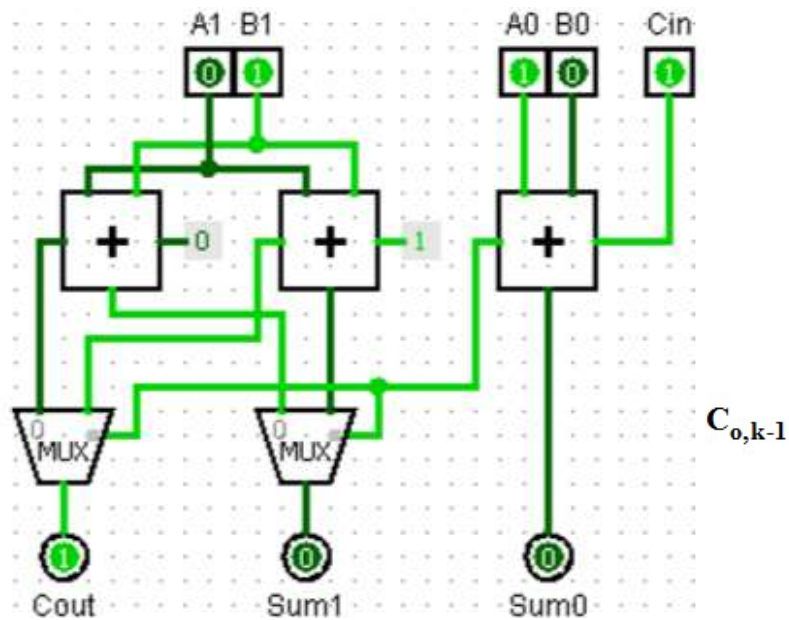
两个逐位进位加法器的进位链构成：

一个最低位进位 $C_{in}=0$ ，另一个最低位进位 $C_{in}=1$ 。这两个加法器的进位链分别计算出对应不同 C_{in} 值的两个“进位输出”。

一个MUX选择这两个“进位输出”中的一个作为最终的“进位输出”，这个MUX的控制信号是前级的进位输出Carry-out

