

# 组合逻辑电路设计

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

**以加法器为例讲述关键路径的时延问题**

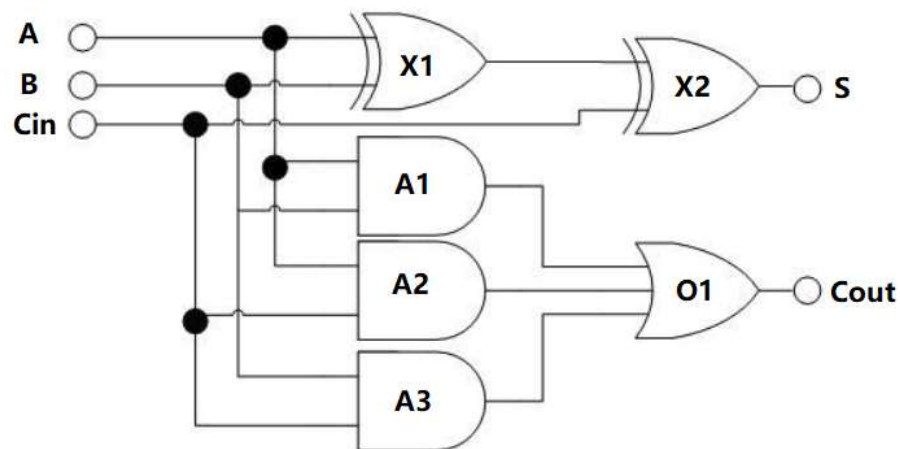
---

# Verilog 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
-

# 全加器



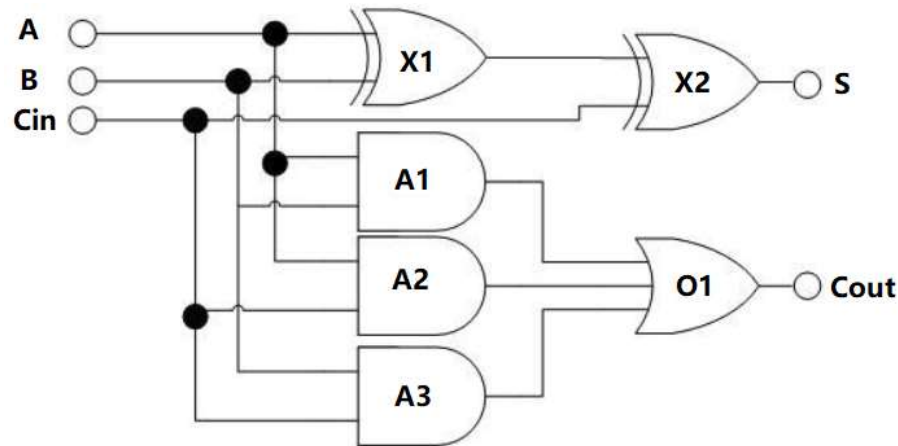
$C_{in}$	A	B	S	$C_{out}$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AC_{in} + BC_{in} + AB$$

# 结构化建模方式

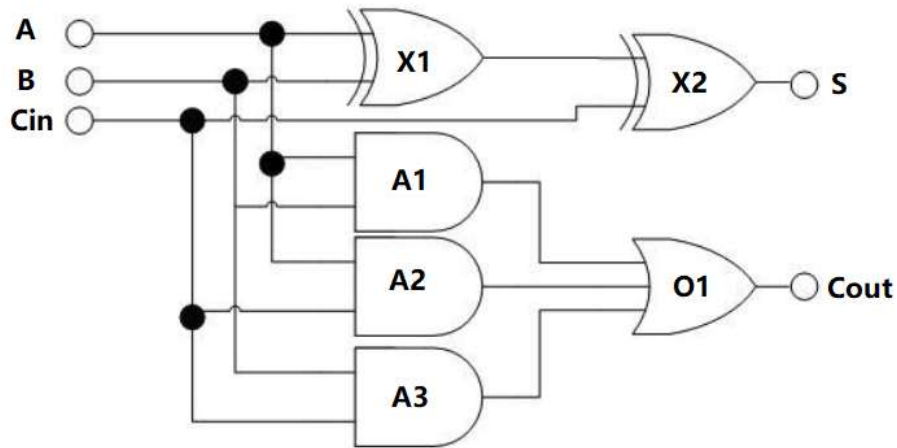
- 通过对电路的层次和组成结构进行描述，并使用线网来连接各器件来描述一个模块的结构。



```
module FA_struct (A, B, Cin,
Sum, out);
input  A;
input  B;
input  Cin;
output Sum;
output Cout;
wire  S1, T1, T2, T3;
// -- statements -- //
xor  x1 (S1, A, B);
xor  x2 (Sum, S1, Cin);
and  A1 (T1, A, B );
and  A2 (T2, B, Cin);
and  A3 (T3, A, Cin);
or   O1 (Cout, T1, T2, T3 );
endmodule
```

# 数据流建模方式

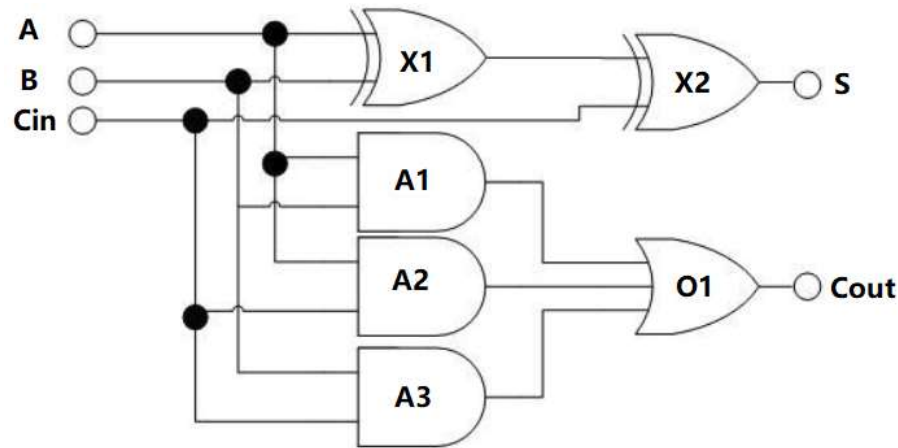
- 对数据流的具体行为的描述
- 连续赋值语句
- 比较复杂的连续赋值语句，描述一些较复杂的线网变量的行为



```
module
FA_flow(A,B,Cin,Sum,Cout)
input    A,B,Cin;
output   Sum, Cout;
wire    S1,T1,T2,T3;
assign   S1 = A ^ B;
assign   Sum = S1 ^ Cin;
assign   T1 = A & B;
assign   T2 = B & Cin;
assign   T3 = A & Cin ;
assign   Cout = T1|T2|T3;
endmodule
```

