FDU 计算机组成与体系结构 4. 数字模块

本文参考以下教材：

- Digital Design and Computer Architecture (D. M. Harris, S. L. Harris 2rd) Chapter 5
- 数字设计和计算机体系结构 (D. M. Harris, S. L. Harris 2rd 陈俊颖译) 第 5 章
- 计算机组成与系统结构 (袁春风、唐杰、杨若瑜、李俊) 第 3 章

欢迎批评指正！

4.1 算术电路 (Arithmetic Circuits)

计算机和数字逻辑 (digital logic) 可以实现很多算术功能：加法、减法、比较、移位、乘法、除法

4.1.1 加法 (Addition)

(1) 半加器 (Half Adder)

考虑构建 1 位半加器：

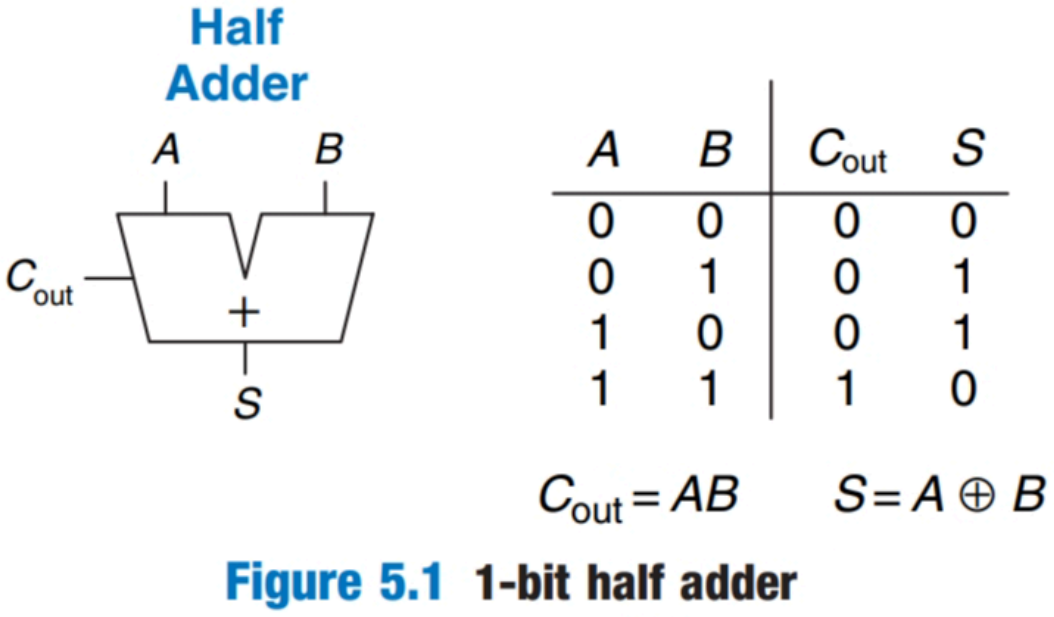
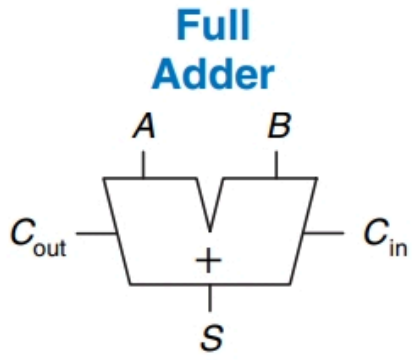


Figure 5.1 1-bit half adder

半加器可以用一个 XOR 门电路和一个 AND 门电路实现.
它有两个输入 A, B 和两个输出 $\begin{cases} S = A \oplus B = (A + B) \bmod 2 \\ C_{\text{out}} = AB \end{cases}$
其中 S 为 A, B 的**模 2 加和** (即异或)，C_{out} 代表**输出进位** (carry out)
在多位加法器中，C_{out} 会进位 (carry in) 到下一个高位.
然而半加器缺少一个输入 C_{in} 来接受前一列的输出进位 C_{out}，下面的全加器将解决这个问题.

(2) 全加器 (Full Adder)

考虑构建 1 位全加器：



$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

C_{in}	A	B	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

Figure 5.3 1-bit full adder

(3) 进位传播加法器 (CPA, Carry Propagate Adder)

一个 N 位加法器将两个 N 位输入 A, B 和一位输入进位 C_{in} 相加，
产生一个 N 位结果 S 和一个输出进位 C_{out}

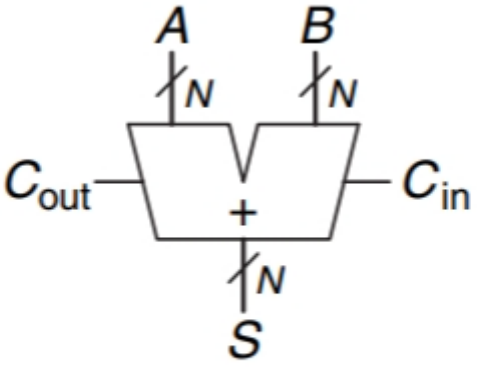


Figure 5.4 Carry propagate adder

最常见的 3 种 CPA 实现分别是串行进位加法器、并行进位加法器和前缀加法器：

- **① 串行进位加法器 (Serial Adder/Ripple-Carry Adder):**
  构造 N 位进位传播加法器的最简单方法就是把 N 个全加器串联起来.
  一级的 C_{out} 就是下一级的 C_{in}
  其缺点是：当 N 比较大时，运算速度会慢下来.
  进位通过进位链形成**行波** (ripple)，即进位信号必须通过加法器的每一位传播.

加法器的延迟会随着位数的增加而增加.

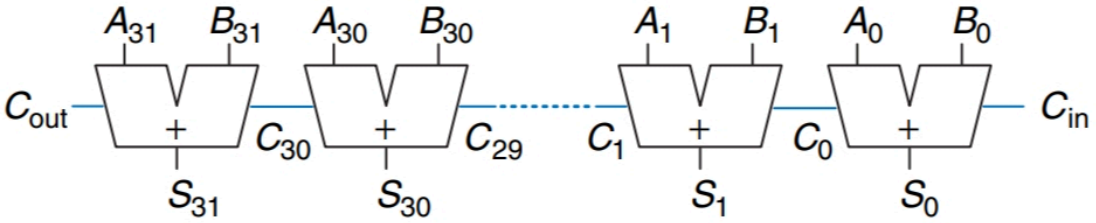


Figure 5.5 32-bit ripple-carry adder

- **② 并行进位加法器 (Parallel Adder/Carry-Lookahead Adder, CLA):**
  它解决进位问题的方法是:
  把加法器分解成若干块，同时增加电路，当每块有进位时就快速确定此块的输出进位.
  因此它不需要等待进位行波通过一块内的所有加法器，而是直接先行通过该块.
  例如，一个 32 位加法器可以分解成 8 个 4 位的块.

  先行进位加法器使用**产生** (G, Generate) 和**传播** (P, Propagate) 两个信号
  来描述一列或者一块如何确定进位输出.