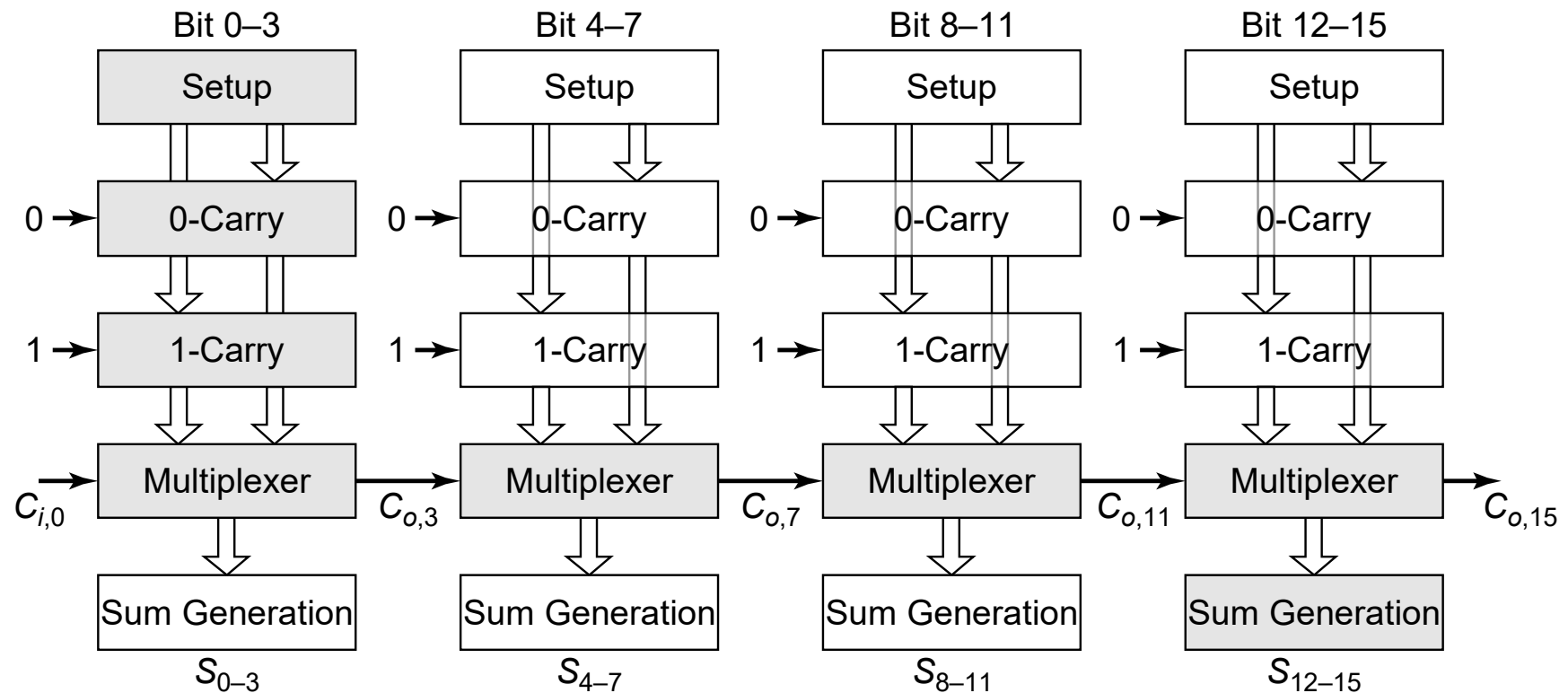


进位选择 (Carry-Select) 加法器



进位选择加法器的关键路径与求和时间

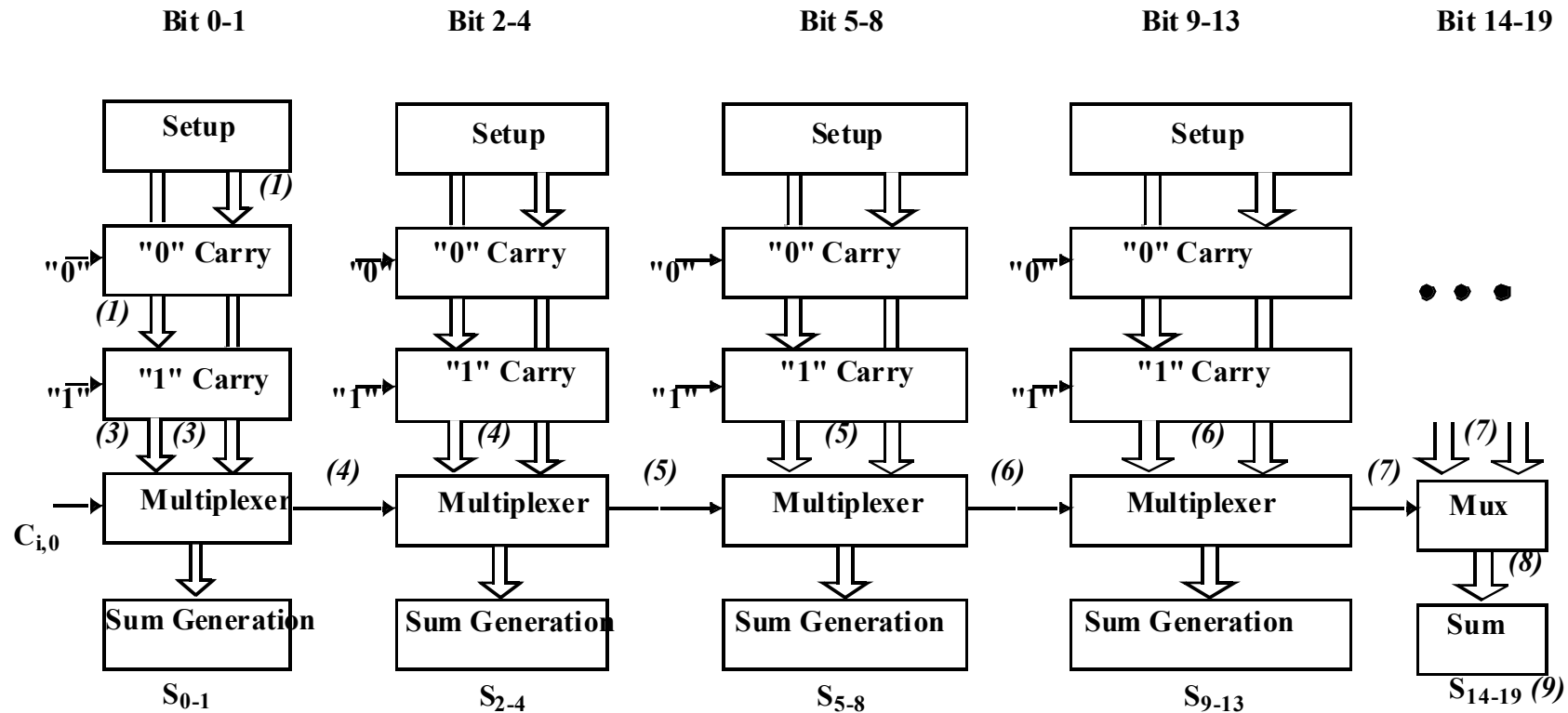
线性进位选择加法器的总进位传播时间与位数N成正比，但比逐位进位加法器快。
硬件开销为一个额外的进位路径和一个MUX，大约比一个逐位进位加法器多30%



