UFSJ
UNIVERSIDADE FEDERAL
DE SÃO JOÃO DEL-REI

Arquitetura de Computadores

Parte 2 - Cap. 5, 6 e Anexo C

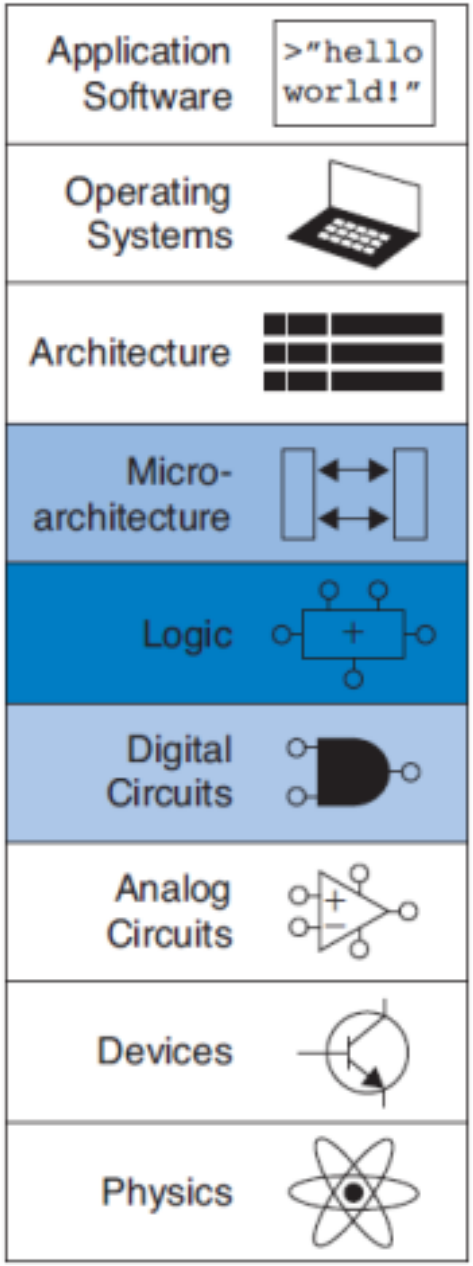
Prof. Erivelton Geraldo Nepomuceno

Departamento de Engenharia Elétrica
Universidade Federal de São João del-Rei

20 de fevereiro de 2018

Blocos de Construção Digital

- Circuitos Aritméticos;
- Sistemas Numéricos;
- Blocos de Construção Sequenciais;
- Matrizes de Memória;
- Matrizes Lógicas.

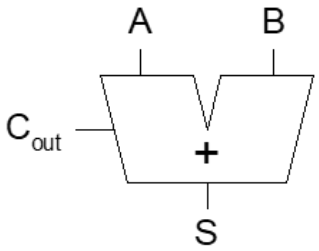


Circuitos Aritméticos

Os circuitos aritméticos são os blocos de construção centrais dos computadores. Computadores e lógica digital executam várias funções aritméticas: adição, subtração, comparações, deslocamentos, multiplicação e divisão. Esta Seção descreve as implementações de hardware para todas estas operações.

Adição

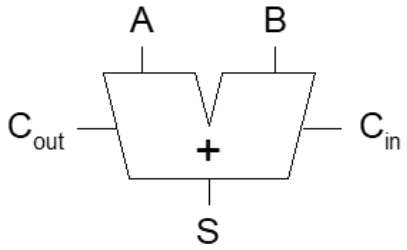
Half Adder



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

$$S = A \oplus B$$
$$C_{out} = AB$$

Full Adder



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

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

Adição Multi-bit

- Tipos de somador Carry Propagate (CPAs):
  - Ripple Carry;
  - Carry-Lookahead;
  - Prefix.

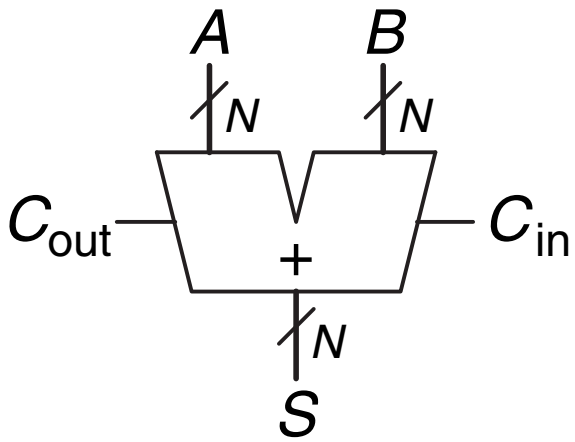


Figura 1: Símbolo.

Somador Ripple-Carry

A maneira mais simples de construir um somador de N-bits com propagação do transporte é encadear N full adders. O C_{out} de um estágio atua como o C_{in} do estágio seguinte.

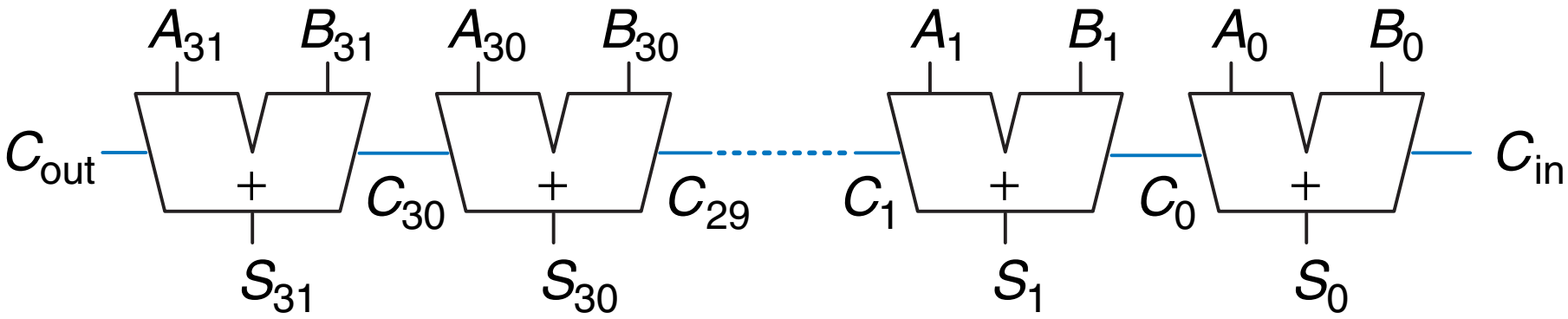


Figura 2: Somador ripple-carry de 32 bits.

Somador Carry-Lookahead

O somador carry lookahead (CLA – carry lookahead adder) é outro tipo de somador com transporte que divide o somador em blocos e fornece circuitos para determinar rapidamente o transporte de saída de um bloco assim que o transporte de entrada é conhecido.

- Algumas definições:
  - A coluna i de um somador é dito gerar um transporte se produz um transporte de saída independente do transporte de entrada;
  - Geração (G_i) e propagação (P_i) de sinais por cada coluna:
    - A coluna i gerará carry out se A_i e B_i são ambos 1.

$$G_i = A_i B_i$$

    - A coluna i propagará carry in para carry out se A_i ou B_1 é 1.

$$P_i = A_i + B_i$$

    - O carry out da coluna i (C_i) é:

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

Somador Carry-Lookahead

- **Exemplo:** Bloco de 4 bits ($G_{3:0}$ e $P_{3:0}$):

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

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

- **Generalizando,**

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

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

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

Somador Carry-Lookahead

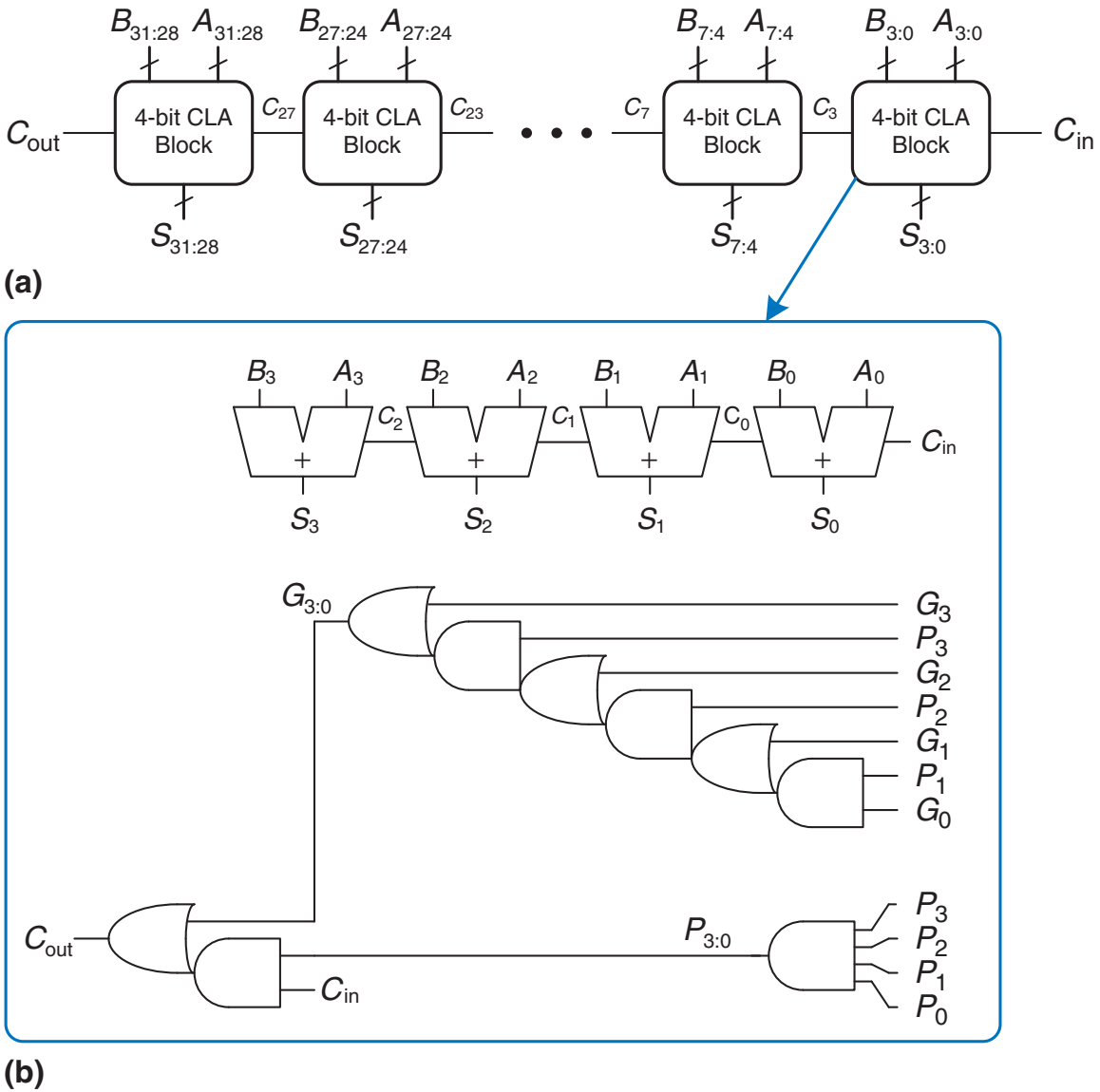


Figura 3: (a)Somador de 32 bits carry-lookahead,(b) bloco de 4 bits.

Somador Carry-Lookahead

Um somador de N-bits dividido em blocos de k-bits tem um atraso

$$t_{CLA} = t_{pg} + t_{pg_block} + (\frac{N}{k} - 1)t_{AND_OR} + kt_{FA}.$$

em que t_{pg} é o atraso das portas AND/OR para gerar P_i e G_i, t_{pg_block} é o atraso para encontrar sinais de $P_{i:j}$ e $G_{i:j}$ para um bloco de k-bits, e t_{AND-OR} é o atraso de C_{in} até C_{out} através da lógica final AND/OR do bloco de k-bits do CLA.

Somador Prefix

A estratégia do somador prefix é calcular o transporte em C_{i-1} para cada coluna i tão rapidamente quanto possível, em seguida, calcular a soma, utilizando

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

Somador Prefix

O caminho crítico para um somador prefix de N-bits envolve a pré-computação de P_i e G_i seguido por $log_2 N$ estágios de células negras prefix para obter todos os prefixos. $G_{i-1:-1}$, prossegue então através da porta XOR final em baixo para calcular S_i. Matematicamente, o atraso de um somador prefix de N-bits é

$$t_{PA} = t_{pg} + log_2 N(t_{pg-prefix}) + t_{XOR}, \quad (1)$$

em que $t_{pg-prefix}$ é o atraso de uma célula negra prefix.

Subtração

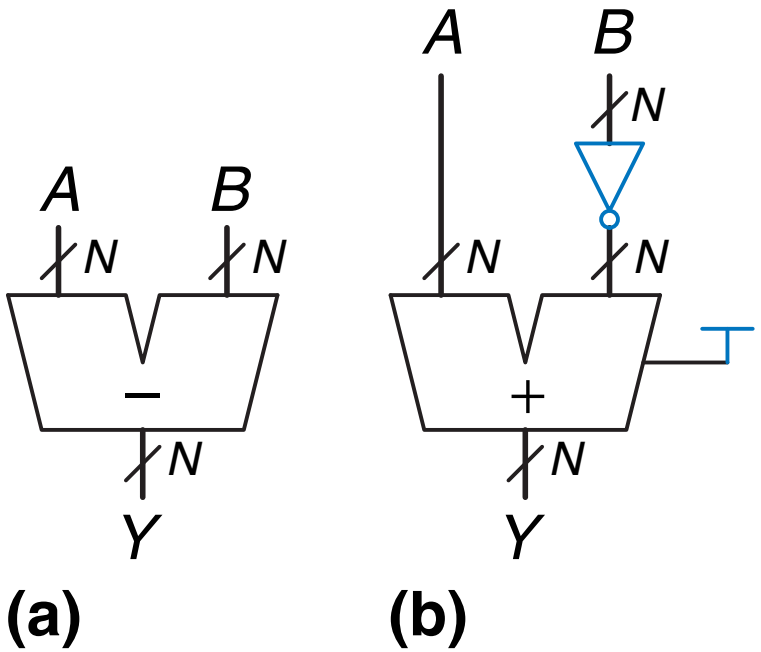


Figura 4: Subtrator: (a) símbolo, (b) implementação.

Comparadores

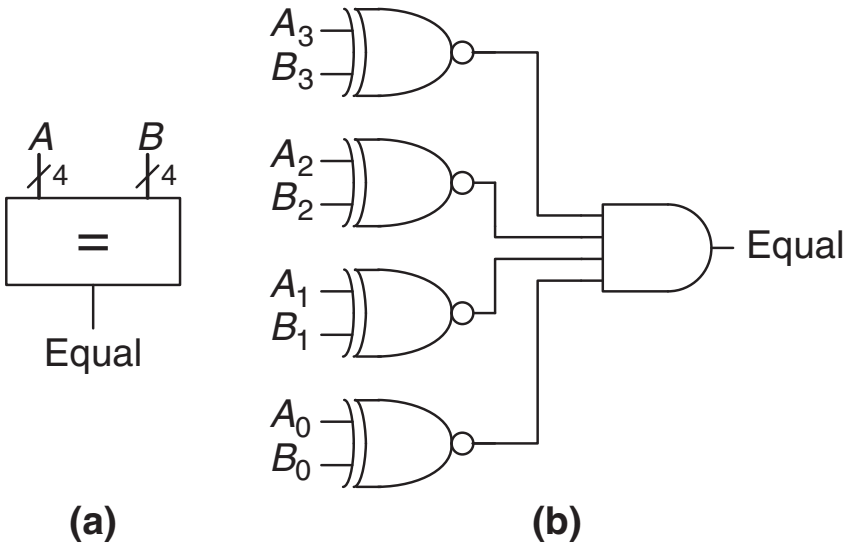


Figura 5: Comparador de 4 bits: (a) símbolo, (b) implementação.

Um comparador de igualdade produz uma saída que indica se A é igual a B (A == B). Um comparador de magnitude produz uma ou mais saídas, indicando os valores relativos de A e B.

