MC542

Organização de Computadores
Teoria e Prática

2007

Prof. Paulo Cesar Centoducatte

ducatte@ic.unicamp.br

www.ic.unicamp.br/~ducatte

MC542

Circuitos Lógicos

ULA, Operações de Ponto Flutuante, Memórias

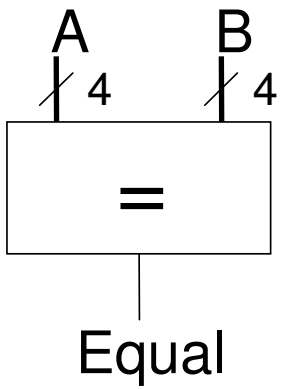
"DDCA" - (Capítulo 5)
"FDL" - (Capítulo)
"COD" - (Capítulo)

Sumário

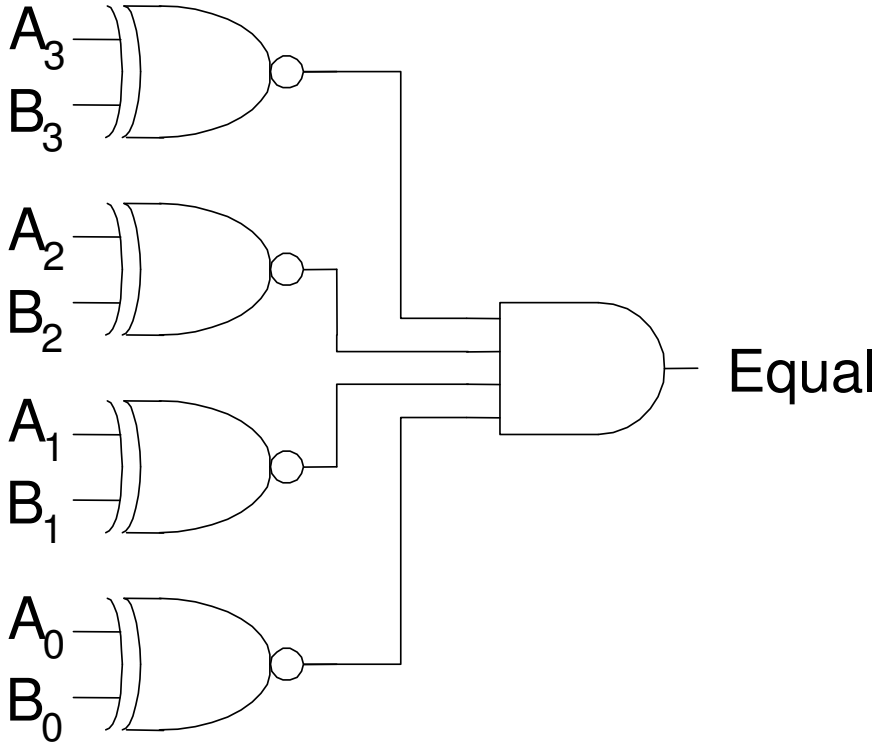
- Comparador
  - Igualdade
  - Menor Que
  - ……
- Arithmetic Logic Unit (ALU)
- Set on Less Than (SLT)
- Shifters
  - Shifters como Multiplicador e Divisor
- Multiplicação
- Divisão
- Sistemas Numéricos
  - Números Fracionários
  - Ponto Flutuante – Arredondamento
  - Ponto Flutuante: Adição
- Memórias

Comparador: Igual

Símbolo

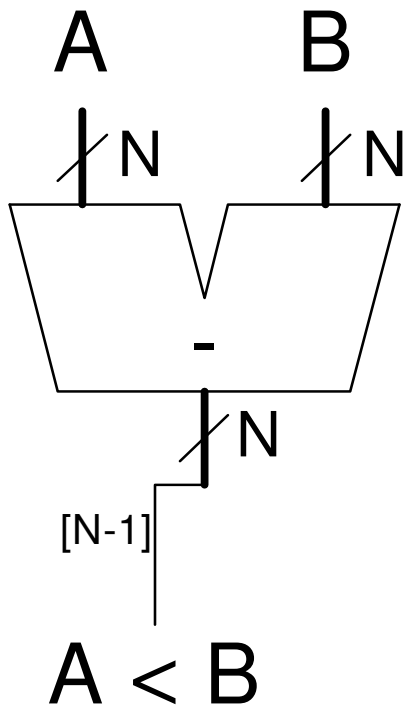


Implementação

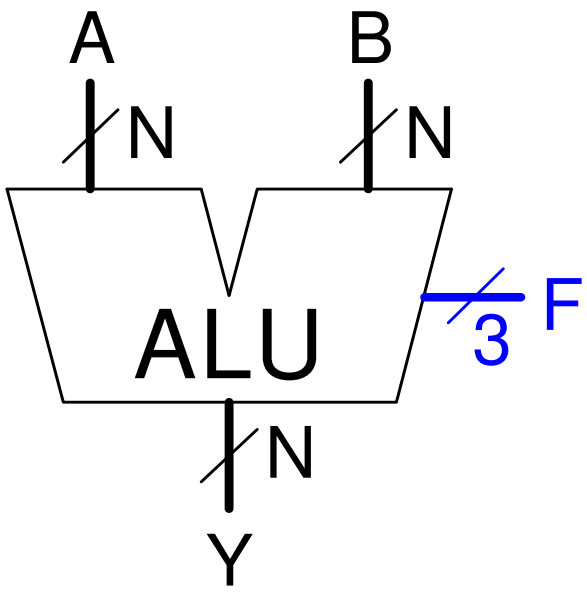


Comparador: Menor Que

$A < B \iff A - B < 0$



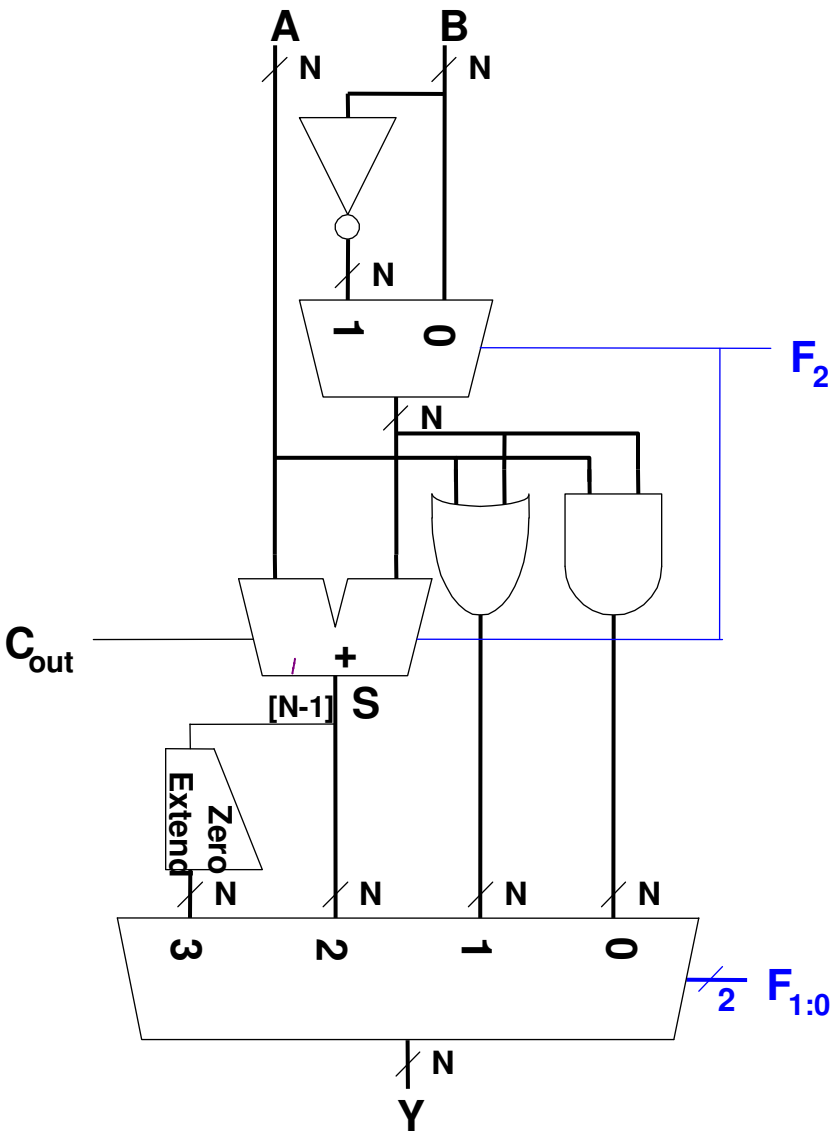
Arithmetic Logic Unit (ALU)



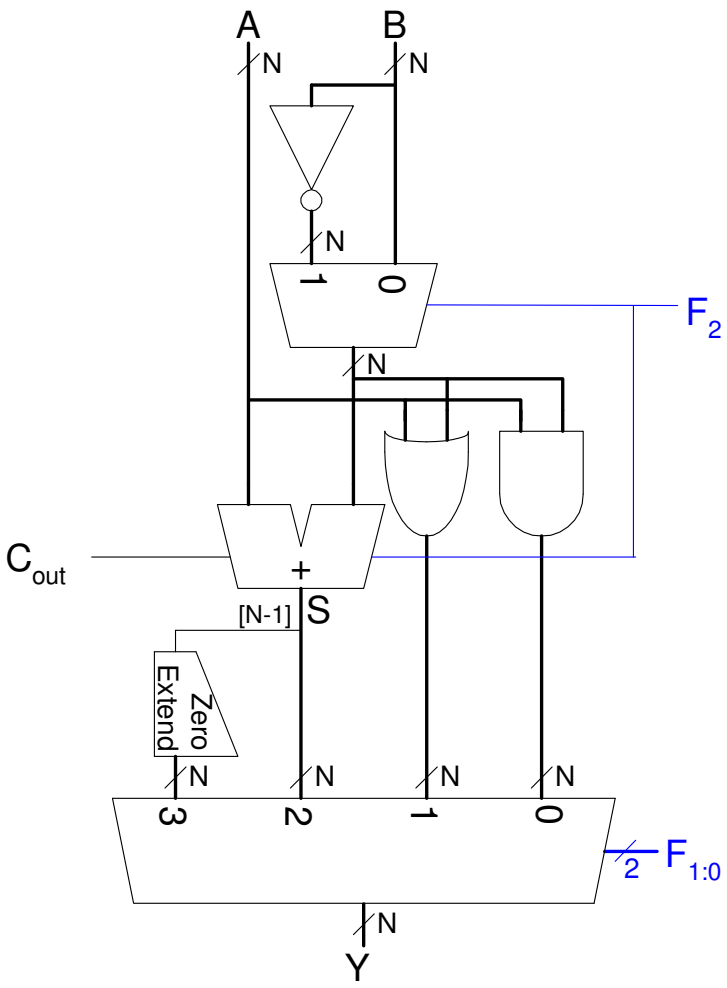
$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

- SLT – Set on Less Than

Projeto de uma ALU

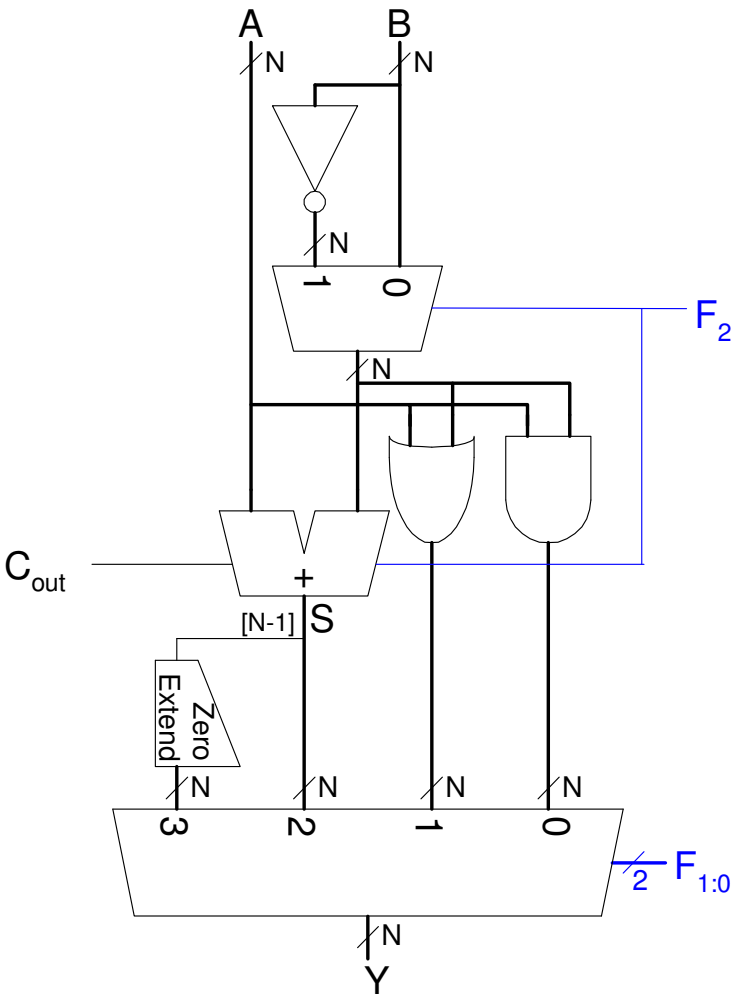


ALU - Funções



$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & B
101	A | ~B
110	A - B
111	SLT

Set Less Than (SLT)

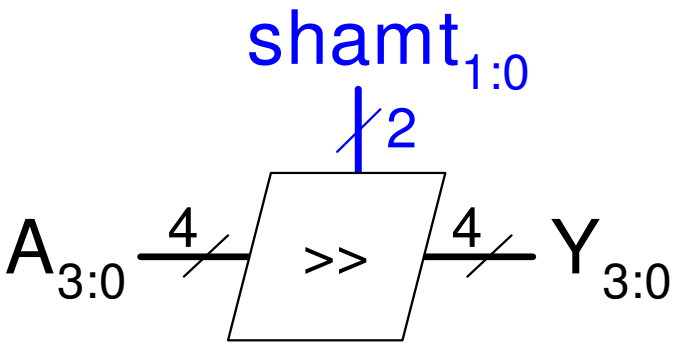


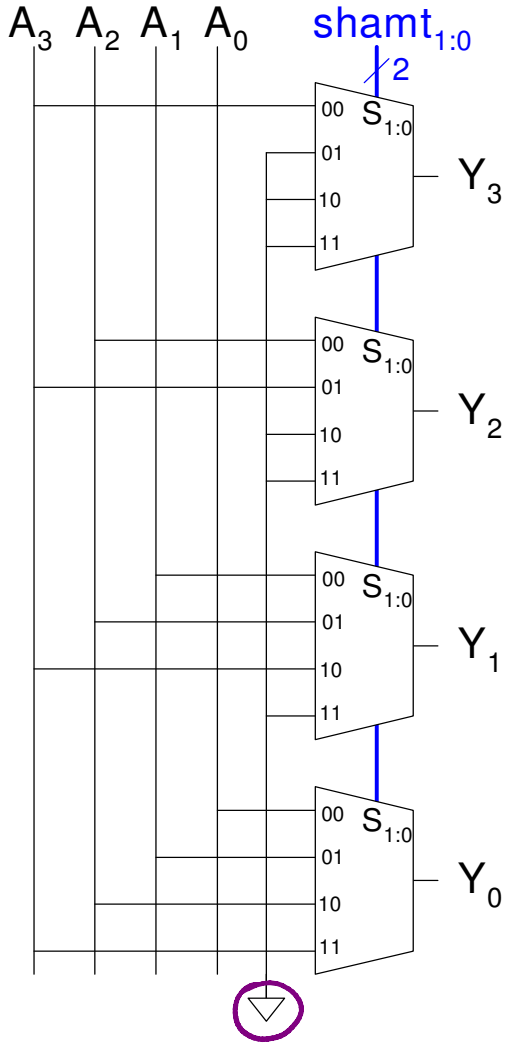
- Configuração da ALU 32-bit para a operação SLT. Suponha que A = 25 e B = 32.
  - Como A é menor que B, Y deve ser a representação de 1 em 32-bit (0x00000001).
  - Para SLT, $F_{2:0}$ = 111.
  - F_2 = 1 configura a unidade adder como um subtrator (25 - 32 = -7).
  - A representação complemento de dois de -7 tem um 1 no bit mais significativo, assim, S_{31} = 1.
  - Com $F_{1:0}$ = 11, o último mux seleciona Y = 1.