  - 在不考虑进位输入的情况下，
    加法器的第 i 列在 A_i 和 B_i 都为 1 时，必定**产生**进位 C_i，
    因此第 i 列的**产生信号** G_i 可以这样计算 $G_i = A_iB_i$
  - 当有输入进位 C_{i-1} 时，
    如果 A_i 或者 B_i 为 1，则第 i 列将**传播**一个输入进位 C_i
    因此第 i 列的**传播信号** P_i 可以这样计算 $P_i = A_i + B_i$

  利用上述定义，可以为加法器的特定列重写进位逻辑.
  如果加法器的第 i 列将产生一个进位 G_i，或者传播一个输入进位 P_iC_{i-1}
  则它将产生输出进位 C_i，表达式为 $C_i = G_i + P_iC_{i-1} = A_iB_i + (A_i + B_i)C_{i-1}$

  产生和传播的定义可以扩展到多位块:

  - 从第 i 列到第 j 列的**传播进位**的条件是:
    从 i 到 j 的所有列都能传播进位，即 $P_{i:j} = P_iP_{i-1}\dots P_j$
  - 从第 i 列到第 j 列的**产生进位**的条件是:
    最高有效列产生一个进位，或者最高有效列传播进位且前一列产生了进位，即有:
    $$\begin{aligned} G_{i:j} &= G_i + P_iG_{i-1:j} \\ &= G_i + P_i(G_{i-1} + P_{i-1}G_{i-2:j}) \\ &= \cdots \\ &= G_i + P_i(G_{i-1} + P_{i-1}(\dots(G_{j+1} + P_{j+1}G_j))) \end{aligned}$$
  - 使用块的产生信号 $G_{i:j}$ 和传播信号 $P_{i:j}$，
    可以根据块的输入进位 C_j 快速计算块的输出进位 $C_i = G_{i:j} + P_{i:j}C_j$

  以四位块为例:

  - 第 0 列:
    $C_0 = G_0 + P_0C_{\text{in}}$
  - 第 1 列:
    $C_1 = G_1 + P_1G_0 + P_1P_0C_{\text{in}}$
  - 第 2 列:
    $$\begin{aligned} C_2 &= G_2 + P_2(G_1 + P_1G_0) + P_2P_1P_0C_{\text{in}} \\ &= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_{\text{in}} \end{aligned}$$

- 第 3 列:
  $C_3 = G_3 + P_3(G_2 + P_2(G_1 + P_1G_0)) + P_3P_2P_1P_0C_{\text{in}}$
  $= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_{\text{in}}$

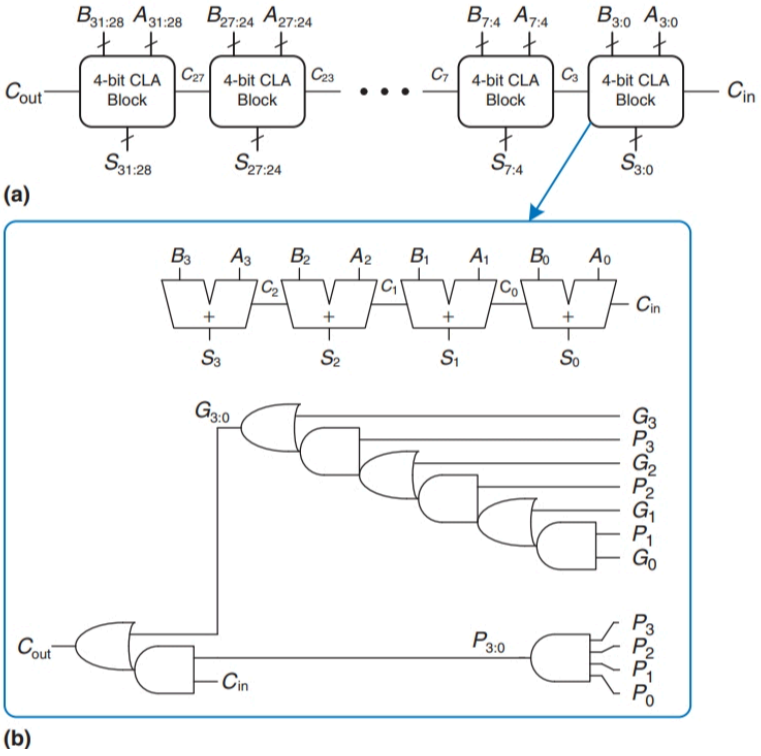


Figure 5.6 (a) 32-bit carry-lookahead adder (CLA), (b) 4-bit CLA block

硬件描述语言提供 `+` 操作来描述**进位传播加法器** (CPA)
以下 SystemVerilog 代码描述了一个有输入/输出进位的进位传播加法器:

```
module adder #(parameter N = 8)
              (input logic [N-1:0]  a, b,
               input logic          cin,
               output logic [N-1:0] s,
               output logic         cout);
    assign {cout, s} = a + b + cin;
endmodule
```

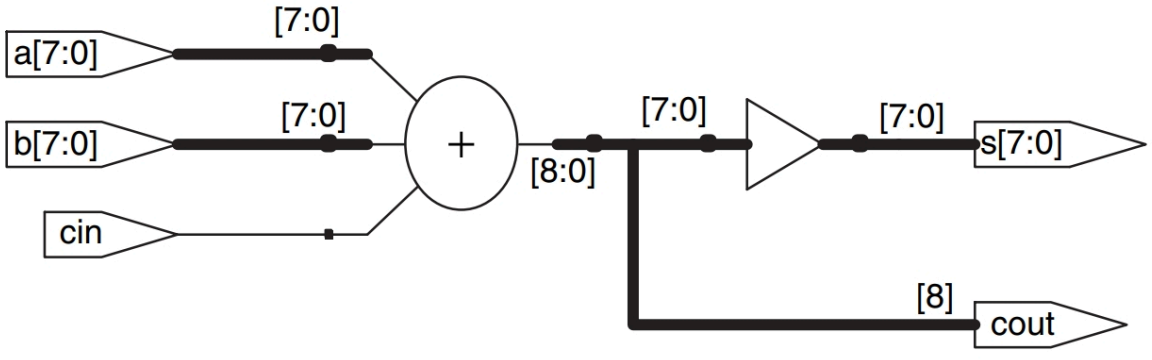
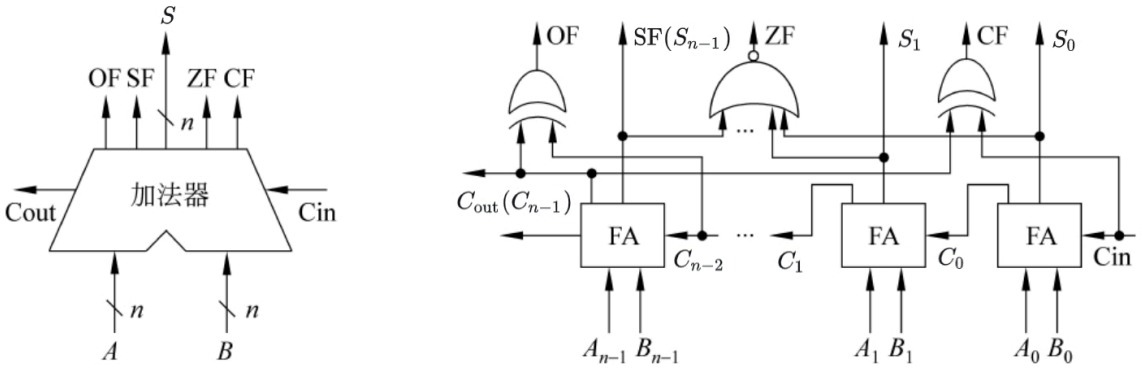