Multiplicação

Um multiplicador N x N multiplica dois números de N-bits e produz um resultado de 2N bits. Os produtos parciais da multiplicação binária são ou o multiplicando ou 0.

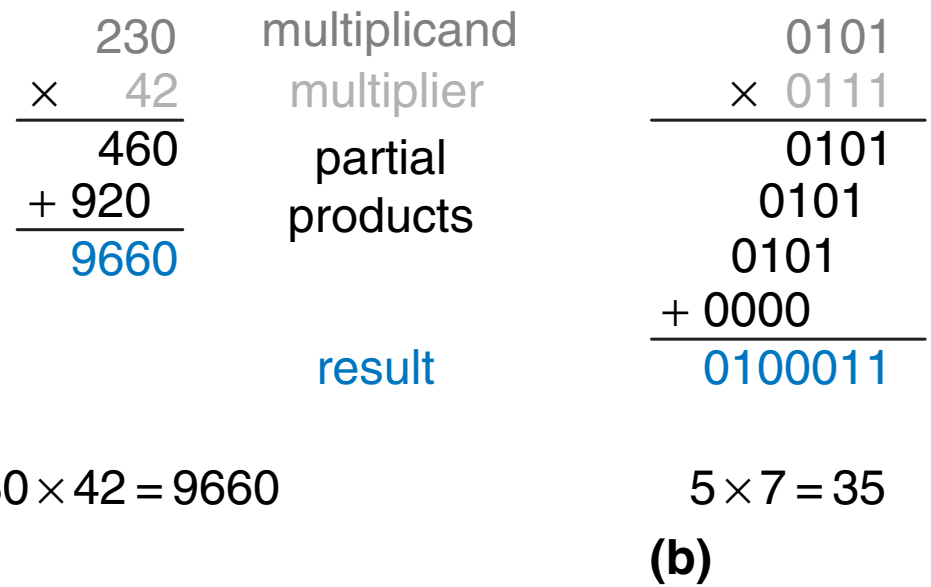


Figura 6: Multiplicação: (a) decimal, (b) binária.

Multiplicação

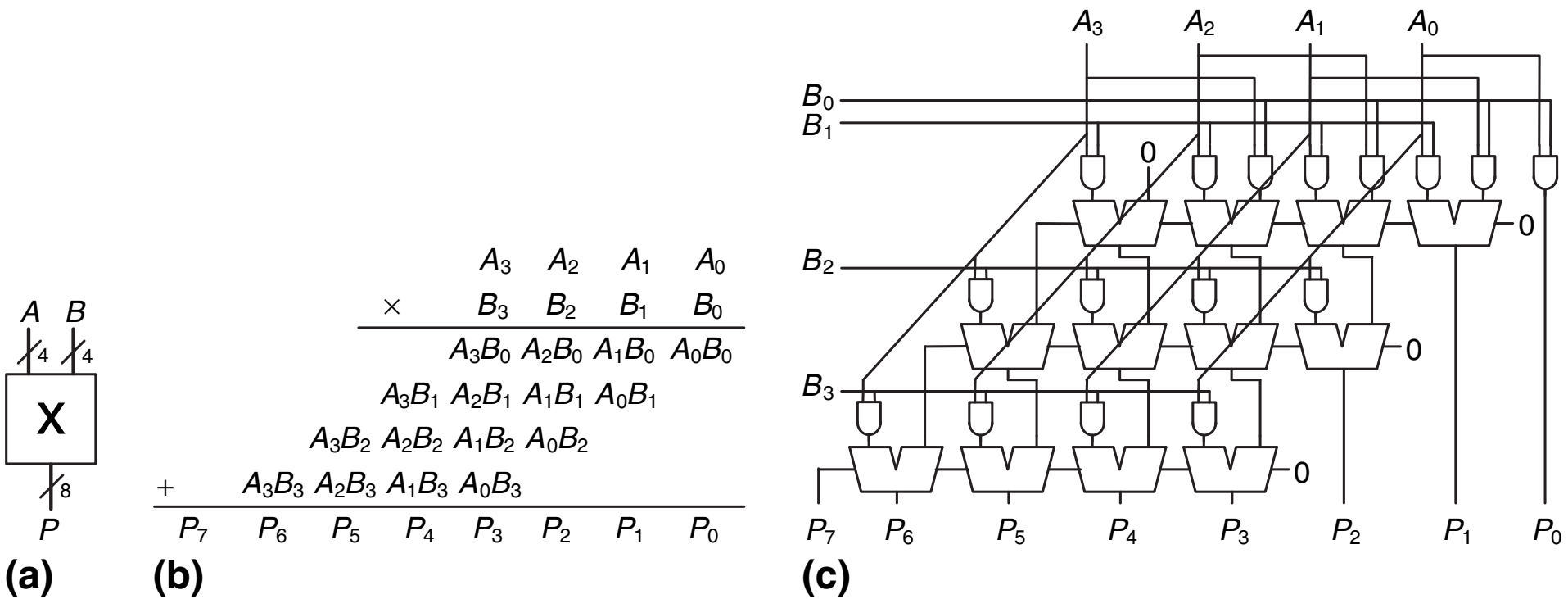


Figura 7: Multiplicação: (a) decimal, (b) binária.

Divisão

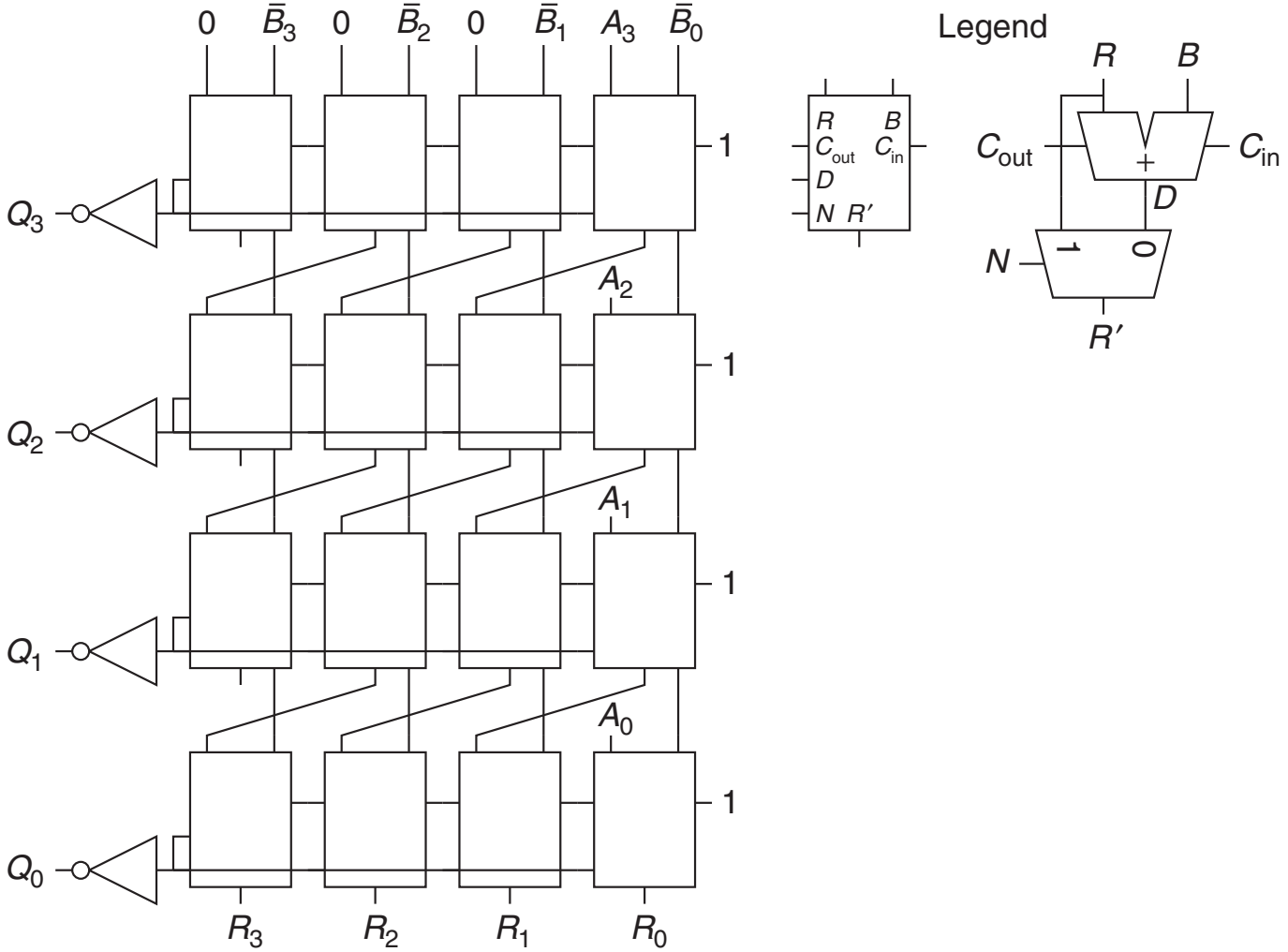


Figura 8: Matriz divisora.

Unidade Lógica Aritmética - ALU

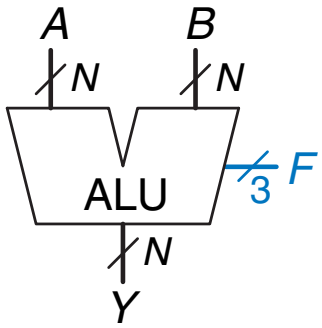


Figura 9: Símbolo de uma ALU.

Tabela 1: Operações da ALU

$F_{2:0}$	Função
000	A AND B
001	A OR B
010	$A + B$
011	não é usado
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	$A - B$
111	se for menor que

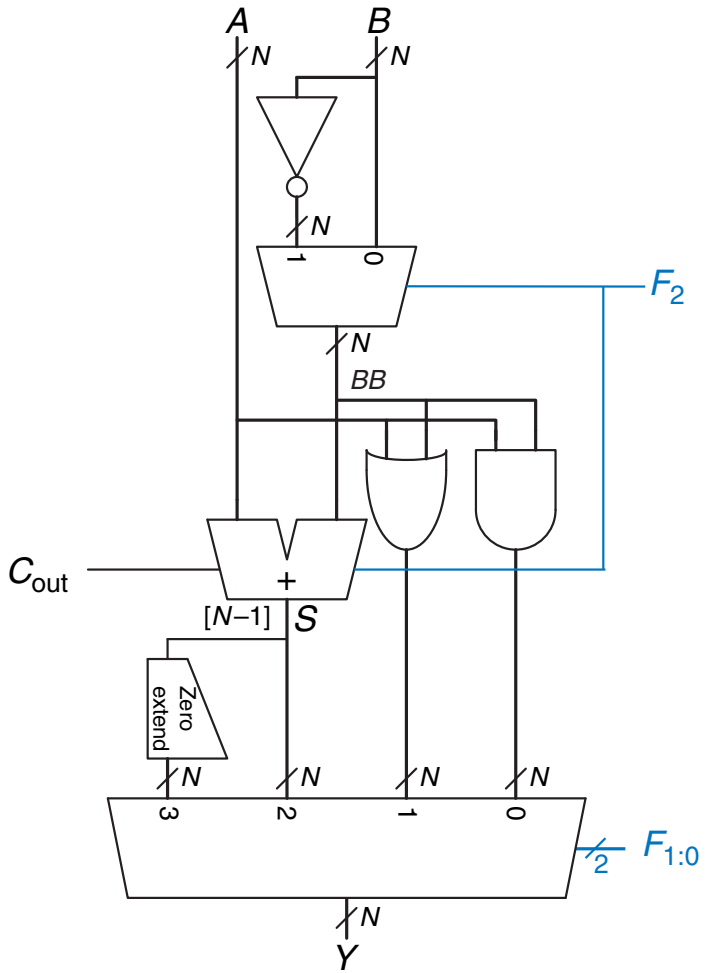


Figura 10: ALU de N bits.

Shifters e Rotators

Os shifters e rotators deslocam os bits e multiplicam ou dividem por potências de 2. Como o nome indica, um shifter desloca um número binário para a esquerda ou para a direita um número especificado de posições. Existem vários tipos de shifters normalmente utilizados:

- shifter lógico: desloca o número à esquerda (LSL) ou à direita (LSR) e preenche os lugares vazios com 0.
  Ex: 11001 LSR 2 = 00110; 11001 LSL 2 = 00100.
- shifter aritmético: é o mesmo que um shifter lógico, mas nos deslocamentos para a direita preenche os bits mais significativos.
  Ex: 11001 ASR 2 = 11110; 11001 ASL 2 = 00100.
- rotator: circula o número tal que os lugares vazios são preenchidos com os bits que saem da outra extremidade.
  Ex: 11001 ROR 2 = 01110; 11001 ROL 2 = 00111.

Shifters e Rotators

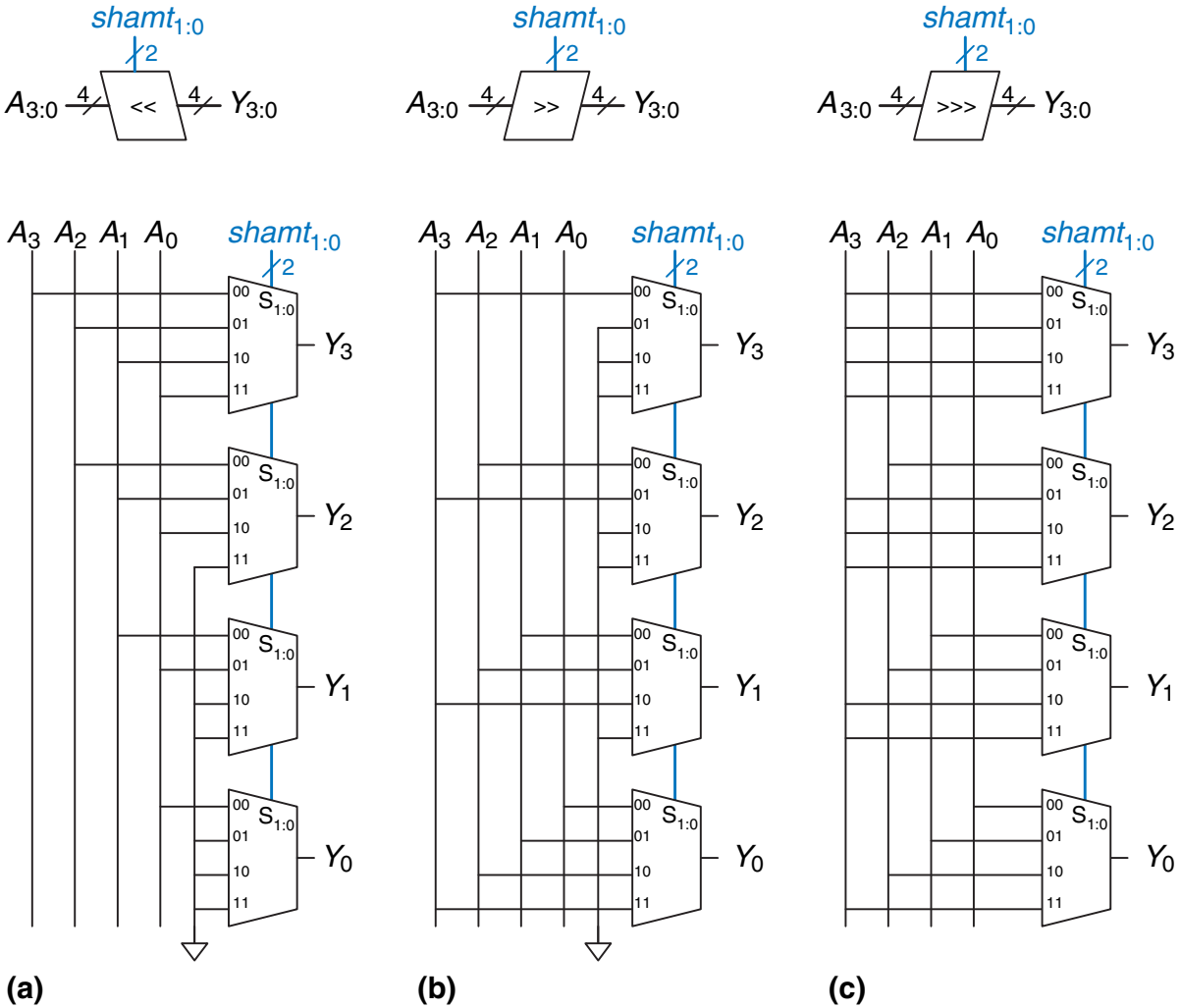


Figura 11: Shifters de 4-bit: (a) shift à esquerda, (b) shift lógico à direita, (c) shift aritmético à direita.

