Resumo - Teste 1

Bloco 1 - Representação da informação nos computadores

- [5] Princípios básicos da arquitetura de computadores

  - → Abstração - Esconder os detalhes sempre que não são necessários
  - → Disciplina - Restringuir intencionalmente as liberdades
  - → Hierarquia - Dividir um sistema em módulos e sub-módulos
  - → Modularidade - Cada um dos módulos tem interfaces bem definidos
  - → Regularidade - Sempre que possível reutilizar os módulos já disponíveis

- Abstração Digital: Considerar apenas um subconjunto discreto de todos os valores possíveis

- Quantidade de informação: A quantidade de informação numa variável descreta com N níveis:

  $$D = \log_2 N \quad (\text{bits}), \quad N \rightarrow \text{Níveis}, \quad D \rightarrow \text{número de bits}$$

  ex: Semáforo com R, G, Y

  $2^1 = 2$, $2^2 = 4$, $2^3 = 8$

  $D = \lceil \log_2 3 \rceil = 2$ ← menor inteiro superior

  ⚠ Se não considerarmos a abstração digital $D = \log_2(+\infty) = +\infty$, o que seria impossível para guardar em memória ($+\infty$ bits)

- Gama de representação: Conjunto de números que podemos utilizar com N algarismos

  - → Decimal: $[0, 10^N - 1]$
  - → Binário: $[0, 2^N - 1]$
    - Sinal e Módulo: $[-(2^{N-1} - 1), 2^{N-1} - 1] = [-2^{N-1} + 1, 2^{N-1} - 1]$ ← 1 bit de sinal
    - Complemento para 2: $[-2^{N-1}, 2^{N-1} - 1]$

• Binário:

→ Com N bits, tem a forma: $b_{N-1}\ b_{N-2} \dots b_2\ b_1\ b_0$

→ Cada bit tem o valor 1 ou 0

→ Em decimal o seu valor é a soma: $2^{N-1} b_{N-1} + \dots + 2^1 b_1 + 2^0 b_0$

→ Gama de representação: $[0, 2^N - 1]$

• Potências de 2:

$2^0 = 1$

$2^1 = 2$

$2^2 = 4$

$2^3 = 8$

$2^4 = 16$

$2^5 = 32$

$2^6 = 64$

$2^7 = 128$

$2^8 = 256$

$2^9 = 512$

$2^{10} = 1024$

$2^{11} = 2048$

• Com 3 bits:

Binário	→	Decimal
0 0 0		0 ⊛
0 0 1		1
0 1 0		2
0 1 1		3
1 0 0		4
1 0 1		5
1 1 0		6
1 1 1		7 ⊛

⊛ Gama de representação $[0, 2^3 - 1] = [0, 7]$ ✓

• Hexadecimal:

→ Um algarismo hexadecimal pode ser: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

→ A base 16 funciona agrupando bits em grupos de 4 ($2^4 = 16$)

→ Em decimal o seu valor é a soma: ex:. $A4F_{16} = \underset{10_{10}}{10} \times 16^2 + \underset{4_{10}}{4} \times 16^1 + \underset{15_{10}}{15} \times 16^0{}_{10}$

ex:. $\underbrace{1010}_{A_{16}}\underbrace{0011}_{3_{16}}{}_2 = A3_{16}$

• Octal:

→ Um algarismo hexadecimal pode ser: 0, 1, 2, 3, 4, 5, 6, 7

→ A base 8 funciona agrupando bits em grupos de 3 ($2^3 = 8$)

→ Em decimal o seu valor é a soma: ex:. $741_{16} = \underset{7_{10}}{7} \times 8^2 + \underset{4_{10}}{4} \times 8^1 + \underset{1_{10}}{1} \times 8^0$

ex:. $\underbrace{1010}_{A_{16}}\underbrace{0011}_{3_{16}}{}_2 = A3_{16}$

- **Sistema de numeração (Tabela):** (4 bits)

Decimal	Binário	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

→ Byte: conjunto de 8 bits (pode representar $2^8 = 256$ valores distintos, [0, 255])

→ Nibble: meio byte, conjunto de 4 bits [0, 15]

→ Palavra: conjunto de bits sobre o qual o processador opera (ex: 64 bits)

→ lsb: bit menos significativo (1001) — lsb
→ msb: bit mais significativo (1001) — msb

→ LSB/MSB: byte menos e mais significativo

Kilo byte $\simeq 10^3 = 1000$

→ Kilo byte : $2^{10} = 1024$ bytes

→ Mega byte : $2^{20} = 1\,048\,576$ bytes $\simeq 10^6$

→ Giga byte : $2^{30} = 1\,073\,741\,824$ bytes $\simeq 10^9$

(...)

• Número com sinal e módulo:

→ O bit menos significativo representa o sinal e os restantes o módulo

→ Gama de representação: $[-2^{N-1}+1 , 2^{N-1}-1]$

• Desvantagens: dupla representação do zero (+0 e -0) → 1000 (-0), 0000 (+0)

operações aritméticos não são imediatos

• Vantagem: intuitiva

• Complemento para 2:

↳ O que realmente é utilizado no computador ☝

→ Igual à representação sem sinal, exceto que o msb (bit mais significativo vale -2^{N-1}

→ Gama de representação: $[-2^{N-1} , 2^{N-1}-1]$

→ Calcular o complemento para 2 de um número consiste em inverter todos os bits e somar 1

→ Extender os números: 0000 0111 = 7; 1111 1000 = -8

• Vantagem: somas funcionam da mesma forma que os números sem sinal

↳ Não esquecer:

Temos overflow na soma se: → 2 positivos a soma é negativa

→ 2 negativos a soma é positiva

↳ Mas nunca ocorre com números de sinal diferente ☝

· Vírgula Fixa:

→ Erro de representação (precisão) é em função dos algarismos da parte fracionária

→ Tem a forma: $b_{N-1} \dots b_1 b_0 , \underbrace{b_{-1} b_{-2} b_{-3} \dots}_{\text{Parte fracionária}}$

→ Somos e subtrações são iguais em complemento para 2 e em número sem sinal, desde que, os vírgulas estejam alinhados

· O erro máximo é sempre metade do valor do último digito representado

ex:. 123,45 ou $b_2 b_1 b_0 , b_{-1} b_{-2}$

$10^{-2} = 0{,}01$ $\qquad$ $2^{-2} = 0{,}25$

$\frac{10^{-2}}{2} = 0{,}005$ $\quad < \quad$ $\frac{2^{-2}}{2} = 0{,}125$

· Problema: Valores muito grandes (ou muito pequenos) em vírgula fixa requer uma enorme quantidade de algarismos $\qquad n_2 = \lceil n_1 \times \log_{n_2} r_1 \rceil$

em binário ↓ ? solução em decimal: $6{,}02 \times 10^{23}$

· Vírgula Flutuante:

→ $X = \pm \underbrace{1}_{\neq 0,\ \text{logo tem de ser 1 ?}} , m \times 2^{exp}$ ← Expoente

Compromisso entre a precisão e a gama de representação:

- · Mantissa: conjunto de números finitos que determinam a precisão (distância entre 2 pontos consecutivos)
- · Expoente: determina a gama de representação

↳ **Formato IEEE 754** (formato global para utilizar vírgula flutuante)

$X = (-1)^S \times 1, \text{Mantissa} \times 2^{(Exp - Bios)}$

S	Exp	Mantissa

Bios

→ Precisão simples: 1 bit + 8 bits + 23 bits = 32 bits — Bios: 127

→ Precisão dupla: 1 bit + 11 bits + 52 bits = 64 bits — Bios: 1023

→ Casos especiais:

Número representado em decimal	S	Exp	Mantissa
0	X	0000 0000	000 0000 0000 0000 0000 0000
$+\infty$	0	1111 1111	000 0000 0000 0000 0000 0000
$-\infty$	1	1111 1111	000 0000 0000 0000 0000 0000
NaN	X	1111 1111	Non-zero