# 行为描述方式建模

- 只关心电路具有什么样的功能，不关心电路结构（组成和连接）
- 一般，用**initial**或**always**块语句描述，可以使用如**if else**、**case**、**for**、**while**等等，描述复杂的电路。



$$S = A \oplus B \oplus Cin$$

$$Cout = ACin + BCin + AB$$

```
module FA_behav1(A, B, Cin,  
Sum, Cout );  
input    A,B,Cin;  
output   Sum,Cout;  
reg     Sum, Cout;  
reg     T1,T2,T3;  
always@ ( A or B or Cin )  
begin
```

**{Cout, Sum} <= A+B+Cin;**

```
end  
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

**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

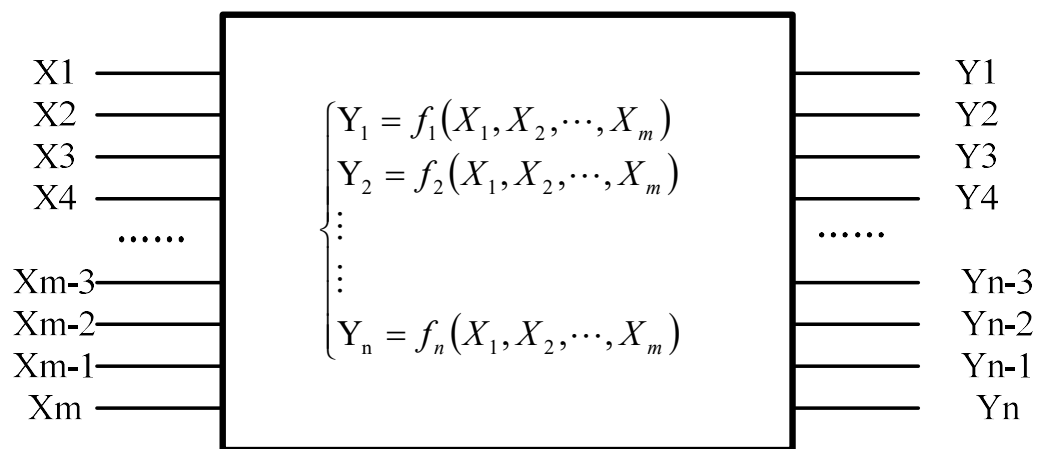
---

- 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的主要区别是什么？**

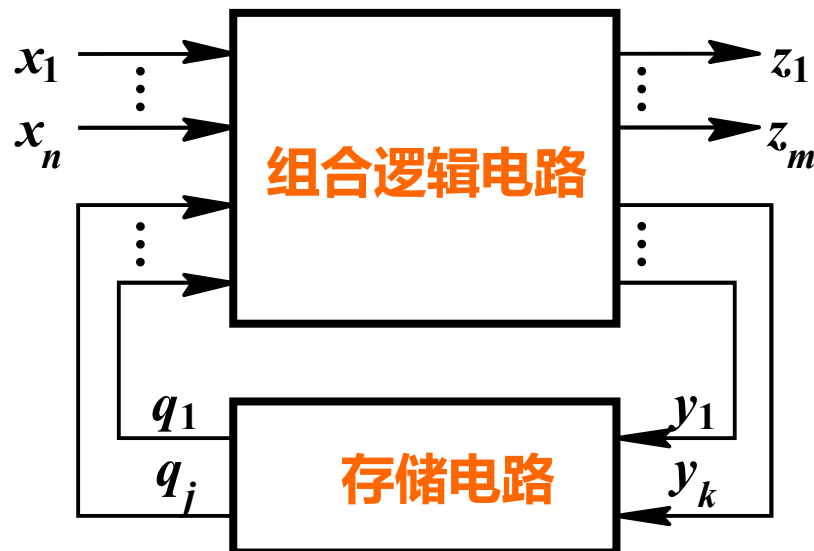
# 组合逻辑电路：电路无记忆功能

- 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）



# 时序电路：有存储器

- 还和电路的内部状态有关
- 还有寄存器等存储单元，  
电路内容有反馈



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

□ 使用**assign**描述

简单逻辑函数

□ 使用**always**描述

复杂组合逻辑

```
assign q = (a1==1?) d : 0 ;
```

```
always @(a1 or d)
begin
    if (a1==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**

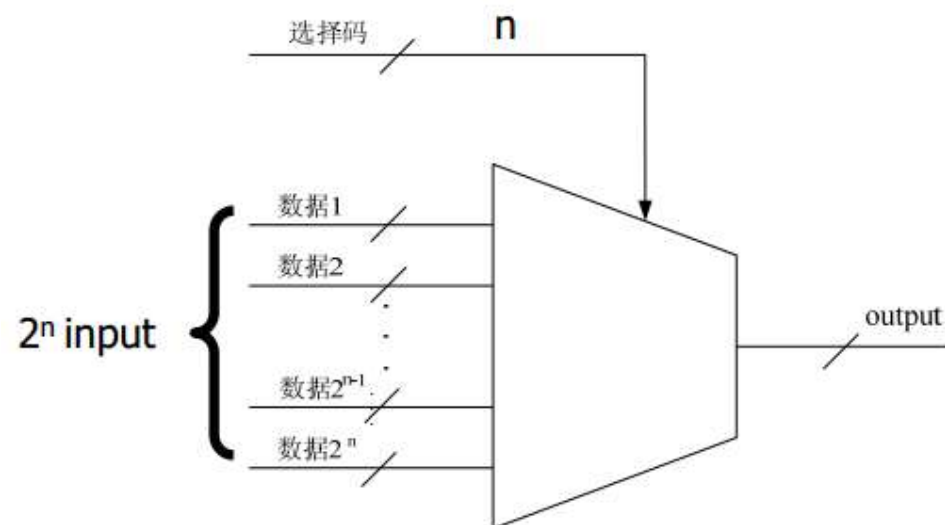
5

**以加法器为例讲述关键路径的时延问题**

---

# 多路选择器

多路选择器是一个多输入、单输出的组合逻辑电路。它根据选择码，从多个输入数据流中选取一个，输出到公共输出端。



其典型Verilog描述语句有以下三种：

- assign语句及条件操作符；
- if...else及其嵌套语句；
- case多分支语句。

# 多路选择器

## ■ assign语句及条件运算符

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

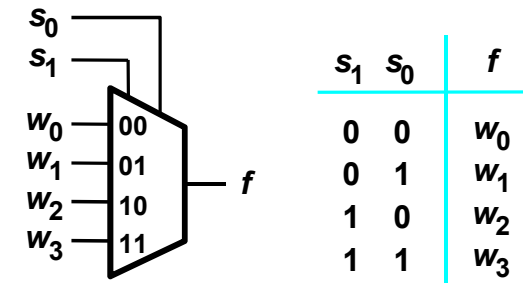
```
    input w0, w1, w2, w3;
```

```
    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

## ■ 采用 if else 分支语句