Sistema Numérico

Números que podem ser representados usando notação binária:

- Números positivos;
- Números negativos;
  - Complemento de 2;
  - Sinal/ Magnitude dos números.

Sistema Numérico

Há dois métodos: ponto fixo e ponto flutuante.

- Ponto Fixo: 1 bit para o sinal, um grupo de bits para representar o número antes do ponto binário e um grupo de bits para representar o número após o ponto binário.

- Ponto (Vírgula) Flutuante: O ponto decimal flutua para a posição imediatamente posterior ao primeiro dígito não nulo. Está é a razão para o nome ponto flutuante.

Ponto Fixo

6,75 usando sistema de ponto fixo com quatro bits inteiros e quatro bits fracionários.

(a) 01101100

(b) 0110,1100

(c) $2^2 + 2^1 + 2^{-1} + 2^{-2} = 6{,}75$

- O ponto binário está implícito;
- O número de bits da parte inteira e da parte fracionária devem ser previamente selecionados.

Exemplo Ponto Fixo

- Representar o número $7{,}5_{10}$ usando 4 bits inteiros e 4 bits fracionários

Exemplo Ponto Fixo

- Representar o número $7{,}5_{10}$ usando 4 bits inteiros e 4 bits fracionários

01111000

Ponto Fixo com sinal

- Representações:
  - Complemento de 2;
  - Sinal/ Magnitude dos números.

Exemplo: Representar o número $-7{,}5_{10}$ usando 4 bits inteiros e 4 bits fracionários

Sinal Magnitude

11111000

Complemento de 2

1. +7.5:	01111000
2. Inversão de bits:	10000111
3. Add 1 ao lsb:	+ 1
	10001000

lsb: bit menos significativo

Sistemas Numéricos de Ponto Flutuante

Semelhante à notação científica decimal

Por exemplo, escreva 273_{10} em notação científica:

$$273 = 2{,}73 \times 10^2$$

Em geral, um número é escrito em notação científica como:

$$\pm M \times B^E$$

M = Mantissa B = base E = expoente

Sistemas Numéricos de Ponto Flutuante

Exemplo: representar o valor 228_{10} usando uma representação de ponto flutuante de 32 bits

Representação Ponto Flutuante

1. Converter o número decimal para binário:

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

2. Escreva o número na notação científica binária:

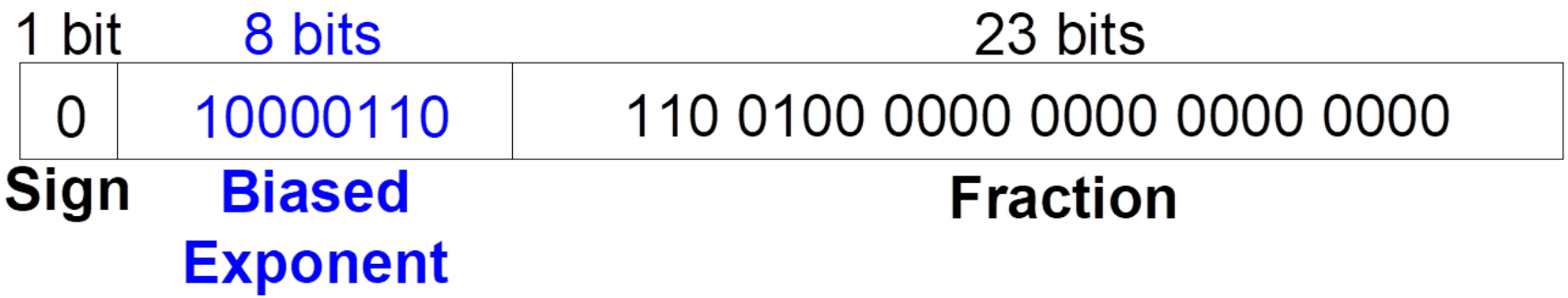
$$11100100_2 = 1{,}11001_2 \times 2^7$$

3. O expoente utiliza uma representação deslocada (biased representation). Biased exponent: $bias = 127(01111111_2)$
   - Expoente deslocado: bias + expoente
   - O Exponente de 7 é armazenado como:

$$127 + 7 = 134 = 0 \times 10000110_2$$

Representação Ponto Flutuante

A representação em ponto flutuante IEEE 754 de 32 bits de 228_{10}



Exemplo Ponto Flutuante

Escreva $-58{,}25_{10}$ em ponto flutuante (IEEE 754)

Exemplo Ponto Flutuante

1. Converter decimal para binário:

$$58{,}25_{10} = 111010{,}01_2$$

2. Escrever em notação científica:

$$1{,}1101001 \times 2^5$$

3. Divida nos campos:
   Bit de sinal: 1 (negativo)
   8 bits de expoente: (127 + 5) = 132 = 10000100_2
   23 bits da mantissa: 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**

Ponto Flutuante: Casos Especiais

Número	Sinal	Expoente	Mantissa
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
$-\infty$	1	11111111	00000000000000000000000
NaN	X	11111111	Não zero

Ponto Flutuante: Precisão

- O formato single possui:
  - 1 bit: Sinal;
  - 8 bits: Expoente;
  - 23 bits: Mantissa;
  - O expoente utiliza uma representação deslocada (biased representation).O expoente é a representação binária de E + 127.
- O formato double possui:
  - 1 bit: sinal;
  - 11 bits: expoente;
  - 52 bits: Mantissa;
  - O expoente utiliza uma representação deslocada (biased representation).O expoente é a representação binária de E + 1023.

Ponto Flutuante: Arredondamento

- A norma IEEE define o valor arredondado correto de x, denotado por $round(x)$ da seguinte forma.
- Se x é um número flutuante então $round(x) = x$. Senão, o valor depende do modo de arredondamento:
  - Arredondamento para baixo: $round(x) = x_-$.
  - Arredondamento para cima: $round(x) = x^+$.
  - Arredondamento em direção a zero: $round(x) = x_-(x > 0)$ ou $round(x) = x^+(x < 0)$
  - Arredondamento para o mais próximo: $round(x)$ é tanto x_- ou x^+, dependendo de qual for mais próximo de x. Se houver um empate, aquele com o último bit significativo igual a zero é escolhido.

Ponto Flutuante: Arredondamento

Exemplo: Arredondar 1,100101 (1,578125) com somente 3 bits na parte fracionária.

Para Baixo: 1,100

Para cima: 1,101

Em direção a zero: 1,100

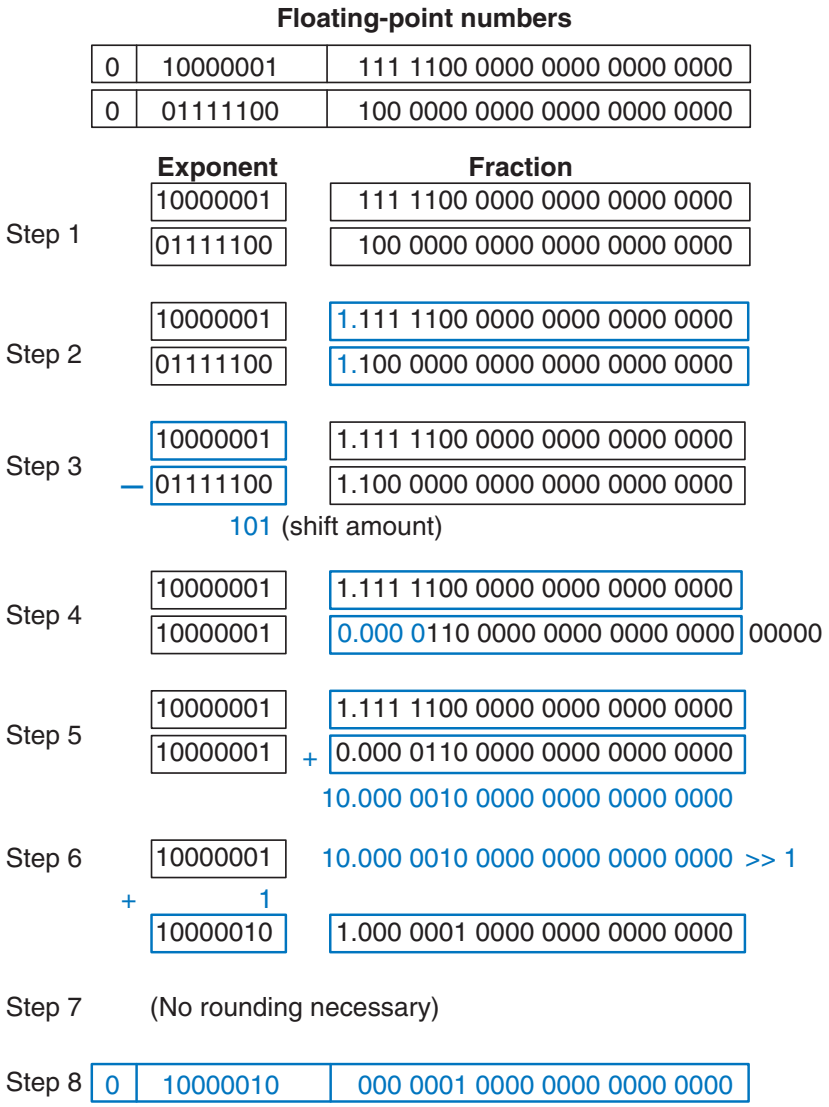
Para o mais próximo: 1,101 (1,625 é mais próximo de 1,578125 do que de 1,5)

Adição de Ponto Flutuante

As etapas para adicionar os números de vírgula flutuante com o mesmo sinal são as seguintes:

1. Extrair os bits de expoente e fracionários.
2. Acrescente o 1 para formar a mantissa.
3. Comparar os expoentes.
4. Desloque a mantissa menor se for necessário.
5. Some as mantissas.
6. Normalize a mantissa e ajuste o expoente, se necessário.
7. Arredonde o resultado.
8. Construa o expoente e o fracionário novamente num número de vírgula flutuante.

Adição de Ponto Flutuante



Contadores

- Um contador binário de $N-bits$, é um circuito aritmético sequencial com entradas de relógio e de *reset* e uma saída Q de $N-bits$;
- O *reset* repõe a saída a 0;
- O contador avança então ao longo de todas as 2^N saídas possíveis numa ordem binária;
- Exemplo: 000, 001, 010, 011, 100, 101, 110, 111, 000, 001, $\cdots$;
- Contador é composto por um somador e um registro *resettable*. Cada ciclo, o contador adiciona 1 ao valor armazenado no registrador.

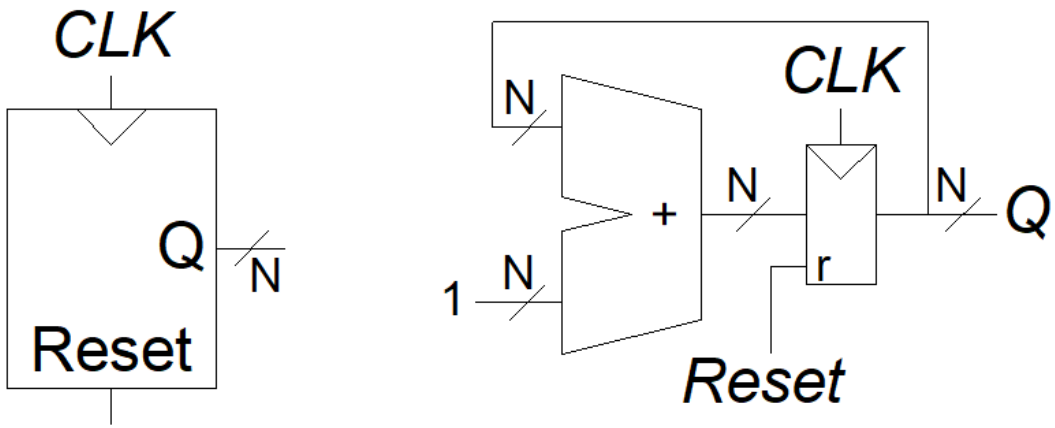


Figura 12: Símbolo e contador de $n-bits$.

Registrador de Deslocamento

- A cada borda ascendente do relógio, um novo bit é deslocado da entrada S_{in} e todo o conteúdo anterior é deslocado para a frente;
- O último bit no registrador de deslocamento está disponível em S_{out};
- A entrada é fornecida em série (um bit de cada vez) no S_{in};
- Depois de N ciclos, as últimas N entradas estão disponíveis em paralelo em Q;
- Um registrador de deslocamento pode ser construído a partir de N flip-flops ligados em série.

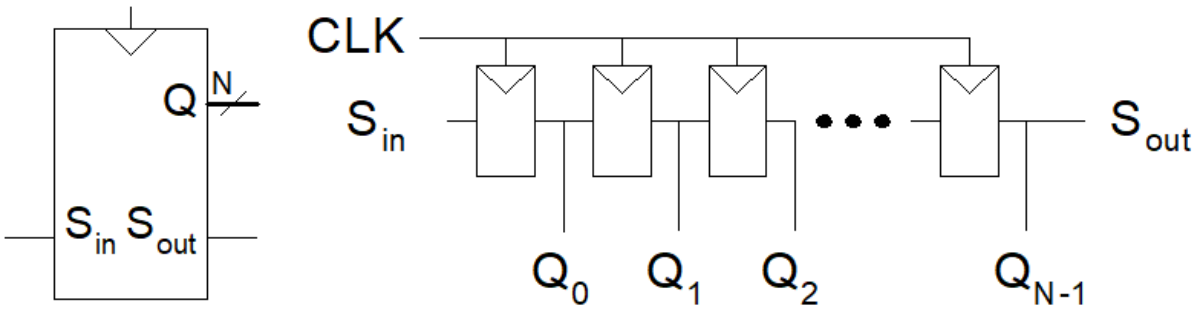


Figura 13: Símbolo e implementação.

Matrizes de Memória

- Os registros construídos de *flip-flops* são uma espécie de memória que armazena pequenas quantidades de dados;
- Tipos de memória: memórias dinâmicas de acesso aleatório (DRAM), memórias estáticas de acesso aleatório (SRAM) e memória só de leitura (ROM);
- Valor de dados $M - bit$ lido/escrito em cada endereço de $N - bit$ exclusivo.

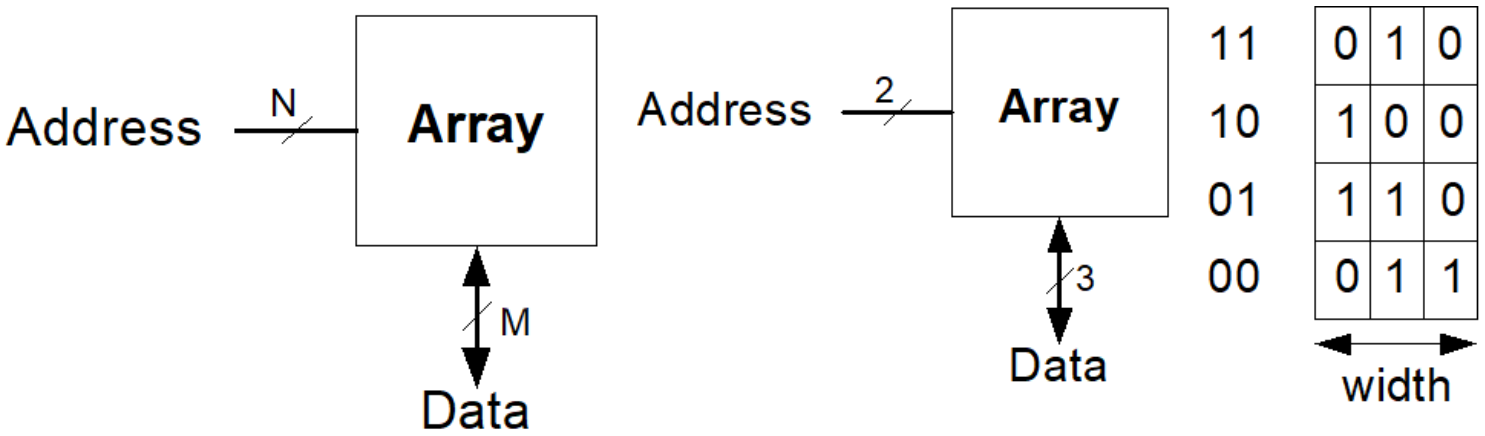


Figura 14: Memória matricial e Matriz de memória 4 x 3.