$$t_{add} = t_{setup} + \left(\frac{N}{M}\right)t_{carry} + Mt_{mux} + t_{sum}$$

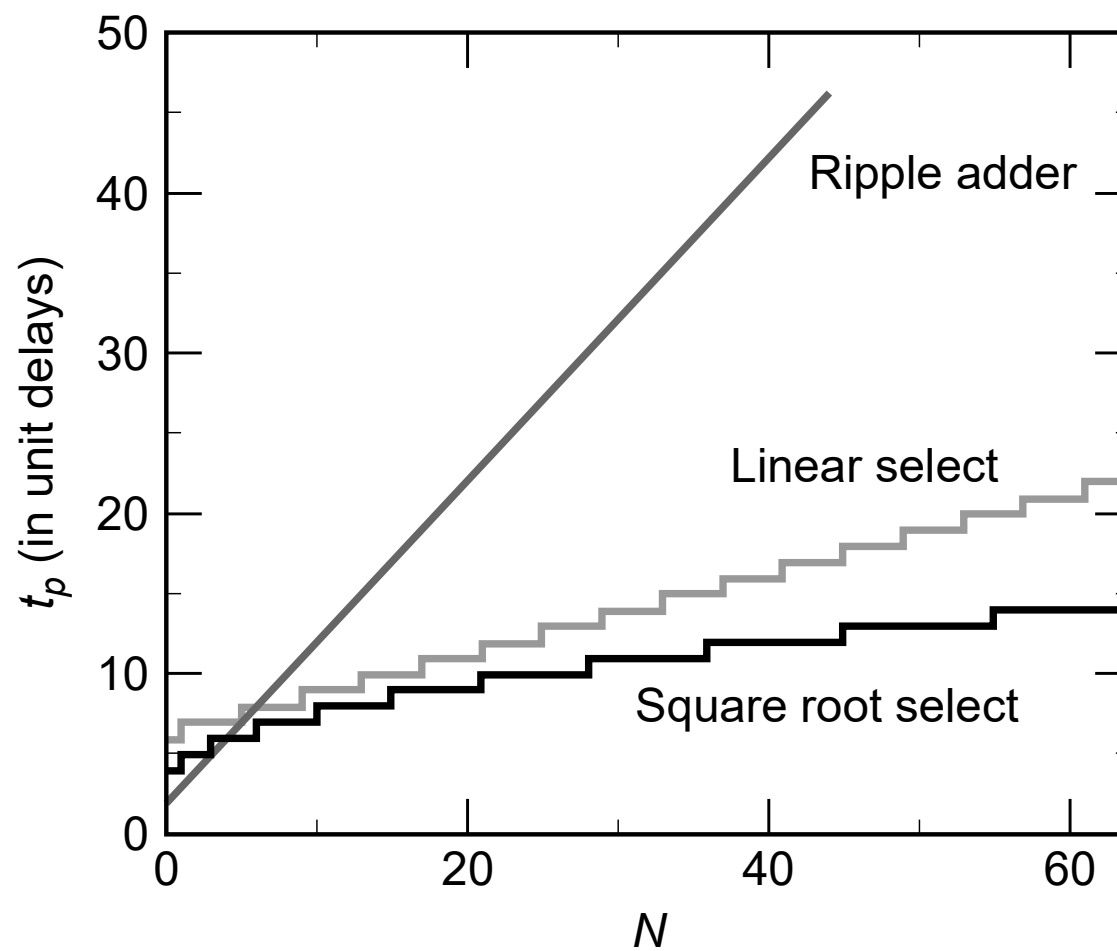
平方根进位选择加法器的求和时间

考虑到前级进位输出要经过一个**MUX**才到达本级的进位输入，因此在两条信号路径之间相差一个延时时间，故本级的位数可以比前一级多一位



$$t_{add} = t_{setup} + P \cdot t_{carry} + (\sqrt{2N})t_{mux} + t_{sum}$$

加法器延时比较



进位产生、进位传播信号

为了利于具体实现，常常定义一些中间信号（注意它们与 C_{in} 无关）

进位产生(**Generate**)信号: $G = AB$

进位传播(**Propagate**)信号: $P = A \oplus B$

这样可以建立起 S 和 C_o 与 G 、 P 之间的关系:

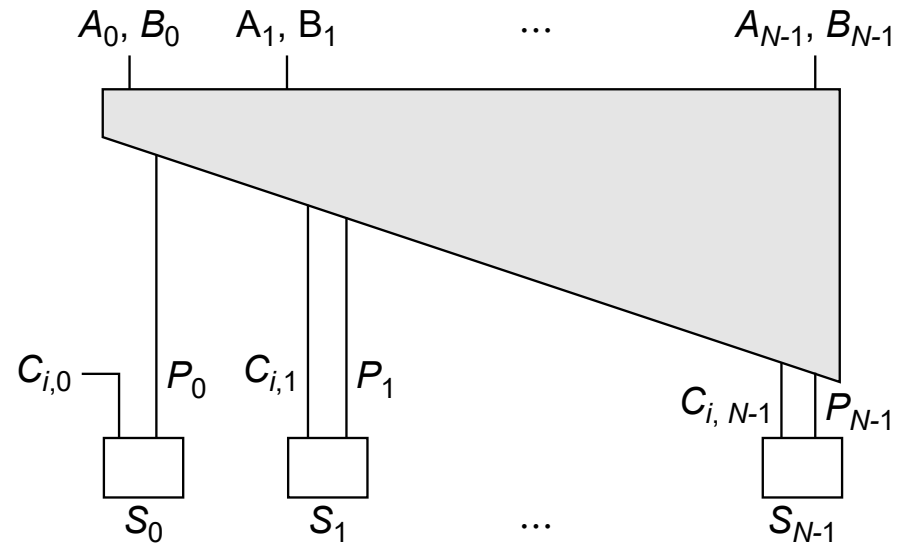
$$C_o(G, P) = G + PC_i$$

$$S(G, P) = P \oplus C_i$$

A	B	C_i	S	C_o	<i>Carry status</i>
0	0	0	0	0	delete
0	0	1	1	0	delete
0	1	0	1	0	propagate
0	1	1	0	1	propagate
1	0	0	1	0	propagate