```
module mux4to1 (w0, w1, w2, w3, S, f);  
    input w0, w1, w2, w3;  
    input [1:0] S;  
    output reg f;  
  
    always @(*)  
        if (S == 2'b00)  
            f = w0;  
        else if (S == 2'b01)  
            f = w1;  
        else if (S == 2'b10)  
            f = w2;  
        else if (S == 2'b11)  
            f = w3;  
endmodule
```

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

s <sub>1</sub>	s <sub>0</sub>	f
0	0	w <sub>0</sub>
0	1	w <sub>1</sub>
1	0	w <sub>2</sub>
1	1	w <sub>3</sub>

(b) Truth table



---

## ■ 采用 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
```

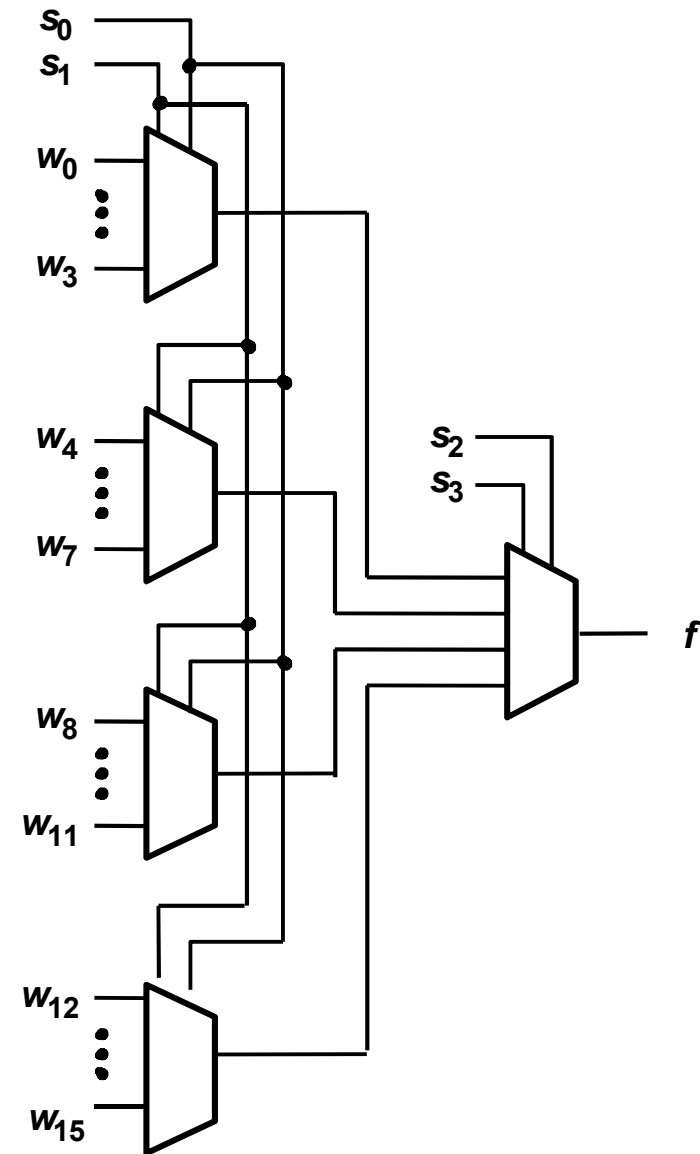
---

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

(b) Truth table

## ■ 结构化描述实现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
```



---

# 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
- 编码器用来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

---

# 描述译码器/编码器的语句

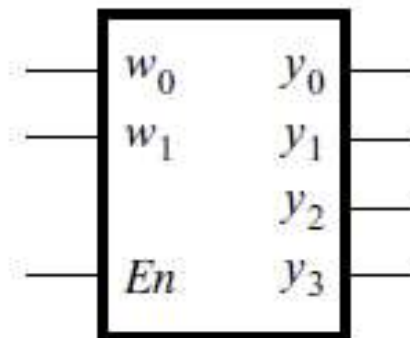
- 通常，采用下列语句描述：
- **if...else**及其嵌套结构
- **case/casex/casez**结构
- **for**循环结构（初学者不建议）

## 2-4译码器

```
module dec2to4 (W, En, Y);  
    input [1:0] W;  
    input En;  
    output reg [0:3] Y;  
  
    always @(W, En)  
        case ({En, W})  
            3'b100: Y = 4'b1000;  
            3'b101: Y = 4'b0100;  
            3'b110: Y = 4'b0010;  
            3'b111: Y = 4'b0001;  
            default: Y = 4'b0000;  
        endcase  
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译码器的另一种实现方式

```

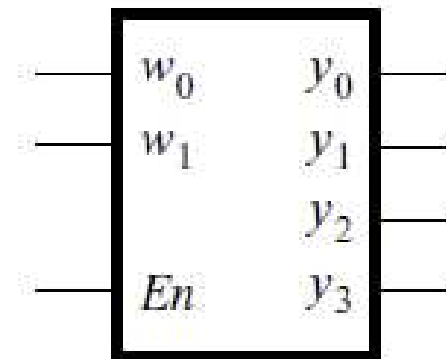
module dec2to4 (W, En, Y);
    input [1:0] W;
    input En;
    output reg [0:3] Y;

    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