Shifters

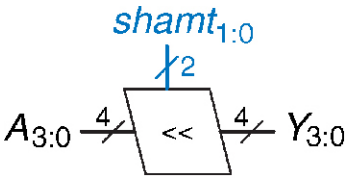
- **Shifter lógico:**
  - Ex: 11001 >> 2 = 00110
  - Ex: 11001 << 2 = 00100
- **Shifter Aritmético:**
  - Ex: 11001 >>> 2 = 11110
  - Ex: 11001 <<< 2 = 00100
- **Rotação:**
  - Ex: 11001 ROR 2 = 01110
  - Ex: 11001 ROL 2 = 00111

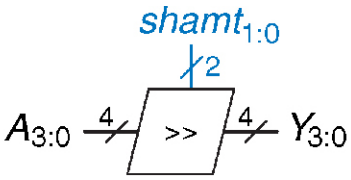
Shifter Rápido

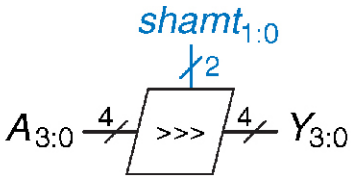


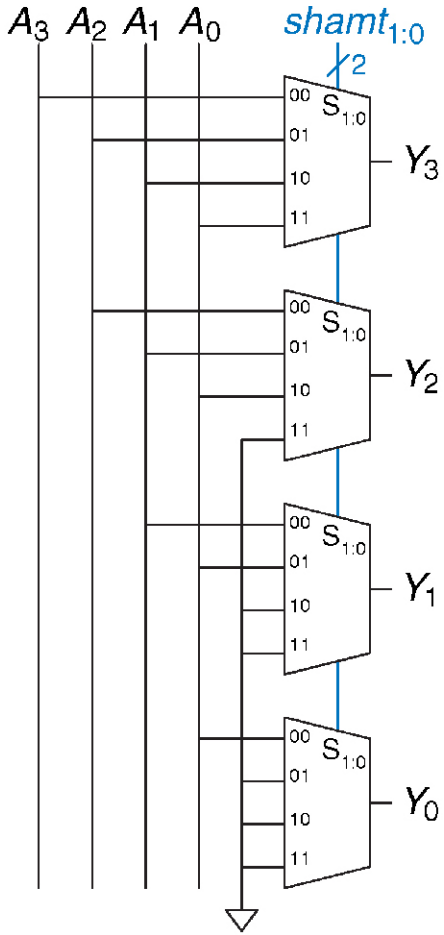


Shifter Rápido

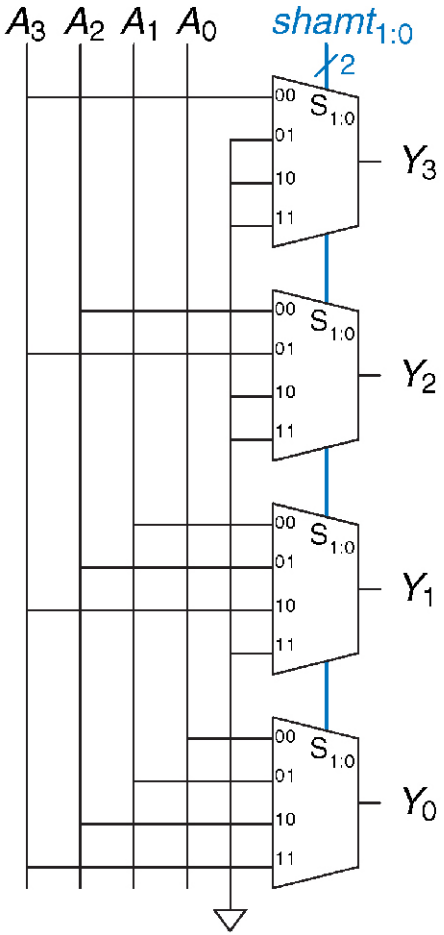




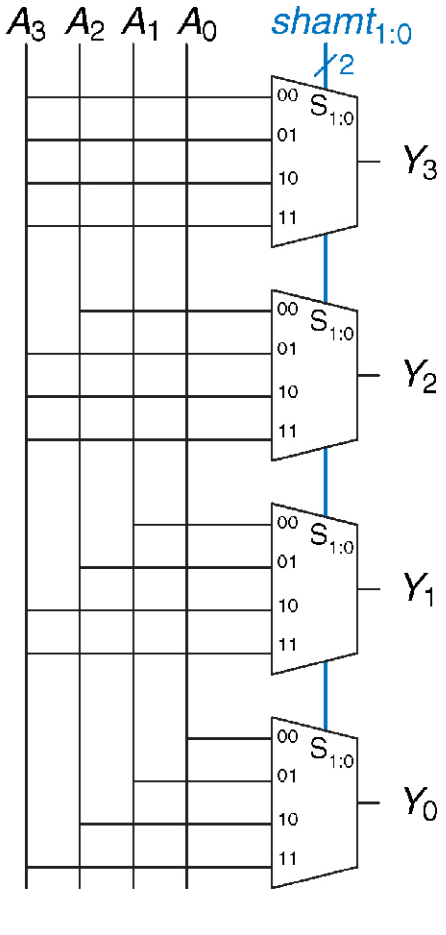




(a)



(b)



(c)

Shifters como Multiplicador e Divisor

- Um shift left de N bits múltiplica o número por 2^N
  - Ex: 00001 << 2 = 00100 ($1 \times 2^2 = 4$)
  - Ex: 11101 << 2 = 10100 ($-3 \times 2^2 = -12$)

- Um shift right aritmético de N divide o número por 2^N
  - Ex: 01000 >>> 2 = 00010 ($8 \div 2^2 = 2$)
  - Ex: 10000 >>> 2 = 11100 ($-16 \div 2^2 = -4$)

Multiplicação

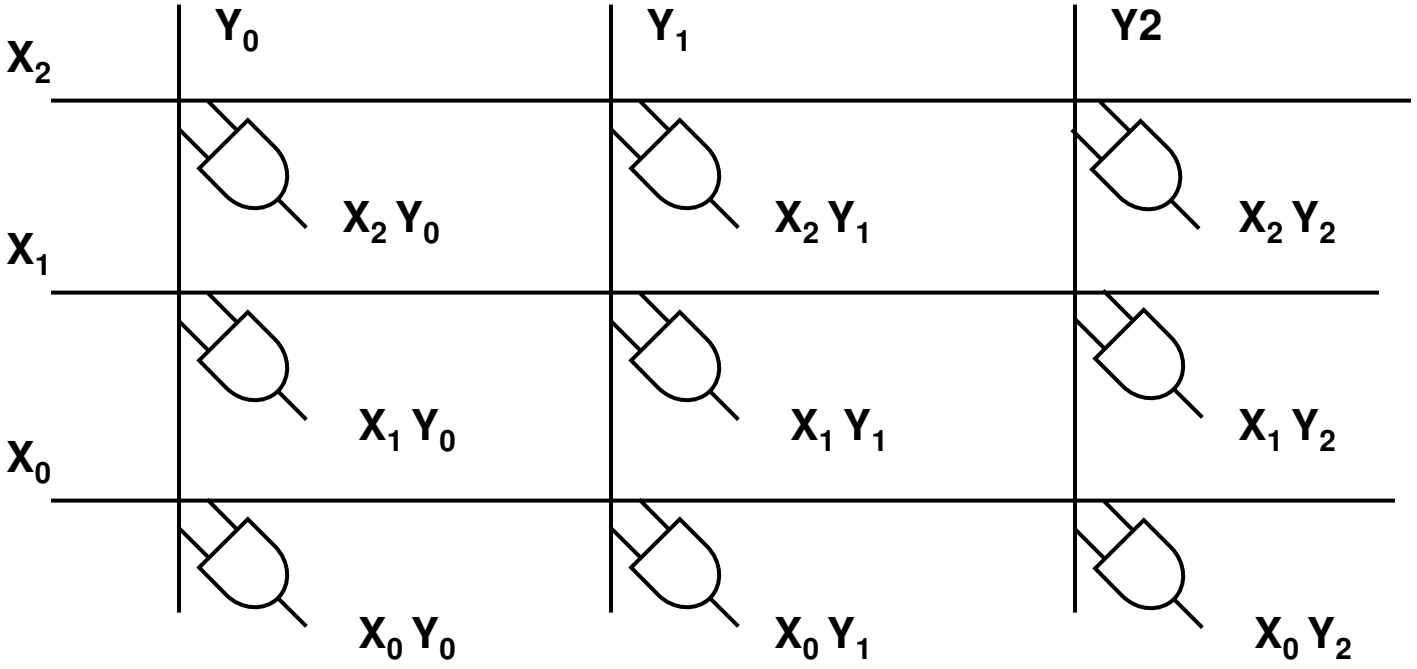
8_{10}	1000	multiplicando
9_{10}	1001	multiplicador
	1000	
	0000	produtos parciais
	0000	
	1000	
72_{10}	1001000	max= $(2^4-1) * (2^4-1) = 225$

225 > 128 $\Rightarrow$ 8 bits

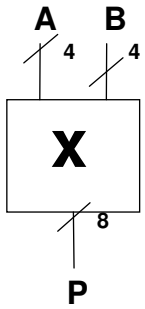
32 * 32 bits $\Rightarrow$ 64 bits

Multiplicação
Geração Rápida dos Produtos Parciais

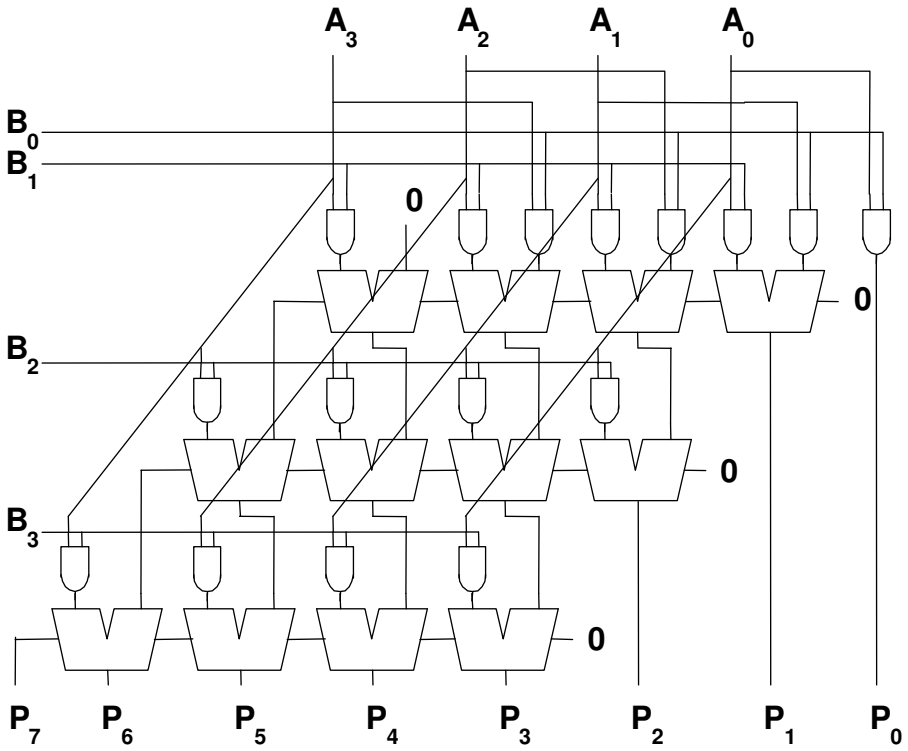
		Y_0	Y_1	Y_2
		X_0	X_1	X_2
		$X_2\ Y_0$	$X_2\ Y_1$	$X_2\ Y_2$
	$X_1\ Y_0$	$X_1\ Y_1$	$X_1\ Y_2$	
$X_0\ Y_0$	$X_0\ Y_1$	$X_0 Y_2$		



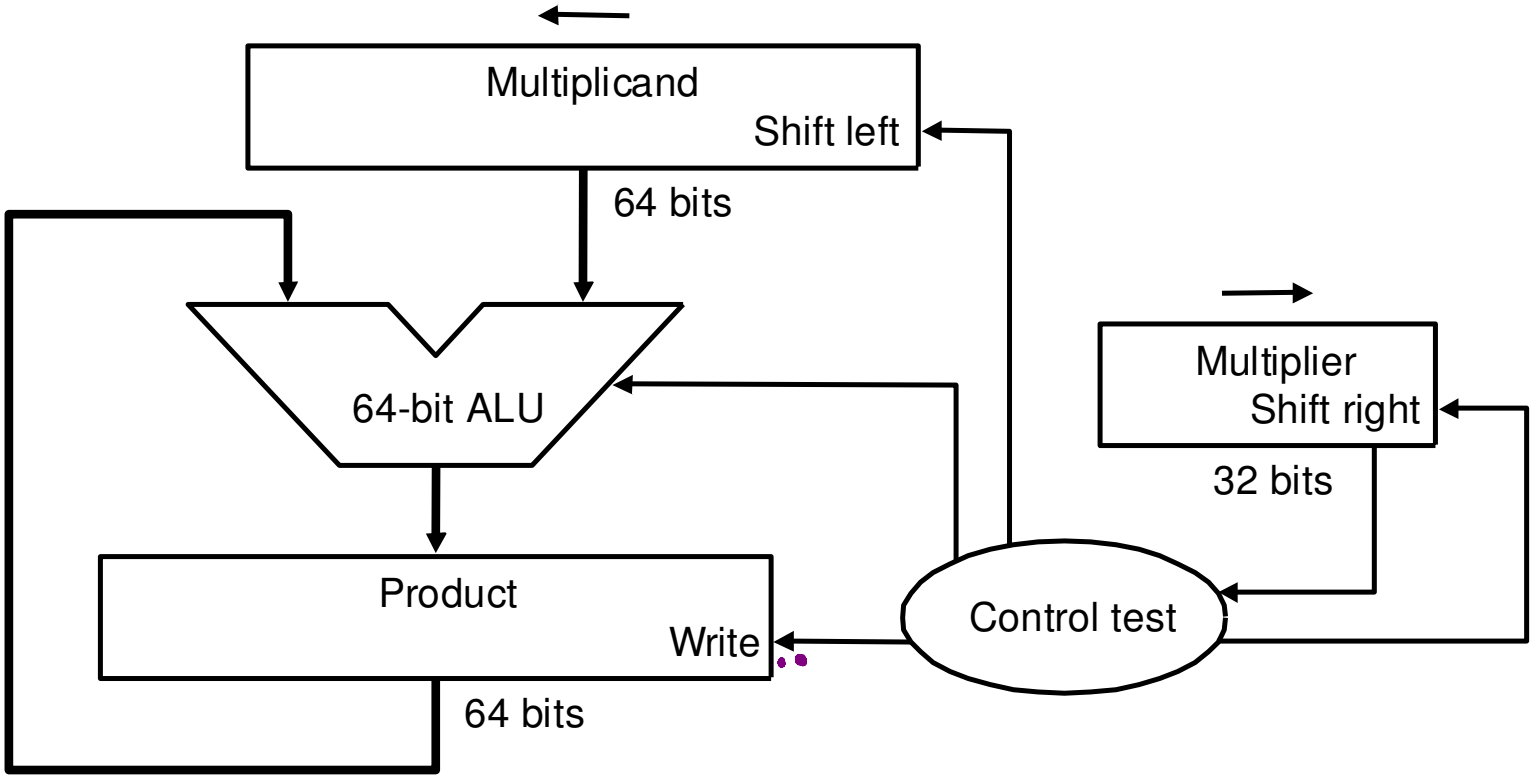
Multiplicador 4x4



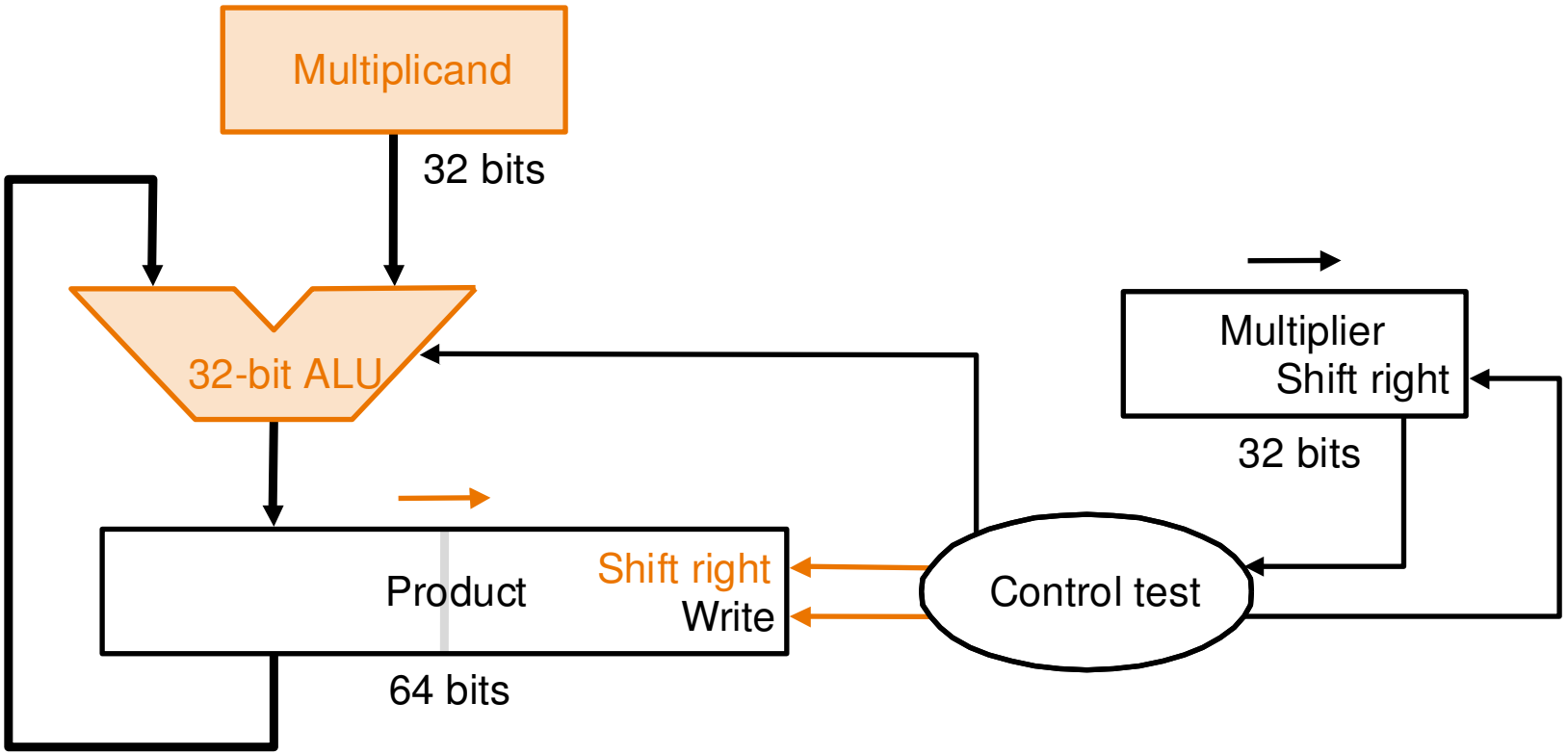
$$
\begin{array}{rcccccccc}
 & & & A_3 & A_2 & A_1 & A_0 & & \\
 & x & & B_3 & B_2 & B_1 & B_0 & & \\
\hline
 & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 & & & \\
 & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & & & & \\
A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & & & & \\
+ \quad A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & &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



