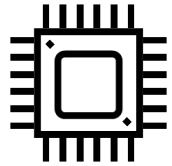
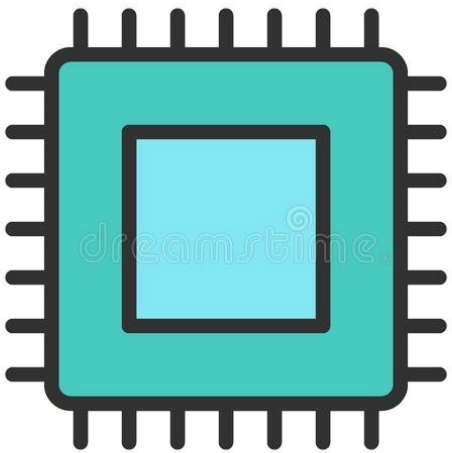
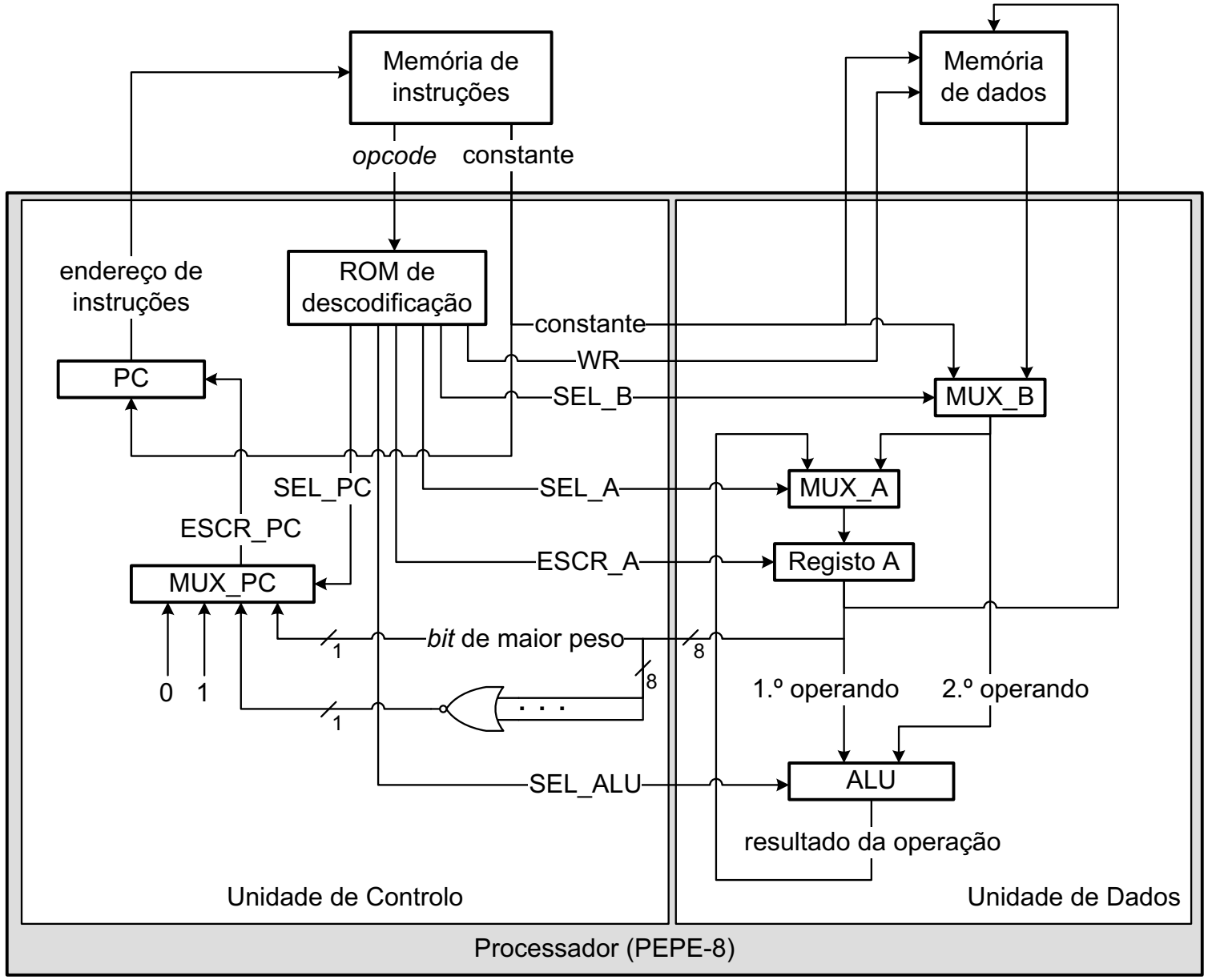
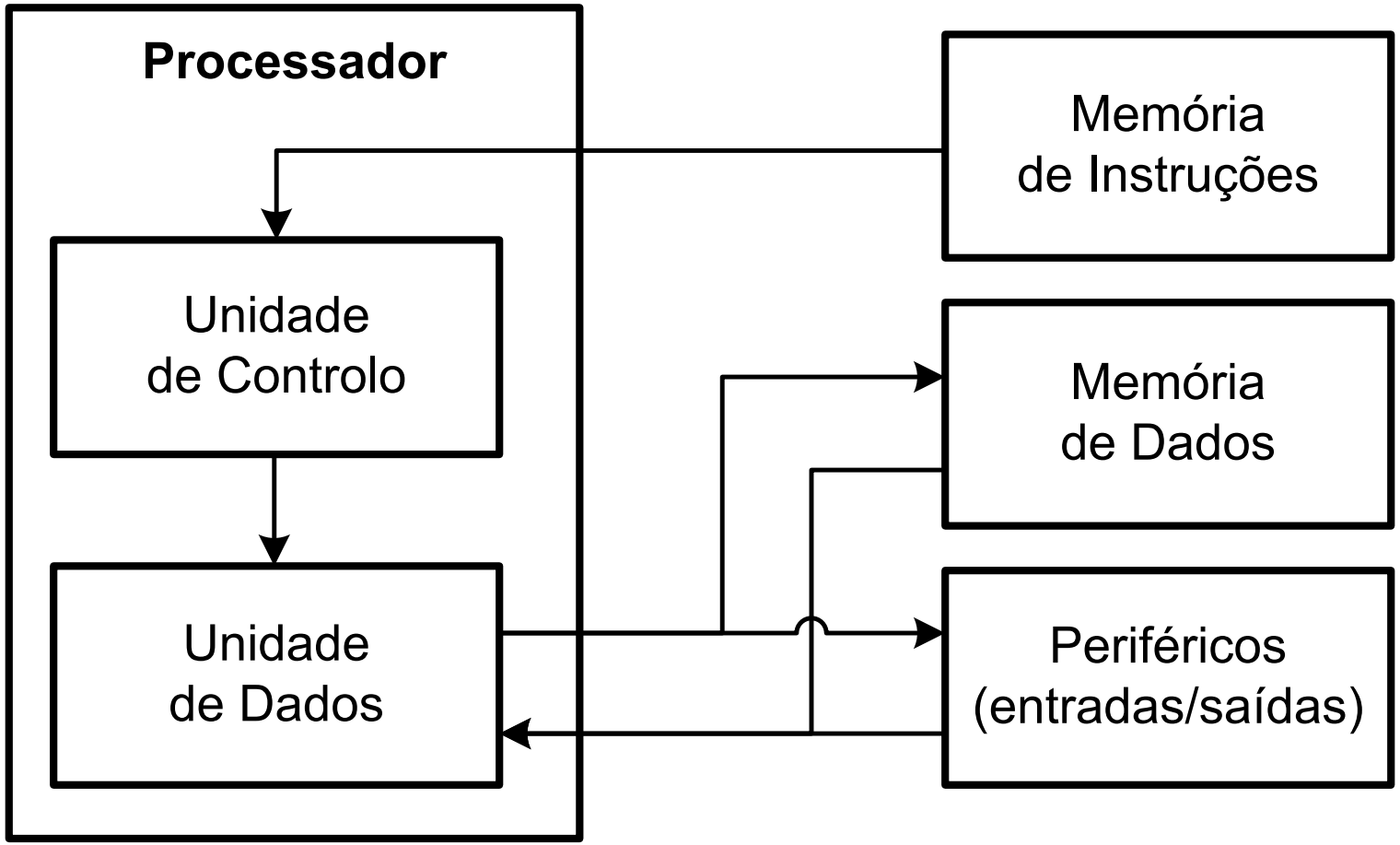


O meu primeiro computador



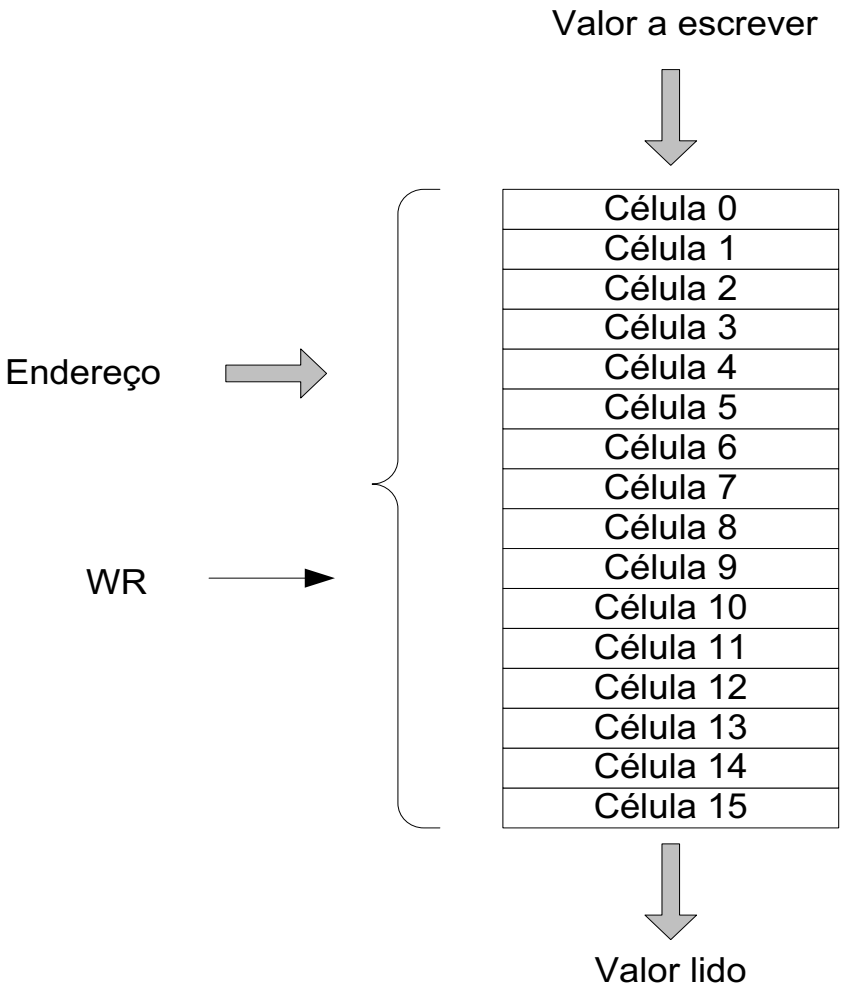

Memória de instruções
Memória de dados
opcode
constante
endereço de instruções
ROM de descodificação
constante
WR
PC
SEL_B
MUX_B
SEL_PC
SEL_A
MUX_A
ESCR_PC
ESCR_A
Registo A
MUX_PC
bit de maior peso
1
8
0
1
8
1.º operando
2.º operando
1
SEL_ALU
ALU
resultado da operação
Unidade de Controlo
Unidade de Dados
Processador (PEPE-8)

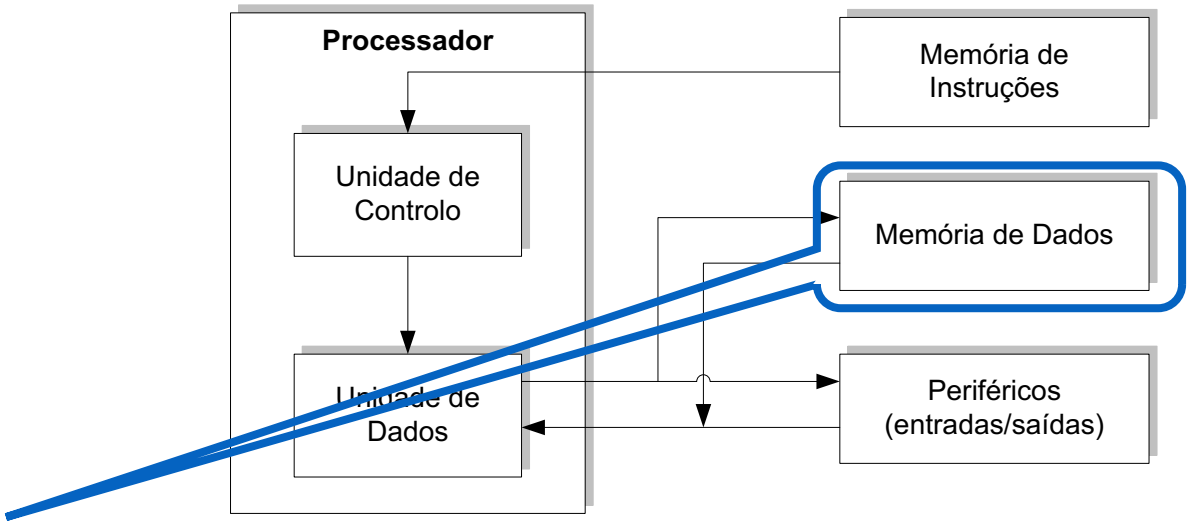
Estrutura de um computador



Memória de dados (RAM)