Organização de Memória

- Leitura de memória: *wordline* é acedida, e a linha correspondente das células de bit coloca as *bitlines* a *HIGH* ou *LOW*;
- Escrita na memória: as *bitlines* são primeiro colocadas a *HIGH* ou *LOW* e depois uma *wordline* é acedida.

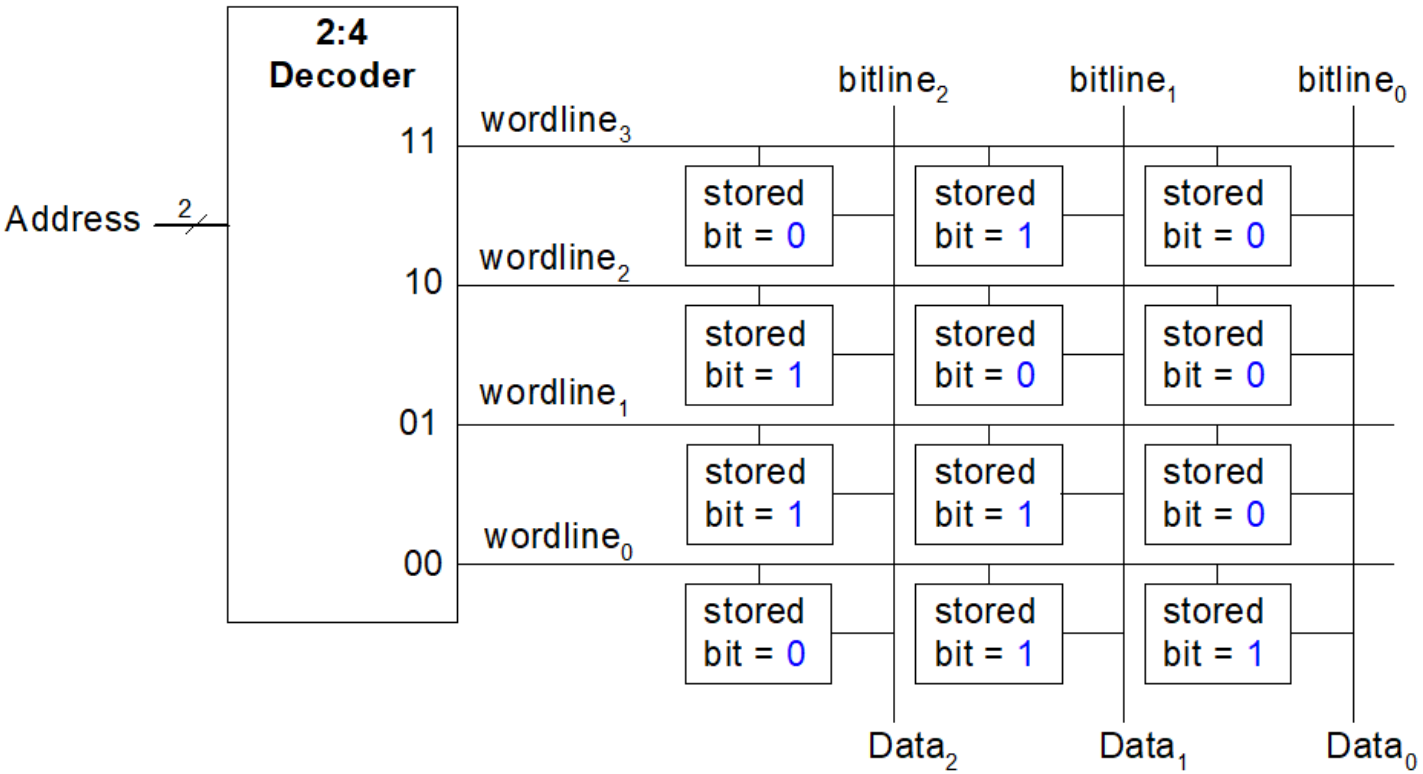


Figura 15: Organização interna de uma matriz de memória de 4 x 3.

Tipos de Memória

- As matrizes de memória são especificadas pelo seu tamanho (depth x width) e número e tipo de portos.
- As memórias são classificadas com base na forma como armazenam os bits na célula de bit.
  - RAM – random access memory: memória de acesso aleatório.
    - Volátil, o que significa que ela perde os seus dados quando a energia é desligada
  - ROM – read only memory: memória apenas de leitura.
    - Não volátil, o que significa que ela mantém os seus dados indefinidamente, mesmo sem uma fonte de energia.

Tipos de RAM

- RAM Dinâmica (DRAM): As RAM dinâmicas armazenam dados como carga num capacitor, ou seja, armazena um bit como a presença ou ausência de carga num capacitor.

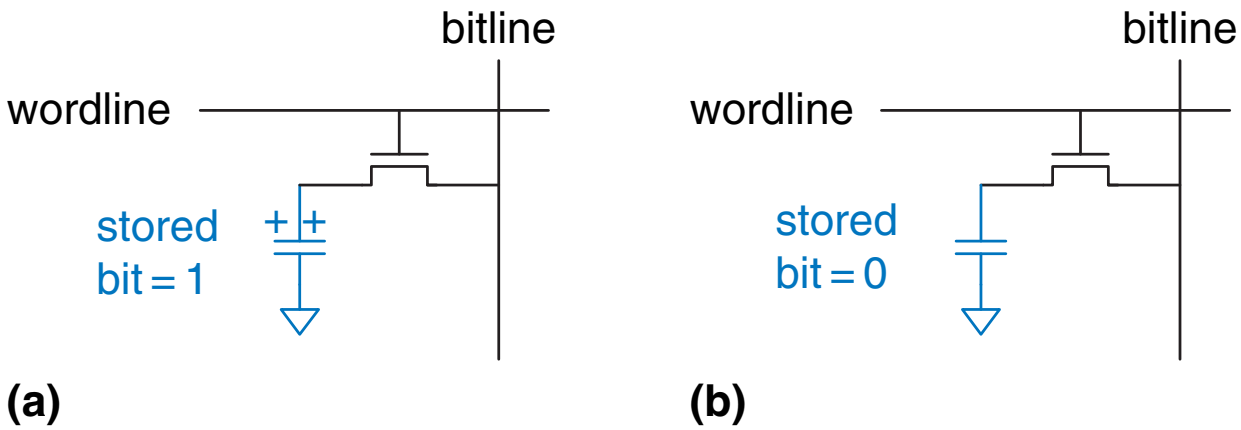


Figura 16: Armazenamento de valores em DRAM.

Tipos de RAM

- RAM estática (SRAM): RAM estáticas usam um par de inversores cross-coupled, nesse caso, os bits armazenados não precisam ser atualizados.

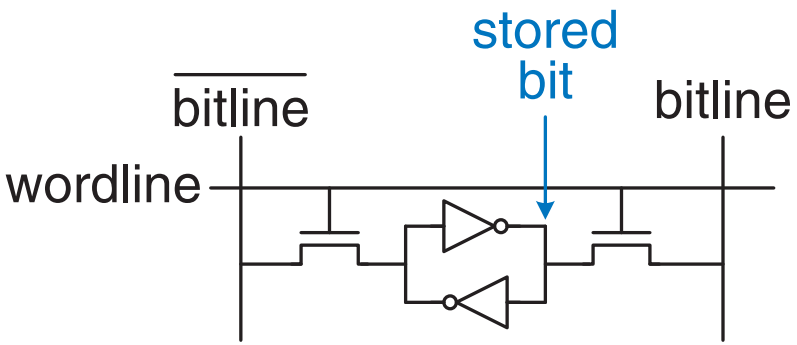


Figura 17: Célula de bit SRAM.

Área e Atraso

Os flip-flops, as SRAM e as DRAM são todas memórias voláteis, mas cada um tem diferentes características de área e atraso. A Tabela 2 mostra uma comparação destes três tipos de memória volátil.

Tabela 2: Comparação de Memórias

Tipo de Memória	Transistor por Bit de Célula	Latência
Flip-flop	~ 20	Rápido
DRAM	6	Médio
SRAM	1	Lento

Banco de Registradores

Os sistemas digitais costumam usar um número de registradores para armazenar variáveis temporárias. Este grupo de registradores, chamado de banco de registro, normalmente é construído como uma matriz SRAM pequena, multi-porto, porque é mais compacta do que um conjunto de flip-flops.

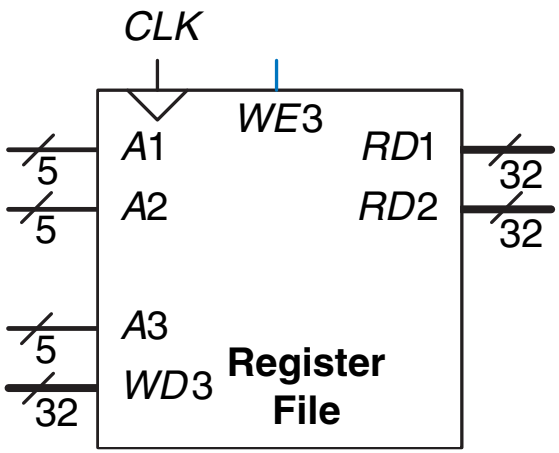


Figura 18: Banco de registros 32 x 32 com dois portos de leitura e um porto de escrita.

Read Only Memory - ROM

As read only memory (ROM) armazenam um bit quanto na presença ou ausência de um transístor. A Figura 19 mostra uma simples célula de bit ROM.

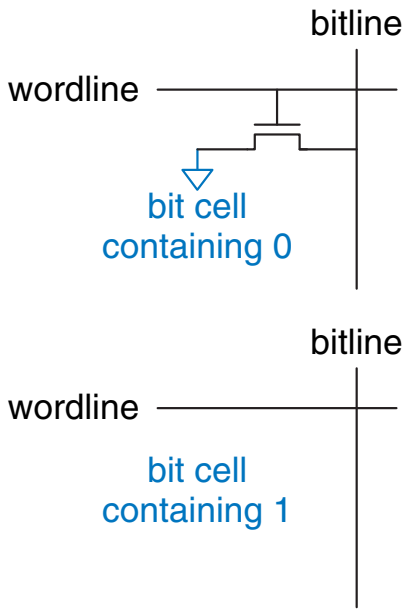


Figura 19: Células de bit ROM contendo 0 e 1.

Read Only Memory - ROM

O conteúdo de uma ROM pode ser indicado usando a notação de ponto. A Figura 20 mostra a notação de ponto para uma ROM de 4-words $\times$ 3-bits.

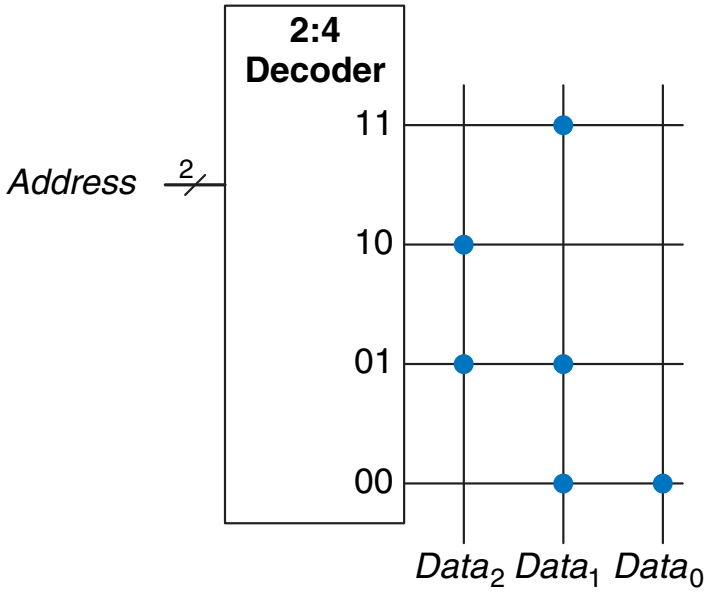


Figura 20: ROM 4 $\times$ 3: notação de ponto.

Read Only Memory - ROM

Conceitualmente, as ROM podem ser construídas usando lógica de dois níveis com um grupo de portas AND seguido por um grupo de portas OR. As portas AND produzem todos os mintermos possíveis e, portanto, formam um decodificador . A Figura 21 mostra a ROM da Figura 20 construída usando um decodificador e portas OR.

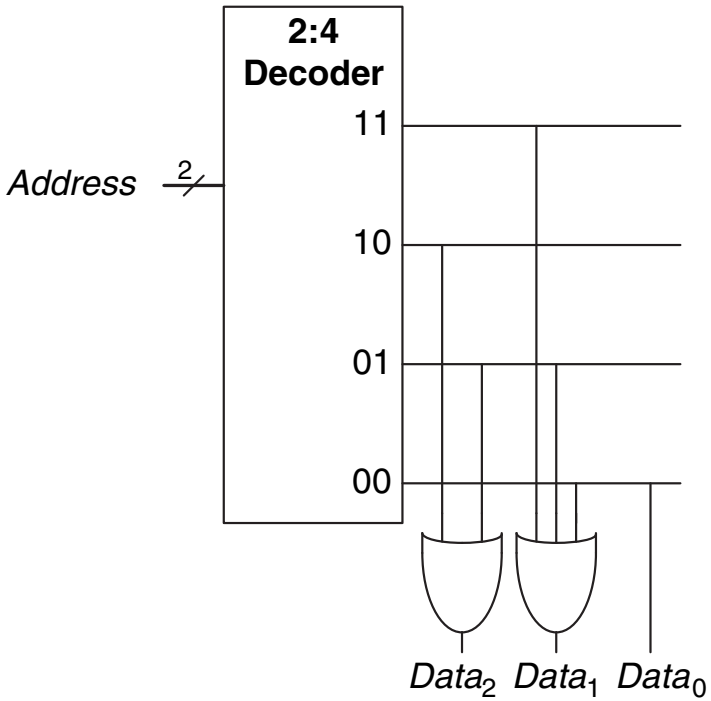


Figura 21: ROM 4 × 3: implementação utilizando portas.

Read Only Memory - ROM

A ROM programável (PROM) coloca um transistor em cada célula de bit, mas fornece uma maneira de ligar ou desligar o transistor à terra. A Figura 22 apresenta a célula de bit para uma ROM programável por fusível. O utilizador programa a ROM através da aplicação de uma tensão elevada para queimar seletivamente os fusíveis.

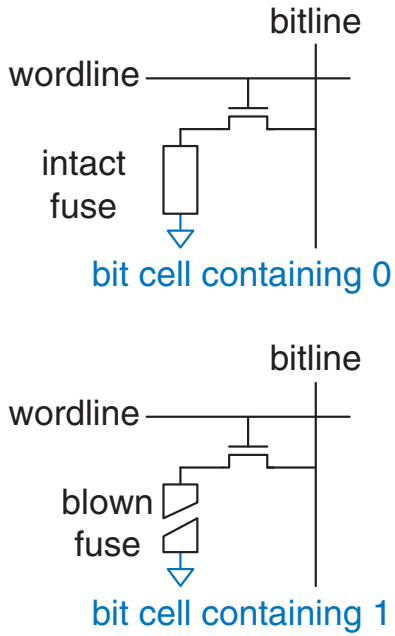


Figura 22: Célula de bit ROM programável por fusíve.