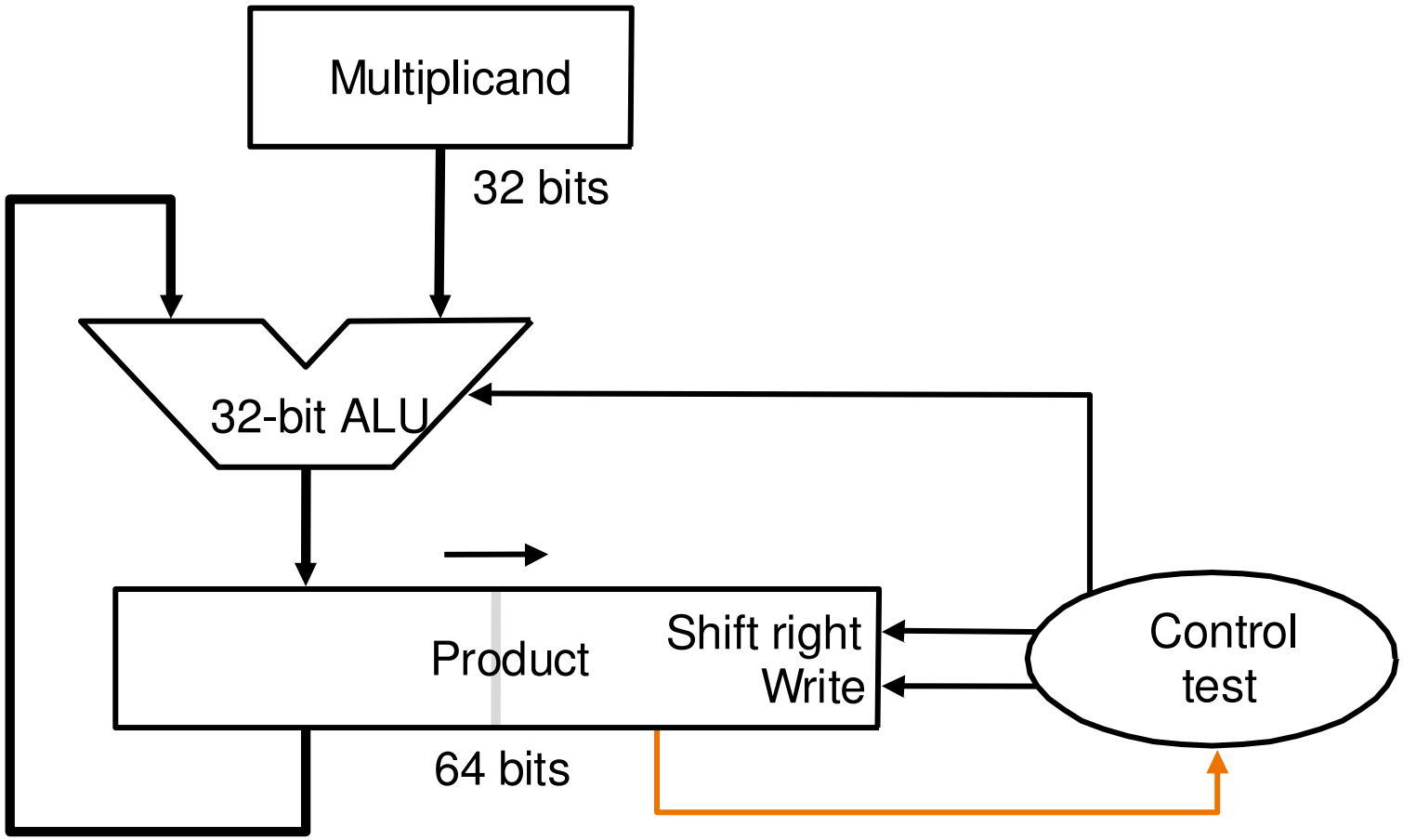
Multiplicador – Primeira Versão



Multiplicador - Segunda Versão



Multiplicador - Versão Final



Divisão

$29 \div 3 \Rightarrow$ $\quad 29 = 3 * Q + R = 3 * 9 + 2$

dividendo, divisor, quociente, resto

$29_{10} = 011101 \qquad 3_{10} = 11$

```
0 1 1 1 0 1 | 1 1
  1 1       ---------
  ---       0 1 0 0 1
  0 0 1 0 1
      1 1
  ---------
      1 0
```

Q = 9 R = 2

Como implementar em hardware?

Alternativa 1: Divisão com Restauração

- hardware não sabe se "vai caber ou não"
- registrador para guardar resto parcial
- verificação do sinal do resto parcial
- caso negativo ⇒ restauração

$29 - 3 * 2^4 = -19$	$q_4 = 1$	
$-19 + 3 * 2^4 = 29$	$q_4 = 0$	Restauração
$29 - 3 * 2^3 = 5$	$q_3 = 1$	
$5 - 3 * 2^2 = -7$	$q_2 = 1$	
$-7 + 3 * 2^2 = 5$	$q_2 = 0$	Restauração
$5 - 3 * 2^1 = -1$	$q_1 = 1$	
$-1 + 3 * 2^1 = 5$	$q_1 = 0$	Restauração
$5 - 3 * 2^0 = 2$	$q_0 = 1$	

$R = 10 = 2$ $\quad q_4q_3q_2q_1q_0 = 01001 = 9$

Alternativa 2: Divisão sem Restauração

Regras

se resto parcial	> 0	próxima operação	subtração
se resto parcial	< 0	próxima operação	soma

objetivo
$R \rightarrow 0$

se operação corrente	+	$q_i = \cancel{1}$
se operação corrente	-	$q_i = 1$

$29 - 3 * 2^4 = -19 < 0$	próx = SOMA	$q_4 = 1$
$-19 + 3 * 2^3 = 5 > 0$	próx = SUB	$q_3 = \cancel{1}$
$5 - 3 * 2^2 = -7 < 0$	próx = SOMA	$q_2 = 1$
$-7 + 3 * 2^1 = -1 < 0$	próx = SOMA	$q_1 = \cancel{1}$
$-1 + 3 * 2^0 = 2$		$q_0 = \cancel{1}$

Resto = 2 Quociente = $1\cancel{1}1\cancel{11}$??

Alternativa 2: Conversão do Resultado

$$1\ \bar{1}\ 1\ \bar{1}\ \bar{1} = (2^4 - 2^3 + 2^2 - 2^1 - 2^0)$$

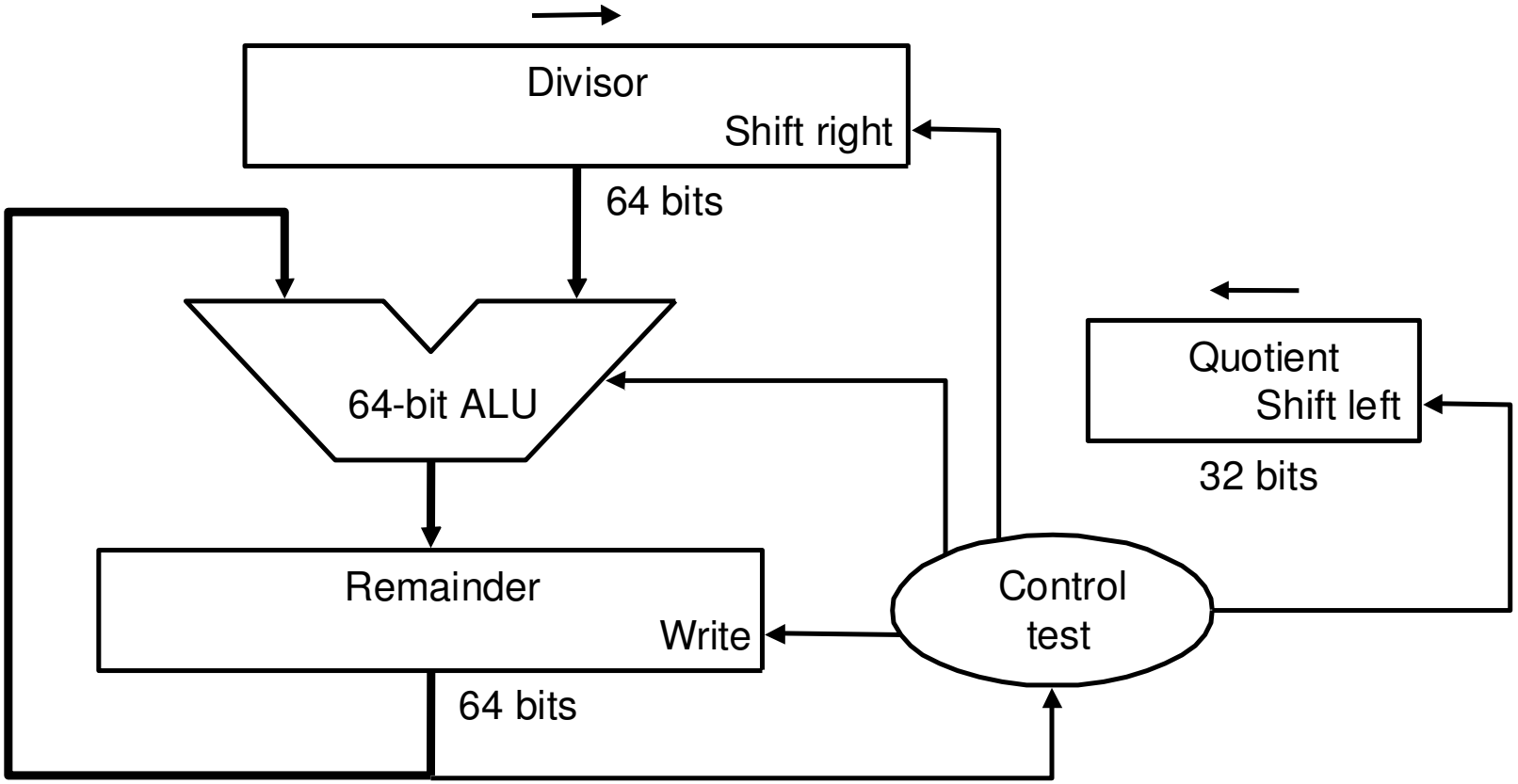
16 - 8 + 4 - 2 - 1

$$\ldots 1\ \bar{1} \ldots = 2^n - 2^{(n-1)} = 2^{(n-1)}(2-1) = 2^{(n-1)}$$

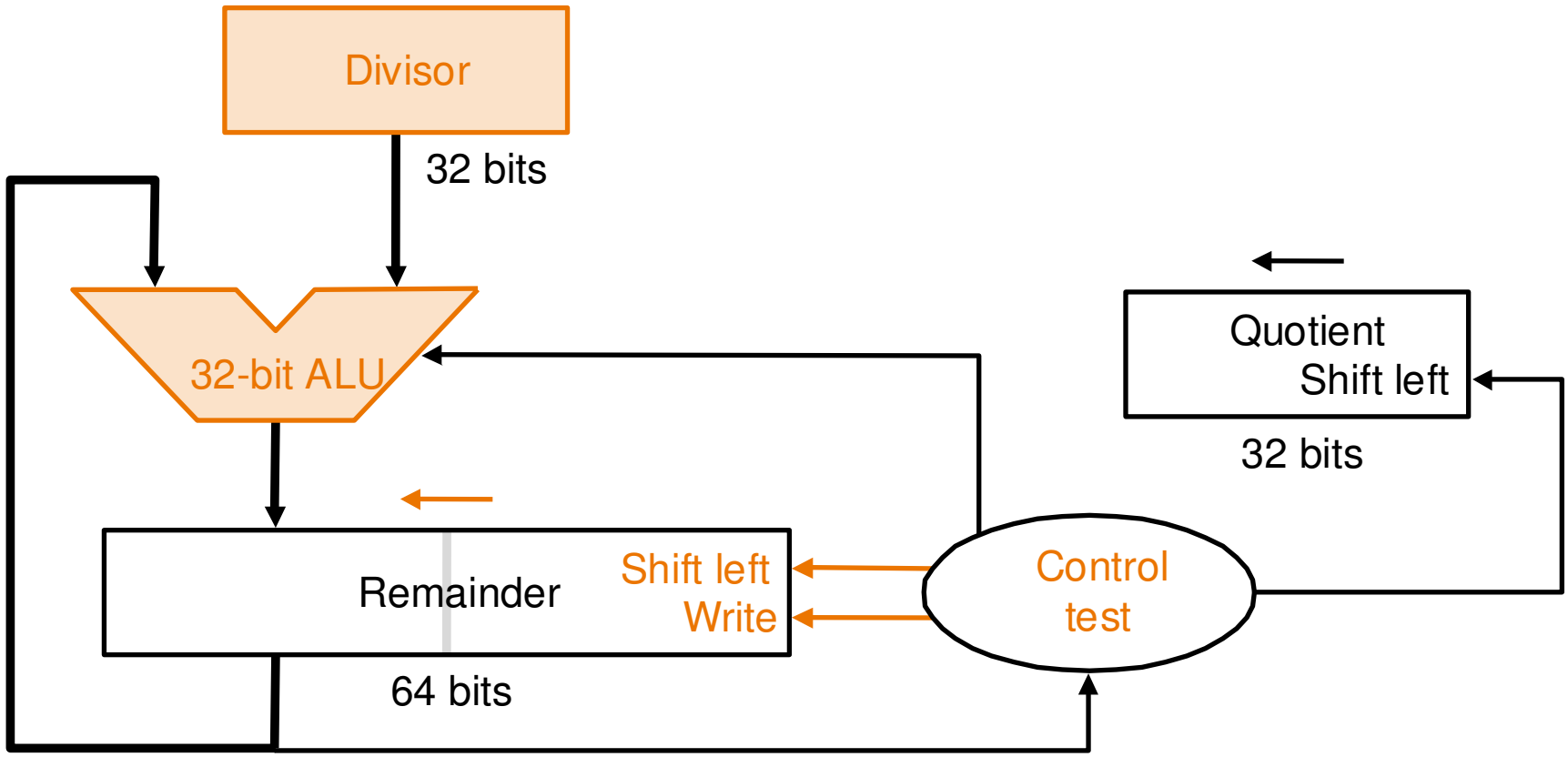
$$\begin{array}{r} 1\ \bar{1}\ 1\ \bar{1}\ \bar{1} \\ 0\ 1\ 1\ \bar{1}\ \bar{1} \\ 0\ 1\ 0\ 1\ \bar{1} \\ 0\ 1 \end{array}$$

- **Nº de somas: 3**
- **Nº de subtrações:2**
- **Total: 5**
- **OBS: se resto < 0 deve haver correção de um divisor para que resto > 0**

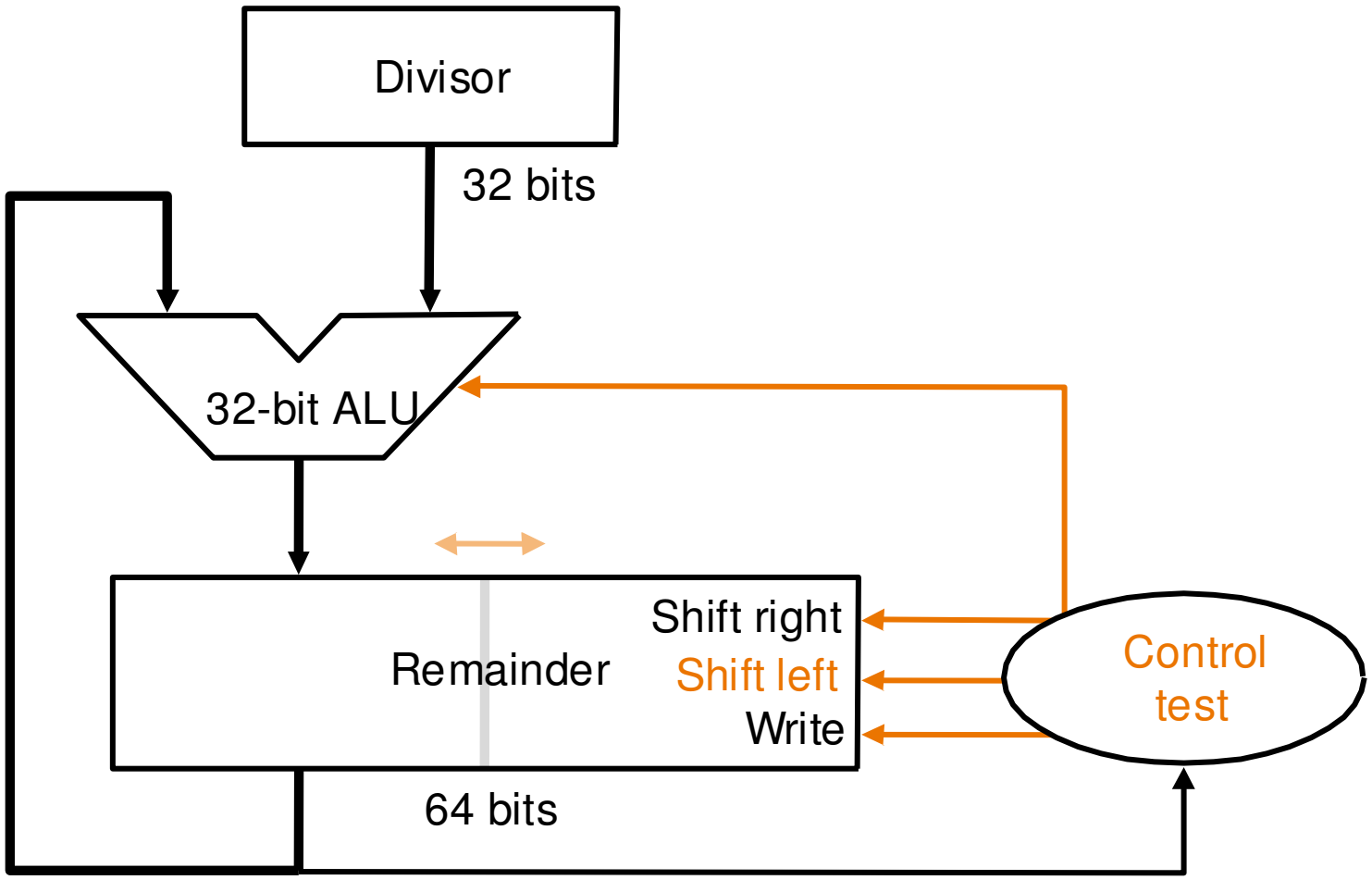
Divisor - Primeira Versão



Divisor - Segunda Versão



Divisor - Versão Final



Sistemas Numéricos

- Que tipo de números queremos representar usando uma representação binária?
  - Números Positivos
    - » Unsigned binary
  - Números Negativos
    - » Complemento de dois
    - » Sinal/magnitude
- E frações?

