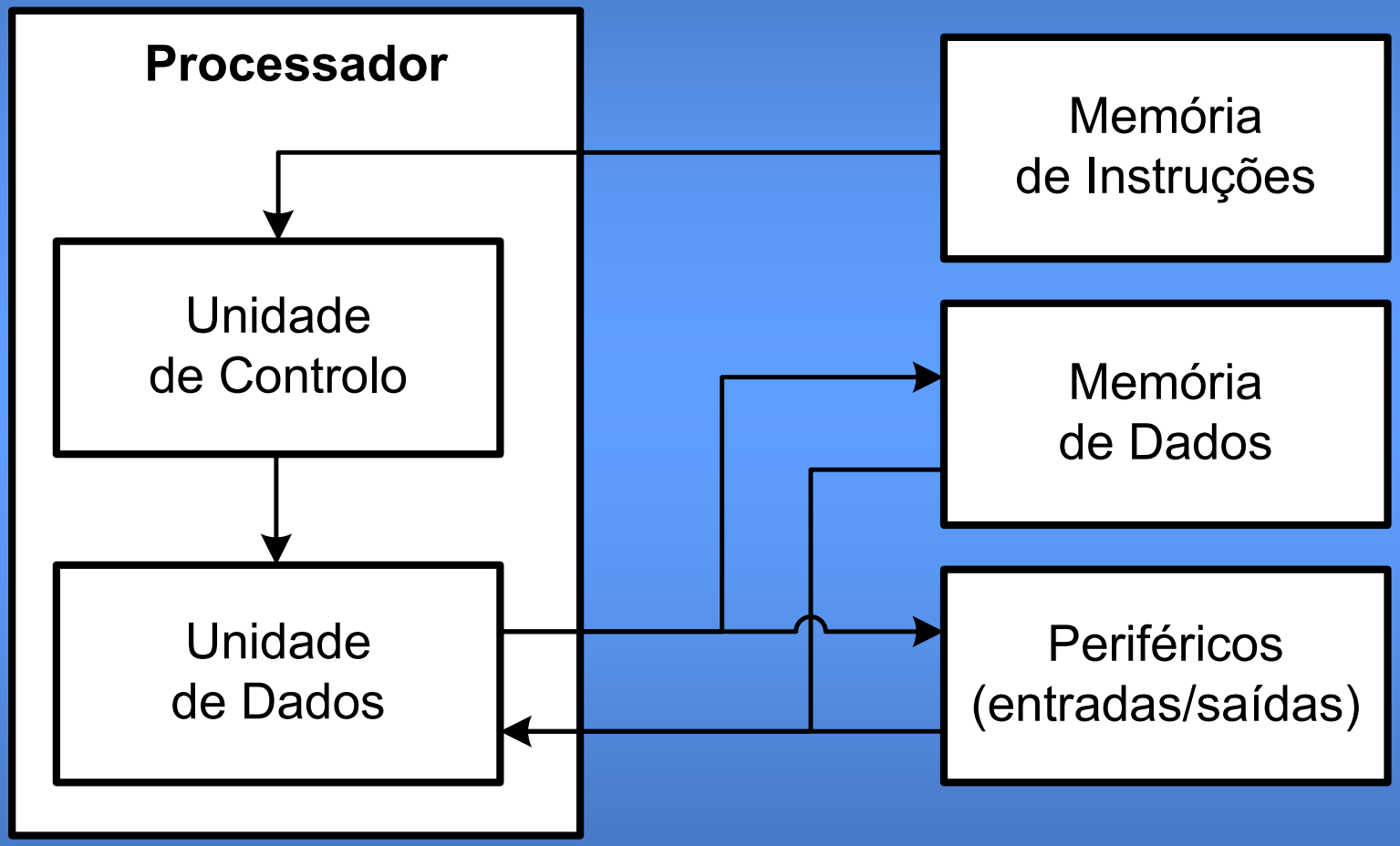
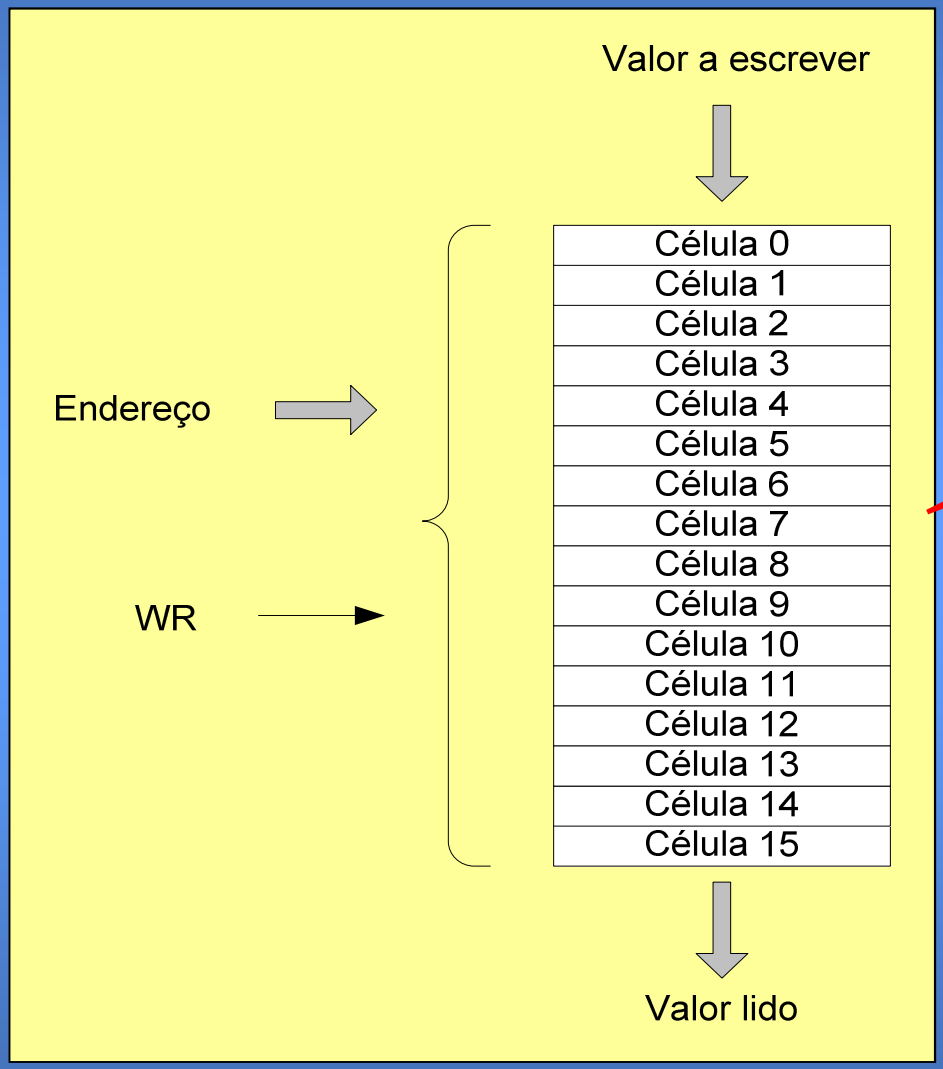
Funcionamento básico de um computador

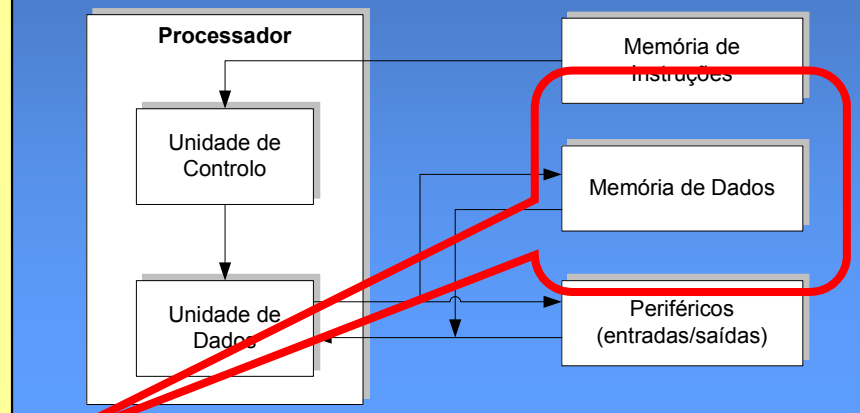
- Processador
  - Unidade de dados
  - Unidade de controlo
- Arquitetura de computador básica
- Linguagem *assembly*
- Exemplos

Estrutura de um computador



Memória: elemento fundamental

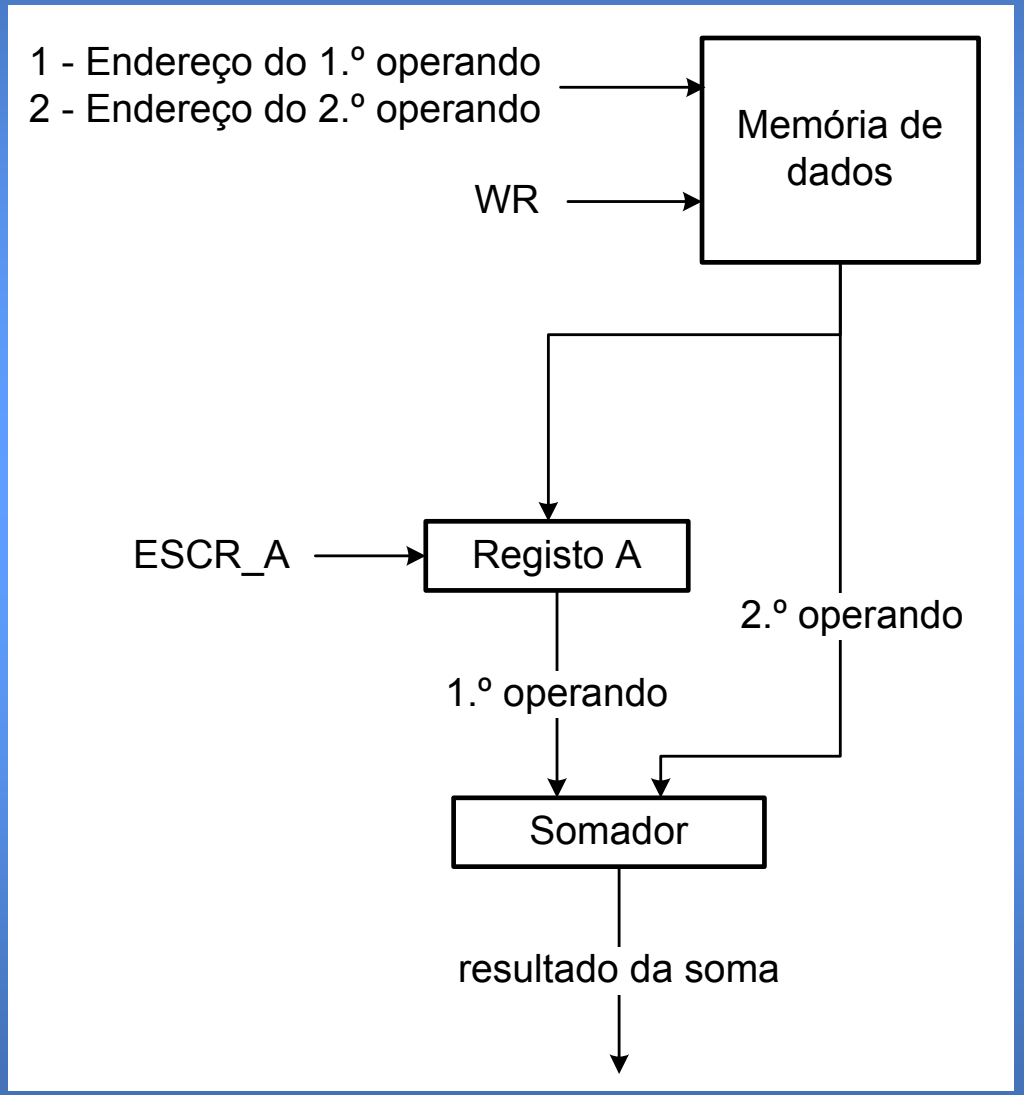




Os periféricos são uma espécie de células de memória cujos bits ligam ao mundo exterior.

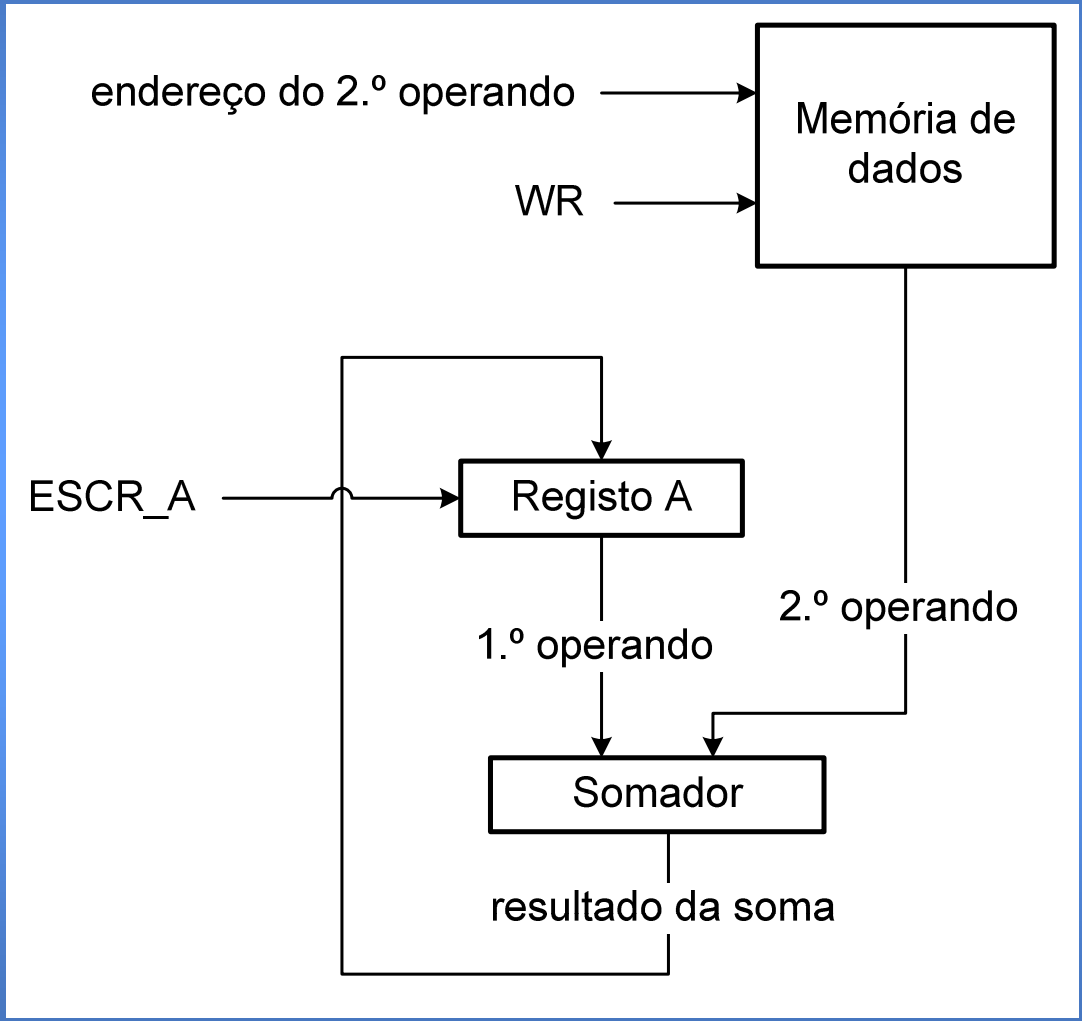
Unidade de dados (versão 1)

- Problema: Um somador tem duas entradas, mas a memória só tem uma saída.
- Solução: um registo para memorizar um dos operandos



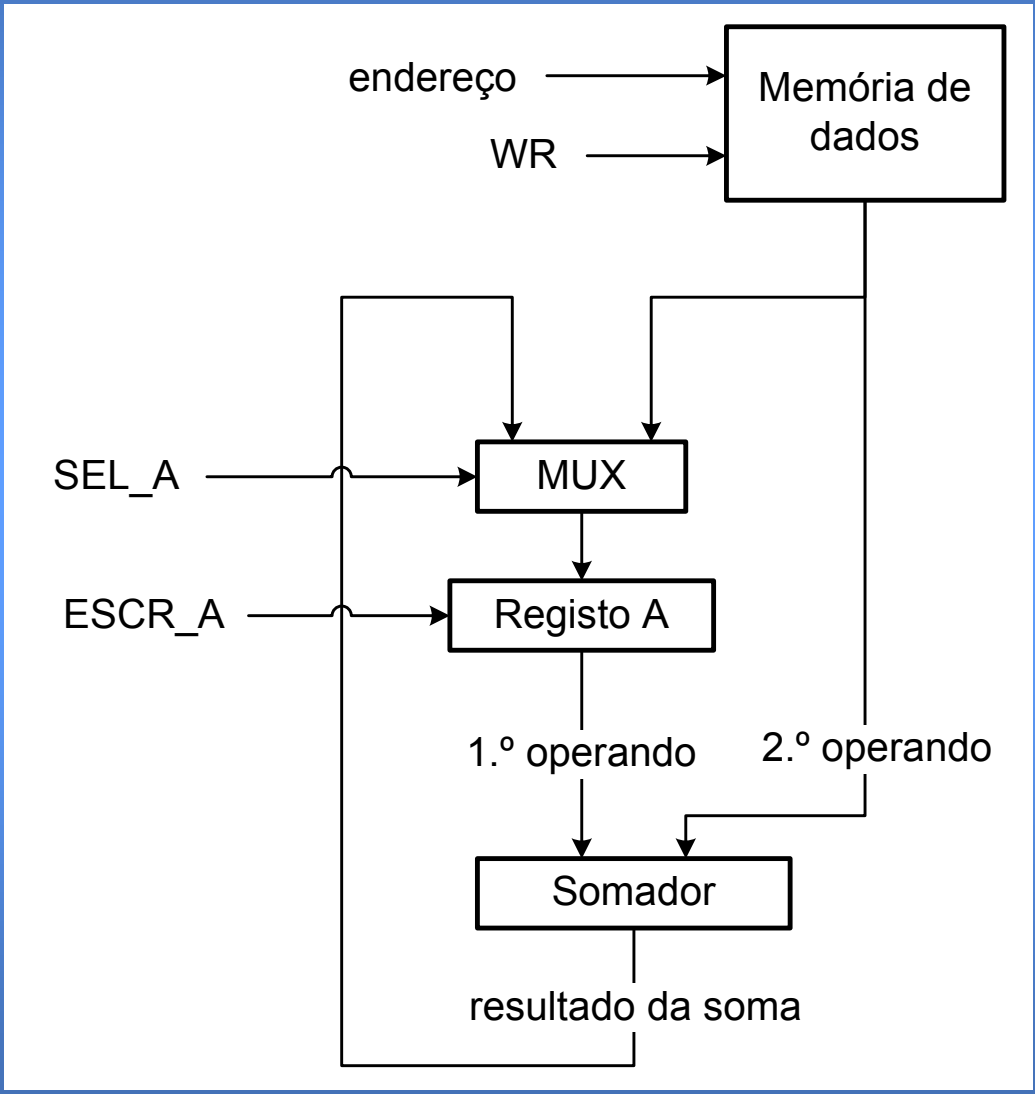
Unidade de dados (versão 2)

- Problema: Resultado não pode ir para a memória
- Solução: Usar o registo para esse efeito (não pode ser *latch*).
- O registo chama-se tradicionalmente "acumulador".