Lógica Usando Matrizes de Memória

Embora elas sejam usadas principalmente no armazenamento de dados, as matrizes de memória também podem executar funções de lógica combinatória.A memória 2^N-word $\times$ M-bits pode executar qualquer função combinatória de N entradas e M saídas. As matrizes de memória usadas para executar lógica são chamadas de lookup tables (LUT). A Figura 23 apresenta uma matriz de memória de 4-word $\times$ 1 bit usada como uma lookup table para executar a função Y = AB.

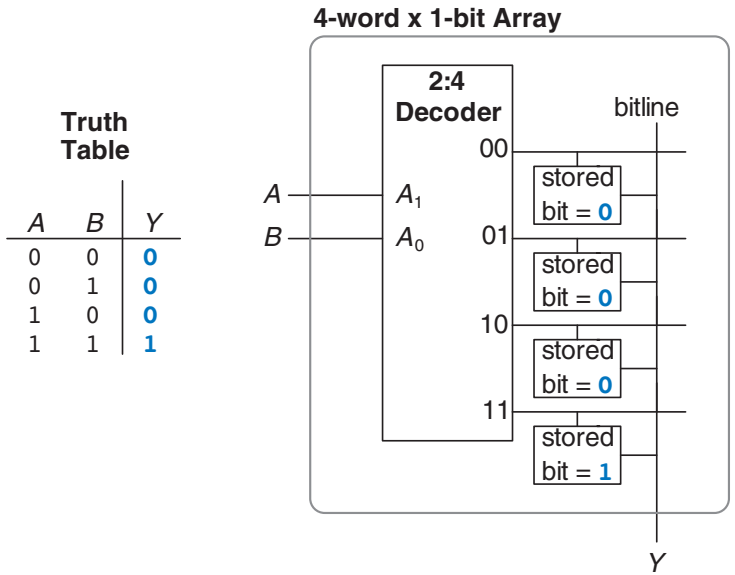


Figura 23: Matriz de memória 4-word $\times$ 1-bit usada como lookup table.

Matrizes Lógicas

- Programmable Logic Arrays (PLA)
  - Executam apenas funções lógicas combinatórias;
- Field Programmable Gate Arrays (FPGA)
  - Podem executar tanto lógica combinatória e como sequencial.

Programmable Logic Arrays (PLA)

- As PLA são construídas a partir de uma matriz AND seguida por uma matriz OR;
- As entradas dão entrada na matriz AND, que produz implicantes, que por sua vez passam pelas portas OR em conjunto para formar as saídas;
- Uma PLA $M \times N \times P - \text{bits}$ tem M entradas, N implicantes e P saídas.

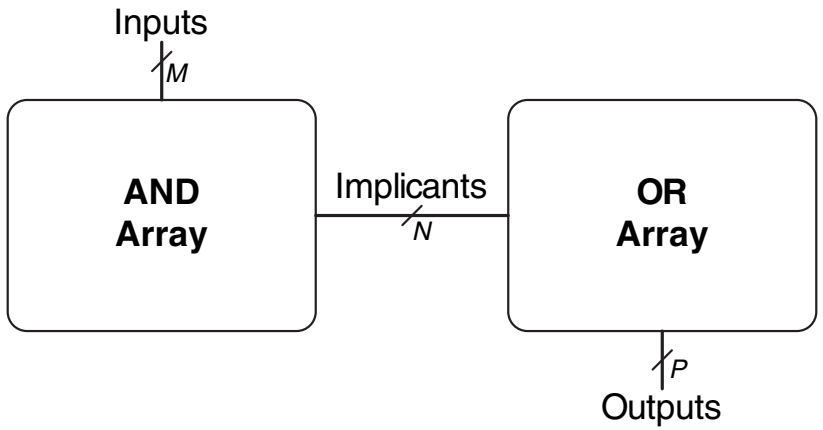


Figura 24: PLA $M \times N \times P - \text{bits}$

PLA: Notação de Ponto

- PLA $3 \times 3 \times 2 - \text{bits}$;
- $X = \overline{A}\overline{B}C + AB\overline{C}$ e $Y = A\overline{B}$;
- Cada linha na matriz forma um implicante.
- Os pontos na matriz OR indicam que implicantes fazem parte da função de saída.

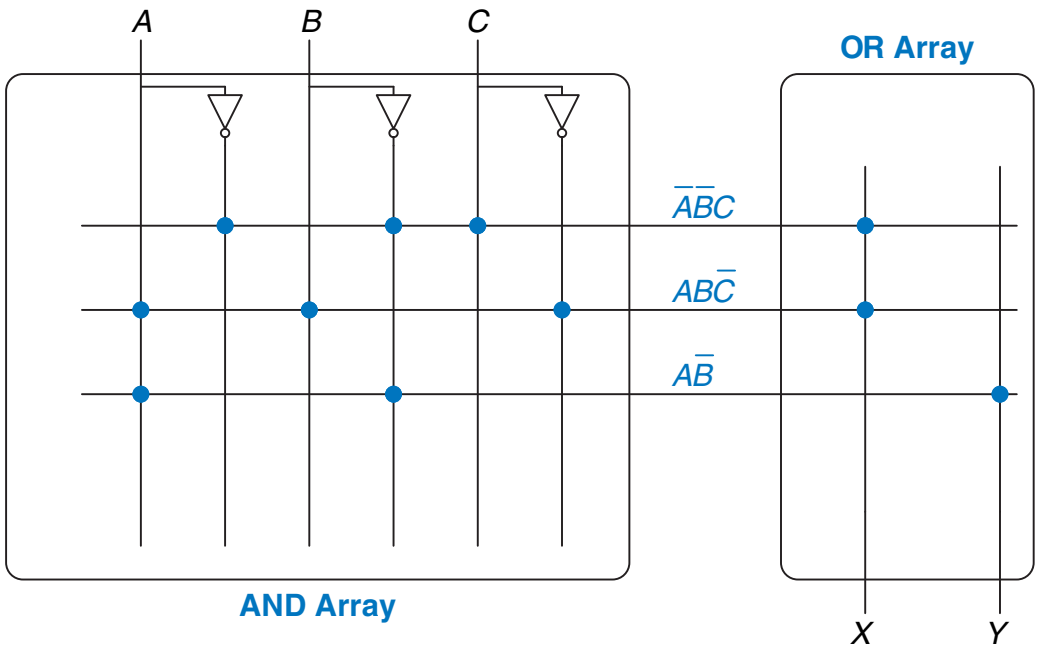


Figura 25: PLA $3 \times 3 \times 2 - \text{bits}$: notação de ponto.

PLAs utilizando lógica de dois níveis

- As PLAs podem ser construídas usando dois níveis lógicos.

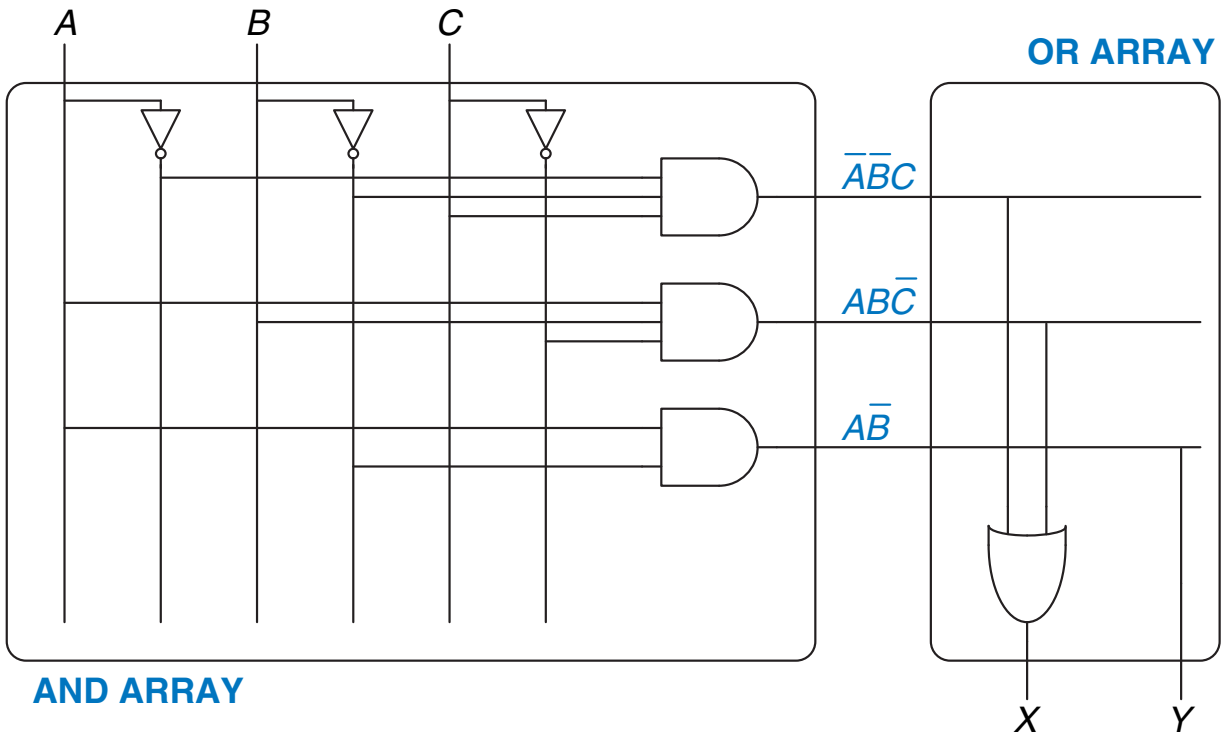


Figura 26: PLA $3 \times 3 \times 2 - \text{bits}$ utilizando lógica de dois níveis.

FPGA: Field Programmable Gate Array

- FPGA é uma matriz de portas reconfiguráveis;
- As FPGA são mais poderosas e mais flexível do que as PLA por várias razões.
  - Elas podem implementar tanto lógica combinatória como lógica sequencial;
  - Elas também podem implementar funções lógicas multi-nível, enquanto as PLA só podem implementar lógica de dois níveis
- As FPGA modernas integram outros recursos úteis, como multiplicadores internos, I/O de alta velocidade, conversores de dados incluindo conversores analógico-digitais, matrizes grandes de RAM e processadores.

FPGA: Field Programmable Gate Array

- Composto de:
  - LEs (logic elements): matriz de elementos lógicos configuráveis, também conhecidos como configurable logic blocks (CLB);
  - IOEs (input output elements): ligam as entradas e as saídas dos LE aos pinos de empacotamento de chip;
  - Canais de encaminhamento programáveis (Programmable interconnection): conecta LEs e IOEs

FPGA: Disposição Geral

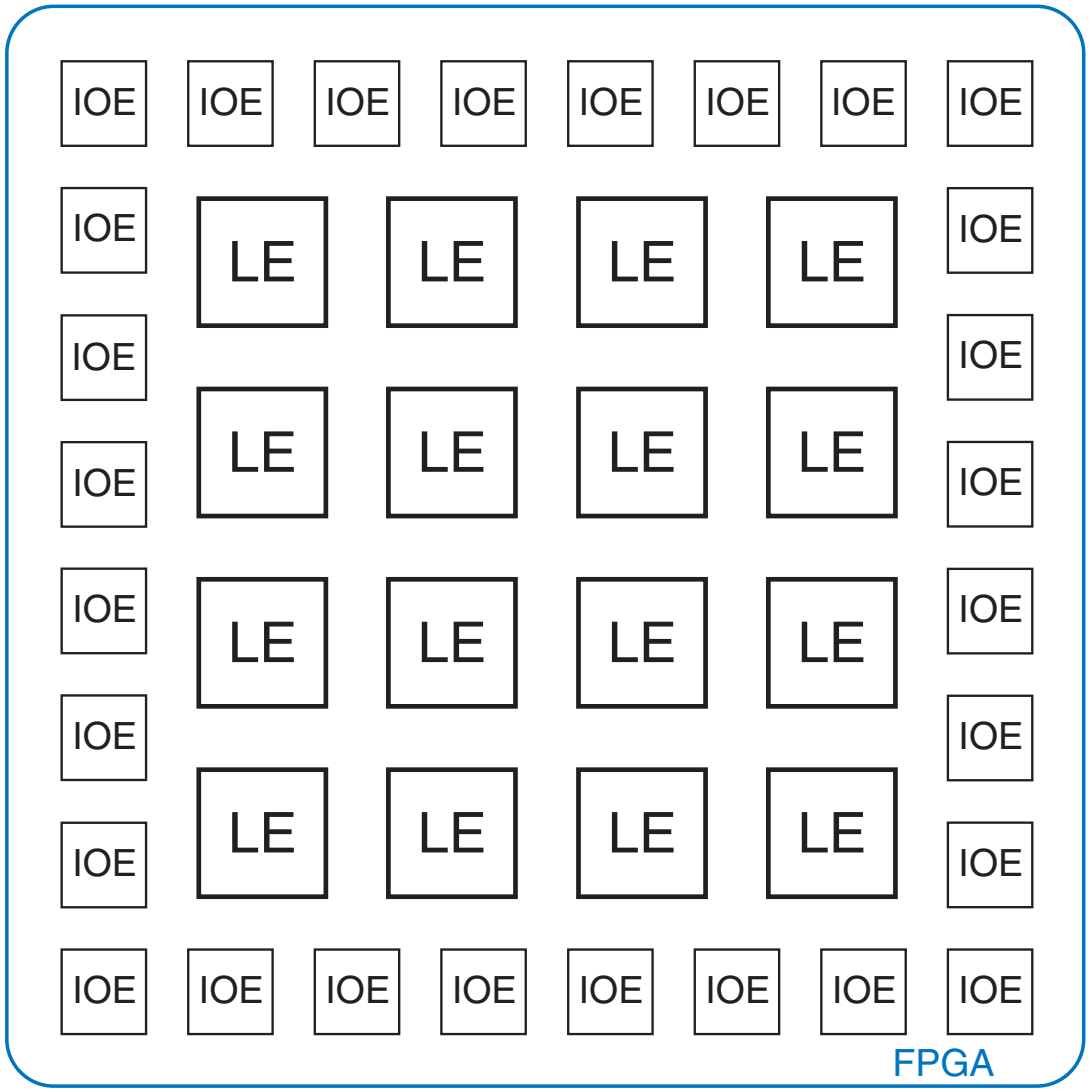


Figura 27: Layout genérico de uma FPGA.

LE: Elementos Lógicos Configuráveis

- Composto de:
  - LUT (lookup tables): executa lógica combinatória;
  - Flip-flops: executa lógica sequencial;
  - Multiplexadores: conecta LUTs e flip-flops.