Memória: elemento fundamental





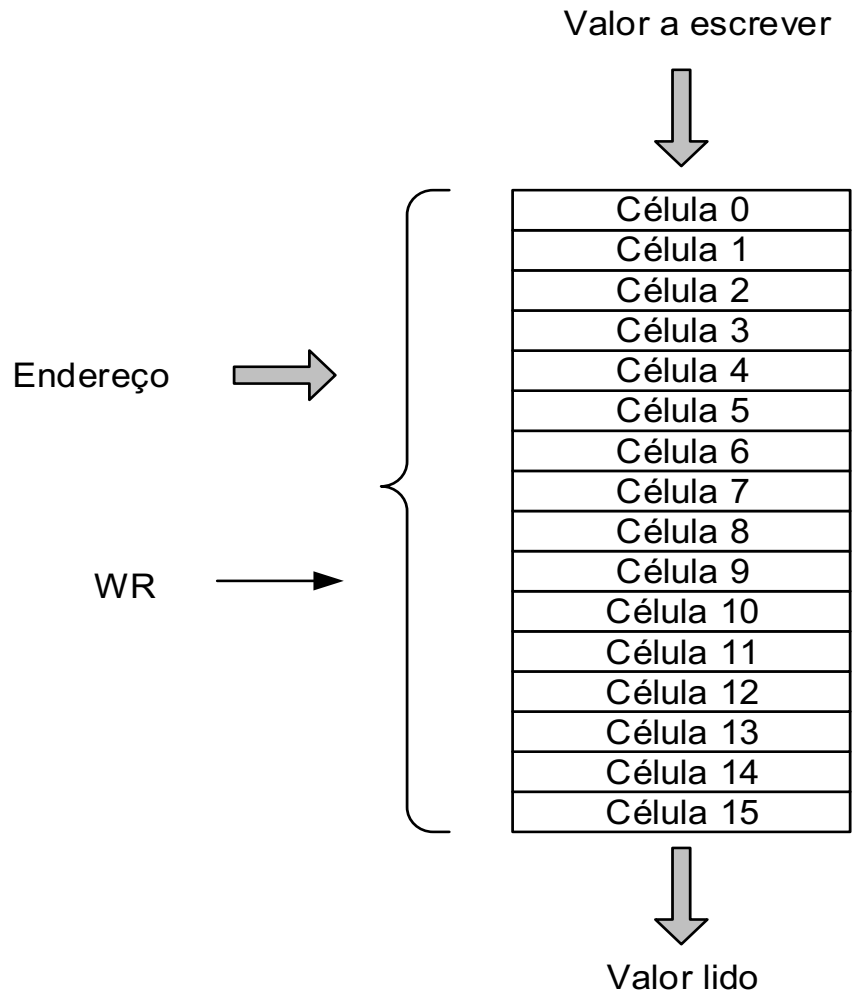
RAM - volátil

ROM - persistente

Os periféricos são uma espécie de células de memória cujos bits ligam ao **mundo exterior**.

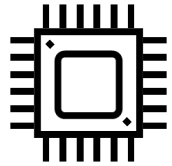
Memória

- **Largura:** Número de bits de cada célula (ou registo).
- **Tamanho:** Número de células (N)
- **Capacidade (bytes):** largura **x** tamanho
- **Endereço:** Número da célula acedida (0 até N-1)
- **WR (write):** indica se acesso é de escrita ou leitura



Processador: unidade de dados

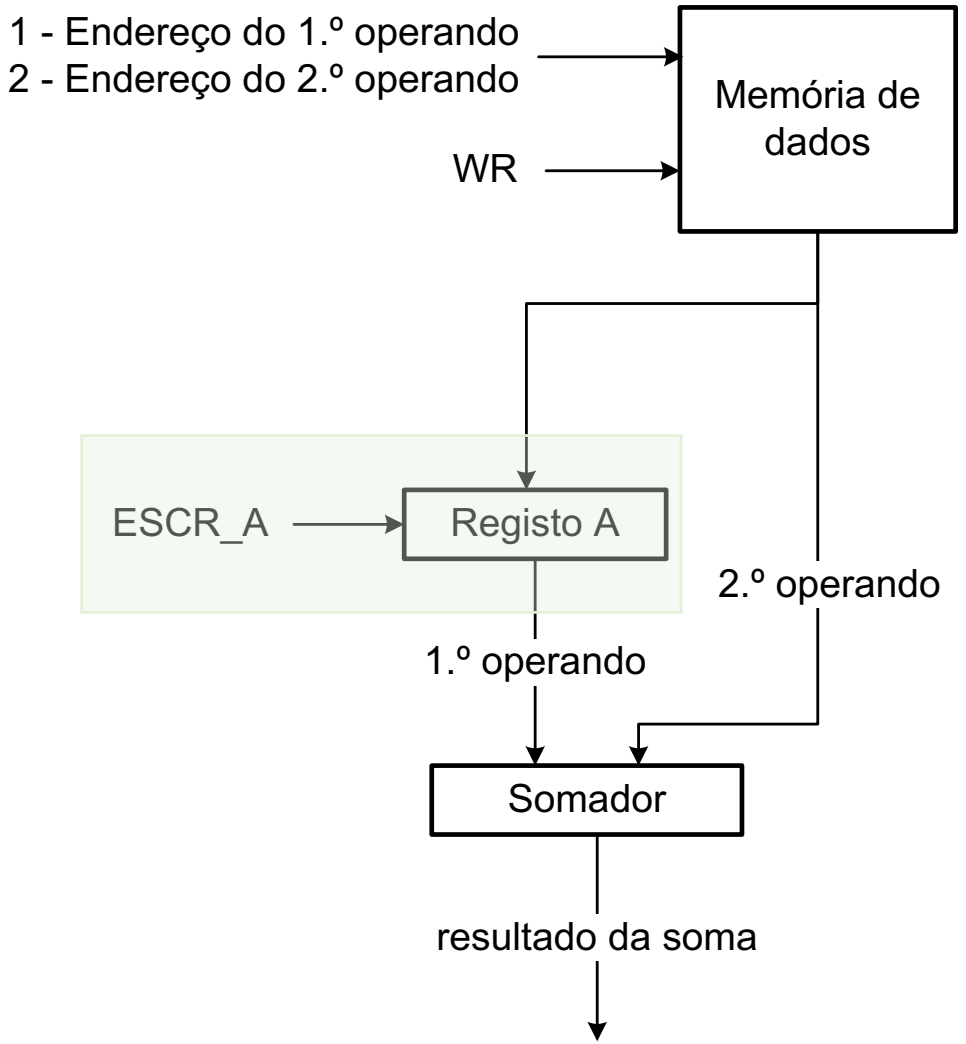
1º objetivo: processar algoritmo da soma



1. O processador **lê** os operandos da memória de dados
2. A unidade de dados do processador inclui um somador que **faz a soma** e produz um resultado
3. O resultado é **armazenado** na memória de dados

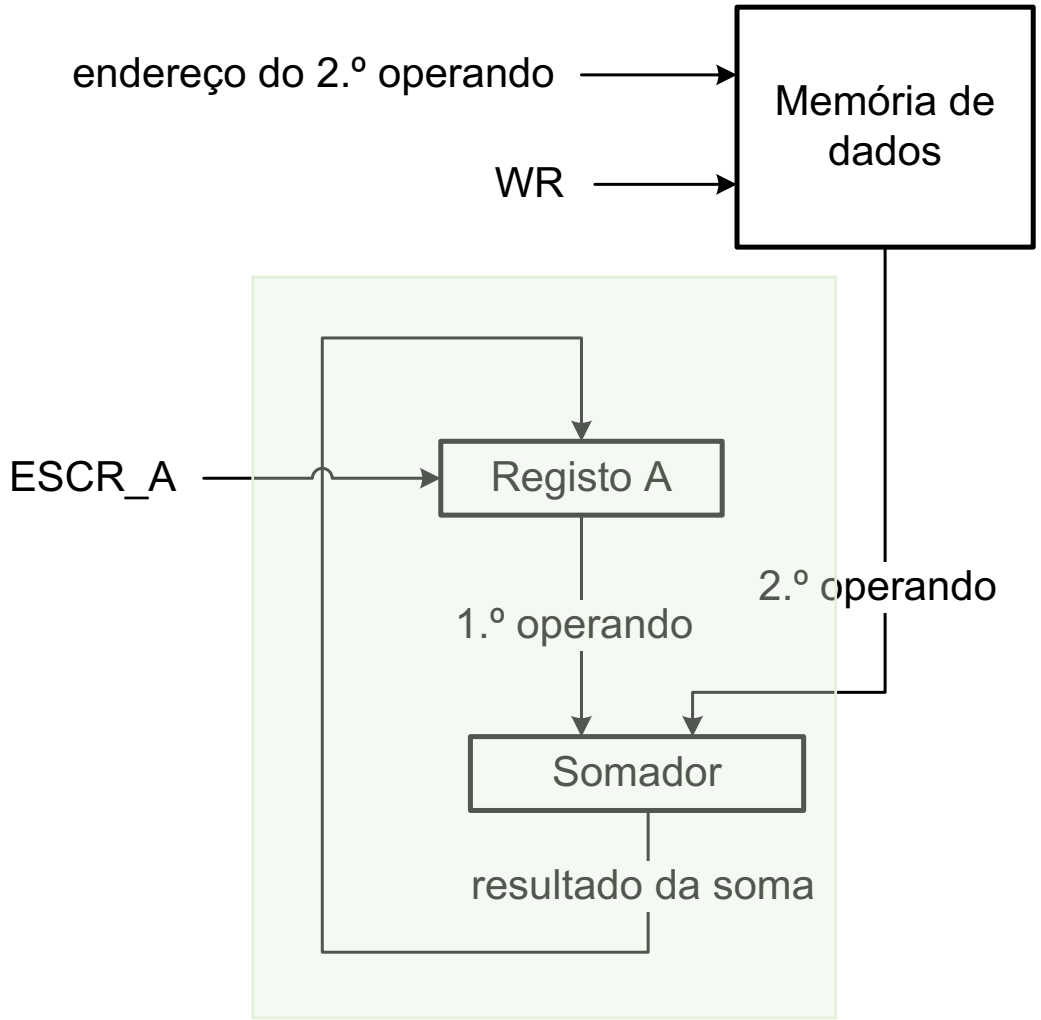
Unidade de dados (versão 1)

- Problema 1: Um somador tem duas entradas, mas a memória só tem uma saída.
- Solução: um **registo** para memorizar um dos operandos
- Controlo: sinal **WR** inativo; sinal **ESCR_A** ativo para escrita no registo, desativo durante operação soma (está a ser lido)



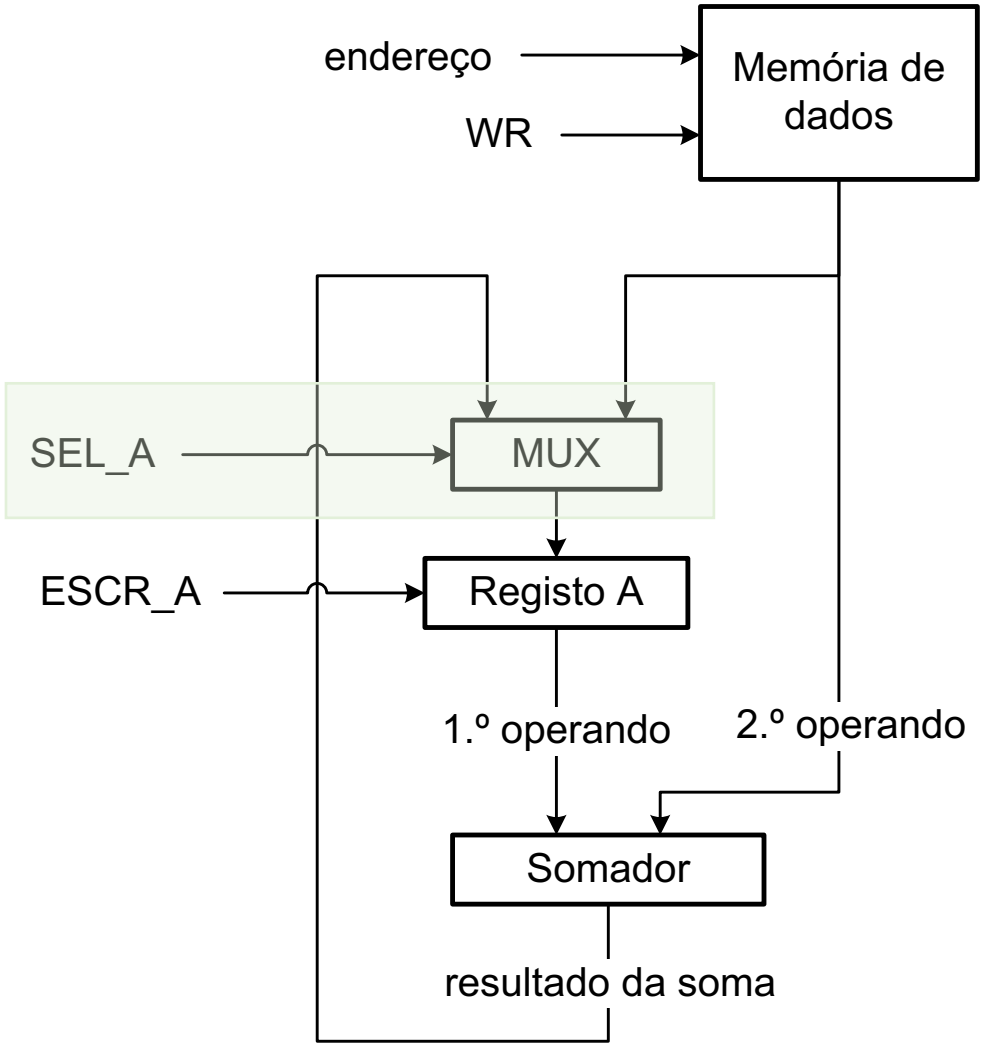
Unidade de dados (versão 2)