Ex:

[1] Que número está representado em formato IEEE 754 precisão simples como:

1011 1111 0100 0000 0000 0000 0000 0000 → 32 bits

Sinal: 1 bit | Expoente: 8 bits | Mantissa: 23 bits

$X = (-1)^{1} \times 1,100 \times 2^{(126-127)}$ (Bias = 127)

$= -1,1_2 \times 2^{-1} = -\left(1+\frac{1}{2}\right)\times\frac{1}{2} = -(0,5+0,25) = -0,75_{10}$

$0111\ 1110 = 2^6+2^5+2^4+2^3+2^2+2^1$
$= 64+32+16+8+4+2$
$= 126$

[2] Determine a representação em formato IEEE 754 precisão simples do número:

$5.5_{10} = 101,10$
$= 1,0110 \times 2^2$
$= (-1)^0 \times 1,0110 \times 2^{(129-127)}$ (Bias)

$0,5 \times 2 = 1,0$
$0,0 \times 2 = 0,0$
⋮

1 bit → Sinal: 0

8 bits → Expoente: $129_{10} = 1000\ 0001$ ($129_{10} = 128+1 = 1000\ 0001$)

23 bits → Mantissa: 0110 0000 0000 0000 0000 000

$X = (-1)^{S} \times 1,\text{Mantissa} \times 2^{(\text{Expoente}-\text{Bias})} = (-1)^0 \times 1,0110 \times 2^{(129-127)}$

Sinal | Expoente | Mantissa

0 100 0000 1011 0000 0000 0000 0000 0000 → (0x40B0 0000)

[3] Somar os números seguintes representados na norma IEEE 756:

0x3FC00000 + 0x40500000

0 011 1111 1 100 0000 0000 0000 0000 0000 + 0 100 0000 0 101 0000 0000 0000 0000 0000

$0111\ 1111_2 = 127_{10}$

$(-1)^0 \times 1,1_2 \times 2^{(127-127)} + (-1)^0 \times 1,101 \times 2^{(128-127)}$

$= 1,1_2 \times 2^0 + 1,101_2 \times 2^1 = 0,11_2 \times 2^1 + 1,101_2 \times 2^1$

$= \begin{array}{r} 0,110 \\ +1,101 \\ \hline 10,011 \end{array} \times 2^1 = 1,0011_2 \times 2^2 = 1,0011_2 \times 2^{(129-127)}$

129_{10}

0 100 0000 1001 1000 0000 0000 0000 0000 = 0x40980000

· Overflow:

→ Overflow significa que o valor não é representável com o nº de bits disponível

→ As máquinas digitais operam com um nº fixo de bits

→ Se o resultado de uma soma exceder o nº de bits disponível

(ex: em complemento para 2 ocorre frequentemente pois temos de especificar antes o número de bits disponíveis)

· ASCII:

→ American Standart Code for Information Interchange

→ É um código binário que codifica 128 sinais (1 byte cada)

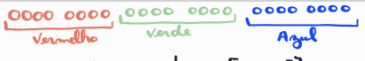
- 95 gráficos (letras, caracteres, símbolos matemáticos, ...)
- 33 controlo (Esc, Del, mudanças de linha, ...)

→ O código ASCII permite representar o alfabeto latino

Nota: O standart UNICODE representa os caracteres em 8, 16 e 32 bits.
Sendo a representação de 8 bits (UTF-8) compatível com código ASCII

· Outro tipo de dados:

→ Imagens: tratadas como array de pixeis → Monocromados: 1 bit ⇒ branco ou preto
→ Cor: RGB ⇒ 8 bits para intensidade de cada cor



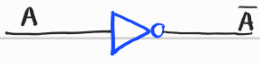
Para cada cor gama de representação = [0, 255]

→ Som: sequência de números armazenados que representam a amplitude sonora ao longo do tempo

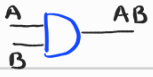
Bloco 2 - Lógica Binária e Álgebra de Boole

Operações lógicas:

A	NOT A $= \overline{A}$
0	1
1	0



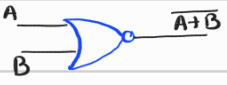
A	B	A AND B $= AB$
0	0	0
0	1	0
1	0	0
1	1	1



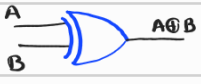
A	B	A NAND B $= \overline{AB}$
0	0	1
0	1	1
1	0	1
1	1	0

A	B	A OR B $= A+B$
0	0	0
0	1	1
1	0	1
1	1	1

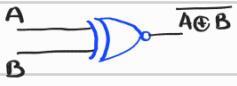
A	B	A NOR B $= \overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0



A	B	A XOR B $= A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0



A	B	A XNOR B $= \overline{A \oplus B}$
0	0	1
0	1	0
1	0	0
1	1	1

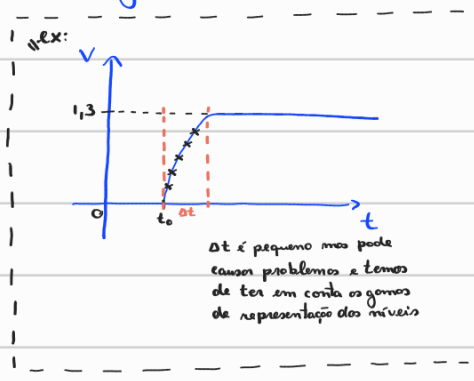


• Níveis lógicos:

→ Os valores existem no circuito como tensões:

- 0 corresponde a ground ou 0 V → GND
- 1 corresponde à tensão de alimentação → VCC
  ≃1,3 V

→ Para resolver o problema dos tensões intermédios há uma gama de tensões que correspondem ao nível lógico 0 e uma gama de representação que corresponde ao 1 0.1 V ⇝ nível 0
1,1 V ⇝ nível 1

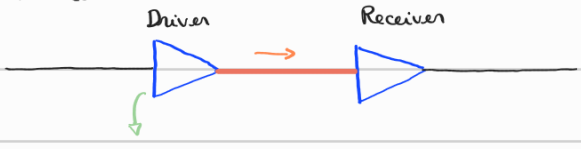


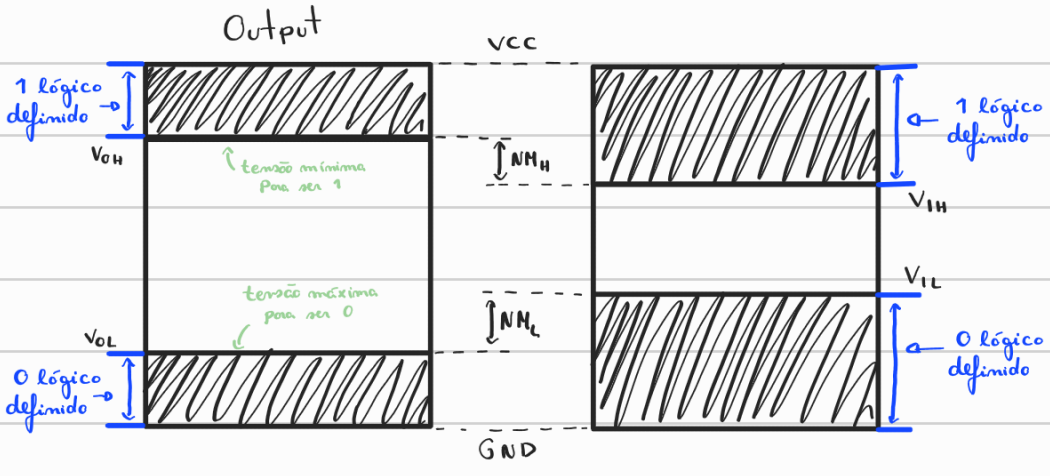
• Ruído:

→ O que é? • Tudo o que degrade a qualidade do sinal
• Pode ser uma resistência, imperfeições de fonte...