Figure 5.8 Synthesized adder

(4) 带标志加法器 (Flagged Adder)

- **溢出标志** (Overflow Flag)：$\mathrm{OF} = C_{\text{out}} \oplus C_{n-2} = C_{n-1} \oplus C_{n-2}$
- **符号标志** (Sign Flag)：$\mathrm{SF} = S_{n-1}$
- **零标志** (Zero Flag)：$\mathrm{ZF} = \overline{S_{n-1}S_{n-2}\cdots S_0}$ ($\mathrm{ZF} = 1$ 当且仅当 $S = 0$)
- **进位标志** (Carry-in Flag)：$\mathrm{CF} = C_{\text{out}} \oplus C_{\text{in}}$



(a) 带标志加法器符号　　(b) 带标志加法器的逻辑电路

图 3.6　用全加器实现 n 位带标志加法器的电路

4.1.2 减法 (Subtraction)

减法非常简单：改变减数的符号，然后做加法.
改变二进制补码的符号就是反转所有的位，然后末位加 1
具体来说，为计算 $Y = A - B$
首先创建减数 B 的二进制补码，然后各位取反、末位加 1 得到 $-B = \overline{B} + 1$
最后将 $-B$ 与 A 相加，得到 Y
上述过程可由进位传播加法器实现，
其中设置输入进位 $C_{\text{in}} = 1$，加数和被加数分别为 $A, \overline{B}$ (如图所示)

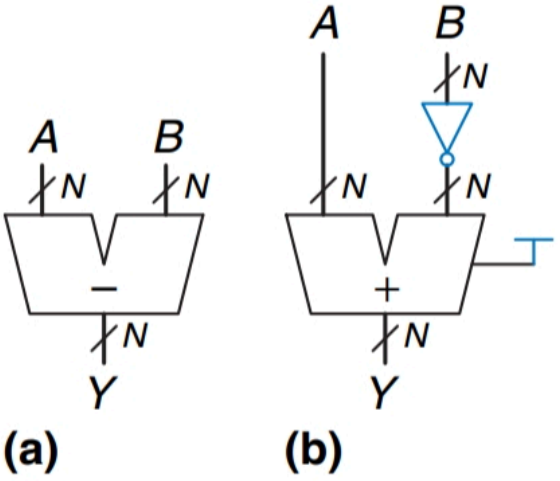


Figure 5.9 Subtractor: (a) symbol, (b) implementation

以下 SystemVerilog 代码描述了一个**减法器**:

```
module subtractor #(parameter N = 8)
                   (input logic [N-1:0]  a, b,
                    output logic [N-1:0] y);
    assign y = a - b;
endmodule
```

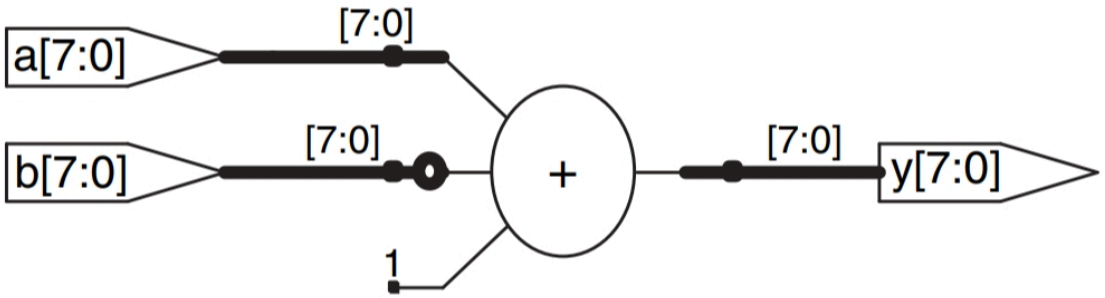


Figure 5.10 Synthesized subtractor

4.1.3 比较器 (Comparators)

比较器用于判断两个二进制数是否相等，或者一个比另一个大还是小.

- **相等比较器** (equality comparator):
  它通过 XNOR 门电路检查 A, B 的每一列的对应位是否相等.
  如果每一位都相等，则 A, B 相等.

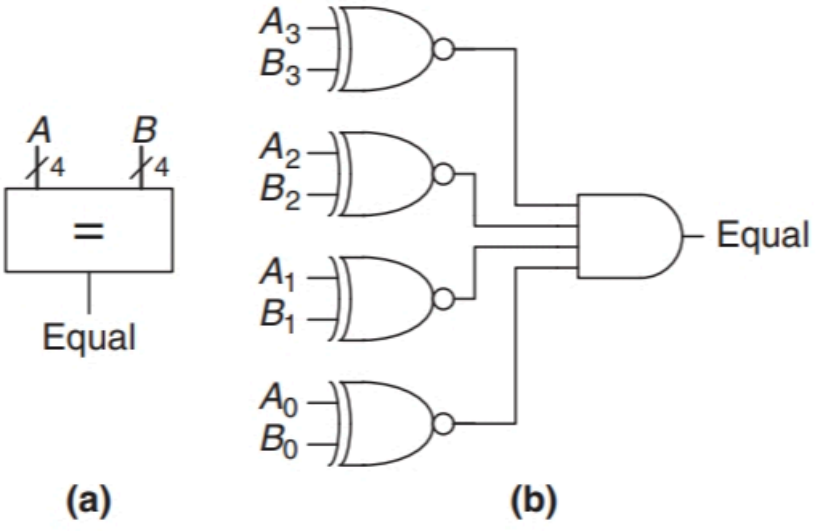


Figure 5.11 4-bit equality comparator: (a) symbol, (b) implementation

- **量值比较器** (magnitude comparator):
  它通过计算 $A - B$ 的值，然后检查结果的符号位 (最高有效位) 判断 A, B 的大小关系.
  若结果为负，则 $A < B$；否则 $A \geq B$

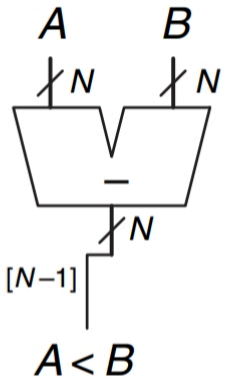


Figure 5.12 *N*-bit magnitude comparator

以下 SystemVerilog 代码描述了一个综合的**比较器**:

```
module comparator #(parameter N = 8)
                   (input logic [N-1:0] a, b,
                    output logic        eq, neq, lt, lte, gt, gte);
    assign eq = (a == b);
    assign neq = (a != b);
    assign lt = (a < b);
    assign lte = (a <= b);
    assign gt = (a > b);
    assign gte = (a >= b);
endmodule
```