- Problema 2: Resultado não pode ir para a memória (está ocupada a ler o 2º operando)
- Solução: **Usar o registo** para esse efeito.
  - O registo chama-se tradicionalmente "acumulador"
- Controlo: quando o sinal do **relógio** muda **ESCR_A** fica ativo



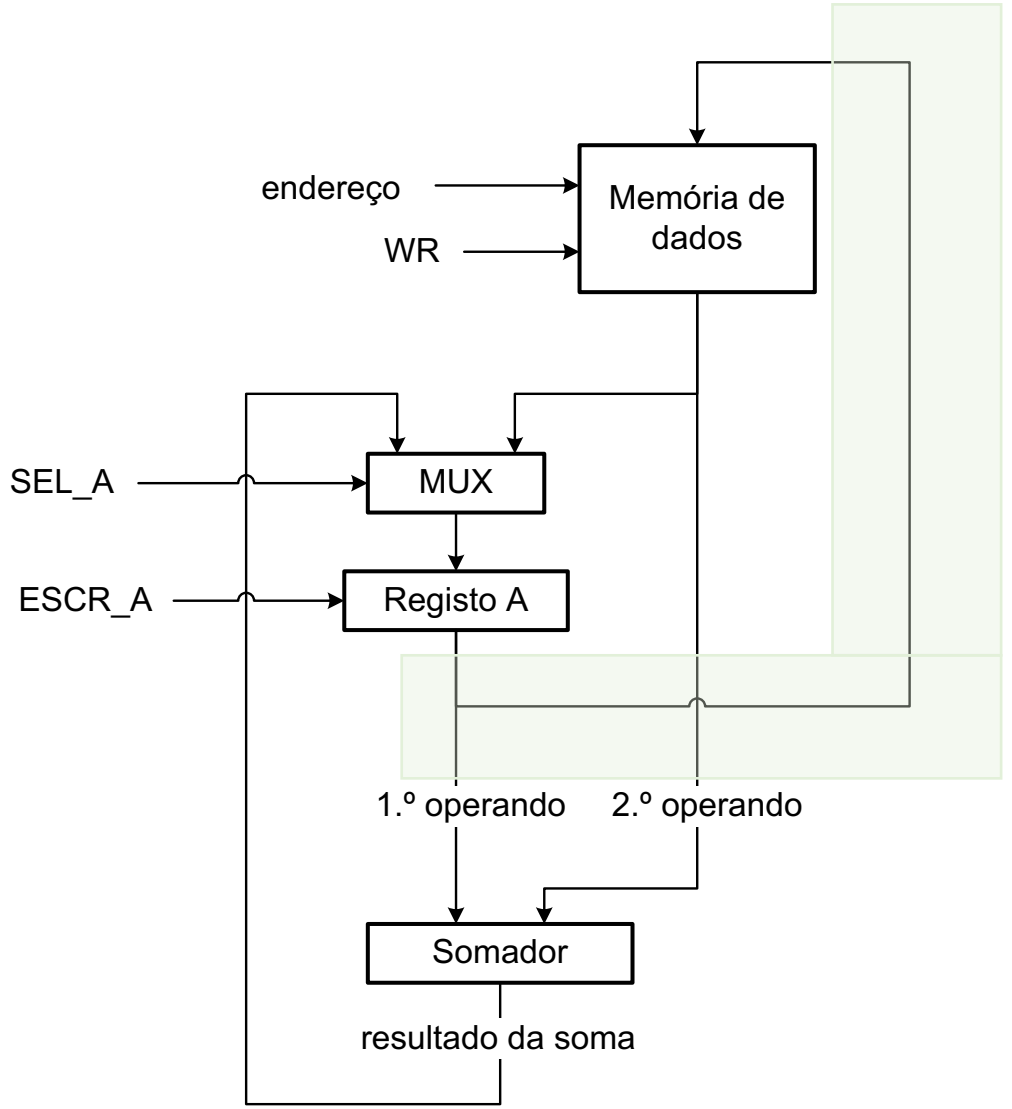
Unidade de dados (versão 3)

- Problema 3: Entrada do registo não pode vir de dois lados (registo e somador)
- Solução: Usar um **multiplexer** para selecionar a entrada.
- Controlo: sinal **SEL_A** define qual das entradas usar a cada momento



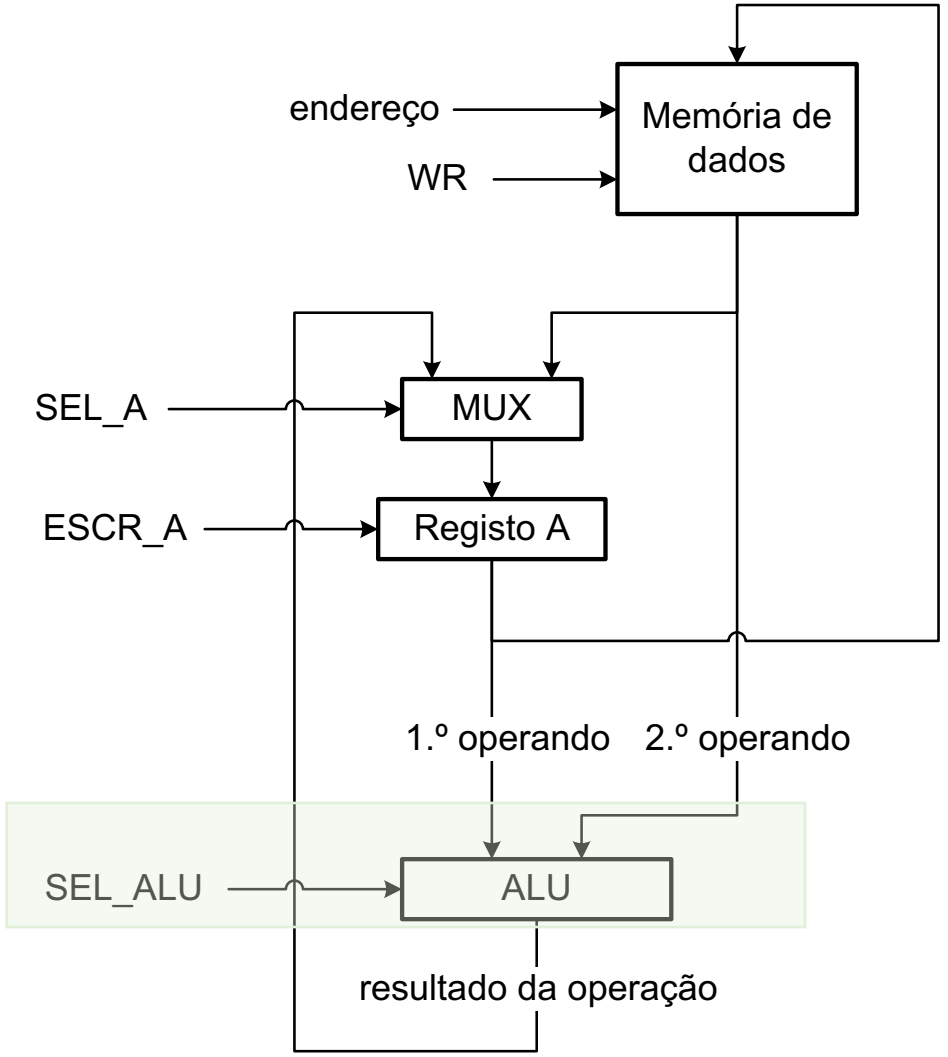
Unidade de dados (versão 4)

- Problema 4: Como guardar resultados na memória?
- Solução: **Ligar** saída do registo à entrada da memória (o resultado vai sempre para o registo, depois copia-se)
- Controlo: Ativa-se sinal **WR**



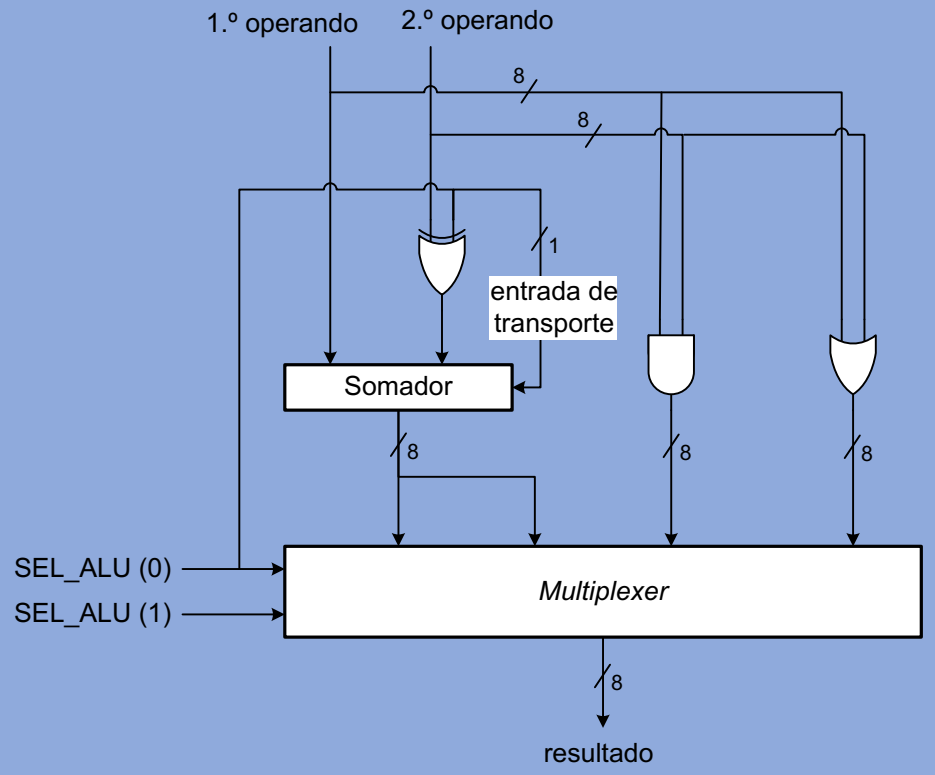
Unidade de dados (versão 5)

- Problema 5: Como suportar várias operações?
- Solução: Usar uma ALU (Arithmetic and Logic Unit).
- Controlo: O sinal **SEL_ALU** seleciona a operação.