→ Para resolver definimos uma margem de ruído

Margem de Ruído:





$$NM_H = V_{OH} - V_{IH} \qquad NM_L = V_{IL} - V_{OL}$$

· Circuitos lógicos:

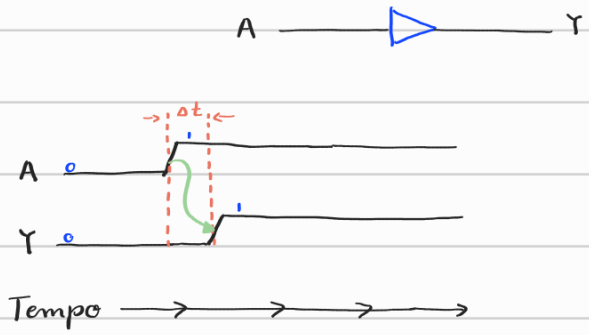
→ Lógica combinatória: Sem memória, as saídas são determinadas apenas pelas entradas

→ Lógica sequencial: Com memória, as saídas são determinadas pelas entradas atuais e passadas

(importante considerar!)

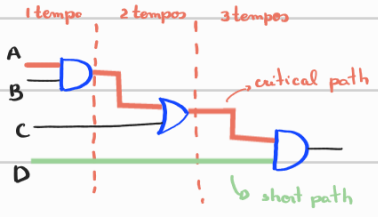
· Considerações sobre o tempo:

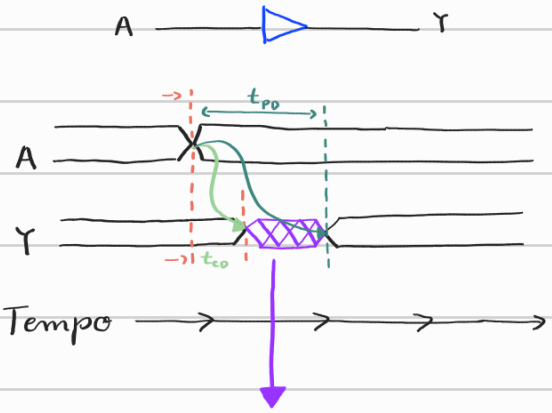
→ A propagação não é instantânea:



· Logo, temos de ter algumas considerações:

Podem acontecer:





t_{PD}: tempo de propagação (tempo máximo que demora a propagar-se da entrada para a saída)

t_{CD}: tempo de contaminação (tempo mínimo que demora a propagar-se da entrada para a saída)

Glitch: Quando apenas uma mudança na entrada produz mais do que uma variação na saída (variações rápidos não previstos)

"metaestabilidade"

· Álgebra de Boole:

Definições:

→ Complemento: $\bar{A}, \bar{B}, \bar{C}$

→ Literal: $A, \bar{B}, C$

→ Implicantes: $AB, \bar{A}B, A\bar{B}\bar{C}$ (produto de literais)

→ Mintermos: $A\bar{B}C, \bar{A}BC, \bar{A}\bar{B}\bar{C}$ (produto que inclui todos os variáveis)

→ Maxtermos: $A+\bar{B}+C, \bar{A}+B+C, \bar{A}+\bar{B}+\bar{C}$ (soma que inclui todos os variáveis)

Por definição a prioridade dos operadores é: NOT > AND > OR

→ Tabela de verdade pode sempre ser descrita como SoP e PoS:

SoP ⟹ 1ª FC: Soma de produtos (mintermos)

PoS ⟹ 2ª FC: Produto de somas (maxtermos)

ex:

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

SoP: $Y = \bar{A}\bar{B} + AB$

ou

PoS: $Y = (A+\bar{B}) \cdot (\bar{A}+B)$

Mais exemplos à frente!

· Axiomas:

Valor binário → $B=0$ se $B \neq 1$

NOT → $\overline{0} = 1$

AND → $0 \cdot 0 = 0$

$1 \cdot 1 = 0$

$0 \cdot 1 = 1 \cdot 0 = 0$

· Teoremas:

1 variável:

Teorema dual

Identidade → $x \cdot 1 = x$ ou $x + 0 = x$

Elemento Absorvente → $x \cdot 0 = 0$ ou $x + 1 = 1$

Idempotência → $x \cdot x = x$ ou $x + x = x$

Involução → $\overline{\overline{x}} = x$

Complementoridade → $x \cdot \overline{x} = 0$ ou $x + \overline{x} = 1$

2 variáveis:

Comutatividade → $x \cdot y = y \cdot x$ ou $x + y = y + x$

Associatividade → $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ ou $(x + y) + z = x + (y + z)$

Distributividade → $(x + y) \cdot (x + z) = x + (y \cdot z)$ ou $(x \cdot y) + (x \cdot z) = x \cdot (y + z)$

Diferente da algebra convencional

Absorção → $x \cdot (x + y) = x$ ou $x + (x \cdot y) = x$

Adjacência → $(x \cdot y) + (x \bar{y}) = x$ ou $(x + y) \cdot (x + \bar{y}) = x$

Simplificação → $(x + \bar{y}) \cdot y = xy$ ou $x \bar{y} + y = x + y$

De Morgan → $\overline{x \cdot y} = \bar{x} + \bar{y}$ ou $\overline{x + y} = \bar{x} \cdot \bar{y}$

Nota: Uma forma de provar os teoremas é por indução perfeita

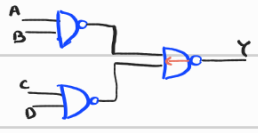
— // —

"Bubble Pushing":

Empurram-se os bolhos:

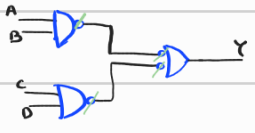
$\Rightarrow \overline{AB} = \bar{A} + \bar{B}$ ✓ De Morgan

Ex:



$\rightarrow Y = \overline{\overline{AB} \cdot \overline{CD}} \overset{\text{De Morgan}}{=} \overline{\overline{AB}} + \overline{\overline{CD}} \overset{\text{Involução}}{=} AB + CD$

||



$\rightarrow Y = AB + CD$ ✓

Duas negações seguidas

· Conjunto completo de operadores:

→ Conjunto de operadores que permite a implementação de qualquer função booleana:

- {AND, OR, NOT}
- {AND, NOT}
- {OR, NOT}
- {NAND}
- {NOR}

Conjunto completo com apenas um tipo de portos facilita o trabalho de quem projeta os circuitos

Podemos reescrever as formas canónicas só com um tipo de portos lógicos

↓

Regularidade favorece facilidade ☺

· Todas as formas canónicas:

→ 1.ª forma canónica SoP (AND-OR)

→ 2.ª forma canónica PoS (OR-AND)

→ 3.ª forma canónica SoP (NAND-NAND)

→ 4.ª forma canónica PoS (NOR-NOR)

ex:

$$f(x,y,z) = \underset{1}{x}\,\underset{1}{y} + \underset{0}{\overline{z}}$$

x	y	z	f
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

1.ª FC (SoP) → $\overline{x}\,\overline{y}\,\overline{z} + \overline{x}\,y\,\overline{z} + x\,\overline{y}\,\overline{z} + x\,y\,\overline{z} + x\,y\,z$ (AND-OR)

↓ NAND

3.ª FC → $\overline{\overline{x}\,\overline{y}\,\overline{z}} \cdot \overline{\overline{x}\,y\,\overline{z}} \cdot \overline{x\,\overline{y}\,\overline{z}} \cdot \overline{x\,y\,\overline{z}} \cdot \overline{x\,y\,z}$ (NAND-NAND)

2.ª FC (PoS) → $(x+y+\overline{z}) \cdot (x+\overline{y}+\overline{z}) \cdot (\overline{x}+y+\overline{z})$ (OR-AND)

↓ NOR

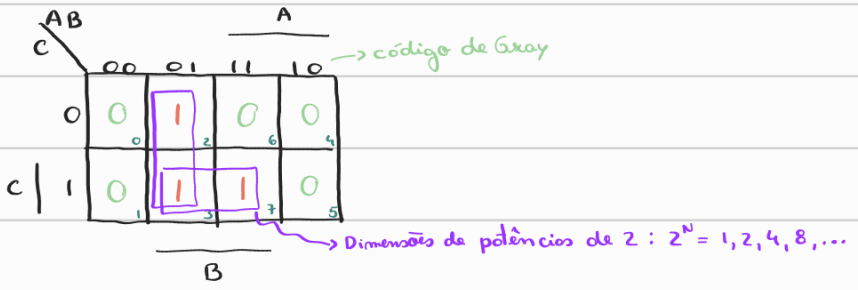
4.ª FC → $\overline{(x+y+\overline{z})} + \overline{(x+\overline{y}+\overline{z})} + \overline{(\overline{x}+y+\overline{z})}$ (NOR-NOR)

Como todo o processo de simplificação é chato e demorado, para uma simplificação mais fácil utilizamos ...