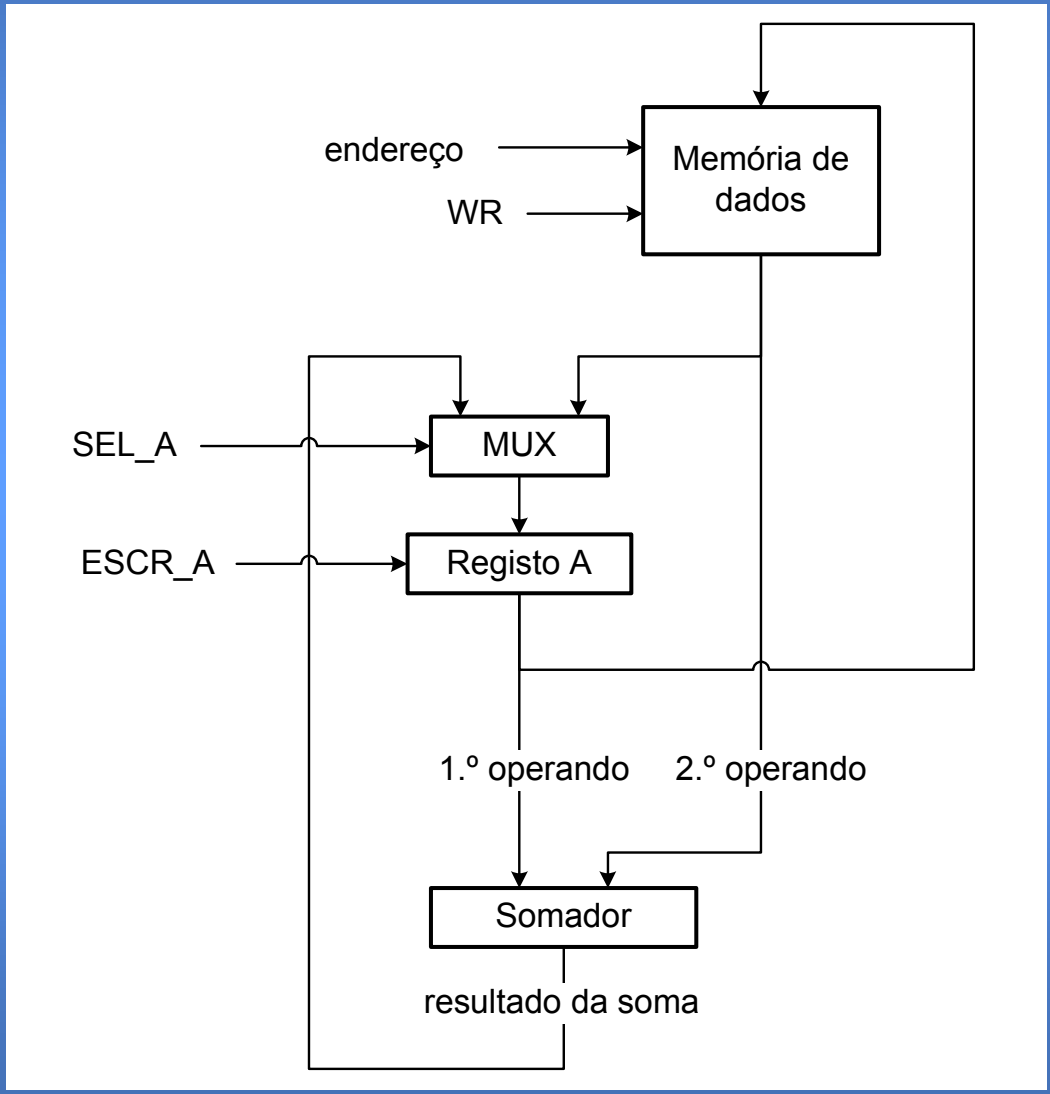
Unidade de dados (versão 3)

- Problema: Entrada do registo não pode vir de dois lados (registo e memória)
- Solução: Usar um multiplexer para selecionar a entrada.



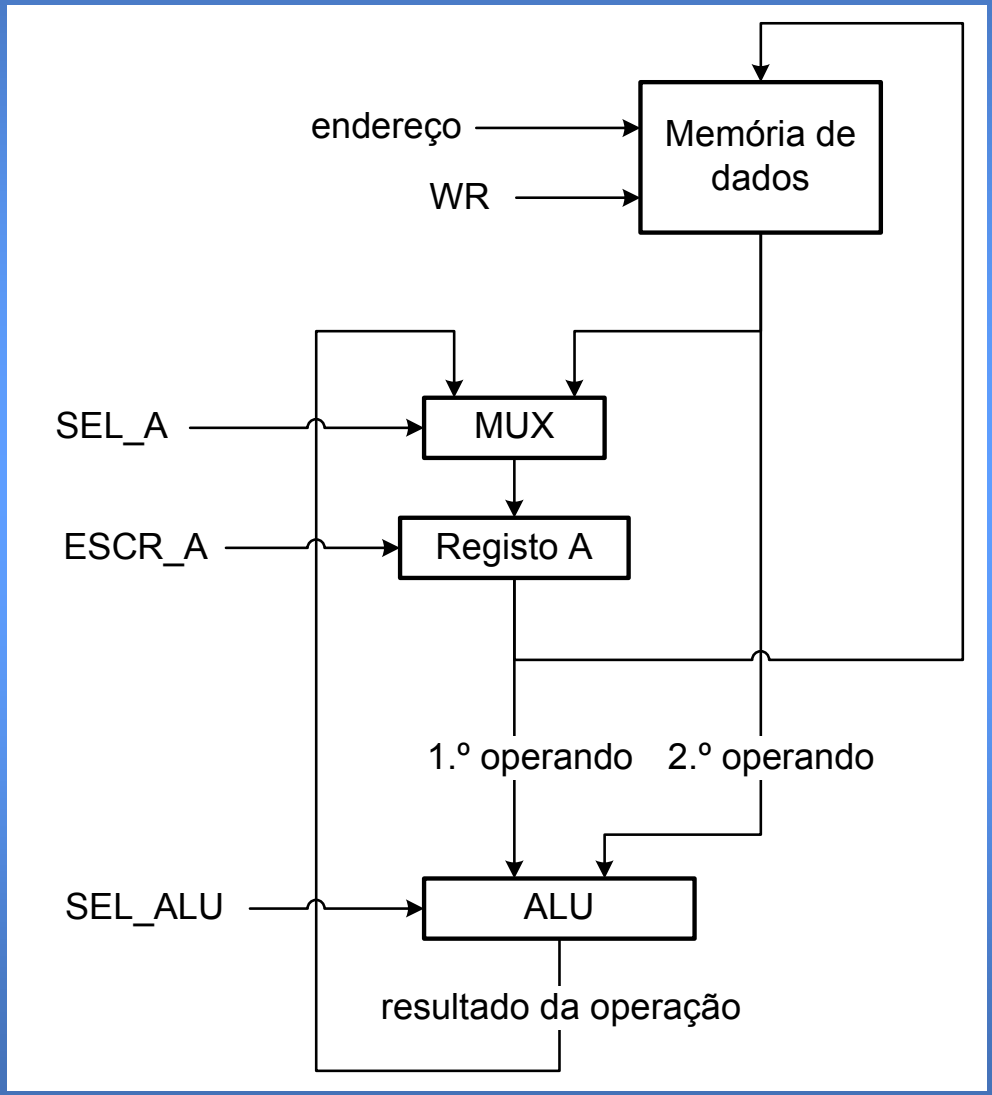
Unidade de dados (versão 4)

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

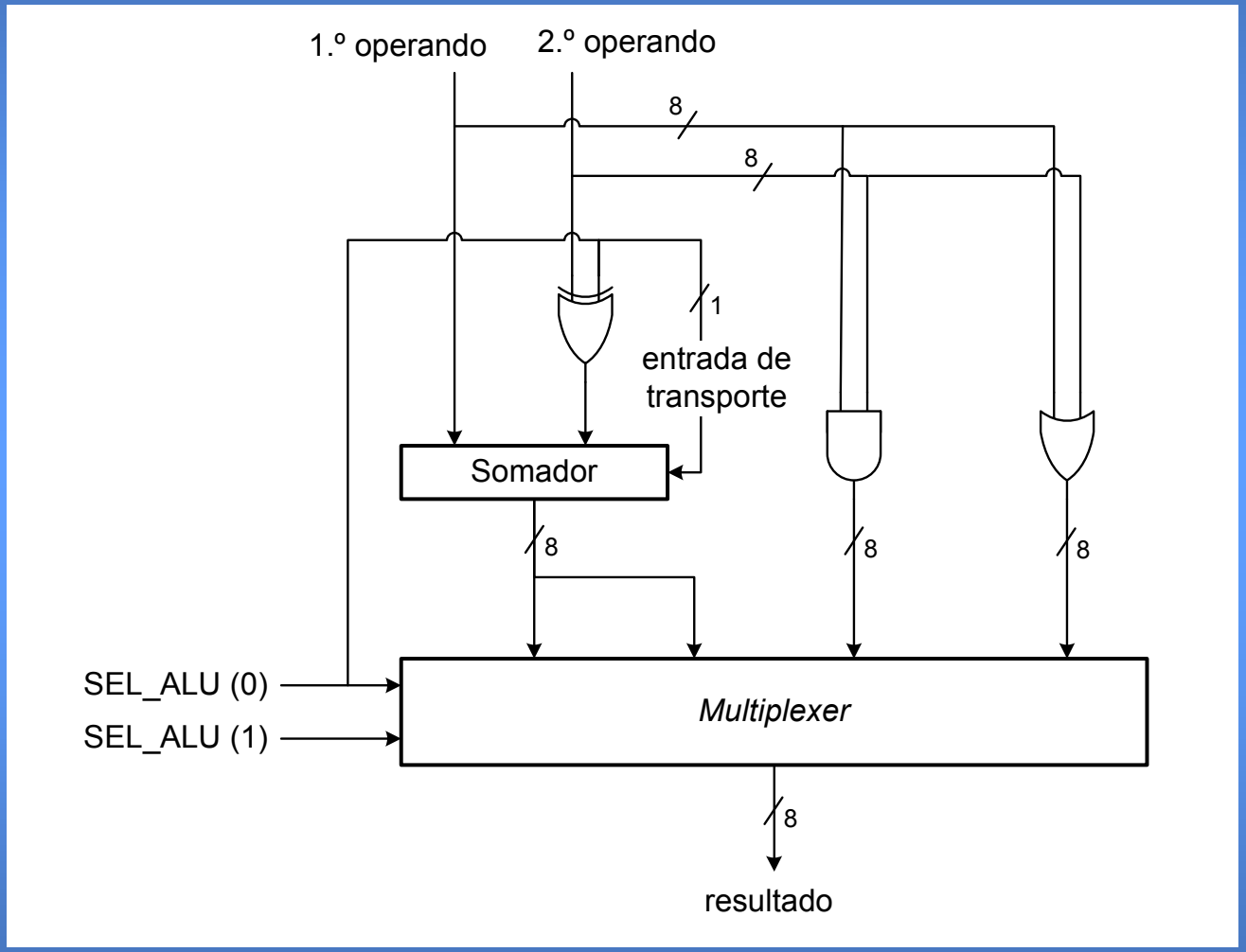


Unidade de dados (versão 5)

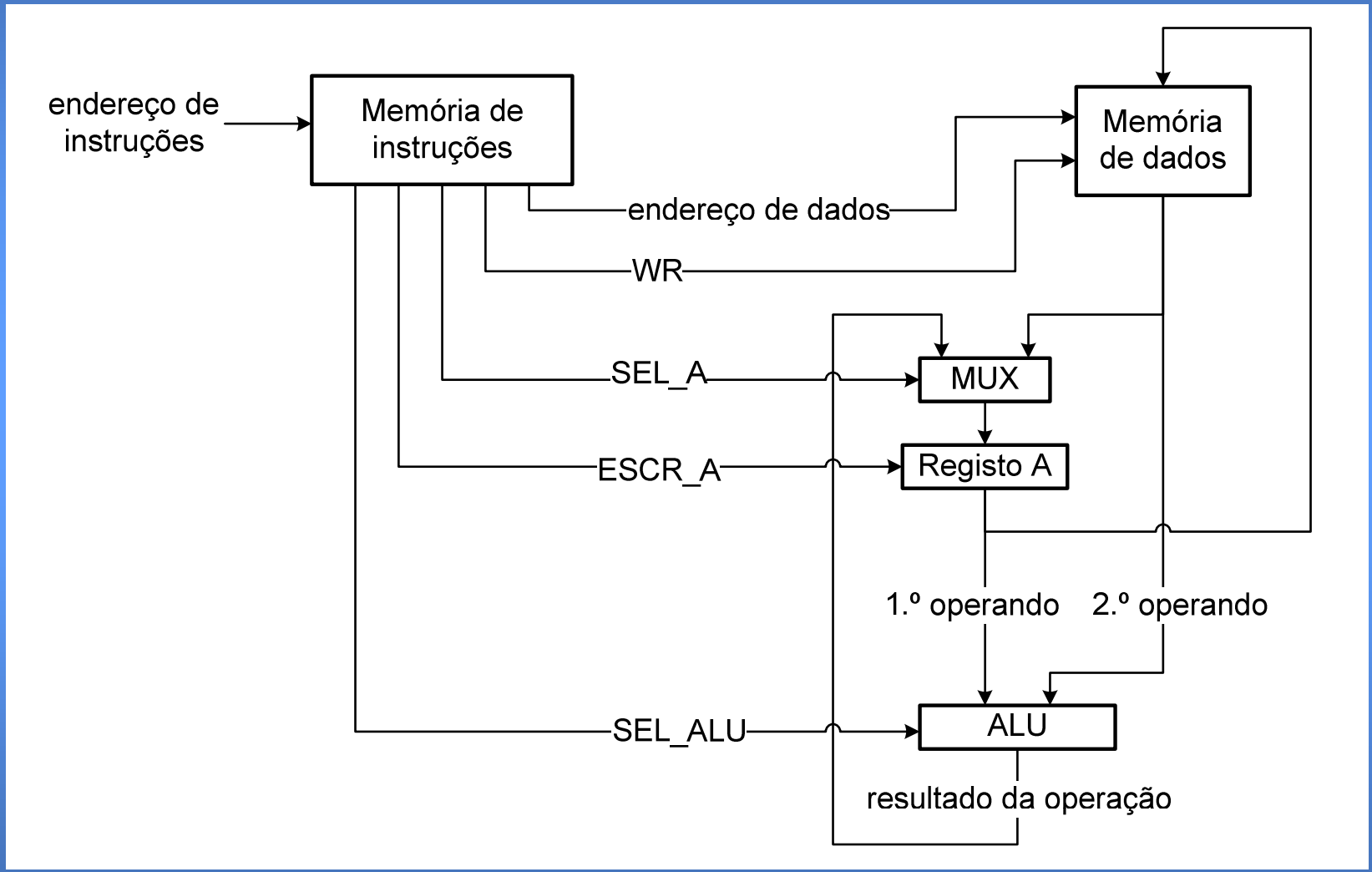
- Problema: Como suportar várias operações?
- Solução: Usar uma ALU (Arithmetic and Logic Unit).
- O sinal SEL_ALU seleciona a operação.