```

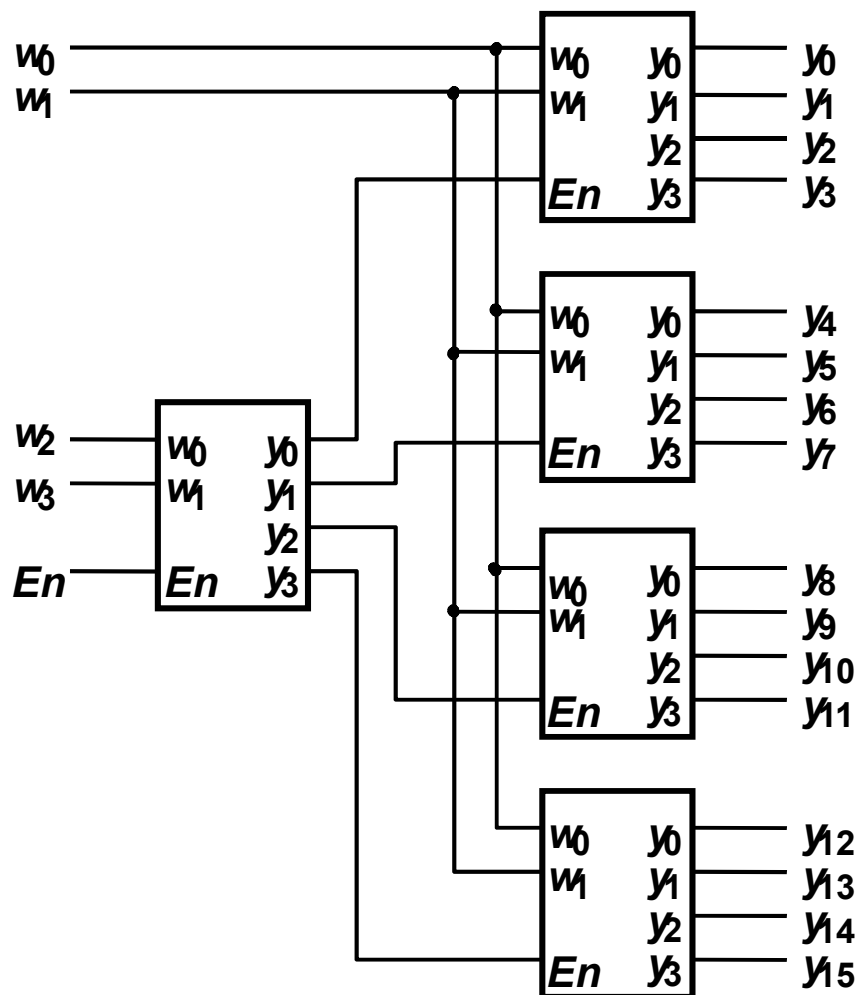
$En$	$w_1$	$w_0$	$y_0$	$y_1$	$y_2$	$y_3$
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

(a) Truth table



(b) Graphical symbol

## ■ 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], M[0:3], En);
```

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

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

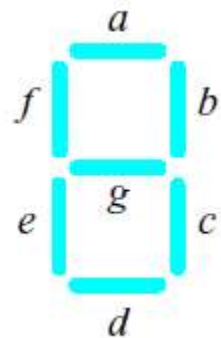
```
    dec2to4 Dec4 (W[1:0], Y[8:11], M[2]);
```

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

```
endmodule
```



# 代码转化器



$w_3$	$w_2$	$w_1$	$w_0$	$a$	$b$	$c$	$d$	$e$	$f$	$g$
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
```

# 优先级编码器

$w_3$	$w_2$	$w_1$	$w_0$	$y_1$	$y_0$
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

```
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**语句实现编解码模块
- 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

Operation	Inputs			Outputs F
	s <sub>2</sub>	s <sub>1</sub>	s <sub>0</sub>	
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 XOR B
OR	1	0	1	A OR B
AND	1	1	0	A 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(s, A, B, F); // 74381 ALU
```

```
input [2:0] S;
```

```
input [3:0] A, B;
```

```
output reg [3:0]
```

```
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

**以加法器为例讲述关键路径的时延问题**

---

---

# 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
- 综合不支持延迟语句 **#**

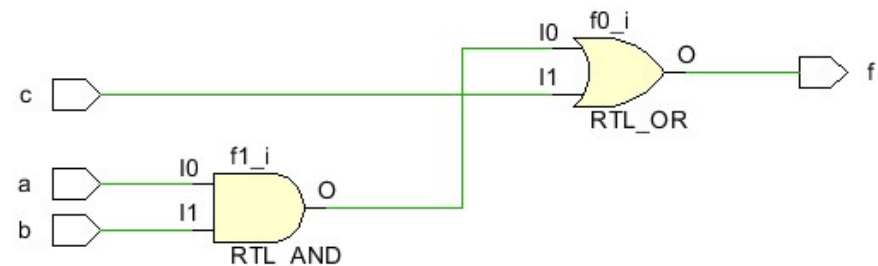
# always块的敏感表

- 实现组合逻辑电路的**always**块敏感表必须写全，否则仿真结果和综合结果会不一致
- 所有赋值号右边出现的信号都要放到敏感表里

```
always @(a or b or c)
    f = a&&b || c ;
```

```
always @(a or b)
    f = a&&b || c ;
```

```
always @(*)
    f = a&&b || c ;
```



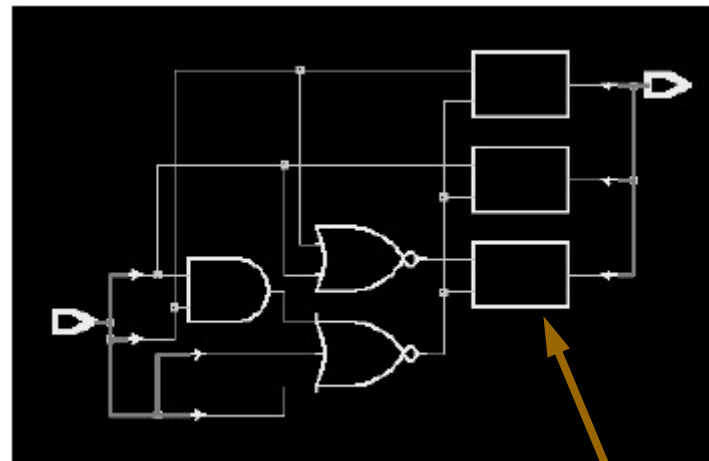


# 避免Latch的产生

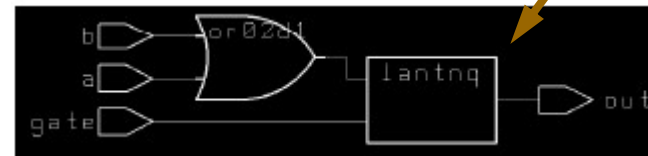
- 实现组合逻辑电路的**always**块中**if**和**case**语句的分支必须写全。

