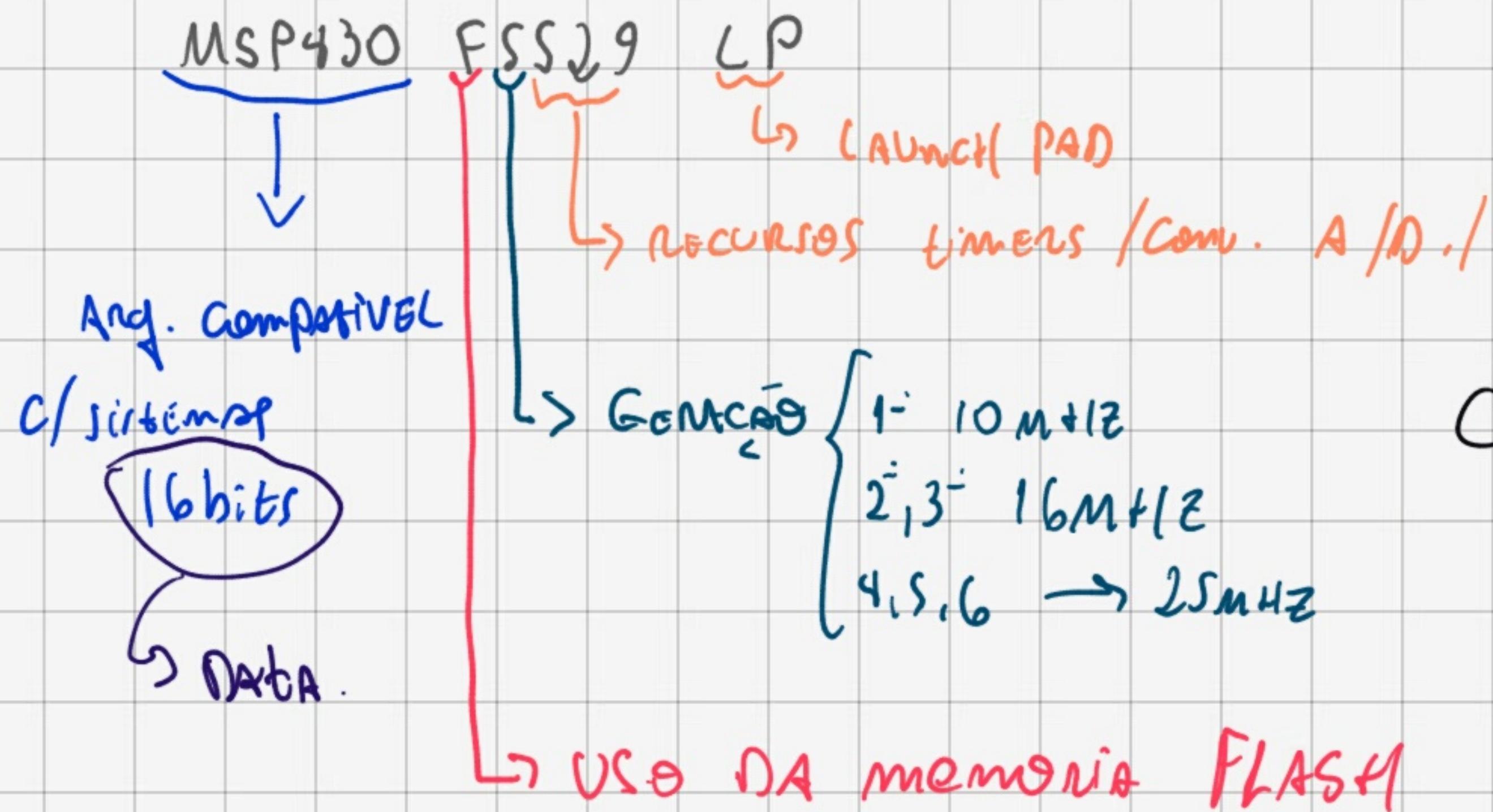


Sistemas Microprocessadores

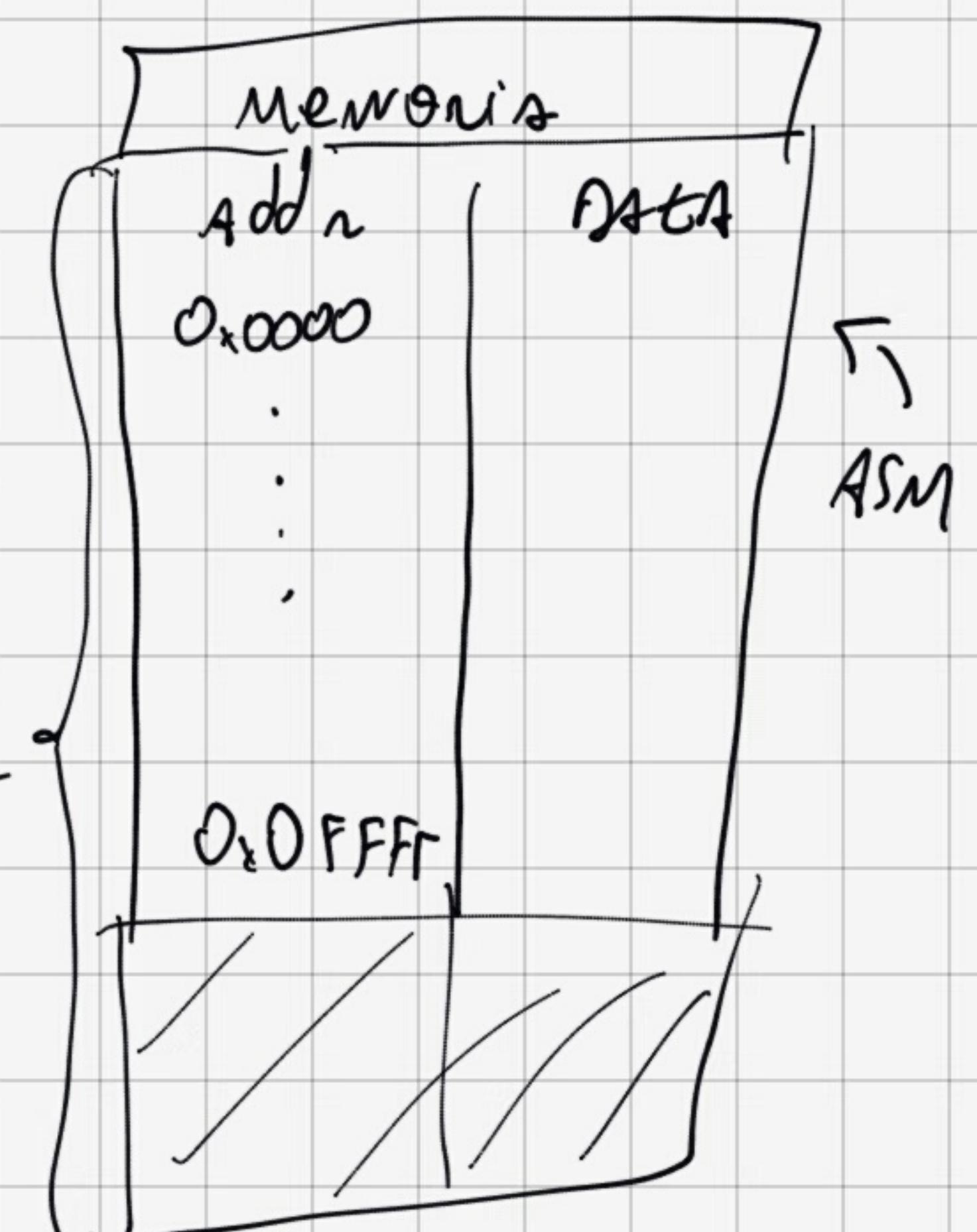
- Sistemas embarcados
- Arq. de processadores digitais
- Sistemas digitais

→ Plataforma de desenvolvimento



CPU x ~ 20 bits

Abbr.