Números Fracionários

- Duas notações são comuns:
  - Ponto Fixo (Fixed-point):
    - » o ponto binário é fixo
  - Ponto Flutuante (Floating-point):
    - » O ponto binário flutua a direita do bit mais significativo

- Já visto

Ponto Flutuante - Arredondamento

- Overflow: quando o valor é muito grande para ser representado
- Underflow: quando o valor é muito pequeno para ser representado
- Modos de Aredondamento:
  - Down
  - Up
  - Toward zero
  - To nearest

Internamente, o processador armazena os números de ponto flutuante com ao menos 2 bits a mais: *guard* e *round*. Um terceiro bit, o *sticky* indica se algum conteúdo significativo foi perdido em arredondamento

- Examplos: arredondar 1.100101 (1.578125) de forma ser representado com 3 bits de fração.
  - Down: 1.100
  - Up: 1.101
  - Toward zero: 1.100
  - To nearest: 1.101 (1.625 é mais próximo de 1.578125 do que 1.5)

Ponto Flutuante: Adição

1. Extrair os bits do expoente e da fração
2. Adicionar o bit 1 à mantissa (prefixo)
3. Compar os expoentes
4. Deslocar a menor mantissa, se necessário
5. Somar as mantissas
6. Normalizar a mantissa e ajustar o expoente, se necessário
7. Arredondar o resultado
8. Montar o expoente e a fração no formato de ponto flutuante

Ponto Flutuante: Exemplo de Adição

Somar os seguintes números em ponto flutuante :

0x3FC00000

0x40500000

1. Extrair o expoente e a fração

1 bit	8 bits	23 bits
0	01111111	100 0000 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

1 bit	8 bits	23 bits
0	10000000	101 0000 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

Para o primeiro número (N1): S = 0, E = 127, F = .1

Para o segundo número (N2): S = 0, E = 128, F = .101

Ponto Flutuante: Exemplo de Adição

2. Adicionar o bit 1 à mantissa (prefixo)

N1: 1.1
N2: 1.101

3. Compar os expoentes

127 – 128 = -1, shift N1 para a direita de 1 bit

4. Deslocar a menor mantissa, se necessário

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

Ponto Flutuante: Exemplo de Adição

5. Somar as mantissas

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

6. Normalizar a mantissa e ajustar o expoente, se necessário

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

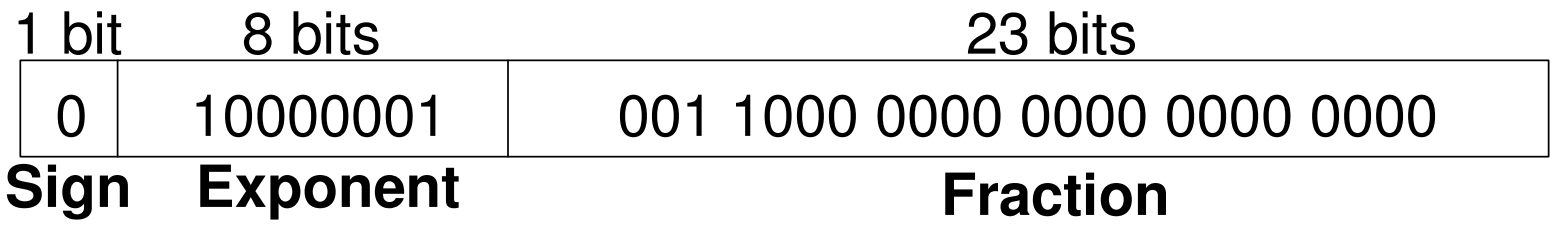
7. Arredondar o resultado

Não é necessário (cabe em 23 bits)

8. Montar o expoente e a fração no formato de ponto flutuante

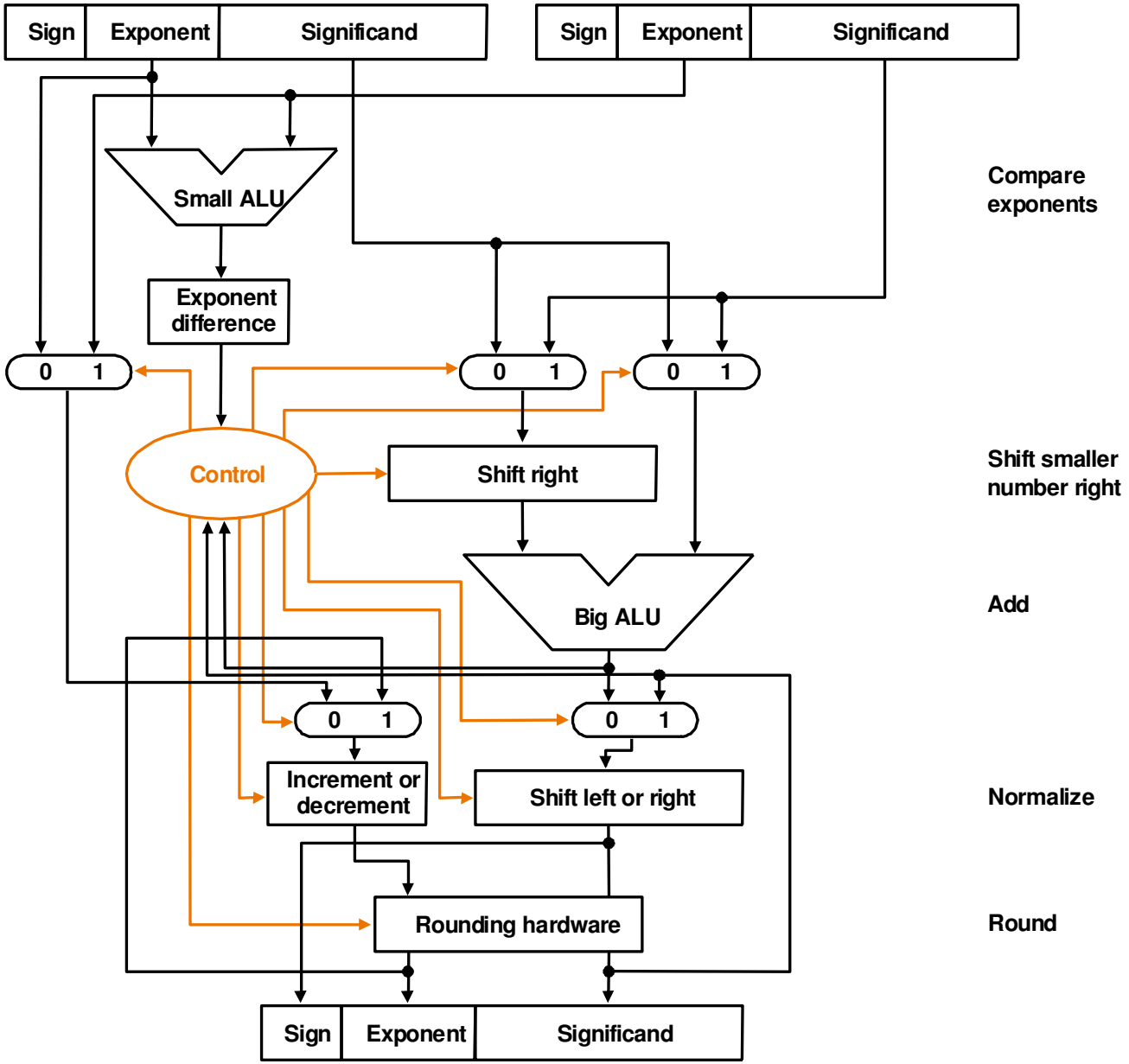
$S = 0$, $E = 2 + 127 = 129 = 10000001_2$, $F = 001100..$

Ponto Flutuante: Exemplo de Adição



Em hexadecimal: 0x40980000

Hardware de Adição de Ponto Flutuante



Floating-point numbers

0	10000001	111 1100 0000 0000 0000 0000
0	01111100	100 0000 0000 0000 0000 0000

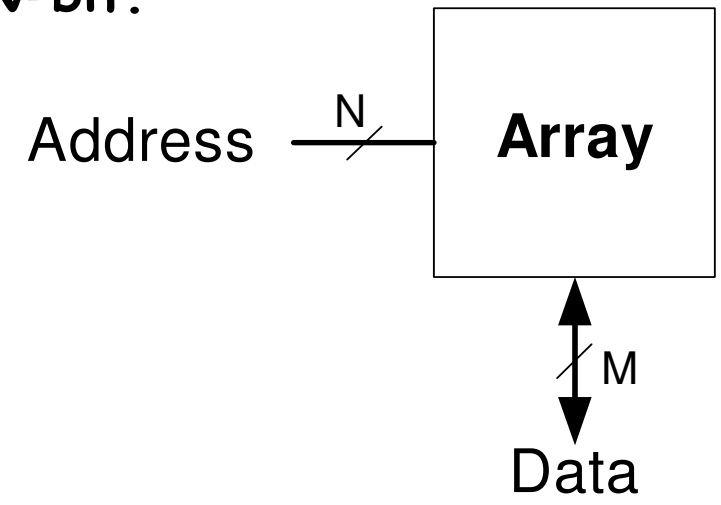
	Exponent	Fraction
Step 1	10000001	111 1100 0000 0000 0000 0000
	01111100	100 0000 0000 0000 0000 0000
Step 2	10000001	1.111 1100 0000 0000 0000 0000
	01111100	1.100 0000 0000 0000 0000 0000
Step 3	10000001	1.111 1100 0000 0000 0000 0000
	− 01111100	1.100 0000 0000 0000 0000 0000
	101 (shift amount)	
Step 4	10000001	1.111 1100 0000 0000 0000 0000
	10000001	0.000 0110 0000 0000 0000 0000 00000
Step 5	10000001	1.111 1100 0000 0000 0000 0000
	10000001	+ 0.000 0110 0000 0000 0000 0000
		10.000 0010 0000 0000 0000 0000
Step 6	10000001	10.000 0010 0000 0000 0000 0000 >> 1
	+ 1	
	10000010	1.000 0001 0000 0000 0000 0000
Step 7	(No rounding necessary)	
Step 8	0 10000010	000 0001 0000 0000 0000 0000

Memórias

- Memory arrays – armazena eficientemente grande quantidadde de dados.

- Três tipos mais comuns de memory arrays:
  - Dynamic random access memory (DRAM)
  - Static random access memory (SRAM)
  - Read only memory (ROM)

- Um dado de valor de *M*-bit pode ser lido ou escrito por vez em um endereço de *N*-bit.

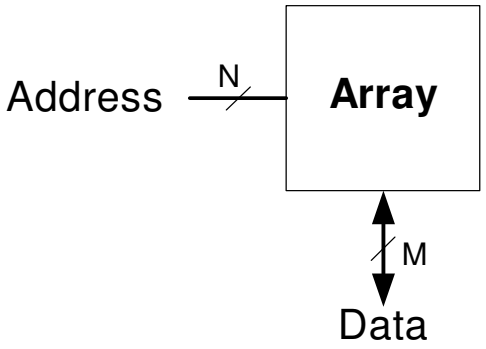


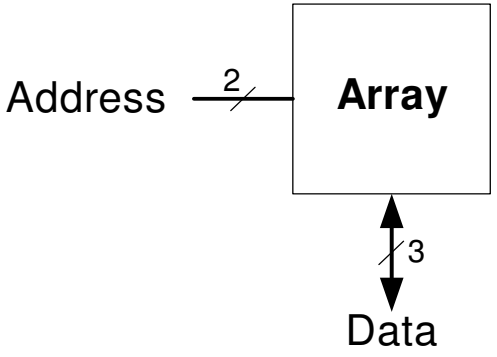
Memórias

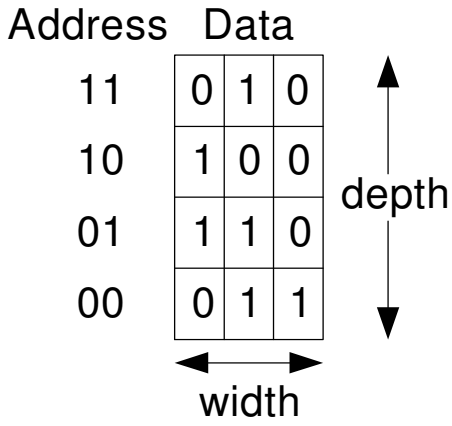
- As memórias são organizadas como um array de duas dimensões de células de bits. Cada celula armazenaum bit.

- Um array com *N* bits de endereço e dados de *M* bits tem 2^N linhas e *M* colunas. Each row of data is called a word.

  - Depth: número de linhas do array
  - Width: número de colunas no array (word size)
  - Array size: dado por depth × width



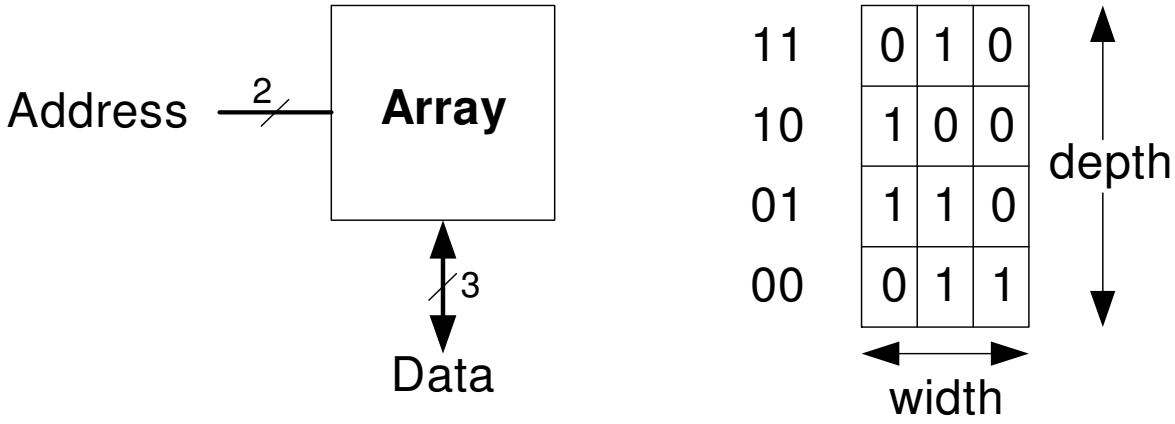




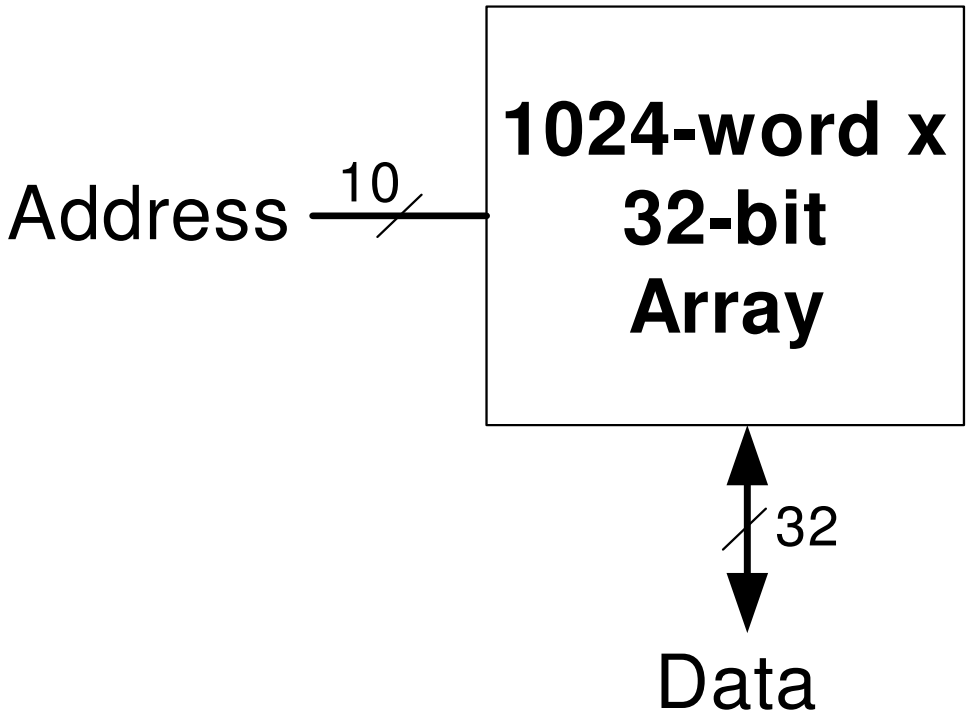
Memoria : Exemplo

- O array abaixo é um array d2^2 × 3-bit.
- Word size de 3-bits.