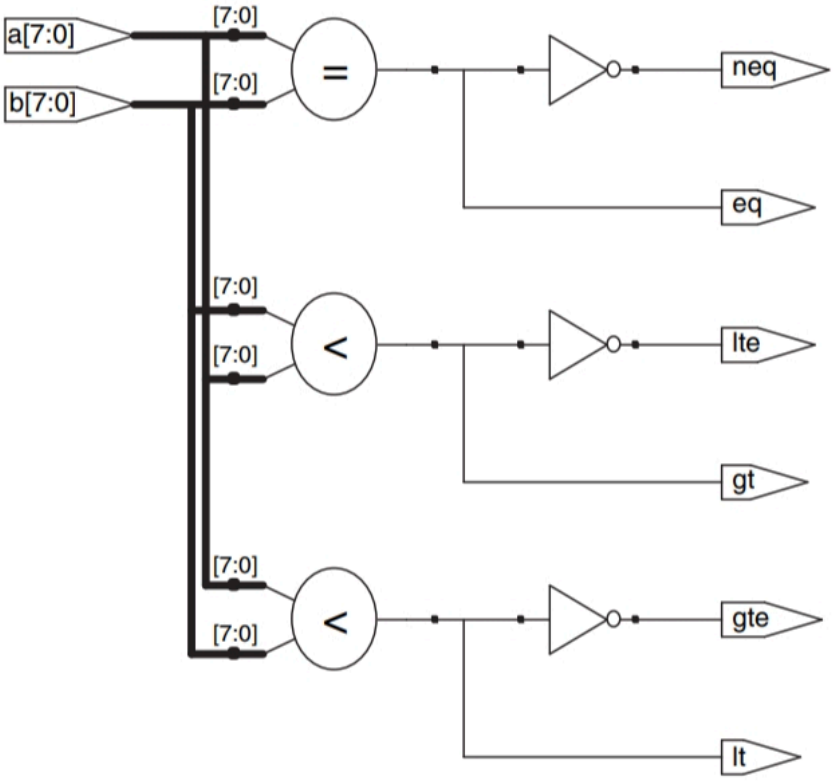


Figure 5.13 Synthesized comparators

4.1.4 算术逻辑单元 (ALU, Arithmetic/Logical Unit)

算术逻辑单元将多种算术和逻辑运算组合到一个单元内.
典型的算术逻辑单元可以执行加法、减法、量值比较、AND 和 OR 运算,
并由操作码 ALUop 决定操作类型 (由多路选择器 (MUX, Multiplexer) 选择输出的结果).

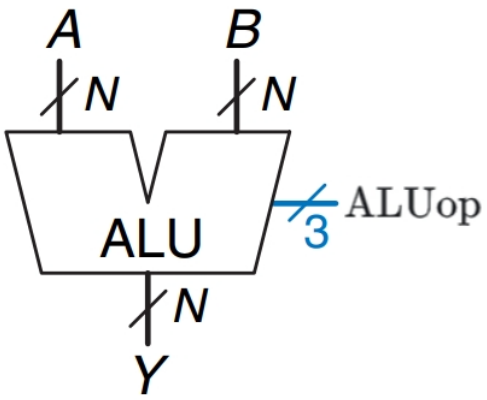


Figure 5.14 ALU symbol

作为一个具体的例子，3 位 ALUop 可以这样设计：

ALUop	Operation
000	A and B
001	A or B
010	$A + B$
011	(not used)
100	A and $\overline{B}$
101	A or $\overline{B}$
110	$A - B$
111	$\mathrm{SLT} = (A < B)$

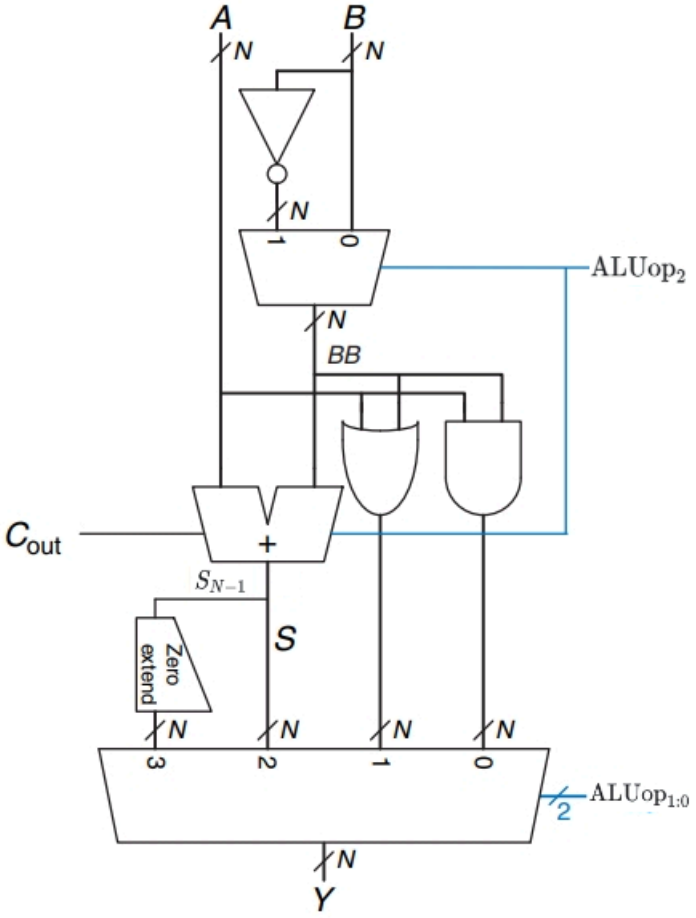


Figure 5.15 ***N*-bit ALU**

其中 $BB = \begin{cases} B, & \text{ALUop}_2 = 0 \\ \overline{B}, & \text{ALUop}_2 = 1 \end{cases}$

而 $Y = \begin{cases} A \text{ and } BB, & \text{ALUop}_{1:0} = 00 \\ A \text{ or } BB, & \text{ALUop}_{1:0} = 01 \\ A + BB, & \text{ALUop}_{1:0} = 10 \ \ (\text{Note that } C_{\text{in}} \text{ is ALUop}_2) \\ \text{zero-extended}\{\text{sign}(A - B)\}, & \text{ALUop}_{1:0} = 11 \end{cases}$

4.1.5 移位器 (Shifters)