```
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
```

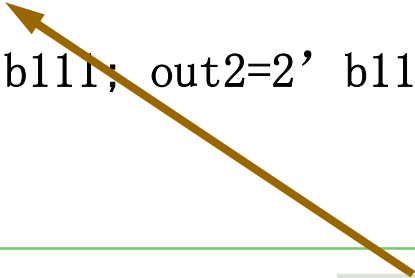


Latch



# 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

# 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

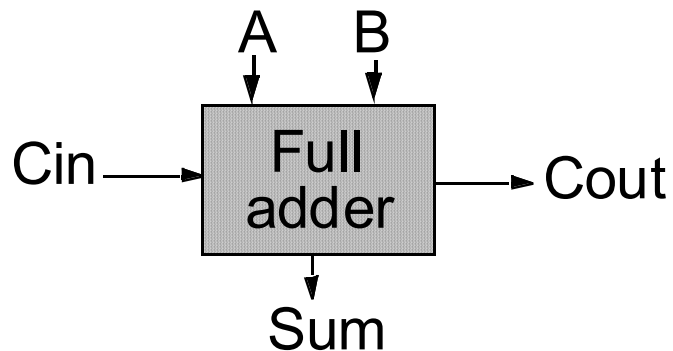
**以加法器为例讲述关键路径的时延问题**

---

# 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

全加器 (**Full-Adder**)



$A$	$B$	$C_i$	$S$	$C_o$	<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

---

# 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）

---

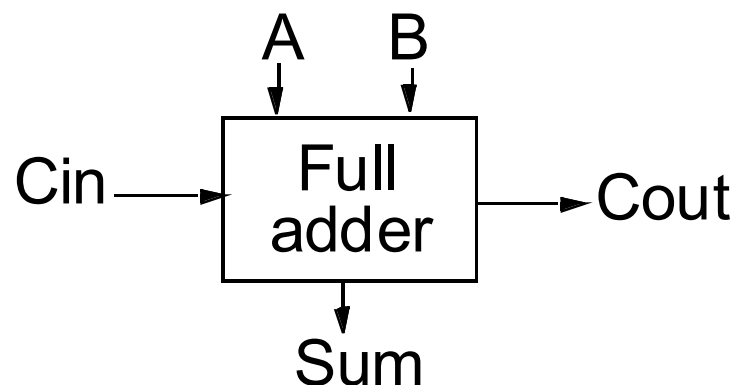
### 用一位全加器组成四位全加器