refletir.com

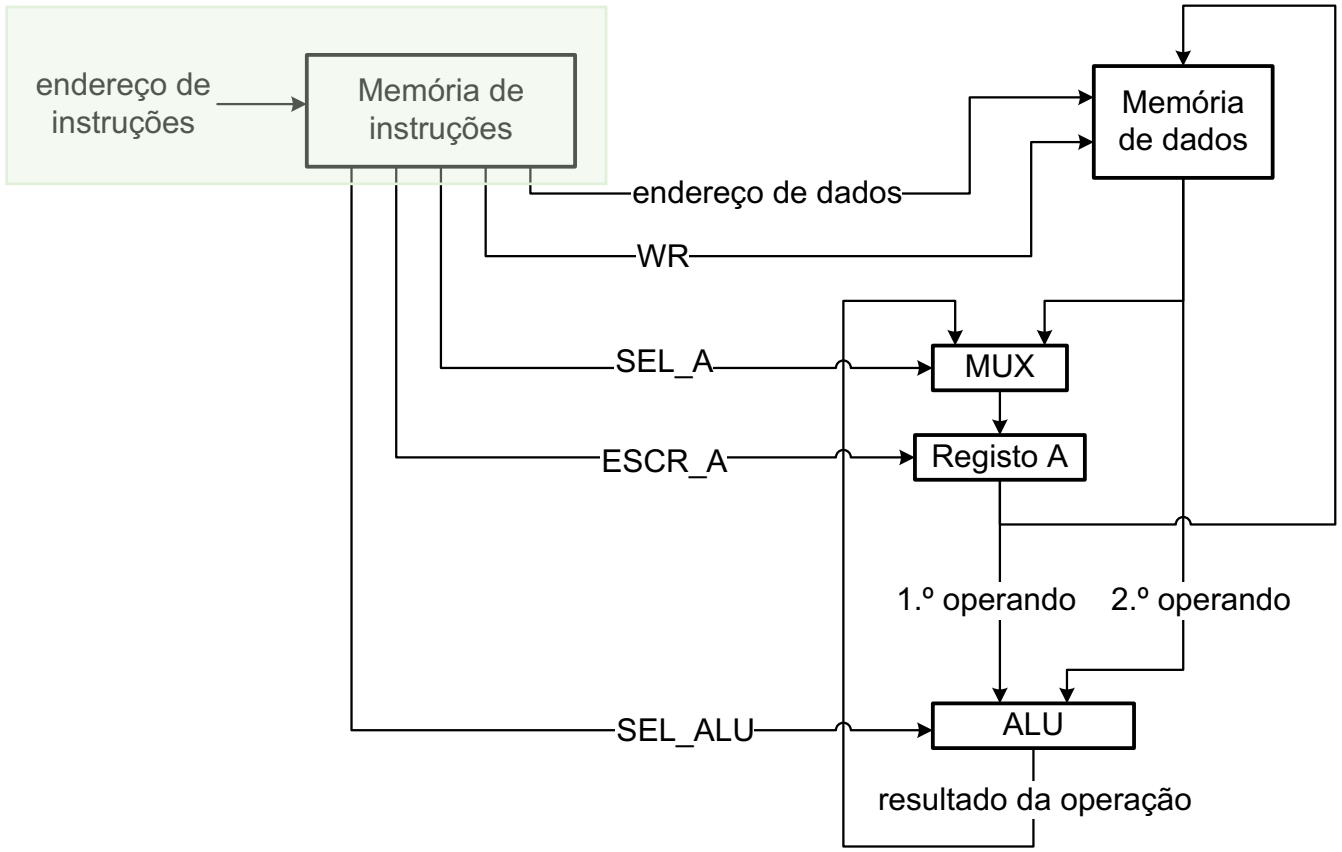
- Considerem que a ALU do nosso primeiro computador executa este conjunto de operações:
  - Soma
  - Subtração
  - AND
  - OR
- Qual o número de bits do sinal **SEL_ALU**?



Processador: unidade de controlo

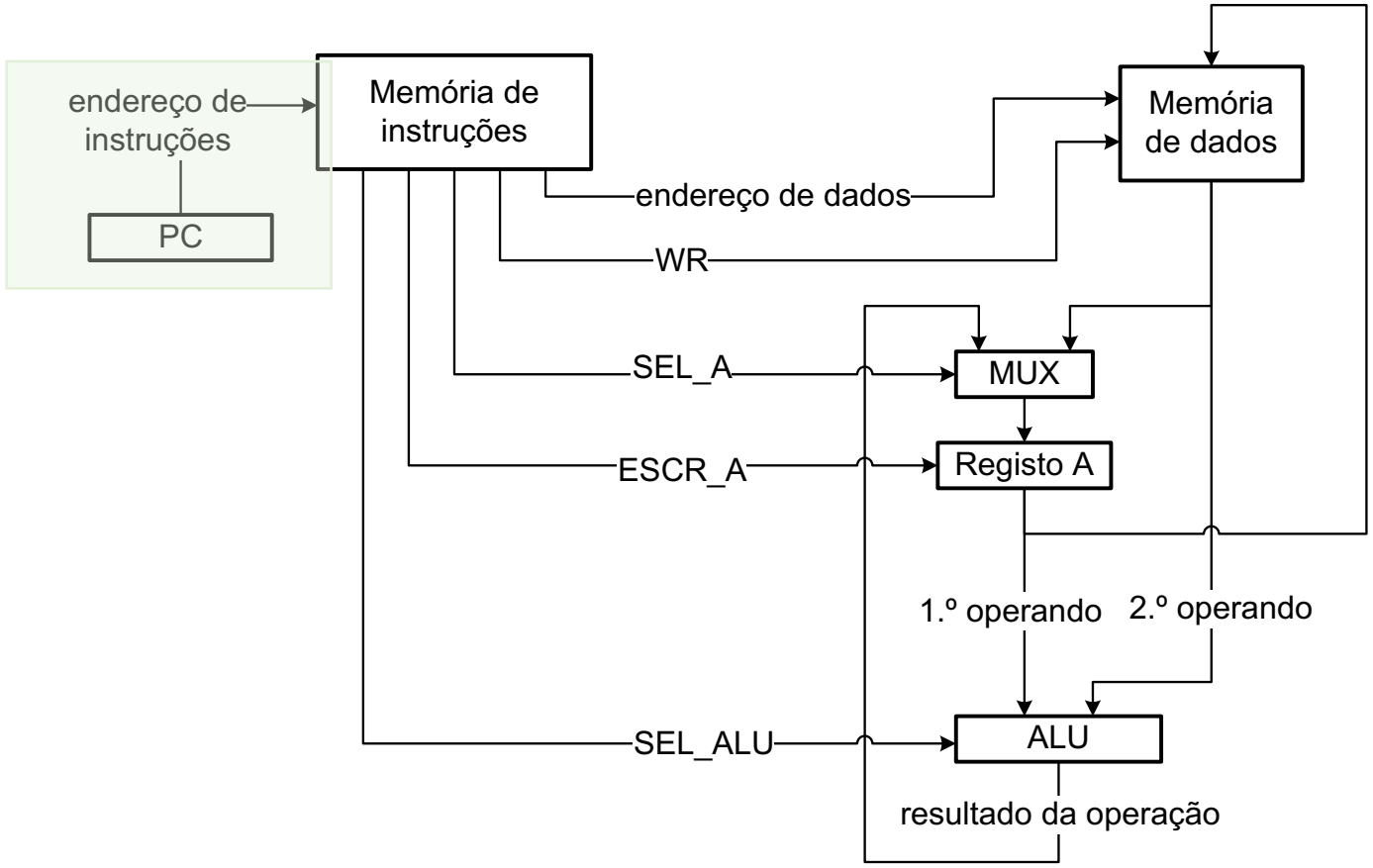
Instruções com sinais de controlo

- Problema 6: Como especificar o valor dos sinais que controlam o circuito?
- Solução: Cada **instrução** (conteúdo de uma célula na memória de instruções) deve **especificar todos os sinais** necessários para se executar



Contador de programa (PC)

- Problema 7: Como indicar quais as instruções (e o seu sequenciamento) que se pretendem executar num dado programa?
- Solução: utiliza-se **um registo (program counter)** que indique quais das instruções da memória de instruções está em execução e um **mecanismo** que indique **qual a instrução seguinte**



Já podemos fazer um programa!

- Objetivo do programa: somar um número com todos os inteiros positivos menores que ele.

$$soma = N + (N-1) + (N-2) + \ldots + 2 + 1$$

1.	soma ← 0	(inicializa **soma** com zero)
2.	iteracao ← N	(inicializa **iteracao** com N)
3.	Se (iteracao < 0) salta para 8	(se **iteracao** for negativo, salta para o fim)
4.	Se (iteracao = 0) salta para 8	(se **iteracao** for zero, salta para o fim)
5.	soma ← soma + iteracao	(adiciona **iteracao** a **soma**)
6.	iteracao ← iteracao – 1	(decrementa **iteracao**)
7.	Salta para 4	(salta para o passo 4)
8.	Salta para 8	(fim do programa)

Variáveis em memória

- **soma** e **iteracao** serão células de memória cujo conteúdo vai variando ao longo do tempo
  - o registo A é só para ir efetuando os cálculos intermédios
- Cada célula de memória tem um endereço (neste exemplo, **soma** fica em 40_H e **iteracao** em 41_H).

1.	**M[40_H]** ← 0	(inicializa **soma** com zero)
2.	**M[41_H]** ← N	(inicializa **iteracao** com N)
3.	Se (**M[41_H]** < 0) salta para 8	(se **iteracao** for negativo, salta para o fim)
4.	Se (**M[41_H]** = 0) salta para 8	(se **iteracao** for zero, salta para o fim)
5.	**M[40_H]** ← **M[40_H]** + **M[41_H]**	(adiciona **iteracao** a **soma**)
6.	**M[41_H]** ← **M[41_H]** – 1	(decrementa **iteracao**)
7.	Salta para 4	(salta para o passo 4)
8.	Salta para 8	(fim do programa)

Constantes simbólicas

- Usar números como endereços é confuso
- Solução: usar **constantes simbólicas** = identificador com um valor
  - Definem-se uma vez e depois podem usar-se como se fosse o número com que foram definidas

```
soma     EQU    40H      (definição do endereço da variável soma)
iteracao EQU    41H      (definição do endereço da variável iteracao)

1.  M[soma] ← 0                          (inicializa soma com zero)
2.  M[iteracao] ← N                      (inicializa iteracao com N)
3.  Se (M[iteracao] < 0) salta para 8    (se iteracao for negativo, salta para o fim)
4.  Se (M[iteracao] = 0) salta para 8    (se iteracao for zero, salta para o fim)
5.  M[soma] ← M[soma] + M[iteracao]      (adiciona iteracao a soma)
6.  M[iteracao] ← M[iteracao] – 1        (decrementa iteracao)
7.  Salta para 4                         (salta para o passo 4)
8.  Salta para 8                         (fim do programa)
```

Endereços de dados e de instruções

- As **variáveis** ficam na **memória de dados**
- As **instruções** ficam na **memória de instruções**
- Cada **passo do algoritmo** é uma **instrução**
- O **número do passo** é o **endereço na memória de instruções**

```
soma     EQU    40H     (definição do endereço da variável soma)
iteracao EQU    41H     (definição do endereço da variável iteracao)

1.  M[soma] ← 0                          (inicializa soma com zero)
2.  M[iteracao] ← N                      (inicializa iteracao com N)
3.  Se (M[iteracao] < 0) salta para 8    (se iteracao for negativo, salta para o fim)
4.  Se (M[iteracao] = 0) salta para 8    (se iteracao for zero, salta para o fim)
5.  M[soma] ← M[soma] + M[iteracao]      (adiciona iteracao a soma)
6.  M[iteracao] ← M[iteracao] – 1        (decrementa iteracao)
7.  Salta para 4                         (salta para o passo 4)
8.  Salta para 8                         (fim do programa)
```

PC (Contador de Programa)

- O **PC vai evoluindo** instrução a instrução
  - NB: os endereços das memórias começam em 0 e não em 1
- Após cada instrução, o PC contém o **endereço da instrução seguinte**
- "**Saltar**" equivale a escrever um **novo valor no PC**.

```
soma     EQU   40H     (definição do endereço da variável soma)
iteracao EQU   41H     (definição do endereço da variável iteracao)

0  M[soma] ← 0                          (inicializa soma com zero)
1  M[iteracao] ← N                      (inicializa iteracao com N)
2  Se (M[iteracao] < 0) PC ← 7          (se iteracao for negativo, salta para o fim)
3  Se (M[iteracao] = 0) PC ← 7          (se iteracao for zero, salta para o fim)
4  M[soma] ← M[soma] + M[iteracao]      (adiciona iteracao a soma)
5  M[iteracao] ← M[iteracao] – 1        (decrementa iteracao)
6  PC ← 3                               (salta para o passo 3)
7  PC ← 7                               (fim do programa)
```

Vamos correr o algoritmo!