· Mapas de Karnaugh: (bastante fácil, basta exemplificar)

1ºex:

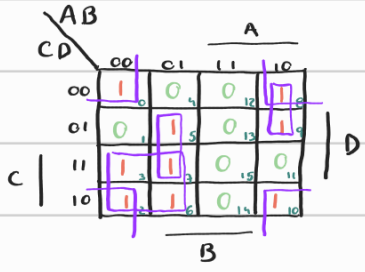
	A	B	C	Y
0-	0	0	0	0
1-	0	0	1	0
2-	0	1	0	1
3-	0	1	1	1
4-	1	0	0	0
5-	1	0	1	0
6-	1	1	0	0
7-	1	1	1	1



$$Y = \bar{A}B + BC$$

2ºex:

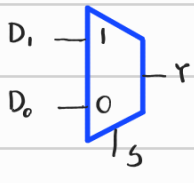
	A	B	C	D	Y
0-	0	0	0	0	1
1-	0	0	0	1	0
2-	0	0	1	0	1
3-	0	0	1	1	1
4-	0	1	0	0	0
5-	0	1	0	1	1
6-	0	1	1	0	1
7-	0	1	1	1	1
8-	1	0	0	0	1
9-	1	0	0	1	1
10-	1	0	1	0	1
11-	1	0	1	1	0
12-	1	1	0	0	0
13-	1	1	0	1	0
14-	1	1	1	0	0
15-	1	1	1	1	0



$$Y = \bar{B}\bar{D} + \bar{A}C + \bar{A}BD + A\bar{B}\bar{C}$$

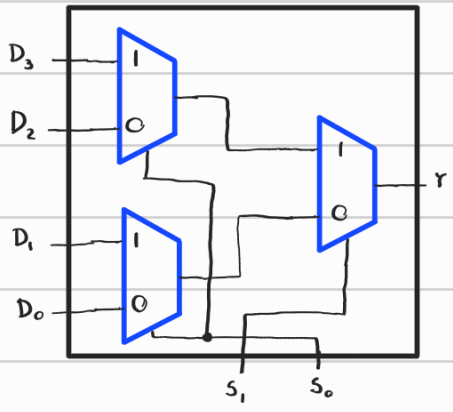
Bloco 3 - Blocos Combinatórios básicos / Circuitos Sequenciais / Máquinas de Estado

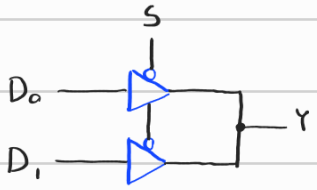
· Multiplexer:



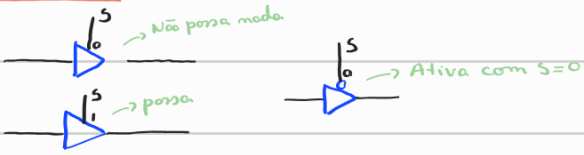
S	Y
0	D_0
1	D_1

Mux 4:1



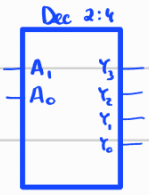


→ Porta tristate:



· Descodificador:

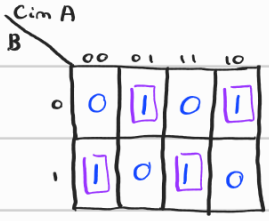
→ O descodificador ativa a saída correspondente ao número que está representado na entrada



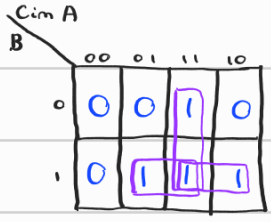
A_1 A_0	Y_3 Y_2 Y_1 Y_0
0 0	0 0 0 1
0 1	0 0 1 0
1 0	0 1 0 0
1 1	1 0 0 0

· Somador completo de um bit:

	Cin	A	B	S	cout
0-	0	0	0	0	0
1-	0	0	1	1	0
2-	0	1	0	1	0
3-	0	1	1	0	1
4-	1	0	0	1	0
5-	1	0	1	0	1
6-	1	1	0	0	1
7-	1	1	1	1	1

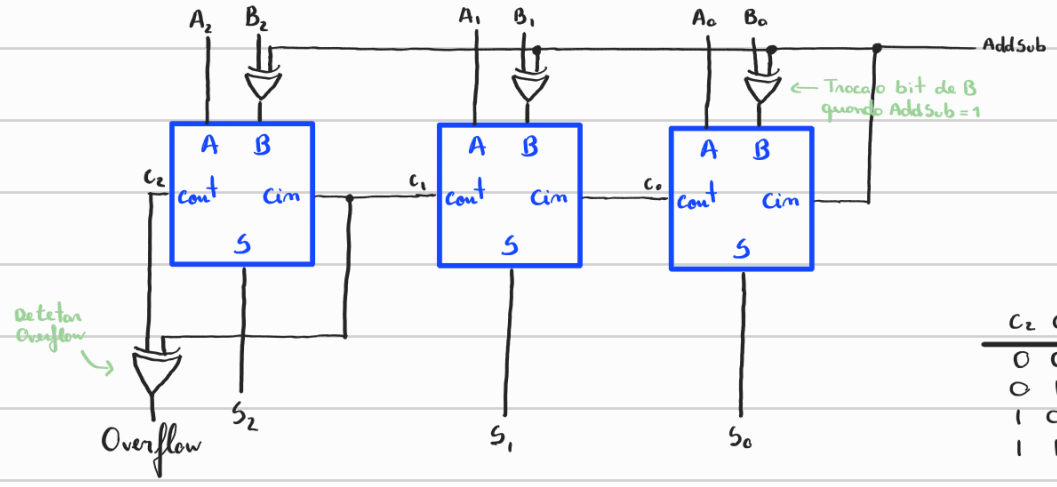


$S = \overline{cin}\,A\,\overline{B} + \overline{cin}\,\overline{A}\,B + cin\,A\,B + cin\,\overline{A}\,B = cin \oplus A \oplus B$



$Cout = cin\,B + cin\,A + A\,B$

ex: Somador/subtrator de 3 bits com deteção de overflow



AddSub	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

$Y = AddSub \oplus B$

C_2 C_1	Overflow
0 0	0
0 1	1
1 0	1
1 1	0

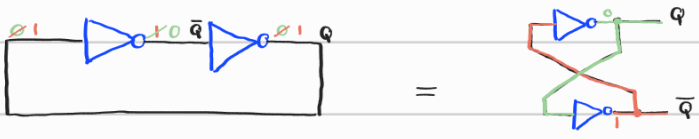
$Overflow = C_2 \oplus C_1$

↳ ▽ Esta solução tem um tempo de atraso muito grande porque a soma do último bit depende do primeiro (os tempos de atraso de cada somador propagam-se, existem melhores implementações)

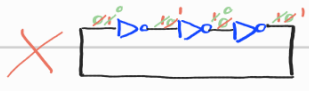
· Circuitos Sequenciais:

→ Circuitos sequenciais são aqueles em que a saída depende do valor atual e do valor passado dos entrados

→ Circuito bi-estável: (estável em 1 ou em 0)

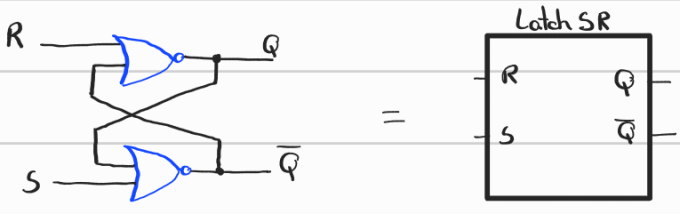


▽ Nem todos os loops geram memórias



ex: Oscilador (não estabiliza)

· Latch SR:



R S	Q
0 0	Qanterior
0 1	1 (Set)
1 0	0 (Reset)
1 1	Inválido

Armazena 1 bit

↓ Estado proibido

· Latch D:

dados — determina quando os dados são escritos

→ Latch D tem duas entradas D e clk

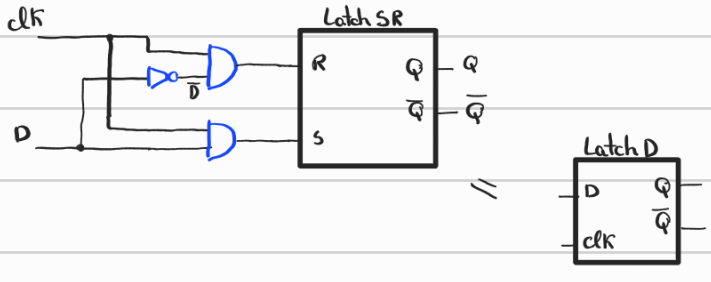
→ Funcionamento:

clk	D	Q
0	X	Qanterior
1	0	0
1	1	1

⇒

clk	Q
0	Qanterior
1	D

Evita o estado inválido do Latch SR

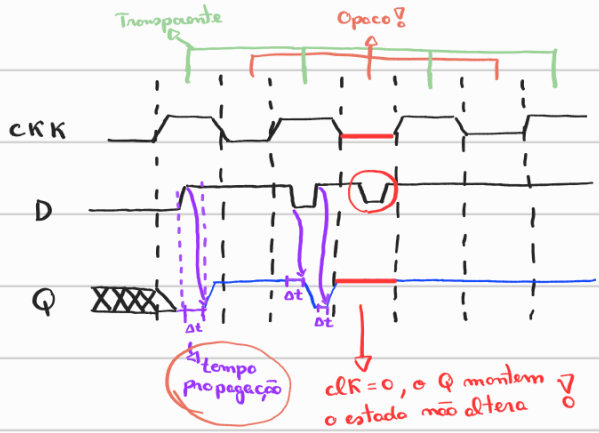


clk	D	$\overline{D}$	S	R	Q	$\overline{Q}$
0	X	$\overline{X}$	0	0	Qanterior	$\overline{Q}$anterior
1	0	1	0	1	0	1
1	1	0	1	0	1	0

Reset

Set

Latch SR nunca vai para o estado inválido

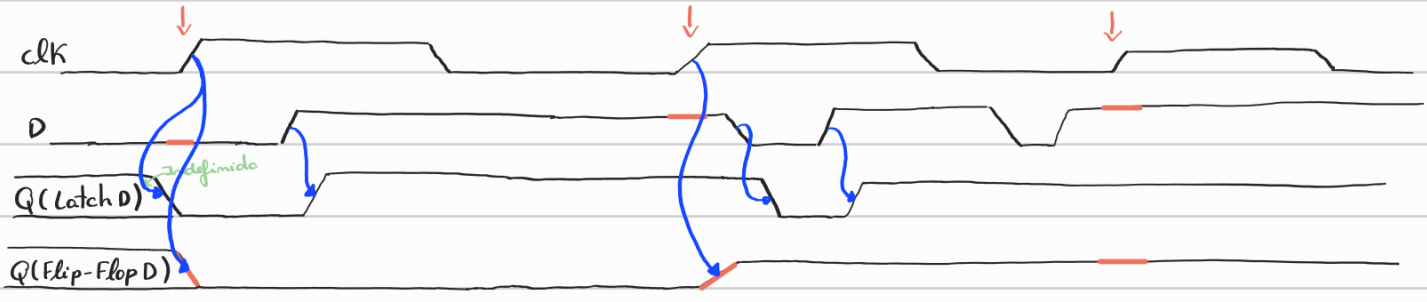


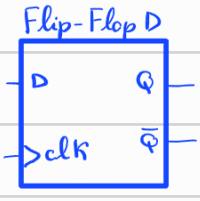
→ Problema: Latch D é transparente de mais ☹ (pode causar problemas por causa dos glitches)

→ Solução: Flip-Flop D

· Flip-Flop D

→ Igual ao Latch D mas só é ativo quando o clock transita de 0 para 1, edge-triggered. (apenas em um instante a escrita é efetuada)





→ Pode ter reset, enable ou set síncronos. (Prioridade normal: Reset > Enable > Set)

↓ síncrono ou assíncrono

• Memórias: (Arrays de Flip-Flops D)

→ Armazenar de modo eficiente grandes quantidades de memória

→ Os três tipos mais comuns são:

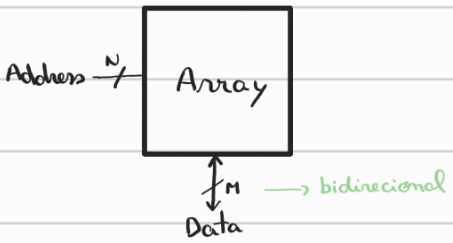
- SRAM → Static Random Acess Memory
- DRAM → Dynamic Random Acess Memory

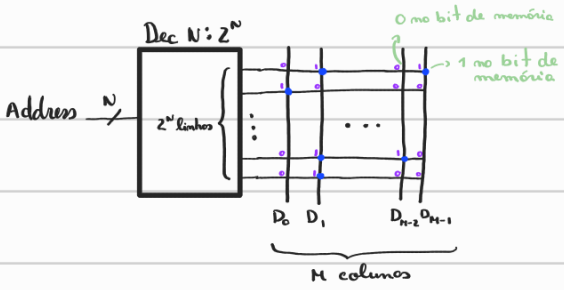
} Voláteis → Necessitam de alimentação, quando é desligada a memória desaparece

Falamos mais para a frente ⊛

- ROM → Read Only Memory

} Não voláteis → Permace independentemente da alimentação





→ N bits de endereço e M bits de dados, a memória tem:

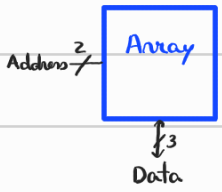
- Número de palavras: 2^N linhas
- Número de bits de cada palavra: M colunas
- Tamanho: $2^N \times M$
- Espaço de endereçamento: $[0, 2^N-1]$ conjunto de endereços possíveis → dimensão = número de palavras
- Endereçabilidade: M número de bits de cada endereço

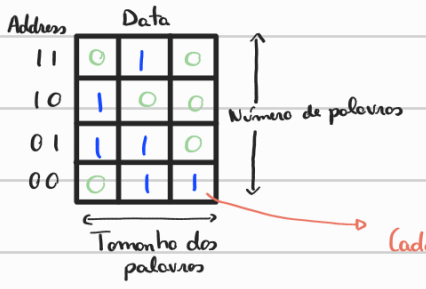
⁉ ≠

ex: array de memória com 2 bits de endereço e 3 bits de dados

Número de palavras = $2^2 = 4$ | Espaço de endereçabilidade = $[0,3]$ de dimensão 4

Tamanho das palavras = 3 | Endereçabilidade = 3 bits





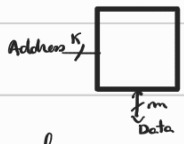
→ Endereço: um número (único) que identifica cada posição de memória. Os endereços são contados sequencialmente começando em 0