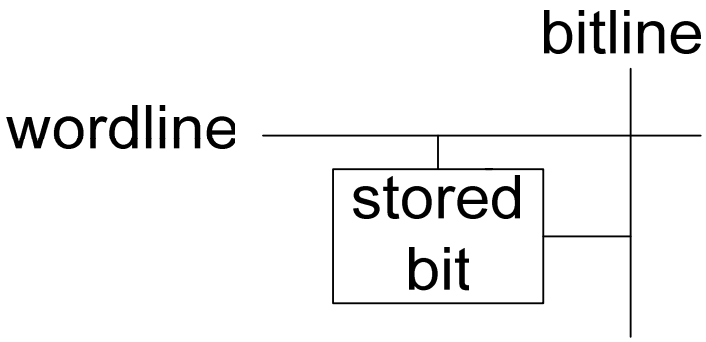
Exemplo:



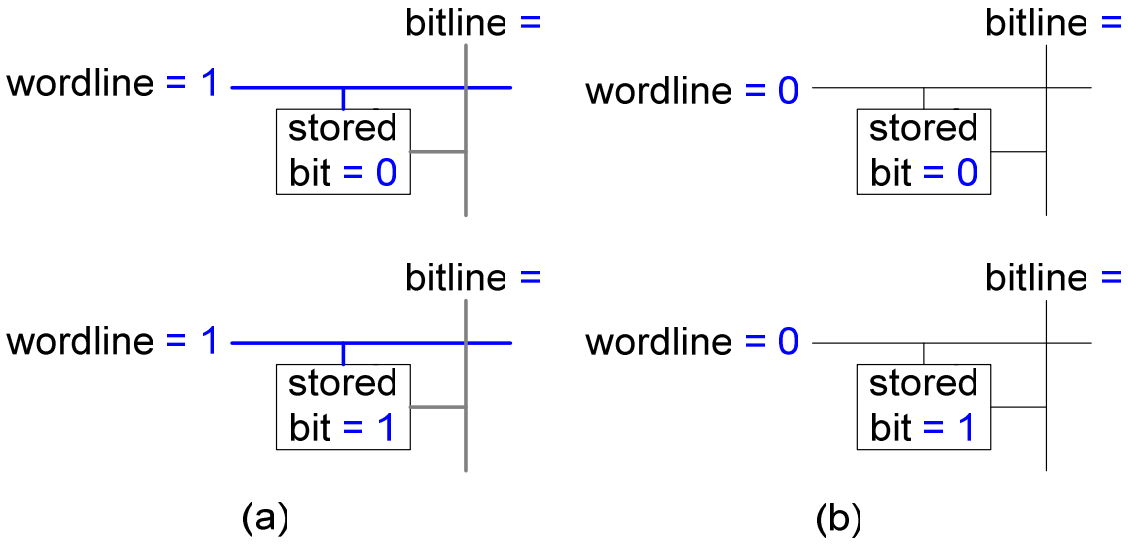
Memória : Exemplo



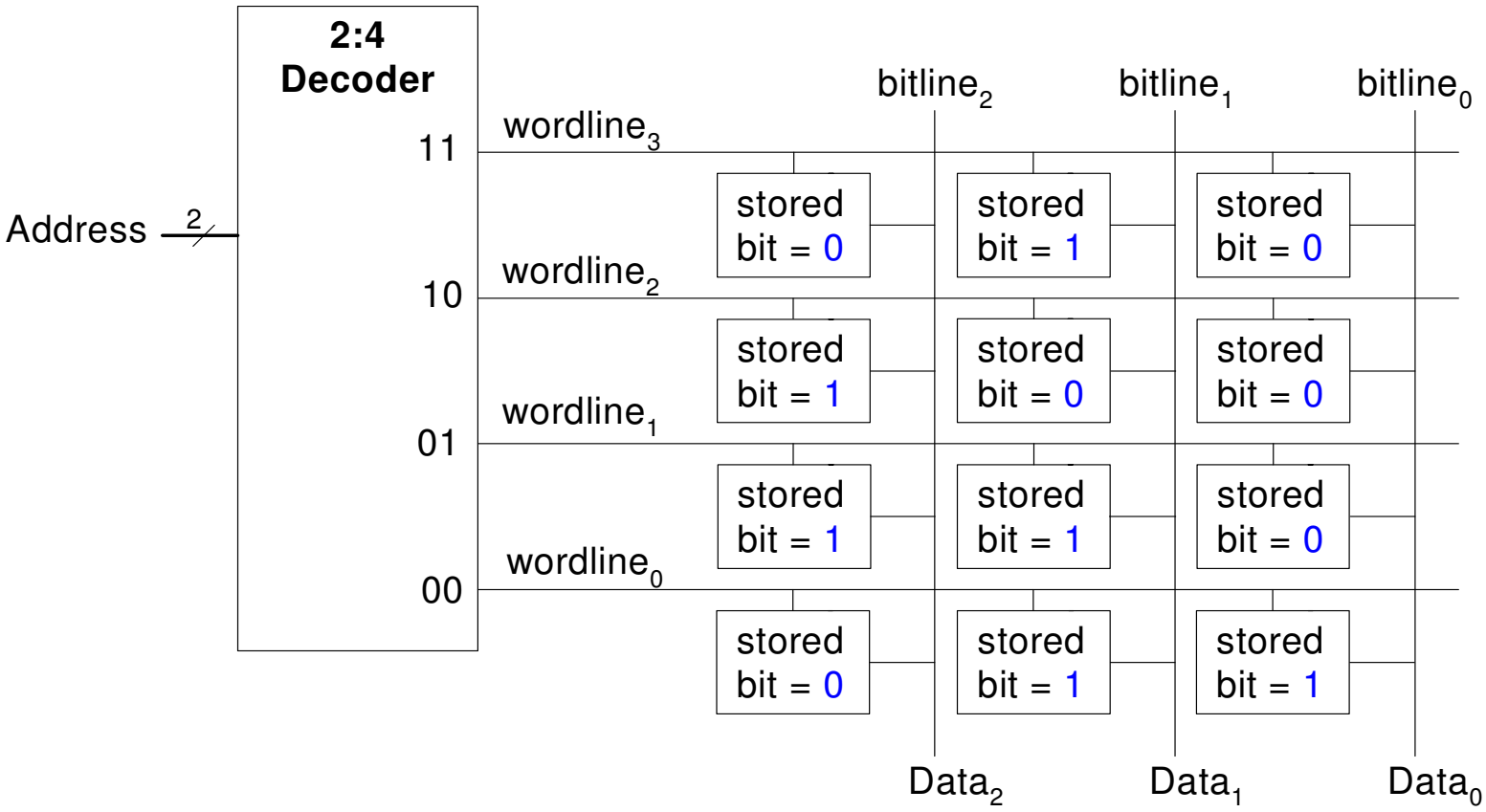
Memória: Célula de bit



Example:



Memória: 4x3


2:4
Decoder
Address
2
11
10
01
00
wordline3
wordline2
wordline1
wordline0
bitline2
bitline1
bitline0
stored bit = 0
stored bit = 1
stored bit = 0
stored bit = 1
stored bit = 0
stored bit = 0
stored bit = 1
stored bit = 1
stored bit = 0
stored bit = 0
stored bit = 1
stored bit = 1
Data2
Data1
Data0

Tipos de Memórias

- Random access memory (RAM): volátil
- Read only memory (ROM): não volátil

RAM

- Random access memory
  - Volátil: perde o dado quando a alimentação é desligada
  - Pode ser lida ou escrita rapidamente
  - A memória principal do seu computador é RAM (specificamente, DRAM)
  - Historicamente denominada de *random* access memory porque qualquer palavra de dado pode ser acessada como qualque outra (em contraste com sequential access memories como fita magnética).

ROM

- Read only memory (ROM)
  - Não volátil: não perdem seus dados quando a alimentação é desligada
  - Pode ser lida rapidamente, porém a escrita é lenta
  - Memórias em câmeras digitais, pen drives são ROMs
  - Historicamente denominadas de *read only* memory porque ROMs eram escritas queimando-se fusíveis. Uma vez que os fusíveis eram escritos não podiam ser escritos novamente.

Tipos de RAM

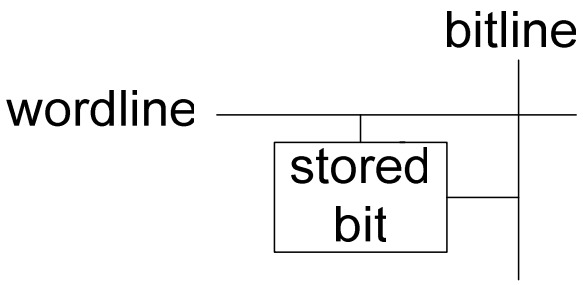
- Os dois tipos de RAM são:
  - Dynamic random access memory (DRAM)
  - Static random access memory (SRAM)

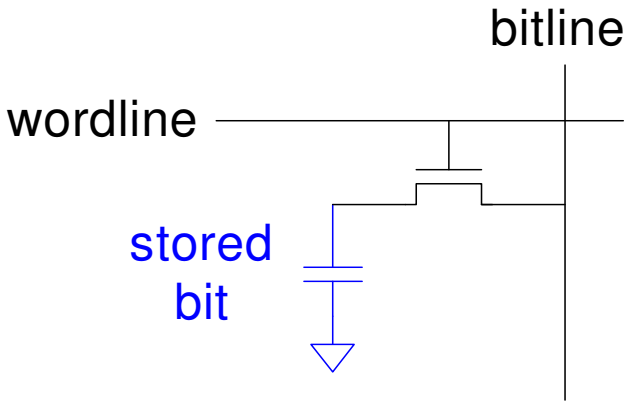
- A diferença é como armazenam os dados:
  - DRAM usa um capacitor
  - SRAM usa cross-coupled inverters ("latch")

DRAM

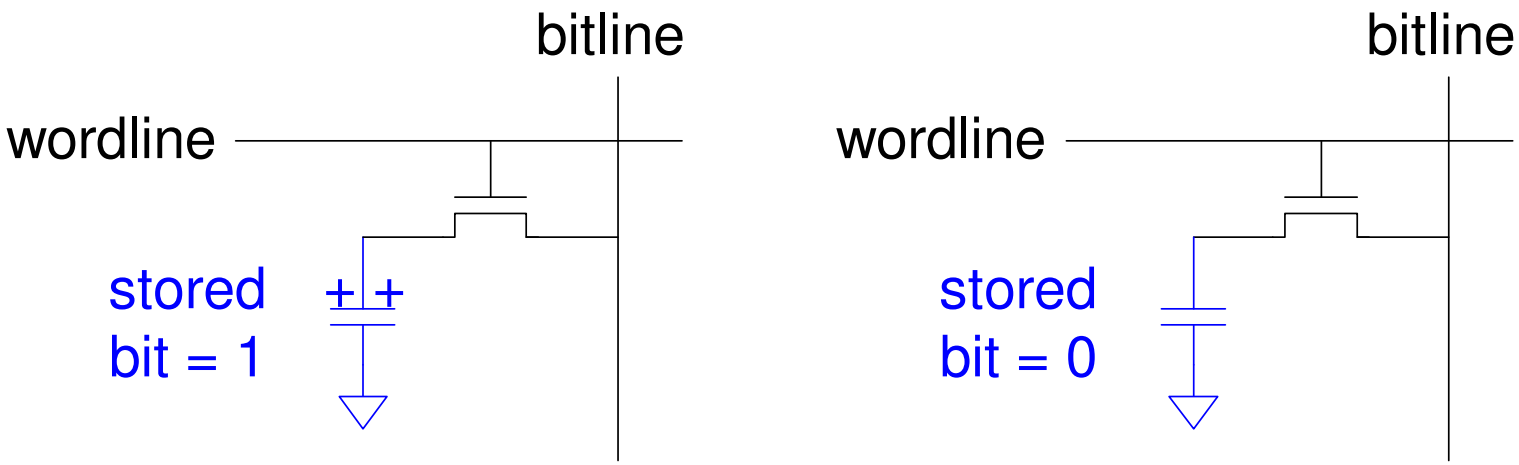
- Data bits são armazenados em um capacitor.

- DRAM denominado de *dynamic* porque os valores necessitam ser reescritos (refreshed) periodicamente e após serem lidos por que:
  - A corrente de fuga do capacitor degrada o valor
  - A leitura destroi o valor armazenado