一个 N 位移位器可以由 N 个 $32:1$ 多路选择器构成，
根据 $\log_2(N)$ 位移位数的值，对输入值 A 进行移位.
以下 SystemVerilog 代码给出了 32 位移位器的实现：

```
module SHIFTER(input  [31:0] A,
               input  [4:0]  SA, // SA for Shift Amount,5位移位数
               input         Right, Arith,
               // Right 用于判别是右移还是左移; Arith 用于判别是算术移位还是逻辑移位
               output [31:0] Y);

    // 临时变量，用于存储各次局部移位的中间结果
    wire [31:0] t0, t1, t2, t3, t4, s0, s1, s2, s3, s4;
    wire [31:0] l0, l1, l2, l3, l4, r0, r1, r2, r3, r4;

    // 生成16位的左移拼接量和右移拼接量
    // 左移拼接量left_shift为16位0
    wire [15:0] left_shift = 16'b0;
```

```
    // indicator是根据最高位(符号位)和Arith的值生成的指示值
    // 右移拼接量right_shift = 16{indicator};
    wire indicator = A[31] & Arith;
    wire [15:0] right_shift = {16{indicator}};

    // 4th    根据移位数的第四位SA[4]对A进行移位
    assign l4 = {A[15:0],left_shift[15:0]};
    assign r4 = {right_shift[15:0],A[15:0]};
    MUX2X32 Mux4LandR(l4,r4,Right,t4);// 根据Right选择左移还是右移，存入t4
    MUX2X32 shift_on_4th(A,t4,SA[4],s4);// 根据移位数的第四位SA[4]对A进行移位

    // 3rd    根据移位数的第三位SA[3]对s4进行移位
    assign l3 = {s4[23:0],left_shift[7:0]};
    assign r3 = {right_shift[7:0],s4[31:8]};
    MUX2X32 Mux3LandR(l3,r3,Right,t3);// 根据Right选择左移还是右移，存入t3
    MUX2X32 shift_on_3rd(s4,t3,SA[3],s3);// 根据移位数的第三位SA[3]对s4进行移位

    // 2nd    根据移位数的第二位SA[2]对s3进行移位
    assign l2 = {s3[27:0],left_shift[3:0]};
    assign r2 = {right_shift[3:0],s3[31:4]};
    MUX2X32 Mux2LandR(l2,r2,Right,t2);// 根据Right选择左移还是右移，存入t2
    MUX2X32 shift_on_2nd(s3,t2,SA[2],s2);// 根据移位数的第二位SA[2]对s3进行移位

    // 1st    根据移位数的第一位SA[1]对s2进行移位
    assign l1 = {s2[29:0],left_shift[1:0]};
    assign r1 = {right_shift[1:0],s2[31:2]};
    MUX2X32 Mux1LandR(l1,r1,Right,t1);// 根据Right选择左移还是右移，存入t1
    MUX2X32 shift_on_1st(s2,t1,SA[1],s1);// 根据移位数的第一位SA[1]对s2进行移位

    // 0th    根据移位数的第0位SA[0]对s1进行移位
    assign l0 = {s1[30:0],left_shift[0]};
    assign r0 = {right_shift[0],s1[31:1]};
    MUX2X32 Mux0LandR(l0,r0,Right,t0);// 根据Right选择左移还是右移，存入t0
    MUX2X32 shift_on_0th(s1,t0,SA[0],s0);// 根据移位数的第0位SA[1]对s1进行移位

    // s0即为最终的移位结果;
    assign Y = s0;
endmodule
```

其中多路选择器 MUX2X32 的实现为：

```
module MUX2X32(input      [31:0] A0,A1,
               input             S,
               output reg [31:0] Y);

    // 表达组合逻辑的always_comb语句
    // 实现的功能其实是Y = (S) ? A1 : A0, 注意问号表达式是先真后假
    always_comb
        case(S)
            1'b0: Y = A0;
            1'b1: Y = A1;
        endcase
endmodule
```

4.2 时序电路模块 (Sequential Building Blocks)

4.2.1 计数器 (Counters)

N 位二进制计数器 (binary counter) 包含时钟和复位输入，以及 N 位输出 Q.
复位 (reset) 将输出初始化为 0,
然后计数器在每个时钟的上升沿递增 1，以二进制顺序输出所有 2^N 种可能的值.
我们可以用加法器和复位寄存器构成 N 位计数器:
每一个时钟周期，计数器对存储在寄存器中的值加 1.

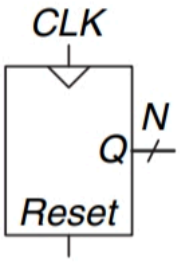


Figure 5.30 Counter symbol

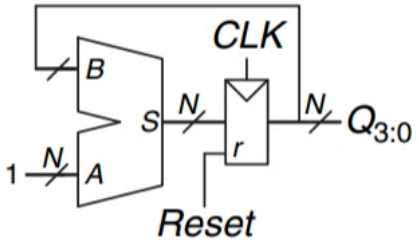


Figure 5.31 N-bit counter

下面的代码描述了一个异步复位二进制计数器:

```
module counter #(parameter N = 8)
                (input  logic         clk,
                 input  logic         reset,
                 output logic [N-1:0] q);
    always_ff @(posedge clk, posedge reset)
        if (reset) q <= 0;
        else q <= q + 1;
endmodule
```

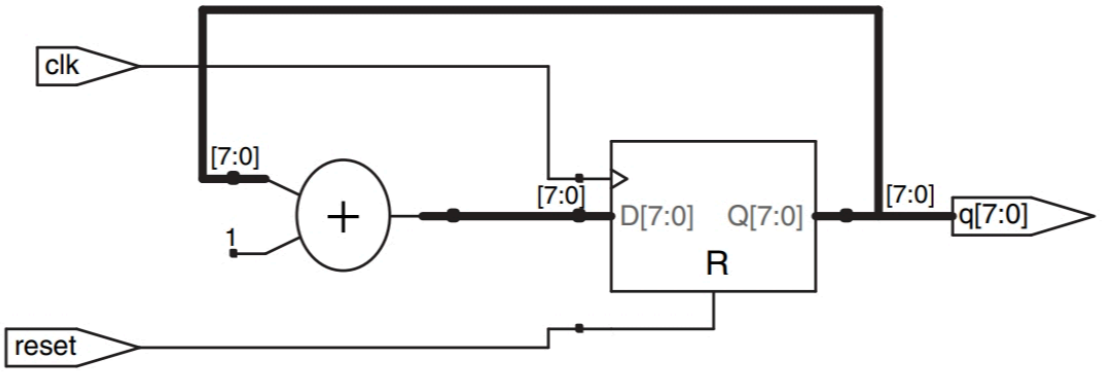


Figure 5.32 Synthesized counter

4.3 存储器阵列 (Memory Arrays)

数字系统还需要**存储器** (memory) 来存储电路需要使用的数据和生成的数据.
由触发器组成的**寄存器**只是一种存储少量数据的存储器.
本节将介绍可以有效存储大量数据的**存储器阵列** (memory array)

4.3.1 概述 (Overview)

存储器阵列的通用电路符号如下图所示:

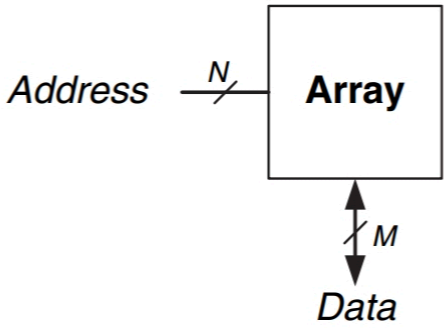


Figure 5.38 Generic memory array symbol

存储器由一个二维存储器单元阵列构成,
它可以读取或者写入**数据** (data) 到阵列中由**地址** (address) 指定的一行.
一个有 N 位地址和 M 位数据的阵列有 2^N 行和 M 列.
每行数据称为一个**字** (word)
因此，这样一个阵列包含了 2^N 个 M 位字.

