西安交通大学
XI'AN JIAOTONG UNIVERSITY

人工智能学院
College of Artificial Intelligence, XJTU

数字设计和计算机体系结构

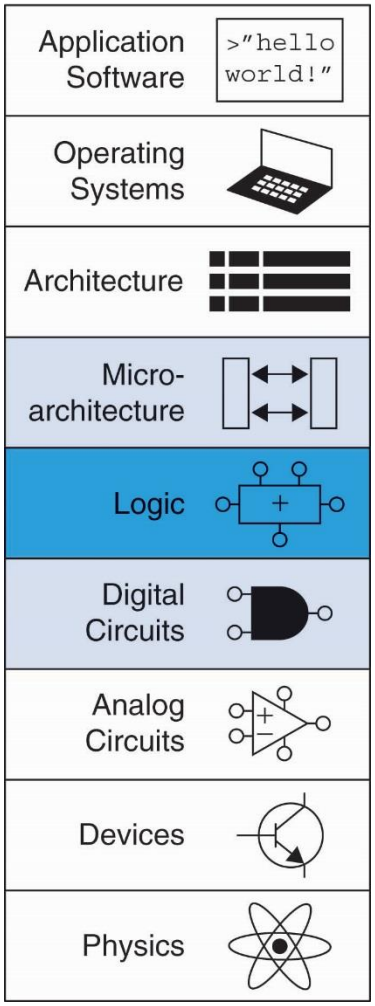
第五章 常见数字模块

刘龙军 副教授

2020年11月12日

第五章 数字模块

- 介绍
- *算术电路
- 时序电路模块
- 存储器阵列
- 可编程逻辑



Modelsim使用说明

1、安装ModelSim仿真软件；

2、打开软件，先建立一个工程project；

3、新建一个systemverilog文件（后缀.sv）；

4、写入module 及testbench编程文件内容；

5、compile编译通过；

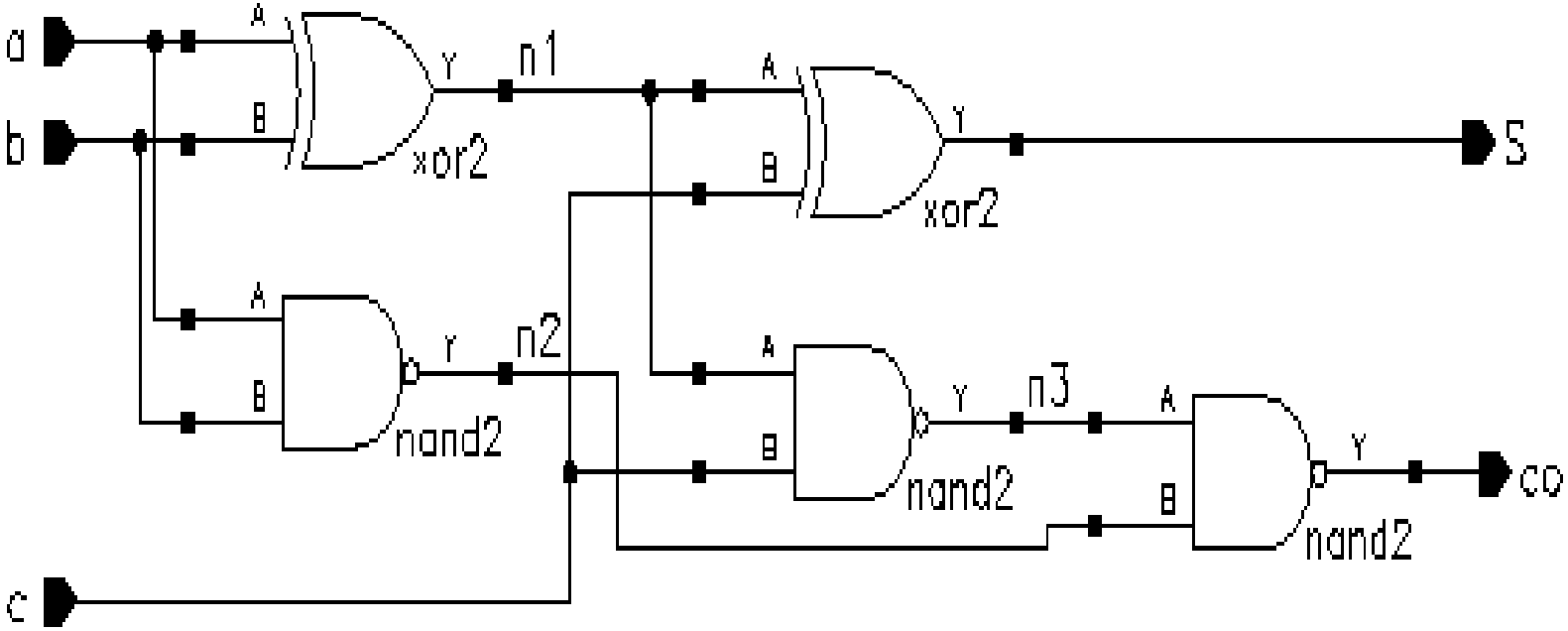
6、启动Simulation选项下的Start Simulation；

7、选择testbench模块，添加信号到Wave中观察；

8、点击Run按钮运行，观察输出波形是否正确；

ELSEVIER

门单元实现1位全加器



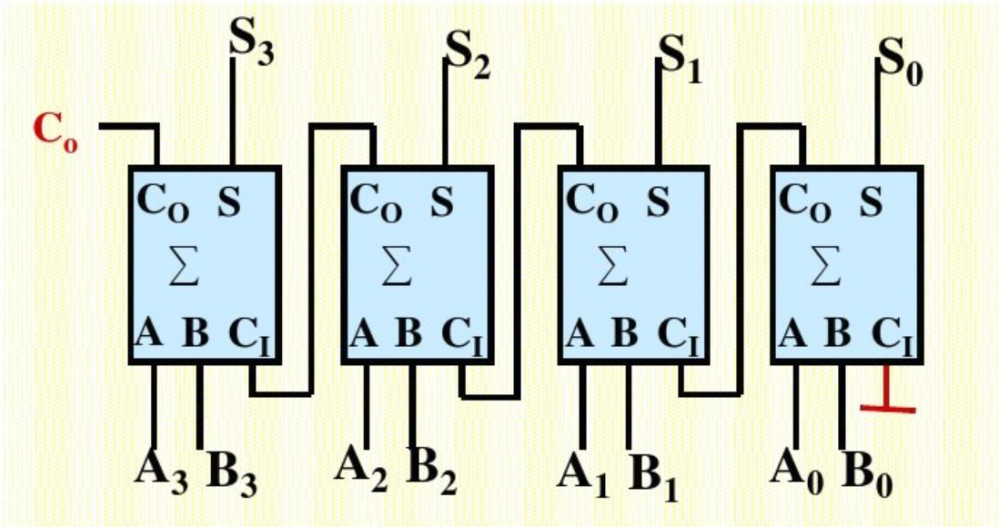
```
module full_adder (co, s, a, b, c);
        input a, b ,c ;
        output co, s ;
            xor  x1 (n1, a, b) ;
            xor  x2 (s, n1, c) ;
            nand  x3(n2, a, b) ;
            nand  x4(n3,n1, c) ;
            nand  x5(co, n3,n2) ;
endmodule
```

```
module    test_full_adder;
wire      co,s;
reg       a,b,c;
initial  begin
            a = 1'b1; b=1'b0; c=1'b1;
   #10   a = 1'b1; b=1'b1; c=1'b0;
   #10   a = 1'b0; b=1'b1; c=1'b0;
   #10   a = 1'b0; b=1'b0; c=1'b1;
   #10   $stop;
end
full_adder     full_adder_instance (co,s,a,
b, c);

endmodule
```

© Digital Design and Computer Architecture, 2nd Edition, 2012

ELSEVIER

4位加法器



4位全加器systemverilog实现

主模块：

```
module full_adder4(a,b,cin,s,cout);
input[3:0]  a,b;
input       cin;
output[3:0]  s;
output      cout;
wire[3:0]     s;
assign {cout,s} = a + b + cin;
endmodule
```

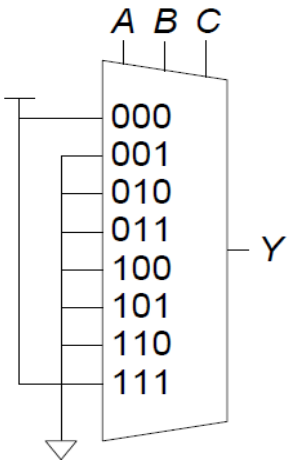
采用1位全加器如何实现？

测试模块：

```
module test_full_adder4;
reg[3:0]      a,b;
reg           cin;
wire[3:0]     s;
wire          cout;
initial  begin
            a = 4'h5; b=4'hb; cin=1'b1;
   #10   a = 4'ha; b=4'hc; cin=1'b0;
   #10   a = 4'h8; b=4'h6; cin=1'b0;
   #10   a = 4'h4; b=4'hf; cin=1'b1;
   #10   $stop;    end
full_adder4   n1(a, b, cin, s, cout);
endmodule
```

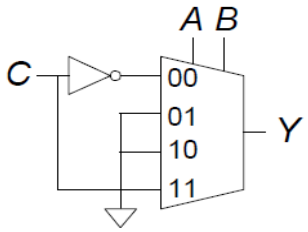
选择器实现逻辑 4

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1



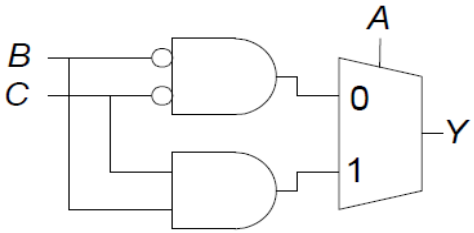
8:1选择器实现

A	B	Y
0	0	$\overline{C}$
0	1	0
1	0	0
1	1	C



4:1选择器实现

A	Y
0	$\overline{B}\overline{C}$
1	BC



2:1选择器实现

选择器systemverilog实现

主模块：

```
module mux4(a,b,c,y);
input  a,b,c;
output y;
wire     y;
assign y= ((~a)&(~b)&(~c) ) | (a&b&c) ;
endmodule
```

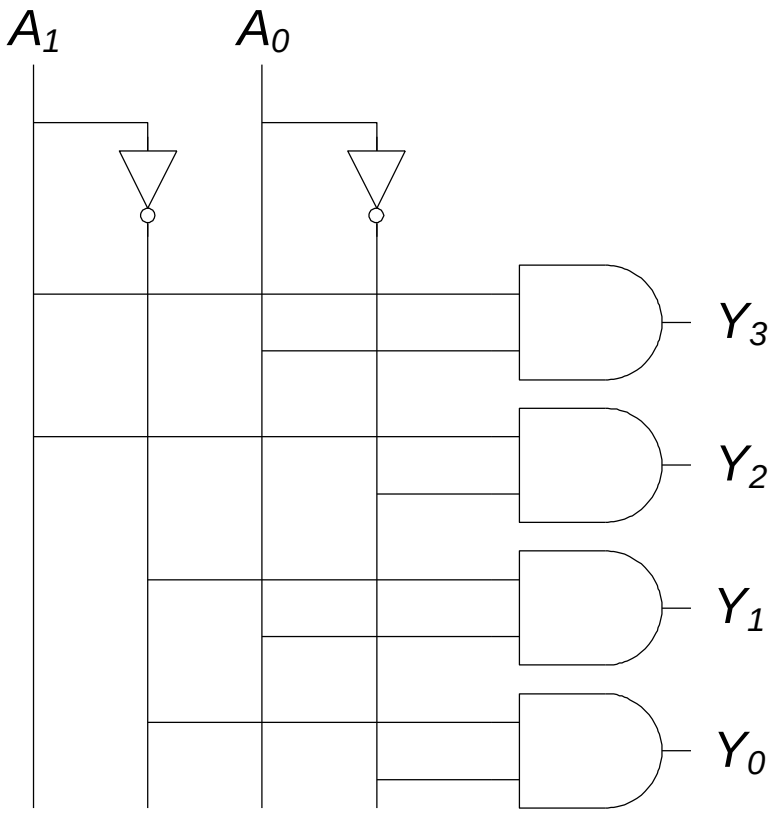
测试模块：

```
module test_mux4;
reg       a,b,c;
initial  begin
            a = 1’b0; b=1’b0; c=1’b0;
    #10   a = 1’b1; b=1’b1; c=1’b0;
    #10   a = 1’b0; b=1’b1; c=1’b0;
    #10   a = 1’b1; b=1’b1; c=1’b1;
    #10   $stop;
end
mux4   n1(a, b, c, y);
endmodule
```

译码器实现

一个N:2^N的译码器可以由2^N个N输入与门通过接收所有输入的值形式或取反形式的各种组合来个构成，译码器的每一个输出代表一个最小项

A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



译码器systemverilog实现

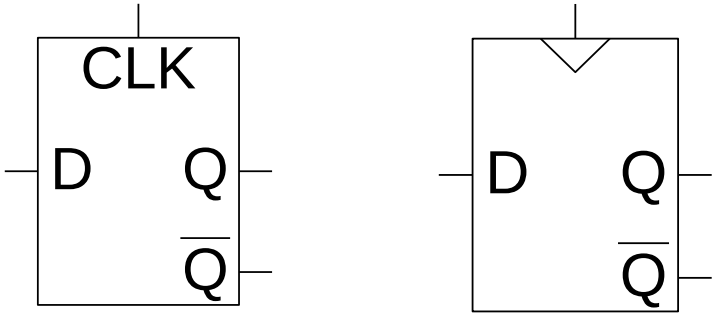
主模块：

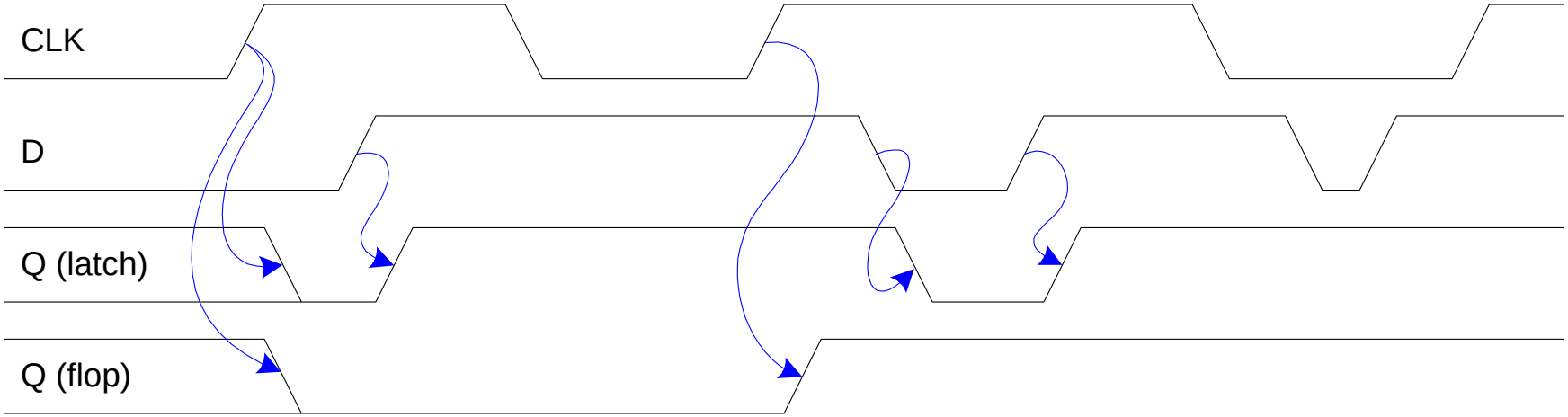
```
module decode4(a1,a0,y3,y2,y1,y0);
input  a1,a0;
output y3,y2,y1,y0;
wire     y3,y2,y1,y0;
assign y0=  ~a1 & ~a0;
assign y1=  ~a1 & a0;
assign y2= a1 & ~a0;
assign y3= a1&a0;
endmodule
```

测试模块：

```
module test_decode4;
reg       a1,a0;
initial  begin
            a1 = 1'b0; a0=1'b0;
    #10   a1 = 1'b0; a0=1'b1;
    #10   a1 = 1'b1; a0=1'b0;
    #10   a1 = 1'b1; a0=1'b1;
    #10   $stop;
end
decode4   n1(a1, a0, y3,y2,y1, y0);
endmodule
```

D 锁存器 vs. D 触发器

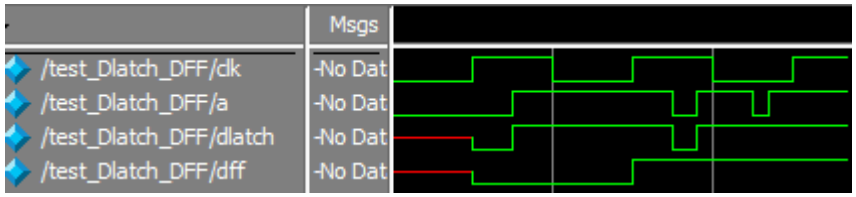




D锁存器D触发器systemverilog实现

主模块：

```
module DLatch_DFF(clk,a,dlatch,dff);
input  clk,a;
output dlatch,dff;
reg      dlatch,dff;
always @(clk or a)
   if(clk)  dlatch <= a;
always @(posedge clk)
   dff <= a;
endmodule
```



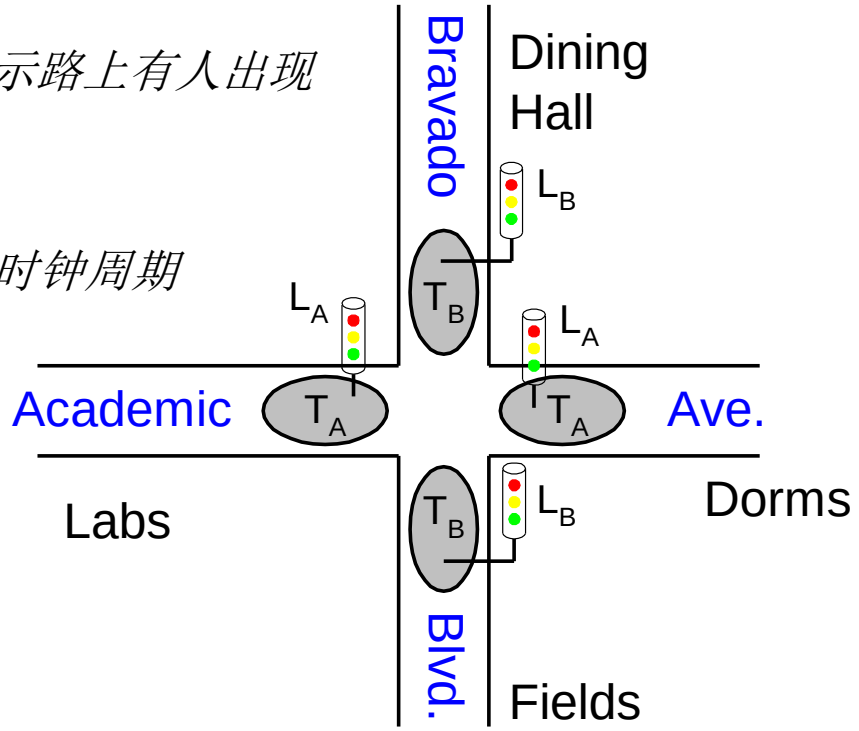
测试模块：

```
module test_Dlatch_DFF;
reg       clk,a;
initial  begin
    a = 1'b0; clk=1'b0; #10 a=1'b0;
    #5   a = 1'b1; #20   a = 1'b0;
    #3   a = 1'b1; #7     a = 1'b0;
    #2   a = 1'b1; #10   $stop;
end
always #10 clk = ~clk;
DLatch_DFF   n1(clk, a, dlatch, dff);
endmodule
```

FSM1 例子

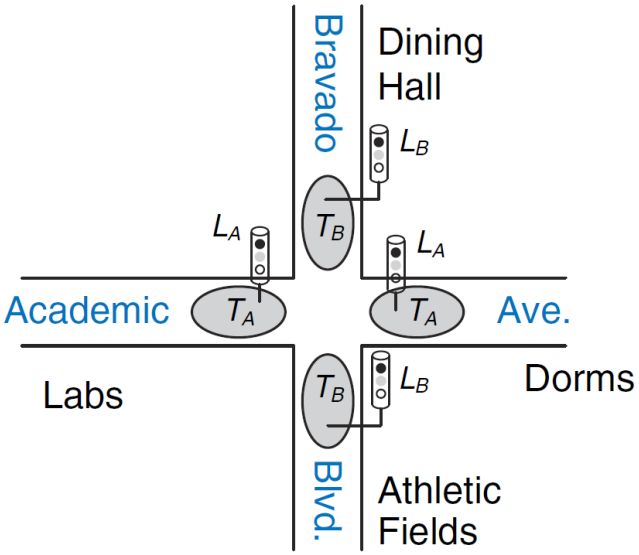
- 交通灯控制器
  - 交通传感器: T_A, T_B
  - 灯: L_A, L_B
  - *传感器输入为TRUE时，表示路上有人出现 否则路上无人*
  - *交通灯显示 红 绿 黄*

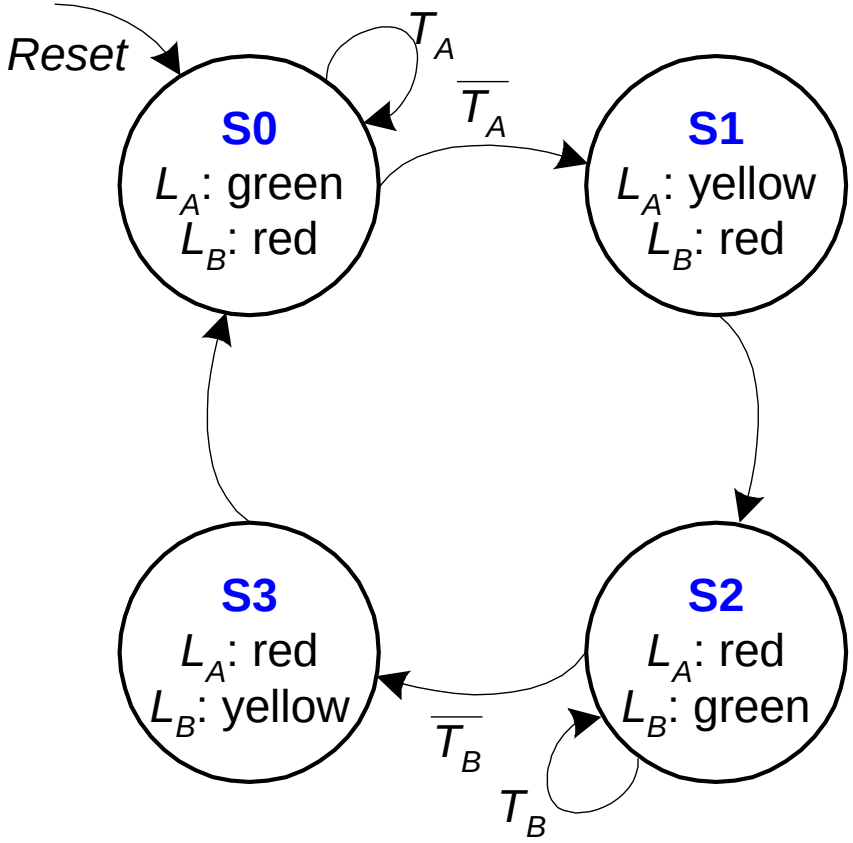
采用一个周期为5秒，每一个时钟周期上升沿，灯将根据交通传感器来改变。



FSM1 状态转换图

- Moore FSM: 输出在每个状态上
- 状态: 圆圈里
- 转换: 箭头





FSM1例子的systemverilog实现

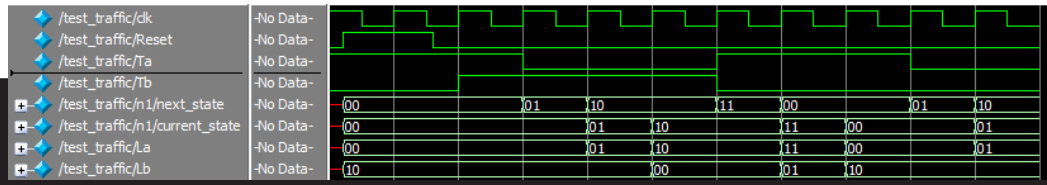
主模块

```
module traffic(clk,Reset,Ta,Tb,La,Lb);
input  clk,Reset,Ta,Tb;
output[1:0]  La,Lb;
parameter S0=2'b00,S1=2'b01,S2=2'b10,S3=2'b11;
reg[1:0]      current_state,next_state,La,Lb;
always_ff @(posedge clk or posedge Reset)
   if(Reset)  current_state <= S0;
   else  current_state <= next_state;
always_comb
   case(current_state)
   S0: if(Ta)  next_state = S0;
       else  next_state = S1;
   S1:  next_state = S2;
   S2:  if(Tb) next_state =  S2;
        else  next_state =  S3;
   S3:  next_state = S0;
   endcase
```

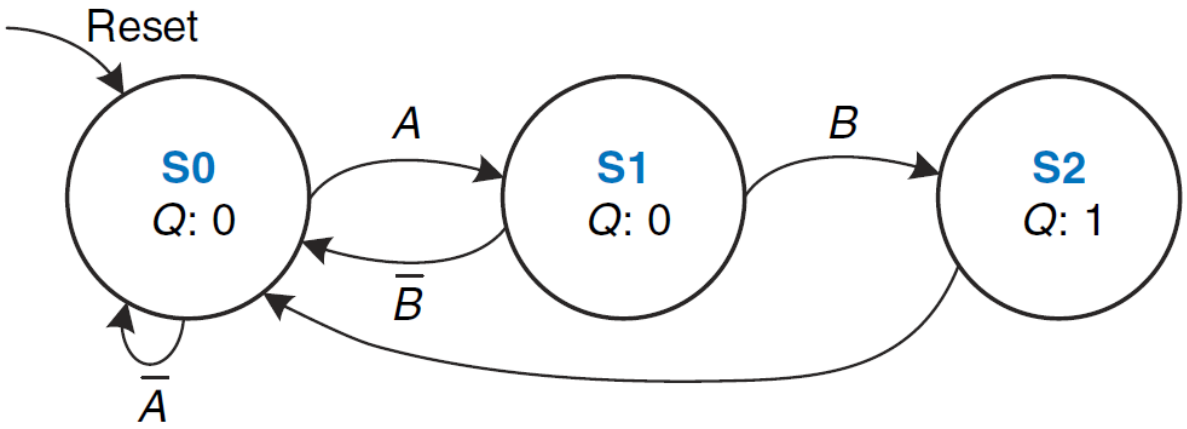
```
always_comb
   case(current_state)
   S0: begin La = 2'b00; Lb = 2'b10; end
   S1: begin La = 2'b01; Lb = 2'b10; end
   S2: begin La = 2'b10; Lb = 2'b00; end
   S3: begin La = 2'b11; Lb = 2'b01; end
   endcase
endmodule
```

测试模块

```
`timescale 1ns/10ps
module test_traffic;
reg       clk,Reset,Ta,Tb;
wire[1:0] La,Lb;
initial  begin
    clk = 1'b1; Reset=1'b0;
    #1 Reset = 1'b1; #7 Reset = 1'b0; end
initial  begin
          Ta = 1'b1;Tb=1'b0;
    #10 Ta = 1'b1;Tb=1'b1;
    #5   Ta = 1'b0;Tb=1'b1;
    #15 Ta = 1'b1;Tb=1'b0;
    #15 Ta = 1'b0;Tb=1'b0;
    #10   $stop;   end
always #2.5 clk = ~clk;
traffic  n1(clk,Reset,Ta,Tb,La,Lb);
endmodule
```



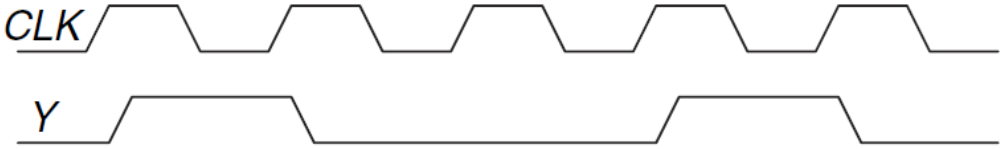
FSM2例子

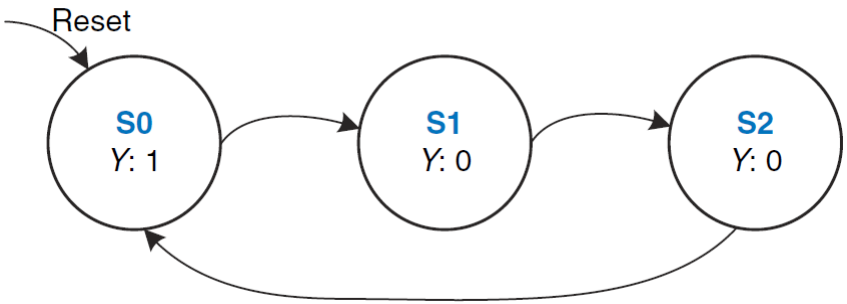


编程实现及Testbench

FSM3例子

三分频器用FSM编程实现。





状态机功能图

编程实现及Testbench

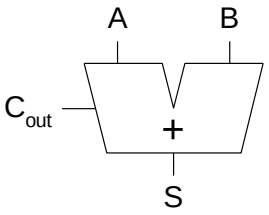
常见数字模块:

- 复用器, 解码器, 编码器,算术运算电路, 计数器, 移位器, 存储器阵列, 逻辑阵列

- **模块构造原则：层次化 hierarchy, 模块化modularity, 规则化regularity:**
  - Hierarchy of simpler components
  - Well-defined interfaces and functions
  - Regular structure easily extends to different sizes
- **用上述模块搭建后续章节的微处理器和存储器**

1-Bit 加法器 Adders

Half Adder

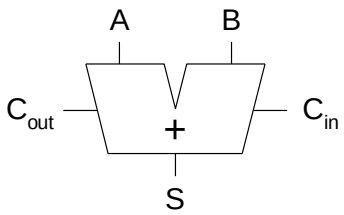


A	B	C_{out}	S
0	0		
0	1		
1	0		
1	1		

S =

C_{out} =

Full Adder



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

S =

C_{out} =

写出SystemVerilog及其Testbench

多比特加法器

- 进位传播加法器 (CPAs)：A，B，S 总线
  - 行波进位加法器Ripple-carry (slow)
  - 先行进位加法器Carry-lookahead (fast)
  - 前缀加法器Prefix (faster)
- 先行进位和前缀加法器更快但需要更多硬件资源（空间换时间）

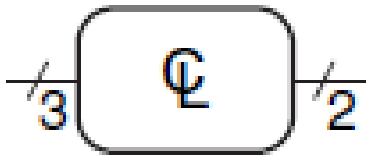
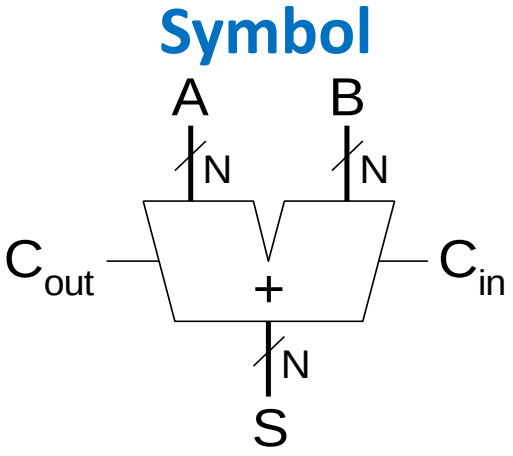
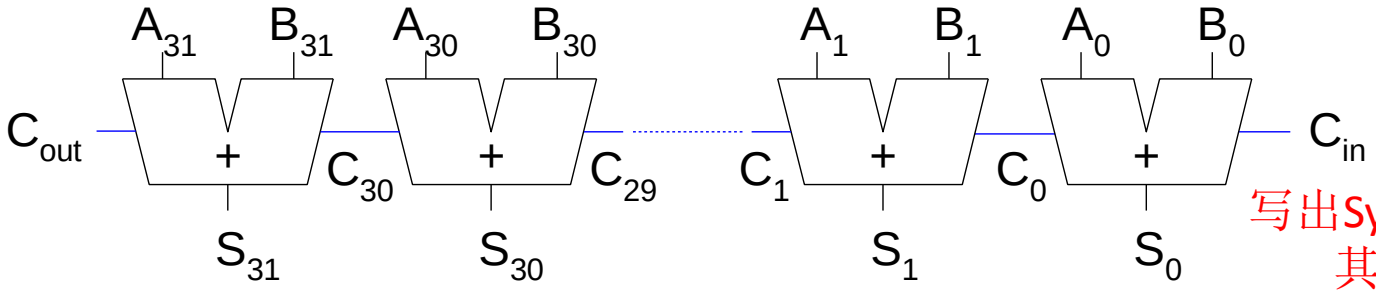


图2-6



行波进位加法器

- 将1bit的加法器通过进位链串联起来
- advantage： Simple
- Disadvantage: **slow**
- **模块化与规整化电路设计的范例**



写出SystemVerilog及其Testbench
调用一位全加器模块

行波进位加法器延迟

$$t_{\text{ripple}} = Nt_{FA}$$

where t_{FA} is the delay of a 1-bit full adder

当组成大规模加法器时运算缓慢的原因

进行一次32位的加法计算就需要至少串行的经过32个全加器，如果CPU的频率是3Ghz，那么一个时钟周期，大约333皮秒内 （0.333纳秒），是无法完成一次简单的加法运算的。

怎样解决进位延迟问题

A	B	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$c_i=A_iB_i+A_ic_{i-1}+B_ic_{i-1}$ $c_i=A_iB_i+(A_i+B_i)c_{i-1}$

$$c_i = A_iB_i + (A_i + B_i)c_{i-1}$$

$$c_i = A_iB_i + (A_i + B_i)\,(A_{i-1}B_{i-1} + c_{i-2}(A_{i-1} + B_{i-1}))$$

$$c_0 = A_0 * B_0$$

先行进位加法器（超前进位加法器CLA）

利用生成与传播信号来计算K比特模块（或一列）计算的输出 C_{out}

- 当生成信号或传播信号有进位时，第i列即进行进位

- Generate (G_i) and propagate (P_i) signals for each column:
  - **生成信号:** Column *i* will generate a carry out if A_i **and** B_i are both 1.

$$G_i =$$

  - **传播信号:** Column *i* will propagate a carry in to the carry out if A_i **or** B_i is 1.

$$P_i =$$

  - **进位输出:** The carry out of column *i* (C_i) is:

$$C_i =$$

先行进位加法器（超前进位加法器CLA）

$c_i = A_iB_i+(A_i+B_i)c_{i-1}$　$c_0=A_0*B_0$

$c_i = A_iB_i+(A_i+B_i)\ (A_{i-1}B_{i-1} + c_{i-2}(A_{i-1} + B_{i-1}))$

$$C_i = A_iB_i + (A_i + B_i)C_{i-1} = G_i + P_iC_{i-1}$$

$$= G_i + P_i(G_{i-1} + P_{i-1}C_{i-2})$$

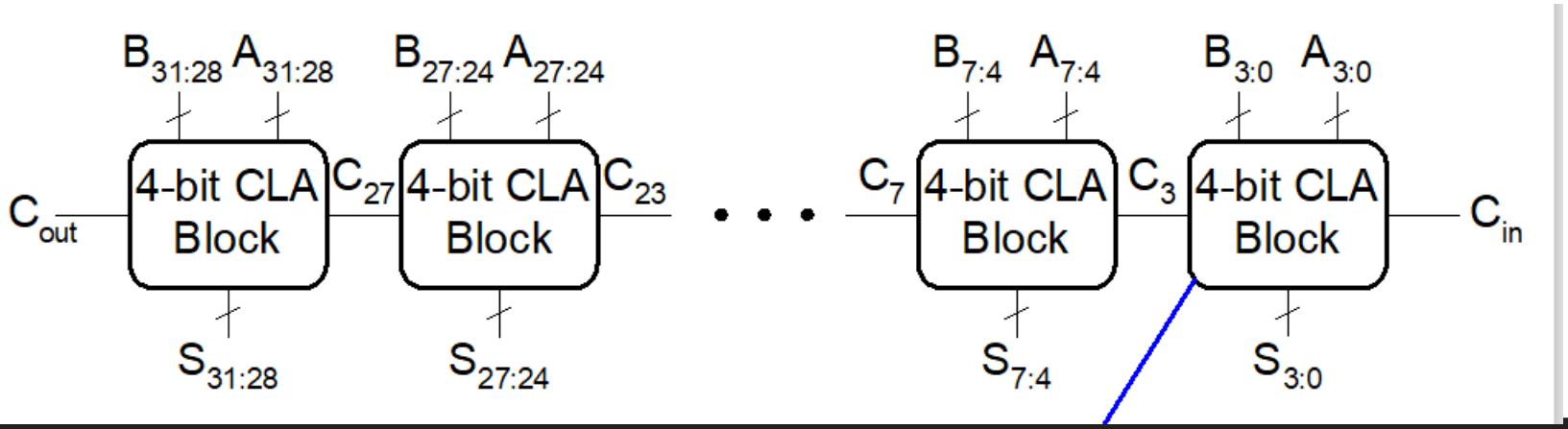
理论上可以实现任何位数的进位都可以化简到最低进位和已知A B信号。但由于成本问题一般都会把32位或者64位的切割成多个16位/8位/4位的块进行超前进位计算，能保证在一个时钟周期内完成基本就可以了，从而取得一个成本和性能上得平衡。

块生成与传播信号（模块化思想）

概念扩展：

如果一个块在不考虑进位输入的情况下也能产生进位输出，则称其**生成进位 generate** $G_{i:j}$

如果一个块在有进位输入时能产生进位，称其为**传播进位 propagate** $P_{i:j}$



块生成与传播信号

- **Example:** 对于4比特块的传播与产生信号 ($P_{3:0}$ and $G_{3:0}$):

产生传播进位的条件：块中所有列（某位不能吸收）都能传播进位：

$$P_{3:0} = P_3 P_2 P_1 P_0$$

产生生成进位的条件：最高有效列产生一个进位，或者最高有效列传播进位且前面的列产生了进位，以此类推：

$$G_{3:0} = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3$$

$$= G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0))$$

块生成与传播信号

- 使用块的生成信号和传播信号，可以根据块的进位输入C_{j-1}，快速计算块的进位输出C_i，

$$P_{i:j} = P_i P_{i-1} P_{i-2} \ldots P_j$$

$$G_{i:j} = G_i + P_i (G_{i-1} + P_{i-1} (G_{i-2} + P_{i-2} \ldots G_j))$$

$$C_i = G_{i:j} + P_{i:j} C_{j-1}$$ （*进位输出的条件：块生成进位或块传播进位*）

32-bit CLA with 4-bit Blocks

块中进位：

$C_i = A_i B_i + (A_i + B_i)C_{i-1} = G_i + P_i C_{i-1}$

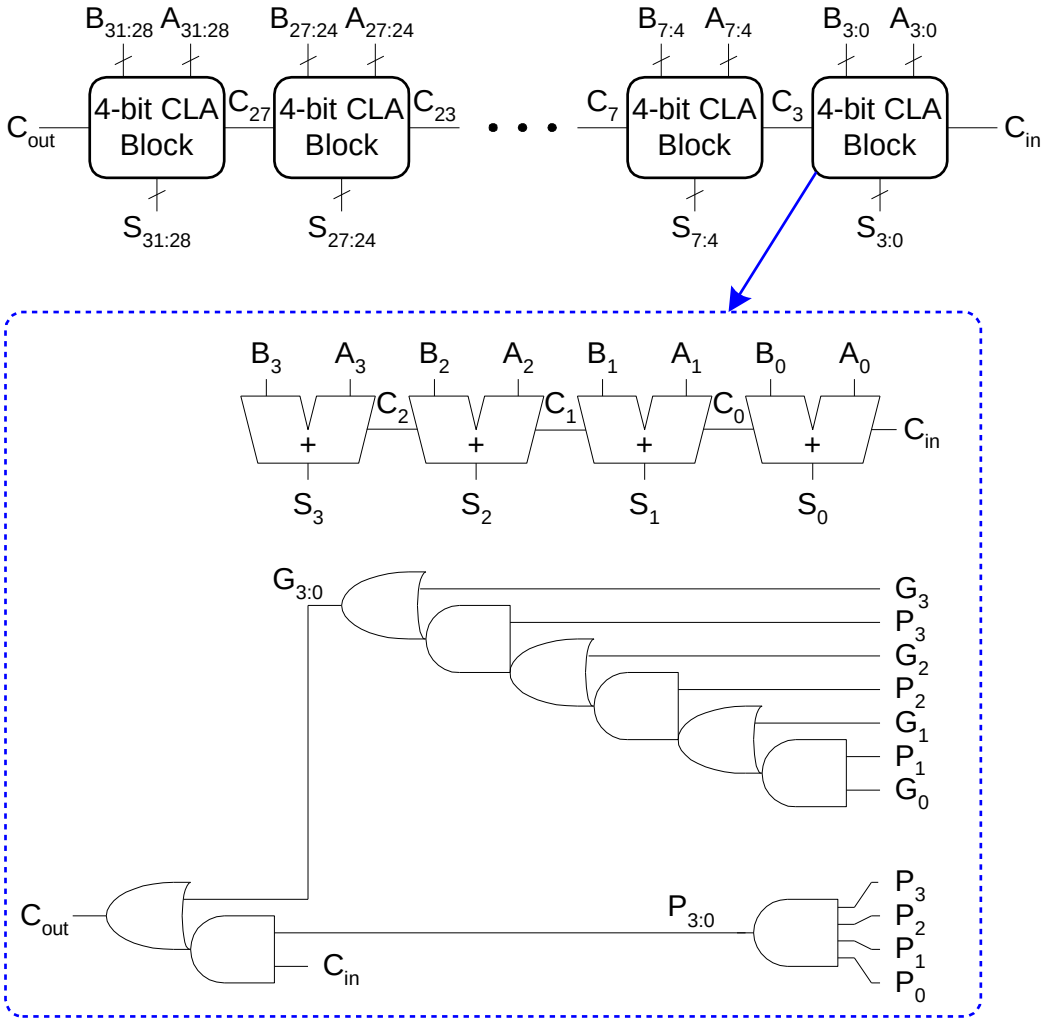
块生成信号：

$G_{i:j} = G_i + P_i (G_{i-1} + P_{i-1} (G_{i-2} + P_{i-2} \ldots G_j))$

块传播信号： $P_{i:j} = P_i P_{i-1} P_{i-2} \ldots P_j$

根据 $C_i = G_{i:j} + P_{i:j} C_{j-1}$，*则*
$C_3 = G_{3:0} + P_{3:0} C_{in}$，*其中*
$G_{3:0} = G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0))$,
$P_{3:0} = P_3 P_2 P_1 P_0$

写出SystemVerilog及其Testbench，其中4-bit CLA作为子模块，其调用4个一位全加器模块



先行进位加法器CLA（超前进位加法器）

- **Step 1:** Compute G_i and P_i for all columns
- **Step 2:** Compute G and P for k-bit blocks
- **Step 3:** C_{in} propagates through each k-bit propagate/generate logic (**meanwhile computing sums 串行的行波加法器**)
- **Step 4:** Compute sum for most significant k-bit block

先行进位加法器CLA

- **Step 1:** Compute G_i and P_i for all columns

$$G_i = A_i B_i$$

$$P_i = A_i + B_i$$

先行进位加法器CLA

- **Step 1:** Compute G_i and P_i for all columns
- **Step 2:** Compute G and P for k-bit blocks，比如4-bit blocks

$$P_{3:0} = P_3 P_2 P_1 P_0$$

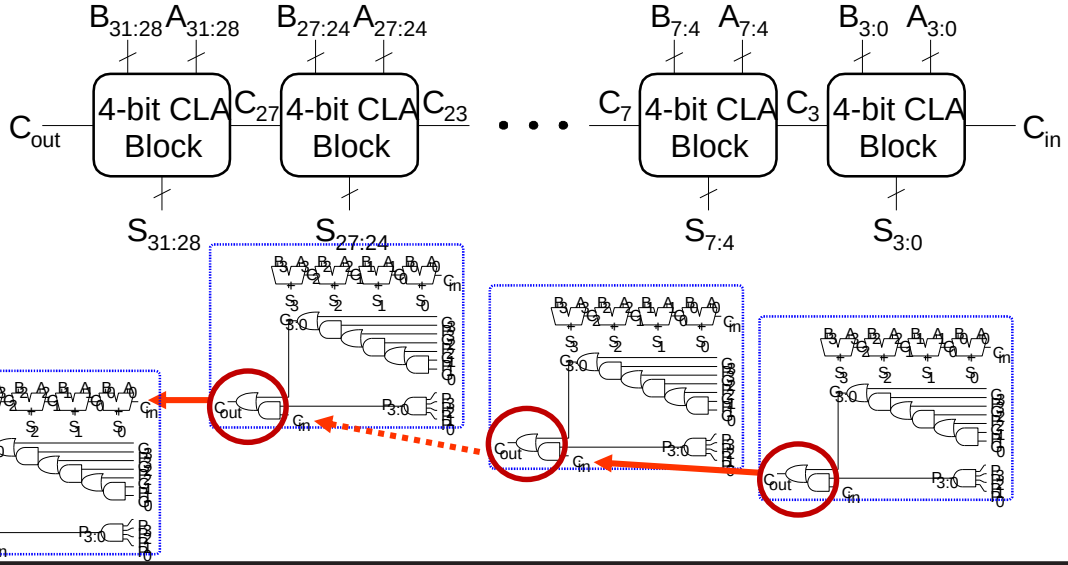
$$G_{3:0} = G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0))$$

先行进位加法器CLA

- **Step 1:** Compute G_i and P_i for all columns
- **Step 2:** Compute G and P for k-bit blocks
- **Step 3:** C_{in} propagates through each k-bit propagate/generate logic (meanwhile computing sums)

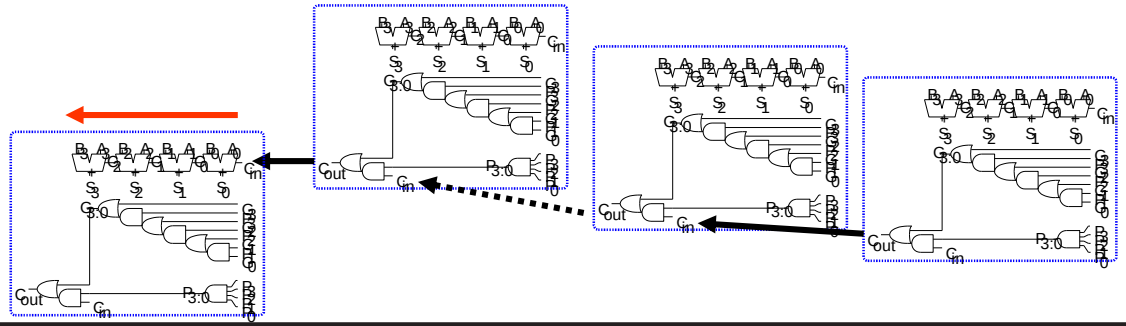
求解块的进位：

$$C_i = G_{i:j} + P_{i:j} C_{j-1}$$



先行进位加法器CLA

- **Step 1:** Compute G_i and P_i for all columns
- **Step 2:** Compute G and P for k-bit blocks
- **Step 3:** C_{in} propagates through each k-bit propagate/generate logic (meanwhile computing sums)
- **Step 4:** Compute sum for most significant k-bit block



先行进位加法器的延迟

For *N*-bit CLA with *k*-bit blocks:

$$t_{CLA}\,(S_{31}) = t_{pg} + t_{pg\_block} + (N/k - 1)t_{AND\_OR} + kt_{FA}$$

- t_{pg} : delay to generate all P_i, G_i
- t_{pg_block} : delay to generate all $P_{i:j}$, $G_{i:j}$
- t_{AND_OR} : delay from C_{in} to C_{out} of final AND/OR gate in *k*-bit CLA block

An *N*-bit carry-lookahead adder is generally much faster than a ripple-carry adder for $N > 16$

举例：对比32位行波进位加法器和4位块组成的32位先行进位加法器的延迟。假设每个2输入门电路的延迟为100ps，全加器的延迟是300ps。

1）32位行波进位加法器延迟32x300ps=9.6ns

2）32位先行进位加法器的延迟：t_{pg}=100ps，t_{pg_block}=600ps，t_{AND_OR}=200ps

则t_{CLA} = 100+600+（32/4 -1）x200+4x300=3.3ns 快近3倍

$$t_{CLA}\,(C_{31}) = t_{pg} + t_{pg\_block} + (N/k\,)t_{AND\_OR}$$ (非关键路径)

前缀加法器 Prefix Adder(PA)

- 扩展了先行进位加法器的产生和传播逻辑，可以进行更快的加法操作
- 尽可能快地计算每一列的进位输入 (C_{i-1})，然后计算该列的和:

$$S_i = (A_i \oplus B_i) \oplus C_{i-1}$$

- 计算*G* and *P* for 1-, 2-, 4-, 8-bit 块, etc. until all G_i (carry in) known
- 以$\log_2 N$ 为步骤进行计算 （*与先行进位分块的区别*）

前缀加法器PA

$$S_i = (A_i \oplus B_i) \oplus C_{i-1}$$

- 定义：Column -1 holds C_{in}, so

$$C_{-1} = G_{-1} = C_{in},\quad C_0 = G_0 + P_0 C_{-1} = G_0 + P_0 G_{-1} = G_{0:-1}$$

- 从块的看 Carry in to column *i* = carry out of column *i-1*:

$$C_{i-1} = G_{i-1:-1}$$

$G_{i-1:-1}$: generate signal spanning columns *i*-1 to -1

- Sum equation:

公式替换

$$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}$$

- 目标: 快速计算 $G_{-1:-1}$, $G_{0:-1}$, $G_{1:-1}$, $G_{2:-1}$, $G_{3:-1}$, $G_{4:-1}$, $G_{5:-1}$, … (called ***prefixes*** *前缀*) ($P_{-1:-1}$, $P_{0:-1}$, $P_{1:-1}$, $P_{2:-1}$, $P_{3:-1}$, $P_{4:-1}$, $P_{5:-1}$, …)

前缀加法器PA

- 对于块信号：Generate and propagate signals for a **block** spanning bits $i{:}j$

$$G_{i:j} = G_{i:k} + P_{i:k}\, G_{k-1:j}$$

（块操作：参考 先行进位加法器）

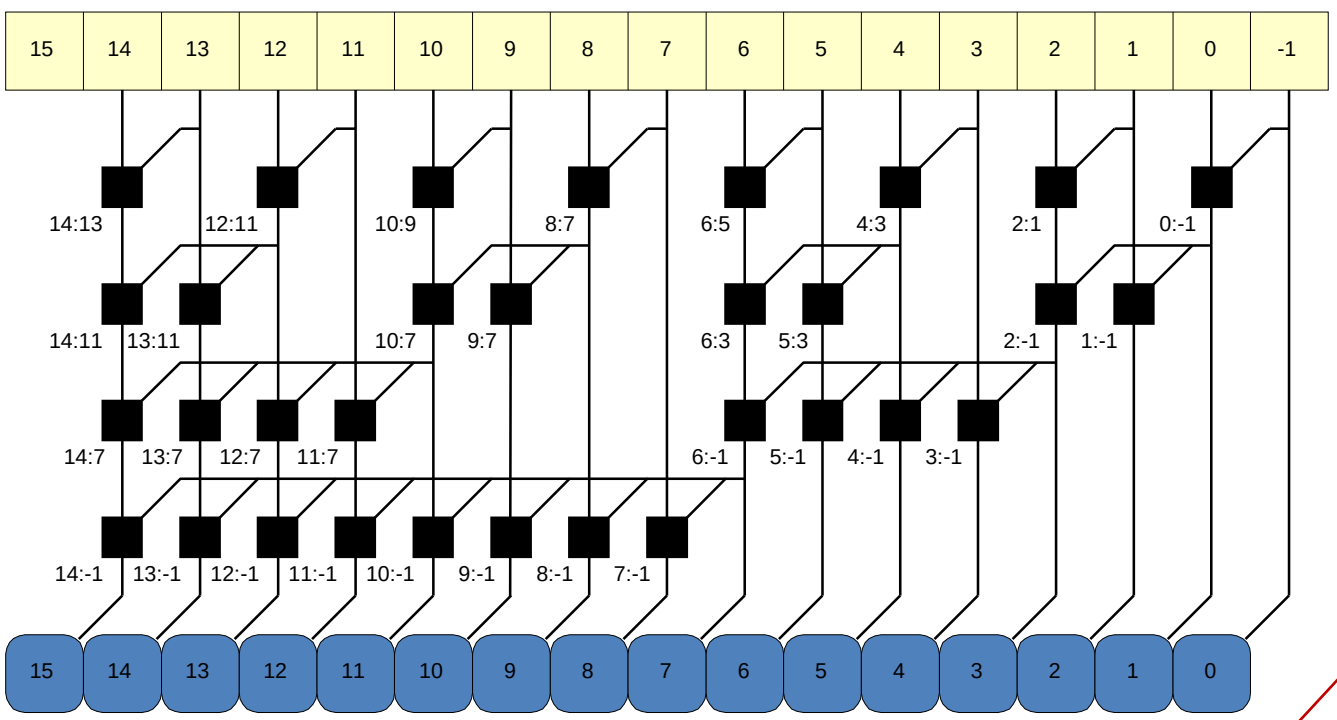
$$P_{i:j} = P_{i:k} P_{k-1:j}$$

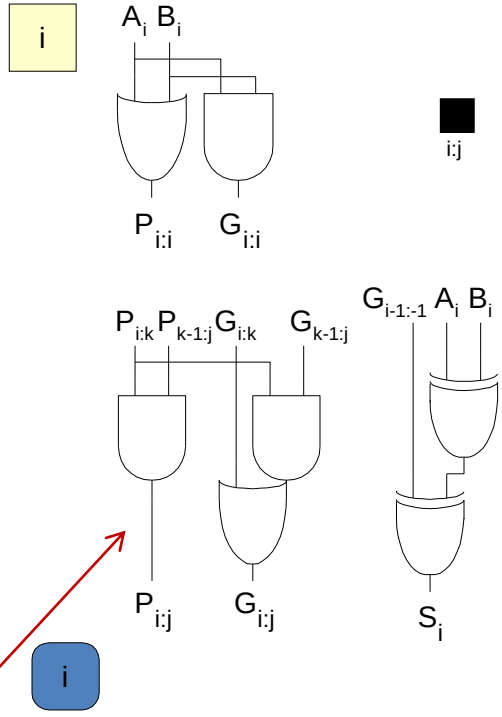
即：$G_{i-1:-1} = G_{i-1:k} + P_{i-1:k}\, G_{k-1:-1}$

$P_{i-1:-1} = P_{i-1:k} P_{k-1:-1}$

- 总之:
  - **Generate: block** $i{:}j$ will generate a carry if:
    - upper part ($i{:}k$) generates a carry or
    - upper part ($i{:}k$) propagates a carry generated in lower part (k-1:j)
  - **Propagate:** block $i{:}j$ will propagate a carry if *both* the upper and lower parts propagate the carry