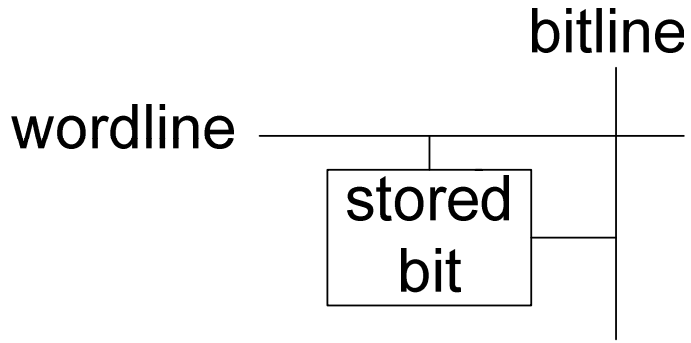


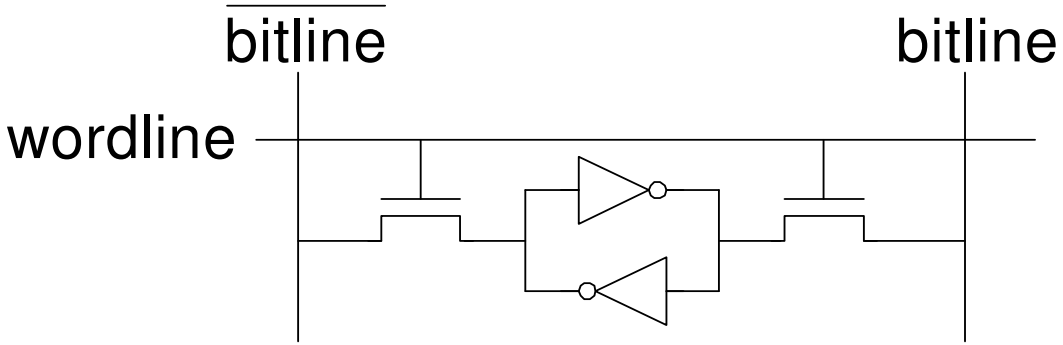


DRAM

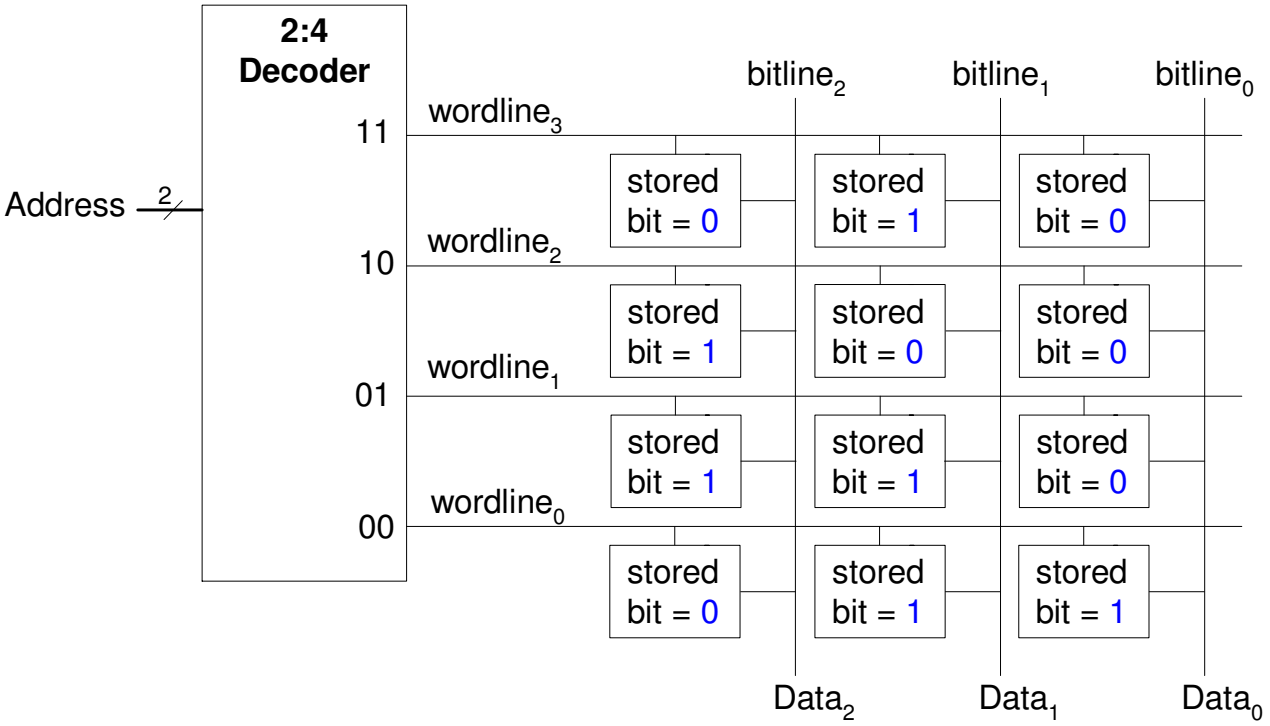

bitline
wordline
stored
bit = 1
+ +
bitline
wordline
stored
bit = 0

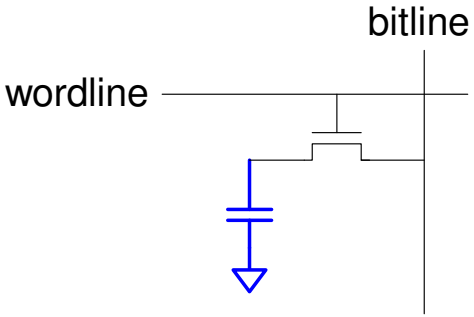
SRAM

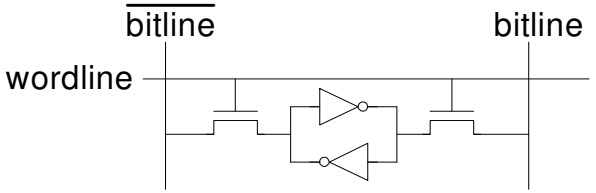




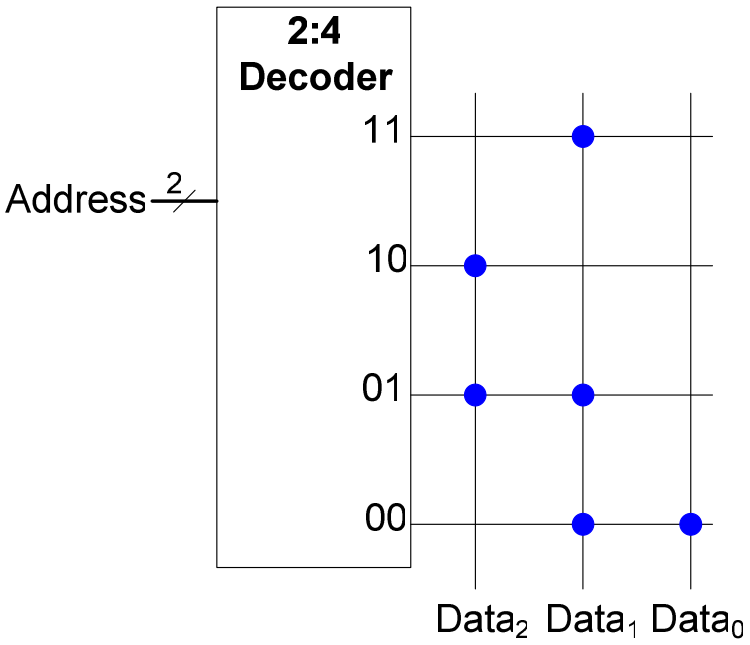
Memória

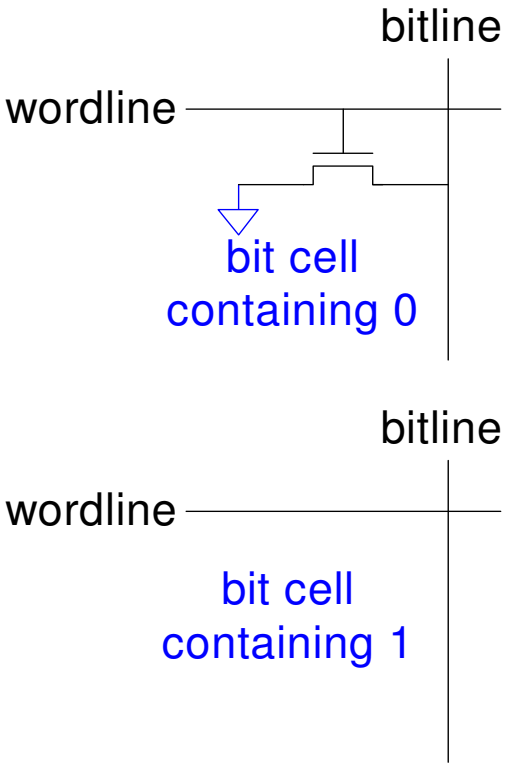

2:4
Decoder
Address
2
11
10
01
00
wordline3
wordline2
wordline1
wordline0
bitline2
bitline1
bitline0
stored bit = 0
stored bit = 1
stored bit = 0
stored bit = 1
stored bit = 0
stored bit = 0
stored bit = 1
stored bit = 1
stored bit = 0
stored bit = 0
stored bit = 1
stored bit = 1
Data2
Data1
Data0