阵列的行数又称**深度** (depth)，列数又称**宽度** (width),
我们一般使用 **(深度 $\times$ 宽度)** 来描述存储器的容量.
一个 1024×32 位阵列的符号如下图所示,
此阵列的总大小为 $32\mathrm{Kb}$ (kilobits, 区别于 KB, KiloBytes)

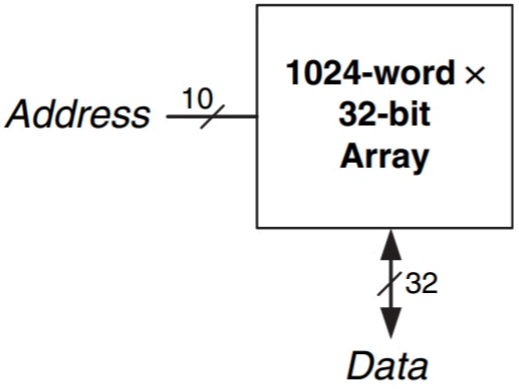


Figure 5.40 32 Kb array: depth = 2^{10} = 1024 words, width = 32 bits

- **(1) 位单元 (Bit Cells):**
  存储器阵列由**位单元**的阵列组成，
  每个位单元存储 1 位数据，与一个**字线** (wordline) 和一个**位线** (bitline) 相连 (如图所示).

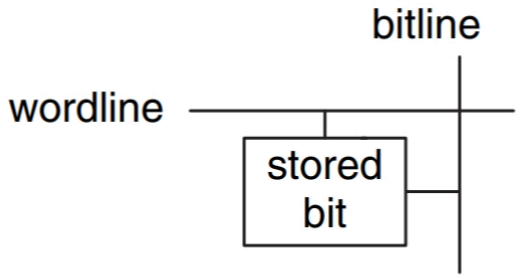


Figure 5.41 Bit cell

  对于每一个地址位的组合，
  存储器将**字线**设置为高电平，并激活此行中的位单元，
  对应的**位线**将传出或传入要存储的位.
  存储位 (stored bit) 的电路因存储器类型的不同而不同.

  - 为读取位单元，位线初始化为**浮空值** z，
    然后字线打开为高电平，允许存储位的值驱动位线为 0 或者 1
  - 为写入位单元，位线强制驱动为**期望写入的值**，
    然后字线打开为高电平，将位线连接到存储位，改写存储位的内容.

- **(2) 存储器的结构 (Organization):**
  一个 4×3 位阵列的内部结构如下图所示:

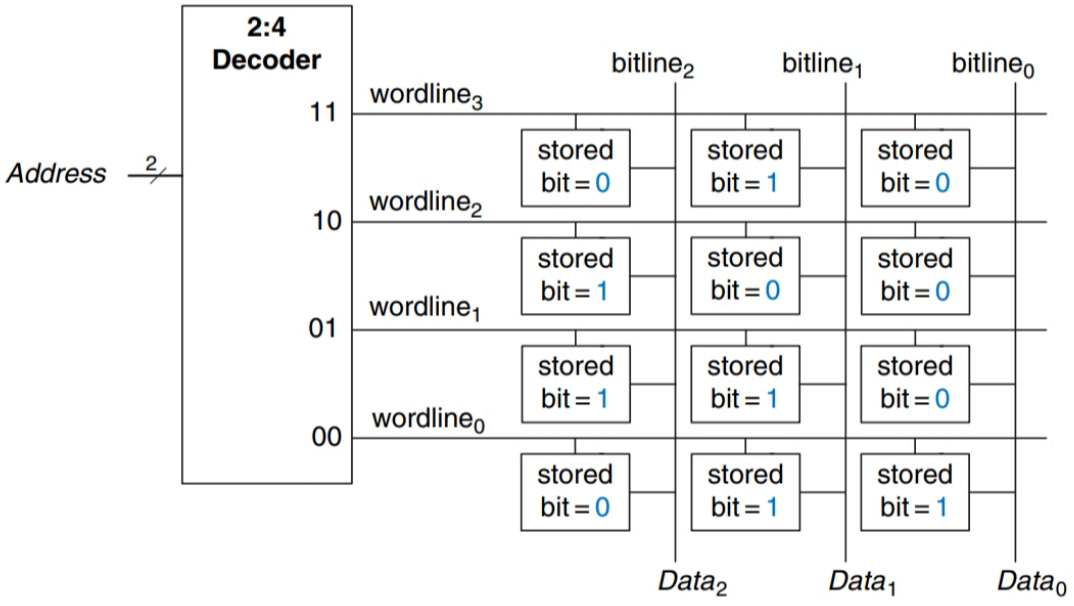


Figure 5.42 4 × 3 memory array

  - 在读存储器时，将地址指定的字线设为高电平，
    对应一整行的位单元驱动各自的位线为存储位的值.
  - 在写存储器时，首先将位线驱动为期望写入的值，
    然后将地址指定的字线设为高电平，
    对应一整行的位单元驱动各自的存储位为位线的值.

- **(3) 存储器端口 (Memory Ports):**
  存储器阵列可以有多个**端口** (port)，称为多端口存储器 (multiported memory)
  每个端口都能提供对一个存储器地址的读/写访问.
  下图描述了一个 3 端口存储器，其中有两个读端口和一个写端口.

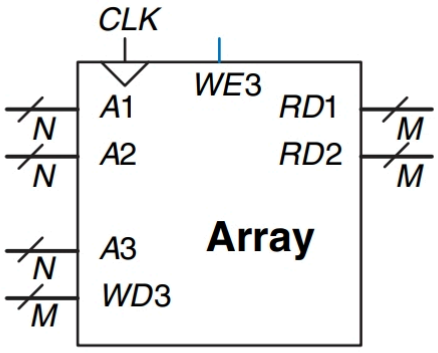


Figure 5.43 Three-ported memory

  端口 1 从地址 $\mathrm{A1}$ 读出数据到 $\mathrm{RD1}$;
  端口 2 从地址 $\mathrm{A2}$ 读出数据到 $\mathrm{RD2}$;
  至于端口 3:
  若写使能 $\mathrm{WE3}$ 有效，则在时钟的上升沿，端口 3 将来自 $\mathrm{WD3}$ 的数据写到地址 $\mathrm{A3}$

- **(4) 存储器类型 (Memory Types):**
  存储器可以根据它们如何在位单元上存储位来分类.
  最常用的分类是:
  随机访问存储器 (Random Access Memory, RAM) 和**只读存储器** (Read Only Memory, ROM)
  由于一些历史原因，RAM 和 ROM 被赋予了上述易引起误解的名称，
  事实上，ROM 同样支持随机访问，更糟糕的是，大多数现代 ROM 既可以读又可以写.
  不过我们只需记住它们最重要的区别:
  - RAM 是**易失的** (volatile)，即当断开电源时它就会丢失数据;
  - ROM 是**非易失的** (non-volatile)，即便断开电源它也可以无期限地保存数据;

  RAM 的两种主要类型为:
  - **动态 RAM (DRAM, Dynamic RAM):**
    它通过电容充放电来存储数据
  - **静态 RAM (SRAM, Static RAM):**
    它使用交叉耦合的反向器来存储数据

  对于 ROM，根据擦写方式的不同可以分为很多不同类型，这里不再赘述.