Número de instruções executadas **20**

Valores após a execução da instrução (PC endereça a seguinte):

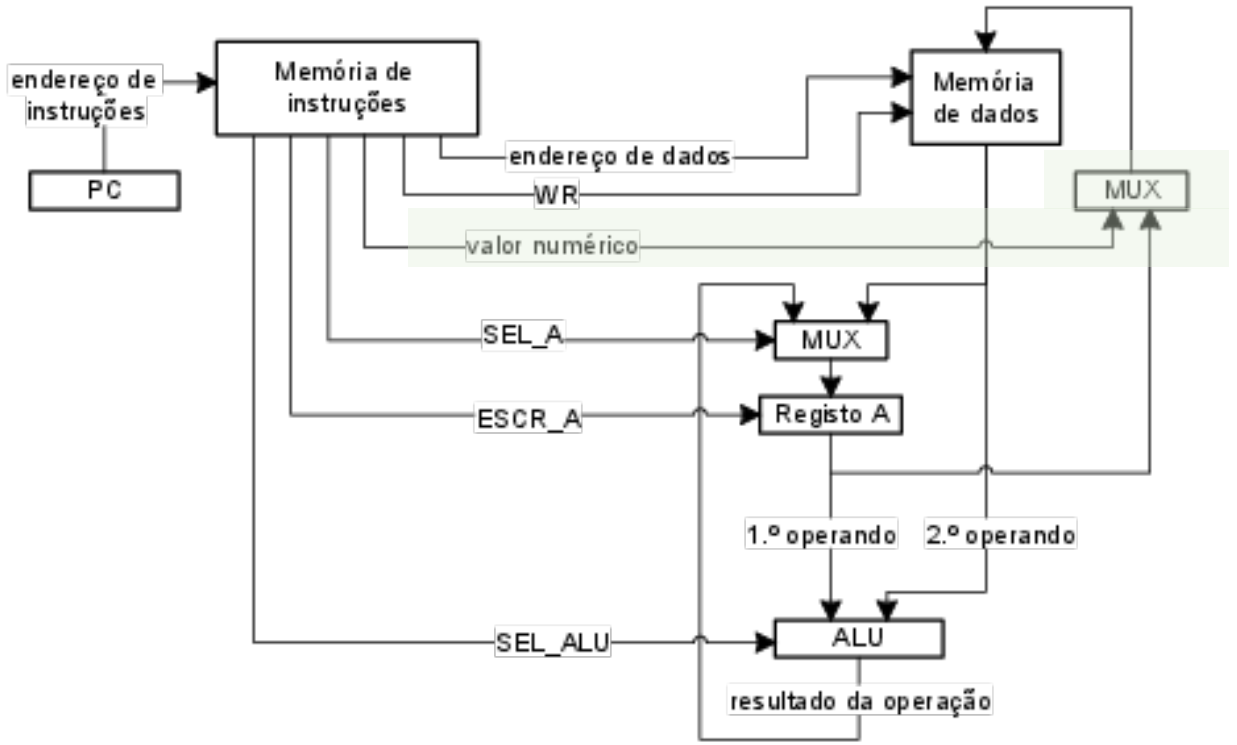
PC 7 **soma** 6 **temp** 0

```
soma    EQU    40H    (definição do endereço da variável soma)
temp    EQU    41H    (definição do endereço da variável temp)
```

```
0    M[soma] ← 0                          (inicializa soma com zero)
1    M[temp] ← N                          (inicializa temp com N)
2    Se (M[temp] < 0) PC ← 7              (se temp for negativo, salta para o fim)
3    Se (M[temp] = 0) PC ← 7              (se temp for zero, salta para o fim)
4    M[soma] ← M[soma] + M[temp]          (adiciona temp a soma)
5    M[temp] ← M[temp] – 1                (decrementa temp)
6    PC ← 3                               (salta para o endereço 3)
7    PC ← 7                               (fim do programa)
```

Constantes simbólicas

- Problema 8: como especificar as constantes no programa
- Solução: Nas **próprias instruções** que as referenciam (é apenas mais um campo de cada célula da memória das instruções



Constantes simbólicas

- Problema 9: só é possível ter uma constante por instrução, e por vezes temos várias
- Solução: transformar instruções com várias constantes em sequências de instruções mais simples (só com uma constante)

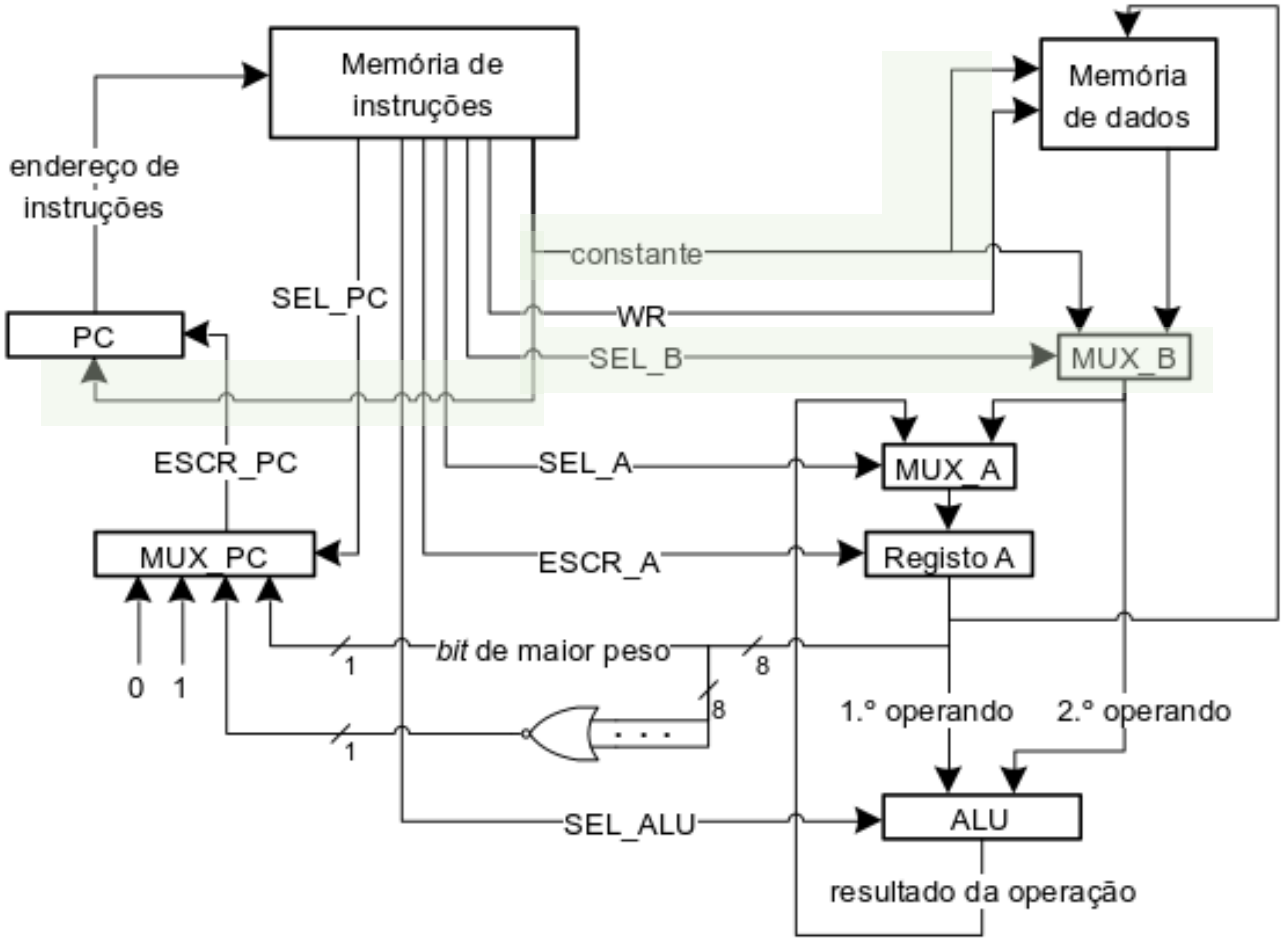
1. M[soma] ← 0	1. A ← 0 2. M[soma] ← A
6. M[iteracao] ← M[iteracao] – 1	6. A ← M[iteracao] 7. A ← A - 1 8. M[iteracao] ← A

Suporte de hardware para constantes

- Problema 10: as constantes têm várias utilizações diferentes, que necessitam de suporte diferenciado do hardware
  - Serem guardadas em registos (e.g., **A ← 0**)
  - Serem usadas como endereço para acesso à memória (e.g., **M[soma]**)
  - Serem guardadas no registo PC para saltos – condicionais ou não – no programa (e.g., **PC ← 7**)
  - Serem usadas como operandos de uma operação aritmética (**A ← A – 1**)
- Solução: Especificar caminhos que permitam às constantes **fluir das instruções para os recursos** em que vão ser usadas

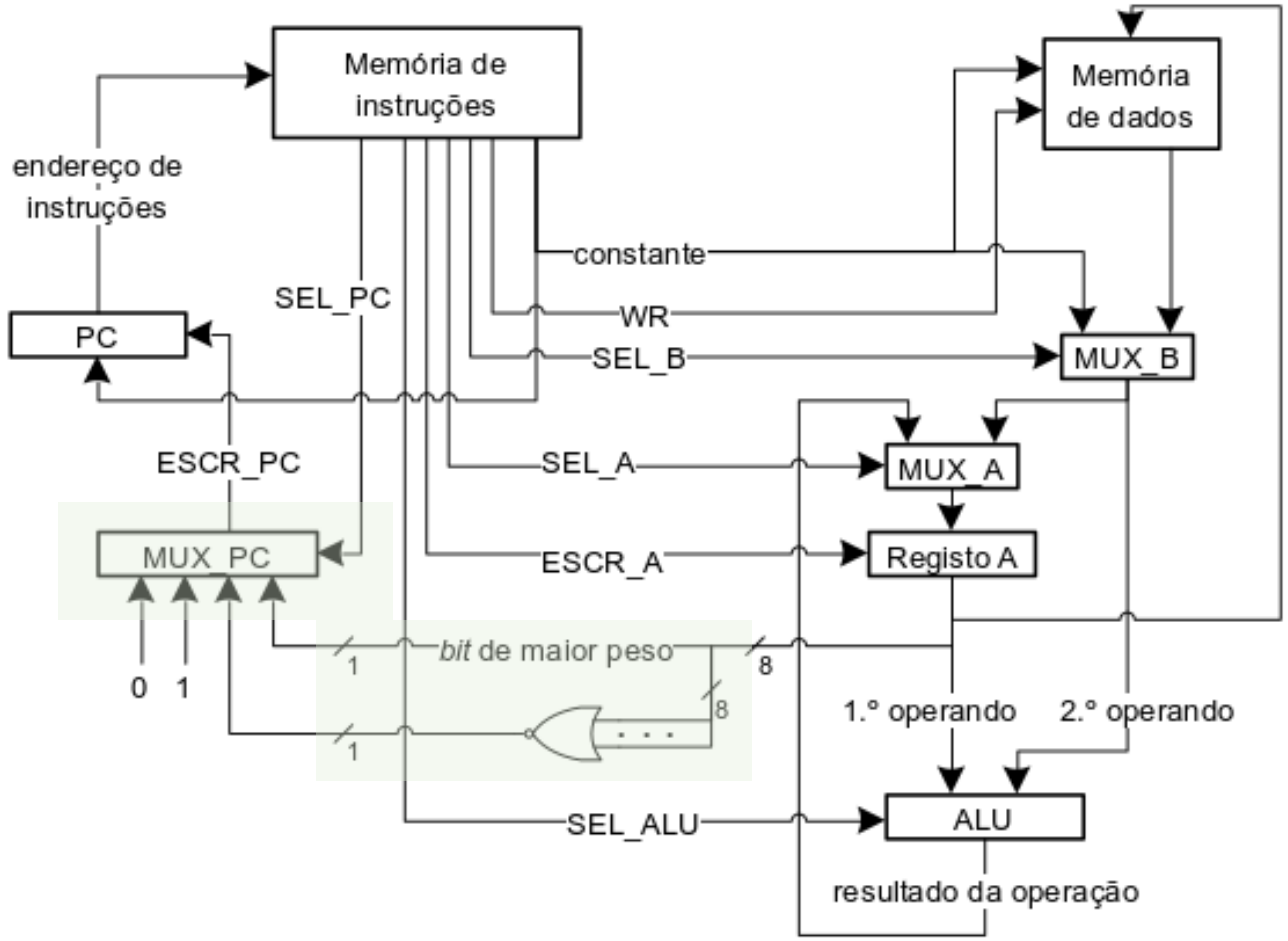
O meu primeiro computador

- **Constante** usada também para endereços
- **MUX_B** suporta ops que guardam constantes no registo A e ops em, que o 2º operando é uma constante. Sinal **SEL_B** controla seleção.
- Entrada do PC já permite **saltos em que a constante** especifica o novo endereço