bitline
wordline

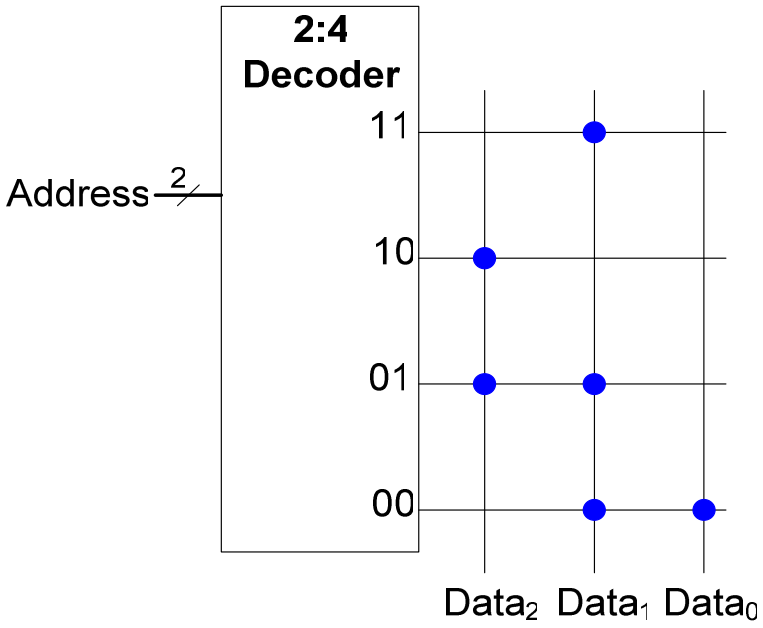

bitline
bitline
wordline

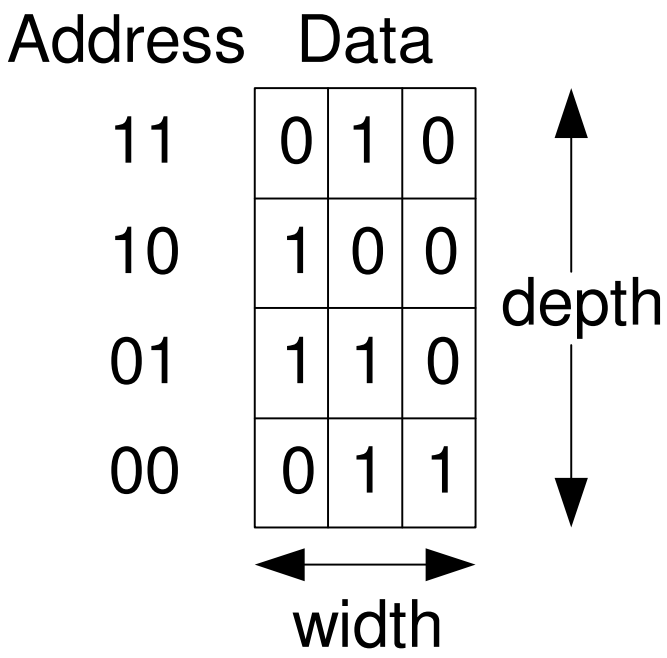
ROM


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

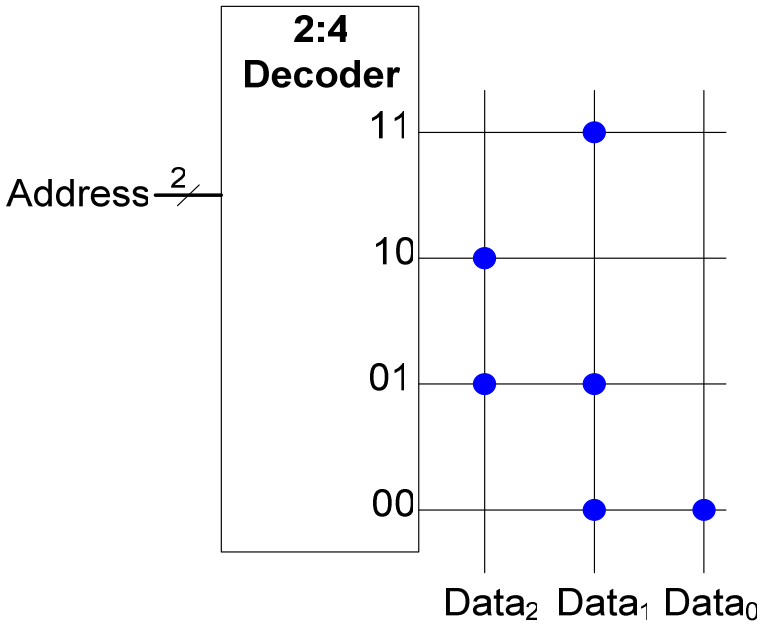

bitline
wordline
bit cell
containing 0
bitline
wordline
bit cell
containing 1

ROM


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


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

Lógica com ROM



$$Data_2 = A_1 \oplus A_0$$

$$Data_1 = A_1 + A_0$$

$$Data_0 = A_1 A_0$$

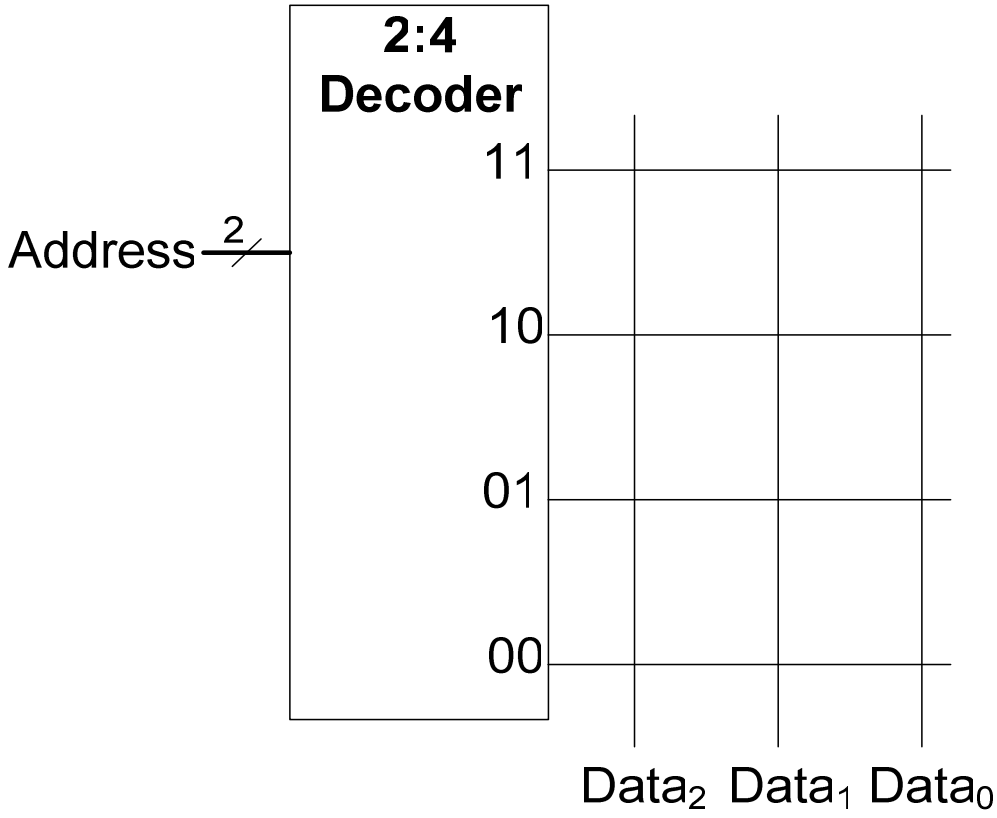
Lógica com ROM
Exemplo

- **Implemente as seguintes funções lógica usando uma ROM $2^2 \times 3$-bit:**

$X = AB$

$Y = A + B$

$Z = AB$



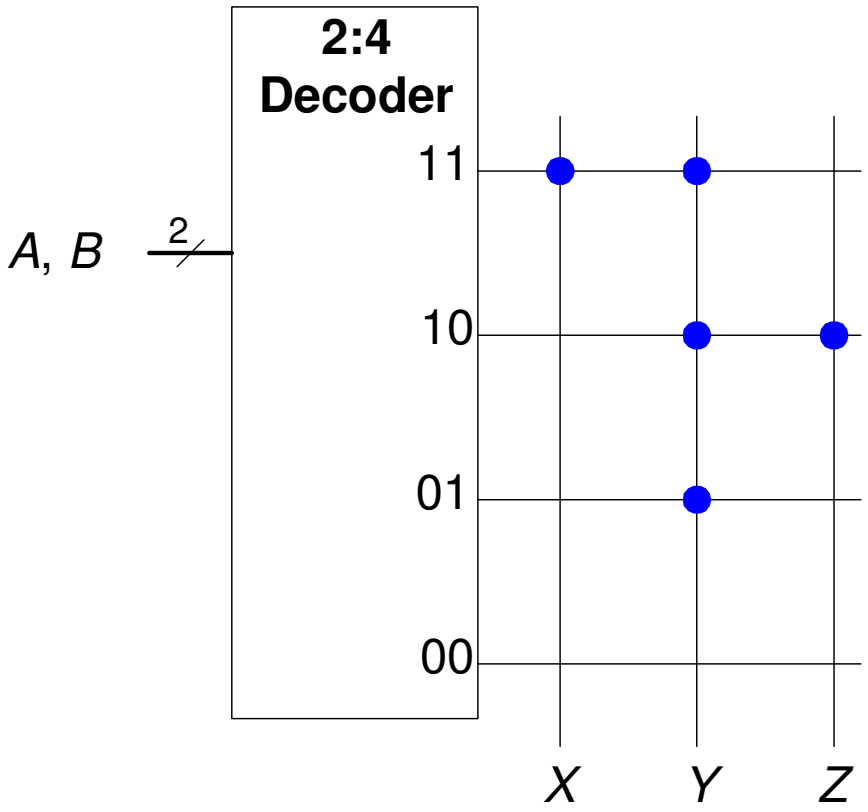
Lógica com ROM
Exemplo

- **Implemente as seguintes funções lógica usando uma ROM $2^2 \times 3$-bit:**

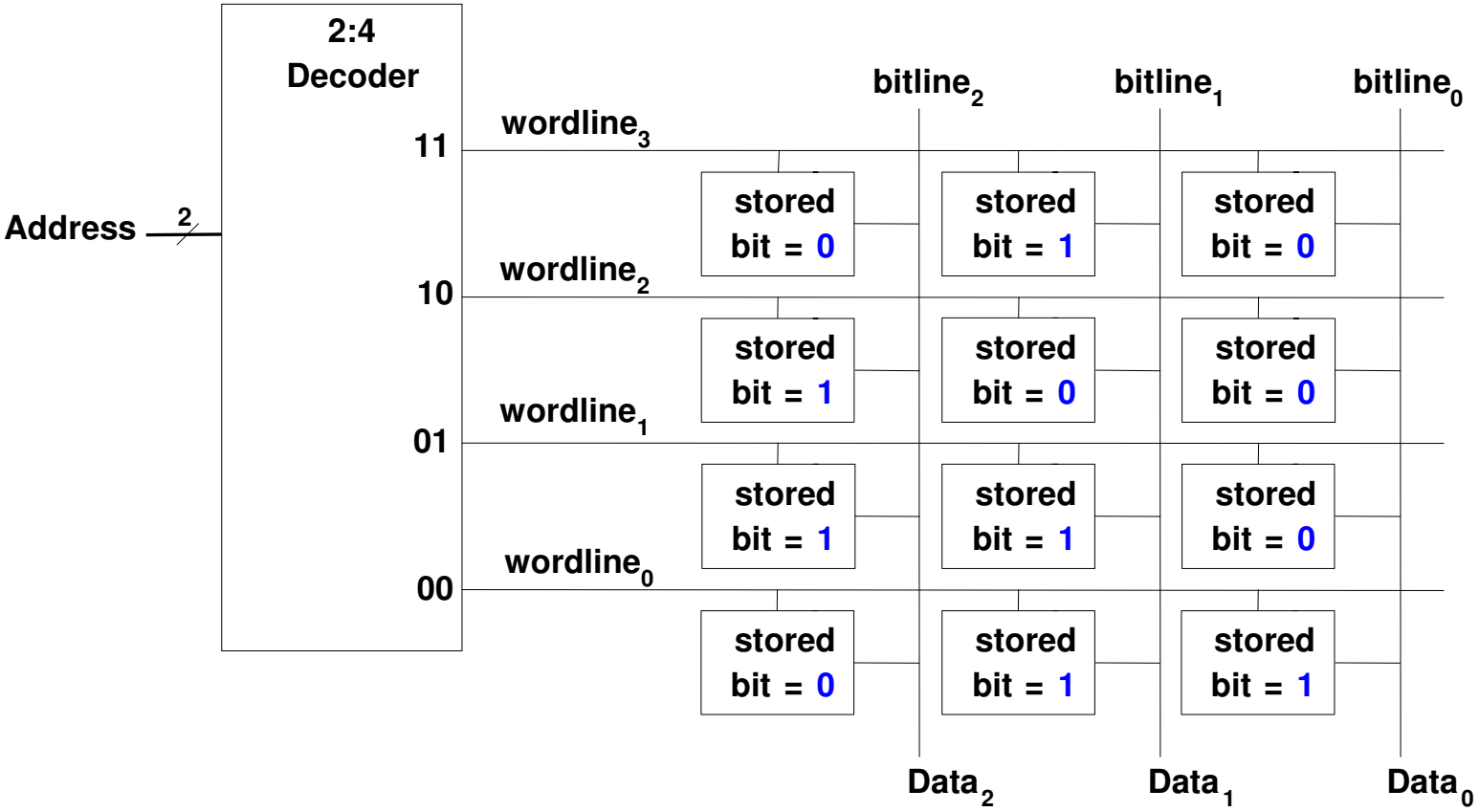
$X = AB$

$Y = A + B$

$Z = AB$



Lógica com Memória



$$Data_2 = A_1 \oplus A_0$$

$$Data_1 = A_1 + A_0$$

$$Data_0 = A_1 A_0$$

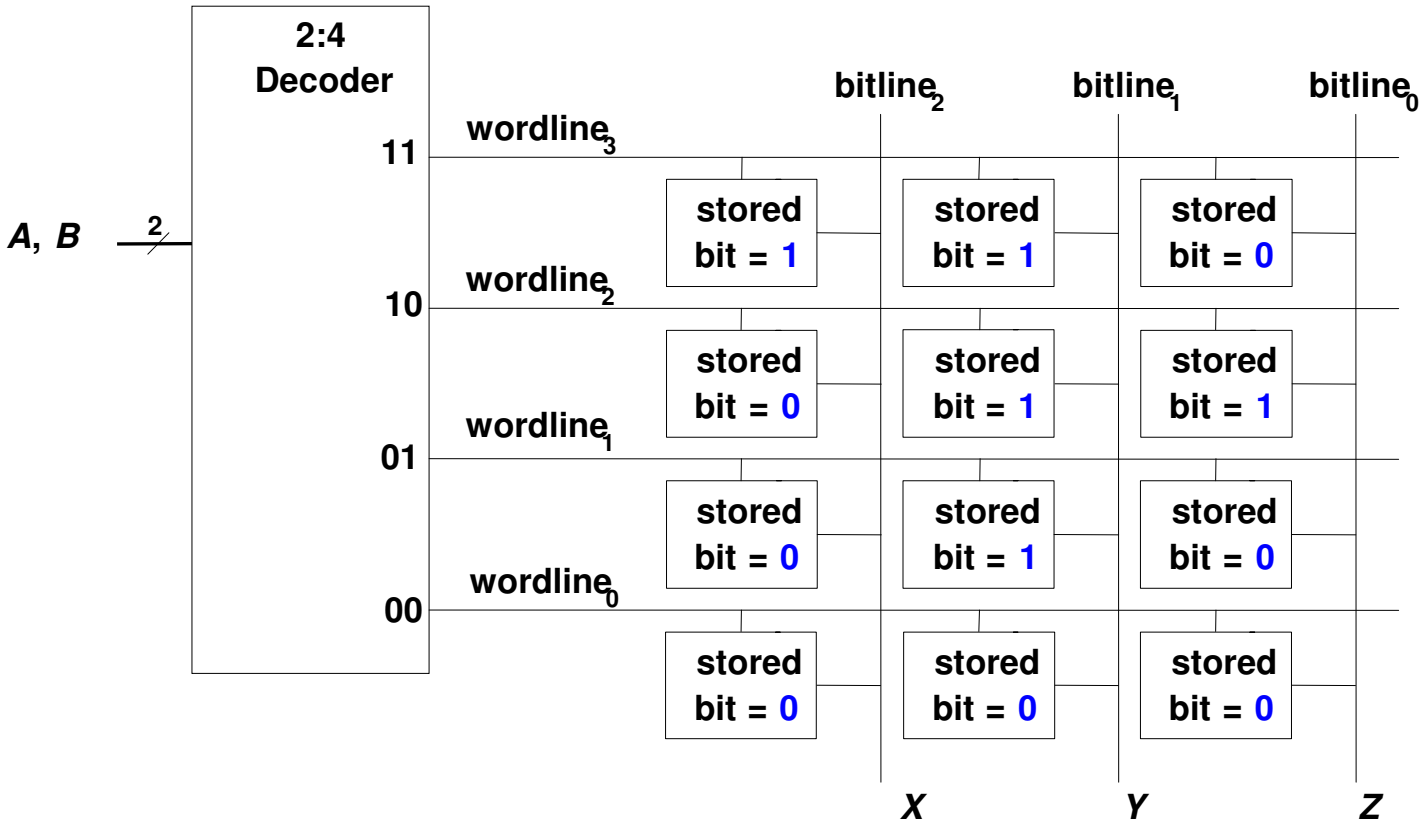
Lógica com Memória

- Implemente as seguintes funções lógicas com uma memória $2^2 \times 3$-bit:

  $X = AB$

  $Y = A + B$

  $Z = AB$



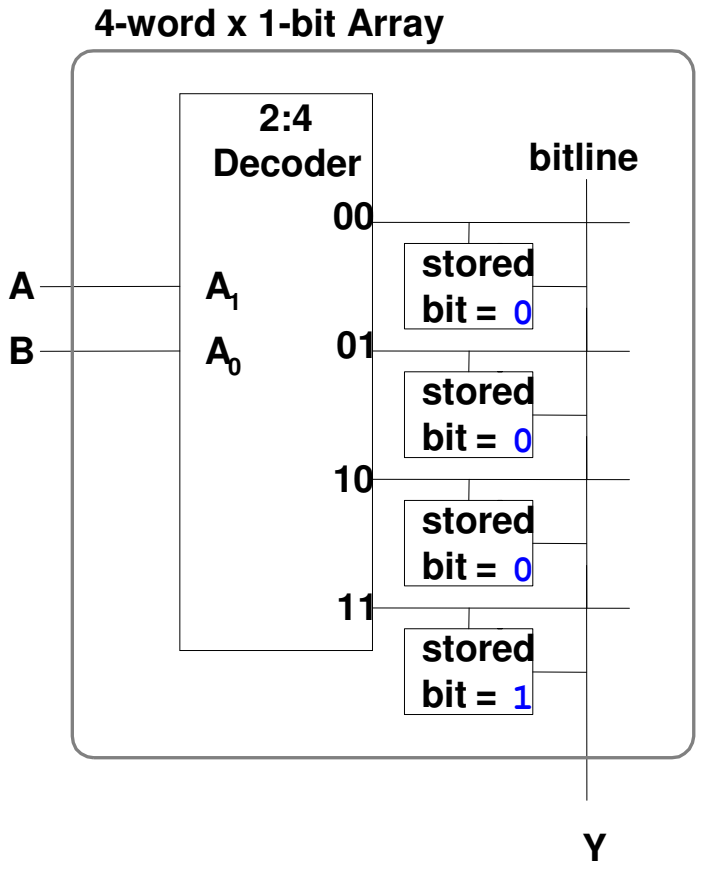
Lógica com Memória

- A memória usada para executar funções lógicas é denominadad *lookup tables* (LUT).

- O usuário tem o valor de saída para cada combinação das entradas (address).

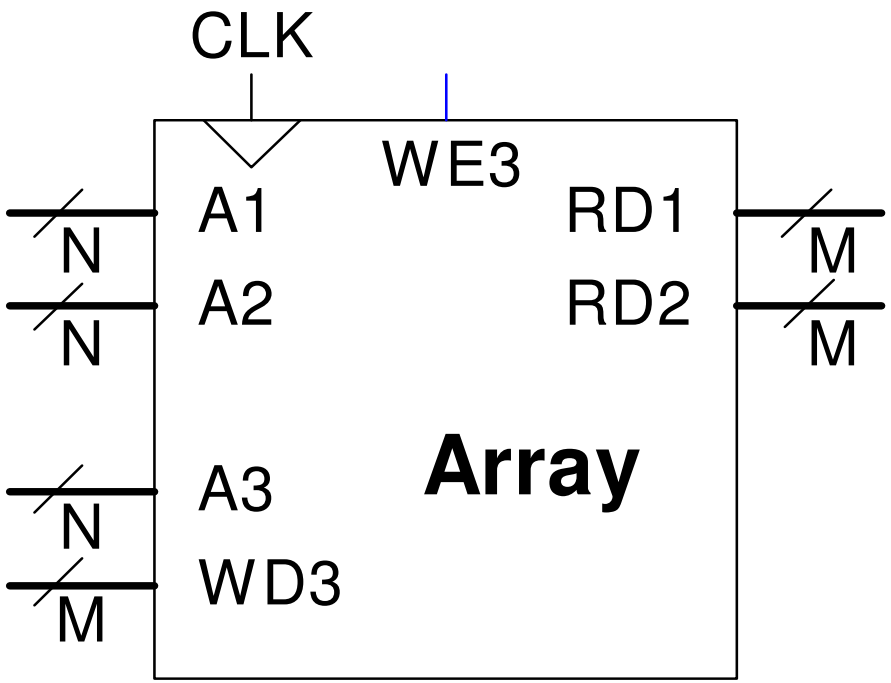
Truth Table

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



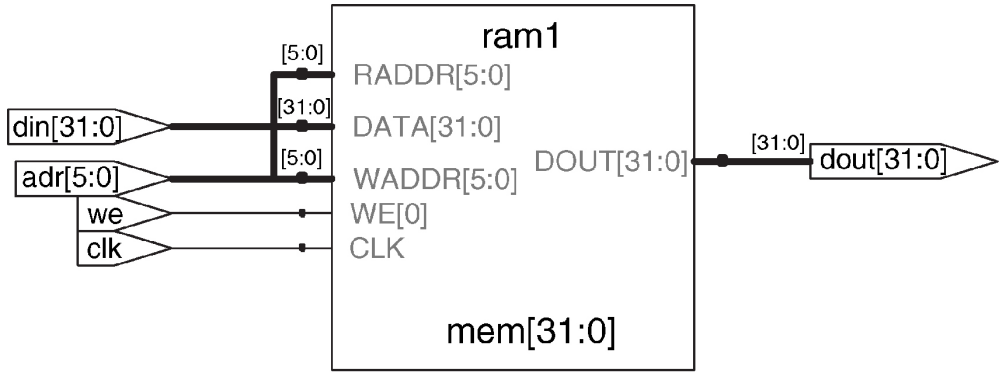
Memórias Multi-Portas

- Porta: par endereço/dado (address/data)
- Memória 3-portas
  - 2 portas de leitura (A1/RD1, A2/RD2)
  - 1 porta de escrita (A3/WD3, WE3 enables writing)



Memória - VHDL

```
Library IEEE;
Use IEEE.STD_LOGIC_ARITH.ALL;
Use IEEE.STD_LOGIC_UNSIGNED.ALL;
```




```
Entity ram Is
  Generic (N:Integer := 6; M: Integer := 32);
  Port (clk,
        we   :  In STD_LOGIC;
        addr :  In STD_LOGIC_VECTOR(N-1 Downto 0);
        din  :  In STD_LOGIC_VECTOR(M-1 Downto 0);
        dout :  In STD_LOGIC_VECTOR(M-1 Downto 0));
End;
```

Memória - VHDL

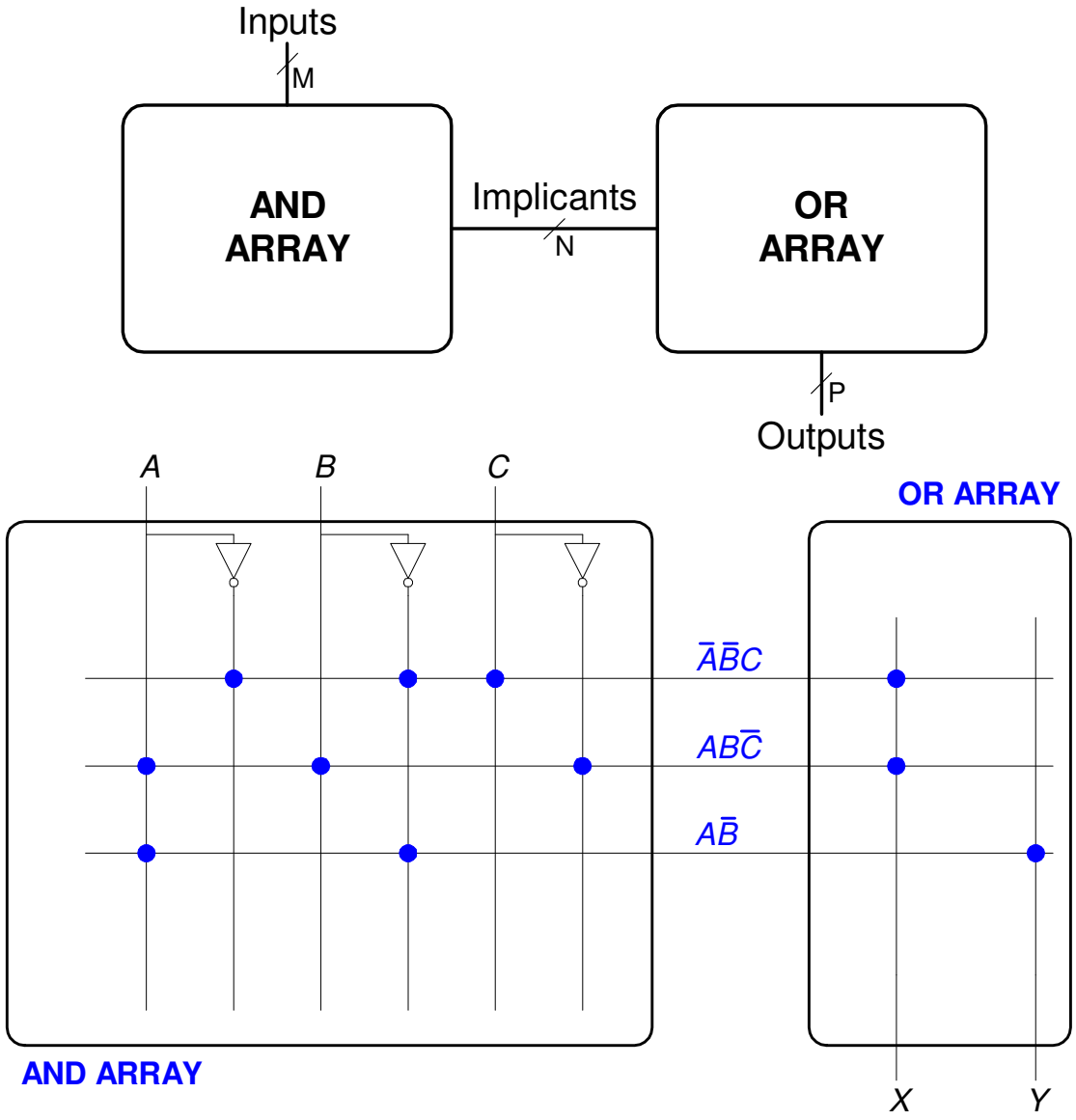
```
Architecture synth Of ram Is
  Type mem_ram Is Array ((2**N-1) Downto 0)
        Of STD_LOGIC_VECTOR(M-1 Downto 0);
  Signal mem: mem_ram;


Begin
   Process (clk)
   Begin
      If clk'event and clk =  '1' Then
         If we = '1' Then
             mem(CONV_INTEGER(addr)) <= din;
         End If;
      End If;
    End Process;

    dout <= mem (CONV_INTEGER(addr));
End;
```

Logic Arrays

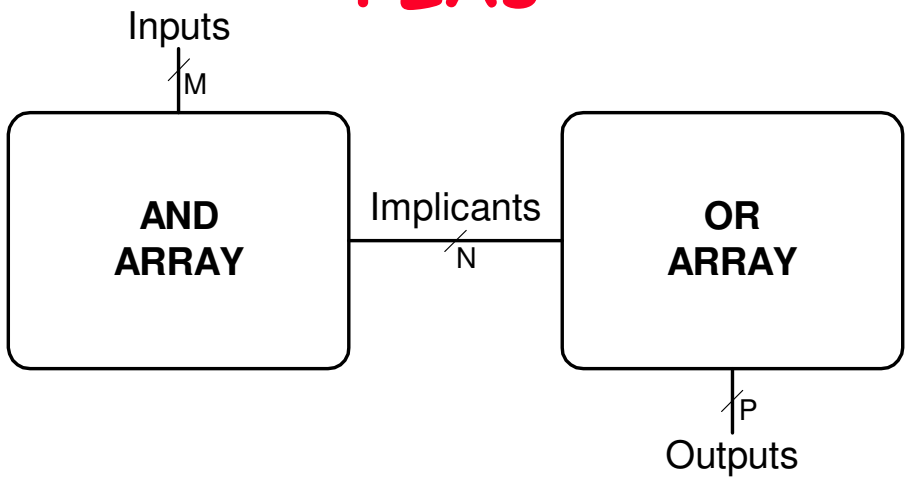
- Programmable Logic Arrays (PLAs)
  - Array de ANDs seguido por um array de ORs
  - Executa somente lógica combinacional
  - Conexões internas "fixas"