\*\* Bibliografia Técnica :

• DATA SHEET }  
• USER'S GUIDE } MSP430FS29

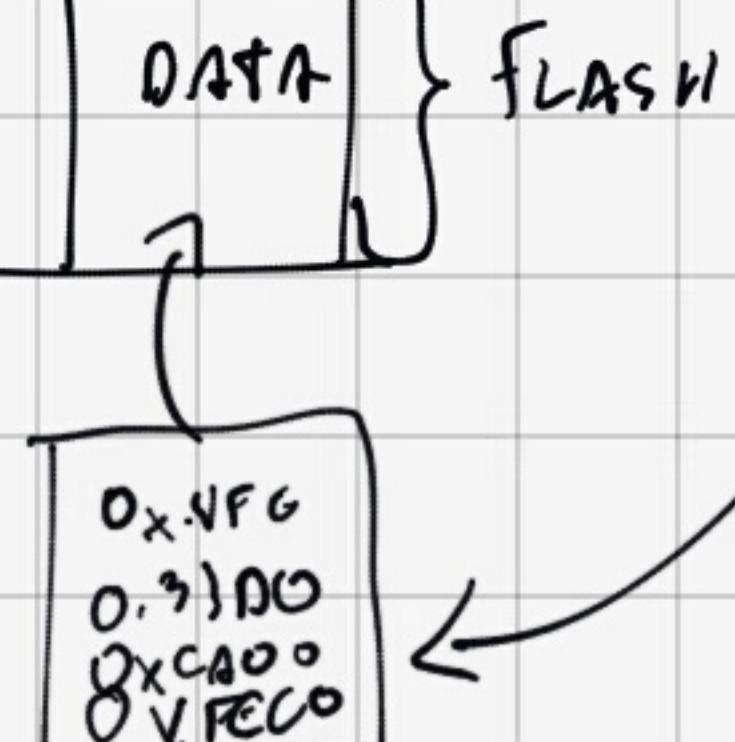
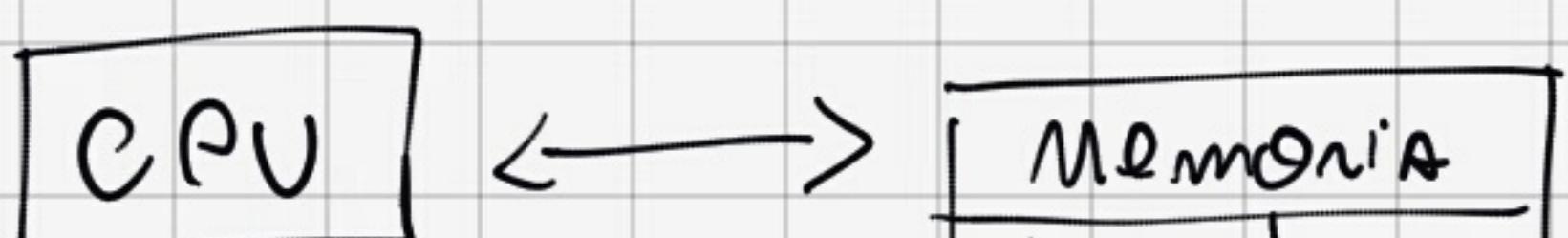
• Launch Pad User's Guide → Esquematicos (Pontas de entradas/saídas)

Ai.com → products → Microcontrollers (mcu)

→ MSP430FSx → user's guide → technical Docs → DataSheet  
MSP430FS2x

\*\* Sistemas Embaçados

→ Dispositivos de hardware programados para executar uma tarefa especializada de maneira repetitiva.



```
int main() {
    config();
    do_something();
}
```