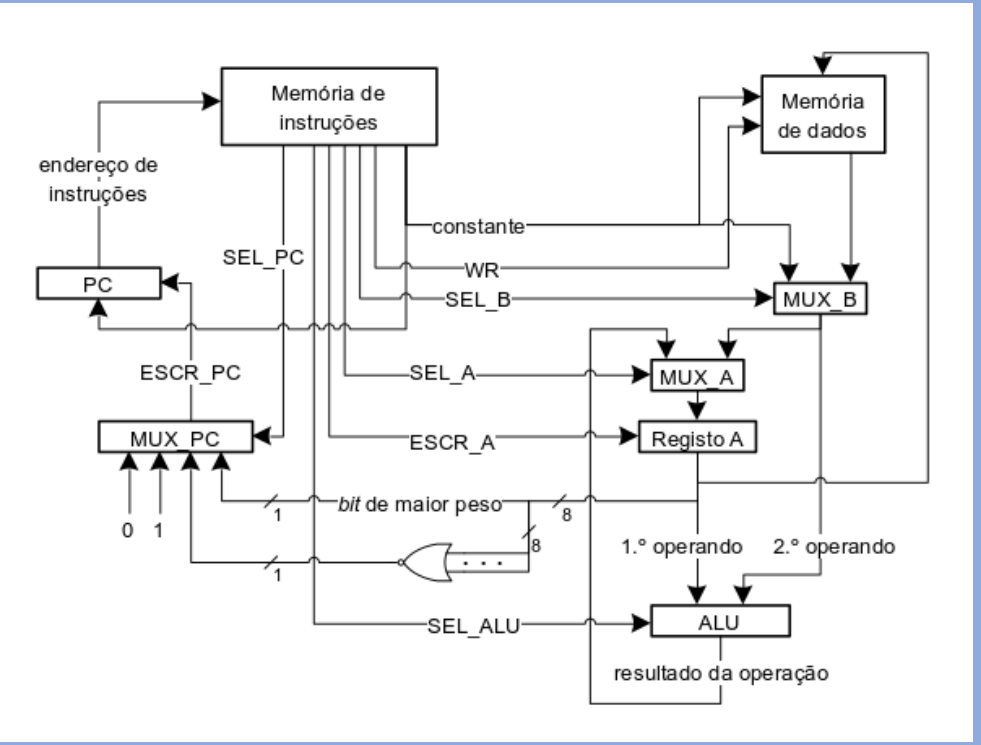
O meu primeiro computador

- Para suportar **saltos condicionais** adiciona-se um multiplexer **MUX_PC** (controlado por sinal **SEL_PC**)
- Situações:
  - Nunca salta
  - Salta sempre, para endereço na constante (e.g., **PC ← 7**)
  - Salta se A = 0
  - Salta se A < 0



refletir.com

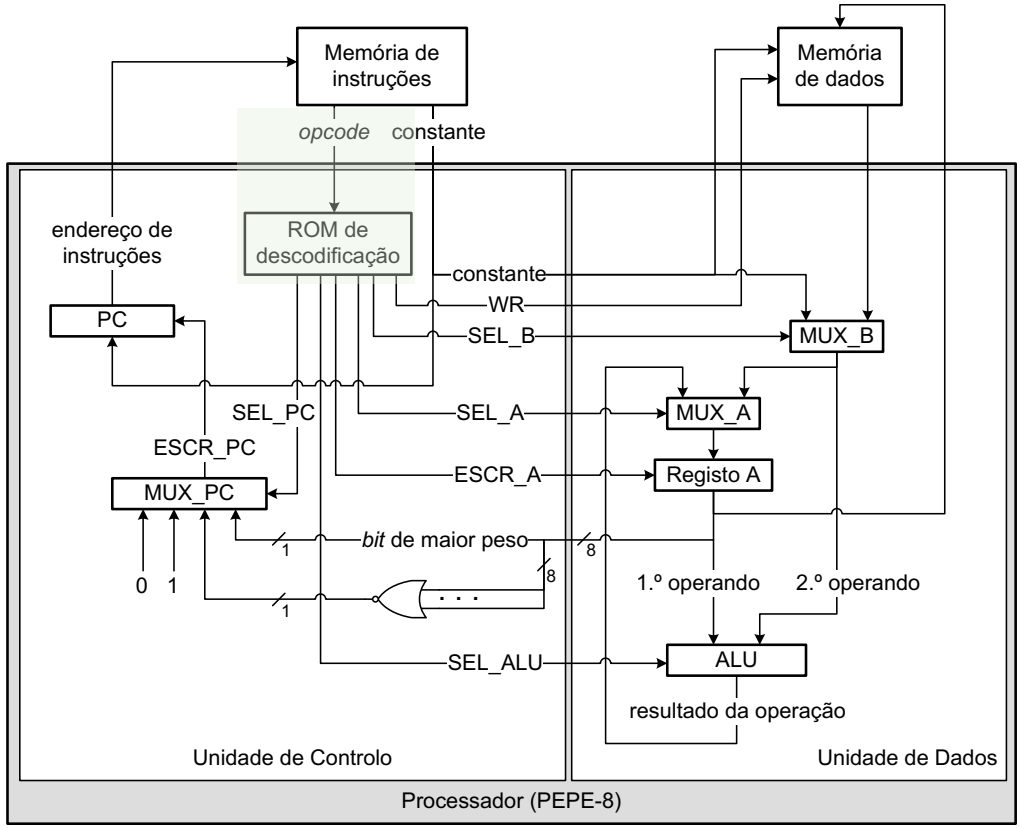
- Quantos bits são necessário para cada instrução, assumindo que o tamanho das células de memória é de 8 bits?
- E qual o tamanho máximo da memória de dados?



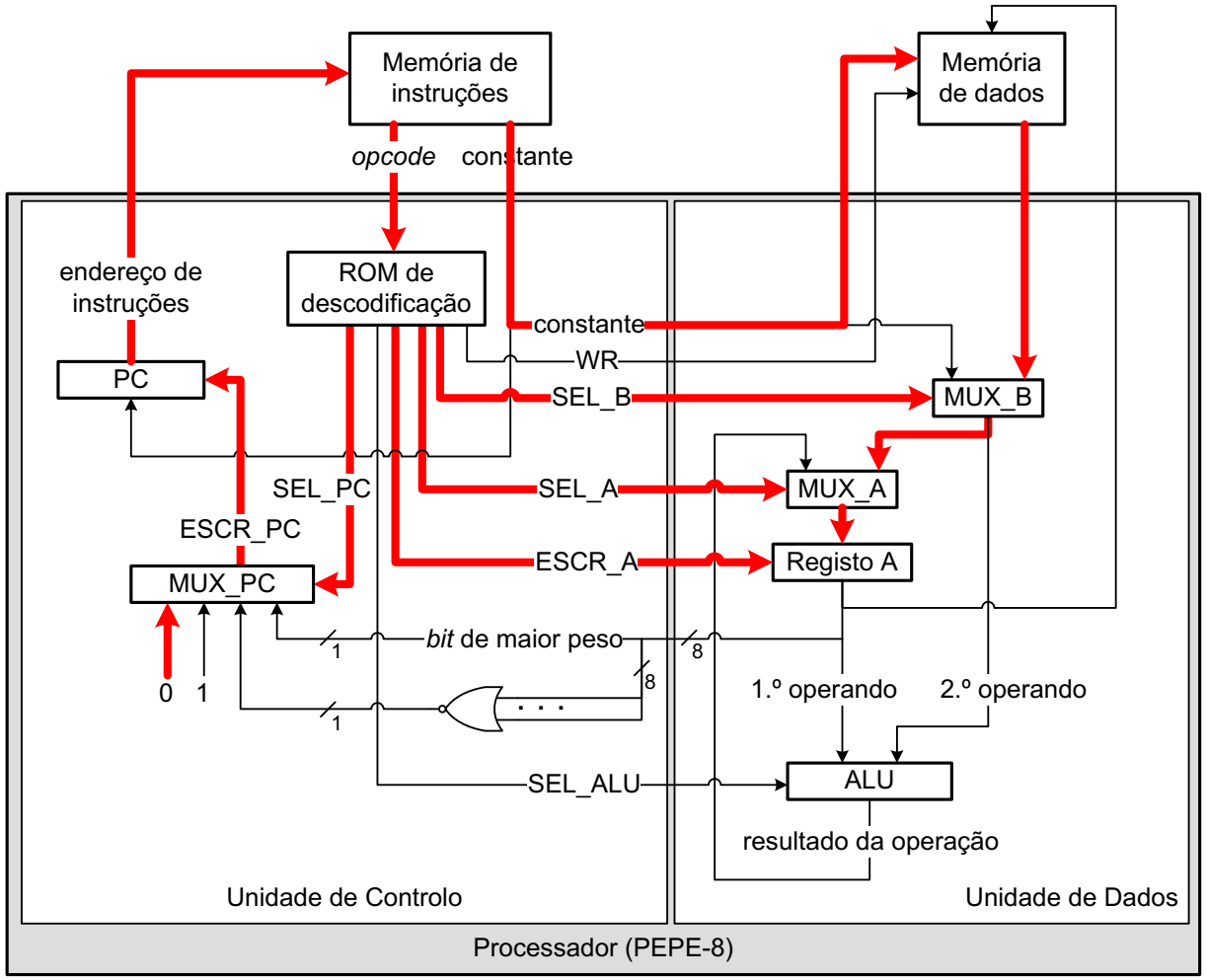
Programação em baixo nível

Arquitetura do PEPE-8

- Há **256 combinações possíveis** dos sinais de controlo
  - Felizmente só são relevantes **15 instruções**
  - Não precisamos de indicar os 8 bits dos sinais
    - Basta um **opcode de 4 bits**, que permite selecionar a instrução

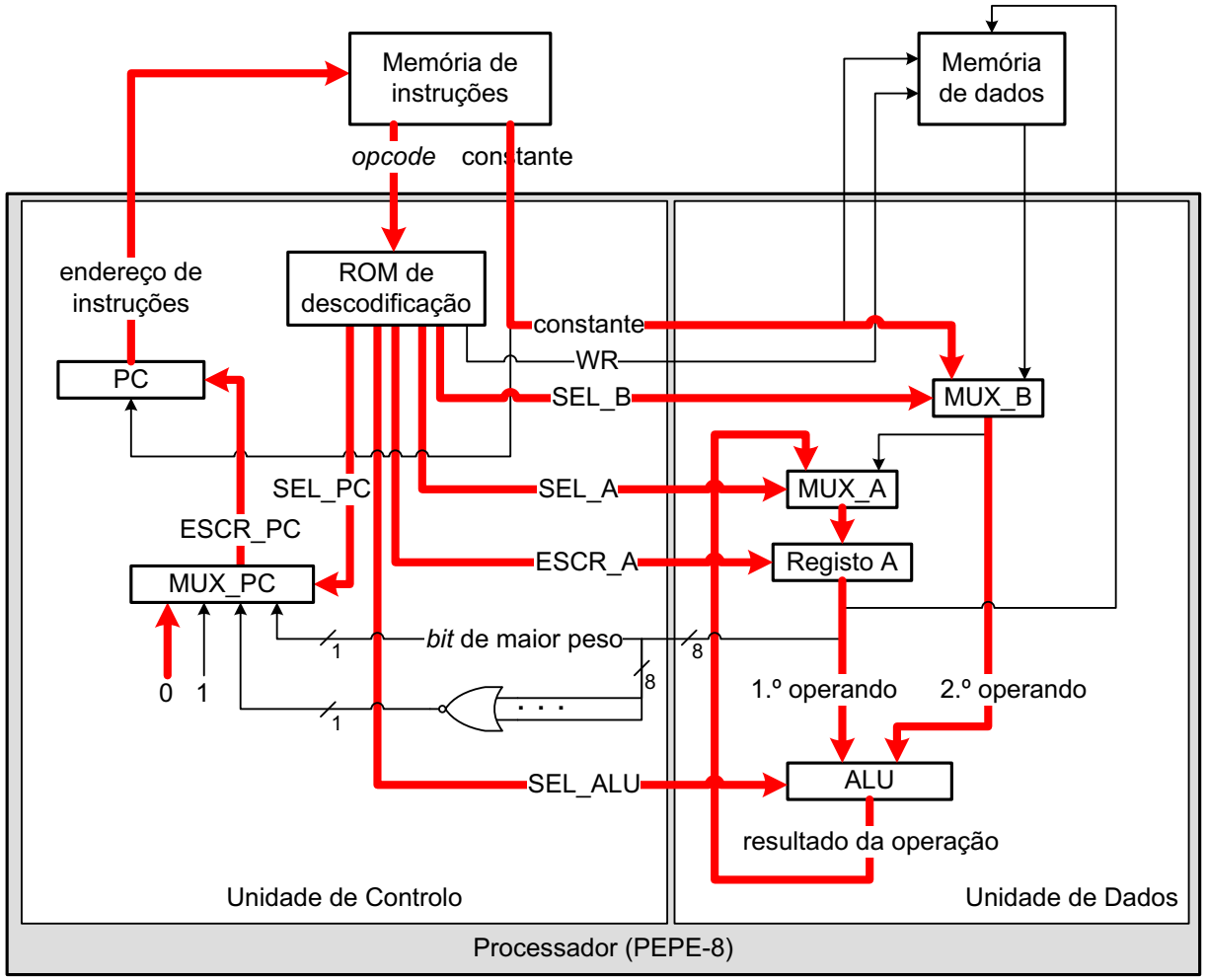


Instrução LD [*endereço*]