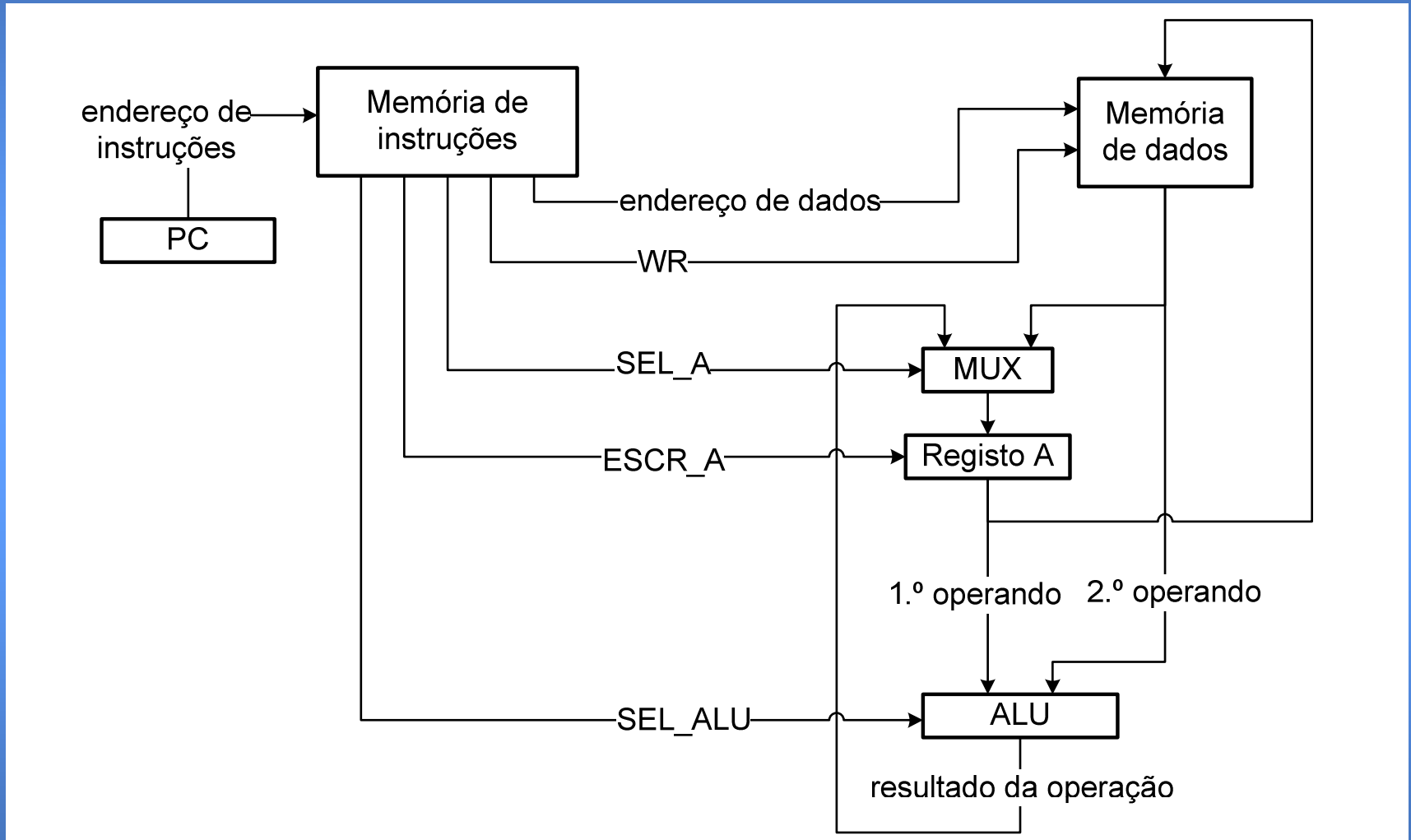
Estrutura da ALU



Instruções com sinais de controlo



Contador de programa (PC)



Vamos fazer um programa!

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

  soma = N + (N-1) + (N-2) + . . . + 2 + 1

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

Variáveis em memória

- **soma** e **temp** 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 (por exemplo, **soma** fica em 40H e **temp** em 41H).

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

Constantes simbólicas

- Endereços só números é muito confuso!
- Constante simbólica = identificador com um valor
- Define-se uma vez e depois pode usar-se como se fosse o número com que ela foi definida.

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

1.  M[soma] ← 0                          (inicializa soma com zero)
2.  M[temp] ← N                          (inicializa temp com N)
3.  Se (M[temp] < 0) salta para 8        (se temp for negativo, salta para o fim)
4.  Se (M[temp] = 0) salta para 8        (se temp for zero, salta para o fim)
5.  M[soma] ← M[soma] + M[temp]          (adiciona temp a soma)
6.  M[temp] ← M[temp] – 1                (decrementa temp)
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 memórias 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)
temp    EQU    41H      (definição do endereço da variável temp)
```

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

Vamos somar!

Número de instruções executadas **20**

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

PC 7 **soma** 6 **ter** 0

```
soma    EQU    40H    (defini                ável soma)
temp    EQU    41H    (d                     iável temp)

⟹ 0    M[soma] ← 0                            ero)
⟹ 1    M[temp] ← N                            )
⟹ 2    Se (M[temp] < 0) P                     salta para o fim)
⟹ 3    Se (M[temp] = 0) P                     ta para o fim)
⟹ 4    M[soma] ← M[soma]                      ma)
⟹ 5    M[temp] ← M[temp] – 1                  p)
⟹ 6    PC ← 3                                 endereço 3)
⟹ 7    PC ← 7                 (fim do programa)
```

Já chega!!!!

Constantes

- Cada instrução pode ter de 1 a 3 constantes.
- Tamanho das instruções:
  - variável (complica o controlo), ou
  - fixo com 3 constantes (desperdício de memória).

Constantes		
soma	0	
temp	N	
temp	0	7
temp	0	7
soma	soma	temp
temp	temp	1
3		
7		