1	0	1	0	1	propagate
1	1	0	0	1	generate
1	1	1	1	1	generate

超前进位的基本原理

- 每一位的进位输出只与最初的进位输入有关
- 各位进位信号同时形成
- 但与门的扇入= $n+1$ ，或门的扇入= $n+1$
- 适合 n 较小的时候 ($n \leq 4$)



$$C_1 = G_1 + P_1 C_0$$

$$C_2 = G_2 + P_2 C_1$$

$$= G_2 + P_2 G_1 + P_2 P_1 C_0$$

$$C_n = G_n + P_n C_{n-1}$$

$$= G_n + P_n G_{n-1} + \dots + P_n P_{n-1} \dots P_2 P_1 C_0$$

4位超前进位加法器

- 进位产生逻辑:

$$C[1]=G_0+P_0\cdot C[0]$$

$$C[2]=G_1+P_1\cdot G_0+P_1\cdot P_0\cdot C[0]$$

$$C[3]=G_2+P_2\cdot G_1+P_2\cdot P_1\cdot G_0+P_2\cdot P_1\cdot P_0\cdot C[0]$$

$$C[4]=G_3+P_3\cdot G_2+P_3\cdot P_2\cdot G_1+P_3\cdot P_2\cdot P_1\cdot G_0+P_3\cdot P_2\cdot P_1\cdot P_0\cdot C[0]$$

- 各位的进位相互独立，进一步减小了延迟

思考：16位超前进位加法器如何实现