A ← *M[endereço]*

Instrução ADD *valor*

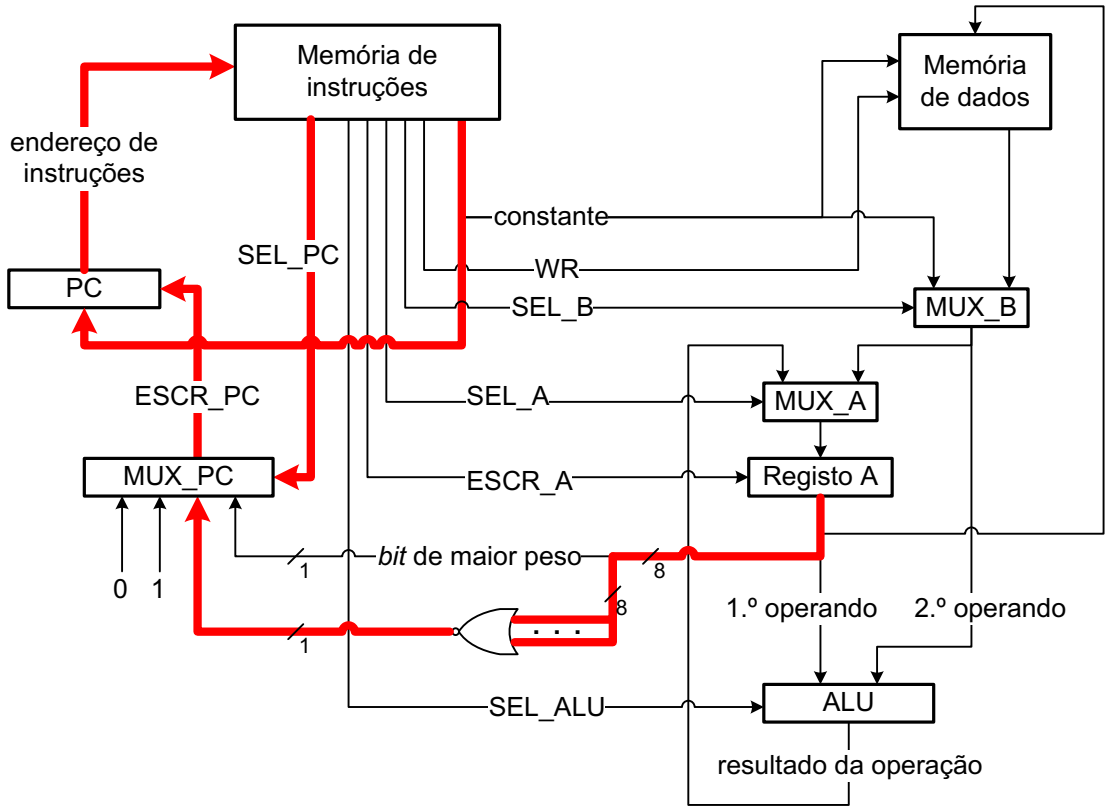


A ← A + *valor*

Instrução *JZ endereço*

- SEL_PC (2 bits):
  - 00: não salta
  - 01: salto incondicional
  - **10: salto condicional (se A = 0)**
  - 11: salto condicional (se A < 0)

(A = 0) : PC ← *endereço*



Linguagem *assembly*

Categoria	Instrução *assembly*		Significado	*Opcode*	Descrição em RTL
Transferência de dados	LD	*valor*	*Load* (imediato)	00H	A ← *valor*
	LD	[*endereço*]	*Load* (memória)	01H	A ← M[*endereço*]
	ST	[*endereço*]	*Store* (memória)	02H	M[*endereço*] ← A
Operações aritméticas	ADD	*valor*	*Add* (imediato)	03H	A ← A + *valor*
	ADD	[*endereço*]	*Add* (memória)	04H	A ← A + M[*endereço*]
	SUB	*valor*	*Subtract* (imediato)	05H	A ← A – *valor*
	SUB	[*endereço*]	*Subtract* (memória)	06H	A ← A – M[*endereço*]
Operações lógicas	AND	*valor*	*AND* (imediato)	07H	A ← A ∧ *valor*
	AND	[*endereço*]	*AND* (memória)	08H	A ← A ∧ M[*endereço*]
	OR	*valor*	*OR* (imediato)	09H	A ← A ∨ *valor*
	OR	[*endereço*]	*OR* (memória)	0AH	A ← A ∨ M[*endereço*]
Saltos	JMP	*endereço*	*Jump*	0BH	PC ← *endereço*
	JZ	*endereço*	*Jump if zero*	0CH	(A=0) : PC ← *endereço*
	JN	*endereço*	*Jump if negative*	0DH	(A<0) : PC ← *endereço*
Diversos	NOP		*No operation*	0EH	

Programação em *assembly*

Programa em RTL		Programa em *assembly*			
0	A ← 0	00H	início:	LD	0
1	M[soma] ← A	01H		ST	[soma]
2	A ← N	02H		LD	N
3	M[iteracao] ← A	03H		ST	[iteracao]
4	(A < 0) : PC ← 12	04H		JN	fim
5	(A = 0) : PC ← 12	05H	teste:	JZ	fim
6	A ← A + M[soma]	06H		ADD	[soma]
7	M[soma] ← A	07H		ST	[soma]
8	A ← M[iteracao]	08H		LD	[iteracao]
9	A ← A – 1	09H		SUB	1
10	M[iteracao] ← A	0AH		ST	[iteracao]
11	PC ← 5	0BH		JMP	teste
12	PC ← 12	0CH	fim:	JMP	fim

Vamos “assemblar”

Opcode	*Assembly*	
00H	LD	*valor*
01H	LD	[*endereço*]
02H	ST	[*endereço*]
03H	ADD	*valor*
04H	ADD	[*endereço*]
05H	SUB	*valor*
06H	SUB	[*endereço*]
07H	AND	*valor*
08H	AND	[*endereço*]
09H	OR	*valor*
0AH	OR	[*endereço*]
0BH	JMP	*endereço*
0CH	JZ	*endereço*
0DH	JN	*endereço*
0EH	NOP	

End.	***Assembly***			**Máquina**
00H	início:	LD	0	00 00H
01H		ST	[soma]	02 40H
02H		LD	N	
03H		ST	[iteracao]	
04H		JN	fim	
05H	teste:	JZ	fim	
06H		ADD		
07H		ST		

- As mnemónicas são convertidas em opcodes
- As constantes simbólicas são convertidas em números

Assumindo: **soma** = 40H; **iteracao** = 41H; **N** = 8

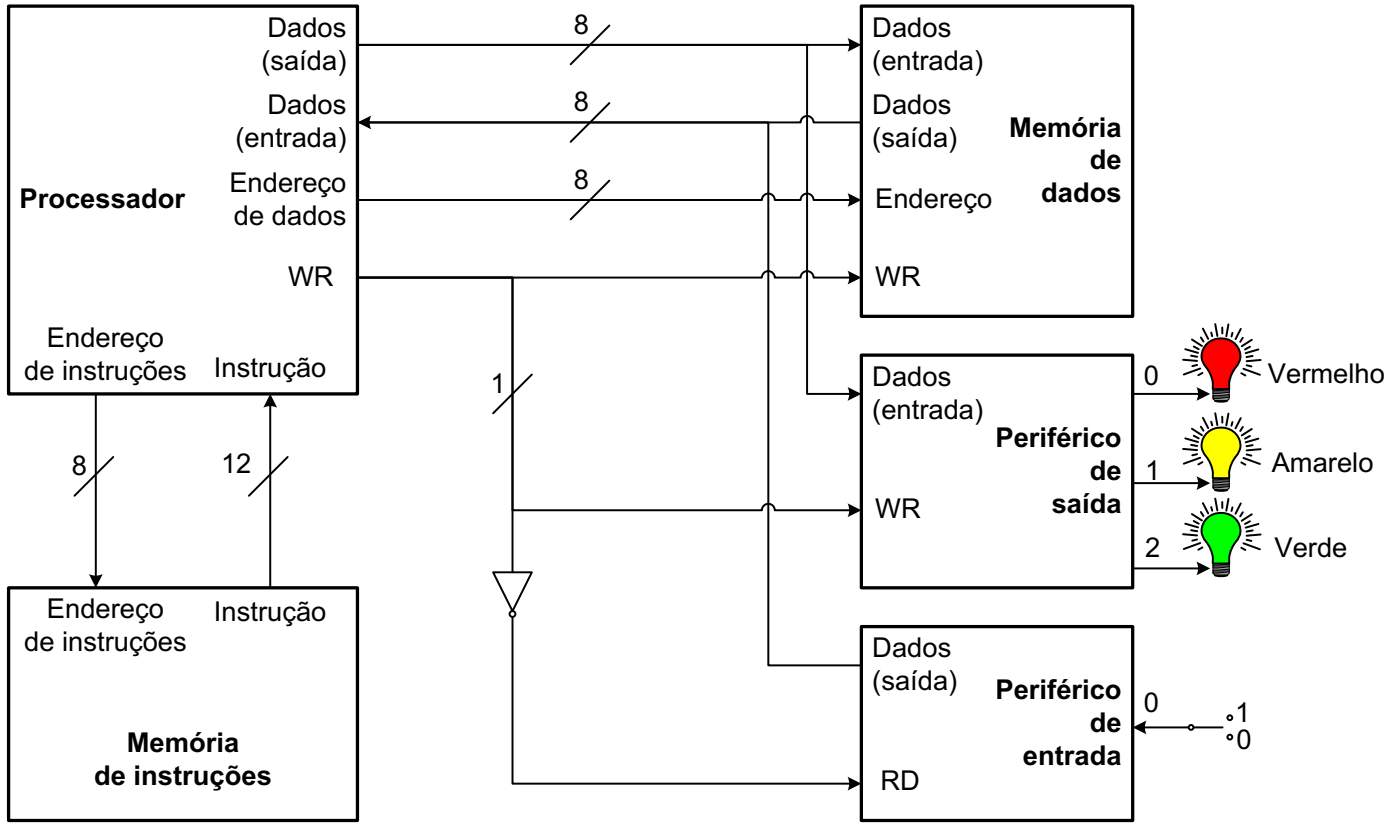
Vamos “assemblar”

Opcode	*Assembly*	
00H	LD	*valor*
01H	LD	[*endereço*]
02H	ST	[*endereço*]
03H	ADD	*valor*
04H	ADD	[*endereço*]
05H	SUB	*valor*
06H	SUB	[*endereço*]
07H	AND	*valor*
08H	AND	[*endereço*]
09H	OR	*valor*
0AH	OR	[*endereço*]
0BH	JMP	*endereço*
0CH	JZ	*endereço*
0DH	JN	*endereço*
0EH	NOP	

End.	***Assembly***			**Máquina**
00H	início:	LD	0	00 00H
01H		ST	[soma]	02 40H
02H		LD	N	00 08H
03H		ST	[iteracao]	02 41H
04H		JN	fim	0D 0CH
05H	teste:	JZ	fim	0C 0CH
06H		ADD	[soma]	04 40H
07H		ST	[soma]	02 40H
08H		LD	[iteracao]	01 41H
09H		SUB	1	05 01H
0AH		ST	[iteracao]	02 41H
0BH		JMP	teste	0B 05H
0CH	fim:	JMP	fim	0B 0CH

Assumindo: **soma** = 40H; **iteracao** = 41H; **N** = 8

Processador, memória e periféricos



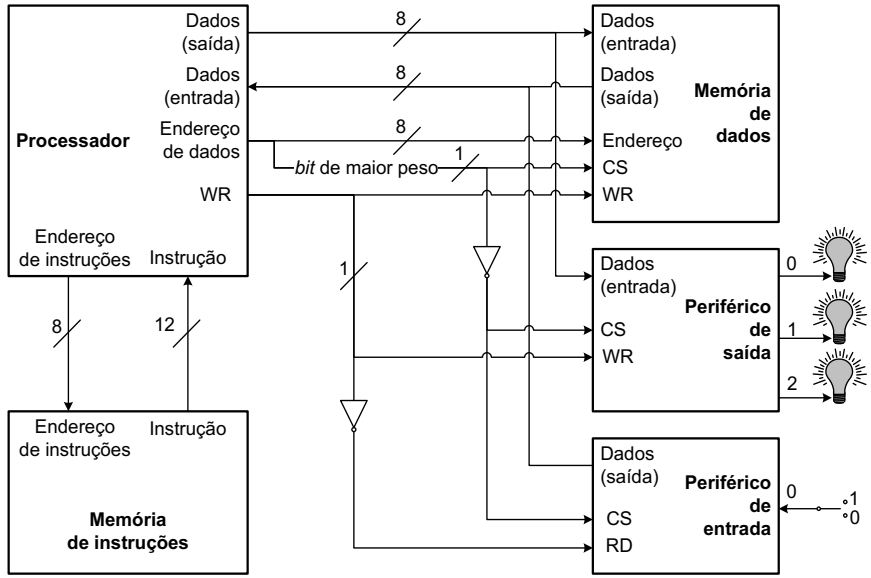
- Para o processador, um **periférico** é apenas uma **memória de dados**
  - Com uma diferença: só suporta uma operação (escrita **ou** leitura)
  - Necessário sinal adicional – CS, próximo slide – para selecionar qual o dispositivo a usar (memória ou periférico)

Programa: semáforo simples

```
; constantes de dados
vermelho     EQU    01H          ; valor do vermelho (lâmpada liga ao bit 0)
amarelo      EQU    02H          ; valor do amarelo (lâmpada liga ao bit 1)
verde        EQU    04H          ; valor do verde (lâmpada liga ao bit 2)

; constantes de endereços
semáforo     EQU    80H          ; endereço 128 (periférico de saída)

; programa
início:      LD     verde        ; Carrega o registo A com o valor para semáforo verde
             ST     [semáforo]   ; Atualiza o periférico de saída
semVerde:    NOP                 ; faz um compasso de espera
             NOP                 ; faz um compasso de espera
             NOP                 ; faz um compasso de espera
             LD     amarelo      ; Carrega o registo A com o valor para semáforo amarelo
             ST     [semáforo]   ; Atualiza o periférico de saída
semAmarelo: LD      vermelho     ; Carrega o registo A com o valor para semáforo vermelho
             ST     [semáforo]   ; Atualiza o periférico de saída
semVerm:     NOP                 ; faz um compasso de espera
             NOP                 ; faz um compasso de espera
             NOP                 ; faz um compasso de espera
             NOP                 ; faz um compasso de espera
             JMP    início       ; vai fazer mais uma ronda
```