16-Bit 前缀加法器





即：$G_{i-1:-1} = G_{i-1:k} + P_{i-1:k}\, G_{k-1:-1}$

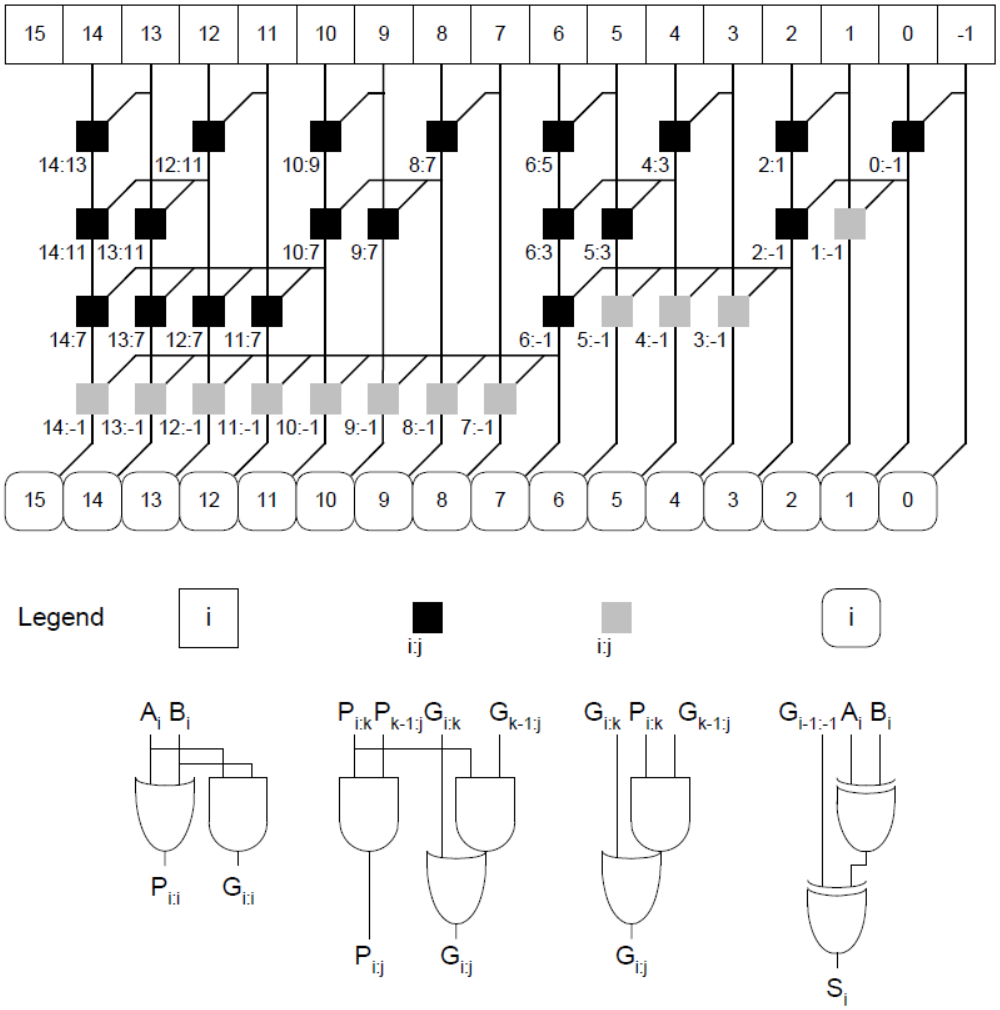
$P_{i-1:-1} = P_{i-1:k} P_{k-1:-1}$

$G_{i:j} = G_{i:k} + P_{i:k}\, G_{k-1:j}$

$P_{i:j} = P_{i:k} P_{k-1:j}$

$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}$

16-Bit 前缀加法器



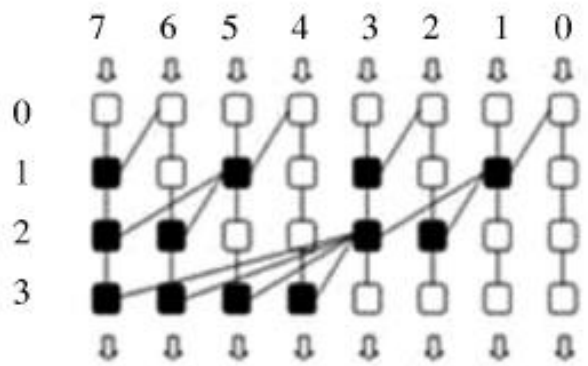
比如：

$S_6 = A_6 \oplus B_6 \oplus G_{5:-1}$

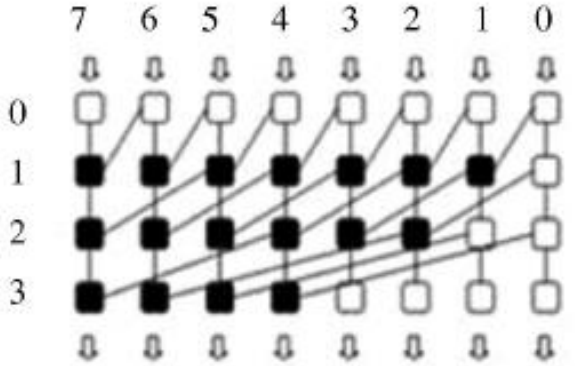
$S_{15} = A_{15} \oplus B_{15} \oplus G_{14:-1}$

写出SystemVerilog及其Testbench

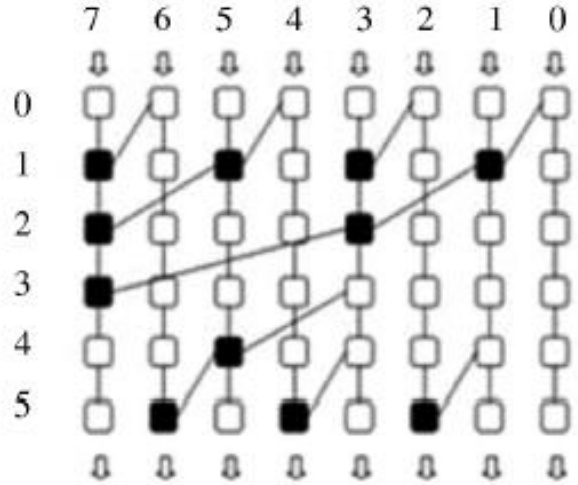
不同结构的前缀加法器



(a) Sklansky结构

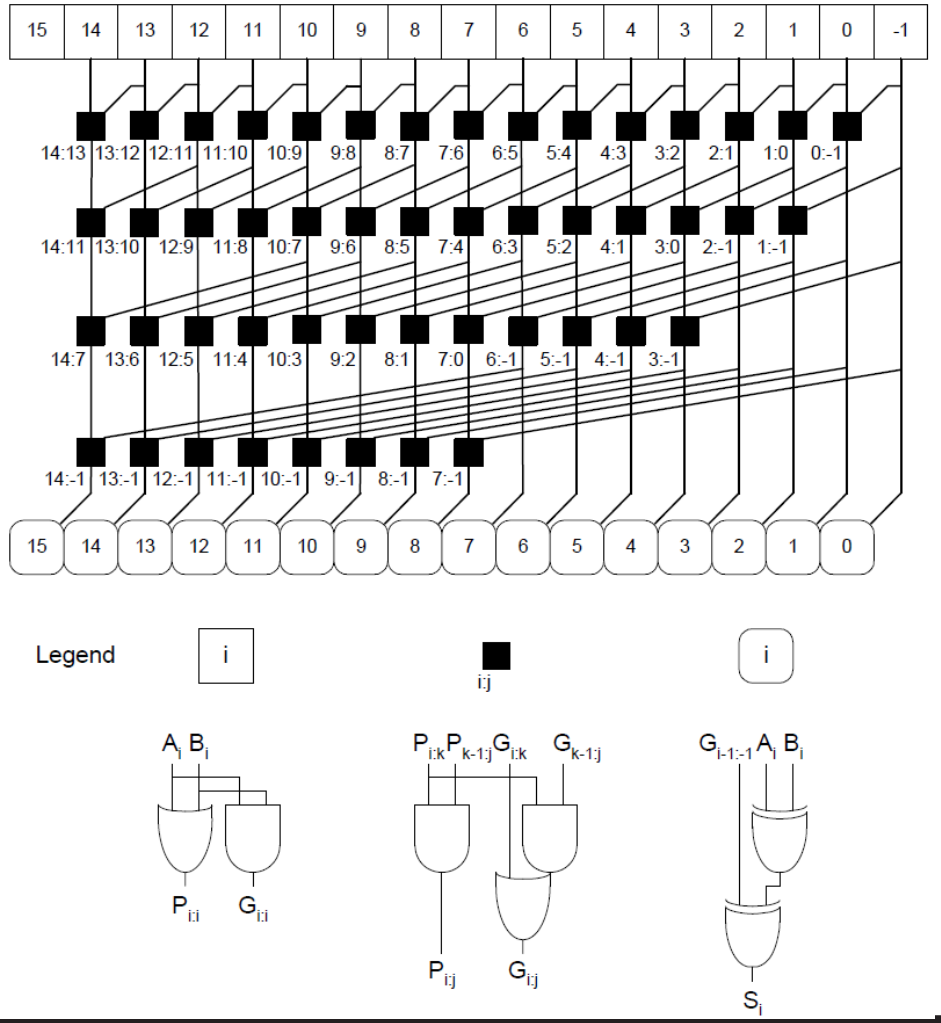


(b) Kogge-Stone结构



(c) Brent-Kung结构

不同结构的前缀加法器


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 -1
14:13 13:12 12:11 11:10 10:9 9:8 8:7 7:6 6:5 5:4 4:3 3:2 2:1 1:0 0:-1
14:11 13:10 12:9 11:8 10:7 9:6 8:5 7:4 6:3 5:2 4:1 3:0 2:-1 1:-1
14:7 13:6 12:5 11:4 10:3 9:2 8:1 7:0 6:-1 5:-1 4:-1 3:-1
14:-1 13:-1 12:-1 11:-1 10:-1 9:-1 8:-1 7:-1
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Legend
i
i:j
i
A_i B_i
P_i:i G_i:i
P_i:k P_k-1:j G_i:k G_k-1:j
P_i:j G_i:j
G_i-1:-1 A_i B_i
S_i

前缀加法器的延迟

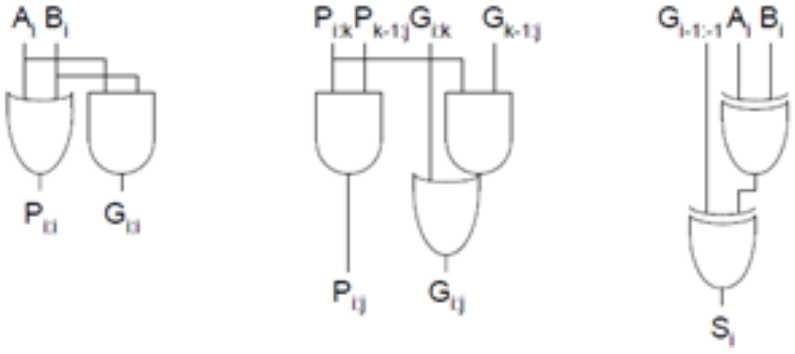
$$t_{PA} = t_{pg} + \log_2 N(t_{pg\_prefix}) + t_{XOR}$$

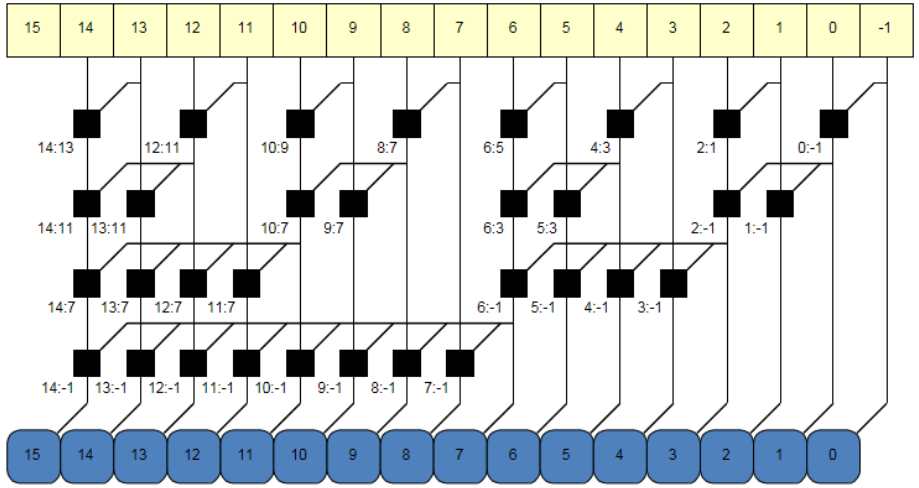
(因$G_{i-1:-1}$是关键路径)

t_{pg}: delay to produce P_i, G_i (AND or OR gate)

t_{pg_prefix}: delay of black prefix cell (AND-OR gate)

t_{XOR}:计算最后的总和Si





加法器延迟对比

延迟对比: 32-bit ripple-carry, CLA, and prefix adders

- CLA has 4-bit blocks
- 2-input gate delay = 10 ps; full adder delay = 30 ps

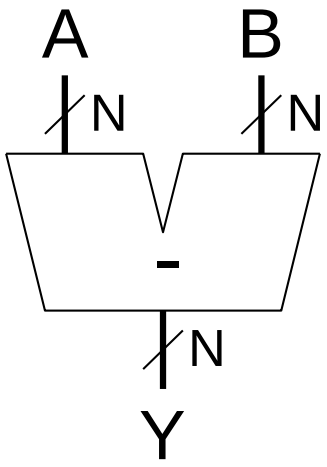
$$t_{\text{ripple}} = Nt_{FA} = 32\text{x}30 \text{ ps}$$
$$= \mathbf{960 \text{ ps}}$$

$$t_{CLA} = t_{pg} + t_{pg\_\text{block}} + (N/k - 1)t_{\text{AND\_OR}} + kt_{FA}$$
$$= [10 + 60 + (7)20 + 4(30)] \text{ ps}$$
$$= \mathbf{330 \text{ ps}}$$

$$t_{PA} = t_{pg} + \log_2 N(t_{pg\_\text{prefix}}) + t_{\text{XOR}}$$
$$= [10 + \log_2 32(20) + 10] \text{ ps}$$
$$= \mathbf{120 \text{ ps}}$$

减法器

Symbol



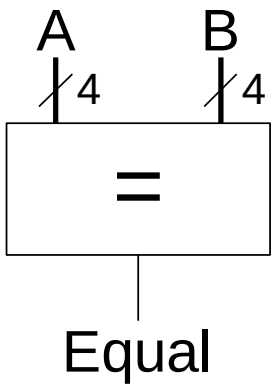
Implementation 补码运算

A B

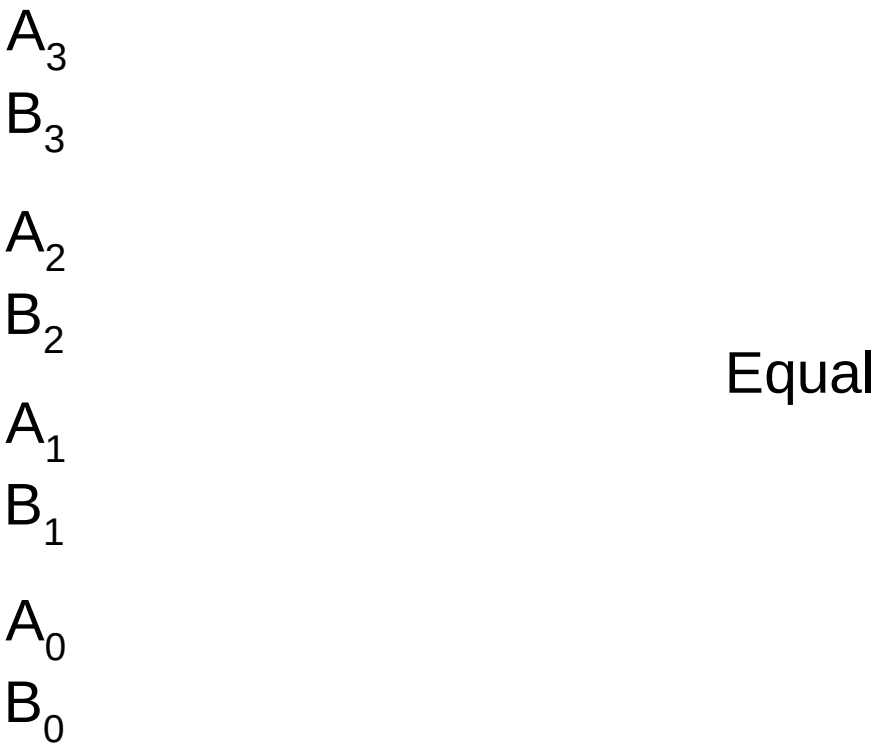
Y

比较器: 相等比较器

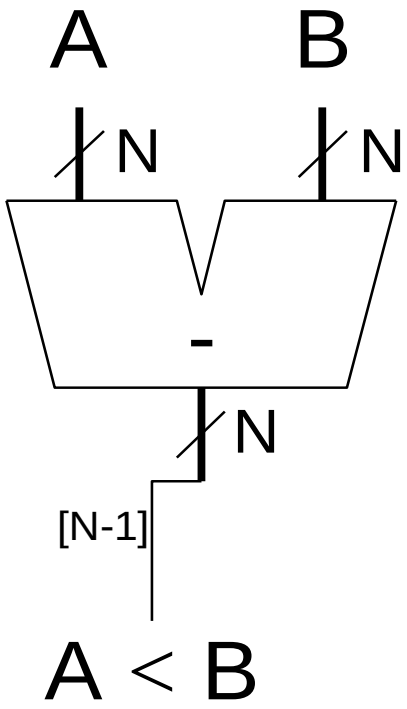
Symbol



Implementation 同或门



比较器: 小于



减法器（看结果的最高值，如1，则小于，如0，则大于或等于）

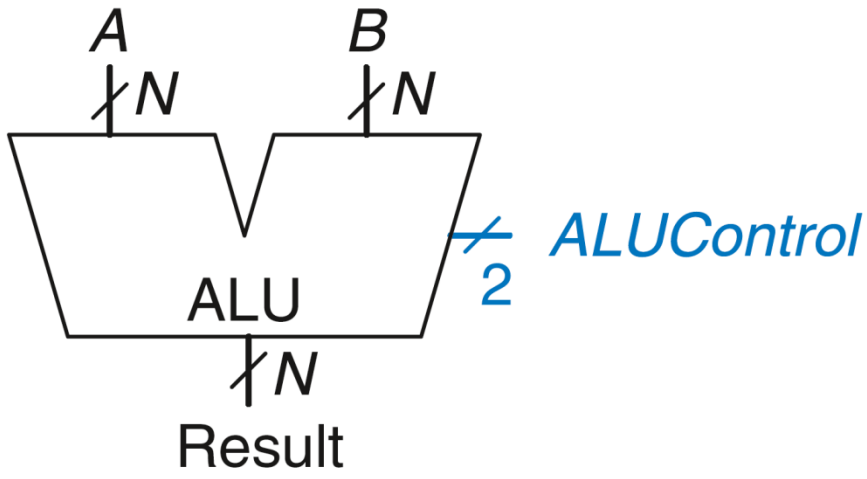
ALU: 算术逻辑单元Arithmetic Logic Unit

ALU 包括的运算: 将多种算术和逻辑运算组合到一个计算单元内（计算机的核心）， 模块化、规则化思想

- Addition
- Subtraction
- AND
- OR

ALU: 算术逻辑单元

$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR



Example: Perform *A* + *B*

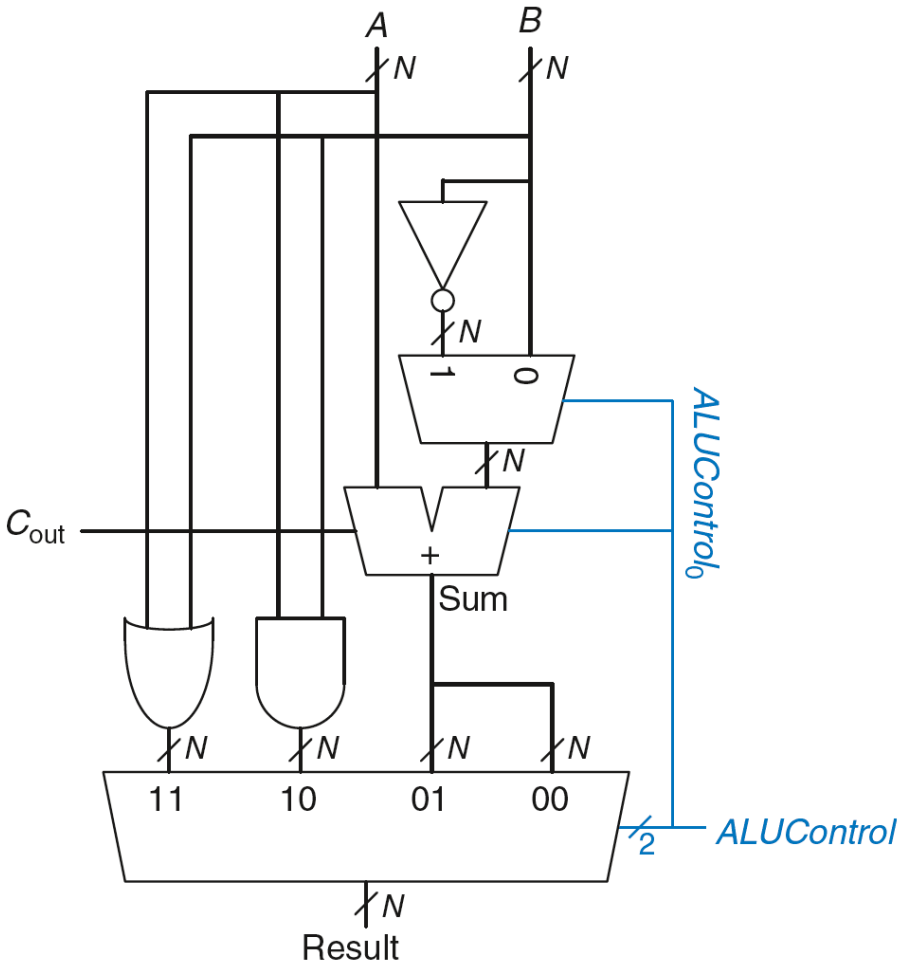
ALUControl = 00

$Result = A + B$

ALU: 算术逻辑单元

$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

Example: **Perform *A* OR *B***



ALU: 算术逻辑单元

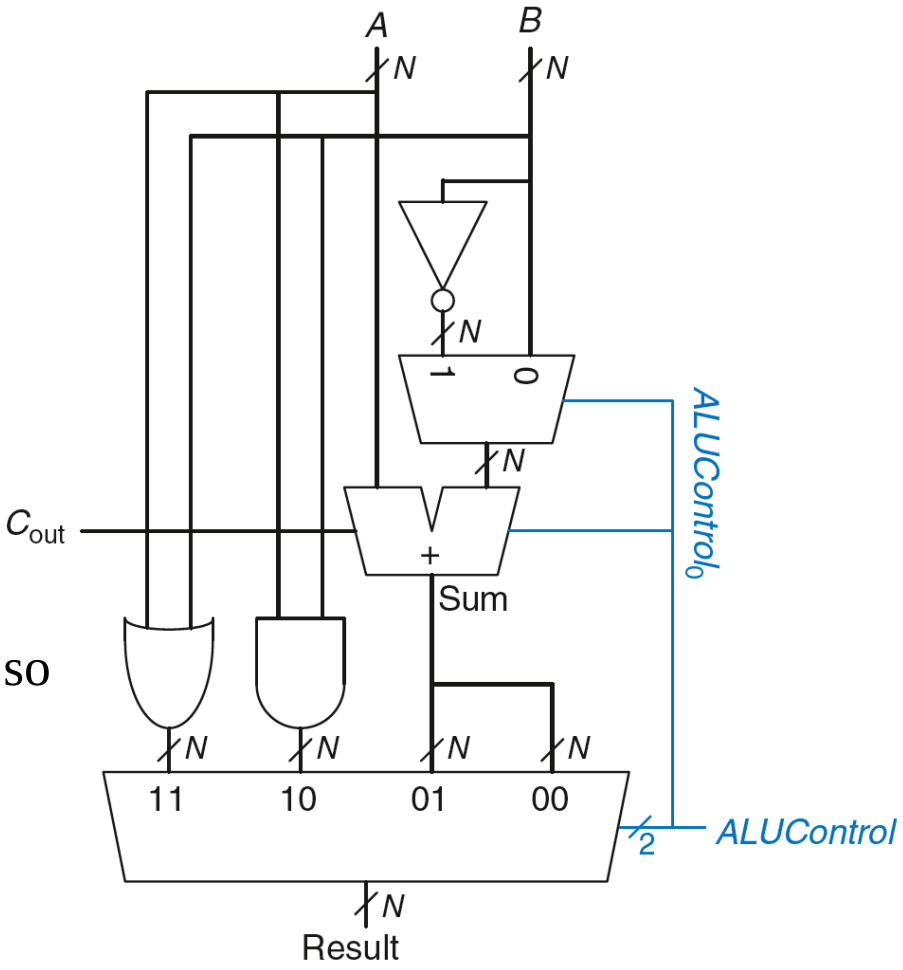
$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

Example: Perform *A* OR *B*

$ALUControl_{1:0}$ = 11

Mux selects output of OR gate as *Result*, so

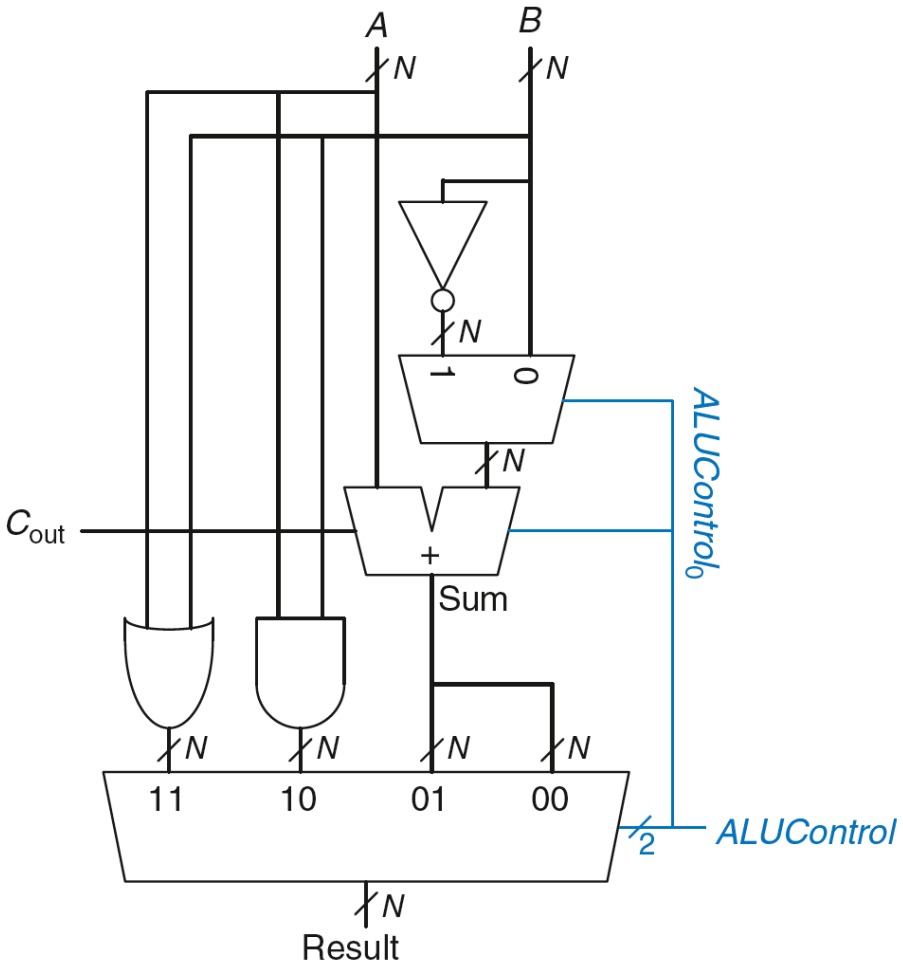
Result* = *A* OR *B



ALU: 算术逻辑单元

$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

Example: **Perform *A* + *B***



ALU: 算术逻辑单元

$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

Example: Perform *A* + *B*

$ALUControl_{1:0}$ = 00

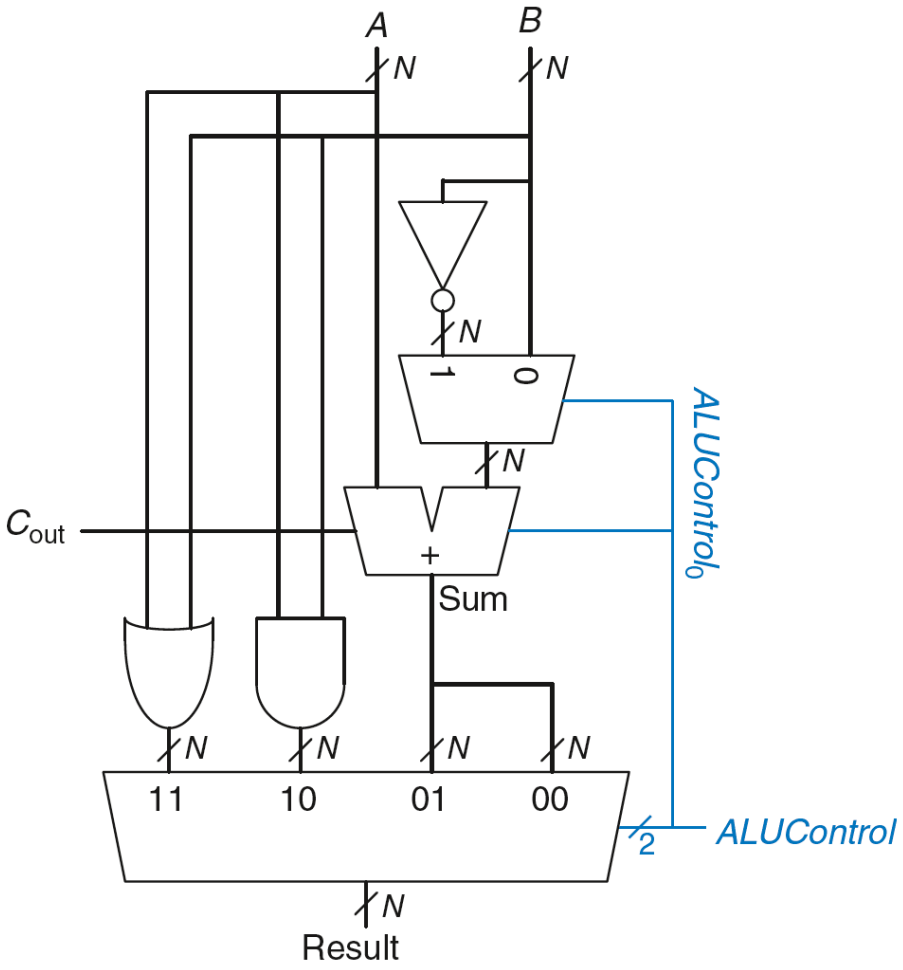
$ALUControl_0$ = 0, so:

C_{in} to adder = 0

2nd input to adder is *B*

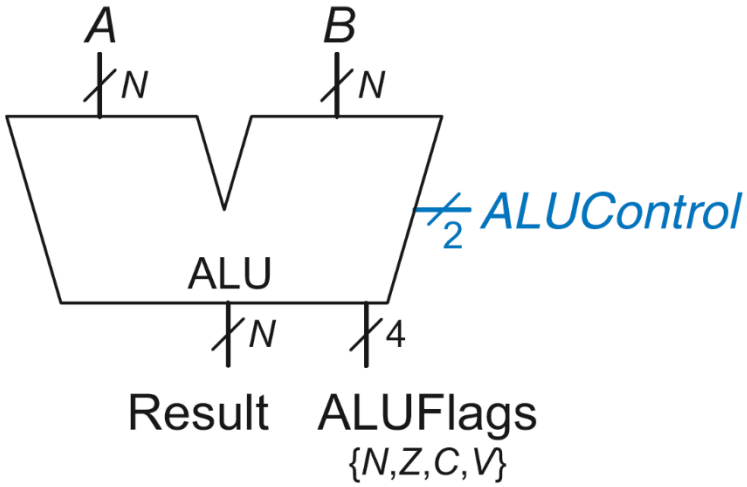
Mux selects *Sum* as *Result*, so

Result* = *A* + *B

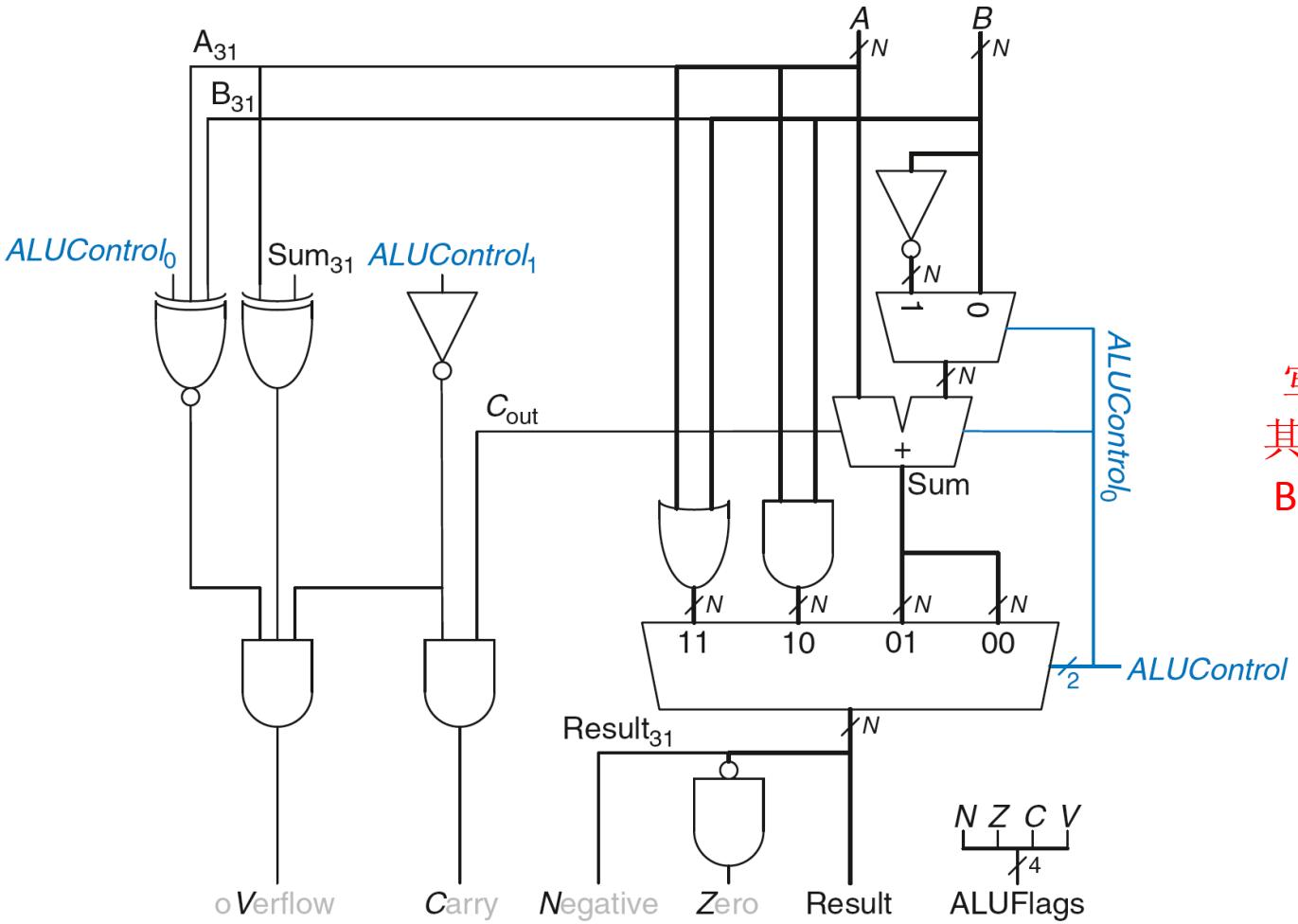


带状态位ALU

Flag	Description
N	*Result* is **N**egative
Z	*Result* is **Z**ero
C	Adder produces **C**arry out
V	Adder o**V**erflowed

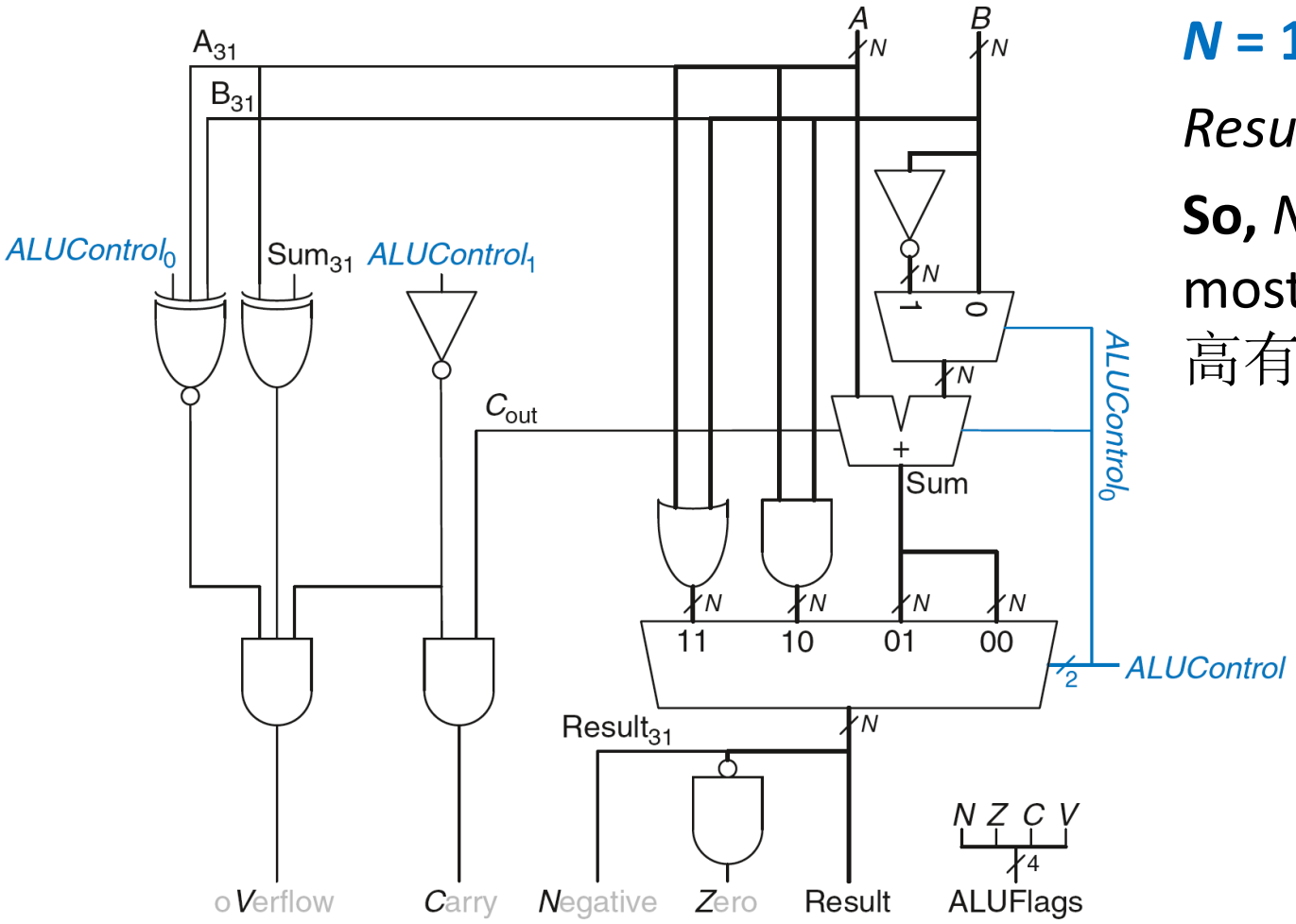


带状态位ALU



写出SystemVerilog及其Testbench，其中A、B为32bits输入（带符号位）

带状态位ALU: Negative

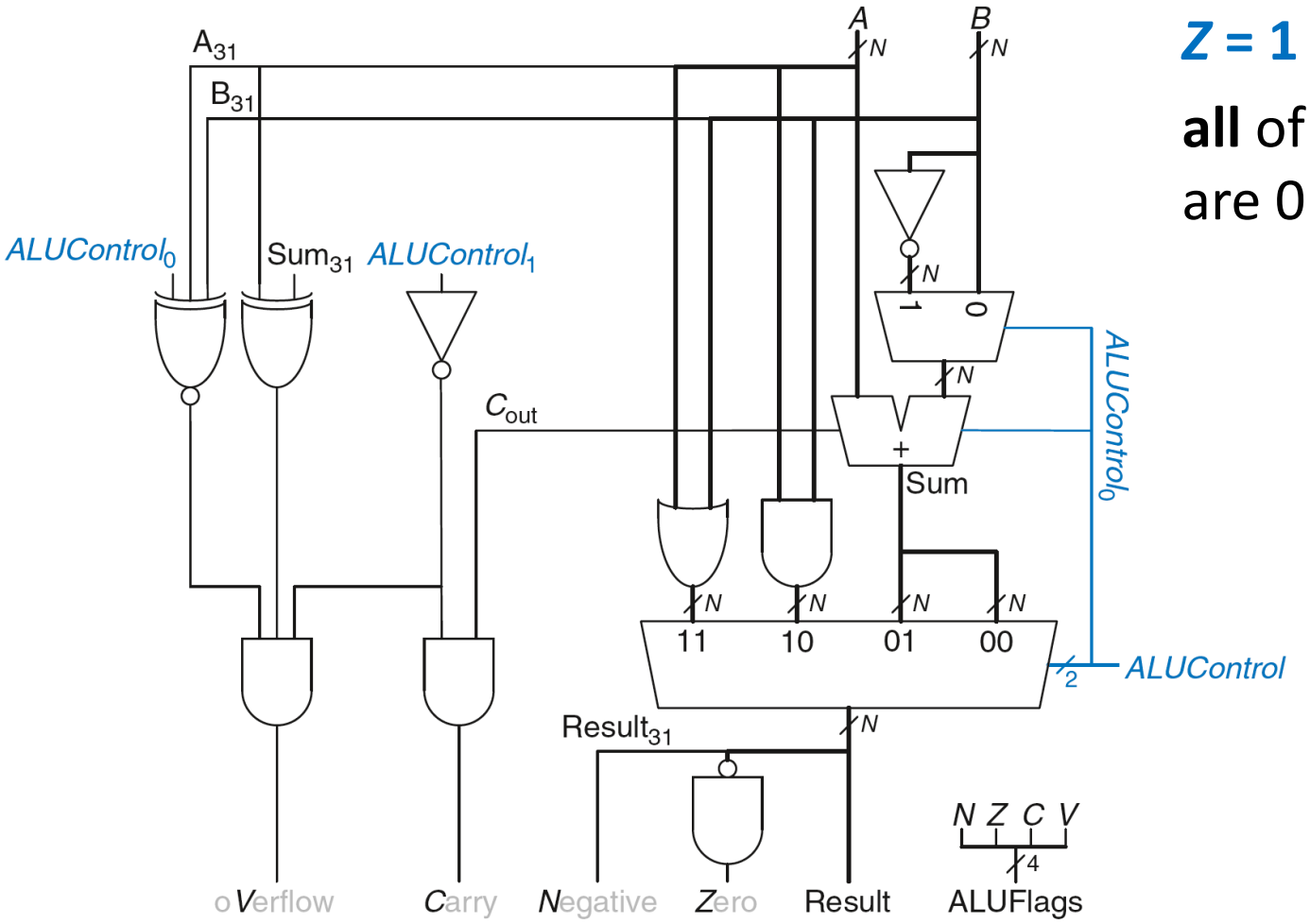


***N* = 1** if:

Result is **negative**

So, *N* is connected to most significant bit（最高有效位） of *Result*

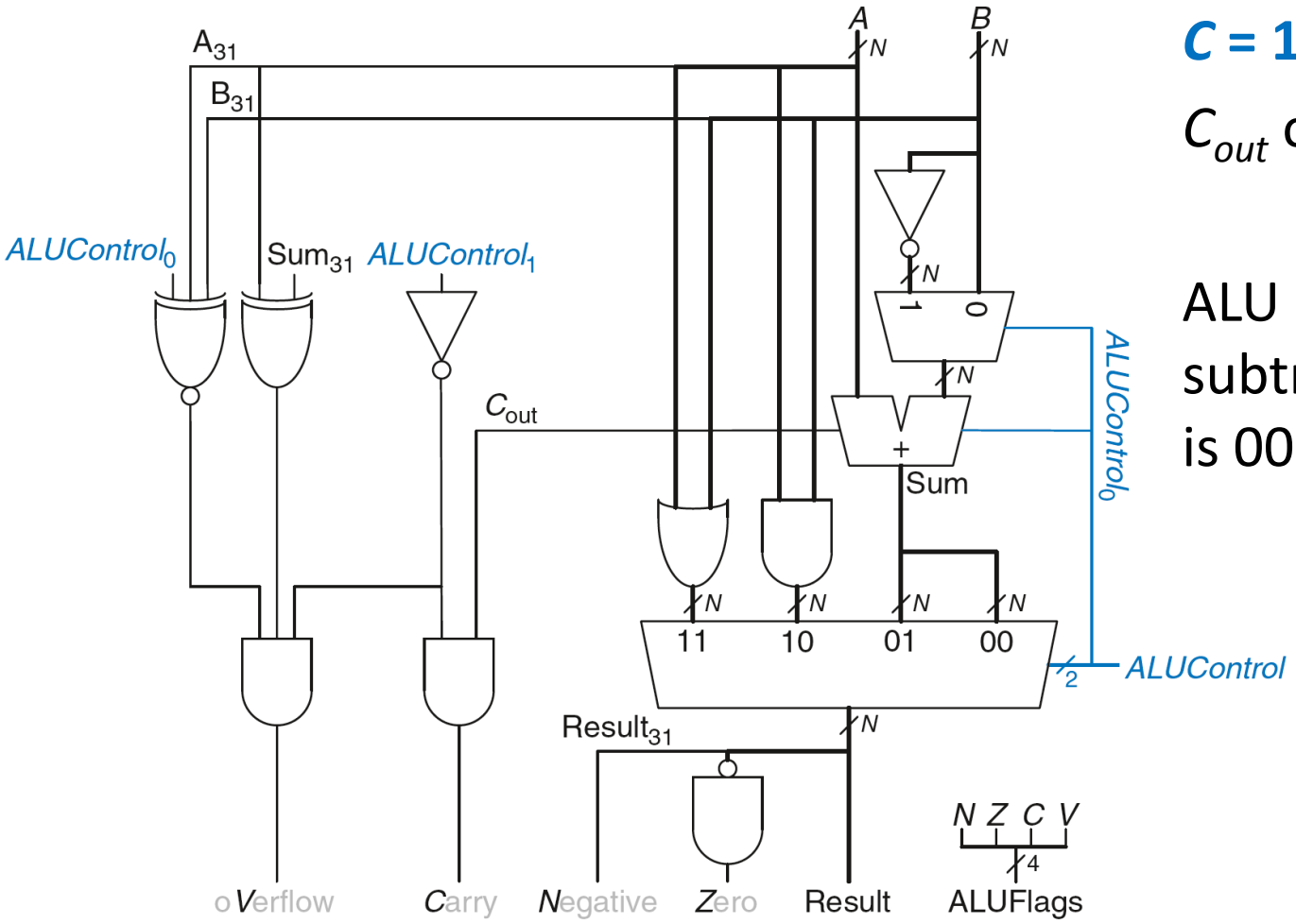
带状态位ALU: Zero



Z = 1 if:

all of the bits of *Result* are 0

带状态位ALU: : Carry



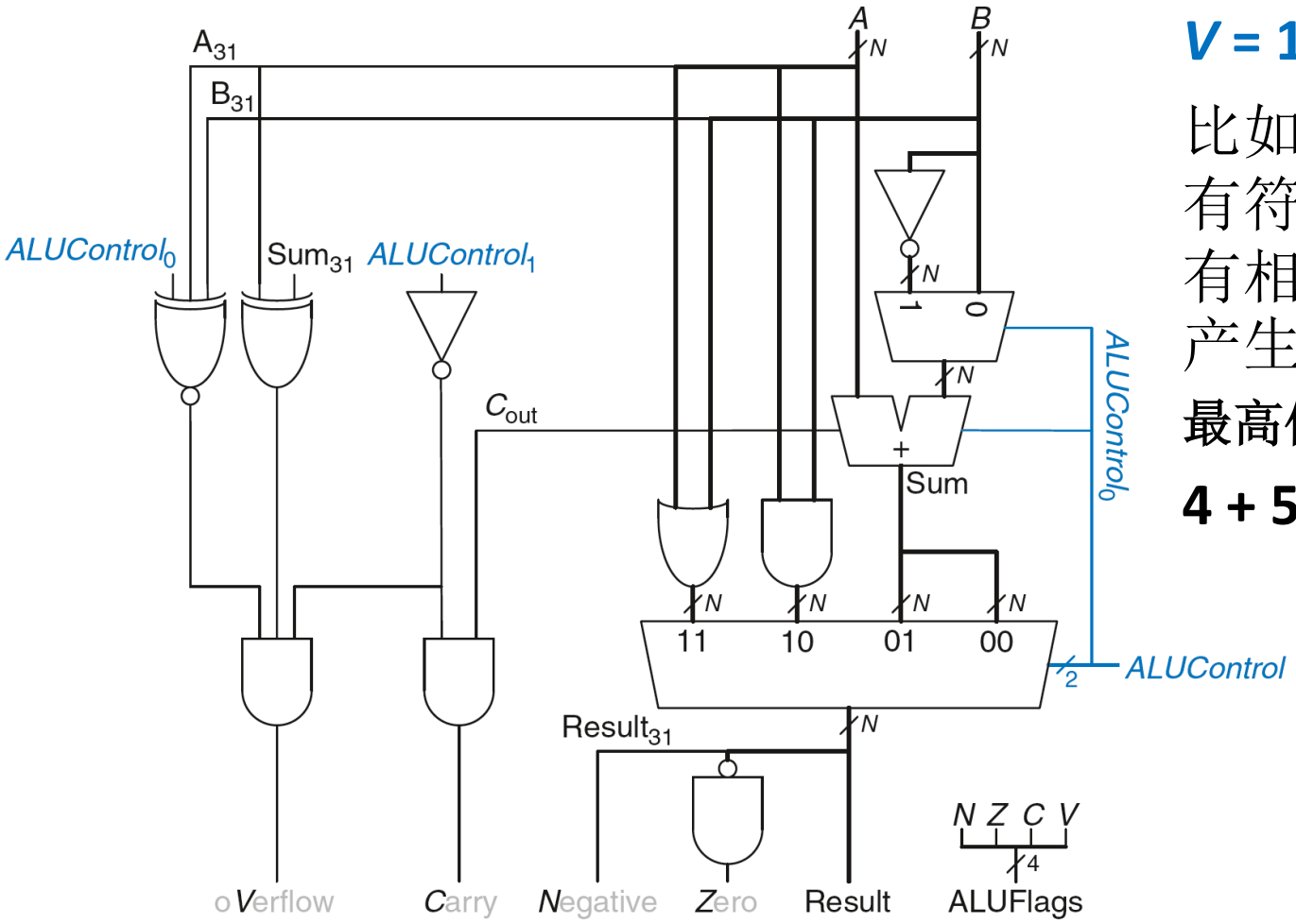
C = 1 if:

C_{out} of Adder is 1

AND

ALU is adding or subtracting (ALUControl is 00 or 01)

带状态位ALU：oVerflow



***V* = 1** if:

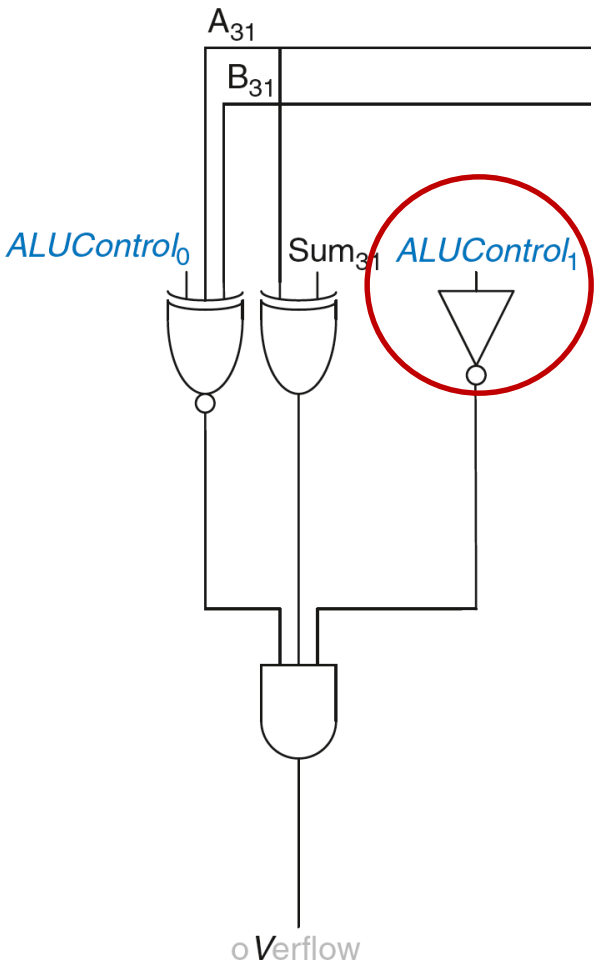
比如：当两个相同的有符号数相加产生具有相反符号的结果时产生溢出。

最高位为符号位，比如：

4 + 5 =0100 + 0101

=1001 = -7

带状态位ALU: oVerflow



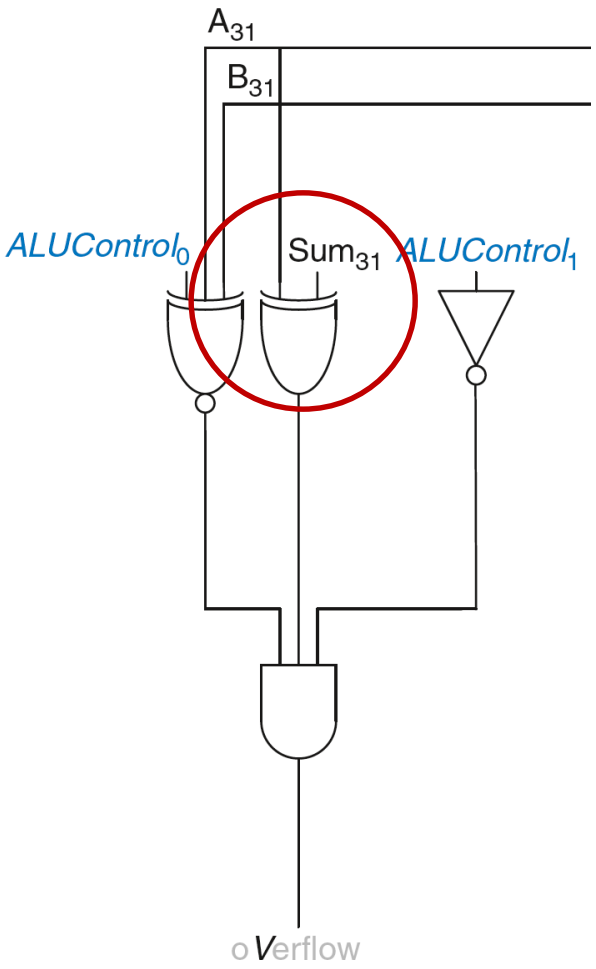
***V* = 1** if:

ALU is performing addition or subtraction

（执行了加或者减操作）

($ALUControl_1 = 0$)

带状态位ALU: oVerflow



***V* = 1** if:

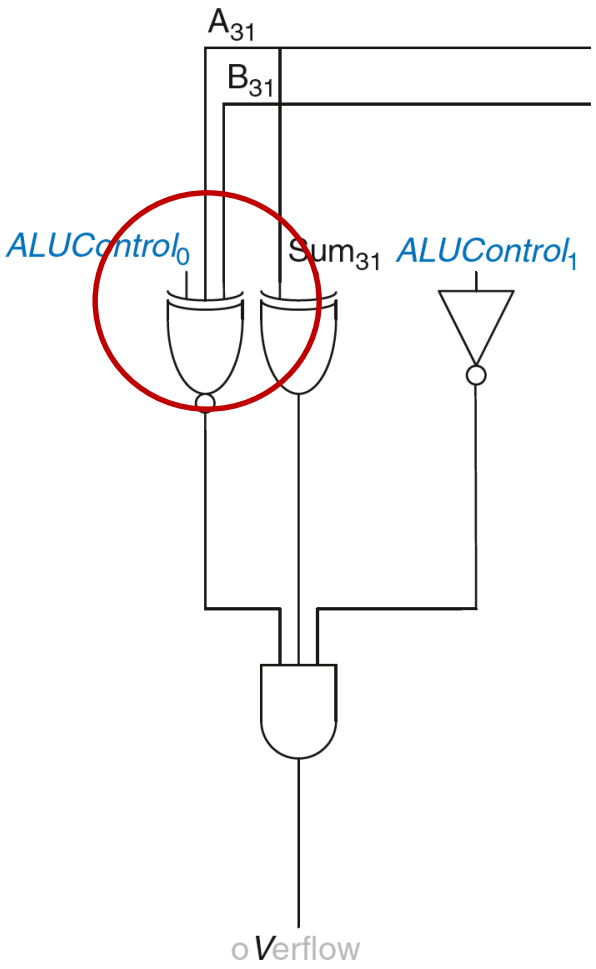
ALU is performing addition or subtraction

($ALUControl_1 = 0$)

AND

A and Sum have opposite signs （符号相反）

带状态位ALU: oVerflow



***V* = 1** if:

ALU is performing addition or subtraction

($ALUControl_1 = 0$)

AND

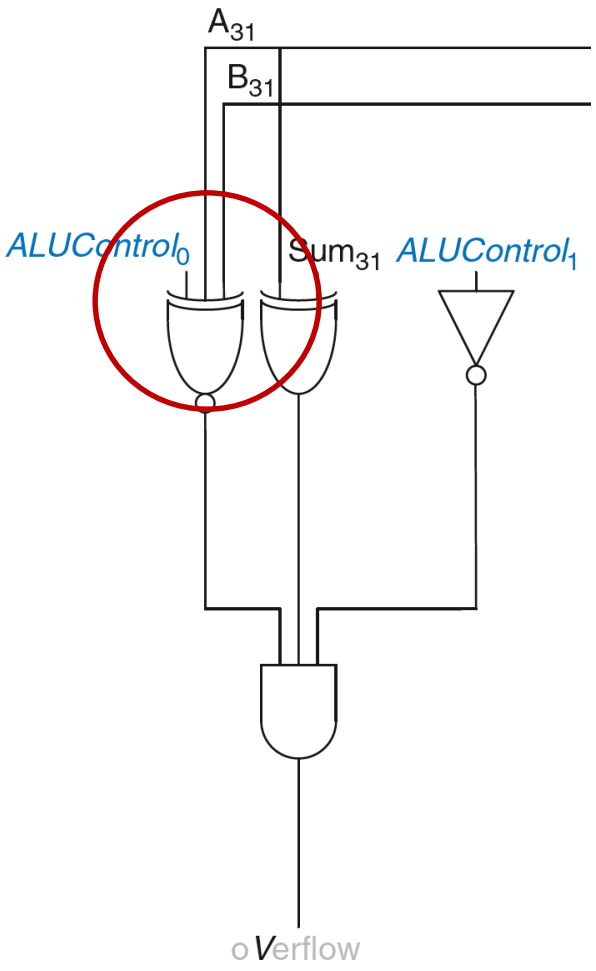
A and Sum have opposite signs

AND

A and B have same signs 执行addition **OR**

A and B have different signs 执行subtraction

带状态位ALU: oVerflow



***V* = 1** if:

ALU is performing addition or subtraction

($ALUControl_1 = 0$)