FPGA: LE Altera Cyclone IV

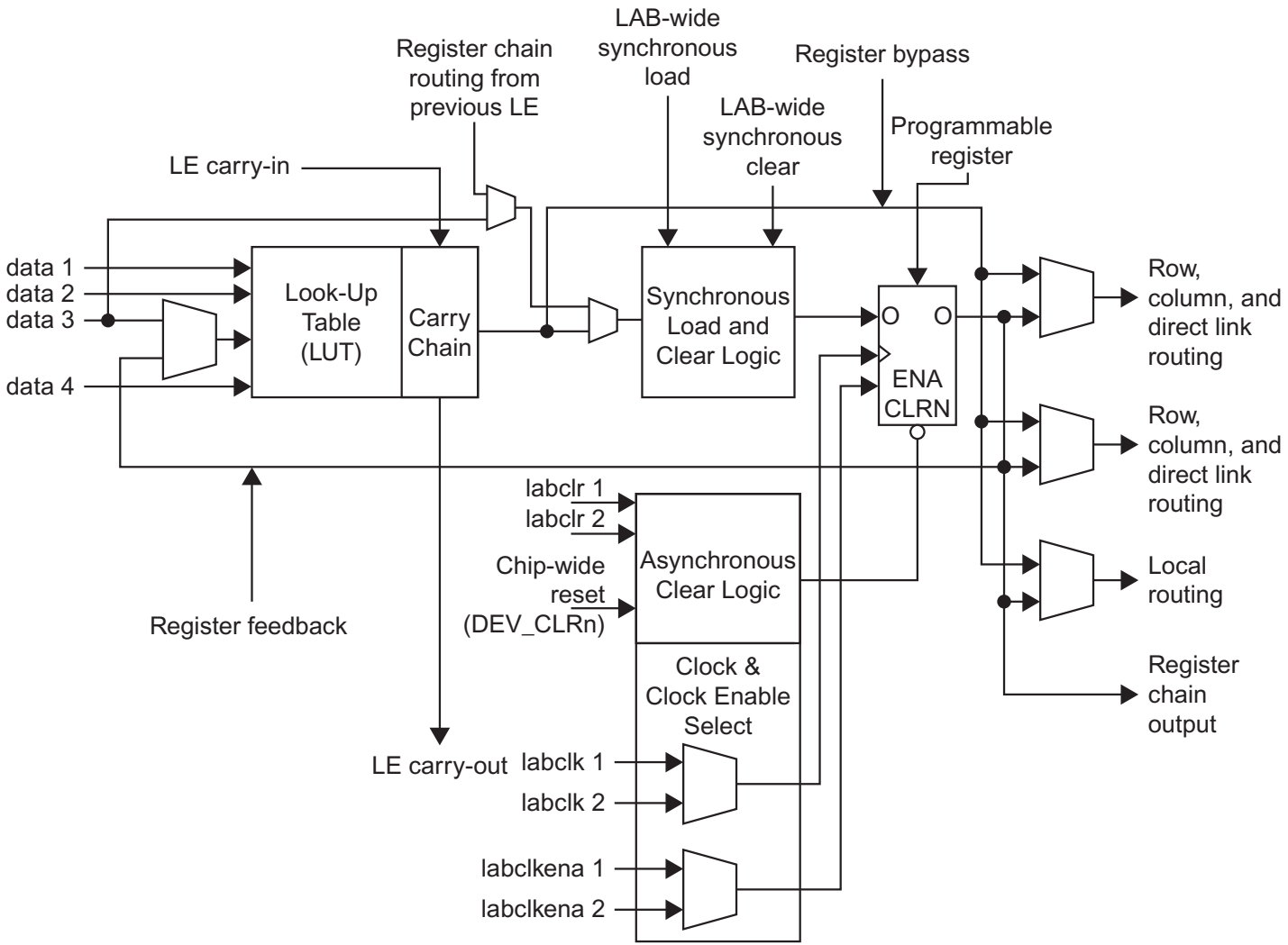


Figura 28: Elemento lógico (LE) do Cyclone^{TM}IV (Reproduzido com permissão da Altera Cyclone IV Handbook ©2010 Altera Corporation).

FPGA: LE Altera Cyclone IV

- O Cyclone IV LE tem:
  - um LUT de 4 entradas;
  - um registro de 1-bit;
  - uma saída combinatória.

Exemplo Configuração LE

Mostre como configurar um LE Cyclone IV LE para realizar as seguintes funções:

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

Exemplo Configuração LE

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

(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

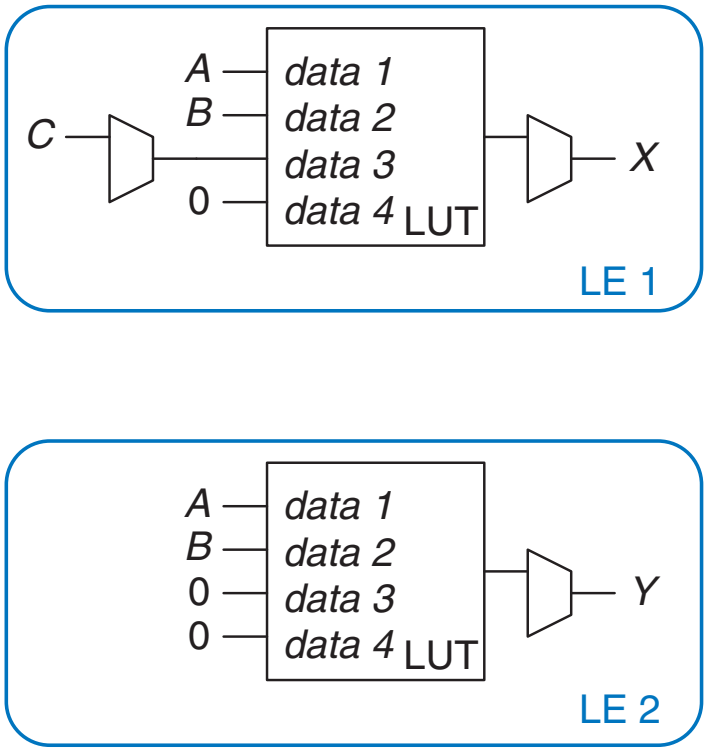
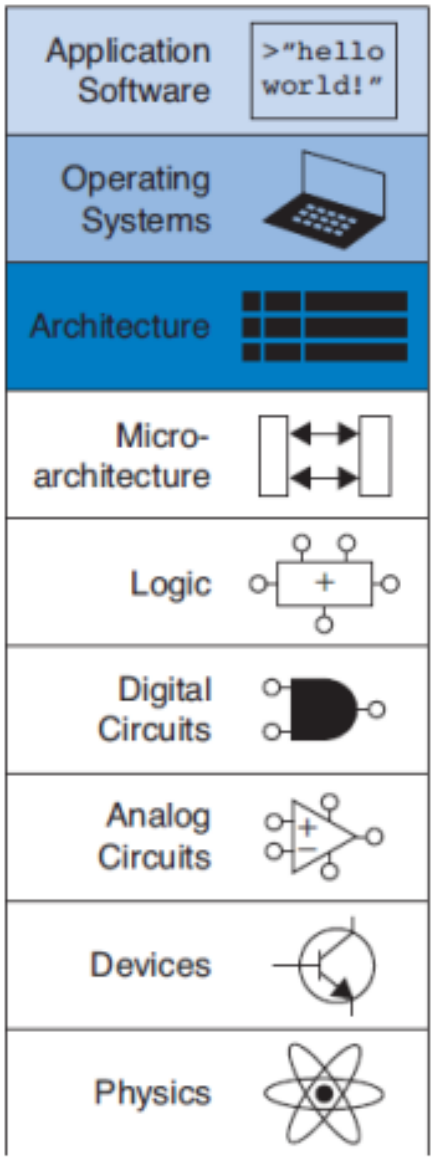


Figura 29: Configuração do LE para duas funções de até quatro entradas cada.

Arquitetura

- Linguagem Assembly.
- Linguagem de Máquina.



Linguagem Assembly

A linguagem assembly é a representação legível para humanos da linguagem nativa dos computadores. Cada instrução da linguagem assembly especifica tanto a operação a ser realizada quanto o operando sobre o qual opera. Discutiremos os seguintes itens:

- Aritmética Simples.
- Registradores.
- Memória.

Linguagem Assembly - Instruções

- Adição

Tabela 3: Adição

Linguagem de alto nível	Código assembly MIPS
$a = b + c;$	add a, b, c

- Subtração

Tabela 4: Subtração

Linguagem de alto nível	Código assembly MIPS
$a = b - c;$	sub a, b, c

Linguagem Assembly - Instruções

O programa em linguagem assembly no Exemplo de Código a seguir requer uma variável temporária t para armazenar o resultado intermediário.

Tabela 5: Código mais complexo

Linguagem de alto nível	Código assembly MIPS
$a = b + c - d;$	sub t, c, d #$t = c–d$ add a, b, t #$a = b + t$

A MIPS possui uma arquitetura de computadores com conjunto de instruções reduzido (reduced instruction set computer – RISC).

Linguagem Assembly - Operandos

Uma instrução opera sobre os operandos. As instruções necessitam de uma localização física, de onde se possam recuperar os dados binários.

- Registradores.
- Memória.
- Constantes.

Linguagem Assembly - Registradores

- A arquitetura MIPS utiliza 32 registradores, chamados de conjunto de registradores ou banco de registradores.
- Registradores são mais rápidos que as memórias.
- Quanto menores os registradores, mais rápido eles podem ser acessados.

Tabela 6: Código mais complexo

Linguagem de alto nível	Código assembly MIPS
$a = b + c;$	# $\$s0 = a$, $\$s1 = b$, $\$s2 = c$ add $\$s0$, $\$s1$, $\$s2$ $\quad \# a = b + c$

Linguagem Assembly - Registradores

Tabela 7: Conjunto de registradores MIPS

Nome	Número	Uso
$0	0	Valor constante 0
$at	1	Assembler temporário
$v0-$v1	2 – 3	Retorna o valor da função
$a0-$a3	4 – 7	Argumentos de função
$t0-$t7	8 – 15	Variáveis temporárias
$s0-$s7	16 – 23	Variáveis salvas
$t8-$t9	24 – 25	Variáveis temporárias
$k0-$k1	26 – 27	Operação de sistemas temporários
$gp	28	Ponteiro global
$sp	29	Ponteiro de pilha
$fp	30	Ponteiro do quadro
$ra	31	Endereço de retorno da função

Linguagem Assembly - Memória

- A memória possui muito mais locais de dados, mas acessá-los leva uma grande quantidade de tempo
- Enquanto o banco de registradores é pequeno e rápido, a memória é grande e lenta.

A Figura 30 mostra um array de memória que é endereçável por palavra. Isto é, cada dado de 32-bits possui um único endereço de 32-bits.

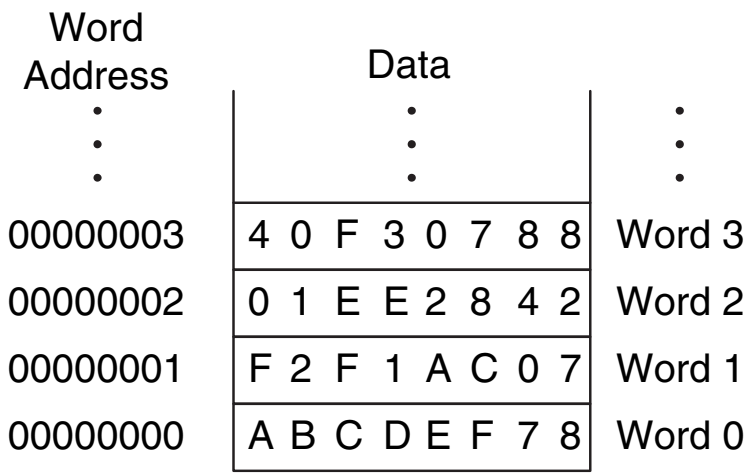


Figura 30: Memória endereçável por byte.

Linguagem Assembly - Memória

A MIPS utiliza a instrução de carregamento de palavra (load word), lw, para ler uma palavra de dado da memória num registrador. O Exemplo a seguir carrega a palavra de memória 1 em $\$s3$.

Tabela 8: Lendo uma memória endereçável por palavra.

Código Assembly
lw $\$s3$, 1($\$0$)

Linguagem Assembly - Memória

- A MIPS utiliza a instrução de armazenamento de palavra, sw, para escrever uma palavra de dado de um registrador na memória.
- O modelo de memória MIPS é endereçável por byte.
- Uma palavra de 32-bits consiste em quatro bytes de 8-bits.

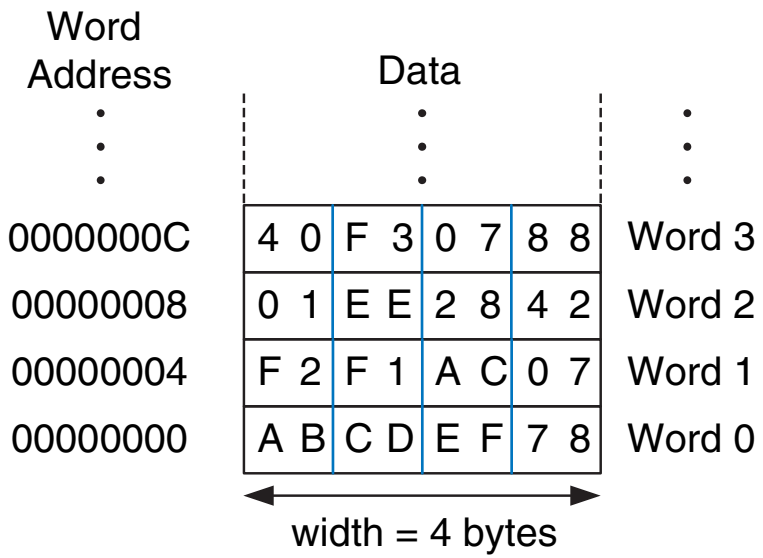


Figura 31: Memória endereçável por byte.

Linguagem Assembly - Memória

Memórias endereçáveis por byte são organizadas nos estilos big-endian e little-endian, como mostrado na Figura 32.

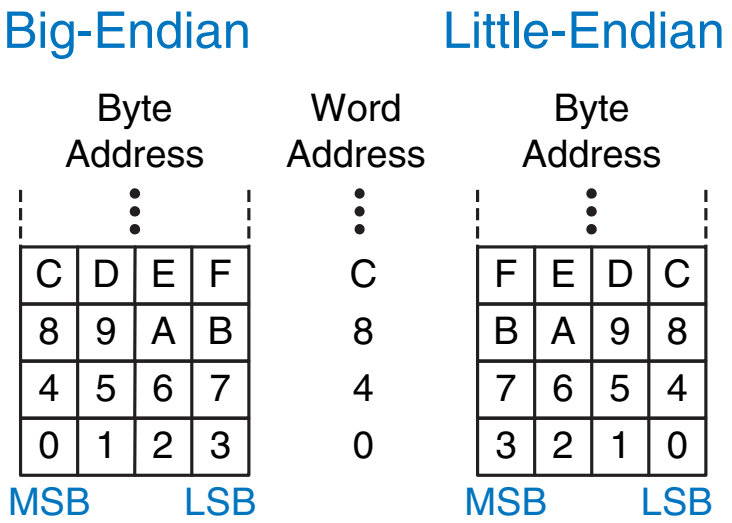


Figura 32: Memória endereçável por byte.

- Em máquinas Big-Endian, os bytes são numerados começando com 0 no byte mais significativo (big end).
- Em máquinas Little-Endian, os bytes são numerados partindo do zero no menos significativo (little end).

Linguagem Assembly - Constantes/Imediatos

As instruções de carregamento e armazenamento de palavras, lw e sw, também ilustram o uso de constantes em instruções MIPS.

Tabela 9: Operandos Imediatos.

Linguagem de alto nível	Código assembly MIPS	
$a = a + 4;$	addi $\$s0$, $\$s1$, 4	$\#a = a + 4$
$b = a - 12;$	addi $\$s1$, $\$s0$, -12	$\#b = a - 12$

Linguagem de Máquina

- Representação utilizando apenas 0's e 1's;
- A arquitetura MIPS utiliza instruções de 32-bits;
- A escolha mais regular é a de codificar todas as instruções em palavras que possam ser armazenadas na memória.
- 3 formatos de instrução:
  - Tipo-R: operam em três registradores;
  - Tipo-I: operam em dois registradores e um imediato de 16-bits;
  - Tipo-J (jump): operam num imediato de 26-bits.

Tipo-R

- “Tipo-R”é uma abreviação para “tipo registrador”;
- Utilizam três registradores como operandos:
  - `rs`, `rt`: registradores fonte;
  - `rd`: registrador destino;
- Outros campos:
  - `op`: operação básica da instrução (opcode);
  - `funct` (função);
  - `shamt` (shift amount): utilizado apenas em operações de deslocamento. Nestas instruções, o valor binário armazenado no campo de 5-bits `shamt` indica a quantidade do deslocamento. Para todas as outras instruções tipo-R, `shamt` é 0;

Tipo-R

op	rs	rt	rd	shamt	funct
6bits	5 bits	5 bits	5 bits	5 bits	6 bits

Exemplo Tipo-R

Código de máquina para as instruções tipo-R `add` e `sub`.

Assembly				Valores do Campo					
				op	rs	rt	rd	shamt	funct
`add`	`$s0`	`$s1`	`$s2`	0	17	18	16	0	32
`sub`	`$t0`	`$t1`	`$t2`	0	11	13	8	0	34
				6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Linguagem de Máquina

op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	(0×02328020)
000000	01011	01101	01000	00000	100010	$(0 \times 016D4022)$
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Exemplo Tipo-R

- Note que o destino é o primeiro registrador numa instrução de linguagem assembly, mas é o terceiro campo numa instrução de linguagem de máquina.