```
module ADDER4BIT ( Ai, Bi, S, OVF);  
    input [3:0] Ai, Bi;  
    output      [3:0] S;  
    wire[2:0] CY;  
    output      OVF;  
    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
```

---

# 简单加法器：二进制加法运算

---



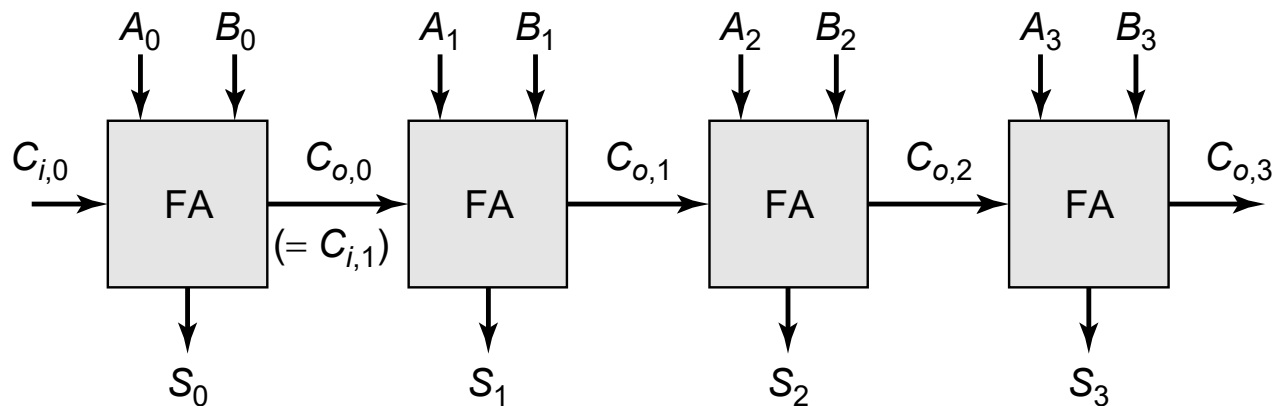
$$\begin{aligned} S &= A \oplus B \oplus C_i \\ &= A\bar{B}\bar{C}_i + \bar{A}B\bar{C}_i + \bar{A}\bar{B}C_i + ABC_i \end{aligned}$$

$$C_o = AB + BC_i + AC_i$$

---

# 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。

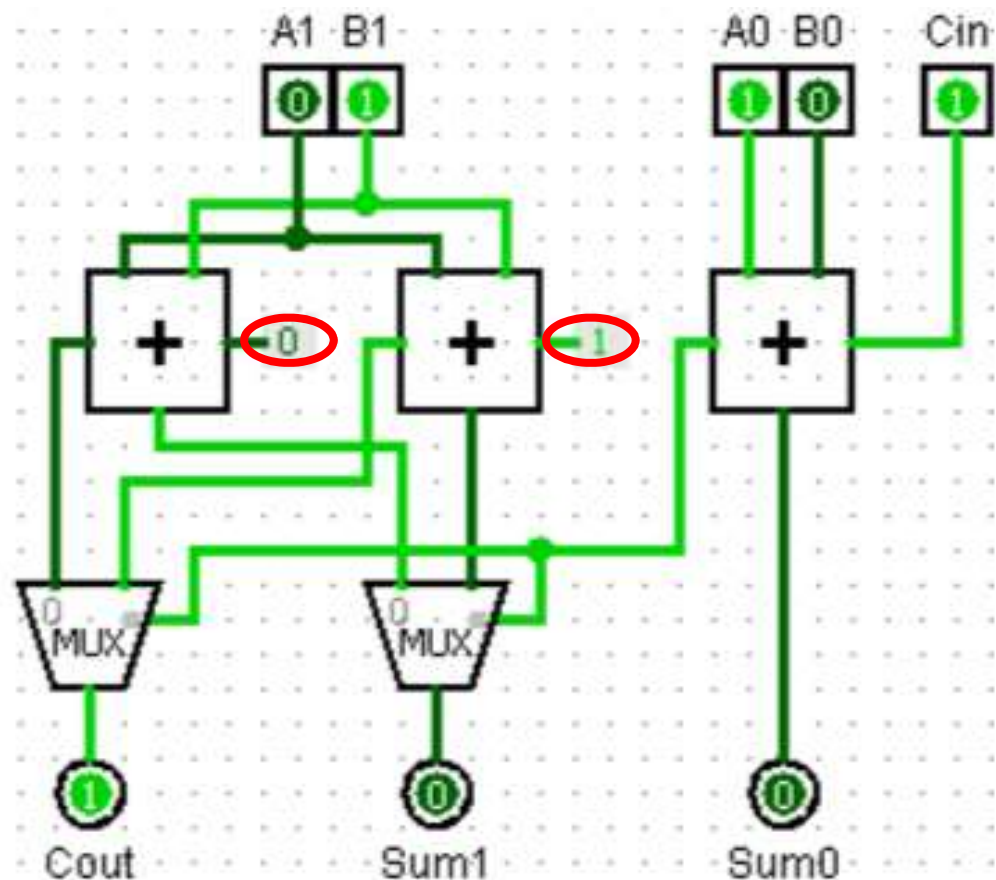


# 进位选择（Carry-Select）加法器

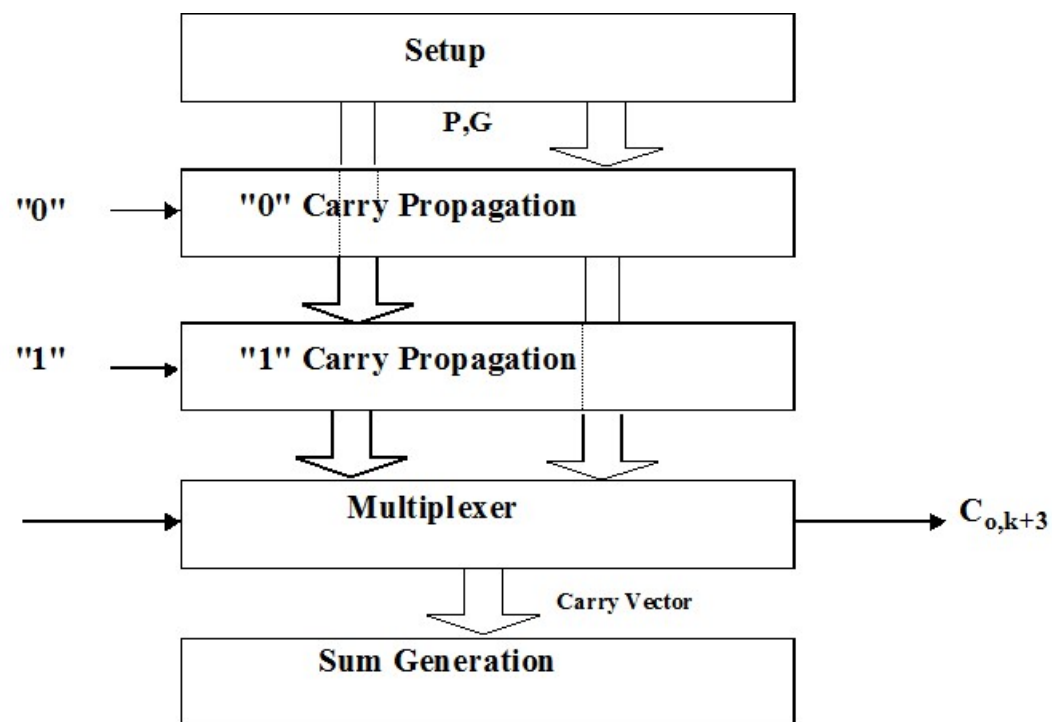