AND

A and Sum have opposite signs

AND

A and B have same signs 执行addition

($ALUControl_0 = 0$) **OR**

A and B have different signs 执行 subtraction

($ALUControl_0 = 1$)

比如：5+6=0101+0110=1011=-5

(-6)-7=1010-0111=0011=3

移位器

逻辑移位器: shifts value to left or right and fills empty spaces with 0's

- Ex: 11001 >> 2 =
- Ex: 11001 << 2 =

LSR LSL

算术移位器: same as logical shifter, but on right shift, fills empty spaces with the old most significant bit (msb) （对有符合乘法和除法有用）

- Ex: 11001 >>> 2 =
- Ex: 11001 <<< 2 =

ASR ASL

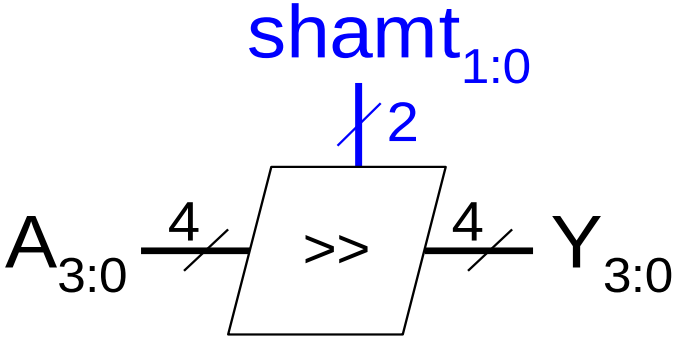
循环移位器: rotates bits in a circle, such that bits shifted off one end are shifted into the other end

- Ex: 11001 ROR 2 =
- Ex: 11001 ROL 2 =

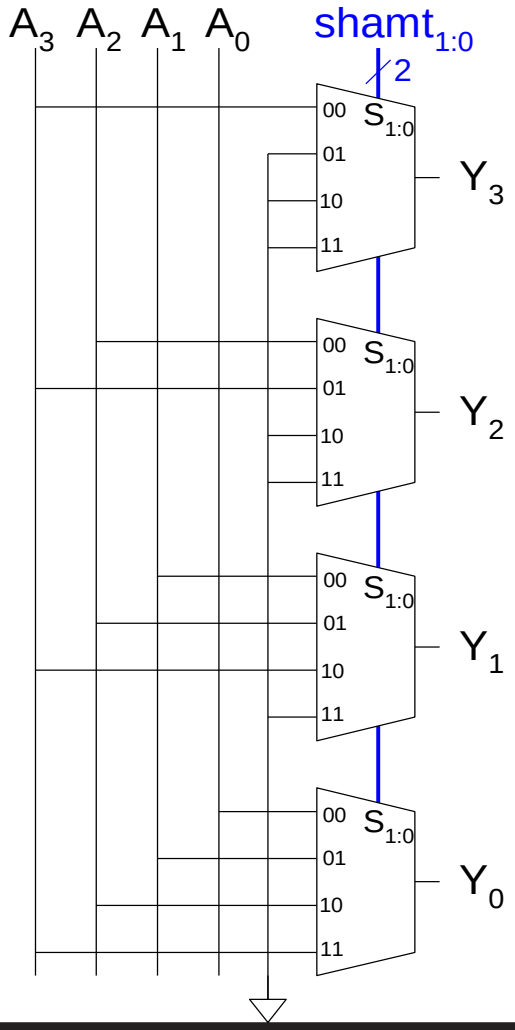
ROR ROL

Copyright © 2007 Elsevier

移位器的设计



一个N位移位器可以用N个N:1选择器构成。根据Log_2N位选择线的值，输入从位0移动到位N-1:



移位器与乘法器和除法器

- $A << N = A \times 2^N$
  - **Example:** 00001 << 2 = 00100 $(1 \times 2^2 = 4)$
  - **Example:** 11101 << 2 = 10100 $(-3 \times 2^2 = -12)$
- $A >>> N = A \div 2^N$
  - **Example:** 01000 >>> 2 = 00010 $(8 \div 2^2 = 2)$
  - **Example:** 11100 >>> 2 = 11111 $(-4 \div 2^2 = -1)$

乘法器

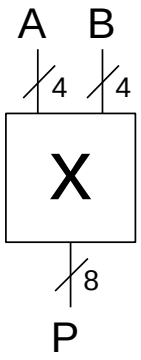
- **部分积：乘数的1位乘以被乘数的所有**
- 移位部分积再相加 即得结果

Decimal		**Binary**
230	multiplicand	0101
x 42	multiplier	x 0111
460	partial products	0101
+ 920		0101
9660		0101
		+ 0000
	result	0100011

230 x 42 = 9660

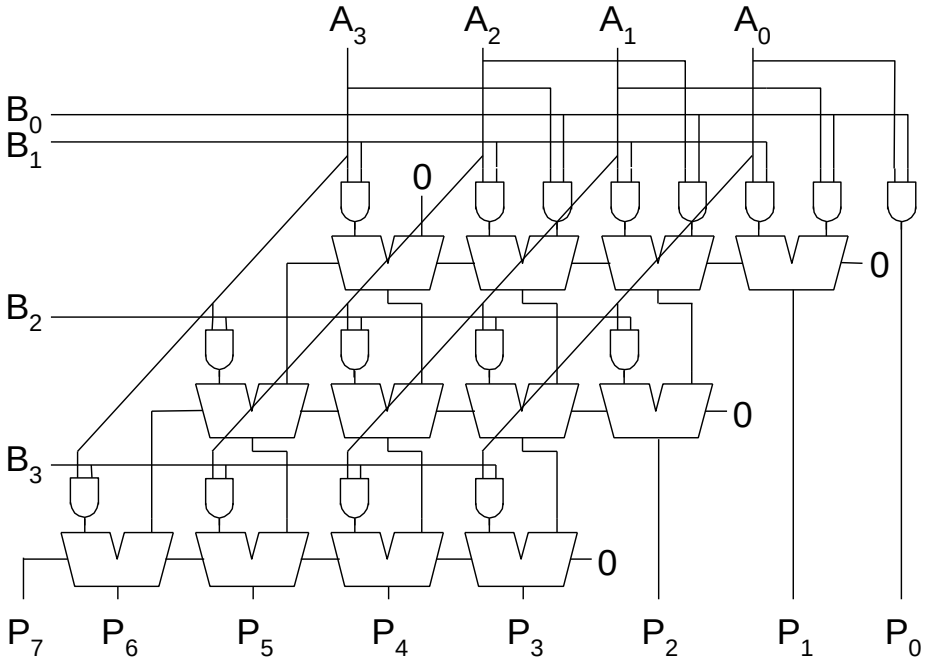
5 x 7 = 35

4 x 4 乘法器



二进制乘法部分积相当于在做AND运算

$$
\begin{array}{rcccccccc}
 & & & A_3 & A_2 & A_1 & A_0 & & \\
\times & & & B_3 & B_2 & B_1 & B_0 & & \\
\hline
 & & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 & & \\
 & & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & & & \\
 & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & & & \\
+ & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & & \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$



除法器

$$A/B = Q + R/B$$

Decimal Example: $2584/15 = 172\ R4$

```
   172 R4
15|2584
  -15
   108
  -105
     34
    -30
      4
```

除法器

$$A/B = Q + R/B$$

Decimal Example: 2584/15 = 172 R4

1：R初始为0，被除数最高有效位成为R的最低位
2：中间余数重复地减去B，看相减结果D
3：如果D为负数，则商Q为0，并忽略该值
4：否则D不为负数，中间余数也更新为差D，并且被除数的最高有效位成为D的最低有效位

```
      172 R4
15 | 2584
     -15
     108
    -105
      34
     -30
       4
```

```
0002
- 15
 -13
```

$\frac{0}{3}\ \frac{\ }{2}\ \frac{\ }{1}\ \frac{\ }{0}$

```
0025
- 15
  10
```

$\frac{0}{3}\ \frac{1}{2}\ \frac{\ }{1}\ \frac{\ }{0}$

```
0108
- 105
    3
```

$\frac{0}{3}\ \frac{1}{2}\ \frac{7}{1}\ \frac{\ }{0}$

```
0034
- 30
   4
```

$\frac{0}{3}\ \frac{1}{2}\ \frac{7}{1}\ \frac{2}{0}$

除法器

A/B = Q + R/B

Decimal: 2584/15 = 172 R4 **Binary: 1101/0010 = 0110 R1**

```
0002
- 15
 -13
```
$\frac{0}{3\ 2\ 1\ 0}$

```
0001
-0010
1111
```
$\frac{0}{3\ 2\ 1\ 0}$

```
0025
- 15
  10
```
$\frac{0\ 1}{3\ 2\ 1\ 0}$

```
0011
-0010
0001
```
$\frac{0\ 1}{3\ 2\ 1\ 0}$

```
0108
-105
   3
```
$\frac{0\ 1\ 7}{3\ 2\ 1\ 0}$

```
0010
-0010
0000
```
$\frac{0\ 1\ 1}{3\ 2\ 1\ 0}$

```
0034
- 30
   4
```
$\frac{0\ 1\ 7\ 2}{3\ 2\ 1\ 0}$

```
0001
-0010
1111
```
$\frac{0\ 1\ 1\ 0}{3\ 2\ 1\ 0}$ R1

除法器

A/B = Q + R/B

R’ = 0

for i = N-1 to 0

  R = {R’ << 1, A_i}

  D = R - B

  if D < 0, Q_i= 0; R’= R

  else Q_i= 1; R’= D

R=R’

注释：R及R’为临时变量

i为了依次从A最高位N取到0，A[N-1:0]

R的高位值是R‘每次移动一位的值，再紧接着取A的i位的值 组成。

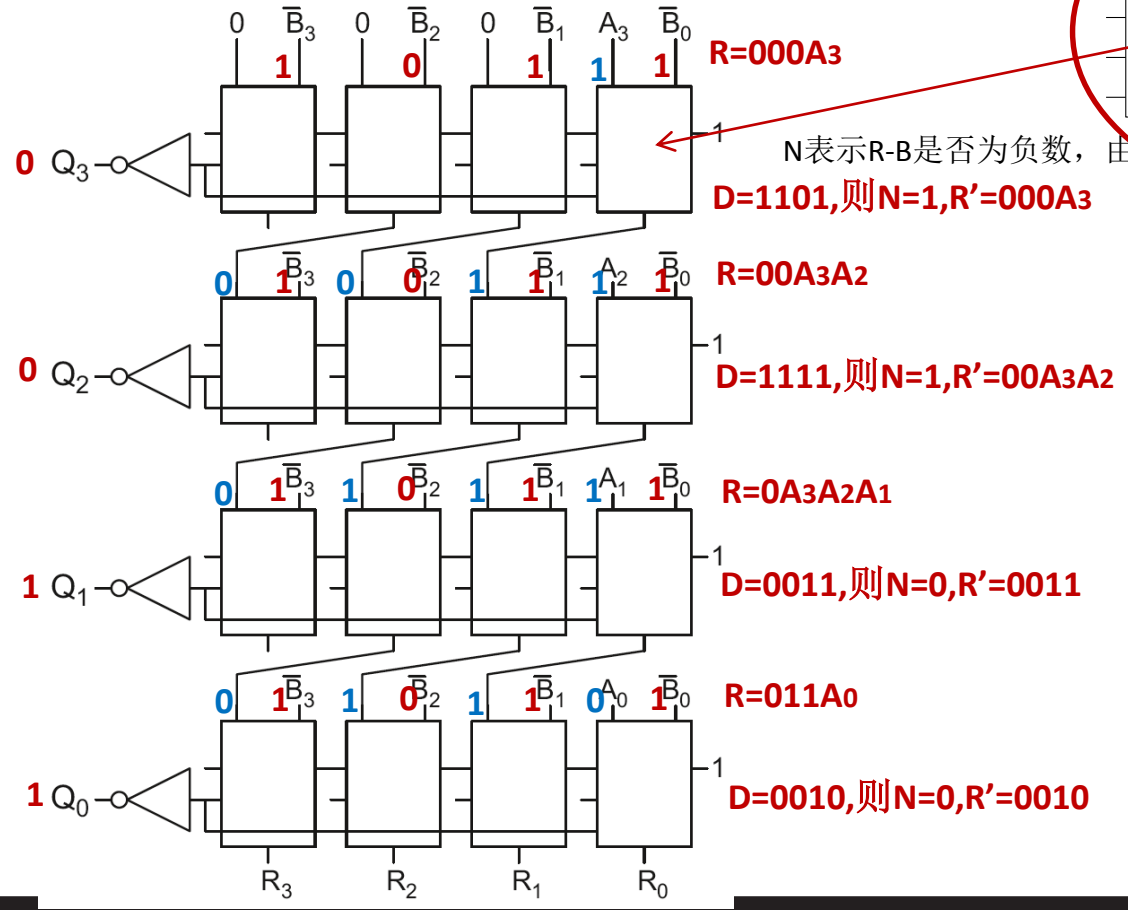
两个4比特数相除，商和余数最多只有4位

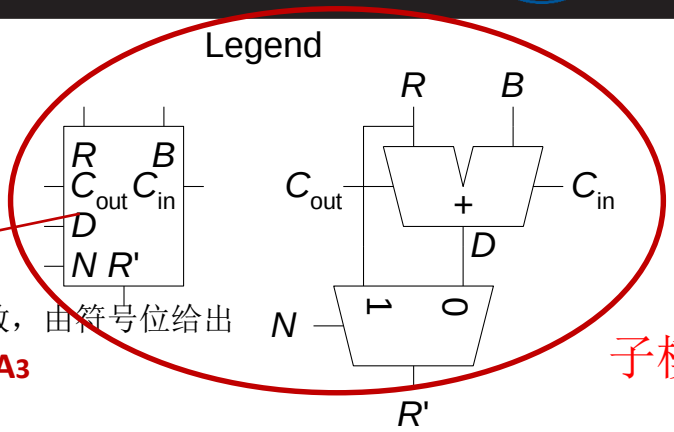
Binary: 1101/10 = 0110 R1

```
 0001
-0010      0 _ _ _
 1111      3 2 1 0

 0011
-0010      0 1 _ _
 0001      3 2 1 0

 0010
-0010      0 1 1 _
 0000      3 2 1 0

 0001
-0010      0 1 1 0   R1
 1111      3 2 1 0
```

4 x 4 除法器

4位阵列除法器，A/B，产生商Q，和余数R
N表示R-B是否为负数，硬件的每一列计算一次循环





Division: A/B = Q + R/B

R′ = 0

for *i* = N-1 to 0

R = {R′ << 1, A_i}

D = R - B

if D < 0 then Q_i=0, R′ =R

else Q_i=1, R′ =D

R=R′

写出SystemVerilog及其Testbench，并复用子模块

举例: 14/4=3+2/4

即1110(A)/0100(B)=0011(Q)+0010(R')/0100(B)

数制

二进制表示整数的方式

- 正整数
  - 无符号二进制数
- 负整数
  - 补码
  - 带符号的

小数表示呢?

数制

小数，小数点咋表示？

- 定点数: binary point fixed （小数点固定）
- 浮点数: binary point floats to the right of the most significant 1 （小数点浮动）

定点数

- 6.75 using 4 **integer bits** and 4 **fraction bits**:

$$01101100$$

$$0110.1100$$

$$2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$$

  - 0.5 0.25 0.125 0.0625 0.03125, 大于则为1 否则为0
  - 小数点被隐藏
  - 小数部分与整数部分事先约定好的

定点数的例子

- Represent 7.5_{10} using 4 integer bits and 4 fraction bits.

带符号定点数

- 表示:
  - Sign/magnitude
  - Two's complement
- **Example:** Represent -7.5_{10} using 4 integer and 4 fraction bits
  - **Sign/magnitude:**
  - **Two's complement:** （补码）

    1. +7.5:
    2. Invert bits:
    3. Add 1 to lsb: + 1

浮点数

- 解决了定点数里面整数与小数位长固定的限制
- 类似于十进制数里面的科学计数法

- For example, write 273_{10} in scientific notation:

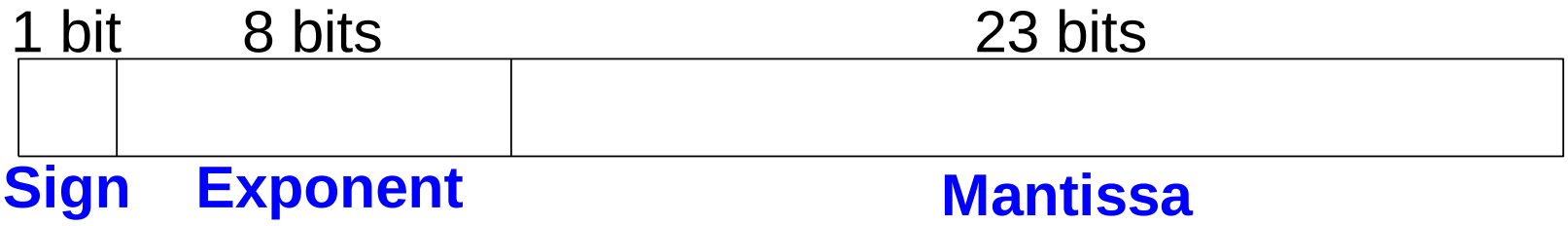
$$\mathbf{273 = 2.73 \times 10^2}$$

- In general, a number is written in scientific notation as:

$$\mathbf{\pm M \times B^E}$$

  - **M** = 尾数
  - **B** = 基数
  - **E** = 阶码
  - In the example, M = 2.73, B = 10, and E = 2

浮点数



- **Example:** represent the value 228_{10} using a 32-bit floating point representation

We show three versions – final version is called the **IEEE 754 floating-point standard**

浮点数表示1

1. Convert decimal to binary

$$228_{10} = 11100100_2$$

2. Write the number in “binary scientific notation”:

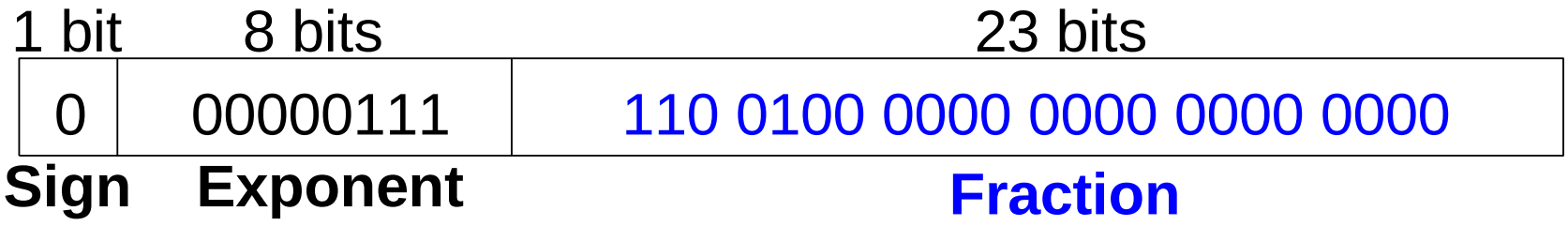
$$11100100_2 = 1.11001_2 \times 2^7$$

3. Fill in each field of the 32-bit floating point number:
   - The sign bit is positive (0)
   - The 8 exponent bits represent the value 7
   - The remaining 23 bits are the mantissa

1 bit	8 bits	23 bits
0	00000111	11 1001 0000 0000 0000 0000
Sign	**Exponent**	**Mantissa**

浮点数表示 2

- First bit of the mantissa is always 1:
  - $228_{10} = 11100100_2 = \mathbf{1.11001} \times 2^7$
- So, no need to store it: *implicit leading 1*
- Store just fraction bits in 23-bit field

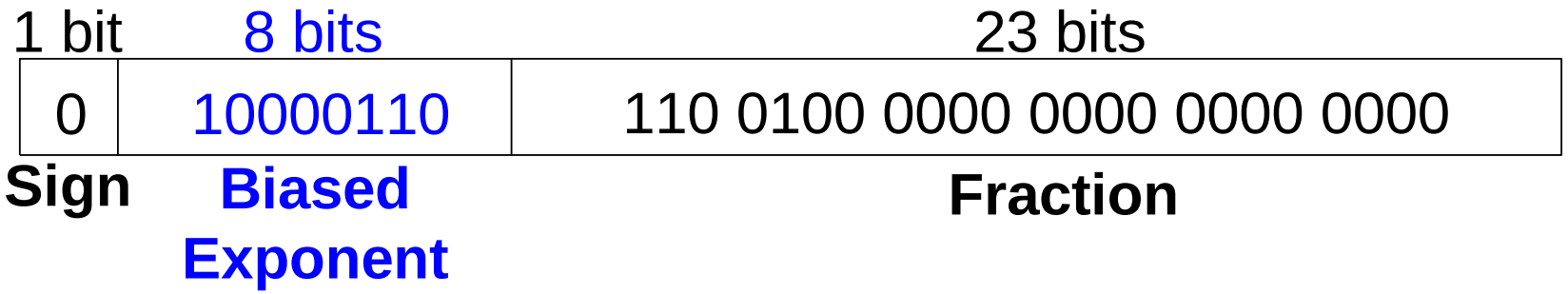


浮点数表示 3

- *Biased exponent*: bias = 127 (01111111_2)
  - 偏置阶码= 偏置+ 阶码 （**阶码需要表示正数和负数**）
  - 阶码： Exponent of 7 is stored as:

$$127 + 7 = 134 = 0x10000110_2$$

- The **IEEE 754 32-bit floating-point representation** of 228_{10}



in hexadecimal: **0x43640000**

Floating-Point Example

Write -58.25_{10} in floating point (IEEE 754)

浮点数例子

Write -58.25_{10} in floating point (IEEE 754)

1. Convert decimal to binary:

   $58.25_{10} = \mathbf{111010.01_2}$

2. Write in binary scientific notation:

   $\mathbf{1.1101001 \times 2^5}$

3. Fill in fields:

   Sign bit: 1 (negative)
   8 exponent bits: (127 + 5) = 132 = $\mathbf{10000100_2}$
   23 fraction bits: **110 1001 0000 0000 0000 0000**

1 bit	8 bits	23 bits
1	100 0010 0	110 1001 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

in hexadecimal: **0xC2690000**

浮点数特殊情况

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
- ∞	1	11111111	00000000000000000000000
NaN	X	11111111	non-zero

浮点数的精度

- 点精度浮点数:
  - 32-bit
  - 1 sign bit, 8 exponent bits, 23 fraction bits
  - bias = 127

- 双精度浮点数:
  - 64-bit
  - 1 sign bit, 11 exponent bits, 52 fraction bits
  - bias = 1023

浮点数的舍入

- **上溢:** number too large to be represented（舍为无穷）
- **下溢:** number too small to be represented （舍为0）
- **舍入模式:**
  - 向下舍
  - 向上舍
  - 向零舍
  - 向最近舍
- **Example:** round 1.100101 (1.578125) to only 3 fraction bits
  - Down: 1.100
  - Up: 1.101
  - Toward zero: 1.100
  - To nearest: 1.101 (1.625 is closer to 1.578125 than 1.5 is)

浮点数加法

1. 提取阶码和小数位
2. 加上前导1，形成尾数
3. 比较阶码
4. 如果需要对较小的尾数移位
5. 尾数相加
6. 规范化尾数，并需要时调整阶码
7. 舍入结果
8. 把阶码和小数组合成浮点数

浮点数加法例子

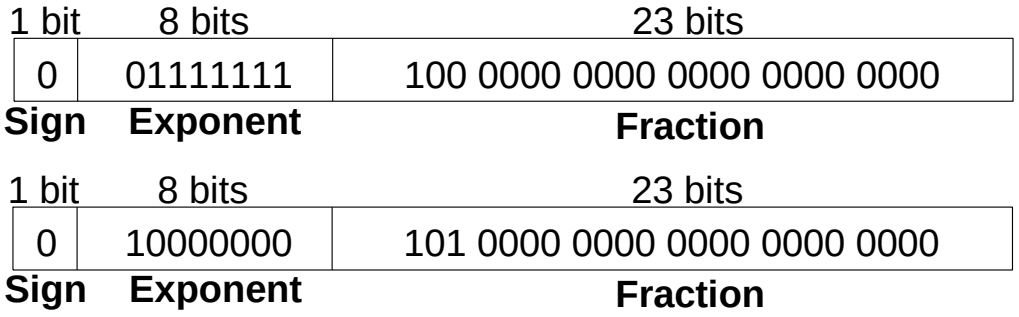
Add the following floating-point numbers:

0x3FC00000

0x40500000

浮点数加法例子

1. Extract exponent and fraction bits



For first number (N1): S = 0, E = 127, F = .1

For second number (N2): S = 0, E = 128, F = .101

2. Prepend leading 1 to form mantissa

N1: 1.1

N2: 1.101

浮点数加法例子

3. Compare exponents

127 – 128 = -1, so shift N1 right by 1 bit

4. Shift smaller mantissa if necessary

shift N1's mantissa: 1.1 >> 1 = 0.11 $(\times 2^1)$

5. Add mantissas

$$\begin{array}{rl} & 0.11 \times 2^1 \\ + & 1.101 \times 2^1 \\ \hline & 10.011 \times 2^1 \end{array}$$

浮点数加法例子

6. **Normalize mantissa and adjust exponent if necessary**

   $10.011 \times 2^1 = 1.0011 \times 2^2$

7. **Round result**

   No need (fits in 23 bits)

8. **Assemble exponent and fraction back into floating-point format**

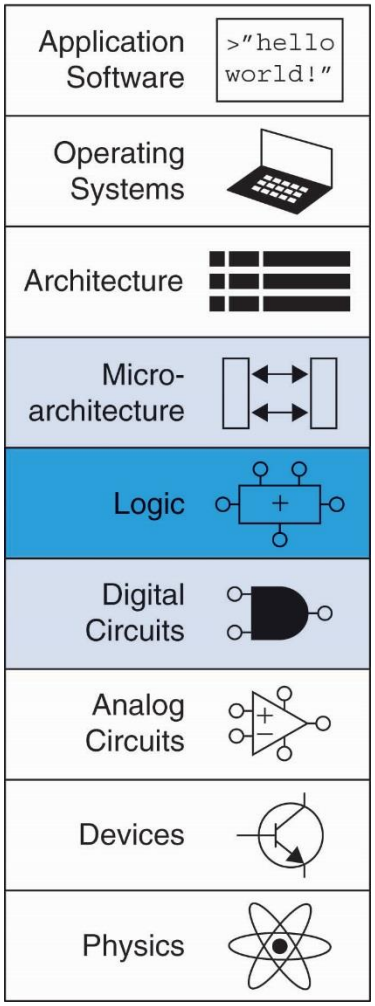
   $S = 0, E = 2 + 127 = 129 = 10000001_2, F = 001100..$

1 bit	8 bits	23 bits
0	10000001	001 1000 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

in hexadecimal: **0x40980000**

第五章 数字模块

- 介绍
- 算术电路
- 时序电路模块
- 存储器阵列
- 可编程逻辑

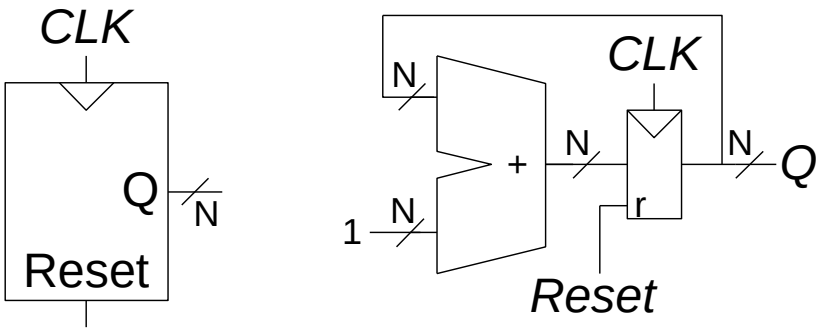


时序电路模块----计数器

- 每一个时钟沿加1

- Example uses:
  - 数字时钟显示
  - 程序计数: keeps track of current instruction executing