TÉCNICO LISBOA

Outro exemplo: contar bits a 1

posição em teste	Máscara	Valor (76H)	Valor AND máscara	Bit a 1	Contador de bits a 1
0	0000 000**1**	0111 011**0**	0000 000**0**	Não	0
1	0000 00**1**0	0111 01**1**0	0000 00**1**0	Sim	1
2	0000 0**1**00	0111 0**1**10	0000 0**1**00	Sim	2
3	0000 **1**000	0111 **0**110	0000 **0**000	Não	2
4	000**1** 0000	011**1** 0110	000**1** 0000	Sim	3
5	00**1**0 0000	01**1**1 0110	00**1**0 0000	Sim	4
6	0**1**00 0000	0**1**11 0110	0**1**00 0000	Sim	5
7	**1**000 0000	**0**111 0110	**0**000 0000	Não	5

1.	**contador** ← 0	(inicializa contador de bits a zero)
2.	**máscara** ← 01H	(inicializa máscara a 0000 0001)
3.	Se (**máscara** ∧ **valor** = 0) salta para 5	(se o bit está a zero, passa ao próximo)
4.	**contador** ← **contador** + 1	(bit está a 1, incrementa contador)
5.	Se (**máscara** ← 80H) salta para 8	(se já testou a última máscara, termina)
6.	**máscara** ← **máscara** + **máscara**	(duplica máscara para deslocar bit para a esquerda)
7.	Salta para 3	(vai testar o novo bit)
8.	Salta para 8	(fim do algoritmo)

Definições

Assembly	
LD	*valor*
LD	[*endereço*]
ST	[*endereço*]
ADD	*valor*
ADD	[*endereço*]
SUB	*valor*
SUB	[*endereço*]
AND	*valor*
AND	[*endereço*]
OR	*valor*
OR	[*endereço*]
JMP	*endereço*
JZ	*endereço*
JN	*endereço*
NOP	

```
valor           EQU     76H     ; Valor cujo número de bits a 1 é para ser contado
máscaraInicial  EQU     01H     ; 0000 0001 em binário (máscara inicial)
máscaraFinal    EQU     80H     ; 1000 0000 em binário (máscara final)

contador        EQU     00H     ; Endereço da célula de memória que guarda
                                ; o valor corrente do contador de bits a 1
máscara         EQU     01H     ; Endereço da célula de memória que guarda
                                ; o valor corrente da máscara
```

1. **contador** ← 0 (inicializa contador de bits a zero)
2. **máscara** ← **01H** (inicializa máscara a 0000 0001)
3. Se (**máscara** ∧ **valor** = 0) salta para 5 (se o bit está a zero, passa ao próximo)
4. **contador** ← **contador** + 1 (bit está a 1, incrementa contador)
5. Se (**máscara** ← **80H**) salta para 8 (se já testou a última máscara, termina)
6. **máscara** ← **máscara** + **máscara** (duplica máscara para deslocar bit para a esquerda)
7. Salta para 3 (vai testar o novo bit)
8. Salta para 8 (fim do algoritmo)

TÉCNICO LISBOA

Inicializar contador e máscara

Assembly	
LD	*valor*
LD	[*endereço*]
ST	[*endereço*]
ADD	*valor*
ADD	[*endereço*]
SUB	*valor*
SUB	[*endereço*]
AND	*valor*
AND	[*endereço*]
OR	*valor*
OR	[*endereço*]
JMP	*endereço*
JZ	*endereço*
JN	*endereço*
NOP	

```
valor           EQU     76H     ; Valor cujo número de bits a 1 é para ser contado
máscaraInicial  EQU     01H     ; 0000 0001 em binário (máscara inicial)
máscaraFinal    EQU     80H     ; 1000 0000 em binário (máscara final)

contador        EQU     00H     ; Endereço da célula de memória que guarda
                                ; o valor corrente do contador de bits a 1
máscara         EQU     01H     ; Endereço da célula de memória que guarda
                                ; o valor corrente da máscara
início:         LD      0               ; Inicializa o registo A a zero
                ST      [contador]      ; Inicializa o contador de bits com zero
                LD      máscaraInicial  ; Carrega valor da máscara inicial
                ST      [máscara]       ; Atualiza na memória
```

1. **contador ← 0** **(inicializa contador de bits a zero)**
2. **máscara ← 01H** **(inicializa máscara a 0000 0001)**
3. Se (máscara ∧ valor = 0) salta para 5 (se o bit está a zero, passa ao próximo)
4. contador ← contador + 1 (bit está a 1, incrementa contador)
5. Se (máscara ← 80H) salta para 8 (se já testou a última máscara, termina)
6. máscara ← máscara + máscara (duplica máscara para deslocar bit para a esquerda)
7. Salta para 3 (vai testar o novo bit)
8. Salta para 8 (fim do algoritmo)

TÉCNICO LISBOA

Testar se o bit é 0 ou 1

Assembly	
LD	*valor*
LD	[*endereço*]
ST	[*endereço*]
ADD	*valor*
ADD	[*endereço*]
SUB	*valor*
SUB	[*endereço*]
AND	*valor*
AND	[*endereço*]
OR	*valor*
OR	[*endereço*]
JMP	*endereço*
JZ	*endereço*
JN	*endereço*
NOP	

```
valor           EQU   76H      ; Valor cujo número de bits a 1 é para ser contado
máscaraInicial  EQU   01H      ; 0000 0001 em binário (máscara inicial)
máscaraFinal    EQU   80H      ; 1000 0000 em binário (máscara final)

contador        EQU   00H      ; Endereço da célula de memória que guarda
                               ; o valor corrente do contador de bits a 1
máscara         EQU   01H      ; Endereço da célula de memória que guarda
                               ; o valor corrente da máscara
início:         LD    0        ; Inicializa o registo A a zero
                ST    [contador] ; Inicializa o contador de bits com zero
                LD    máscaraInicial      ; Carrega valor da máscara inicial
                ST    [máscara] ; Atualiza na memória
teste:          AND   valor    ; Isola o bit que se quer ver se é 1
                JZ    próximo  ; Se o bit for zero, passa à máscara seguinte
                LD    [contador] ; O bit é 1, vai buscar o valor atual do contador
                ADD   1        ; Incrementa-o
                ST    [contador]; e atualiza de novo na memória
próximo:
```

1. contador ← 0 (inicializa contador de bits a zero)
2. máscara ← 01H **(inicializa máscara a 0000 0001)**
3. **Se (máscara ∧ valor = 0)salta para 5 (se o bit está a zero, passa ao próximo)**
4. **contador ← contador + 1 (bit está a 1, incrementa contador)**
5. Se (máscara ← 80H) salta para 8 (se já testou a última máscara, termina)
6. máscara ← máscara + máscara (duplica máscara para deslocar bit para a esquerda)
7. Salta para 3 (vai testar o novo bit)
8. Salta para 8 (fim do algoritmo)

TÉCNICO LISBOA

Testar se já se testou tudo

Assembly	
LD	*valor*
LD	[*endereço*]
ST	[*endereço*]
ADD	*valor*
ADD	[*endereço*]
SUB	*valor*
SUB	[*endereço*]
AND	*valor*
AND	[*endereço*]
OR	*valor*
OR	[*endereço*]
JMP	*endereço*
JZ	*endereço*
JN	*endereço*
NOP	

1. contador ← 0 (inicializa contador de bits a zero)
2. máscara ← 01H (inicializa máscara a 0000 0001)
3. Se (máscara ∧ valor = 0)salta para 5 (se o bit está a zero, passa ao próximo)
4. contador ← contador + 1 (bit está a 1, incrementa contador)
5. **Se (máscara ← 80H) salta para 8** **(se já testou a última máscara, termina)**
6. máscara ← máscara + máscara (duplica máscara para deslocar bit para a esquerda)
7. Salta para 3 (vai testar o novo bit)
8. **Salta para 8** **(fim do algoritmo)**

```
            LD      máscaraInicial  ; Carrega valor da máscara inicial
            ST      [máscara]   ; Atualiza na memória
teste:      AND     valor       ; Isola o bit que se quer ver se é 1
            JZ      próximo     ; Se o bit for zero, passa à máscara seguinte
            LD      [contador]  ; O bit é 1, vai buscar o valor atual do contador
            ADD     1           ; Incrementa-o
            ST      [contador]  ; e atualiza de novo na memória
próximo:    LD      [máscara]           ; Vai buscar de novo a máscara atual
            SUB     máscaraFinal        ; Compara com a máscara final, fazendo a subtração
            JZ      fim                 ; Se der zero, eram iguais e portanto já terminou



fim:        JMP     fim         ; Fim do programa
```

Duplicar a máscara (“deslocar bit 1 para a esquerda”)

Assembly	
LD	*valor*
LD	[*endereço*]
ST	[*endereço*]
ADD	*valor*
ADD	[*endereço*]
SUB	*valor*
SUB	[*endereço*]
AND	*valor*
AND	[*endereço*]
OR	*valor*
OR	[*endereço*]
JMP	*endereço*
JZ	*endereço*
JN	*endereço*
NOP	

1. contador ← 0 (inicializa contador de bits a zero)
2. máscara ← 01H (inicializa máscara a 0000 0001)
3. Se (máscara ∧ valor = 0)salta para 5 (se o bit está a zero, passa ao próximo)
4. contador ← contador + 1 (bit está a 1, incrementa contador)
5. Se (máscara ← 80H) salta para 8 (se já testou a última máscara, termina)
6. **máscara ← máscara + máscara (duplica máscara para deslocar bit para a esquerda)**
7. **Salta para 3 (vai testar o novo bit)**
8. Salta para 8 (fim do algoritmo)

```
            LD      máscaraInicial  ; Carrega valor da máscara inicial
            ST      [máscara]   ; Atualiza na memória
teste:      AND     valor       ; Isola o bit que se quer ver se é 1
            JZ      próximo     ; Se o bit for zero, passa à máscara seguinte
            LD      [contador]  ; O bit é 1, vai buscar o valor atual do contador
            ADD     1           ; Incrementa-o
            ST      [contador]  ; e atualiza de novo na memória
próximo:    LD      [máscara]           ;  Vai buscar de novo a máscara atual
            SUB     máscaraFinal        ; Compara com a máscara final, fazendo a subtração
            JZ      fim                 ; Se der zero, eram iguais e portanto já terminou
            LD      [máscara]           ; Tem de carregar a máscara de novo
            ADD     [máscara]           ; Soma com ela própria para a multiplicar por 2
            ST      [máscara]           ; Atualiza o valor da máscara na memória
            JMP     teste               ; Vai fazer mais um teste com a nova máscara
fim:        JMP     fim         ; Fim do programa
```

TÉCNICO LISBOA

Solução completa

```
valor           EQU     76H     ; Valor cujo número de bits a 1 é para ser contado
máscaraInicial  EQU     01H     ; 0000 0001 em binário (máscara inicial)
máscaraFinal    EQU     80H     ; 1000 0000 em binário (máscara final)

contador        EQU     00H     ; Endereço da célula de memória que guarda
                                ; o valor corrente do contador de bits a 1
máscara         EQU     01H     ; Endereço da célula de memória que guarda
                                ; o valor corrente da máscara
início:         LD      0       ; Inicializa o registo A a zero
                ST      [contador] ; Inicializa o contador de bits com zero
                LD      máscaraInicial      ; Carrega valor da máscara inicial
                ST      [máscara] ; Atualiza na memória
teste:          AND     valor   ; Isola o bit que se quer ver se é 1
                JZ      próximo ; Se o bit for zero, passa à máscara seguinte
                LD      [contador] ; O bit é 1, vai buscar o valor atual do contador
                ADD     1       ; Incrementa-o
                ST      [contador] ; e atualiza de novo na memória
próximo:        LD      [máscara] ; Vai buscar de novo a máscara atual
                SUB     máscaraFinal        ; Compara com a máscara final, fazendo a subtração
                JZ      fim     ; Se der zero, eram iguais e portanto já terminou
                LD      [máscara] ; Tem de carregar a máscara de novo
                ADD     [máscara] ; Soma com ela própria para a multiplicar por 2
                ST      [máscara] ; Atualiza o valor da máscara na memória
                JMP     teste   ; Vai fazer mais um teste com a nova máscara
fim:            JMP     fim     ; Fim do programa
```

Bibliografia

Recomendada

- **[Delgado&Ribeiro_2014]**
  - **Secções 3.1-3.5**

Secundária/adicional

- [Patterson&Hennessy_2021]
  - Cap. 2