4.3.2 动态随机访问存储器 (DRAM)

动态 RAM 通过电容的充电和放电来存储位.
下图描述了一个动态 RAM 位单元.

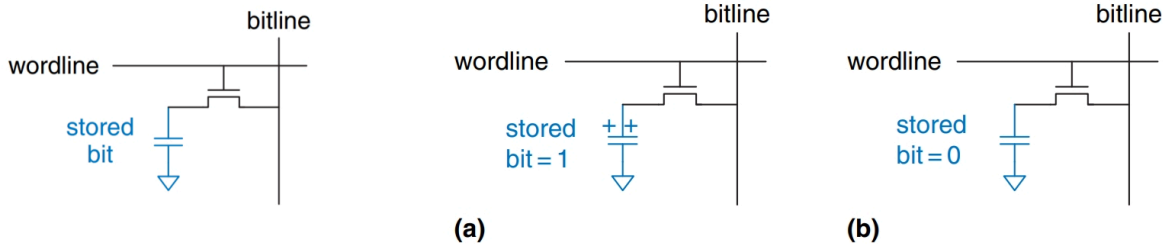


Figure 5.44 DRAM bit cell

Figure 5.45 DRAM stored values

位值存储在电容中，nMOS 晶体管作为开关，决定是不是从位线连接电容.
当字线处于高电平时，nMOS 晶体管为导通状态，存储位的值就可以在位线上传入或传出.
当读时，数据值从电容传送到位线；
当写时，数据值从位线传送到电容；
读破坏存储在电容中的位值，所以在每次读后需要恢复 (重写) 数据；
即使 DRAM 没有被读，电容的电压也会慢慢泄漏，其位值也必须在几毫秒内刷新 (读，然后重写).

4.3.3 静态随机访问存储器 (SRAM)

静态 RAM 之所以称为**静态的** (static)，是因为不需要刷新存储位.
下图描述了一个 SRAM 位单元：

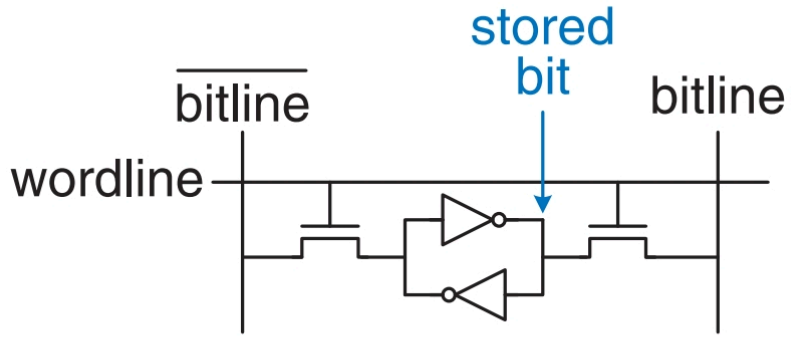


Figure 5.46 SRAM bit cell

数据位存储交叉耦合反相器中.
每个位单元有一条位线和一条输出位线.
当字线处于高电平时，两个 nMOS 晶体管都打开，数据值就从位线上传出或传入.
与 DRAM 不同的是，如果噪声减弱了存储位的值，则交叉耦合反相器将恢复存储值.

4.3.4 面积和延迟 (Area and Delay)

触发器 (flip-flop)、SRAM 和 DRAM 都是**易失的** (volatile) 存储器，
但是它们有着不同的面积 (area) 和延迟 (delay) 特性，参见下表：

Table 5.4 Memory comparison

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

- 触发器可以通过其输出直接访问存储的数据，但它至少需要 20 个晶体管来构成.
  总的来说，晶体管数越多的器件，芯片面积、功耗和成本也更高.
- DRAM 延迟比 SRAM 延迟更长，因为它的位线不是用晶体管驱动的.
  DRAM 必须等待充电，从电容将值移动到位线的速度较慢.
  DRAM 的吞吐量也比 SRAM 的低，因为它必须周期性地 (以及在读取之后) 进行刷新.
- 存储器延迟和吞吐量也与存储器大小有关.
  在其他条件相同的情况下，大容量存储器一般比小容量存储器更慢.

4.3.5 寄存器堆 (Register Files)

数字系统通常使用一组寄存器来存储临时变量.
这组寄存器称为**寄存器堆** (register file)，
它通常由小型多端口 SRAM 阵列组成，因为它比触发器阵列更紧凑.

下图描述了一个 32×32 位 3 端口寄存器堆：

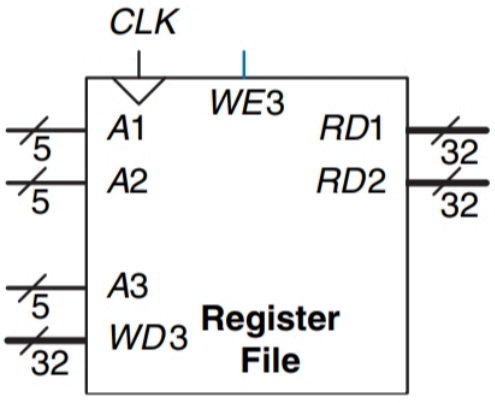


Figure 5.47 32×32 register file with two read ports and one write port

上述寄存器堆有两个读端口 ($A1/RD1$ 和 $A2/RD2$) 和一个写端口 ($A3/WD3$)
地址线 $A1, A2, A3$ 均为 5 位，它们可以访问所有的 $2^5 = 32$ 个寄存器.

端口 1 从地址 $\mathrm{A1}$ 读出数据到 $\mathrm{RD1}$；
端口 2 从地址 $\mathrm{A2}$ 读出数据到 $\mathrm{RD2}$；
至于端口 3：
若写使能 $\mathrm{WE3}$ 有效，则在时钟的上升沿，端口 3 将来自 $\mathrm{WD3}$ 的数据写到地址 $\mathrm{A3}$ 对应的寄存器.
因此，上述寄存器堆可以同时读两个寄存器和写一个寄存器.

4.3.6 只读存储器 (Read Only Memory, ROM)

只读存储器 ROM 以晶体管的存在与否来存储一位.
下图描述了一个简单的 ROM 位单元的两种可能状态：

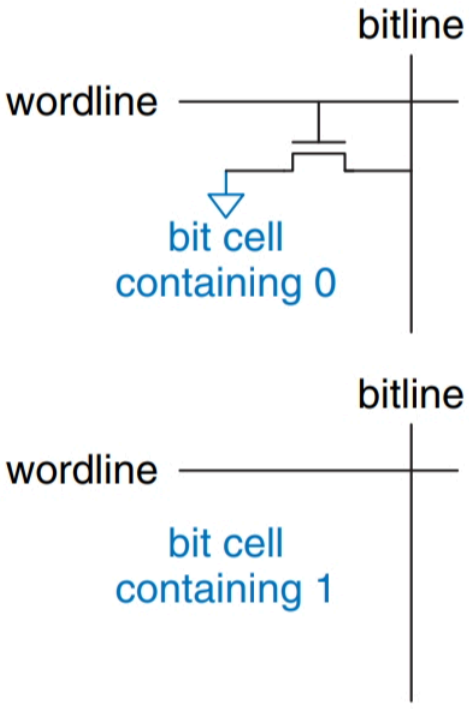


Figure 5.48 ROM bit cells containing 0 and 1

为了读这个位单元，位线被缓慢地拉至高电平，随后打开字线.
如果晶体管存在，则它将使位线变为低电平.
如果晶体管不存在，则位线将保待高电平.
上面讨论的**简单 ROM 位单元**是组合电路，它没有可以 “遗忘” 的状态，
因而是**非易失的** (non-volatile)，而且只能读取，无法写入.