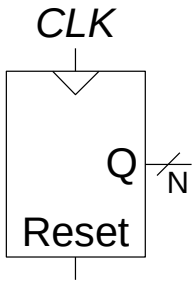
Symbol **Implementation**



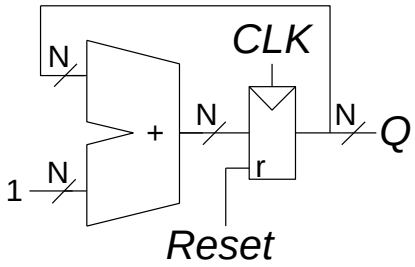
计数器Verilog (FSM style)

```
module counter (input  logic          clk, reset,
                output logic [N-1:0] q);
  logic [N-1:0] nextq;

  // register
  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= nextq;

  // next state
  assign nextq = q + 1;
endmodule
```

Symbol



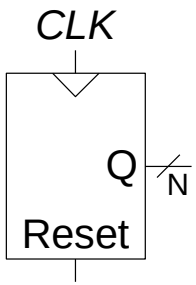
Implementation



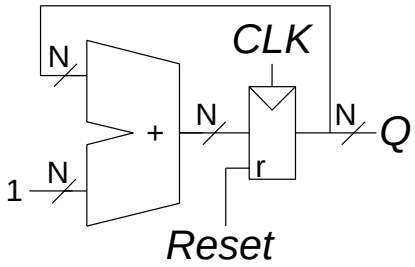
计数器Verilog (better idiom)

```
module counter (input  logic          clk, reset,
                output logic [N-1:0] q);
always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= q+1;
endmodule
```

Symbol



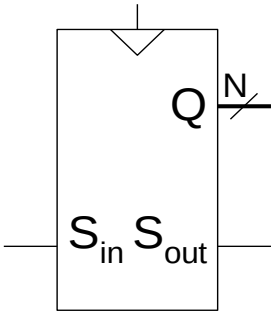
Implementation



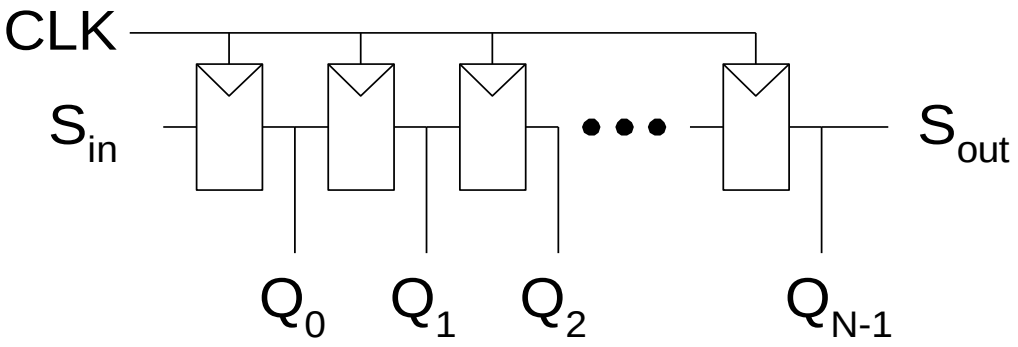
移位寄存器

- 每个时钟沿移动一位
- 最后部分每个时钟沿输出
- 用途： 串并转换器: converts serial input (S_{in}) to parallel output ($Q_{0:N-1}$)

Symbol:

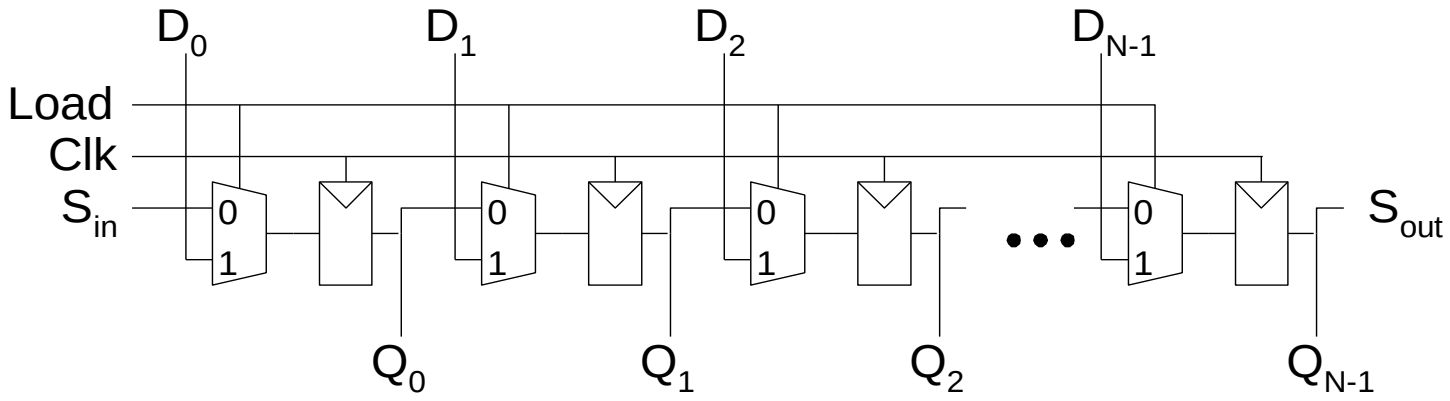


Implementation: 触发器串联而成



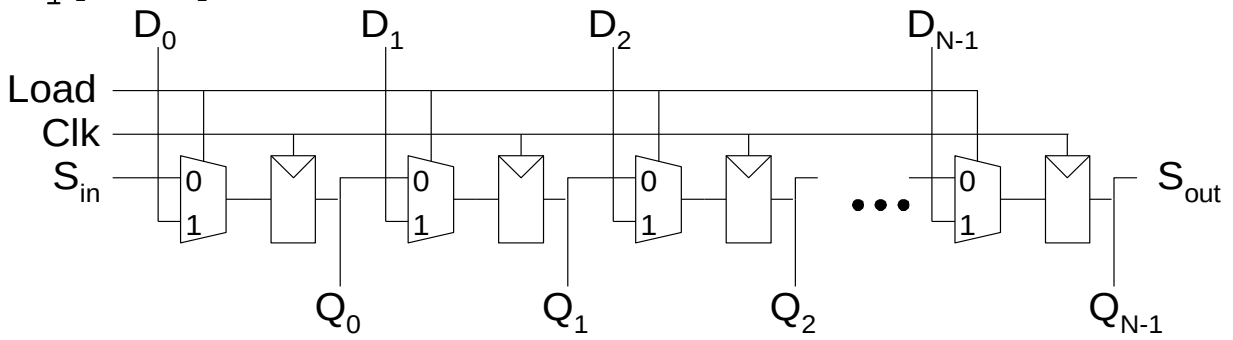
带并行加载的移位寄存器

- When ***Load* = 1**, acts as a normal *N*-bit register
- When ***Load* = 0**, acts as a shift register
- Now can act as a *serial-to-parallel converter* (S_{in} to $Q_{0:N-1}$) or a *parallel-to-serial converter* ($D_{0:N-1}$ to S_{out}最高位输出)



带并行加载的移位寄存器

```
module shiftreg(input  logic          clk,
                input  logic          load, sin,
                input  logic [N-1:0]  d,
                output logic [N-1:0]  q,
                output logic          sout);
always_ff @(posedge clk)
    if (load)  q <= d;
    else       q <= {q[N-2:0], sin};
  assign sout = q[N-1];
endmodule
```



扫描链

- 组合逻辑的测试
- 时序逻辑的测试（可扫描触发器Scan-DFF）

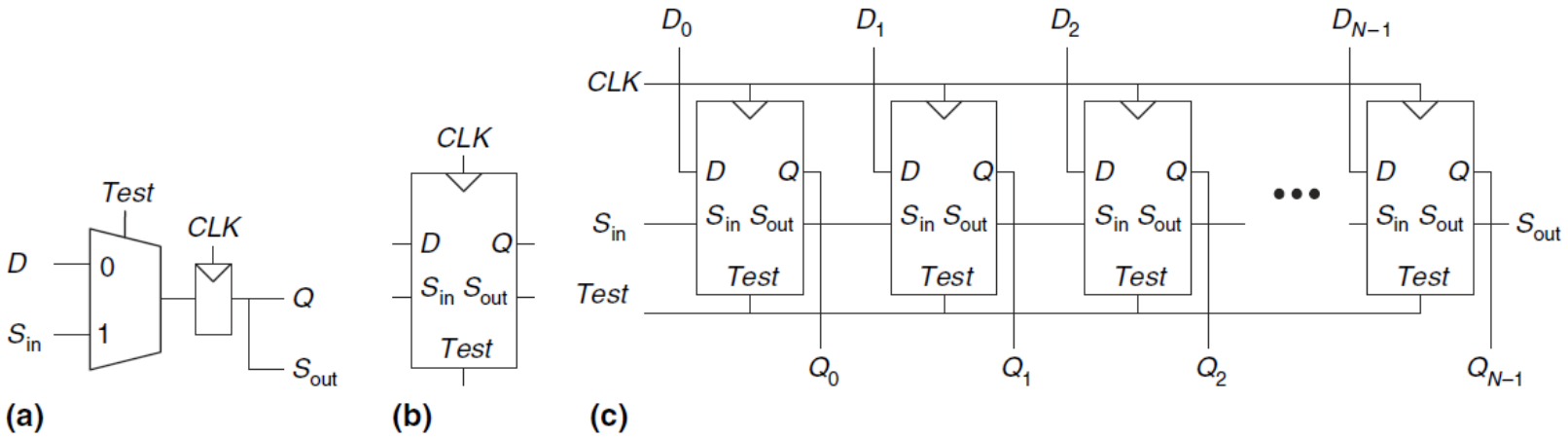
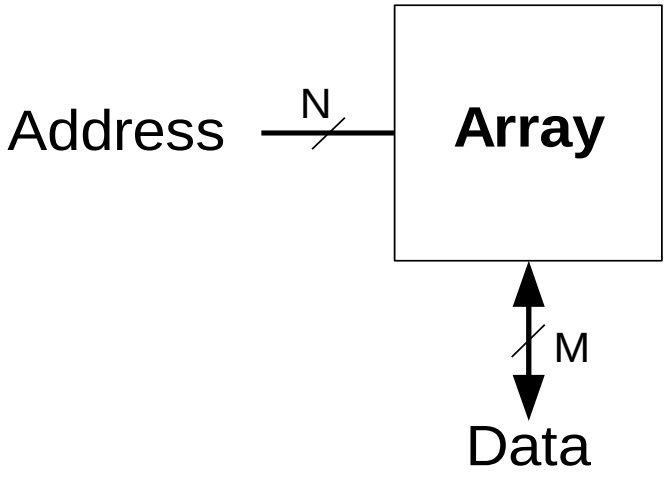


Figure 5.37 Scannable flip-flop: (a) schematic, (b) symbol, and (c) *N*-bit scannable register

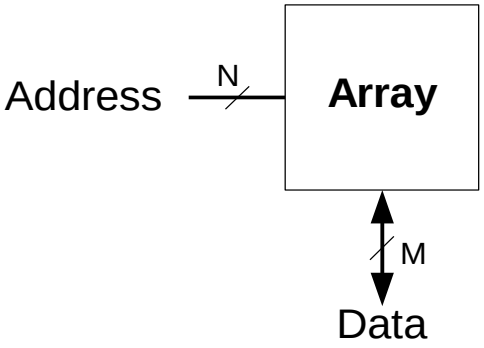
存储阵列

- 高效存储大规模数据:
  - Dynamic random access memory (***DRAM***)动态随机访问存储器
  - Static random access memory (***SRAM***)静态随机访问存储器
  - Read only memory (***ROM***)只读存储器
- ***M***-bit data value read/ written at each unique ***N***-bit address
- 表示2^N个地址（深度），M位的字

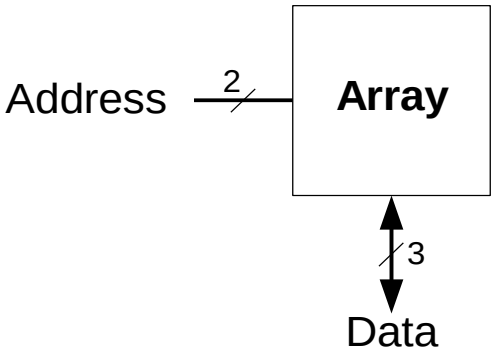


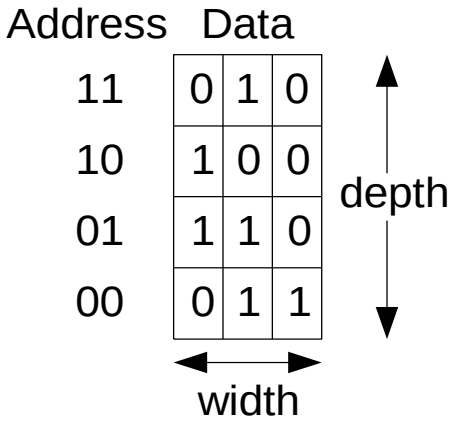
存储阵列

- 二维存储器单元阵列构成
- 每一个位单元 存储1bit数据
- ***N*** address bits and ***M*** data bits:
  - 2^N rows and M columns
  - **Depth:** number of rows (number of words)
  - **Width:** number of columns (size of word)
  - **Array size:** depth × width = $2^N \times M$

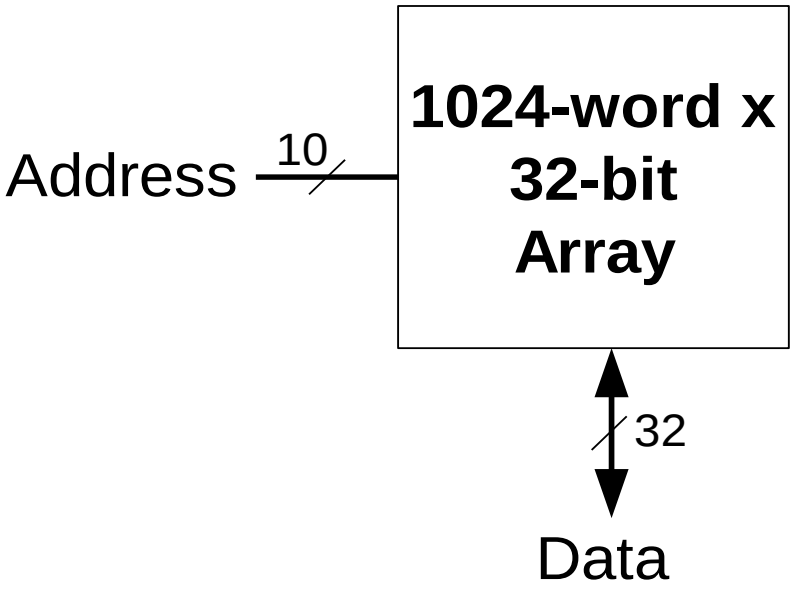


- 比如：
- $2^2 \times 3$-bit array
- Number of words: 4
- Word size: 3-bits



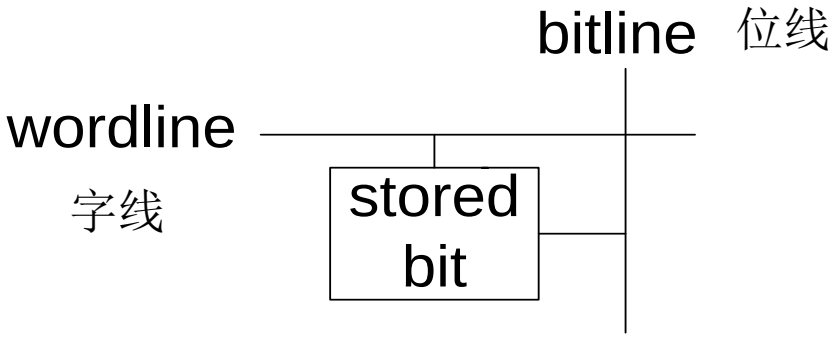


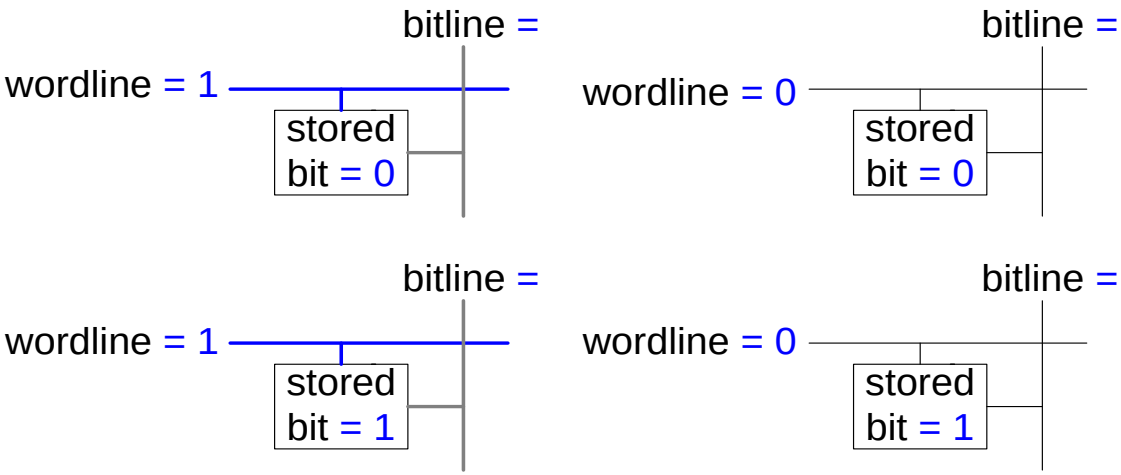
存储阵列举例



1024字x32位阵列的符号，此阵列总大小：32Kb

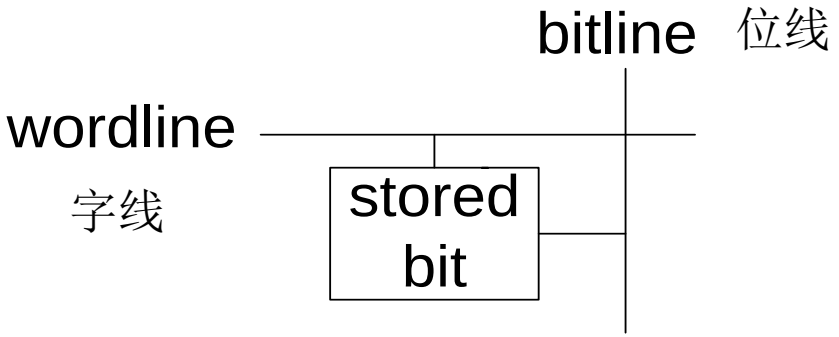
存储阵列的位单元


bitline
位线
wordline
字线
stored
bit

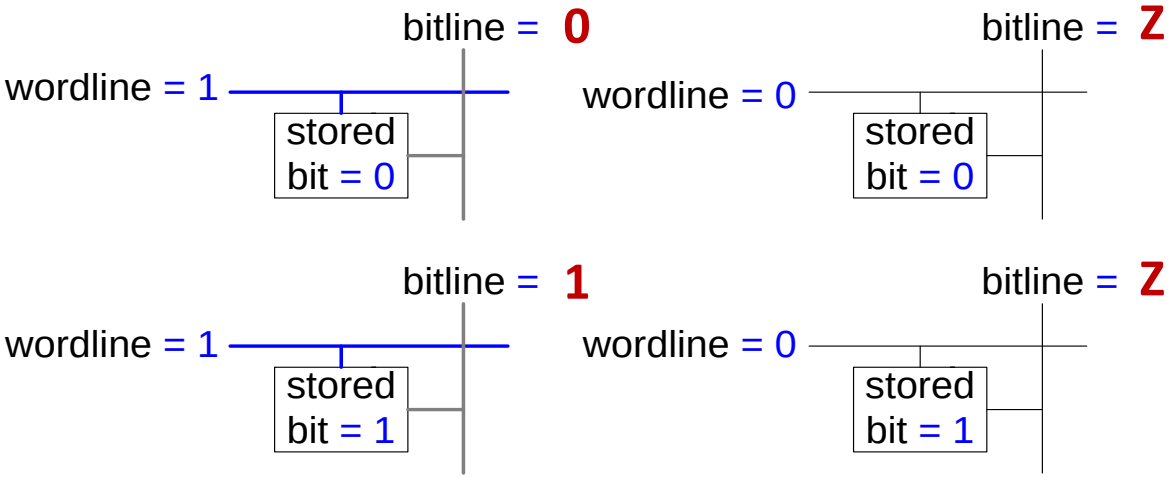

bitline =
wordline = 1
stored
bit = 0
bitline =
wordline = 0
stored
bit = 0
bitline =
wordline = 1
stored
bit = 1
bitline =
wordline = 0
stored
bit = 1

(a) (b)

存储阵列的位单元



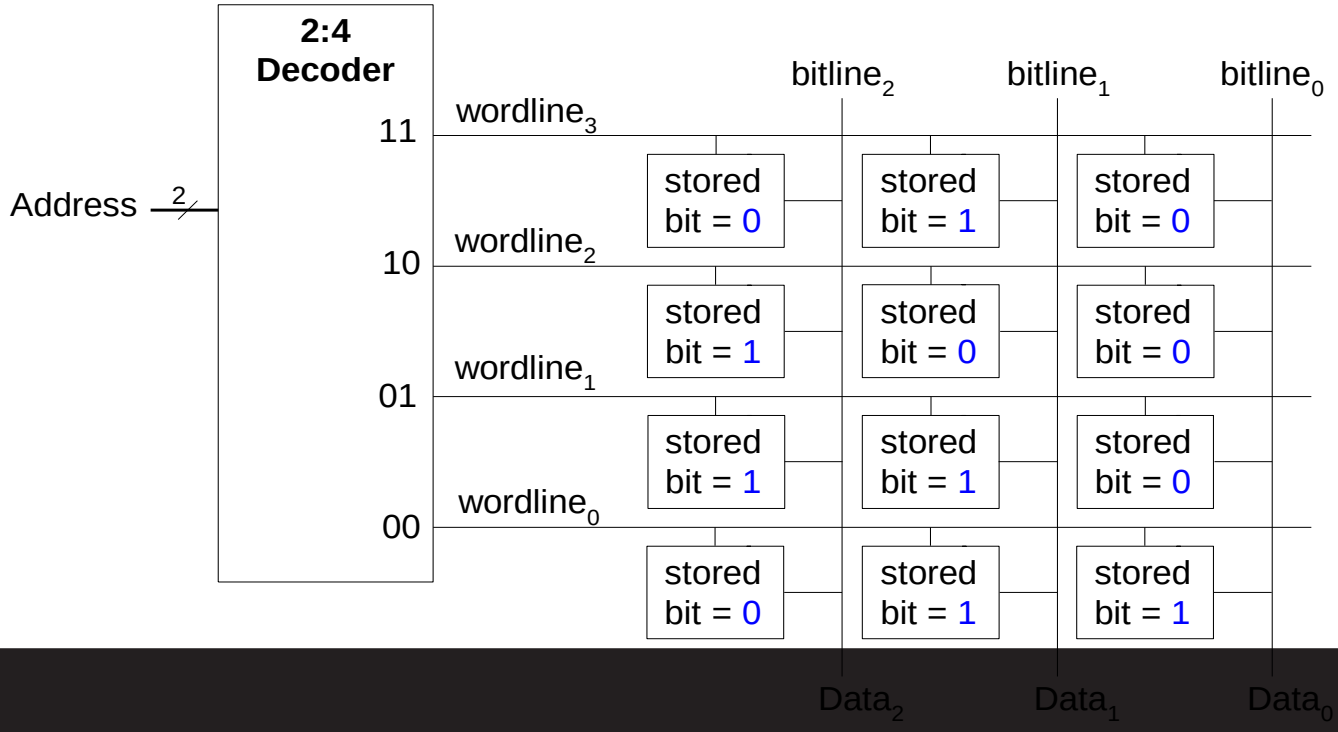
字线为1，则位线可输入或输出值 读取位单元，位线需初始化为浮空Z



(a) (b)

存储阵列

- **Wordline:**
  - 像一个使能信号
  - 单行memeory的读写
  - 对应统一地址
  - 每次只有一个字线位高



存储的类型

- Random access memory (RAM): **volatile** 易失
- Read only memory (ROM): **nonvolatile** 非易失

RAM: Random Access Memory


- **Volatile:** loses its data when power off
- Read and written quickly
- Main memory in your computer is RAM (DRAM)

ROM: Read Only Memory

- **Nonvolatile:** retains data when power off
- Read quickly, but **writing is impossible or slow**
- **Flash memory** in cameras, thumb drives （USB）, and digital cameras are all ROMs

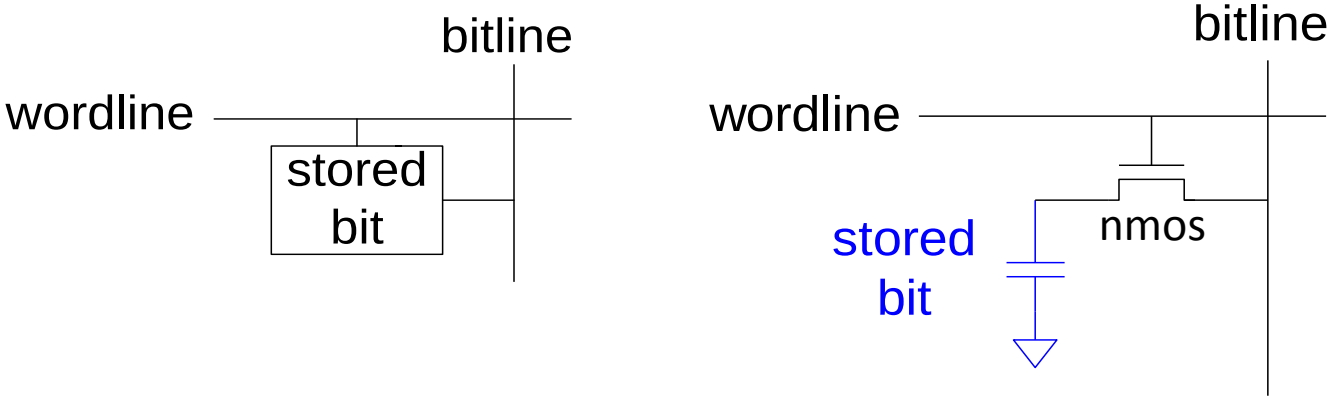
Historically called *read only* memory because ROMs were written at manufacturing time or by burning fuses. Once ROM was configured, it could not be written again. This is no longer the case for Flash memory and other types of ROMs.