- Por exemplo, a instrução em assembly `add $s0, $s1, $s2` tem `rs = $s1 (17), rt = $s2 (18),` e `rd = $s0 (16).`

Tipo-I

- “Tipo-I”é uma abreviação para “tipo imediato”;
- Utilizam três registradores como operandos:
  - `rs`, `rt`: registradores fonte;
  - `imm`: mantém um imediato de 16-bits;
- Outros campos:
  - `op`: operação básica da instrução (`opcode`);
  - A operação é determinada exclusivamente pelo `opcode`;
  - `rs` e `imm` são sempre utilizados como operandos fonte;
  - `rt` é utilizado como destino por algumas instruções (como `addi` e `lw`), mas também como outra fonte por outras (como `sw`).

Tipo-I

op	rs	rt	imm
6bits	5 bits	5 bits	16 bits

Exemplo Tipo-I

Exemplos de codificação de instruções tipo-I.

Assembly			
`addi`	`$s0,`	`$s1,`	5
`addi`	`$t0,`	`$s3,`	-12
`lw`	`$t2,`	`32($0)`	
`sw`	`$s1,`	`4($t1)`	

Valores do Campo

op	rs	rt	imm
8	17	16	5
8	19	8	-12
35	0	10	32
43	9	17	17
6 bits	9	5 bits	16 bits

Linguagem de Máquina

op	rs	rt	imm				
001000	10001	10000	0000	0000	0000	0101	(0×22300005)
001000	10011	01000	1111	1111	1111	0100	$(0 \times 2268FFF4)$
100011	00000	01010	0000	0000	0010	0000	$(0 \times 8C0A0020)$
101011	01001	10001	0000	0000	0000	0100	$(0 \times AD310004)$
6 bits	5 bits	5 bits	16 bits				

Exemplo Tipo-I

- Valores negativos em imediatos são representados utilizando-se notação de 16-bits com complemento de 2;
- Numa instrução de linguagem assembly, `rt` é listado primeiro, quando utilizado como destino, mas é o segundo campo de registrador numa instrução de linguagem de máquina.

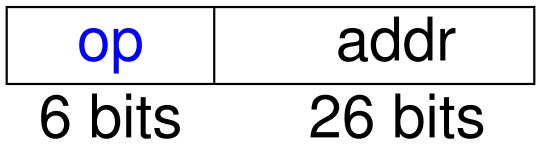
Observe a diferença na ordem dos registradores em assembly e em linguagem de máquina:

```
addi rt, rs, imm
lw rt, imm(rs)
sw rt, imm(rs)
```

Tipo-J

- “Tipo-J”é uma abreviação para “tipo salto (jump)”;
- Esse formato é utilizado apenas em instruções de salto;
- Esse formato de instrução utiliza um único operando de endereço de 26-bits, `addr`;
- Assim como outros formatos, as instruções tipo-J iniciam-se com um `opcode` de 6 bits. Os bits remanescentes são utilizados para especificar o endereço, `addr`.

Tipo-J



Revisão: Formato de Instruções

Tipo-R

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Tipo-I

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Tipo-J

op	addr
6 bits	26 bits

Instruções

Instrução	Exemplo	Semântica	Tipo
add	add $rd,$rs,$rt	rd ← rs + rt	R
sub	sub $rd,$rs,$rt	rd ← rs - rt	R
and	and $rd,$rs,$rt	rd ← rs & rt	R
or	or $rd,$rs,$rt	rd ← rs | rt	R
nor	nor $rd,$rs,$rt	rd ← ~(rs | rt)	R
addi	addi $rt,$rs,cte	rt ← rs + cte	I
andi	andi $rt,$rs,cte	rt ← rs & cte (16 bits lsb)	I
ori	ori $rt,$rs,cte	rt ← rs | cte (16 bits lsb)	I
nori	nori $rt,$rs,cte	rt ← ~(rs + cte) (16 bits lsb)	I
beq	beq $rt,$rs,LABEL	SE rs=rt pule para LABEL	R
slt	slt $rd,$rs,$rt	SE rs<rt SETA rd=01 SENÃO rd=00	R
slti	slti $rt,$rs,cte	SE rs<cte SETA rt=01 SENÃO rt=00	I
lw	lw rt, cte($rs)	rt ← MEM[rs+cte]	I
sw	lw rt, cte($rs)	MEM[rs+cte] ← rt	I
j	j LABEL	pule para LABEL	J

A Potência do Programa Armazenado

- Um programa escrito em linguagem de máquina é uma série de números de 32-bits que representam as instruções;
- Essas instruções podem ser armazenadas na memória;
- Rodar um programa diferente não requer grandes quantidades de tempo e esforço para reconfigurar ou remontar o *hardware*, apenas requer a escrita de um novo programa na memória;
- Instruções num programa armazenado são recuperadas, ou buscadas, na memória e executadas no processador;
- Mesmo programas grandes e complexos são simplificados numa série de leituras de memória e execução de instruções;
- Em programas MIPS, as instruções são normalmente armazenadas partindo do endereço $0\text{x}00400000$.

Programa Armazenado

Para executar o código da Figura 33, o sistema operacional coloca no PC no endereço $0x00400000$. O processador lê a instrução naquele endereço de memória e executa a instrução $0x8C0A0020$. O processador então incrementa o PC em 4, para $0x00400004$, recuperando e executando aquela instrução, e se repete.

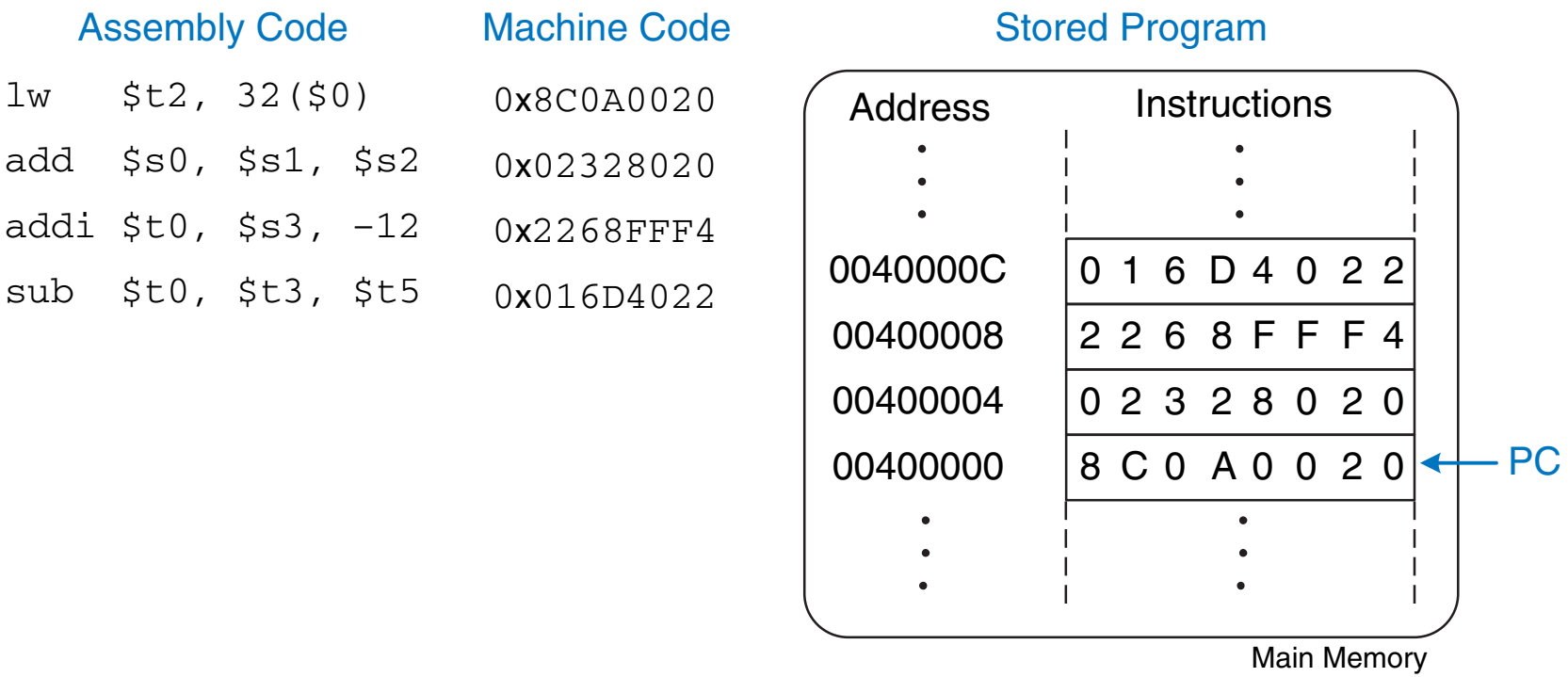


Figura 33: Programa Armazenado.

Interpretando Códigos em Linguagem de Máquina

- Primeiro devem-se decifrar os campos em cada palavra de instrução de 32-bits;

- Diferentes instruções utilizam diferentes formatos, mas todos os formatos iniciam-se com um campo de opcode de 6-bits. Se ele é 0, a instrução é do tipo-R. Caso contrário, ele é do tipo-I ou do tipo-J.

Interpretando Códigos em Linguagem de Máquina

Traduzir $0x2237FFF1$ e $0x02F34022$ em linguagem de máquina para linguagem assembly.

- Representar cada instrução em binário e então olhar os seis bits mais significativos para encontrar o opcode de cada instrução;
- O opcode determina como interpretar os bits restantes;
- Os opcodes são 001000_2 (8_{10}) e 000000_2 (0_{10}), indicando uma instrução `addi` e uma instrução tipo-R;
- O campo `funct` da instrução tipo-R é 100010_2 (34_{10}), indicando uma instrução `sub`.

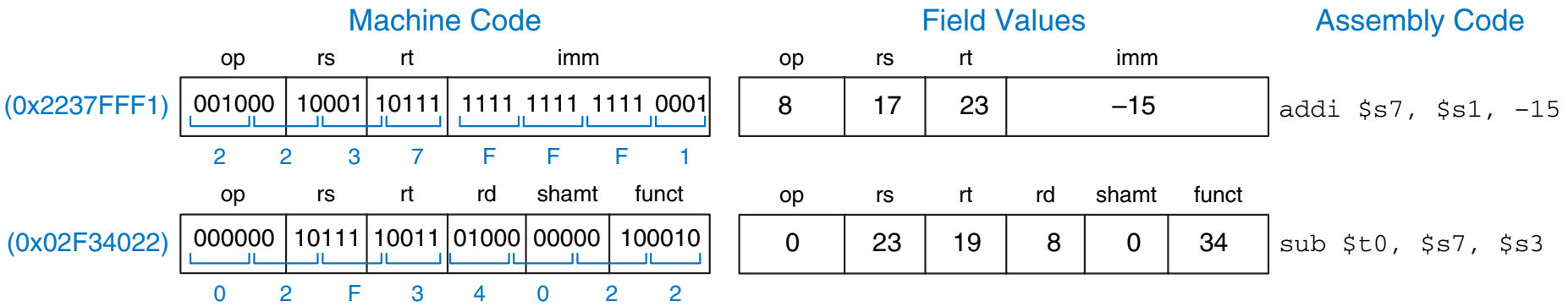
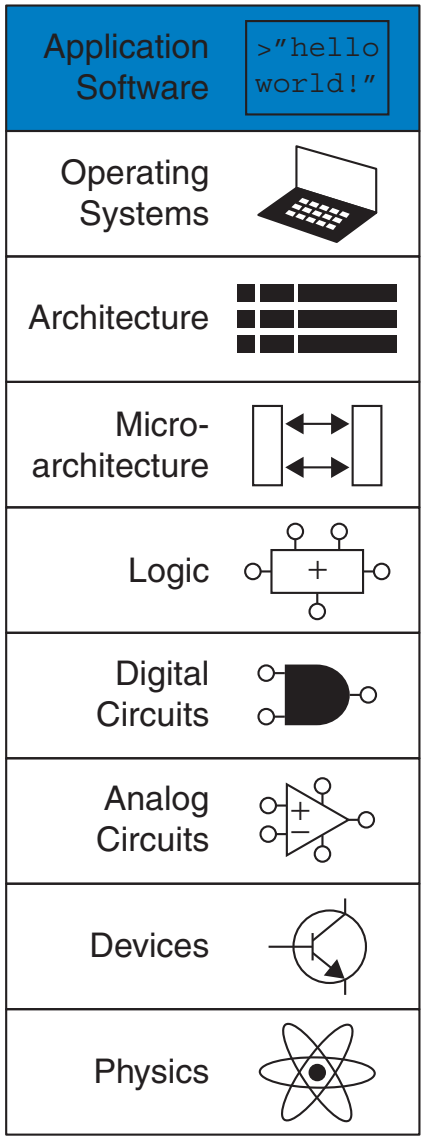


Figura 34: Tradução de código de máquina para código assembly.

Programação em C

- Introdução em C;
- Exemplos de programas;
- Compilação e execução em C;
- Variáveis e tipos de dados;
- Operadores;
- Controle de fluxo;
- Funções e Bibliotecas.



Programação em C

- Uma das linguagens mais populares de programação de todos os tempos é chamada C;
- Ela foi desenvolvida por um grupo que incluía Dennis Ritchie e Brian Kernighan, da Bell Laboratories, entre 1969 e 1973, para re-escrever o sistema operacional *UNIX* a partir de seu código *assembly* original;
- Sua popularidade é fixada por vários fatores:
  - Disponibilidade em diversas plataformas, de supercomputadores a microcontroladores embarcados;
  - Nível moderado de abstração, fornecendo uma produtividade maior que a linguagem *assembly*;
- As seguintes seções descrevem a sintaxe de um programa em C, discutindo as funções, controle de fluxo e bibliotecas.

Rodando um programa C

- O programa é primeiramente compilado na máquina desejada utilizando-se um compilador C;
- Existem versões ligeiramente diferentes do compilador C, incluindo o *cc (C compiler)*, ou *gcc (GNU C compiler)*;
- O *gcc* pode ser baixado gratuitamente e roda diretamente em máquinas *Linux* e pode ser acessado pelo ambiente *Cygwin* em máquinas *Windows*.
- Programa C simples, `hello.c`:
- ```
  // Write "Hello world!" to the console
  #include <stdio.h>
  int main(void){
  printf("Hello world!\n");
  }
  ```
  ```

Rodando um programa C

- Todos os programas devem incluir a função `main`;
- A maioria dos programas utiliza outras funções, definidas em outro lugar do código C e/ou numa biblioteca;
- Cabeçalho: `#include <stdio.h>`: O cabeçalho inclui as bibliotecas de funções necessárias para o programa. Neste caso, o programa utiliza a função `printf`, que é parte da biblioteca I/O padrão, `stdio.h`;
- Função main: `int main(void)`: A execução do programa ocorre rodando-se o código dentro de `main`, o `int` denota que a função `main` devolve, ou retorna, um resultado inteiro;
- Corpo: `{printf("Hello world!\n");` : Chamada para a função `prinf`, a qual imprime a frase `"Hello world"` seguida de um caractere de quebra de linha indicado pela sequência especial `"\n"`.

Rodando um programa C

O processo geral descrito abaixo para a criação de um arquivo C, compilação e execução, é o mesmo para qualquer programa em C;

- 1 - Crie o arquivo texto, por exemplo, `hello.c`;
- 2 - Numa janela do terminal, mude o diretório corrente para o que contém o arquivo `hello.c`, e digite `gcc hello.c` no *prompt de comando*;
- 3 - O compilador cria um arquivo executável. Por padrão, o executável é chamado `a.out` (ou `a.exe`, em máquinas *Windows*);
- 4 - No *prompt de comando*, digite `./a.out` (ou `./a.exe` no *Windows*) e pressione *Enter*;
- 5 - `"Hello world!"` aparecerá na tela.

Compilação

- Brevemente, um compilador é um pedaço de software que lê um programa numa linguagem de alto nível e o converte num arquivo de código máquina chamado de executável;
- Pré-processar o arquivo incluindo as bibliotecas referenciadas;
- Traduzir o código de alto nível em instruções simples nativas para o processador, que são representadas em binário, chamadas código máquina;
- Compilar todas as instruções num único arquivo binário que pode ser lido e executado pelo computador;

#define

- Constantes são nomeadas utilizando-se a diretiva `#define`;
- Globalmente definidas também são chamadas *macros*.