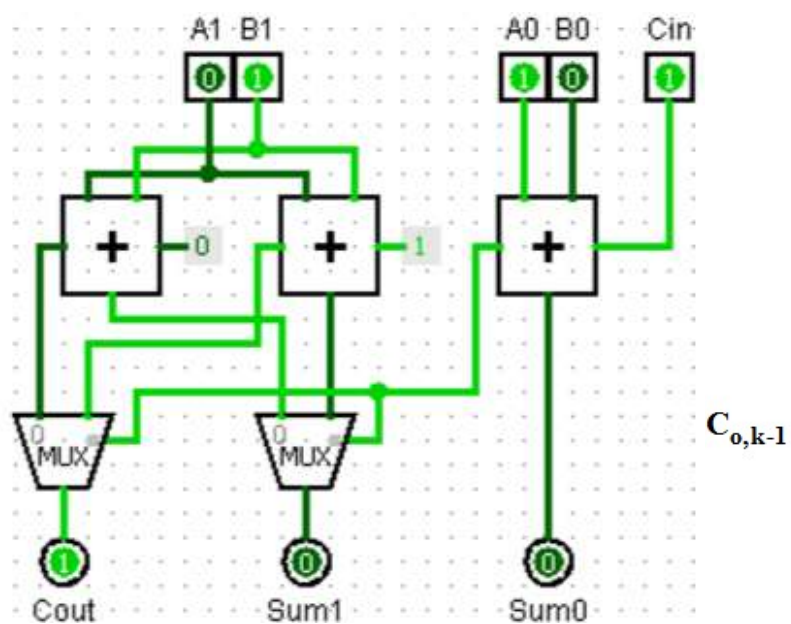
两个逐位进位加法器的进位链构成：

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

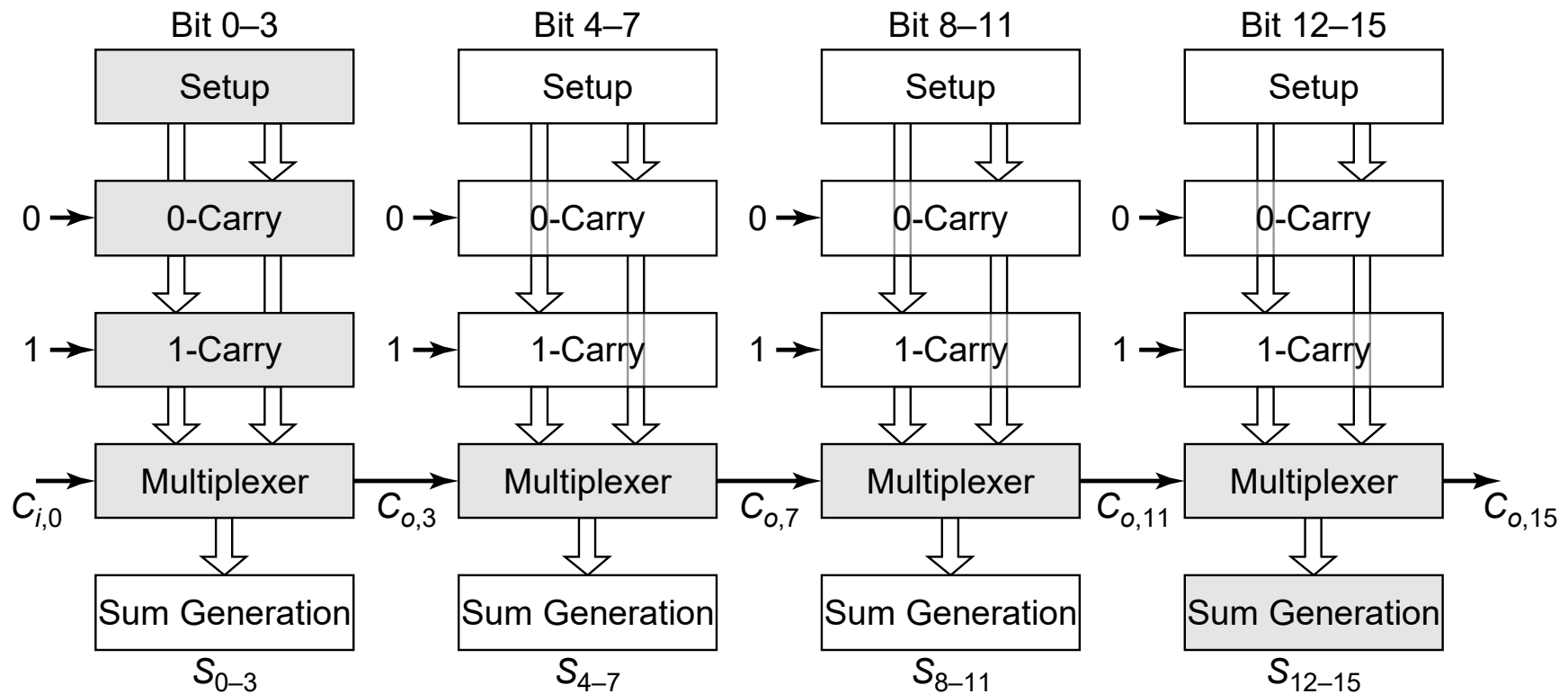


# 进位选择 (Carry-Select) 加法器



# 进位选择加法器的关键路径与求和时间

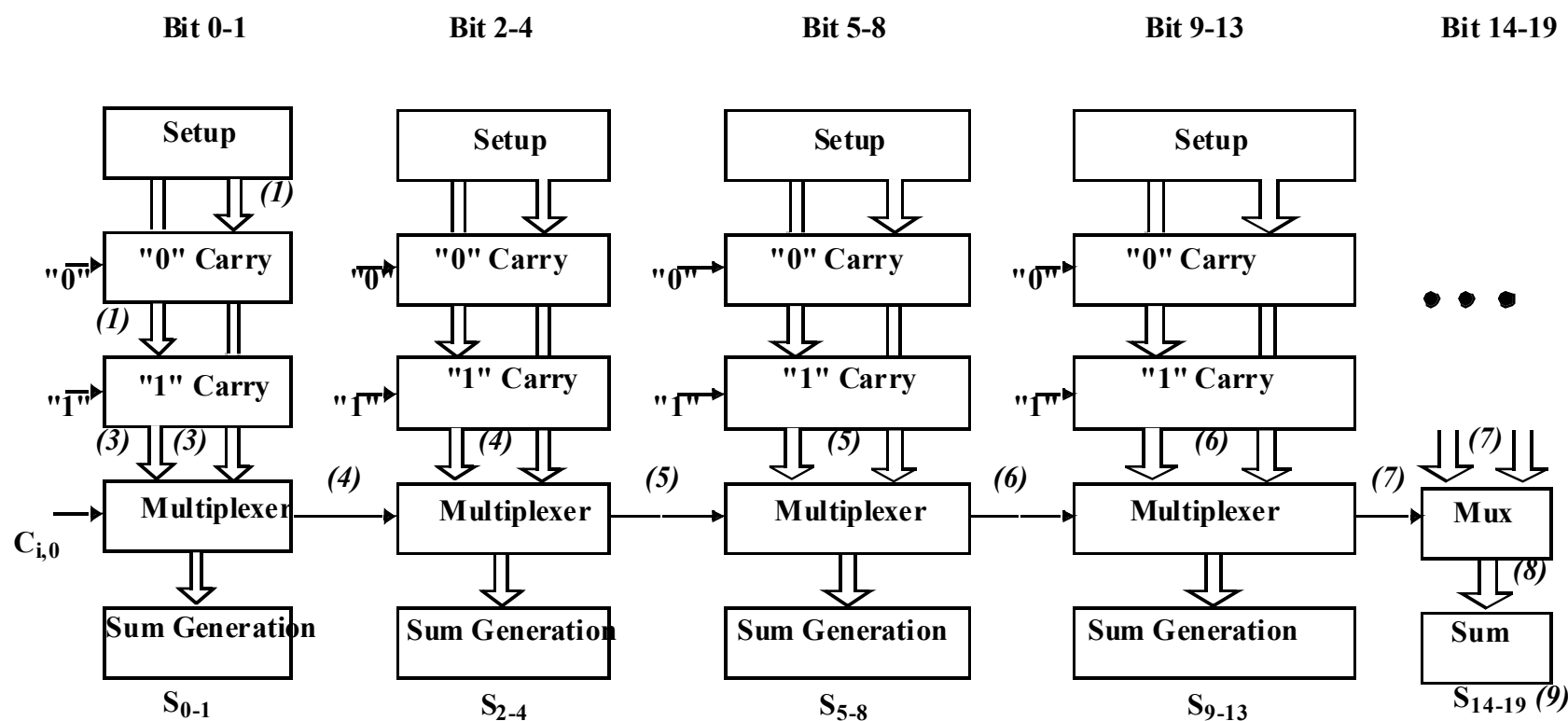
线性进位选择加法器的总进位传播时间仍与位数N成正比，但比逐位进位加法器快。  
硬件开销为一个额外的进位路径和一个MUX，大约比一个逐位进位加法器多30%



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

# 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



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

---

# 进一步优化方法（平方根进位选择加法器）

平方根进位选择加法器的延时：

假设**N**位的加法器含有**P**个级，且第一级加是**M**位，后续级逐级增加一位，于是：