RAM类型

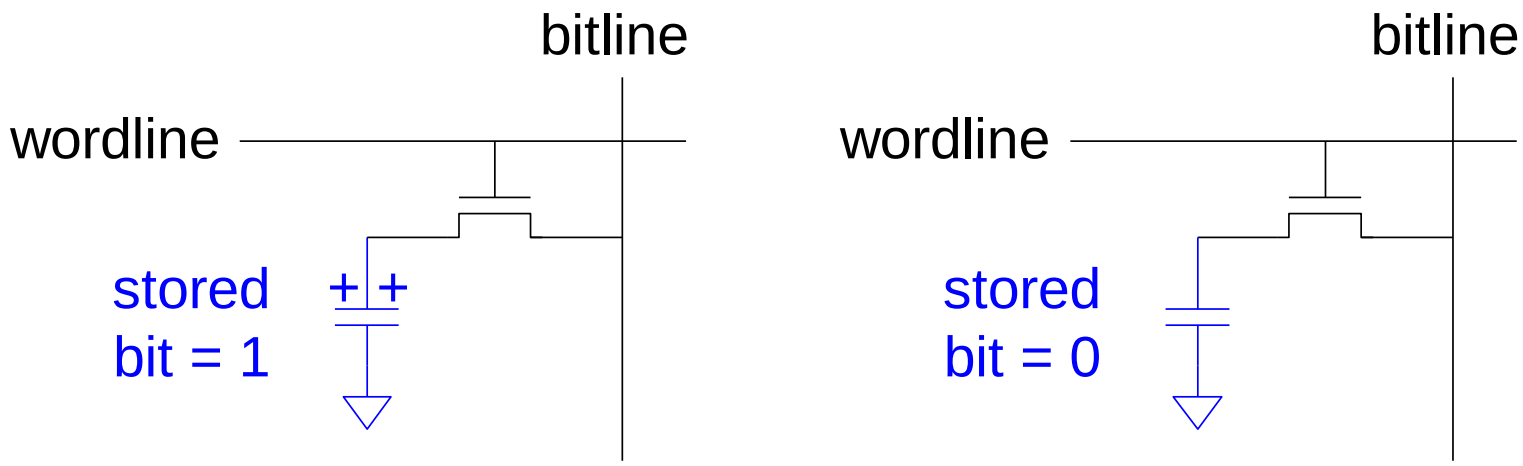
- **DRAM** (Dynamic random access memory)
- **SRAM** (Static random access memory)
- Differ in how they store data:
  - DRAM uses a capacitor 电容
  - SRAM uses cross-coupled inverters 交叉耦合反相器

DRAM

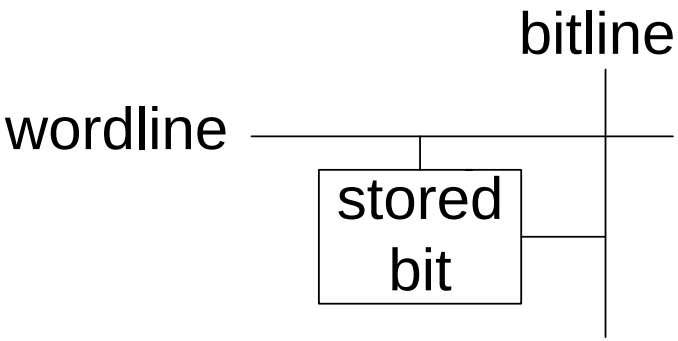
- Data bits stored on capacitor
- *Dynamic* because the value needs to be refreshed (rewritten) periodically and after read:
  - Charge leakage from the capacitor degrades the value
  - Reading destroys the stored value,因此需要周期性刷新

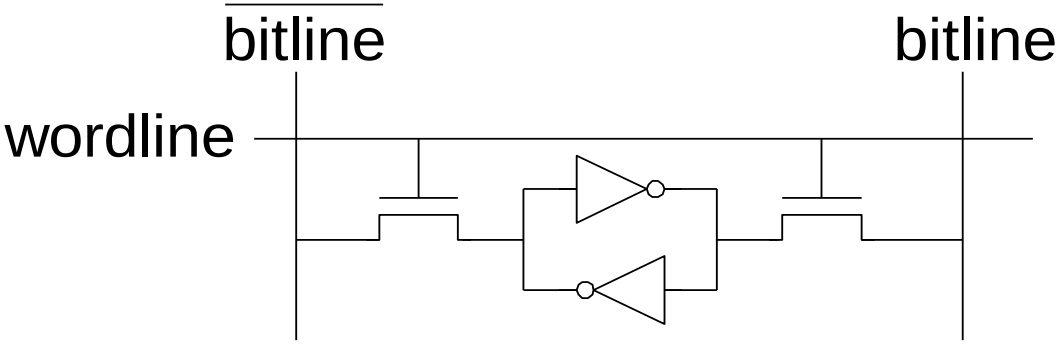


DRAM

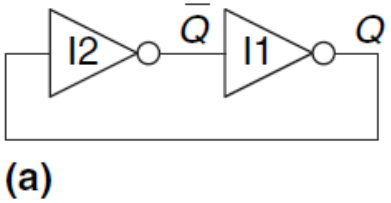

bitline
wordline
stored
bit = 1
+ +
bitline
wordline
stored
bit = 0

SRAM

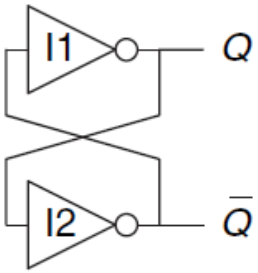




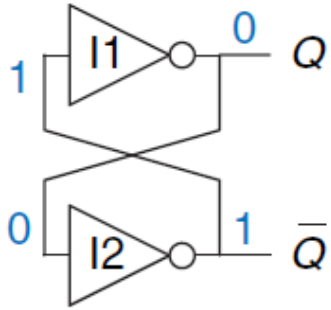
SRAM


I2
$\bar{Q}$
I1
Q

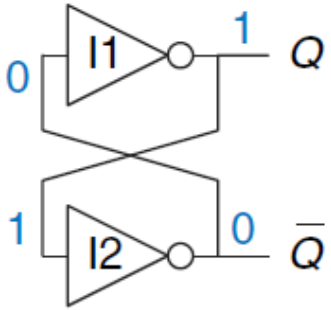
(a)


I1
Q
I2
$\bar{Q}$

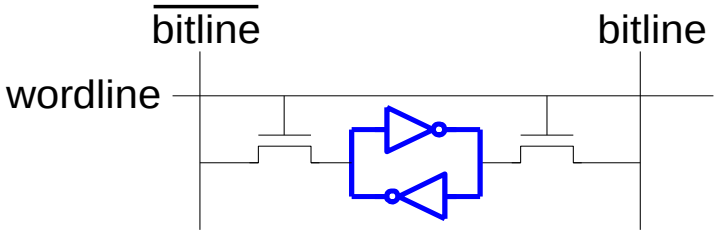
(b)


1
I1
0
Q
0
I2
1
$\bar{Q}$

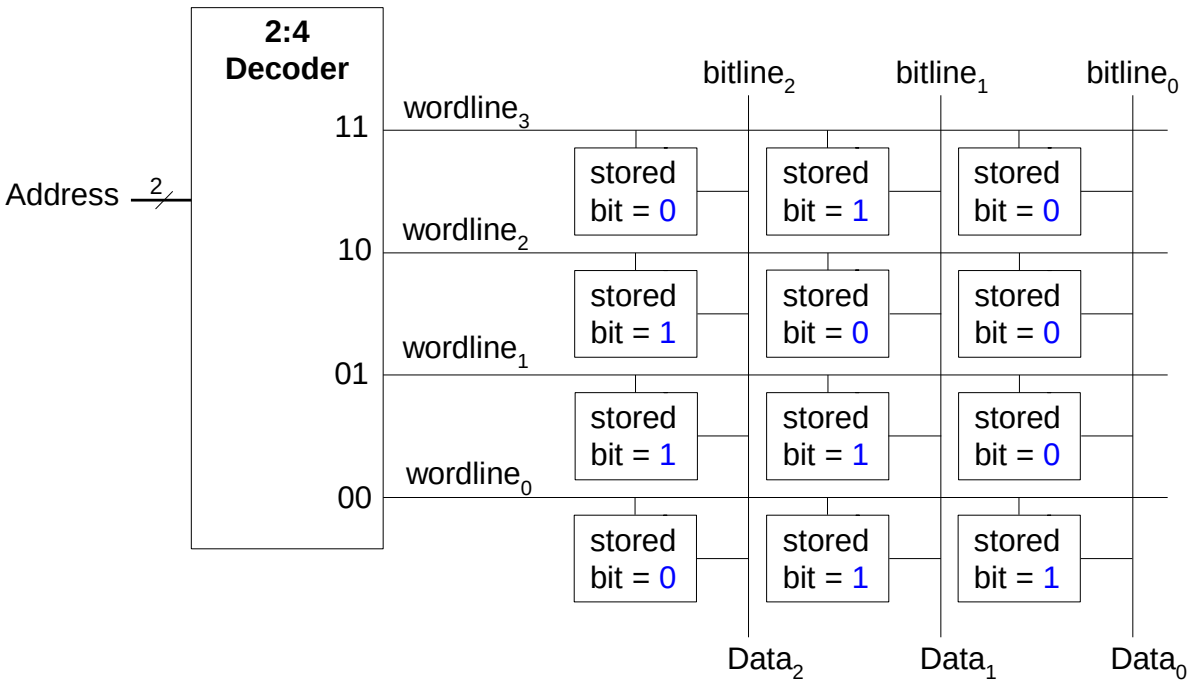
(a)


0
I1
1
Q
1
I2
0
$\bar{Q}$

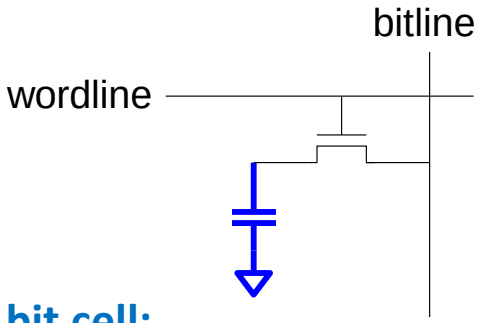
(b)


bitline
bitline
wordline

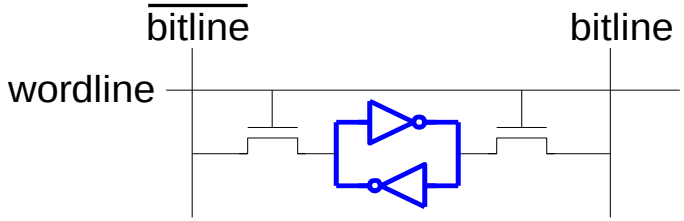
Memory Arrays Review



DRAM bit cell:



SRAM bit cell:

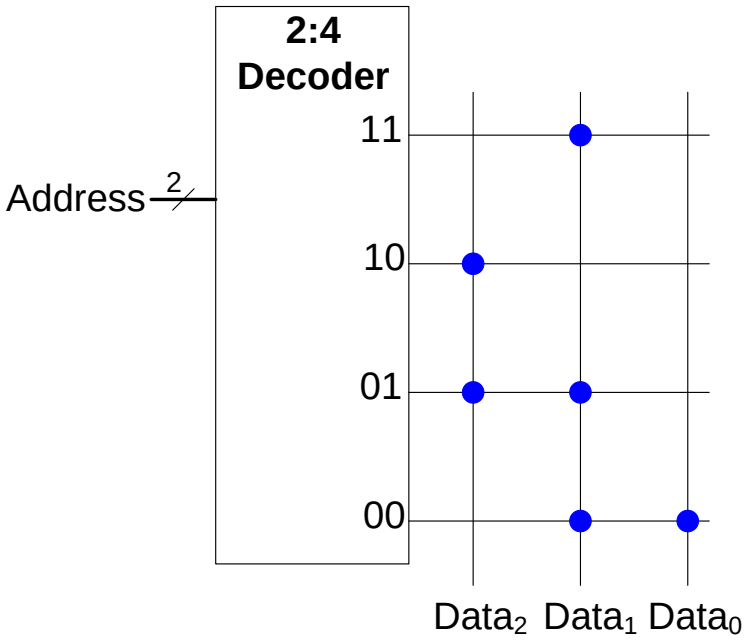


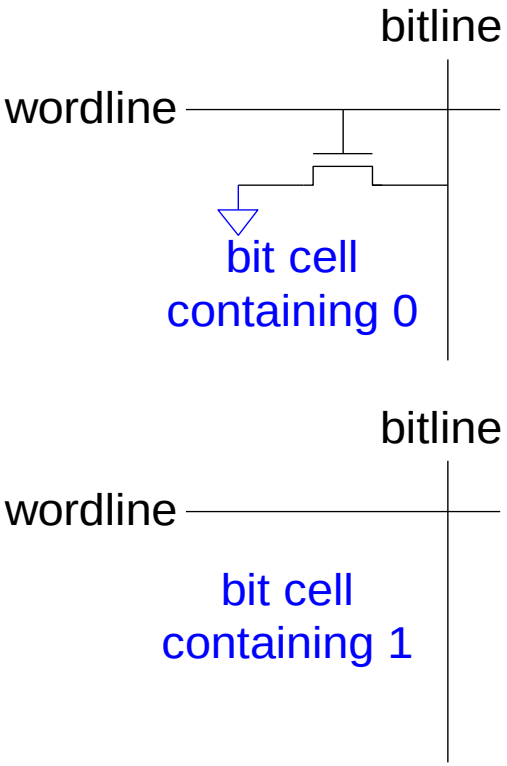
Memory Type	Transistors per Bit Cell	Latency
flip-flop	~20	fast
SRAM	6	medium
DRAM	1	slow

寄存器文件

数字系统中临时变量的存储通常由一组寄存器来存储
这组寄存器称为寄存器文件。
构成方式：小型多端口SRAM阵列或触发器
通常由小型多端口SRAM阵列组成----比触发器阵列更紧凑

ROM设计逻辑

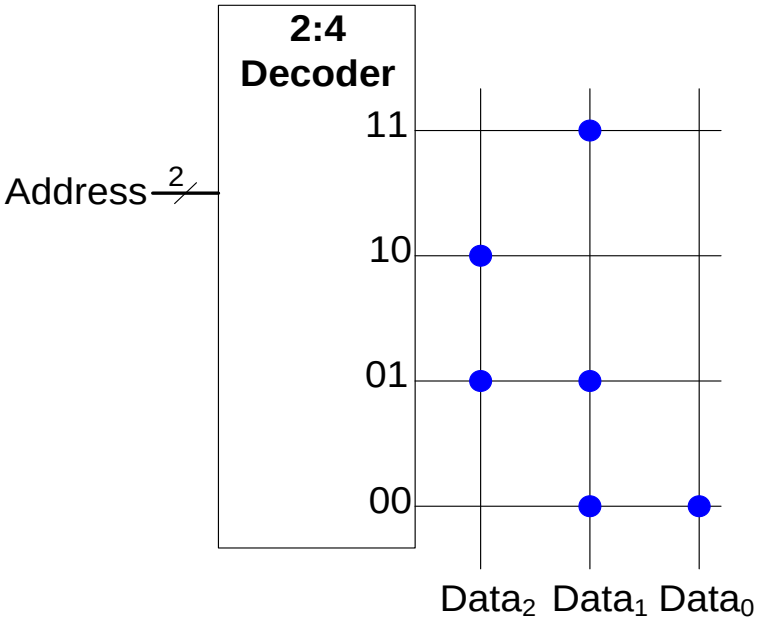

2:4
Decoder
Address
2
11
10
01
00
Data2 Data1 Data0


bitline
wordline
bit cell
containing 0
bitline
wordline
bit cell
containing 1

ROM 的类型

Type	Name	Description
ROM	Read Only Memory	Chip is hardwired with presence or absence of transistors. Changing requires building a new chip.
PROM	Programmable ROM	Fuses in series with each transistor are blown to program bits. Can't be changed after programming.
EPROM	Erasable Programmable ROM	Charge is stored on a floating gate to activate or deactivate transistor. Erasing requires exposure to UV light.
EEPROM	Electrically Erasable Programmable ROM	Like EPROM, but erasing can be done electrically.
Flash	Flash Memory	Like EEPROM, but erasing is done on large blocks to amortize cost of erase circuit. Low cost per bit, dominates nonvolatile storage today.

ROM 存储

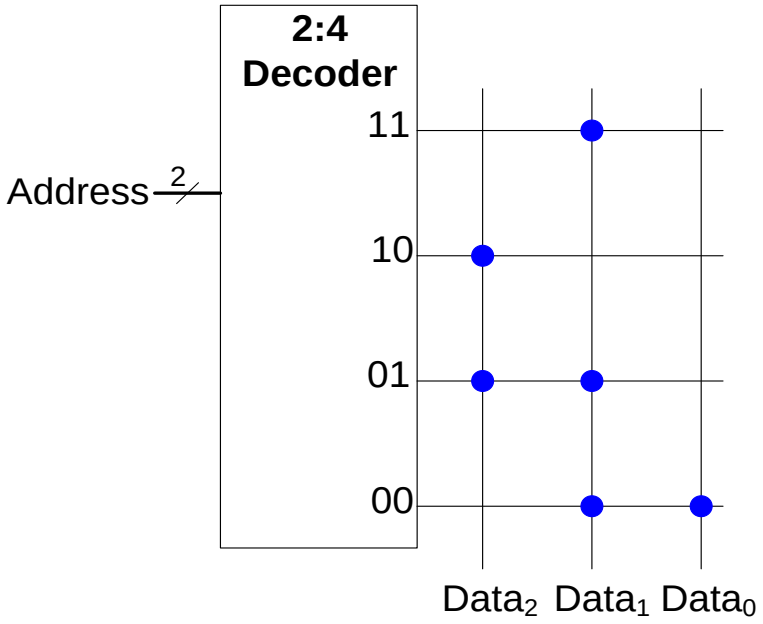


Address	Data		
11	0	1	0
10	1	0	0
01	1	1	0
00	0	1	1

depth

width

ROM 逻辑



$$Data_2 = A_1 \wedge A_0$$

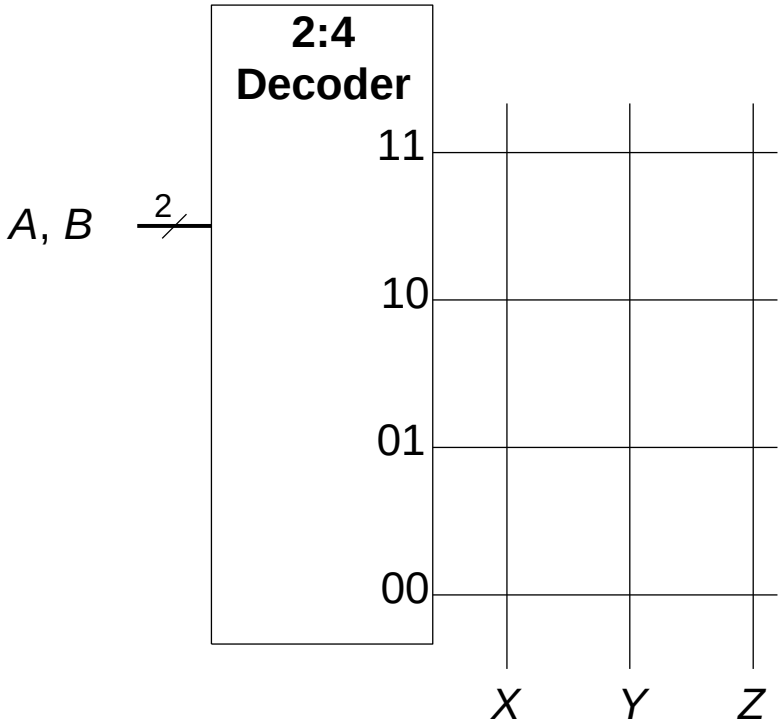
$$Data_1 = \overline{A_1} + A_0$$

$$Data_0 = \overline{A_1}\,\overline{A_0}$$

使用存储器做逻辑运算

Implement the following logic functions using a $2^2 \times 3$-bit ROM:

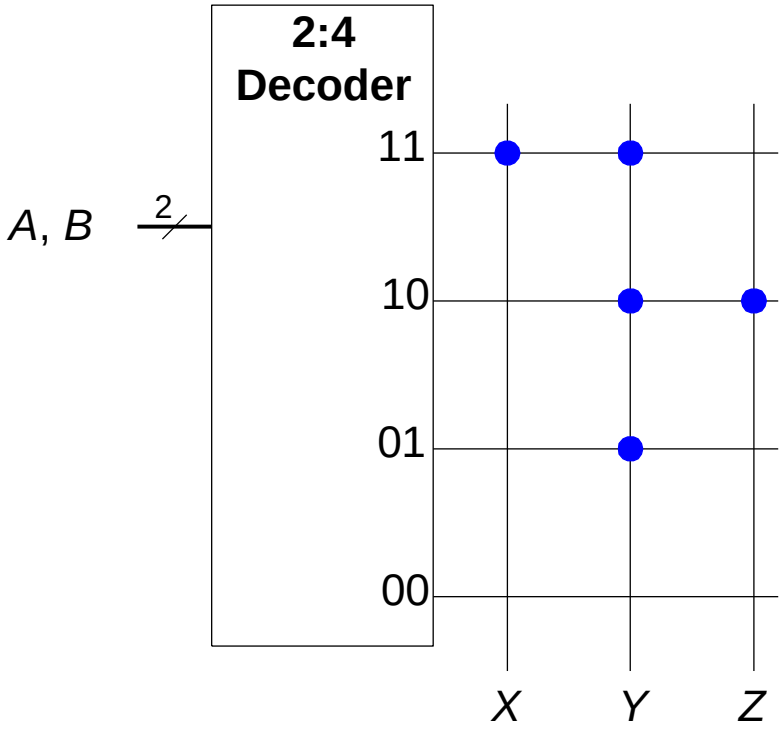
- $X = AB$
- $Y = A + B$
- $Z = A\overline{B}$



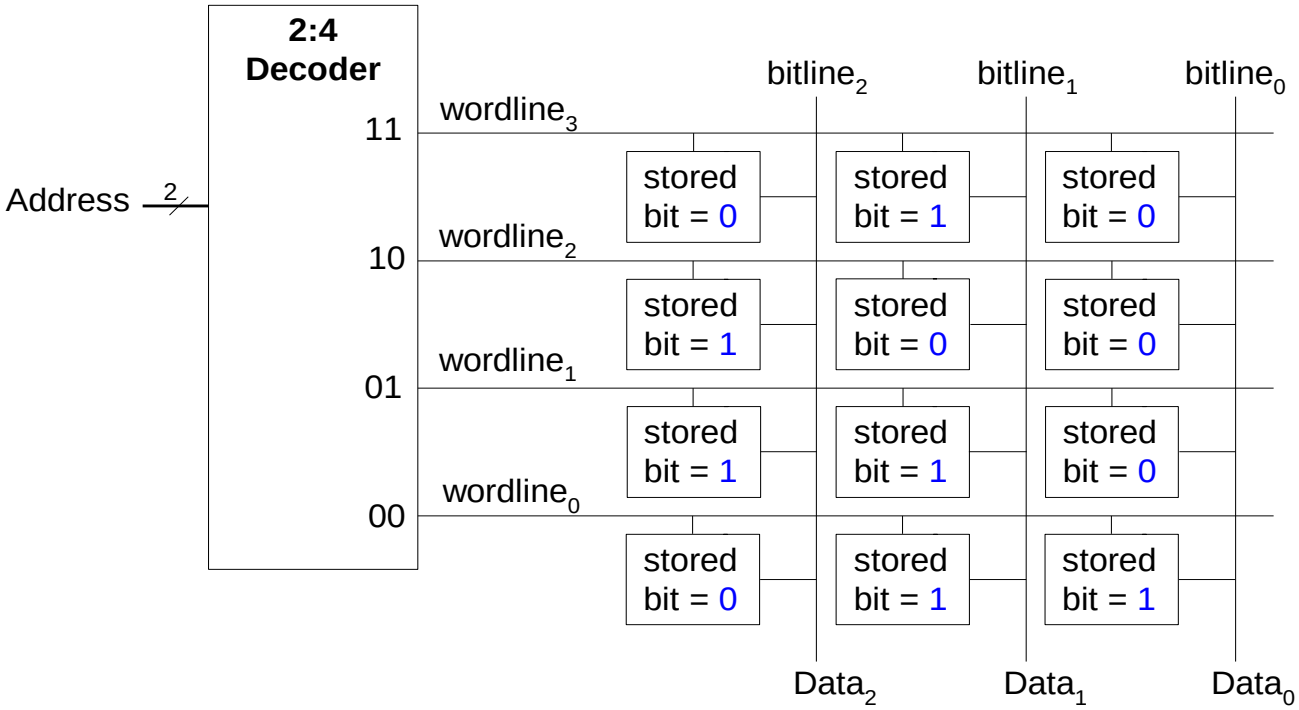
使用存储器做逻辑运算

Implement the following logic functions using a $2^2 \times 3$-bit ROM:

- $X = AB$
- $Y = A + B$
- $Z = A\overline{B}$



使用存储器做逻辑运算



$$Data_2 = A_1 \oplus A_0$$

$$Data_1 = \overline{A}_1 + A_0$$

$$Data_0 = \overline{A}_1 \overline{A}_0$$

使用存储器做逻辑运算

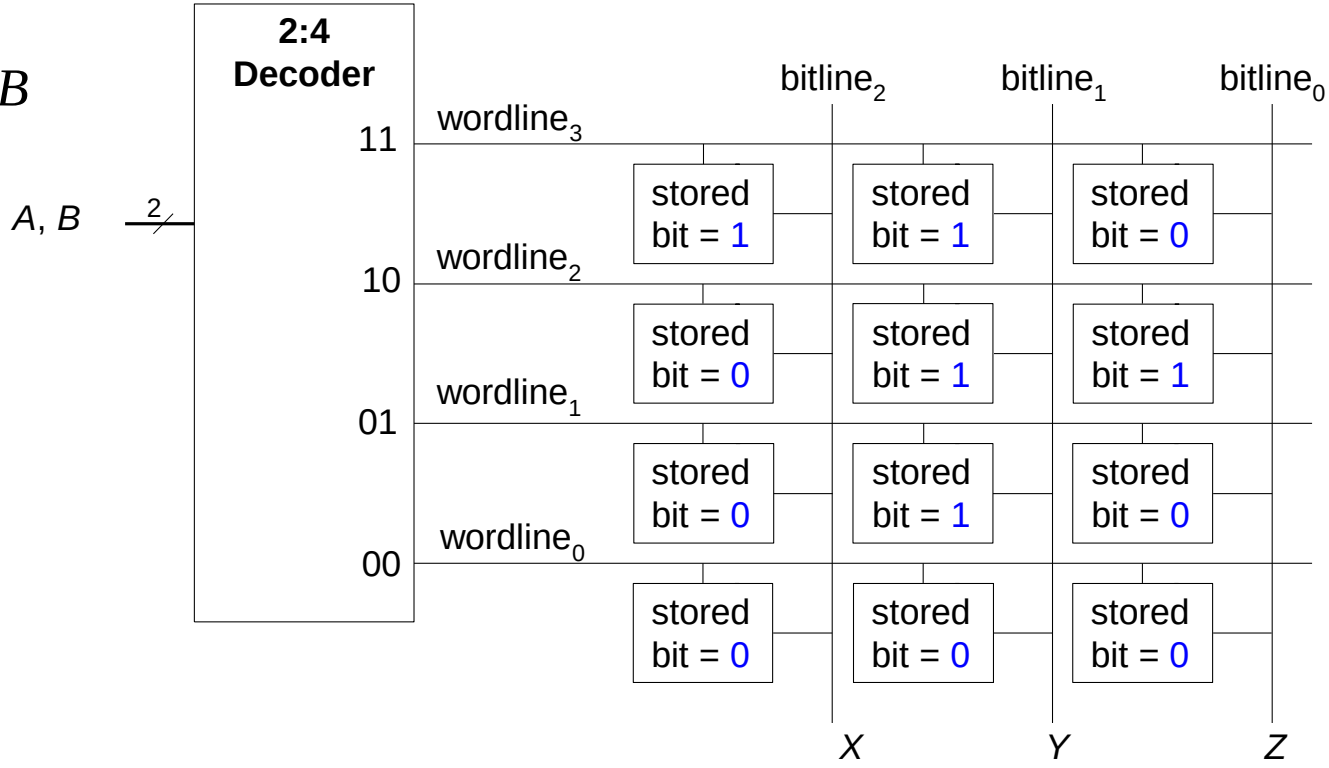
Implement the following logic functions using a $2^2 \times 3$-bit memory array:

- $X = AB$
- $Y = A + B$
- $Z = A\overline{B}$

使用存储器做逻辑运算

Implement the following logic functions using a $2^2 \times 3$-bit memory array:

- $X = AB$
- $Y = A + B$
- $Z = A\overline{B}$

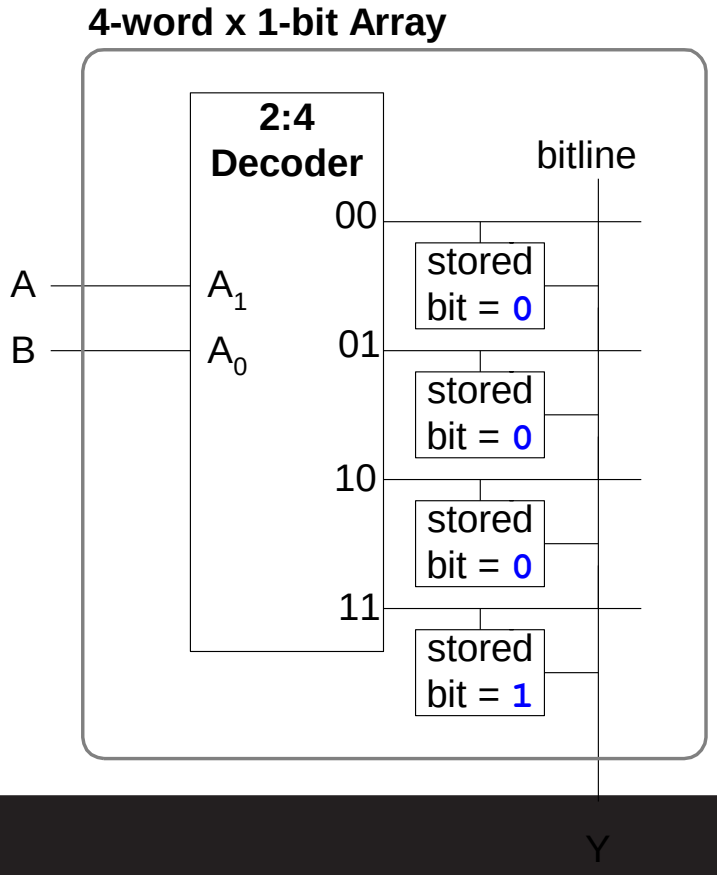


使用存储器做逻辑运算

Called *lookup tables* (查找表LUTs): look up output at each input combination (address)