```
0   M[soma] ← 0
1   M[temp] ← N
2   Se (M[temp] < 0) PC ← 7
3   Se (M[temp] = 0) PC ← 7
4   M[soma] ← M[soma] + M[temp]
5   M[temp] ← M[temp] – 1
6   PC ← 3
7   PC ← 7
```

Solução para as constantes

- Decompôr instruções com várias constantes em instruções mais simples, cada uma só com uma constante.
- Já não há endereços e valores numéricos na mesma instrução.
- Usa-se um registo auxiliar (A) para guardar valores entre instruções.

0	M[soma] ← 0	0 1	A ← 0 M[soma] ← A
1	M[temp] ← N	2 3	A ← N M[temp] ← A
2	Se (M[temp] < 0) PC ← 7	4	(A < 0) : PC ← 12
3	Se (M[temp] = 0) PC ← 7	5	(A = 0) : PC ← 12
4	M[soma] ← M[soma] + M[temp]	6 7	A ← A + M[soma] M[soma] ← A
5	M[temp] ← M[temp] – 1	8 9 10	A ← M[temp] A ← A – 1 M[temp] ← A
6	PC ← 3	11	PC ← 5
7	PC ← 7	12	PC ← 12

Que operações são necessárias?

- Para correr o programa anterior, o hardware tem de suportar as seguintes operações:

Operação	Exemplo
Escrever uma constante no registo A	A ← 0
Armazenar o registo A na memória	M[soma] ← A
Ler uma célula de memória e escrever no registo A	A ← M[temp]
Operação com o registo A e uma célula de memória como operandos	A ← A + M[soma]
Operação com o registo A e uma constante como operandos	A ← A – 1
Salto incondicional	PC ← 7
Salto condicional	(A = 0) : PC ← 12

- Esta notação designa-se RTL (*Register Transfer Language*)

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 $\wedge$ *valor*
	AND	[*endereço*]	*AND* (memória)	08H	A ← A $\wedge$ M[*endereço*]
	OR	*valor*	*OR* (imediato)	09H	A ← A $\vee$ *valor*
	OR	[*endereço*]	*OR* (memória)	0AH	A ← A $\vee$ 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[temp] ← A	03H		ST	[temp]
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[temp]	08H		LD	[temp]
9	A ← A – 1	09H		SUB	1
10	M[temp] ← A	0AH		ST	[temp]
11	PC ← 5	0BH		JMP	teste
12	PC ← 12	0CH	fim:	JMP	fim

Vamos "assemblar"!

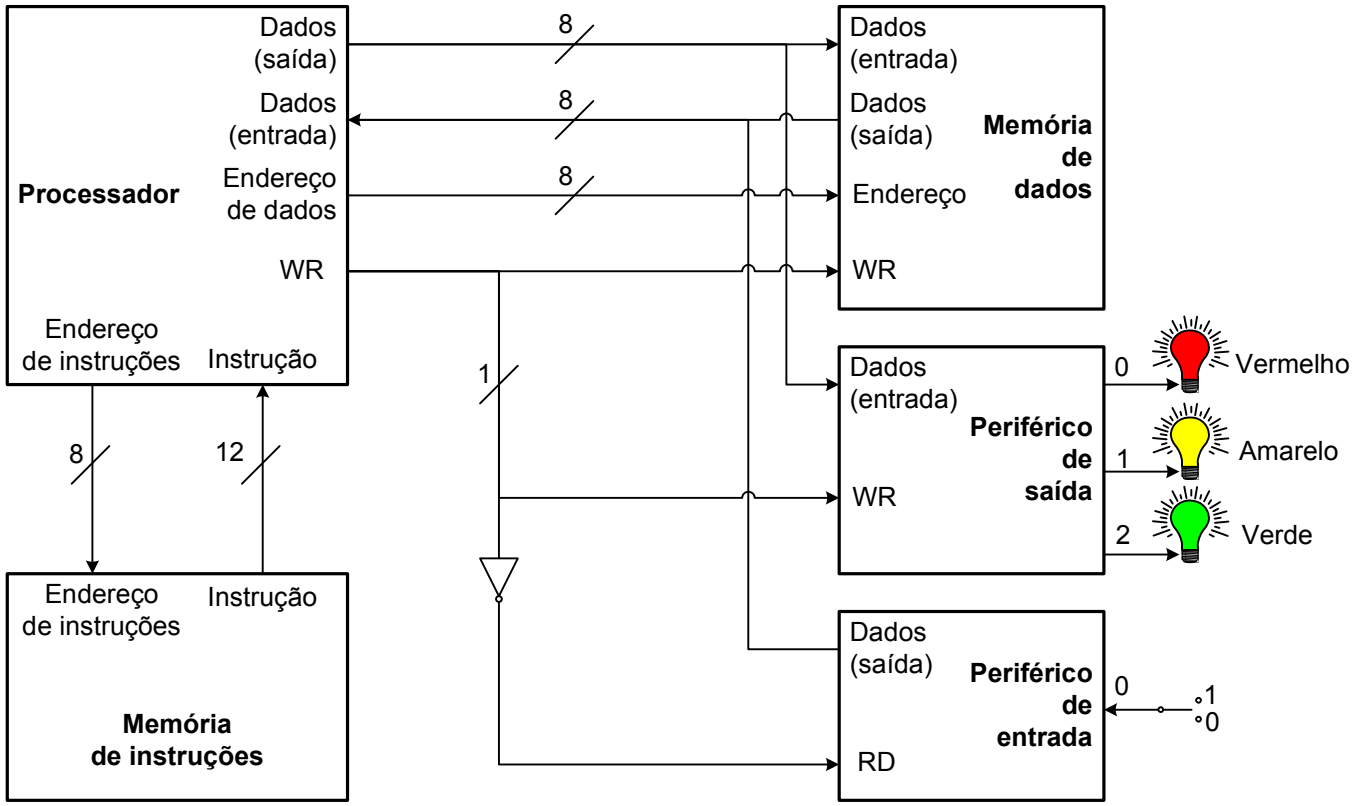
Opcode	*Assembly*	
00H	LD	*valor*
01H	LD	[*endereço*]
02H	ST	[*endereço*]
03H	ADD	*valor*
04H	ADD	[*endereço*]
05H	SUB	*valor*
06H	SUB	[*endereço*]
07H	AND	*valor*
08H	AND	[*er*
09H	OR	*va*
0AH	OR	[*er*
0BH	JMP	*en*
0CH	JZ	*en*
0DH	JN	*en*
0EH	NOP	

- As mne
  em opc
- As cons
  converti

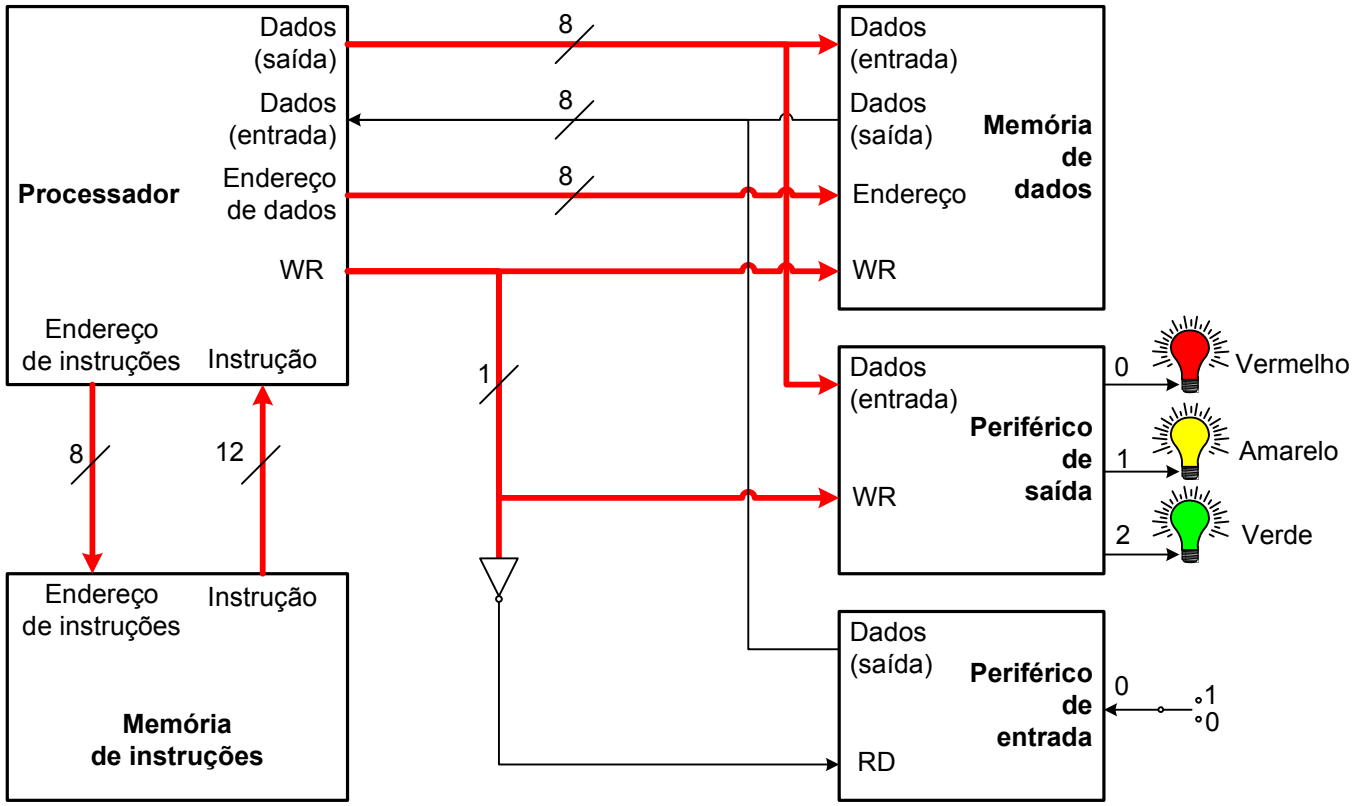
End.	*Assembly*		**Máquina**
00H	início: LD	0	00 00H
01H	ST	[soma]	02 40H
02H	LD	N	00 08H
03H	ST	[temp]	02 41H
04H	JN	fim	0D 0CH
05H	teste: JZ	fim	0C 0CH
06H	ADD	[soma]	04 40H
07H	ST	[soma]	02 40H
08H	LD	[temp]	01 41H
09H	SUB	1	05 01H
0AH	ST	[temp]	02 41H
0BH	JMP	teste	0B 05H
0CH	fim: JMP	fim	0B 0CH

- Assumindo: **soma** = 40H; **temp** = 41H; **N** = 8

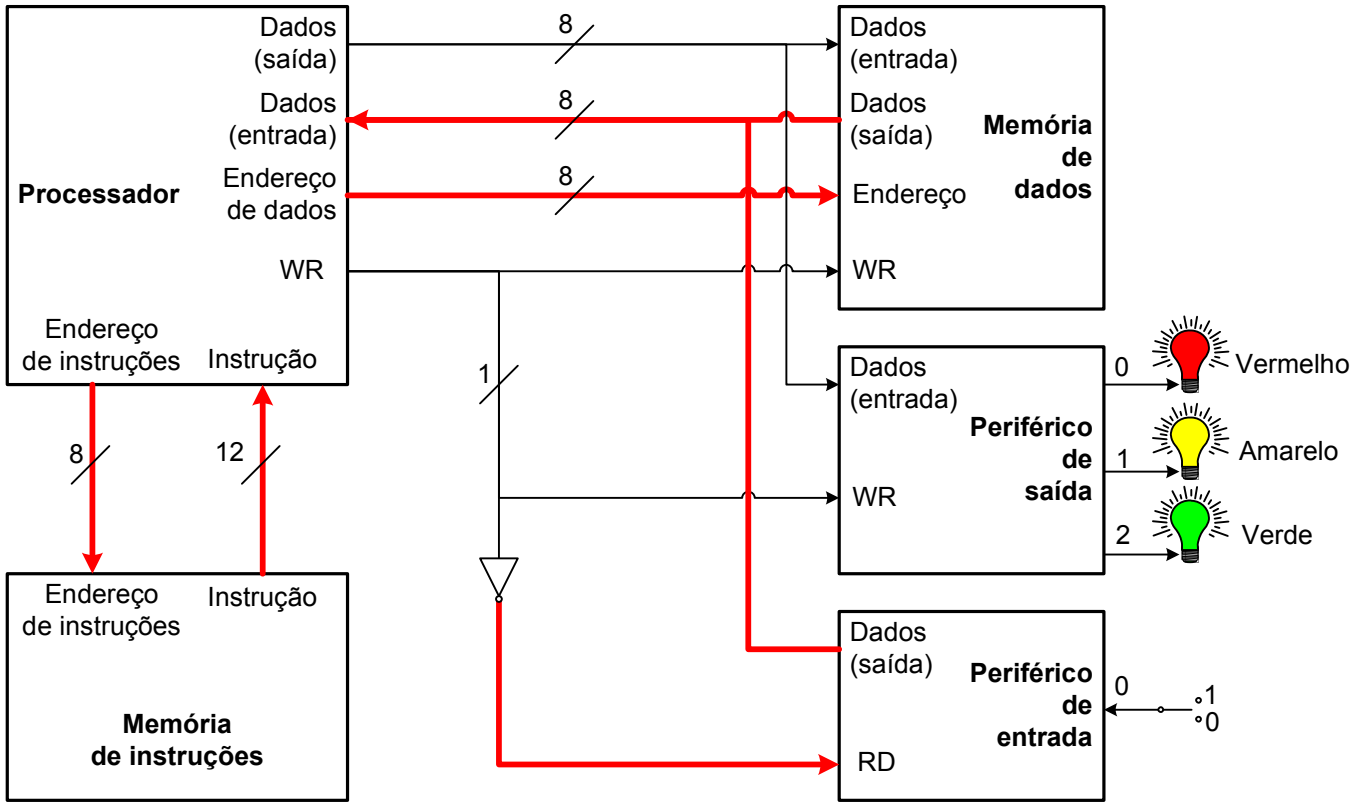
Além da memória: periféricos



Problema na escrita

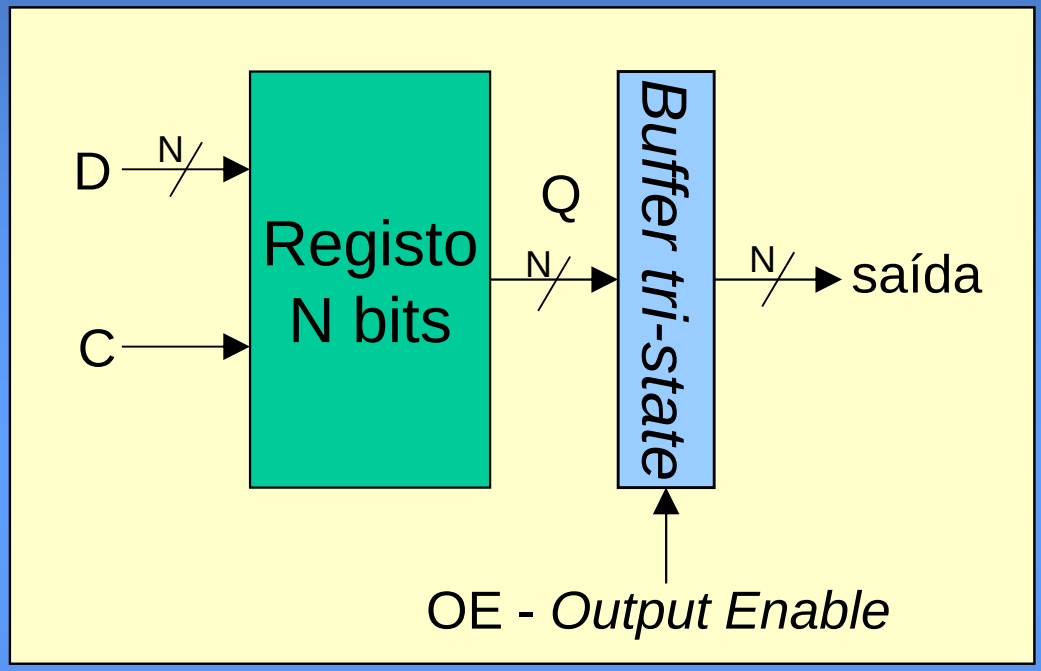


Problema na leitura



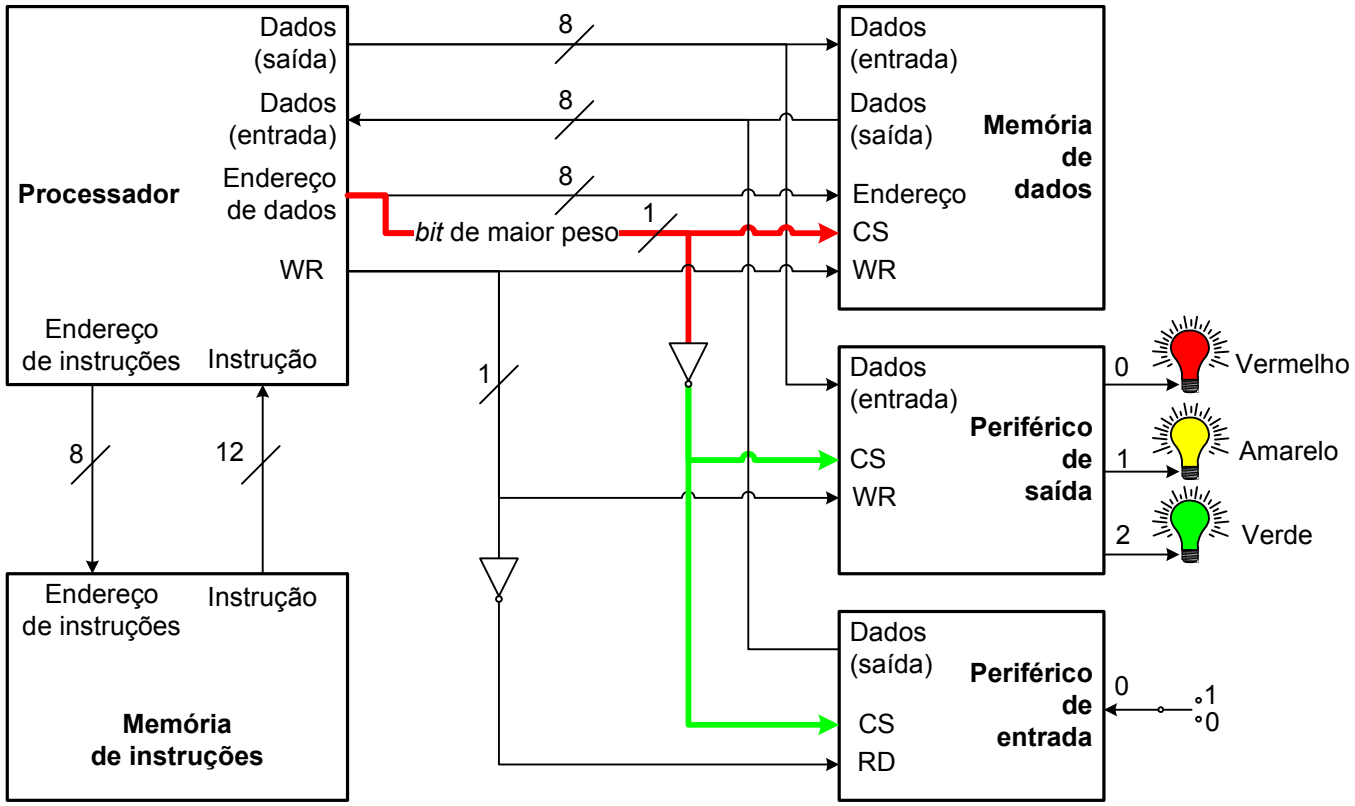
Registos com saída *tri-state*

- Saídas normais ligadas entre si podem dar conflito.
- Com saída em *tri-state*, podem ligar-se várias saídas entre si (mas só uma pode ter o *buffer tri-state* ligado!)



Output Enable	Saída
Activo	Q
Inactivo	Desligada

Solução: acesso selectivo



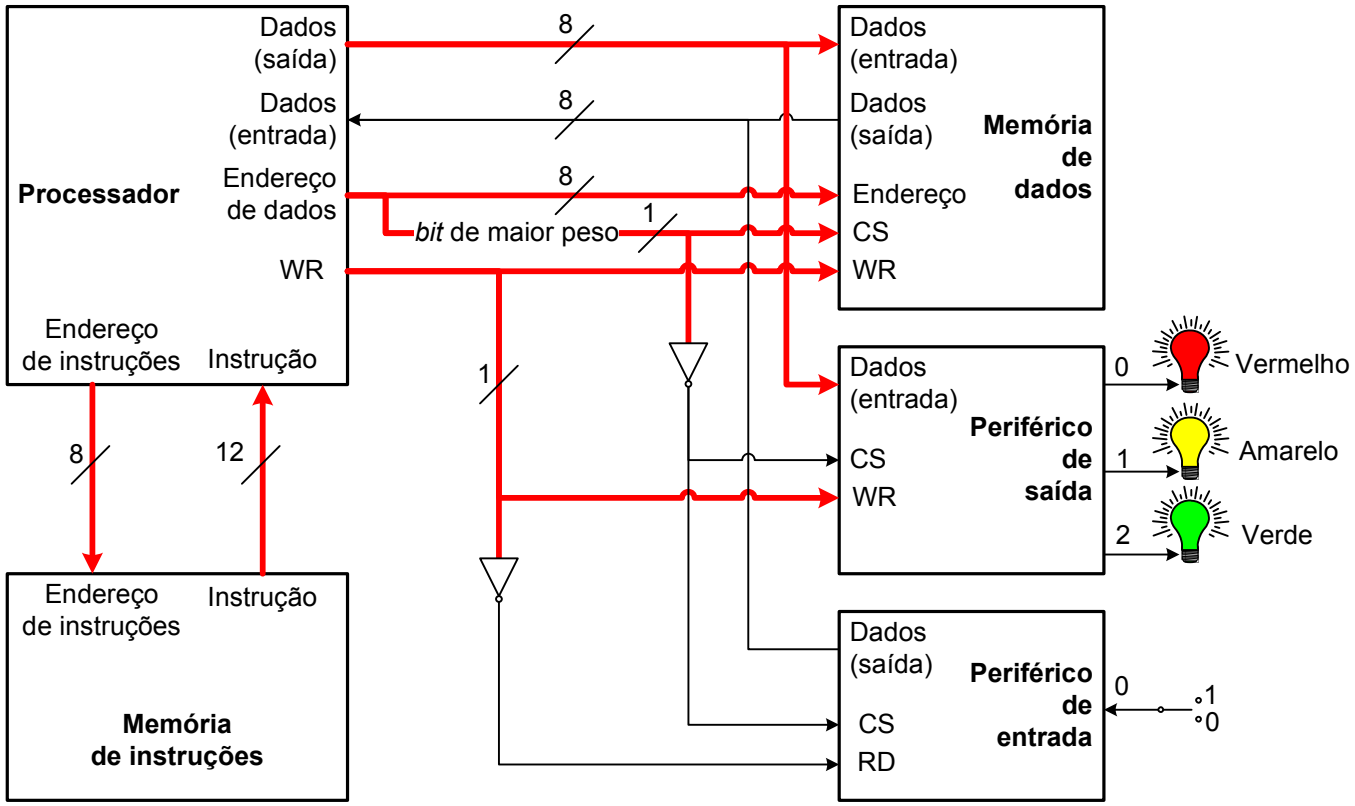
Acesso à memória/periféricos

Sinais			WR	
			Inativo (1) - leitura	Ativo (0) - escrita
CS	Inativo (1)		Sem efeito no dispositivo	
	Ativo (0)	Memória	Lê dados	Escreve dados
		Periférico de entrada	Lê dados	Sem efeito
		Periférico de saída	Sem efeito	Escreve dados

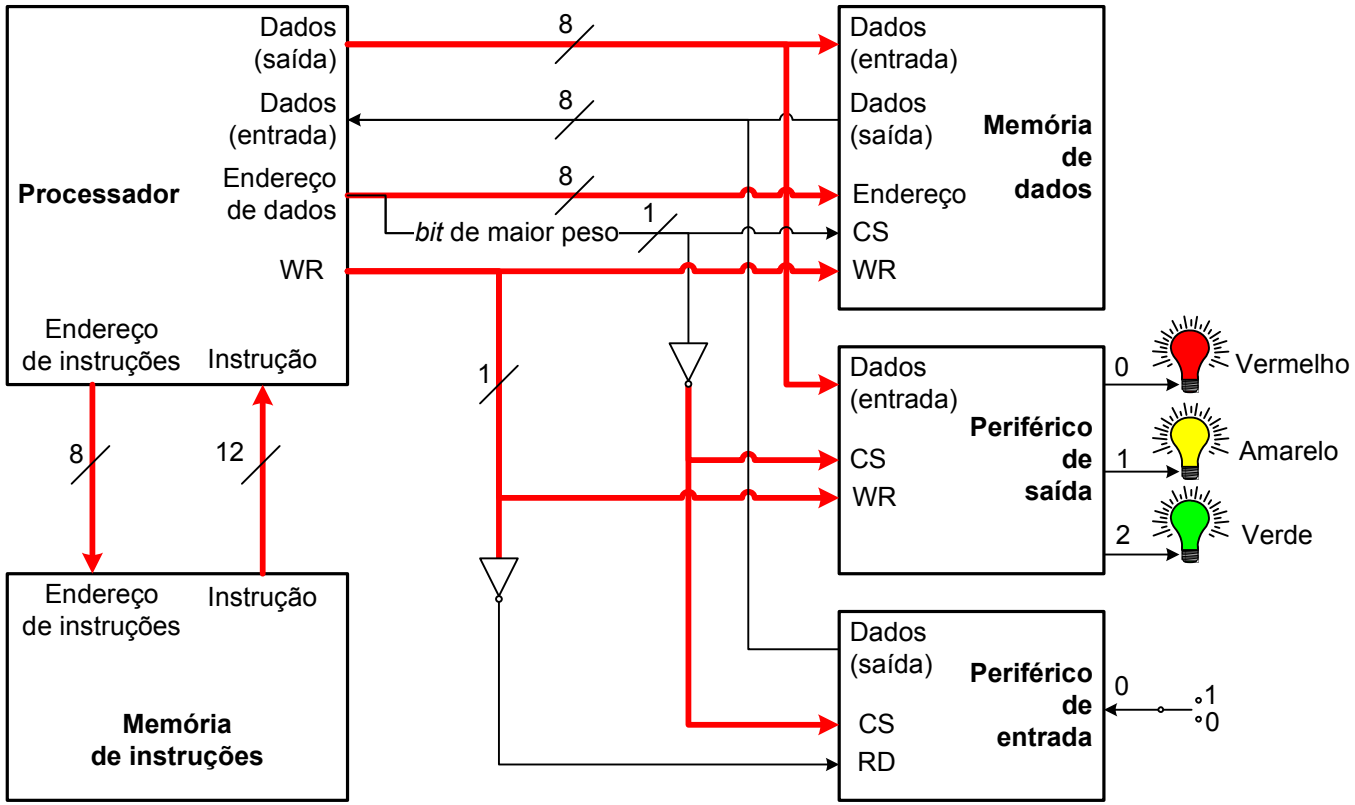
Mapa de endereços

Dispositivo	**Gama de endereços em <u>escrita</u> (**WR **ativo)**	**Gama de endereços em <u>leitura</u> (**WR **inativo)**
Memória de dados	0 – 127 (00H – 7FH)	0 – 127 (00H – 7FH)
Periférico de saída	128 – 255 (80H – FFH)	Nenhum (o sinal **WR** do periférico está inativo)
Periférico de entrada	Nenhum (o sinal **RD** do periférico está inativo)	128 – 255 (80H – FFH)

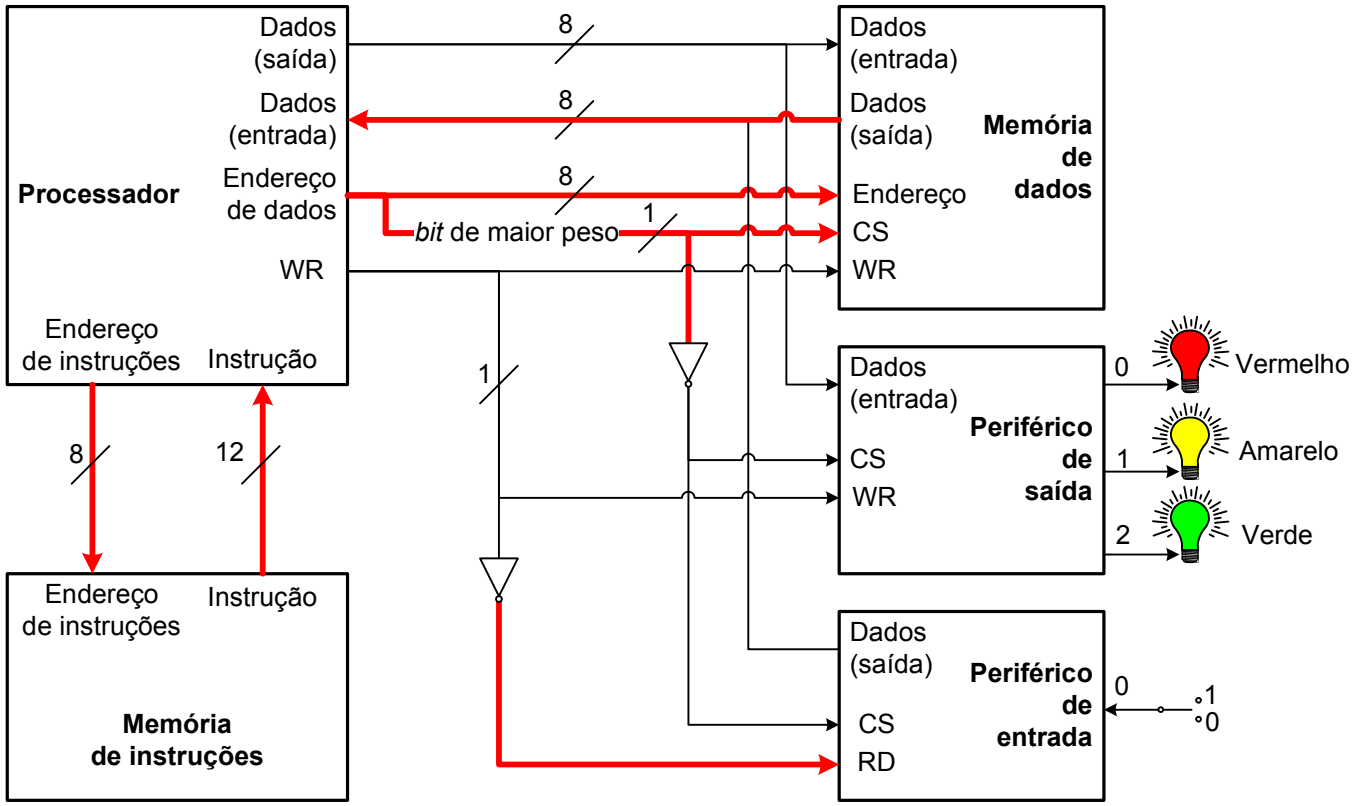
Escrita na memória de dados



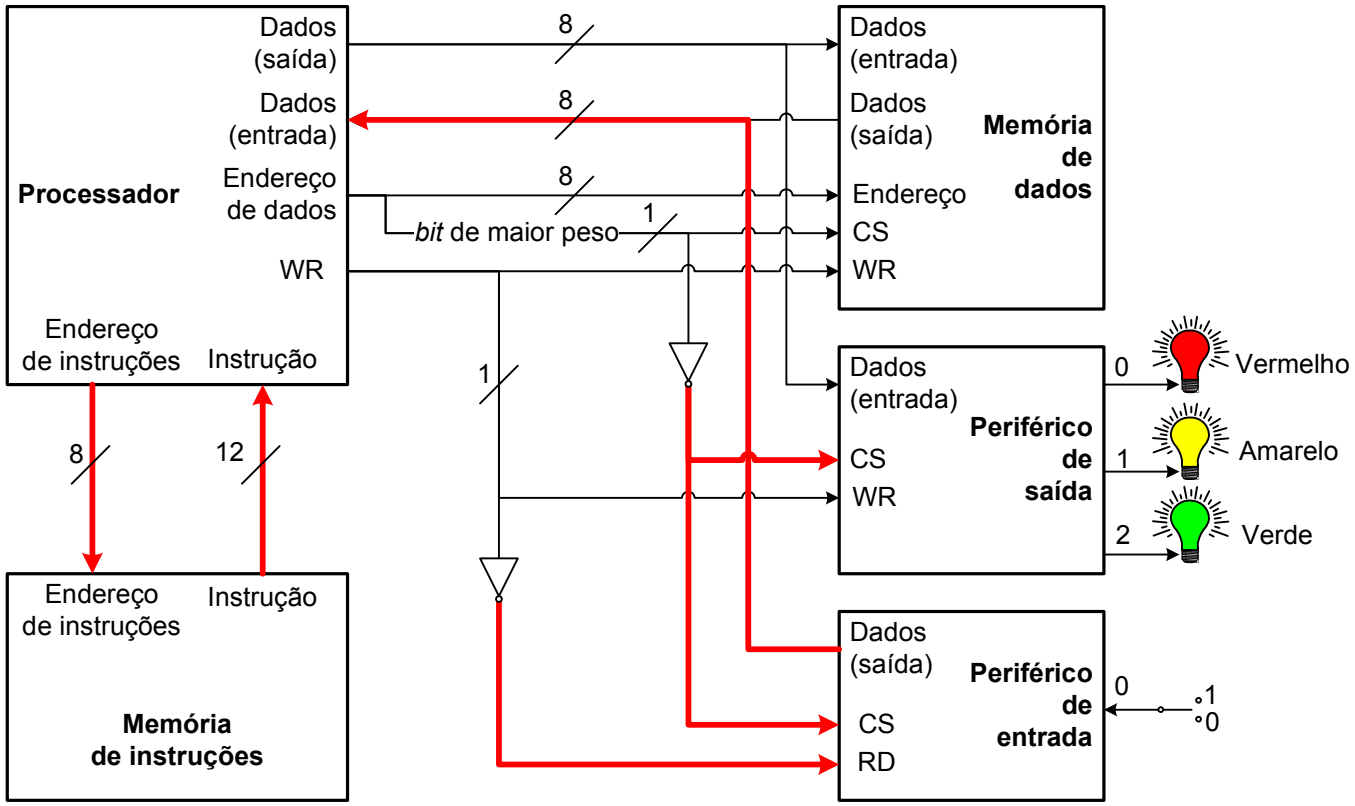
Escrita no periférico de saída



Leitura da memória de dados



Leitura do periférico de entrada

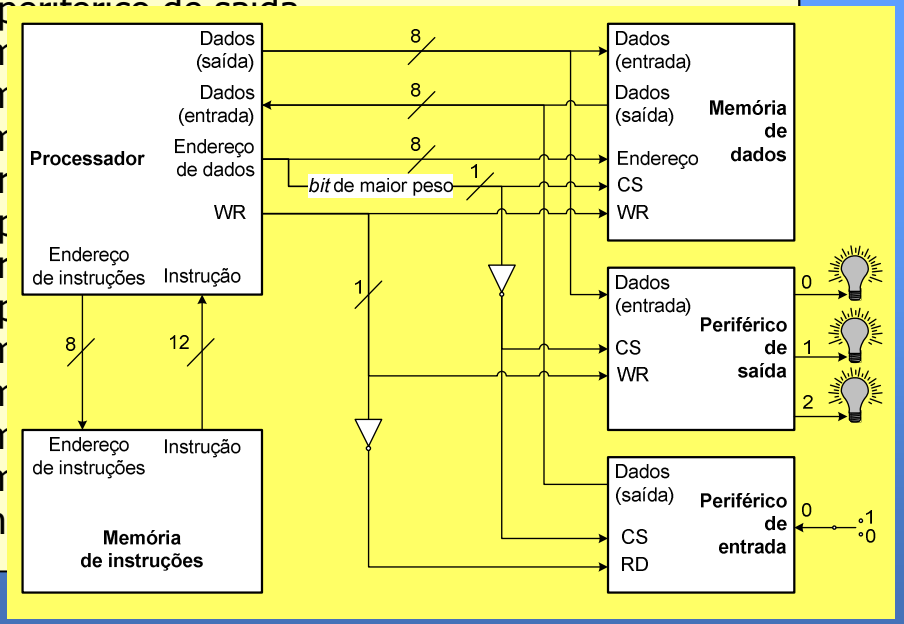


Programa: semáforo simples

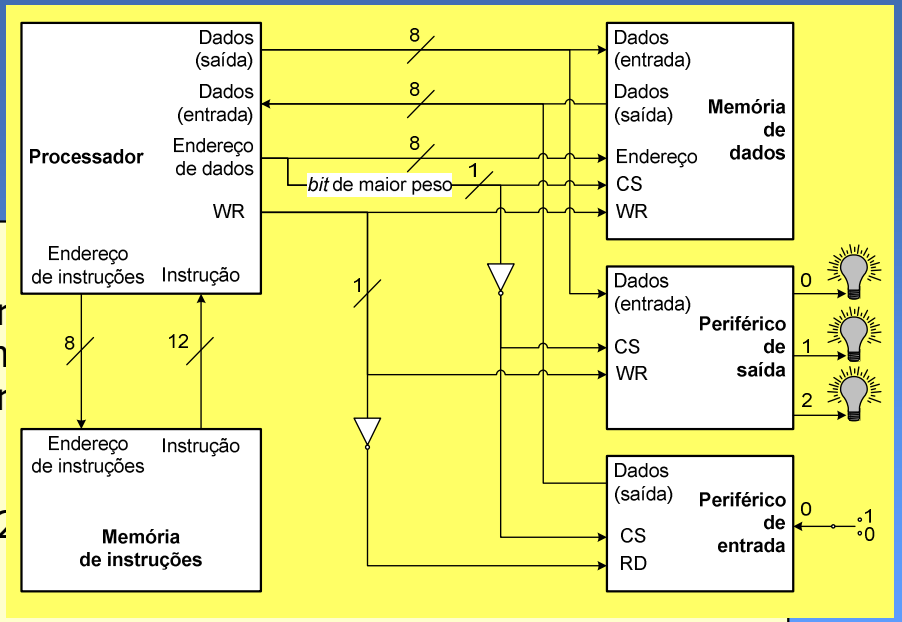
```
; constantes de dados
vermelho    EQU   01H          ; valor do vermelho (lâmpada liga ao bit 0)
amarelo     EQU   02H          ; valor do amarelo (lâmpada liga ao bit 1)
verde       EQU   04H          ; valor do verde (lâmpada liga ao bit 2)