$$\begin{aligned} N &= M + (M + 1) + (M + 2) + \dots + (M + P - 1) \\ &= MP + \frac{P(P-1)}{2} = \frac{P^2}{2} + P(M - \frac{1}{2}) \end{aligned}$$

若 **M** << **N**，（如**M**=**2**，**N**=**64**），则

$$N = \frac{P^2}{2} + P(M - \frac{1}{2}) \approx \frac{P^2}{2}$$

于是

$$P = \sqrt{2N}$$

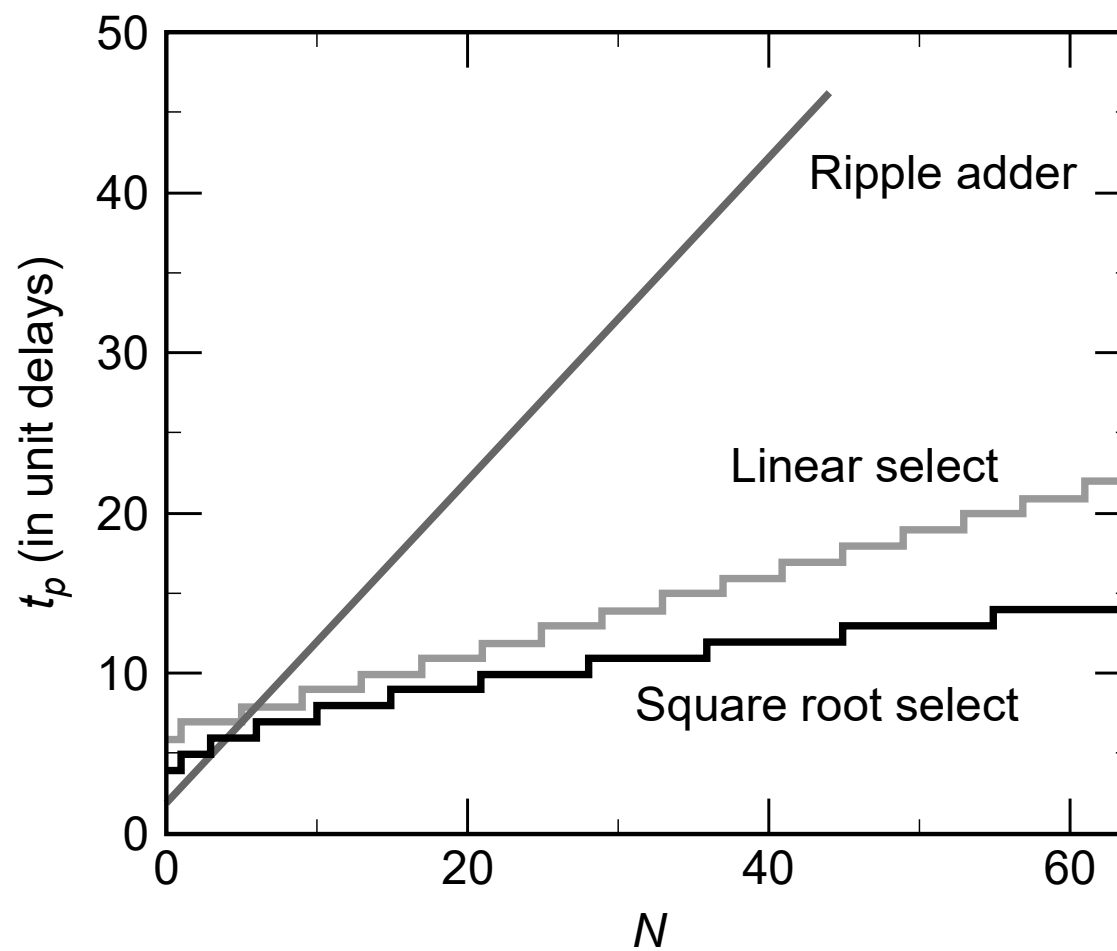
此时延时正比于 $\sqrt{2N}$ （亚线性关系），而不是**N**（线性关系）

当**N**很大时，延时几乎变为常数。

---

# 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:

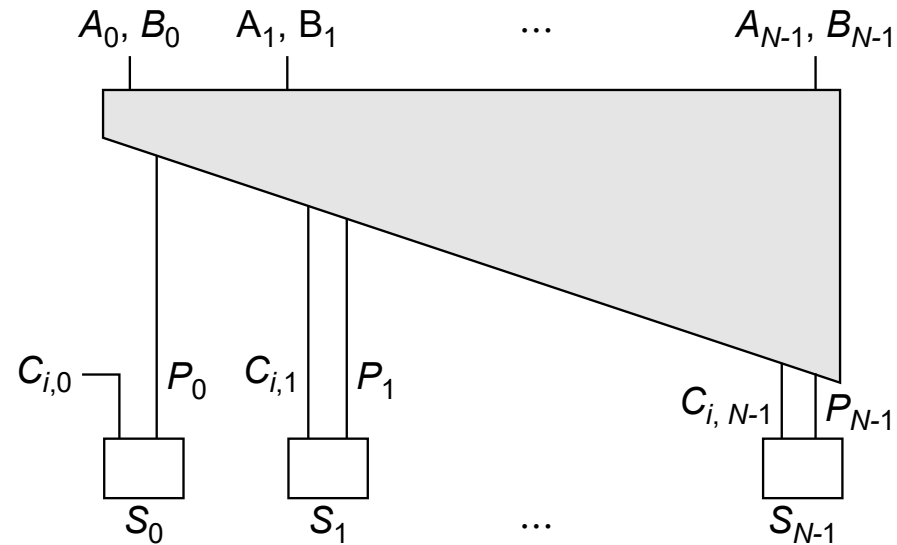
$$C_o(G, P) = G + PC_i$$

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

---

# 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

---