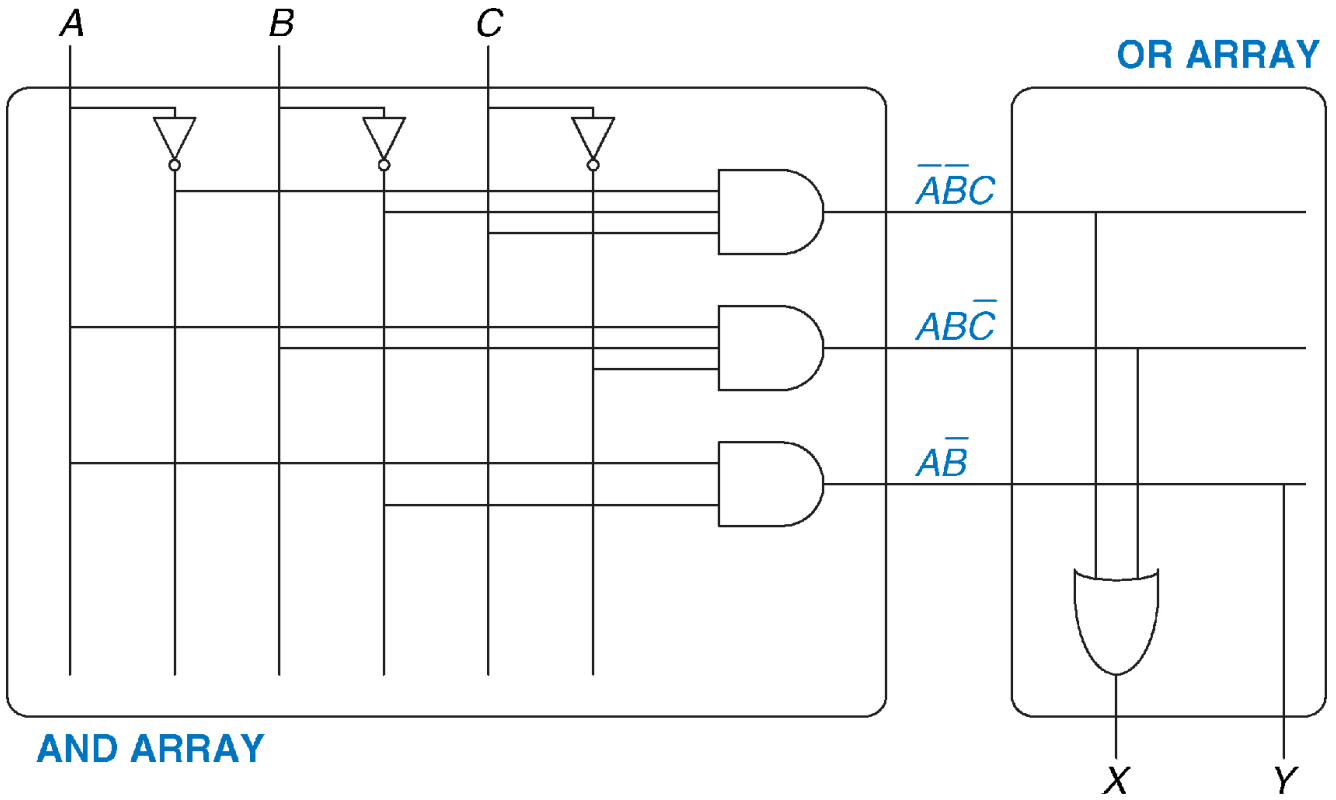
- Field Programmable Gate Arrays (FPGAs)
  - Array de blocos lógicos configuráveis (CLBs)
  - Executa lógica combinacional e seqüencial
  - Conexões internas programáveis

PLAs


Inputs
M
AND ARRAY
Implicants
N
OR ARRAY
P
Outputs
A
B
C
OR ARRAY
$\bar{A}\bar{B}C$
$AB\bar{C}$
$A\bar{B}$
AND ARRAY
X
Y

PLAs

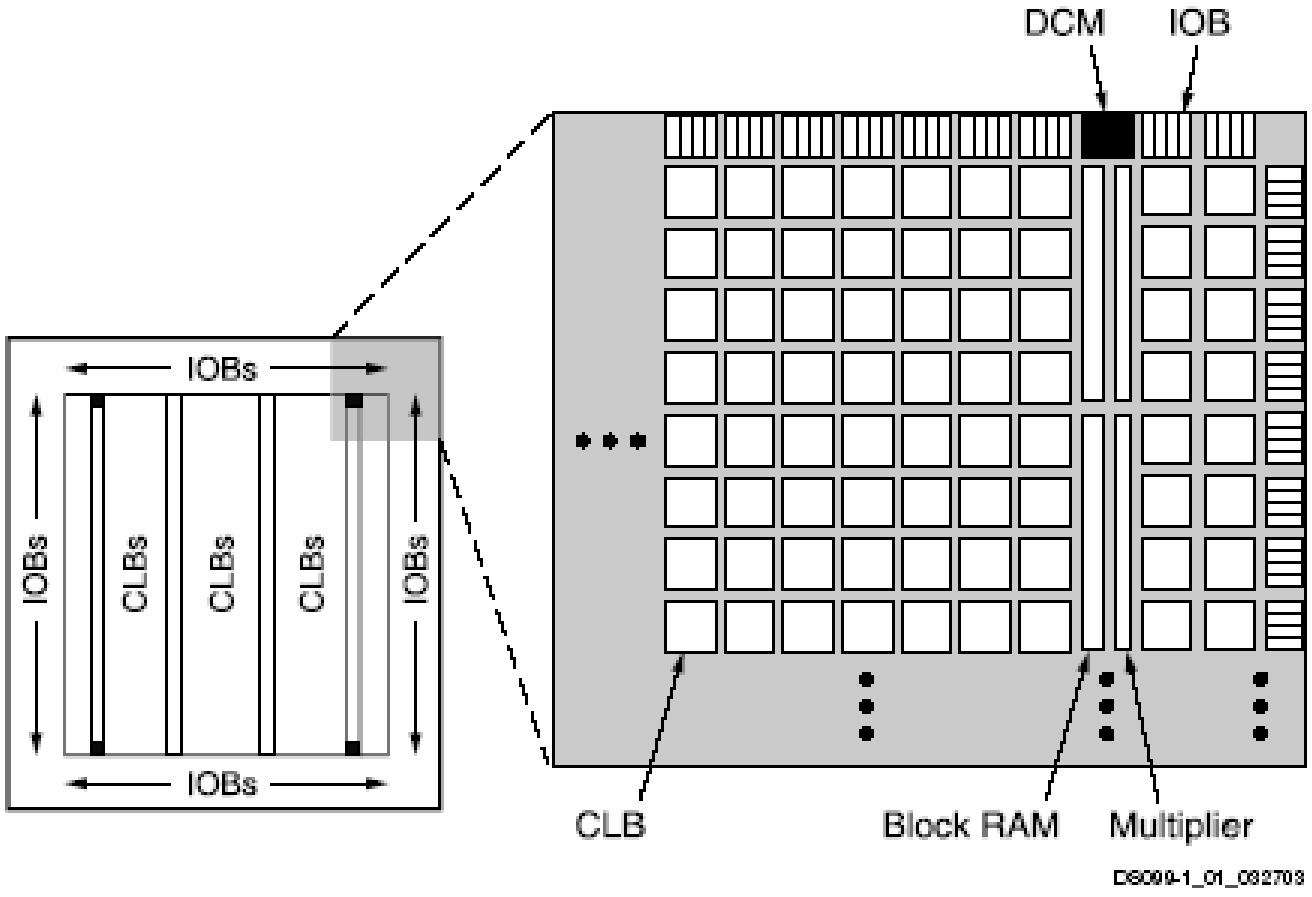




FPGAs

- Composto por:
  - CLBs (Configurable logic blocks): executa a lógica
  - IOBs (Input/output buffers): interface com o mundo exterior
  - Programmable interconnection: usado para conectar CLBs e IOBs
  - Algumas FPGAs incluem outros blocos funcionais como somadores, multiplicadores e RAMs

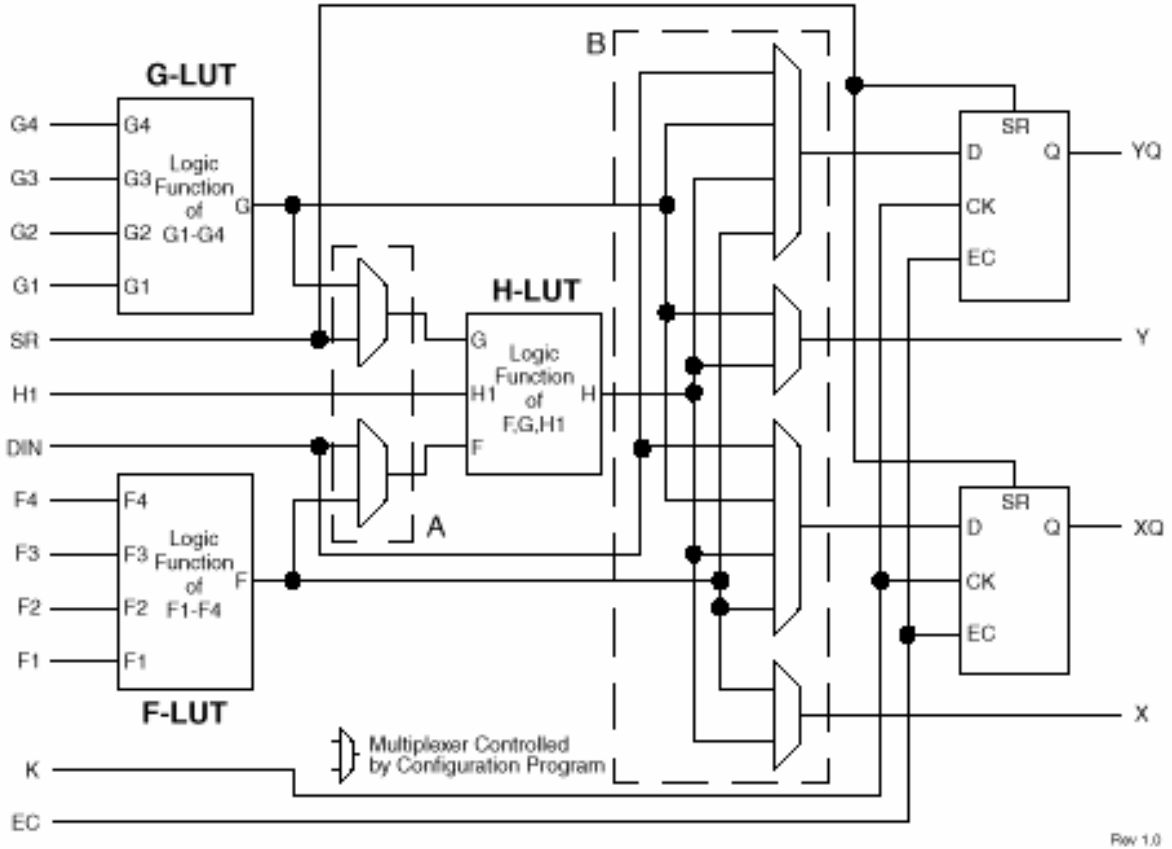
FPGA Xilinx Spartan 3



CLBs

- **Composto por:**
  - **LUTs (lookup tables): executa lógica combinacional**
  - **Flip-flops: executa funções seqüenciais**
  - **Multiplexers: conecta LUTs e flip-flops**

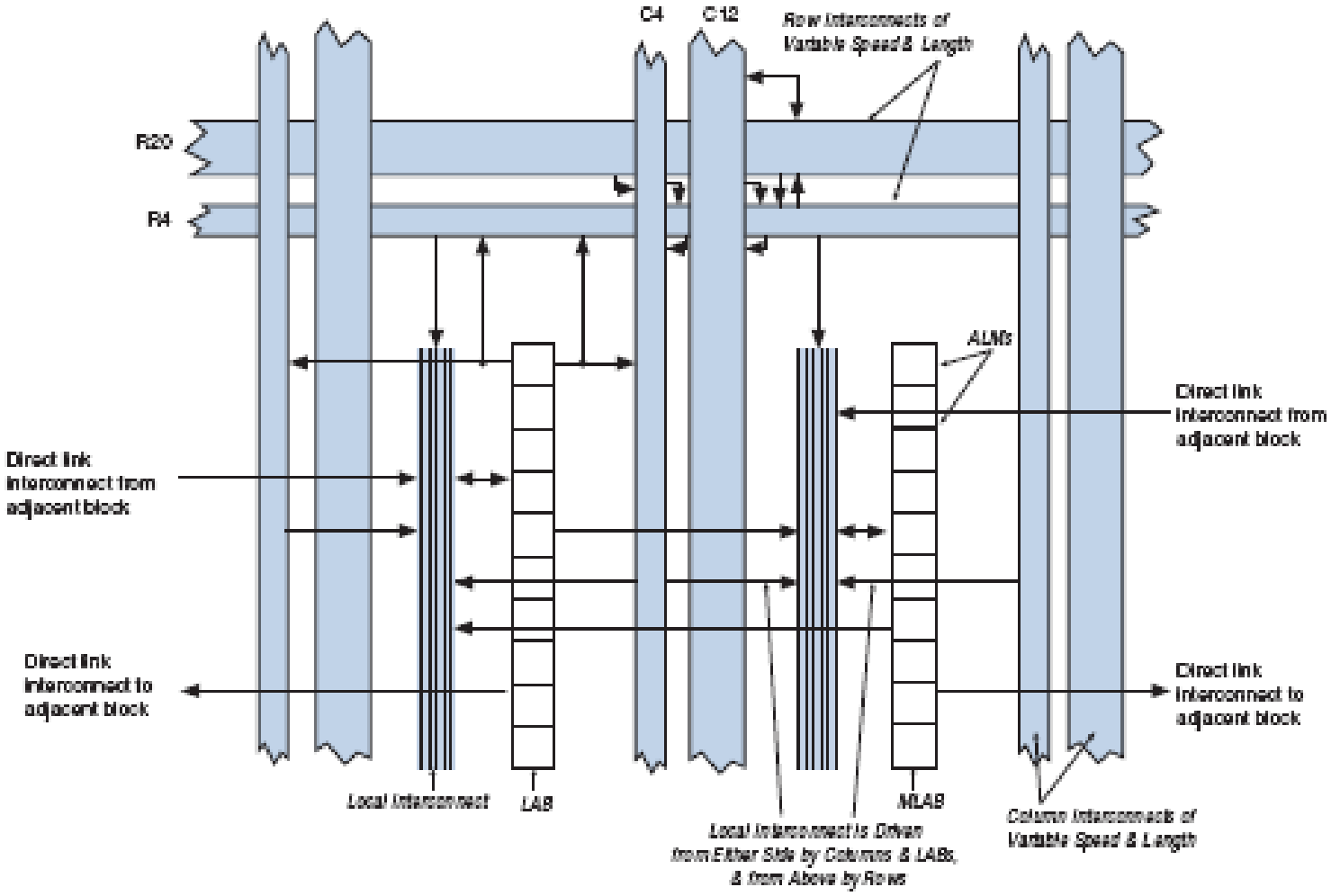
Xilinx Spartan CLB



FPGA Stratix III

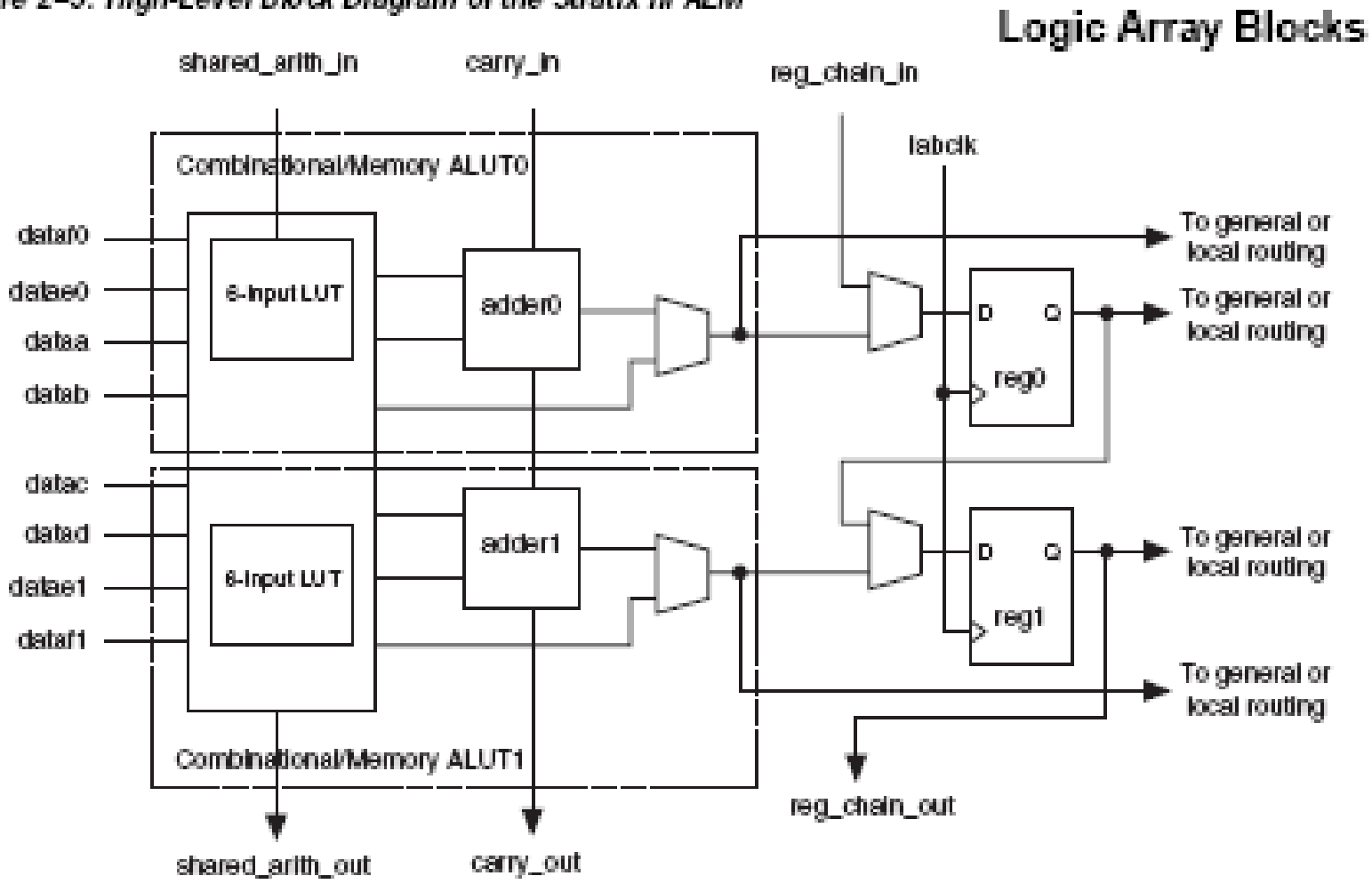
Logic Array Blocks

Figure 2–1. Stratix III LAB Structure



FPGA Stratix III

Figure 2–5. High-Level Block Diagram of the Stratix III ALM



FPGA Stratix III

Figure 2–6. Stratix III ALM Details