→ Espaço de endereçamento: a gama total de endereços que o CPU consegue referenciar

(depende da dimensão de barramento de endereços = 2^N)

ex: CPU com um barramento de endereços de 16 bits (16/)

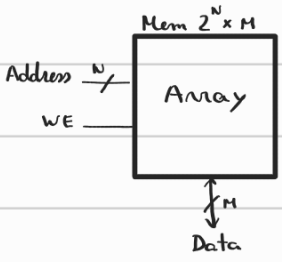
→ Pode gerar endereços na gama: $[0x0000, 0xFFFF] = [0, 2^{16}-1]$



Ex: Seja uma memória de $2^K \times m$ bits

- Endereço é o identificador de localização com K bits
- Espaço de endereçamento é o conjunto de endereços entre 0 e $2^K - 1$
- O conteúdo da memória são as palavras de m bits guardados em cada endereço

→ Operações de leitura vs Operações de escrita:



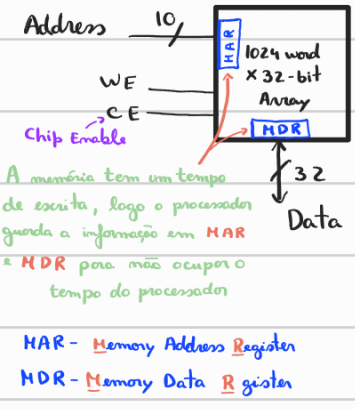
→ WE - Write Enable

→ WE = 1 ⟶ Escrita (o valor dos linhos Data é guardado no Address)

→ WE = 0 ⟶ Leitura (o valor guardado no Address é colocado nos linhos Data)

Se WE for active low funciona ao contrário

→ Ciclo Básico de Acesso à Memória:



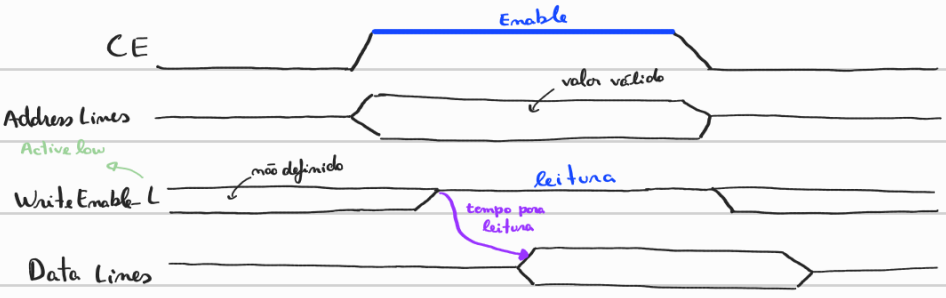
A memória tem um tempo de escrita, logo o processador guarda a informação em MAR e MDR para não ocupar o tempo do processador

MAR - Memory Address Register

MDR - Memory Data Register

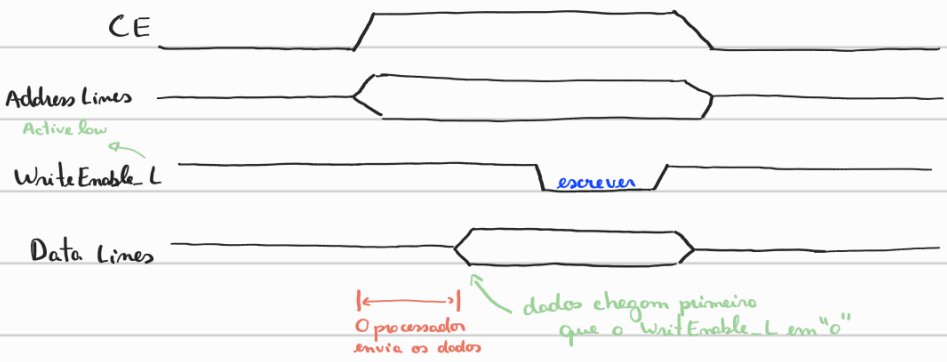
→ Para uma leitura (LOAD) a CPU: (Não confundir!)

1. Ativa o sinal CE (seleciona a memória)
2. Coloca o endereço a ler no barramento de endereços
3. Envia o sinal de Read (não escrita)
4. Lê o conteúdo da memória no barramento de dados



→ Para uma escrita (STORE) a CPU:

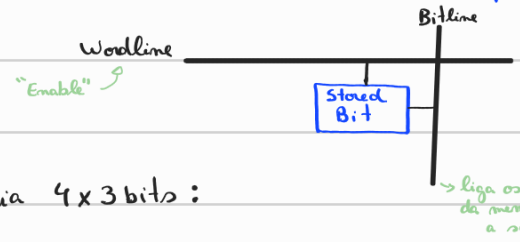
1. Ativa o sinal CE (seleciona a memória)
2. Coloca o endereço a ser escrito no barramento de endereços
3. Coloca o valor a escrever no barramento de dados
4. Ativa o sinal de Write



→ Células de memória:

→ Unidade capaz de armazenar um bit – tem, pelo menos, 2 interfaces:
- → Um para saber quando é endereçada
- → Um para receber/transmitir informação

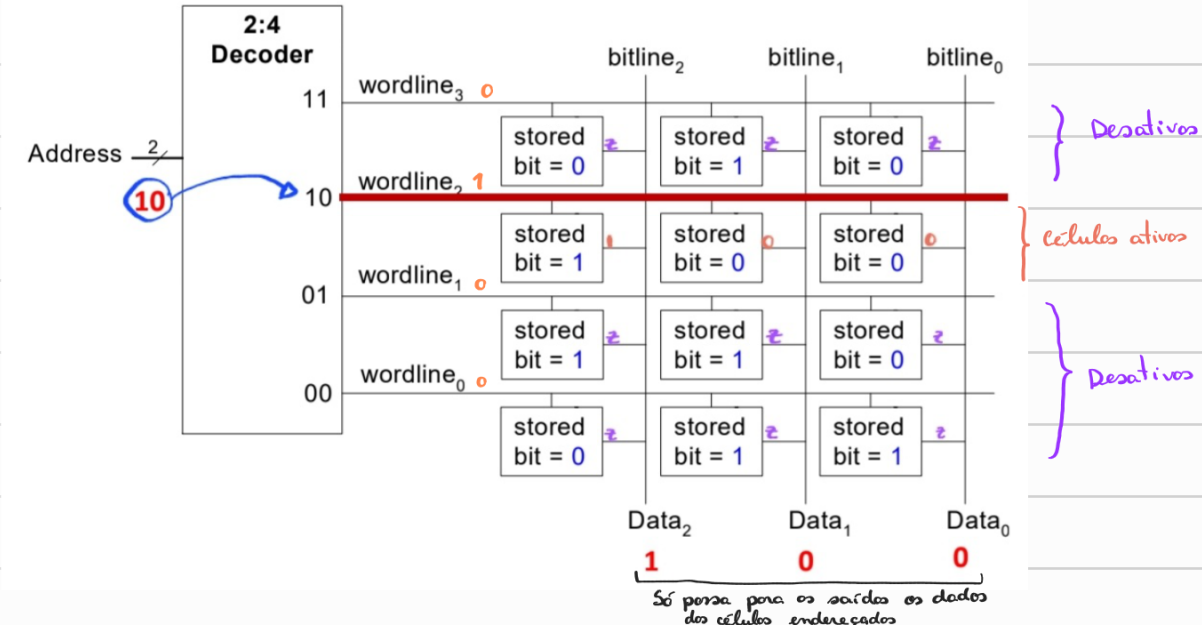
→ Como a memória está organizada em "palavras" o endereçamento é comum a todos os bits de cada palavra: wordline



↓
Funciona como um enable, seleciona a linha a ser acedida

⚠ Só pode estar uma wordline ativa de cada vez

Ex: memória 4 x 3 bits:



· Tipos de Memórias:

→ Os três tipos mais comuns são:

- SRAM → Static Random Acess Memory
- DRAM → Dynamic Random Acess Memory

} Voláteis

- ROM → Read Only Memory

} Não voláteis

→ SRAM: Static RAM (tempo de acesso mais curto)

- bit guardado em dois inversores acoplados
- o valor é mantido enquanto o circuito estiver alimentado, por isso chama-se estática
- cada célula precisa de um circuito complexo que evita as perdas

→ DRAM: Dynamic RAM

- bit é guardado como uma carga num condensador (↳ perdas de carga) — Guarda por pouco tempo (ao fim de algum tempo é impossível distinguir um 0 de um 1)
- leitura da memória é distrutiva
- chama-se dinâmica porque o valor precisa de ser reescrito periodicamente (*)
  e também sempre que é lido (construção mais simples mas precisa de ser reescrito)

(*) em um certo tempo temos de "refrescar" a posição para evitar perdas ⇒ Circuito de refrescamento

Nota: Em uma escala grande, ganhamos em cada célula e podemos construir o circuito de refrescamento e ainda ter mais células pelo mesmo preço e com menos espaço

→ RAM vs ROM:

- RAM: Random Acess Memory
  - ↳ Volátil
  - ↳ Pode ser lida e escrita rapidamente

Qualquer posição de memória é acedido com igual facilidade

- ROM: Read Only Memory (Antigamente eram de leitura apenas)
  - ↳ Não volátil
  - ↳ Acesso de leitura é rápido
  - ↳ Acesso para escrita é impossível ou lento

Hoje em dia, já dá para "escrever" muito rápido com "distruição e reprogramação" ↓ 0

· Máquinas de Estado:

Circuito que:

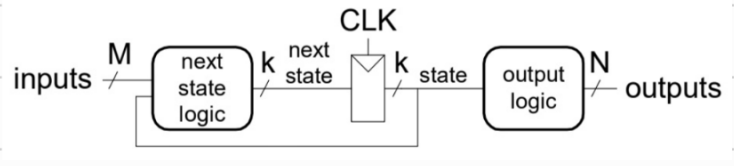
→ Tem vários estados possíveis

→ O estado atual é armazenado num registo (conjunto de Flip-Flops)

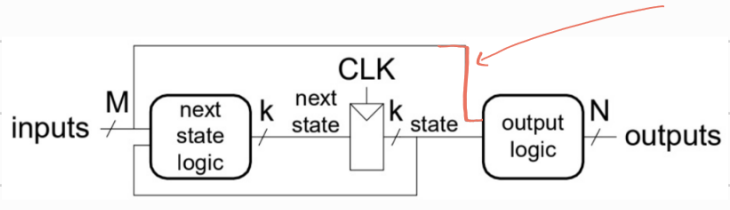
→ O estado muda na transição do relógio 

· Tipos de máquinas de estado:

Esta cadeira → → Síncronos (máquinas de Moore) - saída depende do estado atual



→ Assíncronos (máquinas de Mealy) - saída depende do estado e das entradas



→ Procedimento de projeto de Máquinas de Estado (Moore):

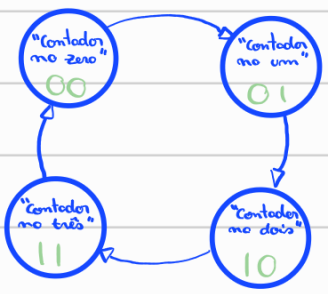
1. Identificar entradas e saídas
2. Desenhar o diagrama de estados e transições
3. Escrever a tabela de estados (em função do estado atual e das entradas)
4. Escrever a tabela das saídas (em função do estado)
5. Escrever (e simplificar) as equações booleanas: lógica do próximo estado e lógica das saídas
6. Desenhar o esquema do circuito

ex: Contador de módulo 4 → 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, ...

- Saída atual é igual ao estado atual
- O próximo estado apenas depende do estado atual

1 Não tem entradas e tem 2 bits de saída (Y_1 e Y_0)

2 4 estado ⇒ 2 bits de estado



3

Estado atual		Estado seguinte	
S_1	S_0	S_1^+	S_0^+
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	0

4

Estado Atual		Saídas	
S_1	S_0	Y_1	Y_0
0	0	0	0
0	1	0	1
1	0	1	0
1	1	1	1

5

$S_1^+ = S_1 \oplus S_0$

$S_0^+ = \overline{S_0}$

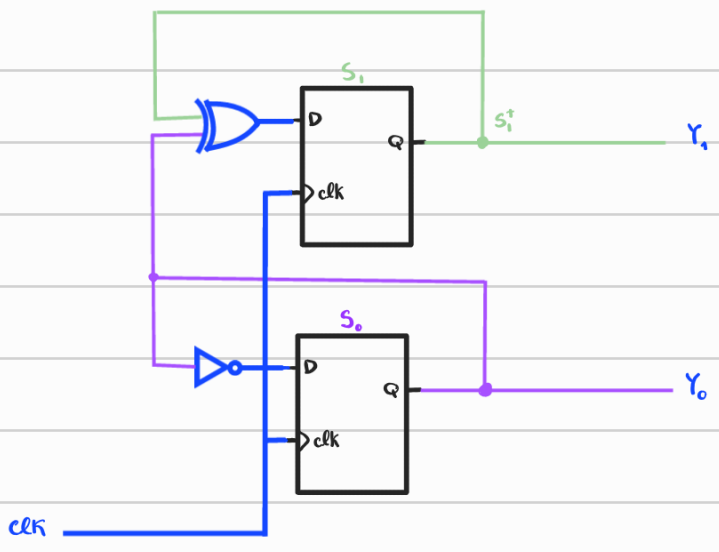
→ Equações de estado seguinte

$Y_1 = S_1$

$Y_0 = S_0$

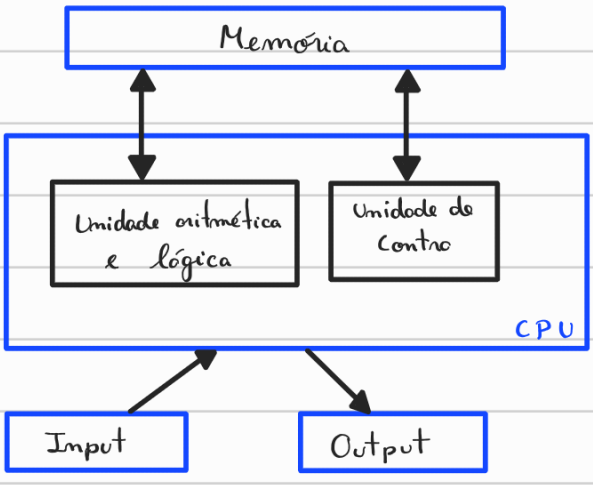
→ Equações das saídas

6



- **ENIAC:**
  - → Primeiro computador eletrónico
  - → Utilizado para cálculo balístico
  - → Demorava dias a reprogramar
  - → John Von Neumann propôs a armazenagem dos dados e do programa na mesma memória (Assim, os programas poderiam ser guardados e reaproveitados)