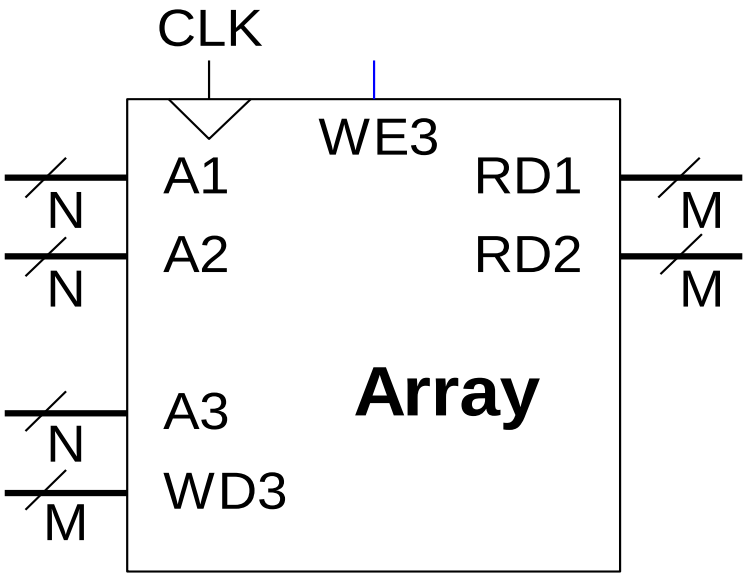
Truth Table

A	B	Y
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**



多端口存储

- **Port:** address/data pair
- 3-ported memory
  - 2 read ports (A1/RD1, A2/RD2)
  - 1 write port (A3/WD3, WE3 enables writing)
- **Register file:** small multi-ported memory



SystemVerilog Memory Arrays

```
// 256 x 64 memory module with one read/write port
module dmem(input  logic        clk, we,
            input  logic [7:0]  a,
            input  logic [63:0] wd,
            output logic [63:0] rd);

  logic  [63:0] RAM[255:0];

  always @(posedge clk)
    begin
      rd <= RAM[a];    // synchronous read
      if (we)
        RAM[a] <= wd; // synchronous write
    end
endmodule
```

写Testbench测试

SystemVerilog Register File

```
// 16 x 32 register file with two read, 1 write port
module rf(input  logic        clk, we3,
          input  logic [3:0] a1, a2, a3,
          input  logic [31:0] wd3,
          output logic [31:0] rd1, rd2);

  logic  [31:0] RAM[15:0];

  always @(posedge clk)  // synchronous write
    if (we3)
        RAM[a3] <= wd3;
  assign rd1 = RAM[a1];  // asynchronous read
  assign rd2 = RAM[a2];
endmodule
```

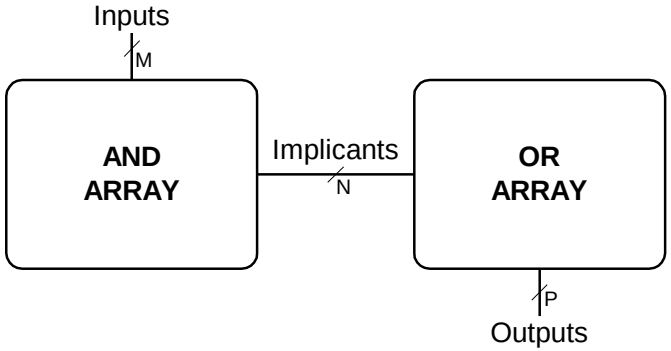
写Testbench测试

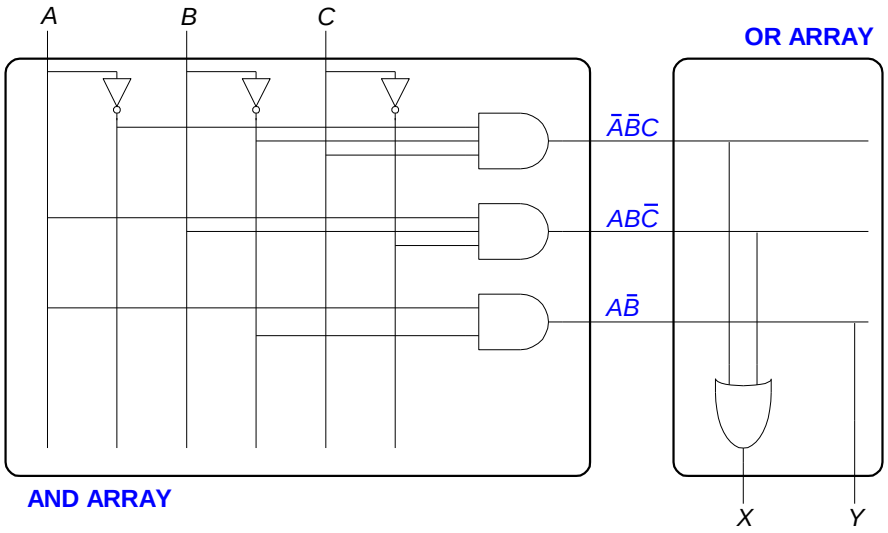
逻辑阵列

- **PLAs** (可编程逻辑阵列)
  - 与门阵列+或门阵列
  - 只能实现组合逻辑
  - 内部连接固定
- **FPGAs** (现场可编程逻辑门阵列)
  - 逻辑单元（Logic Elements (LEs)）阵列
  - 可实现组合逻辑与时序逻辑
  - 内部连接可编程

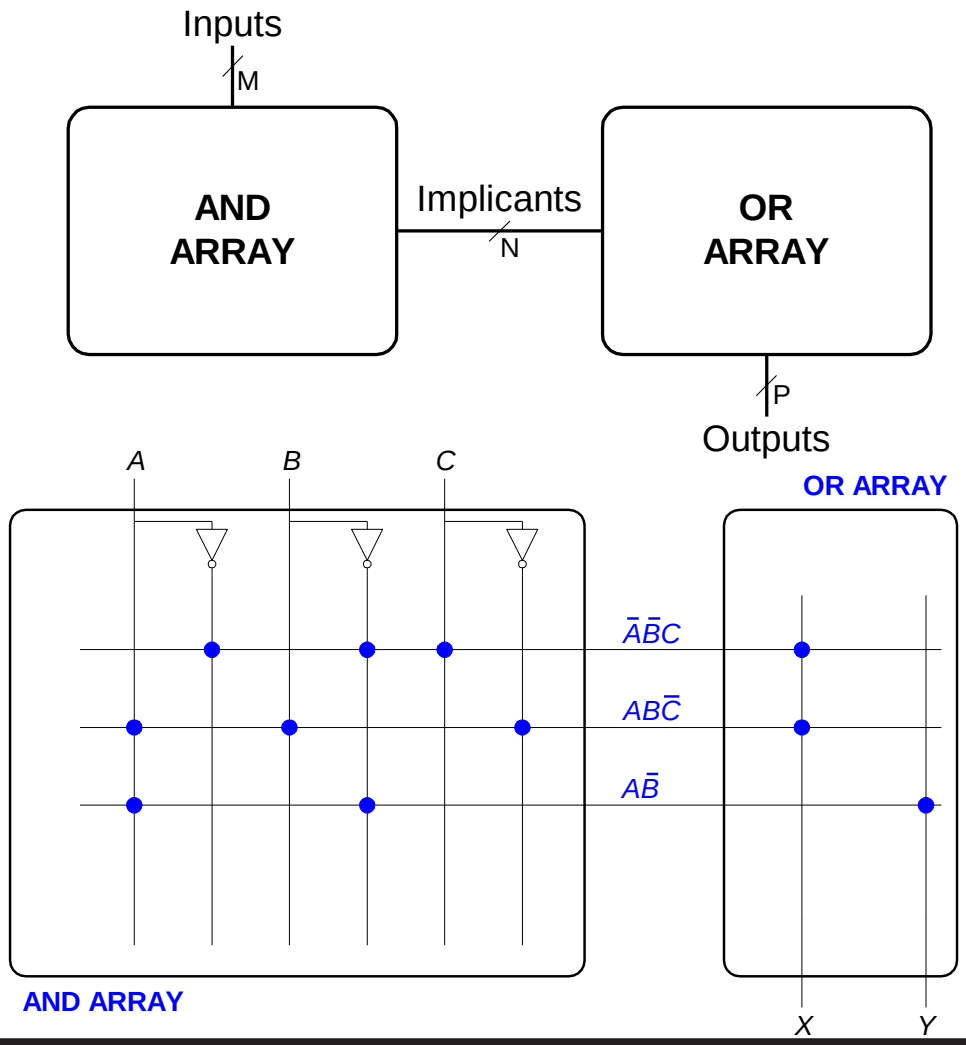
PLAs

- $X = \overline{A}\,\overline{B}C + AB\overline{C}$
- $Y = A\overline{B}$





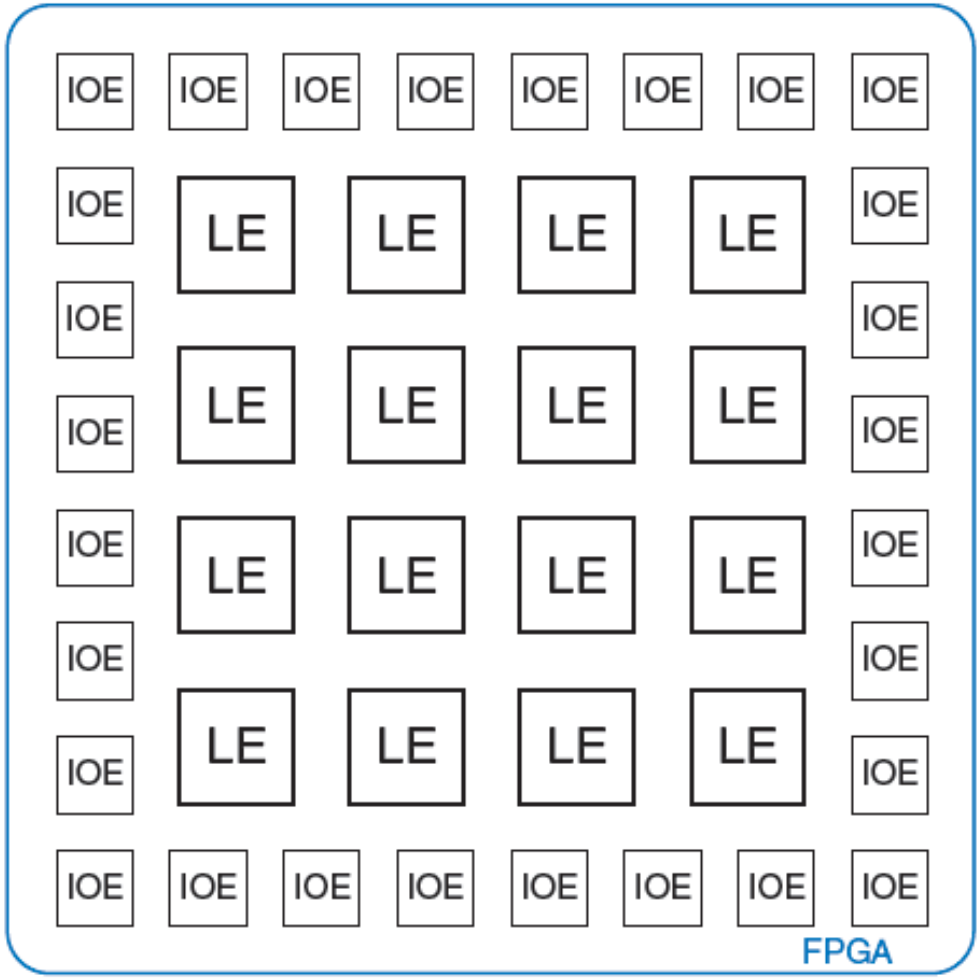
PLAs: Dot Notation点号表示法



FPGA: Field Programmable Gate Array

- Composed of:
  - **LEs** (Logic elements)逻辑单元: perform logic
  - **IOEs** (Input/output elements)输入/输出接口: interface with outside world
  - **Programmable interconnection可编程布线通道:** connect LEs and IOEs
  - Some FPGAs include other building blocks such as multipliers and RAMs （IP,Intellectual Property）

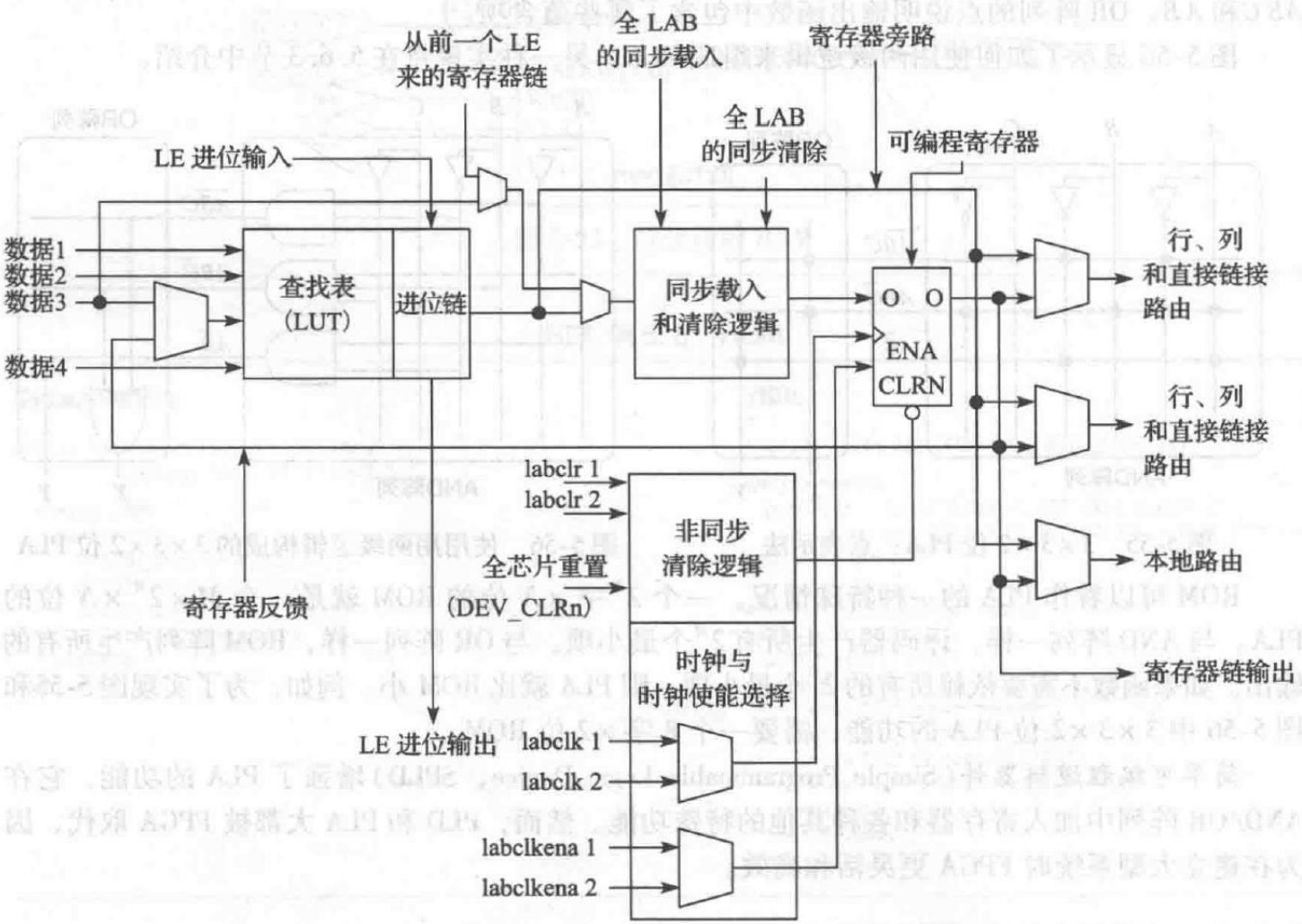
General FPGA Layout



LE: Logic Element

- Composed of:
  - **LUTs** (lookup tables): 执行组合逻辑
  - **Flip-flops:** 执行时序逻辑
  - **Multiplexers:** 连接LUTs and flip-flops

Altera Cyclone IV LE



Altera Cyclone IV LE

- The Altera Cyclone IV LE has:
  - 1 four-input LUT
  - 1 registered output
  - 1 combinational output

LE Configuration Example

Show how to configure a Cyclone IV LE to perform the following functions:

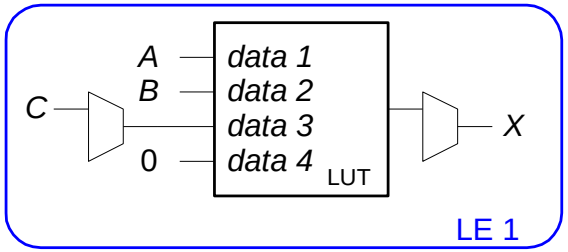
- $X = \overline{A}\,\overline{B}C + AB\overline{C}$
- $Y = A\overline{B}$

LE Configuration Example

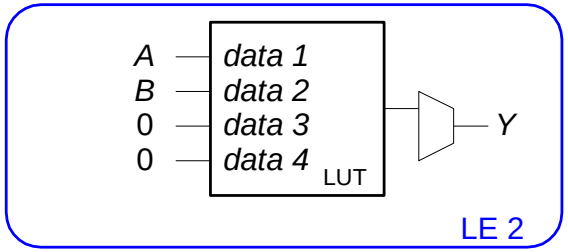
Show how to configure a Cyclone IV LE to perform the following functions:

- $X = \overline{A}\,\overline{B}C + AB\overline{C}$
- $Y = A\overline{B}$

(A) data 1	(B) data 2	(C) data 3	data 4	(X) LUT output
0	0	0	X	0
0	0	1	X	1
0	1	0	X	0
0	1	1	X	0
1	0	0	X	0
1	0	1	X	0
1	1	0	X	1
1	1	1	X	0



(A) data 1	(B) data 2	data 3	data 4	(Y) LUT output
0	0	X	X	0
0	1	X	X	0
1	0	X	X	1
1	1	X	X	0



FPGA Design Flow

Using a CAD tool (such as Altera's Quartus II)

- **Enter the design** with a HDL
- **Simulate** the design
- **Synthesize** design and map it onto FPGA
- **Download the configuration** onto the FPGA
- **Test** the design

This is an iterative process!

本章结束

练习

以下 3 种 64 位加法器的延迟是多少？假设每一个 2 输入门的延迟是 150ps，全加器的延迟是 450ps。

(a) 行波进位加法器

(b) 4 位单元的先行进位加法器

(c) 前缀加法器

$$t_{\text{ripple}} = Nt_{\text{FA}} = 64(450\ \text{ps}) = 28.8\ \text{ns}$$

$$t_{CLA} = t_{\text{pg}} + t_{\text{pg\_block}} + \left(\frac{N}{k} - 1\right)t_{\text{AND\_OR}} + kt_{\text{FA}} \qquad 150$$

$$t_{CLA} = \left[150 + (6 \times 150) + \left(\frac{64}{4} - 1\right)300 + (4 \times 450)\right] = 7.35\ \text{ns}$$

$$t_{PA} = t_{\text{pg}} + \log_2 N(t_{\text{pg\_prefix}}) + t_{\text{XOR}}$$

$$t_{PA} = [150 + 6(300) + 150] = 2.1\ \text{ns}$$

以下数字系统的数据表示区间分别是多少：

(a) 24 位无符号定点数，其中包含 12 位整数位和 12 位小数位。

(b) 24 位带符号的原码定点数，其中包含 12 位整数位和 12 位小数位。

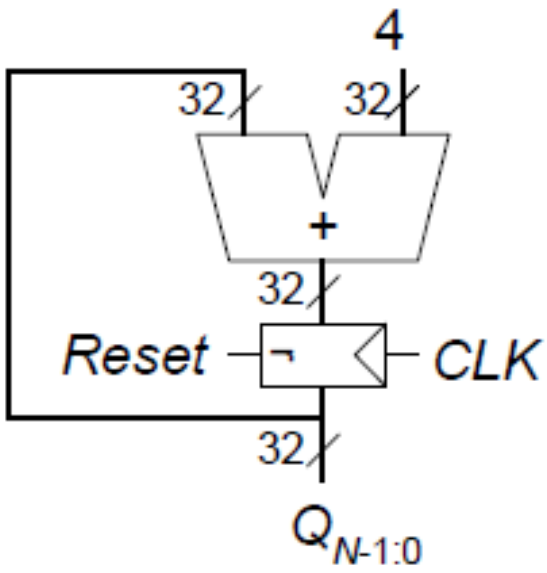
(c) 24 位 2 进制补码定点数，其中包含 12 位整数位和 12 位小数位。

(a) $\left[0, \left(2^{12} - 1 + \frac{2^{12} - 1}{2^{12}}\right)\right]$

(b) $\left[-\left(2^{11} - 1 + \frac{2^{12} - 1}{2^{12}}\right), \left(2^{11} - 1 + \frac{2^{12} - 1}{2^{12}}\right)\right]$

(c) $\left[-\left(2^{11} + \frac{2^{12} - 1}{2^{12}}\right), \left(2^{11} - 1 + \frac{2^{12} - 1}{2^{12}}\right)\right]$

设计一个 32 位计数器，在每一个时钟沿加 4。计数器输入信号包括：复位和时钟。当复位时，所有输出为 0。

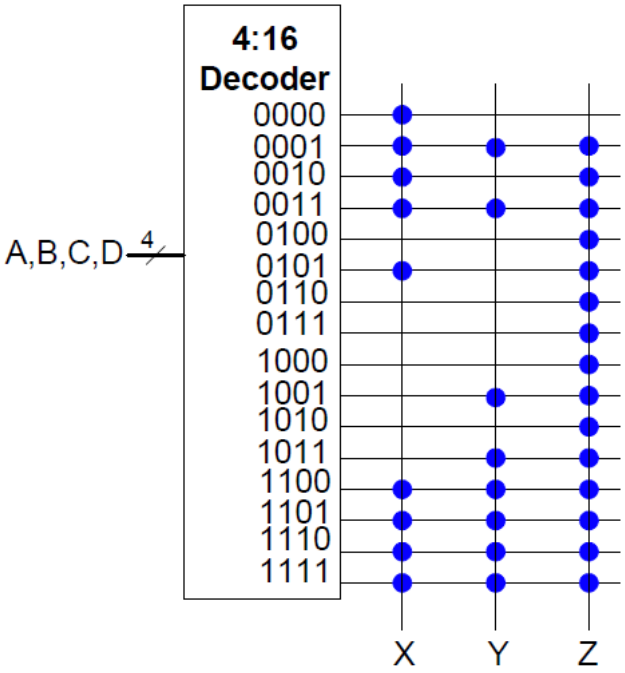


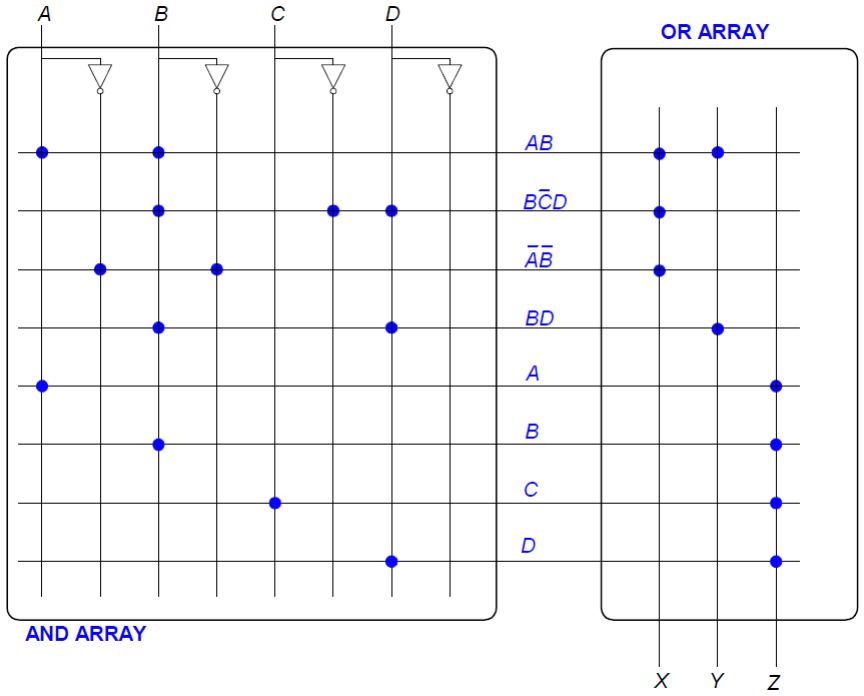
使用一个单独的 16 × 3 ROM 实现以下的函数。使用点表示法说明 ROM 的内容。　　使用 4 × 8 × 3 PLA 实现

(a) $X = AB + B\overline{C}D + \overline{A}\overline{B}$

(b) $Y = AB + BD$

(c) $Z = A + B + C + D$





* 参考学习知识点

回顾

一位全加器： $S = A \oplus B \oplus C_{in}$

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

$C_{out} = A\&B + (A+B)\&C_{in}$

行波进位加法器：串联。

先行进位加法器：

1：把加法器分成若干块 （块内加分是行波进位加法器），同时增加电路（增加的电路计算进位）。

2：当每一块有进位时就快速确定此块的输出进位 （优点不需要等待块内行波加法器内所有加法器的延迟，而是直接先行经过该块）

3：G 和 P 来描述一列或一块如何进位输出

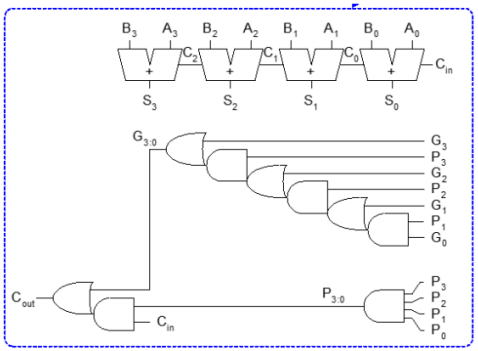
4：一列： i 列产生进位：$G_i = A_i\&B_i$ i 列传播进位：$P_i = A_i+B_i$

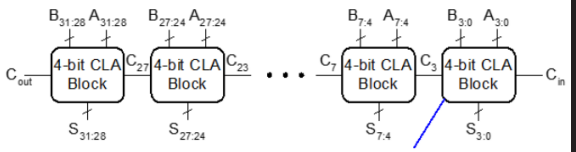
i 列进位：$C_i = G_i + P_i\& C_{i-1} = A_iB_i + (A_i+B_i)\& C_{i-1}$

5：一块： 块索引 i:j 列产生进位：$G_{i:j} = G_i + P_i \& G_{i-1}$ 块索引 i:j 列传播进位：$P_{i:j} = P_i \& P_{i-1} \& P_{i-2} \ldots \& P_j$

块索引 i:j 列进位：$C_i = C_{i:j} = G_{i:j} + P_{i:j} C_j$

- **Step 1:** Compute G_i and P_i for all columns
- **Step 2:** Compute G and P for k-bit blocks
- **Step 3:** C_{in} propagates through each k-bit propagate/generate logic (**meanwhile computing sums** 串行的行波加法器)
- **Step 4:** Compute sum for most significant k-bit block





回顾

前缀加法器：

1：扩展了先行进位加法器的产生和传播逻辑。首先以两列一组计算 G 和 P，之后是 4 位（列）、8 位（列）、16 位（列）的块，直到产生每一列的生成信号，然后从这些生成信号中计算和。

2：尽可能快地计算每一列 i 的进位输入 C_{i-1}，然后使用 计算求和：

$$S_i = (A_i \oplus B_i) \oplus C_{i-1}$$

3：定义 i=-1，代表 C_{in}，$G_{-1} = C_{in}$ 且 $P_{-1} = 0$，因为 如果从列-1 到 i-1 的块产生一个进位，那么列 i-1 将产生进位输出，所以 Ci-1=Gi-1：-1.产生的进位要么在 i-1 列中产生，要么在前一列中产生并传播，因此

$$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}$$

4： 由于 A_i 及 B_i 已知，则快速计算的主要任务就是求出 $G_{i-1:-1}$，即：$G_{-1:-1}$，$G_{0:-1}$，$G_{1:-1}$, $G_{2:-2}$, $G_{3:-1}$ ..... $G_{N-2:-1}$ 及 $P_{-1:-1}$，$P_{0:-1}$，$P_{1:-1}$, $P_{2:-2}$, $P_{3:-1}$.....$P_{N-2:-1}$

$$G_{i:j} = G_{i:k} + P_{i:k} G_{k-1:j}$$

$$P_{i:j} = P_{i:k} P_{k-1:j}$$

5: 和先行进位相似的地方，但又起着决定性不同的地方！！！