; 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 p
semVerde:   NOP                ; faz um com
            NOP                ; faz um com
            NOP                ; faz um com
            LD    amarelo      ; Carrega o r
            ST    [semáforo]   ; Atualiza o p
semAmarelo: LD    vermelho     ; Carrega o r
            ST    [semáforo]   ; Atualiza o p
semVerm:    NOP                ; faz um com
            NOP                ; faz um com
            NOP                ; faz um com
            NOP                ; faz um com
            JMP   início       ; vai fazer m
```



Programa (cont.)

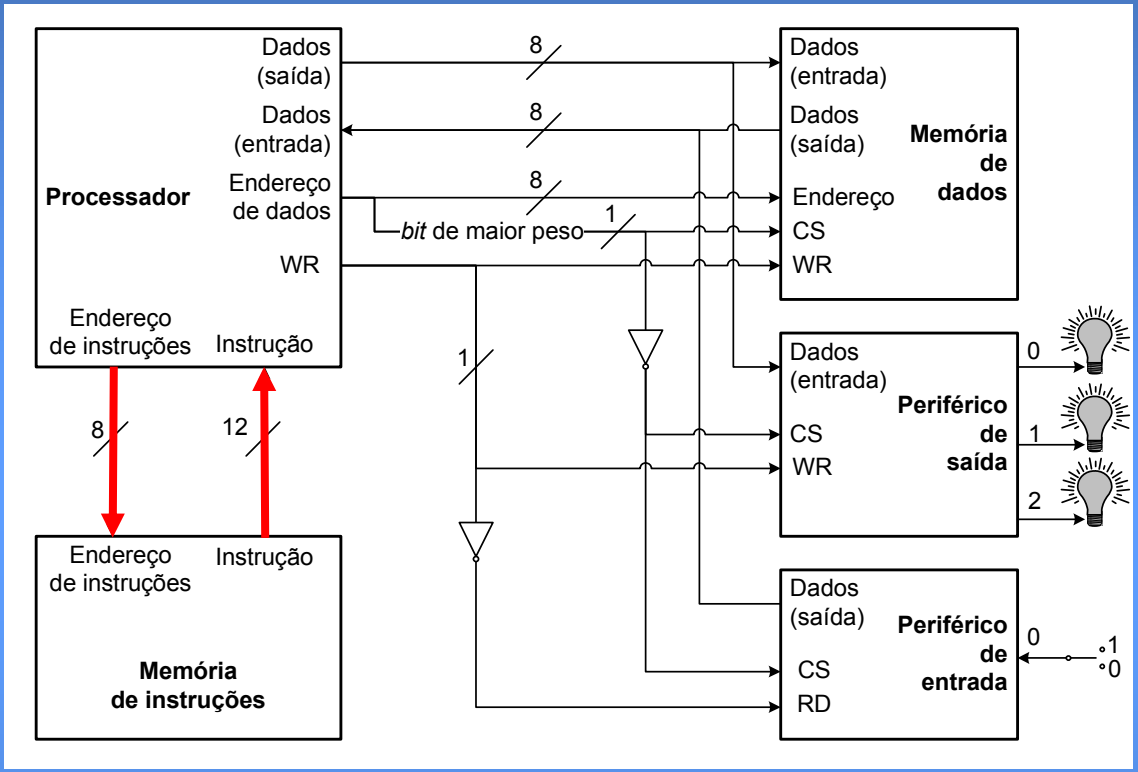



```
; constantes de dados
vermelho    EQU   01H          ; valor do ver
amarelo     EQU   02H          ; valor do am
verde       EQU   04H          ; valor do ver

; constantes de endereços
semáforo    EQU   80H          ; endereço 12

; 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
```

Vamos correr o programa!



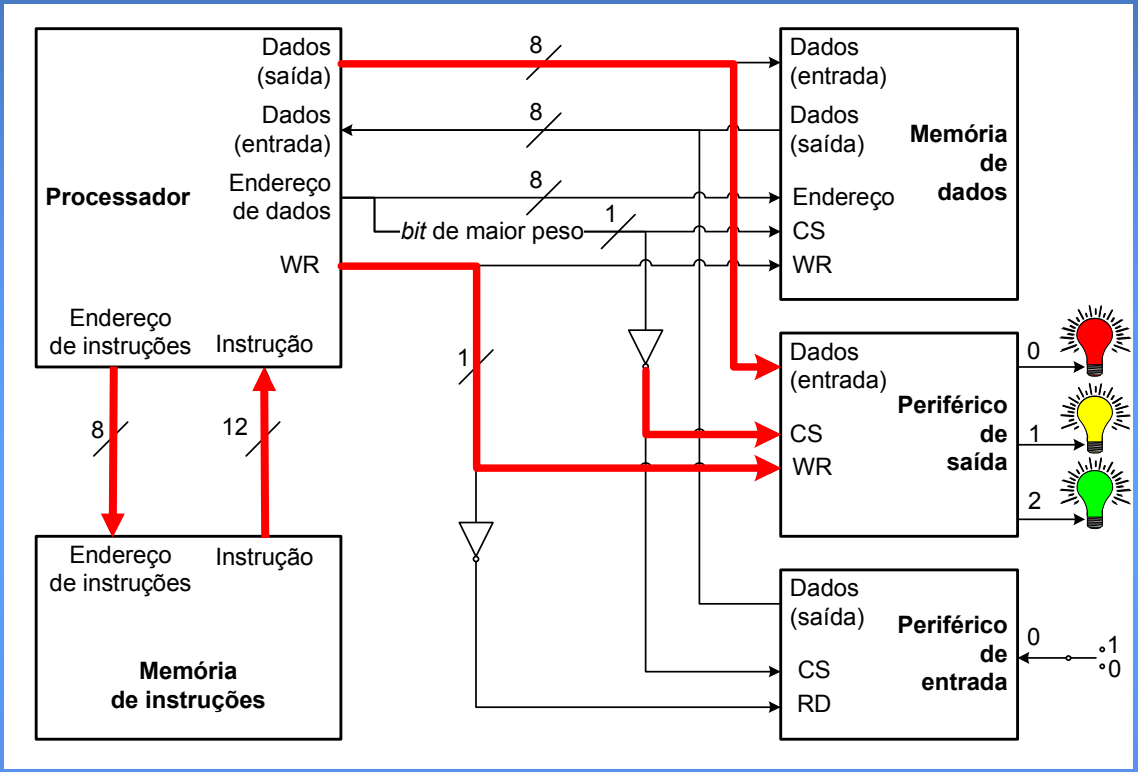

```
início:     LD      verde
            ST      [semáforo]
semVerde:   NOP
            NOP
            NOP
            LD      amarelo
            ST      [semáforo]
semAmarelo: LD      vermelho
            ST      [semáforo]
semVerm:    NOP
            NOP
            NOP
            NOP
            JMP     início
```

 Passo a passo

 Contínuo

 Seguinte

Execução passo a passo

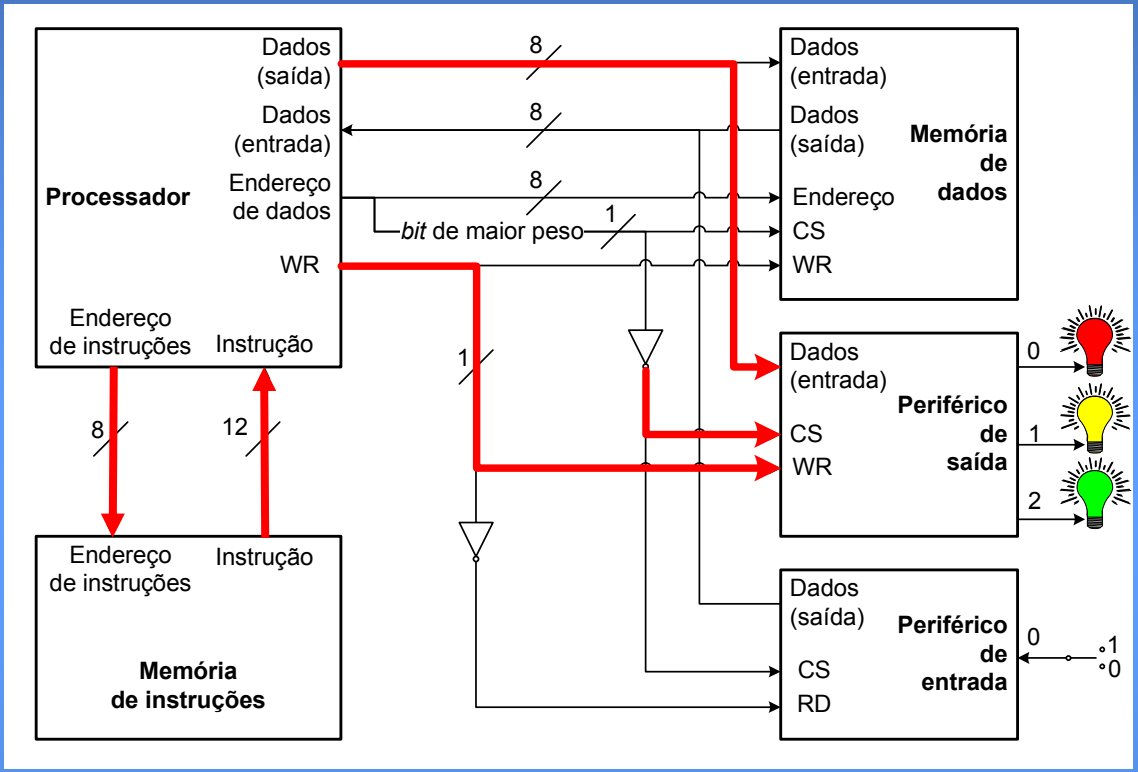



```
início:       LD    verde
              ST    [semáforo]
semVerde:     NOP
              NOP
              NOP
              LD    amarelo
              ST    [semáforo]
semAmarelo:   LD    vermelho
              ST    [semáforo]
semVerm:      NOP
              NOP
              NOP
              NOP
              JMP   início
```

Terminar

Executar uma instrução

Execução contínua




```
início:       LD    verde
              ST    [semáforo]
semVerde:     NOP
              NOP
              NOP
              LD    amarelo
              ST    [semáforo]
semAmarelo:   LD    vermelho
              ST    [semáforo]
semVerm:      NOP
              NOP
              NOP
              NOP
              JMP   início
```

Relógio de 1Hz

Terminar

Outro exemplo: contar bits a 1 em 76H

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)

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

Programa para contar bits a 1

```
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
```

Definições

```
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
```

Inicializa "contador"

```
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
```

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)

Inicializa "máscara"

```
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
```

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)

AND de "máscara" e "valor"

```
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
```

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)

Incrementa "contador"

```
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
```

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)

Testa "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)

```
                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
```

Duplica "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)

```
                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
```

Salta para "teste"

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)

```
                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
```

Salta para "fim"

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)**

```
                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
```

Um só registo → muitas instruções

```
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 b   o (máscara final)

contador        EQU   00H       ; Endereço da célu   que guarda
                                ; o valor
máscara         EQU   01H       ; E
                                ;
início:         LD    0
                ST    [contador]
                LD    máscaraInicial
                ST    [máscara]
teste:          AND   valor
                JZ    próximo
                LD    [contador]
                ADD   1
                ST    [contador]
próximo:        LD    [máscara]
                SUB   máscaraFinal
                JZ    fim
                LD    [máscara]
                ADD   [máscara]
                ST    [máscara]
                JMP   teste
fim:            JMP   fim
```

Isto é só para

Tudo na memória! Mas tudo passa pelo registo!

variáveis na memória!

Se houvesse 3 registos…

```
valor               EQU    76H        ; Valor cujo n
máscaraInicial      EQU    01H        ; 0000 000
máscaraFinal        EQU    80H        ; 1000 00