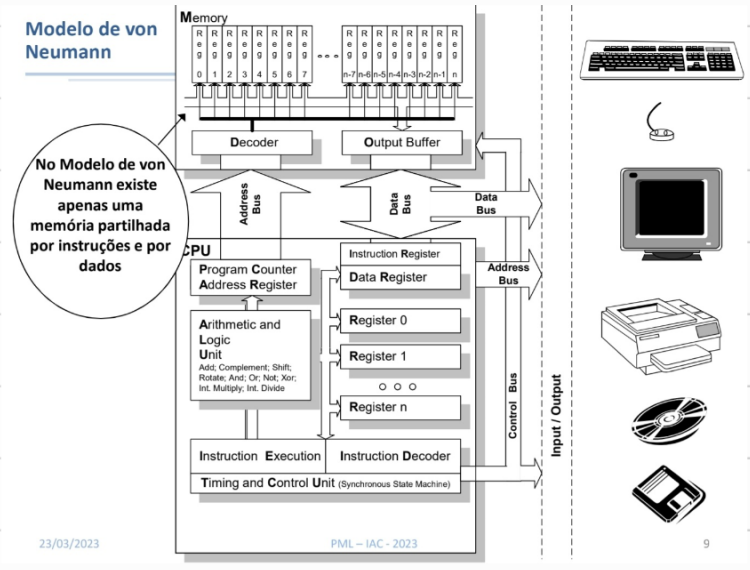
- **Modelo de Von Neumann:**

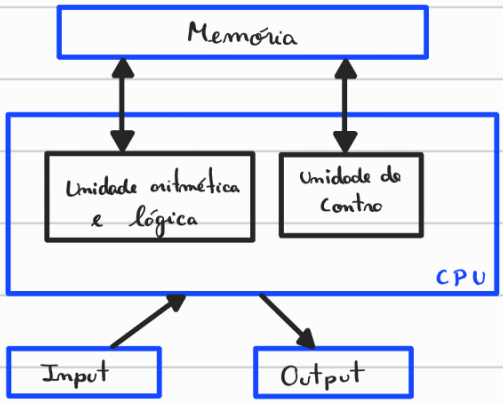


"Existe apenas uma memória partilhada por instruções e dados"

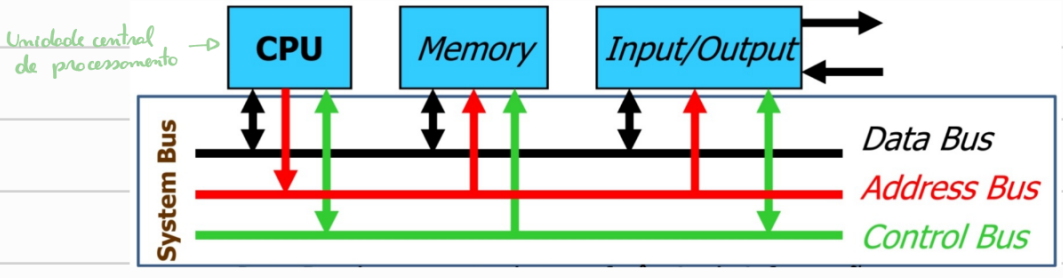
→ Unidades fundamentais que constituem um computador:

- **Unidades de entrada:** permitem a receção de informação vinda do exterior (dados, programas) e que é armazenada em memória
- **Unidades de saída:** permitem o envio de resultados para o exterior
- **Memória:** armazenamento de
  - Programas
  - Dados de processamento (dados de entrada/saída)
  - Resultados
- **CPU:** processamento da informação através da execução do programa armazenado em memória





→ Arquitetura básica, no modelo de Von Neumann, de um sistema computacional (Simplificado)



Data Bus: barramento de transferência de informação (CPU ↔ Memory, CPU ↔ Input/Output)

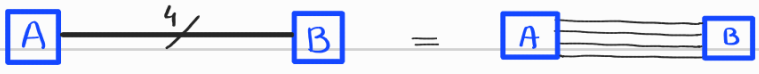
Address Bus: identifica a origem/destino da informação

Control Bus: sinais de protocolo que especificam o modo como a transferência de informação deve ser feita

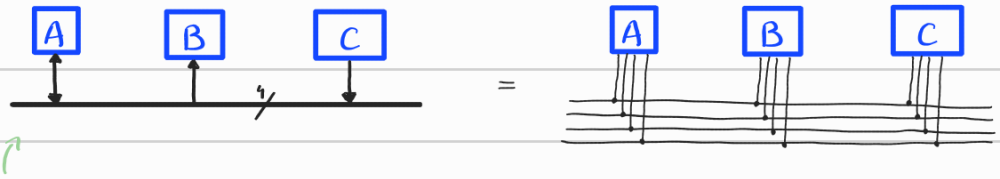
Garante sincronismo

→ Barramento e Barramento partilhado:

Barramento:



Barramento partilhado:



Fundamental o algoritmo para gestão do barramento

· Unidade de Processamento (CPU):

→ É organizada em vários unidades:

- Unidade de controlo (responsável pela sequencialização da execução de cada instrução)
- Unidade Aritmética e Lógica (→ ALU) (executar operações)
- Registos (armazenamento temporário de dados e valores de controlo)

memórias mais rápidas

→ Cumprimento da Palavra (número de bits da ALU e dos registos)

"processador funciona com n bits"

→ Consiste em duas secções:

- Secção de dados (datapath)
  - Registos internos
  - Unidade Aritmética e Lógica
- Unidade de controlo (responsável pela coordenação dos elementos do datapath, durante a execução de um programa)
  - Máquinas de estado síncronos
    ↳ controla os estados do sistema dependendo das instruções e do estado atual

→ Unidade de Controlo:

- controla a execução do programa (determina: a operação a ser realizada no presente / a próxima operação a ser realizada)
- lê uma instrução na memória (instruction Fetch) e interpreta a instrução (instruction Decode) gerando os sinais de controlo que indicam à unidade de processamento o que fazer

Mesa de trabalho →

Instruction Register

→ contem a instrução a ser executada

Program Counter

→ contem o endereço da próxima execução

· Noção de "instrução":

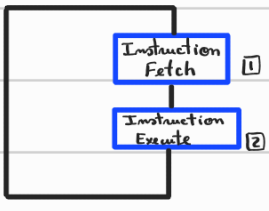
→ Independente do CPU e da sua estrutura interna, uma instrução deve permitir responder às seguintes questões:

- Qual a operação a realizar?
- Qual a localização dos operandos (se existem)? (reg. interno / memória)
- Onde colocar o resultado? (reg. interno / memória)
- Qual a próxima instrução? (instrução seguinte / endereço da próxima instrução)

→ Níveis de representação de instruções: contém indicações explícitos e implícitos

→ Ciclo básico de execução de instrução:

1. Processador acede à memória e lê a próxima instrução a ser executada (· acede à memória · traz a instrução)
2. Processador executa a instrução



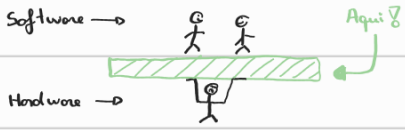
· Arquitetura de computadores: = Arquitetura do conjunto de instruções (ISA) ⊕ Organização da máquina

→ Conjunto de instruções: coleção de todos as instruções que o processador pode executar
↓ estrutura de processador para executar o conjunto de instruções

→ Microarquitetura: organização do processador (principais unidades de funcionamento e respetivos ligações e controlo)

1 arquitetura ⋐ Pode conter vários microarquiteturas

→ **Arquitetura do conjunto de instruções**: descreve as funcionalidades (organização do fluxo de dados e da unidade de controlo) independentemente da implementação ("modelo de programação")



- **Aspetos a definir numa arquitetura:**
  - → Instruções suportadas
  - → Como organizar a memória e os acessos
  - → Quantos registos (muitos ou poucos?)
  - → Tipos de dados e estruturas suportados
  - → Modos de endereçamento e de acesso a dados e instruções
  - → Qual o formato das instruções (tamanho variável / fixo)
  - → Condições de Exceção (condições inesperadas)
- **Fatores a ter em conta:**
  - → Aplicações a que se destina
  - → Linguagem de programação
  - → Sistema operativo
  - → Possibilidades tecnológicos
  - → Compatibilidade histórica
- **Objetivos de uma arquitetura:**
  - → Implementação eficiente e simples em hardware
  - → Fácil de entender e programar
  - → compiladores eficientes

→ **Organização da máquina**: (responde a algumas questões...)

- Características operativas e de performance dos principais unidades de funcionamento
- De que modo os componentes são interligados
- Fluxo de informação entre componentes
- Lógica e meios através dos quais esse fluxo é controlado
- Coreografia dos Unidades Funcionais para implementar a arquitetura do conjunto de instruções