ROM 的内容可用**点表示法** (dot notation) 来描述，
在字线 (行) 和位线 (列) 交叉处的标点表示此数据位为 1 (如图所示)

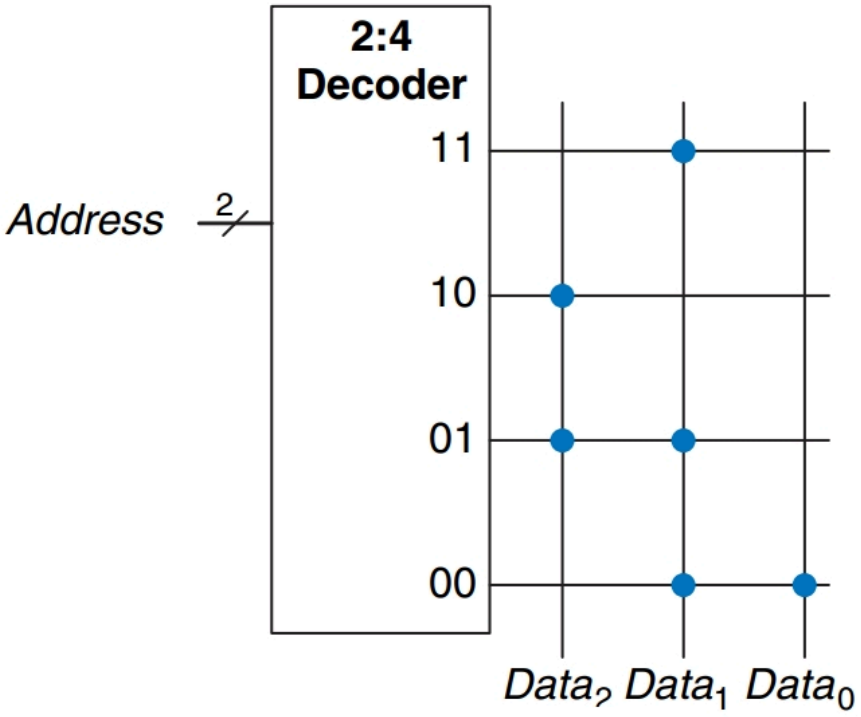


Figure 5.49 4 × 3 ROM: dot notation

在制造时，ROM 位单元的内容可以用晶体管的有无来确定.
可编程 ROM (PROM, Programable ROM) 在每个位单元都放置一个晶体管，
然后提供方法决定晶体管是否接地.

- **熔丝烧断可编程 ROM** (fuse-programable ROM):
  又称**一次可编程 ROM** (one-time programmable ROM)
  使用者通过应用高电压有选择地熔断熔丝来对 ROM 编程.
  若熔丝存在，则晶体管接地，位单元的位值保持 0;
  若熔丝熔断，则晶体管与地断开，位单元的位值保持 1;

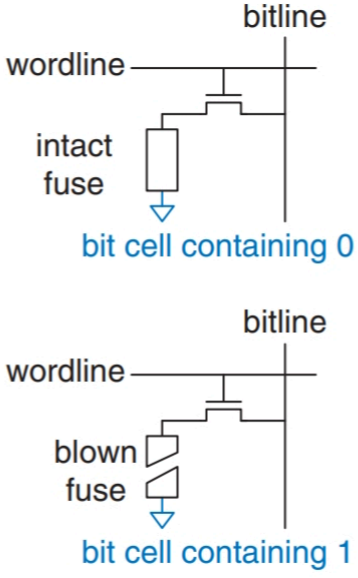


Figure 5.51 Fuse-programmable ROM bit cell

- **可重复编程 ROM** (Reprogrammable ROM) 提供一种可修改机制来确定晶体管是否接地.
  例如**可擦除 PROM** (EPROM, Erasable PROM)
  它使用**浮动栅晶体管** (floating-gate transistor) 代替 nMOS 晶体管和熔丝.
  原理不多赘述，总之大多数现代 ROM 不再只读，它们也可以写入.
  RAM 和 ROM 的不同在于，ROM 的**写入时间更长** (涉及擦除机制)，但它是**非易失的** (non-volatile)

4.3.7 存储器的 HDL 代码

下面的代码描述了一个 $2^N \times M$ 位的 RAM:

```
module ram #(parameter N = 6, M = 32)
            (input  logic         clk,
             input  logic         we,
             input  logic [N-1:0] adr,
             input  logic [M-1:0] din,
             output logic [M-1:0] dout);

    logic [M-1:0] mem [2**N-1:0];
    always_ff @(posedge clk)
        if (we) mem [adr] <= din;
    assign dout = mem[adr];

endmodule
```

上述 RAM 有一个同步写使能 `we`
当写使能 `we` 有效时，在时钟的上升沿就会发生写入.
而读则可以立即得到结果.
当第一次加电时，RAM 的初始内容是不可预知的.

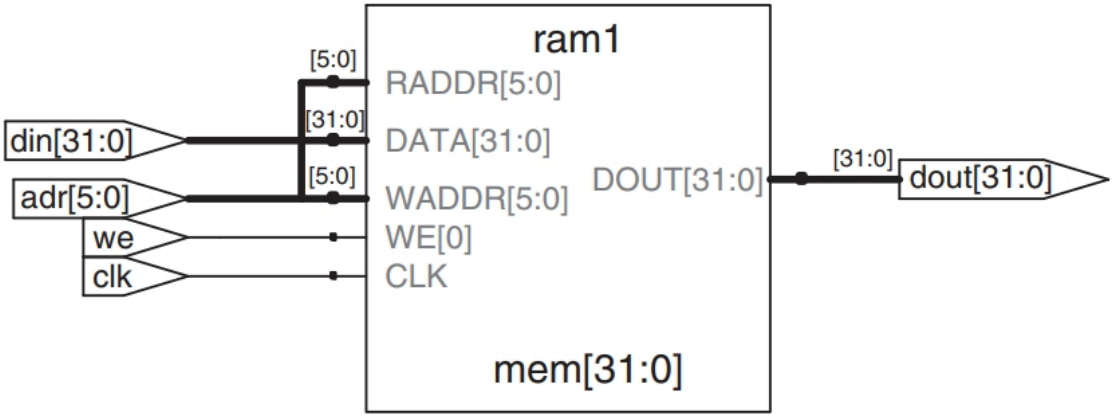


Figure 5.53 Synthesized ram

THE END