; Utilização dos registos:
; A – valores intermédios
; B – valor atual do contador de bits a 1
; C – valor atual da máscara

contador    EQU    00H

início:     MOV    B, 0               ; I
            MOV    C, máscaraInici
            MOV    A, C
teste:      AND    A, valor
            JZ     próximo            ; Se o bit for ze
            ADD    B, 1               ; O bit é 1
próximo:    CMP    C, máscaraFi
            JZ     acaba              ; Se forem ig
            ADD    C, C               ; Soma máscar
            JMP    teste              ; Vai fazer mais
acaba:      MOV    [contador], B      ; Se for necess
                                      ; de bits a 1 nu
fim:        JMP    fim                ; Fim do progra
```

MOV de

tem

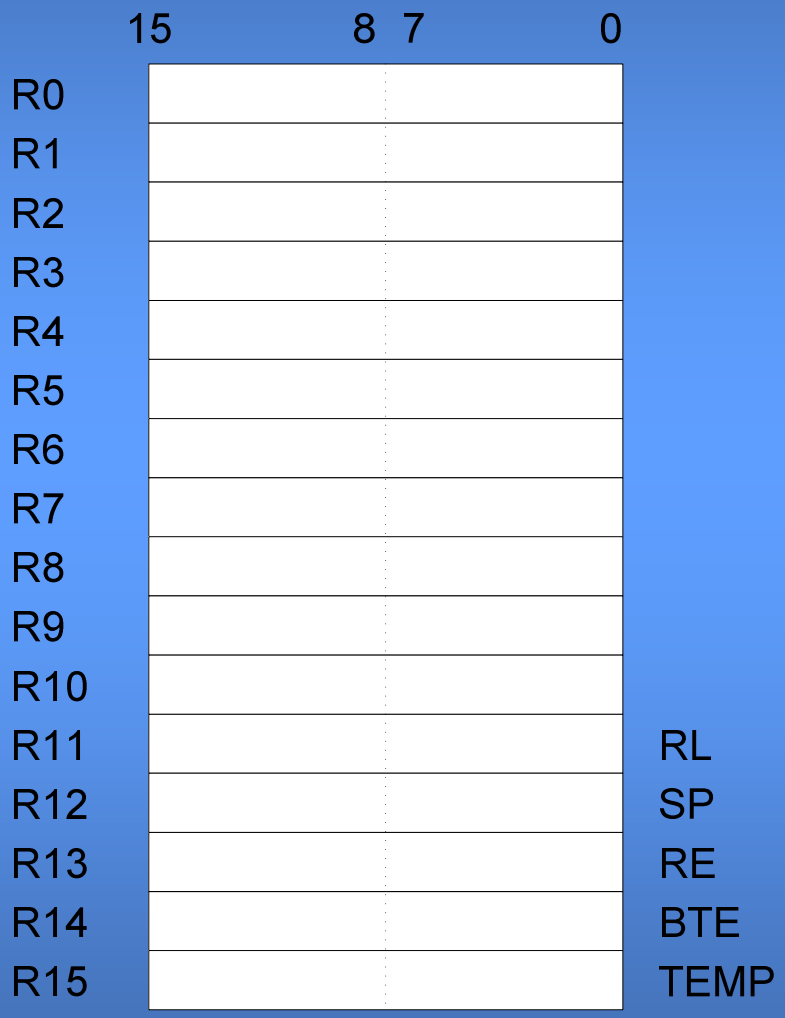
os

te)

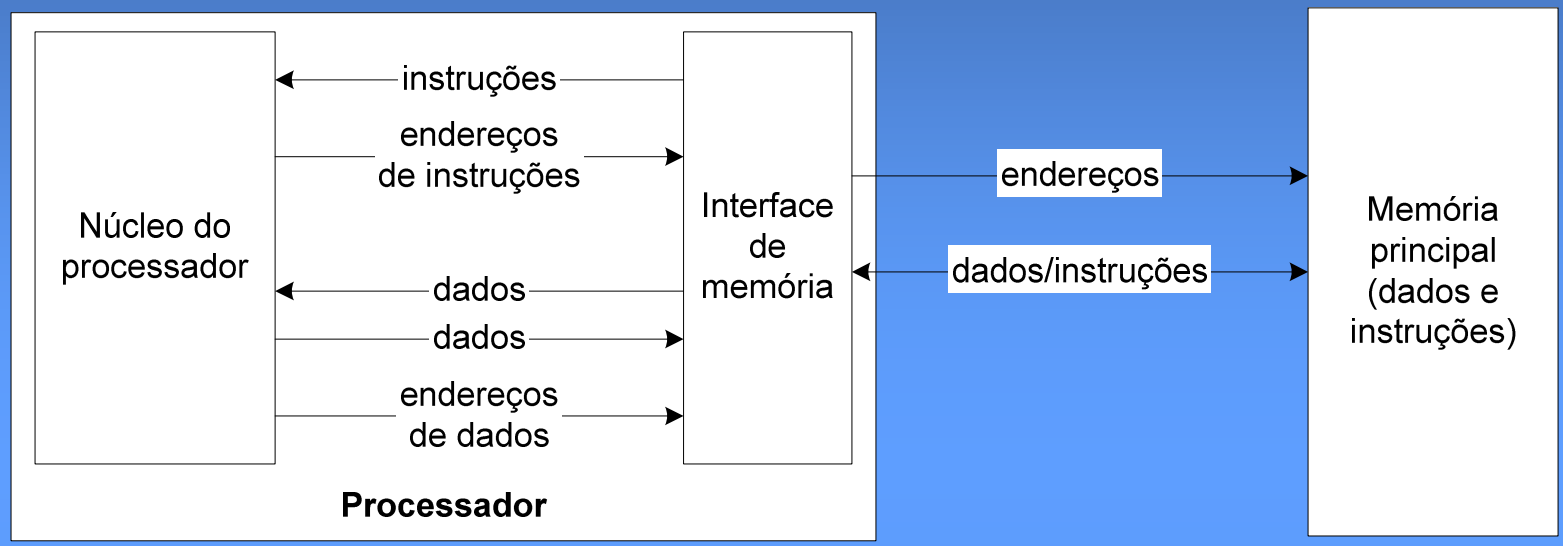
Incrementar a variável é só assim!

Subtrai os operandos, mas não armazena o resultado (atualiza bits de estado)

Banco de registos do PEPE-16

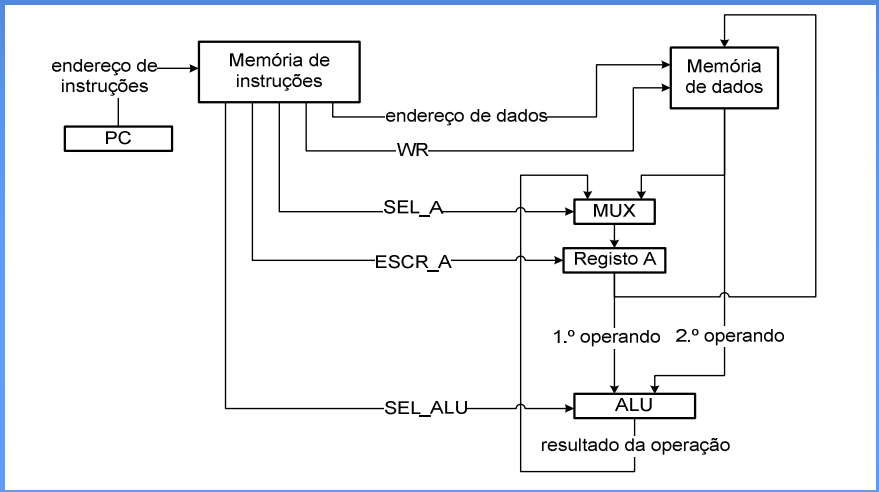


A ilusão de memórias separadas



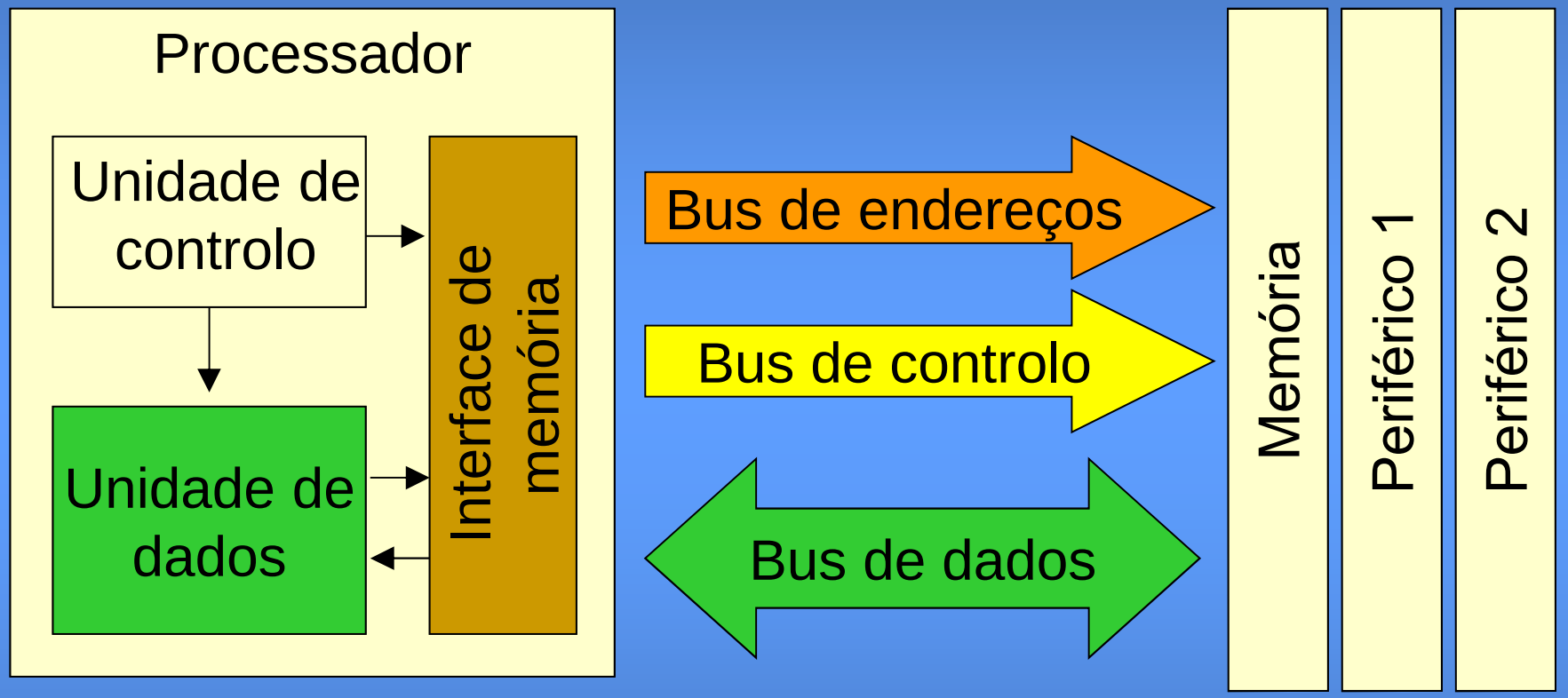
- As memórias comerciais só têm uma ligação de dados bidireccional (para poupar pinos e reduzir o custo)
- O núcleo do processador "vê" quatro ligações unidirecionais.
- A interface de memória arbitra e coordena os acessos.

PEPE-8: processador de instruções uniciclo



- O PEPE-8 executa as instruções num só ciclo de relógio, mas exige ligações à memória separadas.
- Parece rápido, mas não é bem assim (a frequência de relógio é limitada pela operação mais lenta!). A sua grande vantagem é ser simples.

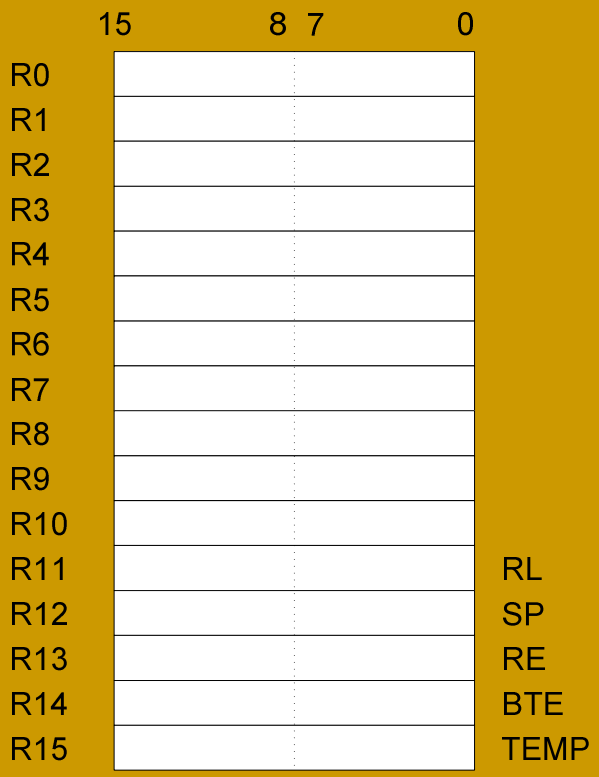
PEPE-16: processador de instruções multi-ciclo



- O PEPE-16 já pode usar as memórias comerciais.
- Cada instrução tem de demorar vários ciclos de relógio (de frequência mais elevada).