```
#define MAXPERGUNTAS 5
```

#include

- Declarações de variáveis, valores definidos, e definições de funções localizadas num *header file* (cabeçalho) podem ser utilizadas por outro arquivo, adicionando-se a diretiva do pré-processador `#include`;

Variáveis

- As variáveis nos programas em C possuem tipo, nome, valor, e localização na memória;
- Ex: `char x;` , declaração que indica que a variável é do tipo `char` (a qual é um tipo de 1 *byte*), e o nome da variável é x. O compilador decide em que lugar da memória deve-se colocar essa variável de 1 *byte*.

Visão de C da memória

- O C considera a memória como um grupo de *bytes* consecutivos, onde a cada *byte* de memória é atribuído um único número indicando seu endereço, como mostrado na Figura 36;
- Uma variável ocupa um ou mais *bytes* na memória, e o endereço das variáveis de múltiplos *bytes* é indicado pelo número mais baixo do *byte*. O tipo de uma variável indica se o *byte* é interpretado como inteiro, número de vírgula flutuante, ou outro tipo.

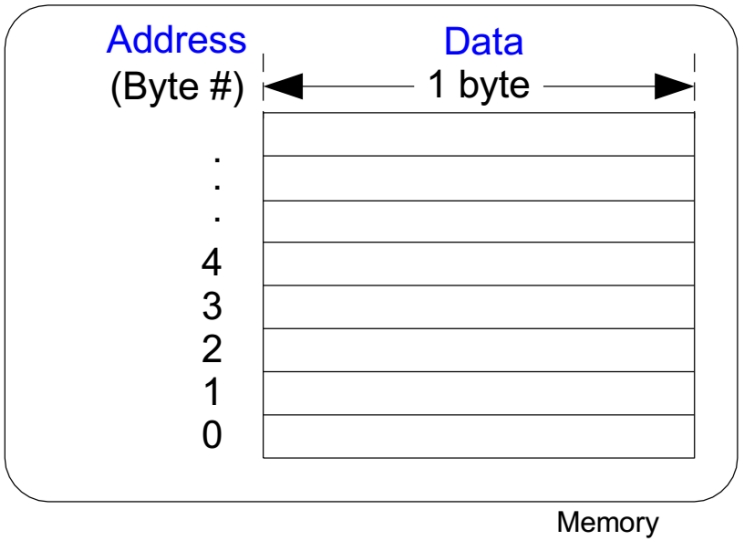


Figura 35: Variáveis na memória.

Tipos Primitivos de Dados

- Os dados podem ser amplamente caracterizados como inteiros, variáveis de ponto flutuante, ou caracteres;
- Um inteiro representa um número em complemento de 2, ou sem sinal, dentro de uma faixa finita;
- Uma variável de ponto flutuante utiliza a representação IEEE de ponto flutuante para descrever números reais com faixa e precisão finitas;
- O tamanho de um tipo `int` é dependente da máquina, geralmente do tamanho da palavra nativa da máquina;
- A Figura 37 lista o tamanho e a faixa de cada tipo primitivo.

Tipos Primitivos de Dados

Type	Size (bits)	Minimum	Maximum
`char`	8	$-2^{-7} = -128$	$2^7 - 1 = 127$
`unsigned char`	8	0	$2^8 - 1 = 255$
`short`	16	$-2^{15} = -32,768$	$2^{15} - 1 = 32,767$
`unsigned short`	16	0	$2^{16} - 1 = 65,535$
`long`	32	$-2^{31} = -2,147,483,648$	$2^{31} - 1 = 2,147,483,647$
`unsigned long`	32	0	$2^{32} - 1 = 4,294,967,295$
`long long`	64	-2^{63}	$2^{63} - 1$
`unsigned long long`	64	0	$2^{64} - 1$
`int`	machine-dependent		
`unsigned int`	machine-dependent		
`float`	32	$\pm 2^{-126}$	$\pm 2^{127}$
`double`	64	$\pm 2^{-1023}$	$\pm 2^{1022}$

Figura 36: Tipos e tamanhos primitivos de dados.

Tipos Primitivos de Dados

- "x" requer um *byte* de dados, "y" requer dois, e "z" requer quatro. O programa decide onde esses *bytes* são armazenados na memória, mas requerem a mesma quantidade de dados;
- Os endereços de x, y e z nesse exemplo são 1, 2 e 4;

```
unsigned char x = 42; short y = -10; unsigned long z = 0;
```

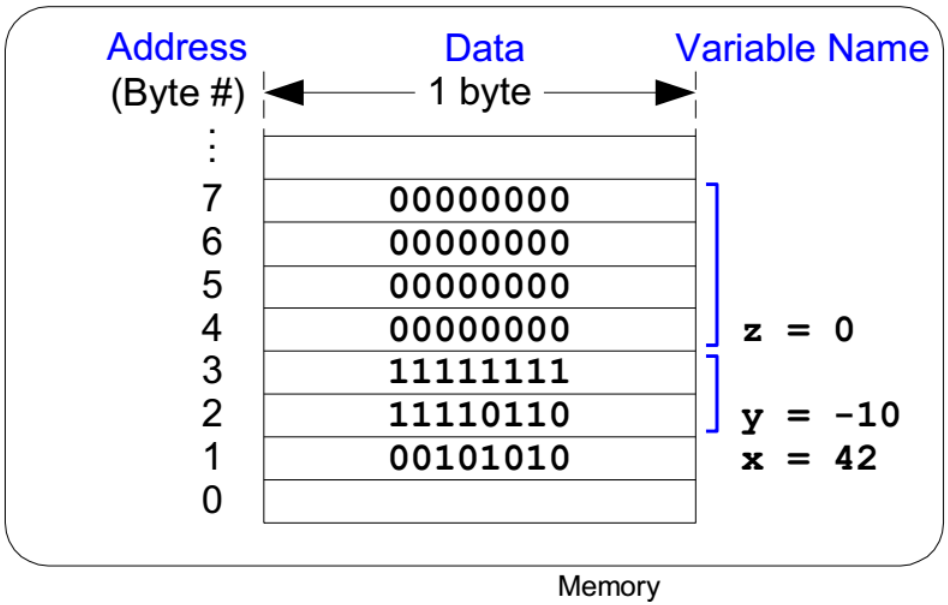


Figura 37: Armazenamento de variáveis na memória.

Operadores em C: Parte 1

Category	Operator	Description	Example
Monadic	++	post-increment	`a++; // a = a+1`
	--	post-decrement	`x--; // x = x-1`
	&	memory address of a variable	`x = &y; // x=the memory // address of y`
	~	bitwise NOT	`z = ~a;`
	!	Boolean NOT	`!x`
	-	negation	`y = -a;`
	++	pre-increment	`++a; // a = a+1`
	--	pre-decrement	`--x; // x = x-1`
	(type)	casts a variable to (type)	`x = (int) c; // cast c to an // int and assign it to x`
	sizeof()	size of a variable in bytes	`long int y; x = sizeof(y); // x = 4`
Multiplicative	*	multiplication	`y = x * 12;`
	/	division	`z = 9 / 3; // z = 3`
	%	modulo	`z = 5 % 2; // z = 1`
Additive	+	addition	`y = a + 2;`
	-	subtraction	`y = a - 2;`

Operadores em C: Parte 2

Category	Operator	Description	Example
Bitwise Shift	<<	bitshift left	`z = 5 << 2; // z = 0b0001 0100`
	>>	bitshift right	`x = 9 >> 3; // x = 0b0000 0001`
Relational	==	equals	`(y == 2)`
	!=	not equals	`(x != 7)`
	<	less than	`(y < 12)`
	>	greater than	`(val > max)`
	<=	less than or equal	`(z <= 2)`
	>=	greater than or equal	`(y >= 10)`
Bitwise	&	bitwise AND	`y = a & 15;`
	|	bitwise OR	`y = a | b;`
	^	bitwise XOR	`y = 2 ^ 3;`
Logical	&&	Boolean AND	`(x && y)`
	||	Boolean OR	`(x || y)`
Ternary	? :	ternary operator	`y = x ? a:b; // if x is true, // y=a, else y=b`

Operadores em C: Parte 3

Category	Operator	Description	Example
Assignment	=	assignment	`x = 22;`
	+=	addition and assignment	`y += 3;` `// y = y + 3`
	-=	subtraction and assignment	`z -= 10;` `// z = z - 10`
	*=	multiplication and assignment	`x *= 4;` `// x = x * 4`
	/=	division and assignment	`y /= 10;` `// y = y / 10`
	%=	modulo and assignment	`x %= 4;` `// x = x % 4`
	>>=	bitwise right-shift and assignment	`x >>= 5;` `// x = x>>5`
	<<=	bitwise left-shift and assignment	`x <<= 2;` `// x = x<<2`
	&=	bitwise AND and assignment	`y &= 15;` `// y = y & 15`
	|=	bitwise OR and assignment	`x |= y;` `// x = x | y`
	^=	bitwise XOR and assignment	`x ^= y;` `// x = x^y`

Chamadas de Função

- Modularidade é a chave para a boa programação;
- Um programa grande é dividido em pequenas partes chamadas funções que, similarmente aos módulos de hardware, possuem entradas, saídas e comportamento bem definidos;
- A função `sum3`, declaração da função começa com o tipo de retorno, `int`, seguido pelo nome, `sum3`, e as entradas contidas entre parênteses `(int a, int b, int c);`
- Chaves `{ }` são utilizadas para limitar o corpo da função, que pode conter zero ou mais declarações. A declaração `return` indica o valor que a função deve retornar;

```
// Retorna a soma de três variáveis de entrada
int sum3(int a, int b, int c) {
int result = a + b + c;
return result;
}
```

Declaração de Controle de Fluxo

- A linguagem C fornece declarações de controle de fluxo para laços condicionais e *loops*;
- Condicionais executam uma declaração apenas se uma condição é alcançada;
- Um *loop* executa repetidamente uma declaração até que uma condição seja alcançada;

Declarações Condicionais

- Declarações `if`, `if/else`, e `switch/case`;

Loops

- `while`, `do/while` e `for` são construtores de loop comuns.

Ponteiros

- Um ponteiro é um endereço de uma variável;
- Ex: `salary1` e `salary2` são variáveis que contém inteiros, e `ptr` é uma variável que pode manter o endereço de um inteiro. O compilador irá atribuir localizações arbitrarias na *RAM* para essas variáveis;

```
// Exemplo de manipulação de ponteiros
int salary1, salary2; // números de 32 bits.
int *ptr;             // especificando o endereço.
salary1 = 67500;      // salary1 = \$67,500 = 0x000107AC.
ptr = &salary1;       // ptr = 0x0070, endereço de salary1.
salary2 = *ptr + 1000;
```

- `/*` Dereferência `ptr` para dar o conteúdo do endereço `70 = \$67.500`, então adiciona `\$1.000` e define `salary2` como `\$68.500 */`.

Ponteiros

- Ao usar uma variável do tipo ponteiro, do operador "*" “dereferência” um ponteiro, retornando o valor armazenado no endereço de memória indicado contido no ponteiro;
- O operador & é pronunciado “endereço de”, e ele produz o endereço de memória da variável sendo referenciada;

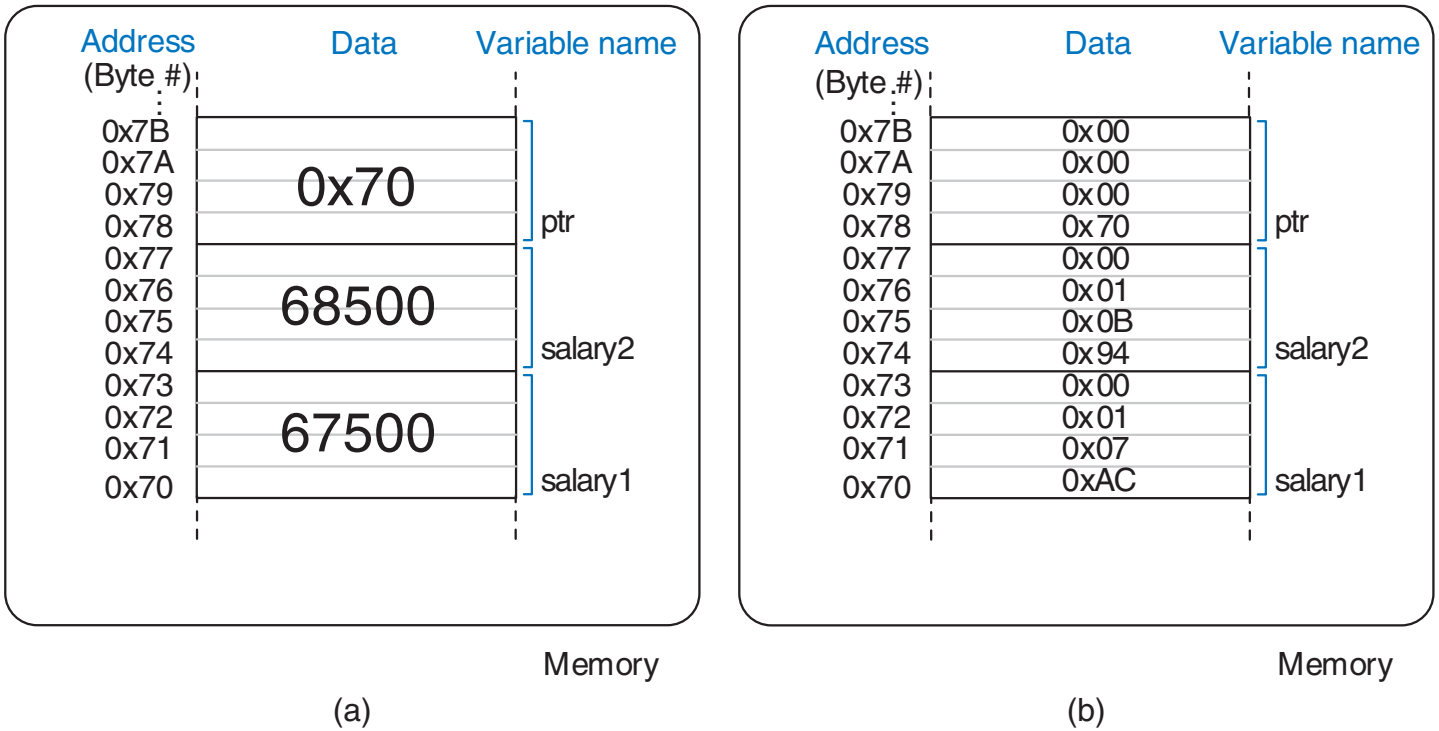


Figura 38: Conteúdos de memória (a) como valor e (b) por *byte* usando memória *little endian*.

Bibliotecas padrão

- Programadores comumente utilizam uma variedade de funções padrão, como de impressão e trigonométricas;

stdio

- Biblioteca padrão de entrada/saída, `stdio.h` ;
- `#include <stdio.h>` no topo do arquivo C;
- A declaração *print formatted*, `printf`, mostra texto na consola. O seu argumento de entrada requerido é uma *string* entre aspas `" "`.

```
// Simples função de impressão
#include <stdio.h>
int num = 42;
int main(void) {
printf("A resposta é %d.\n", num);
}
```

Bibliotecas padrão

math

- A biblioteca matemática `math.h` fornece funções matemáticas comumente utilizadas;
- Incluir `#include <math.h>`;

```
// funções matemáticas
#include <stdio.h>
#include <math.h>
int main(void) {
float a, b, c, d, e, f, g, h;
a = cos(0); // 1, note: o argumento em radianos
b = 2 * acos(0); // pi (acos significa arc cosseno)
c = sqrt(144); // 12
d = exp(2); // e^2 = 7.389056,
printf("a = %.0f, b = %f, c = %.0f, d = %.0f,
\n",a, b, c, d);
}
```