um processador

Função	Saída (F)
000	A
001	A + B
010	A - B
011	A + B + carry



áquina de estados

sto

Circuito combinatório

entradas

saídas

Bus end.

dados

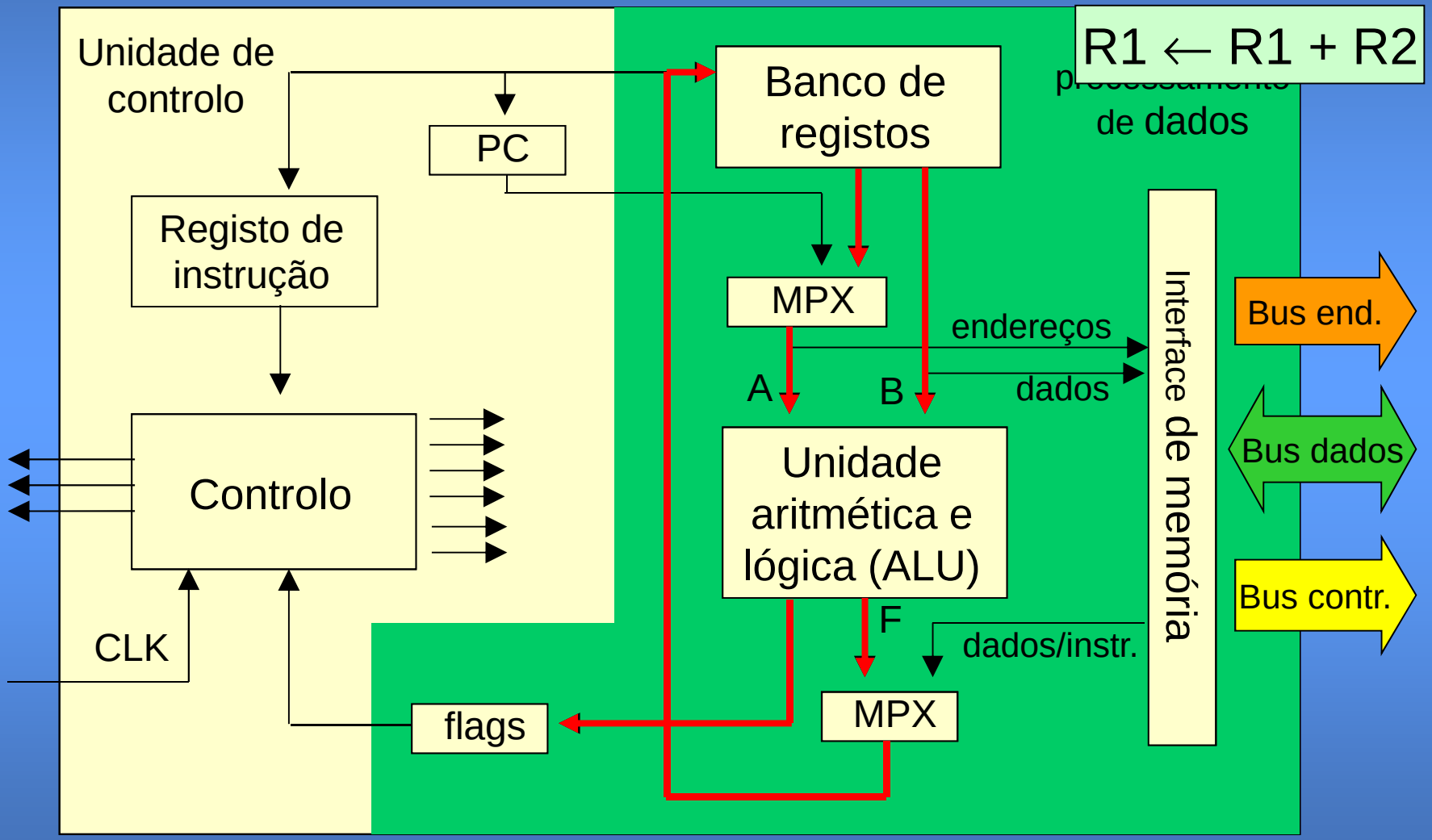
Bus dados

Bus contr.

dados/instr.

erface de memória

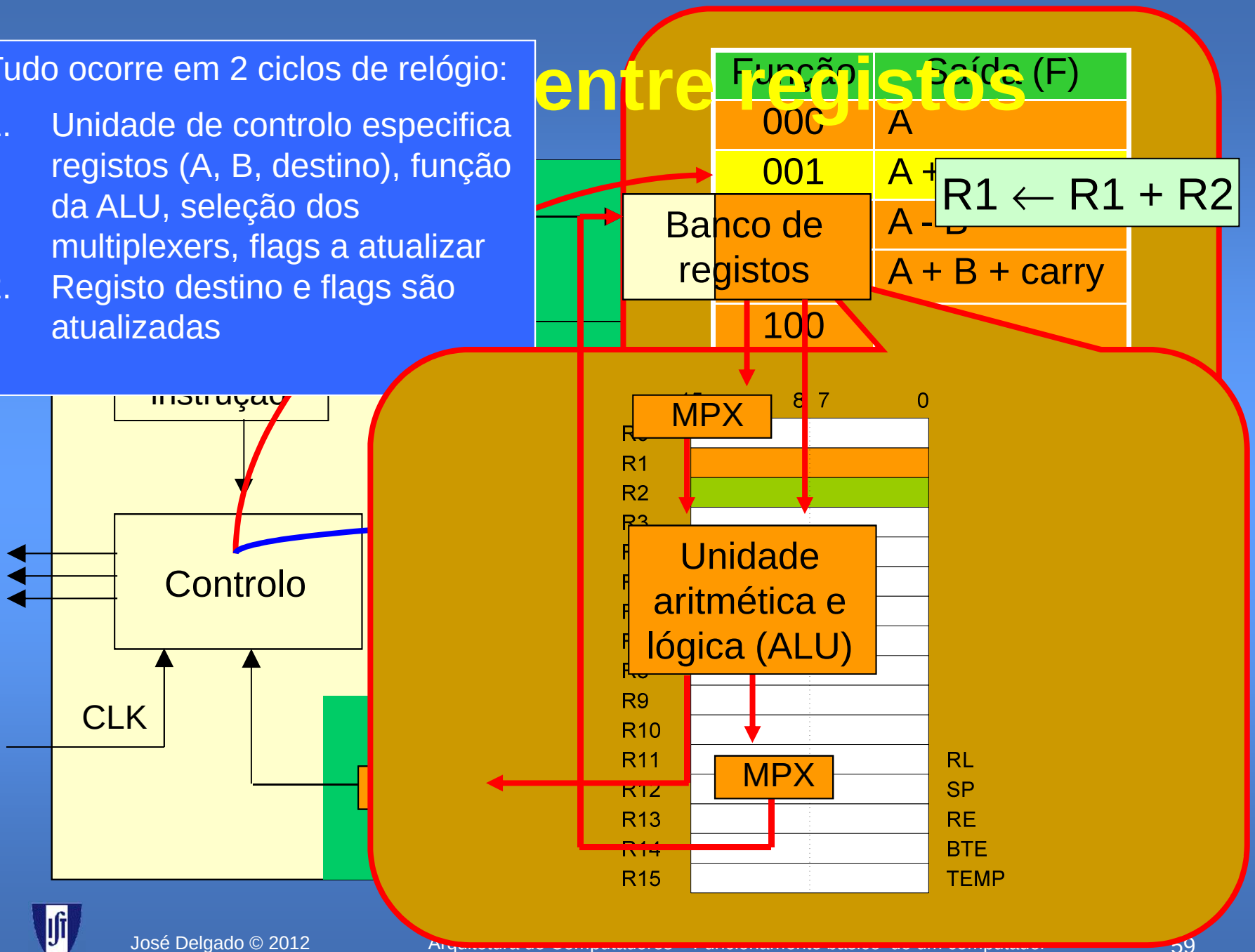
Operação entre registos



entre registos

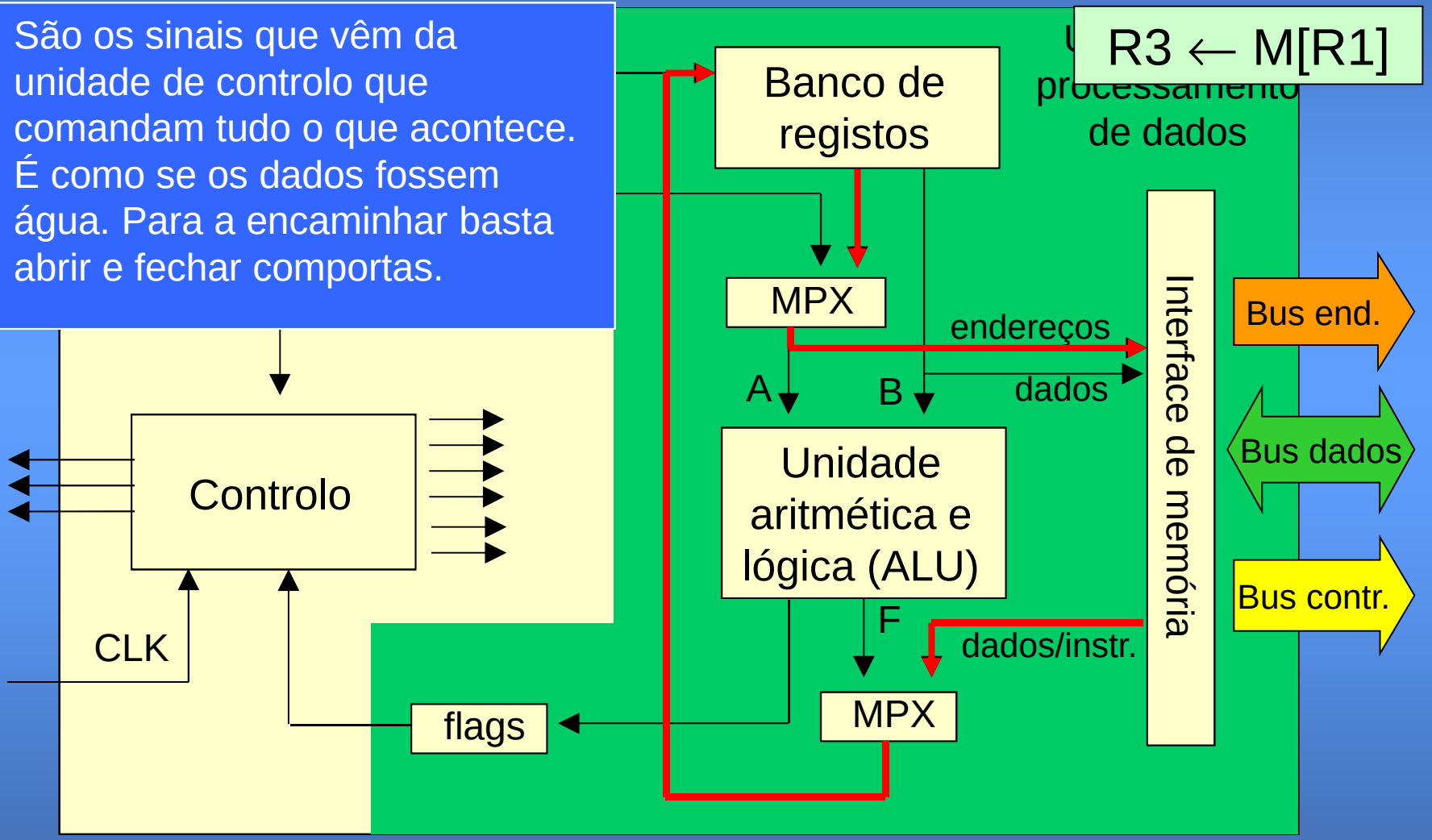
Tudo ocorre em 2 ciclos de relógio:

1. Unidade de controlo especifica registos (A, B, destino), função da ALU, seleção dos multiplexers, flags a atualizar
2. Registo destino e flags são atualizadas



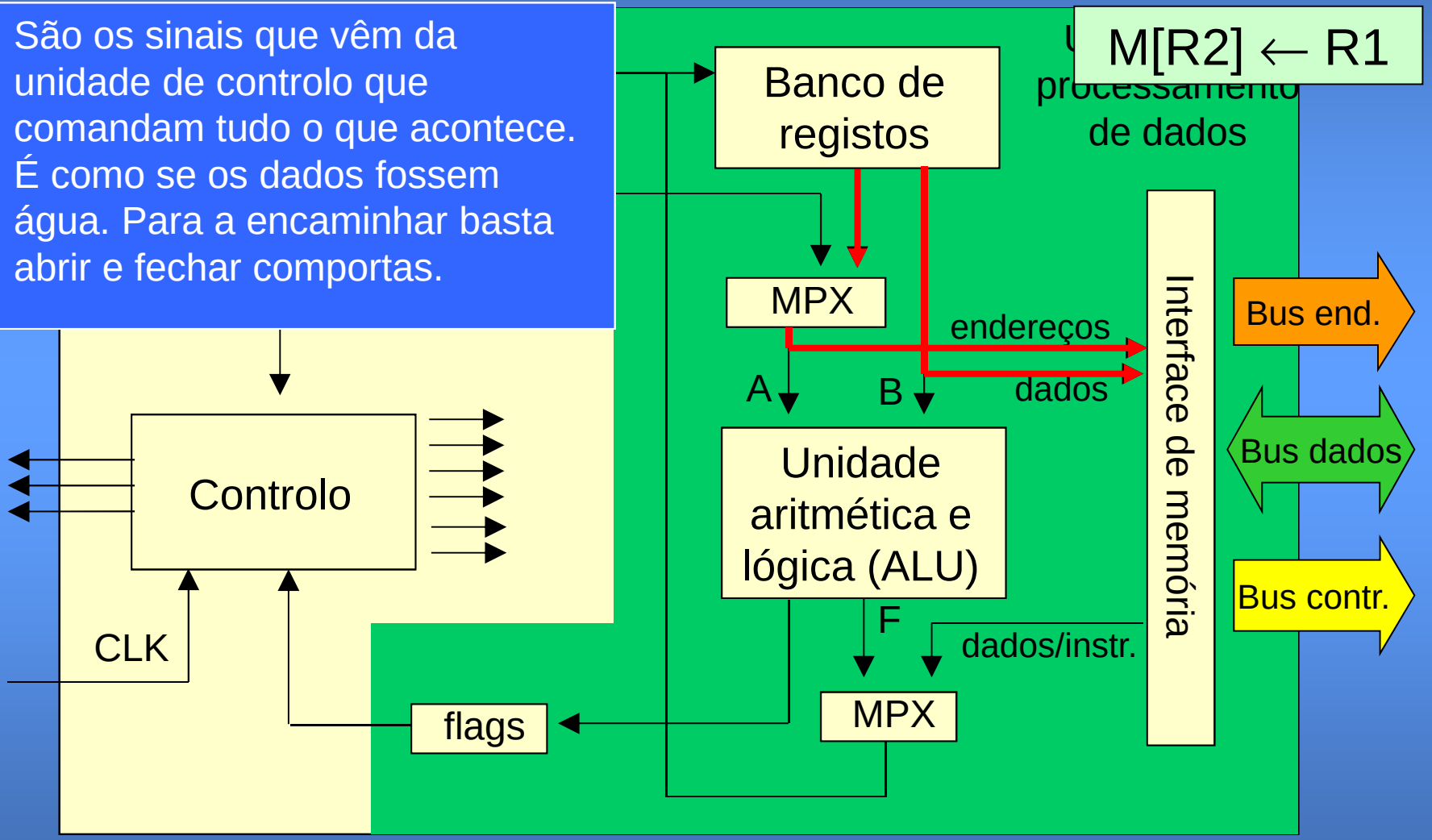
Leitura da memória

- São os sinais que vêm da unidade de controlo que comandam tudo o que acontece.
- É como se os dados fossem água. Para a encaminhar basta abrir e fechar comportas.

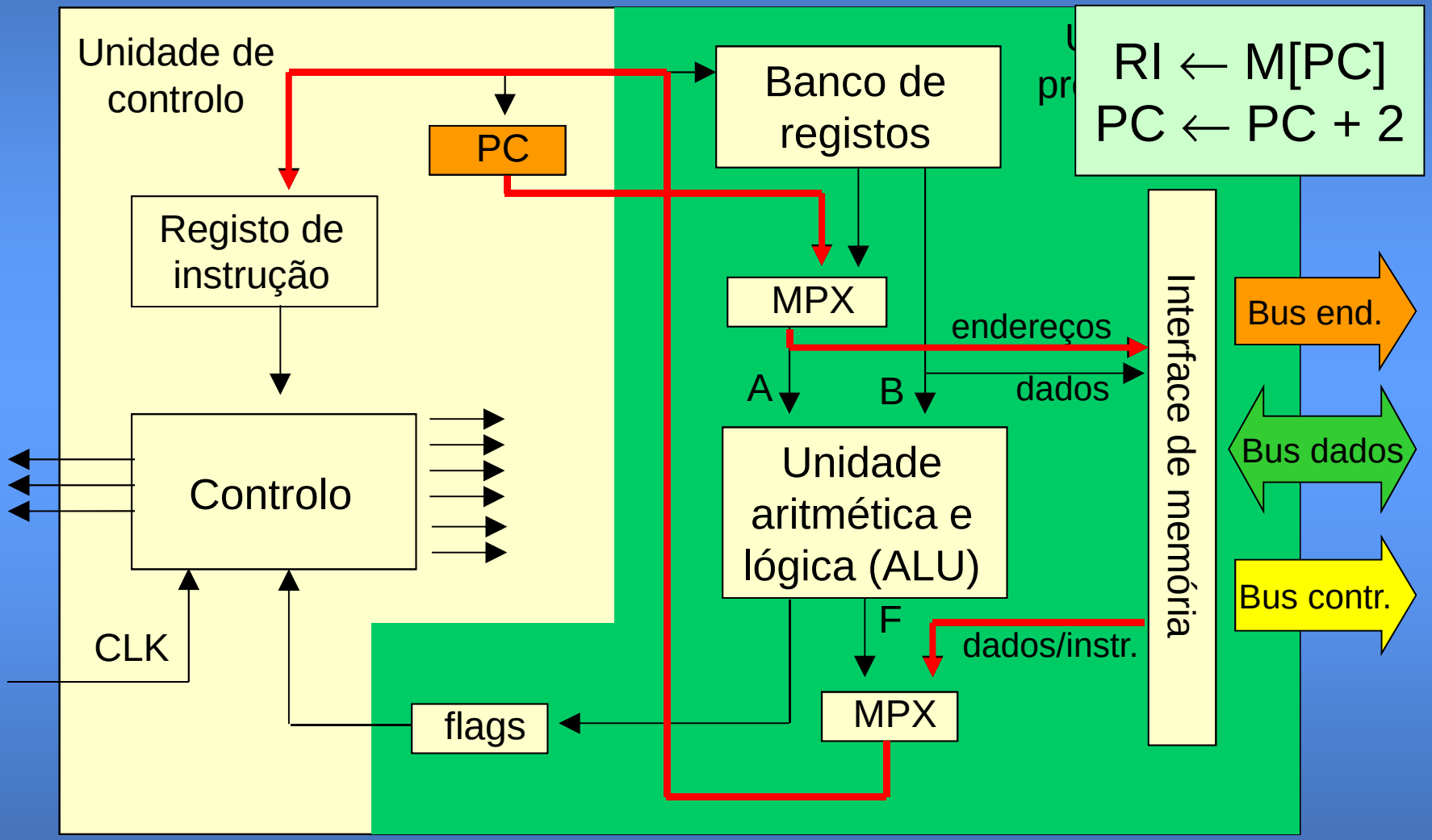


Escrita na memória

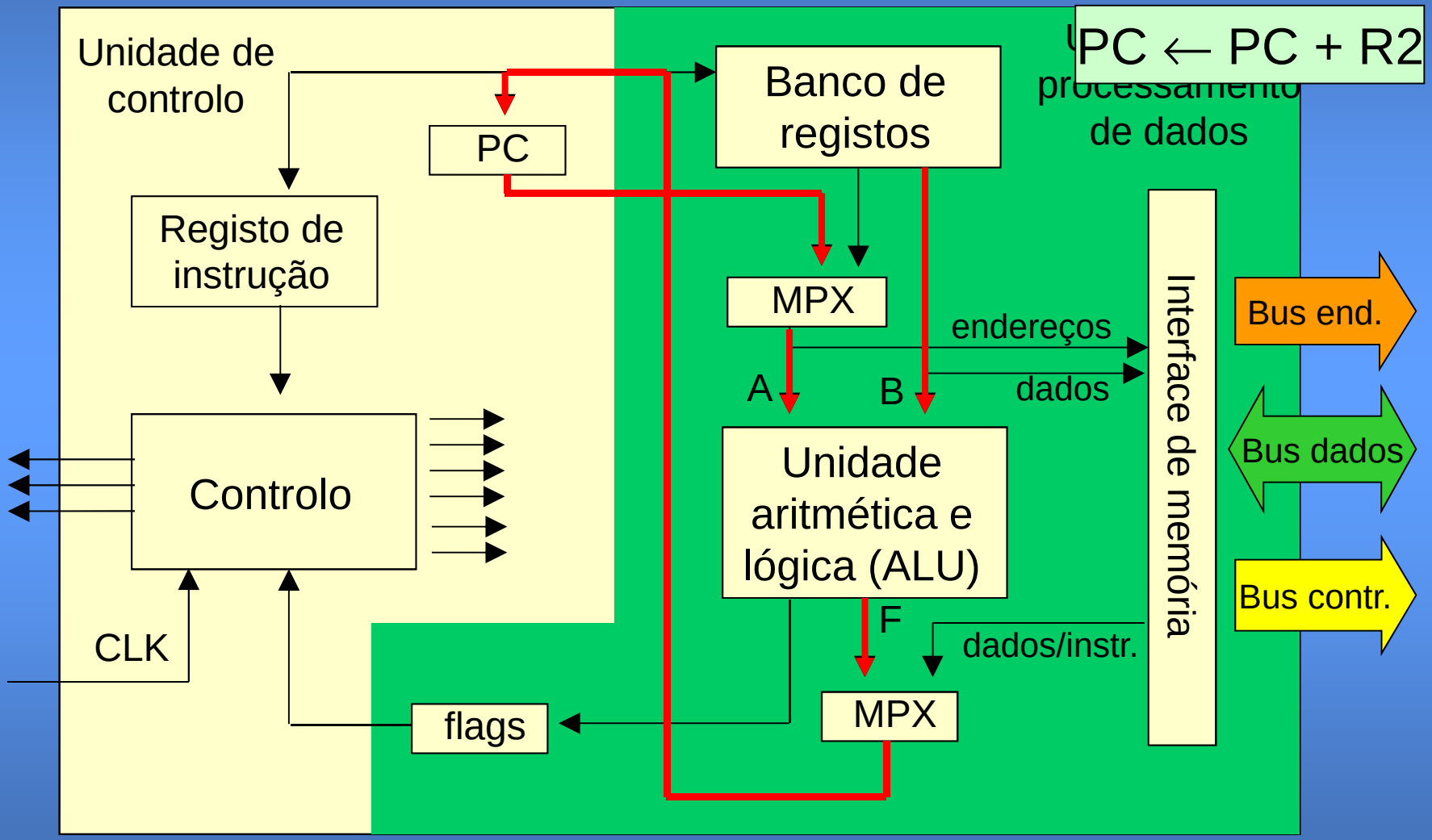
- São os sinais que vêm da unidade de controlo que comandam tudo o que acontece.
- É como se os dados fossem água. Para a encaminhar basta abrir e fechar comportas.



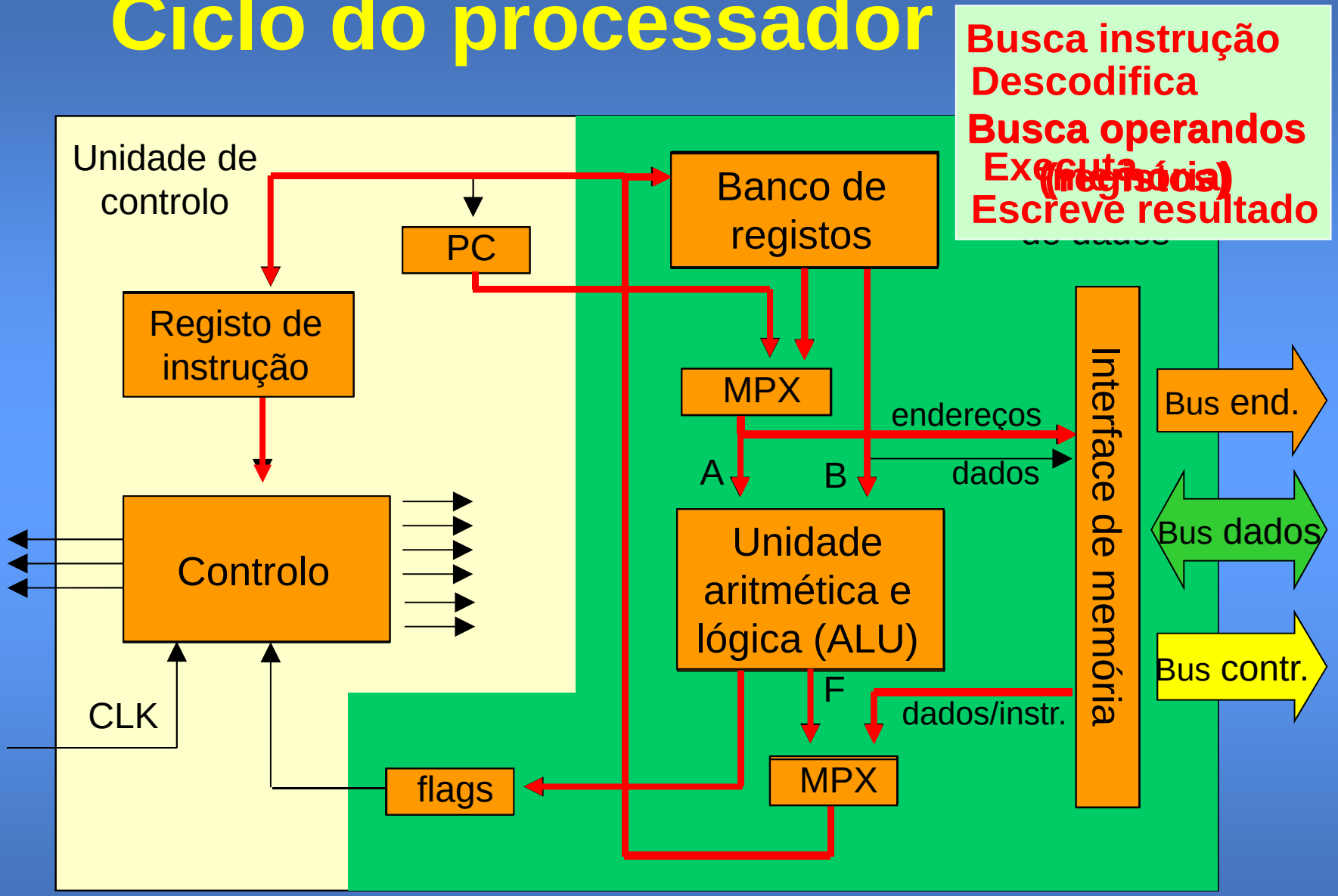
Busca de instrução (*fetch*)



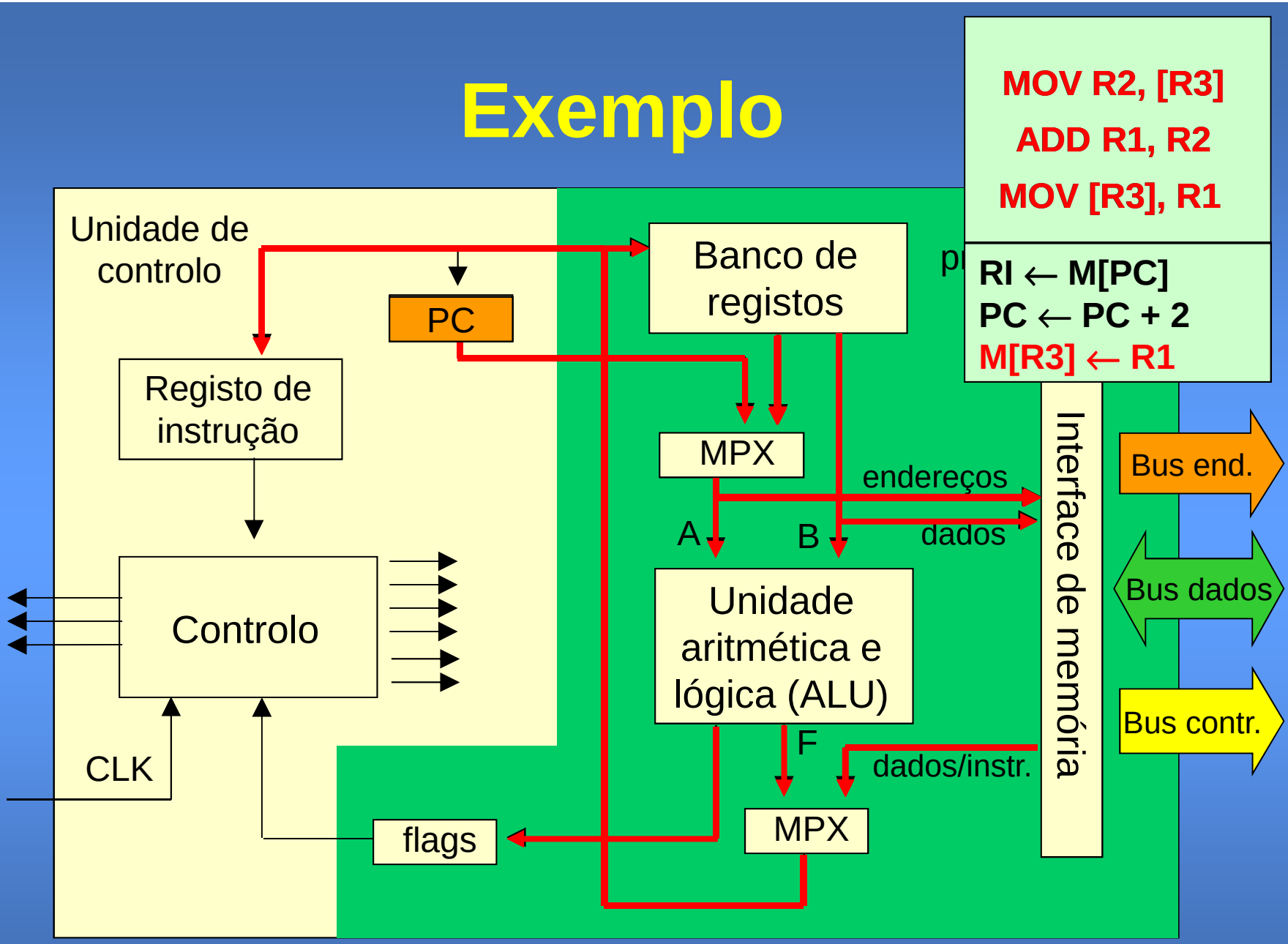
Instrução de salto



Ciclo do processador



Exemplo


MOV R2, [R3]
ADD R1, R2
MOV [R3], R1
RI ← M[PC]
PC ← PC + 2
M[R3] ← R1
Unidade de controlo
PC
Banco de registos
Registo de instrução
MPX
endereços
dados
A
B
Unidade aritmética e lógica (ALU)
Controlo
CLK
flags
F
dados/instr.
MPX
Interface de memória
Bus end.
Bus dados
Bus contr.

Exercícios

1. Indique quais os estágios fundamentais do ciclo de instrução de um processador e a sua finalidade.
2. Sobre o diagrama básico do processador indique quais as principais operações elementares para ler da memória e executar as seguintes instruções:

```
ADD   R1, 1        ; R1 ← R1 + 1
AND   R1, R2       ; R1 ← R1 ∧ R2
MOV   R2, [R3]     ; R2 ← M[R3]
MOV   [R1], R2     ; M[R1] ← R2
```

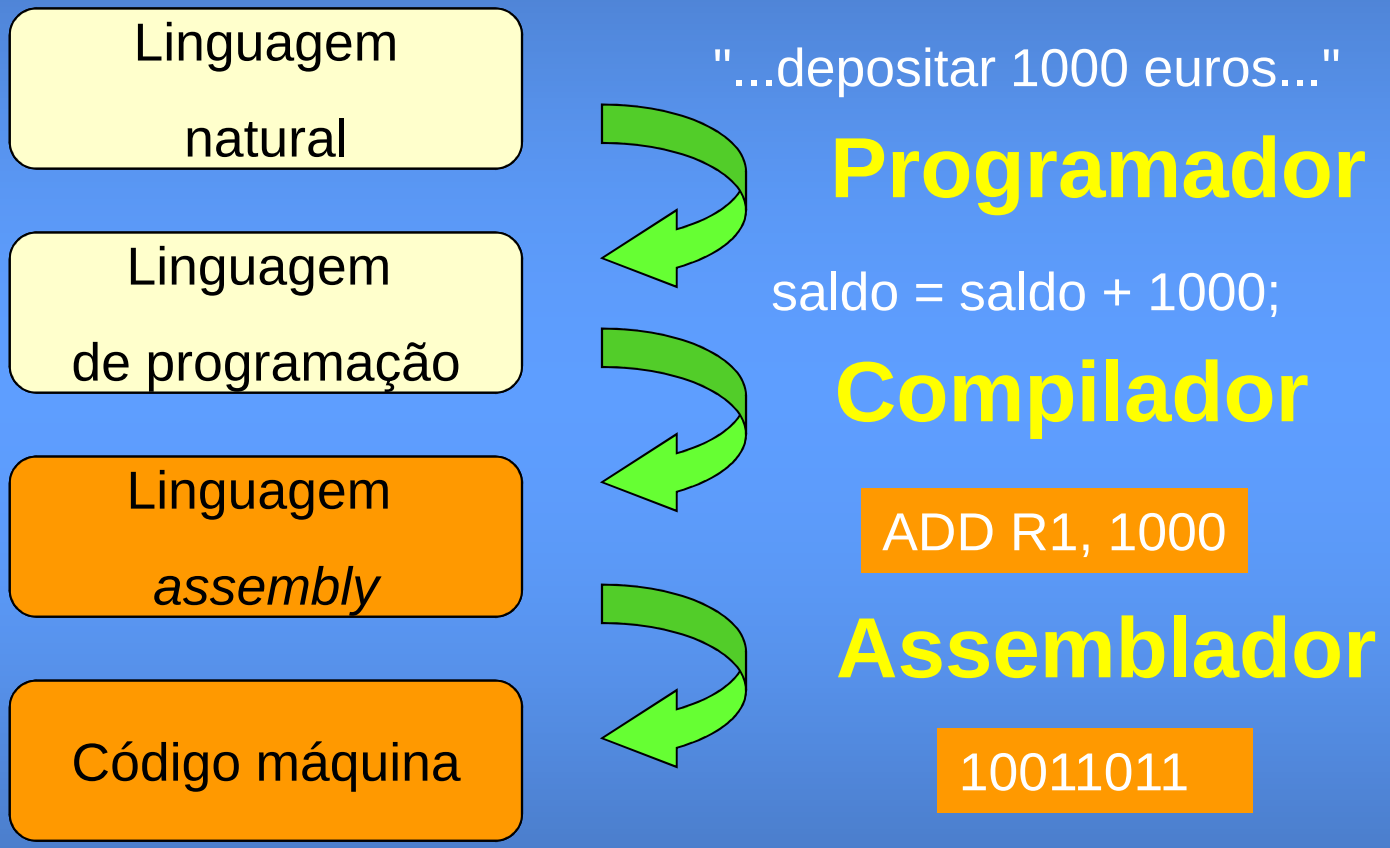
Exercícios (cont.)

3. Um processador efetuou as seguintes operações elementares:

RI ← M[PC]
PC ← PC + 2
R1 ← M[R3]
RI ← M[PC]
PC ← PC + 2
R2 ← R2 ∧ R1
RI ← M[PC]
PC ← PC + 2
M[R1] ← R2
RI ← M[PC]
PC ← PC + 2
PC ← PC + FFFEH

a) Quantas instruções executou nesta sequência?
b) Que instruções executou? Basta descrever com uma frase simples o que cada instrução faz.
c) Suponha que o processador (e os seus registos) são de 16 bits. Diga qual a instrução que o processador executará a seguir à última desta sequência.

Programação do computador



Linguagem *assembly*

- Uma instrução por linha
- Formatação rígida
- Comentários qb (escritos logo ao fazer o código!)
- As instruções refletem diretamente os recursos do processador

Comentários das instruções

- Em cada linha, o assembler ignora o caráter ";" e os que se lhe seguem (até ao fim dessa linha)
- Praticamente todas as linhas de *assembly* devem ter comentário, pois a programação é de baixo nível.
- Exemplo:

Sem comentários. O que é isto?!

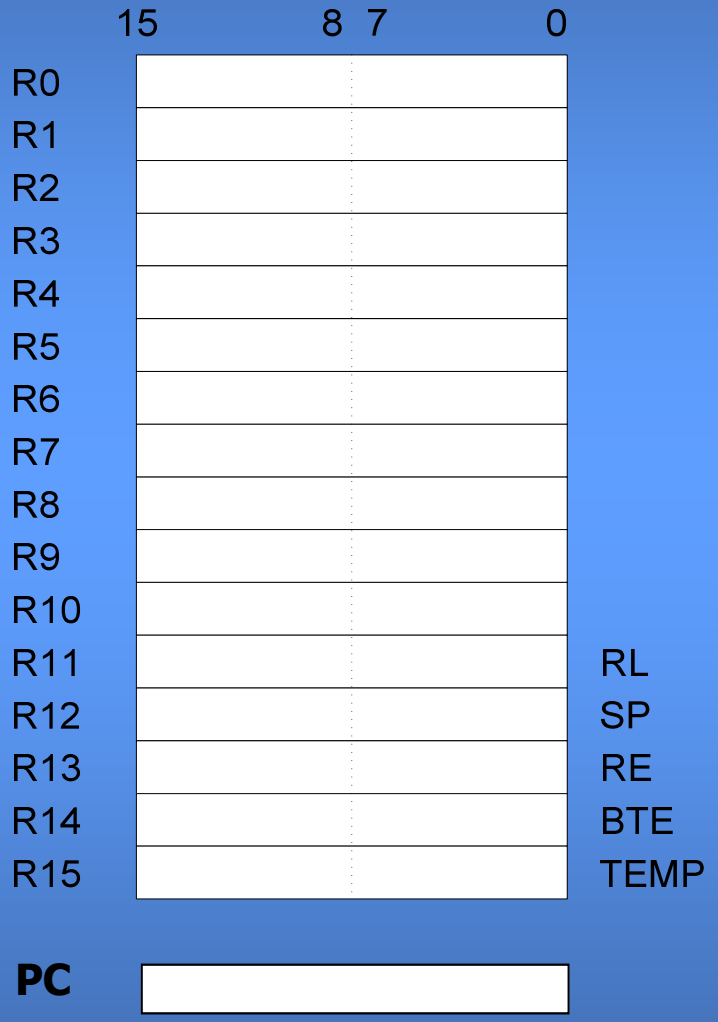
```
            PLACE   1000H       ; Início da zona de dados
Saldo:      WORD    0           ; variável Saldo inicializada a 0
            PLACE   0000H       ; Início do programa em 0000H
Deposita:   MOV     R3, 100     ; Valor a depositar na conta
            MOV     R1, Saldo   ; Endereço da variável Saldo em R1
            MOV     R2, [R1]    ; Lê o valor do saldo
            ADD     R2, R3      ; Acrescenta o valor a depositar ao saldo
            MOV     [R1], R2    ; Escreve o novo valor do Saldo
            . . .               ; . . .
```

Classes de instruções

Classe de instruções	Descrição e exemplos
Instruções aritméticas	Lidam com números em complemento para 2 ADD, SUB, CMP, MUL, DIV
Instruções de bit	Lidam com sequências de bits AND, OR, SET, SHR, ROL
Instruções de transferência de dados	Transferem dados entre dois registos ou entre um registo e a memória MOV, SWAP
Instruções de controlo de fluxo	Controlam a sequência de execução de instruções, podendo tomar decisões JMP, JZ, JNZ, CALL, RET

Registos do processador

- Os recursos mais importantes que as instruções manipulam são os registos.
- O PEPE tem os seguintes registos (todos de 16 bits):
  - PC (Program Counter);
  - 16 registos (R0 a R15), sendo alguns especiais (a ver mais tarde)



Bits de estado (*flags*)

- Fazem parte do Registo de Estado (RE).
- Fornecem indicações sobre o resultado da operação anterior (nem todas as instruções os alteram).
- Podem influenciar o resultado da operação seguinte.

Bit de estado mais importantes:	Fica a 1 se o resultado de uma operação:
(Z) Zero	for zero
(C) Transporte (*carry*)	tiver transporte
(V) Excesso (*overflow*)	não couber na palavra do processador
(N) Negativo	for negativo

Exemplo com o simulador

```
; Exemplos de instruções simples no PEPE

        MOV     R2, 25H     ; coloca uma constante de um byte no R2
        MOV     R3, 1234H   ; coloca uma constante de dois bytes no R3
        ADD     R2, R3      ; exemplo de soma
        SUB     R2, R2      ; para dar zero e ver os bits de estado

        MOV     R4, 5       ; ilustrar ciclo de 5 iterações
ciclo:  SUB     R4, 1       ; decrementa contador e afecta bits de estado
        JNZ     ciclo       ; salta se ainda não tiver chegado a 0

        SHR     R3, 2       ; deslocamento de 2 bits à direita

fim:    JMP     fim         ; forma expedita de "terminar"
```

Implementar no PEPE

- Objectivo: somar um número com todos os inteiros positivos menores que ele.

  soma = N + (N-1) + (N-2) + . . . + 2 + 1

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

Outro exemplo: contar bits a 1 em 76H

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)

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