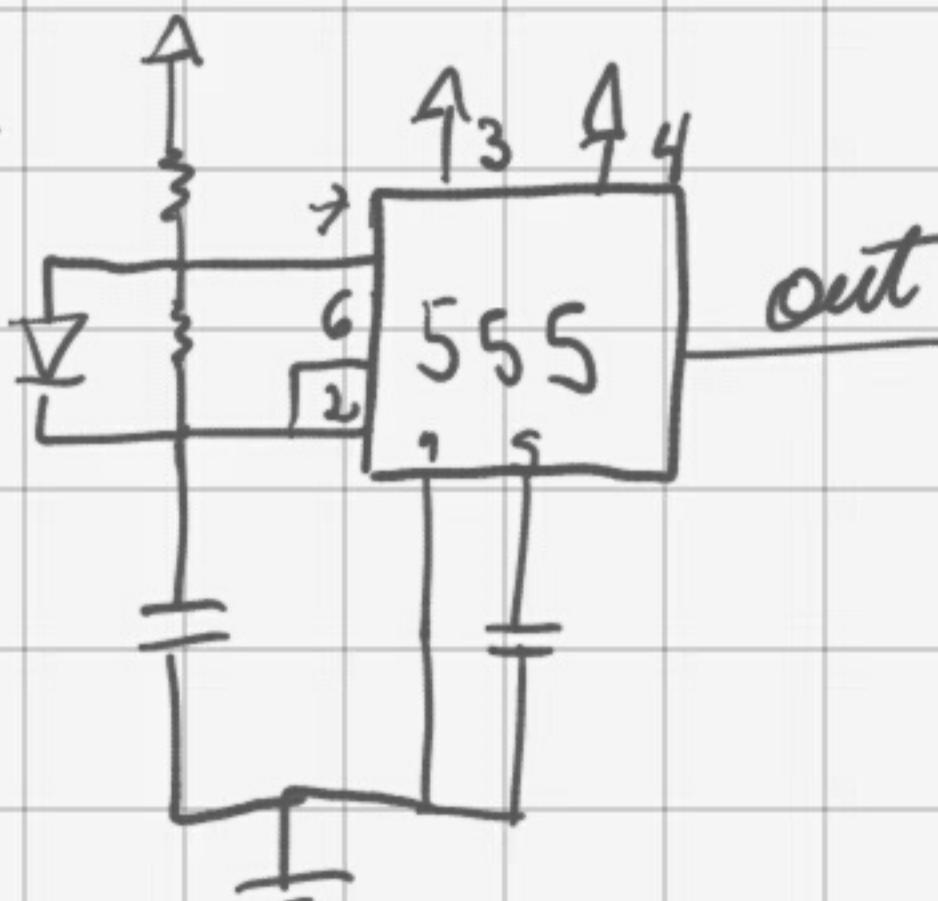
Compilator e Linker C

# \* Sistemas Microprocessados / Embaixadores / Cyber Physical systems \*

Exemplos:

- Carros c/ eletrônica embarcada
- CELULARES
- REDEADORES
- Escova de dente elétrica

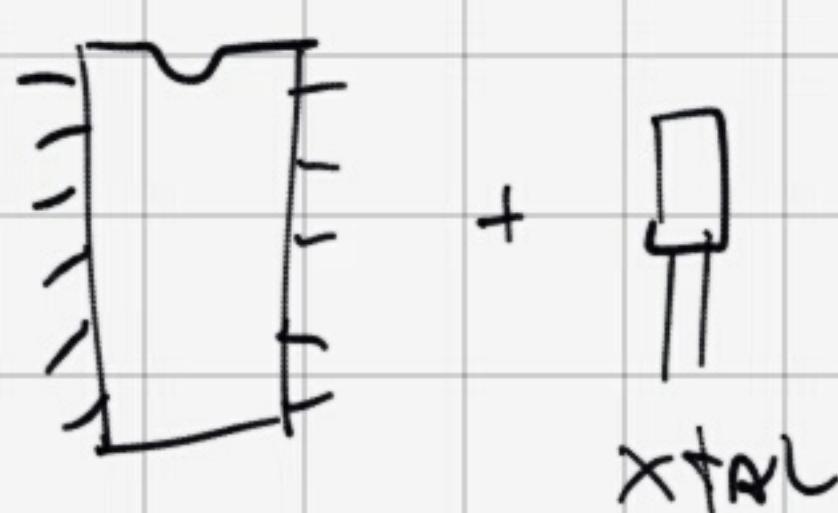
→ Projeto Escova de Dente (Opção) - timer 555



+ Resolução de Problemas  
- Funcionalidade limitada.

2º opção) - OC4060 → Contador MOS

DI P14



+ Contar tempo  
+ Precisão  
- Funcionalidade limitada

3º opção)

MSP430	{	J: \$ 7,33
10: \$ 6,61		
100: :		
1000: :		

SCM ACADEMICA: 24 a 28  
minicurso com Daniel Caffé  
↳ Line board

Microcontrolador VS (micro) Processador

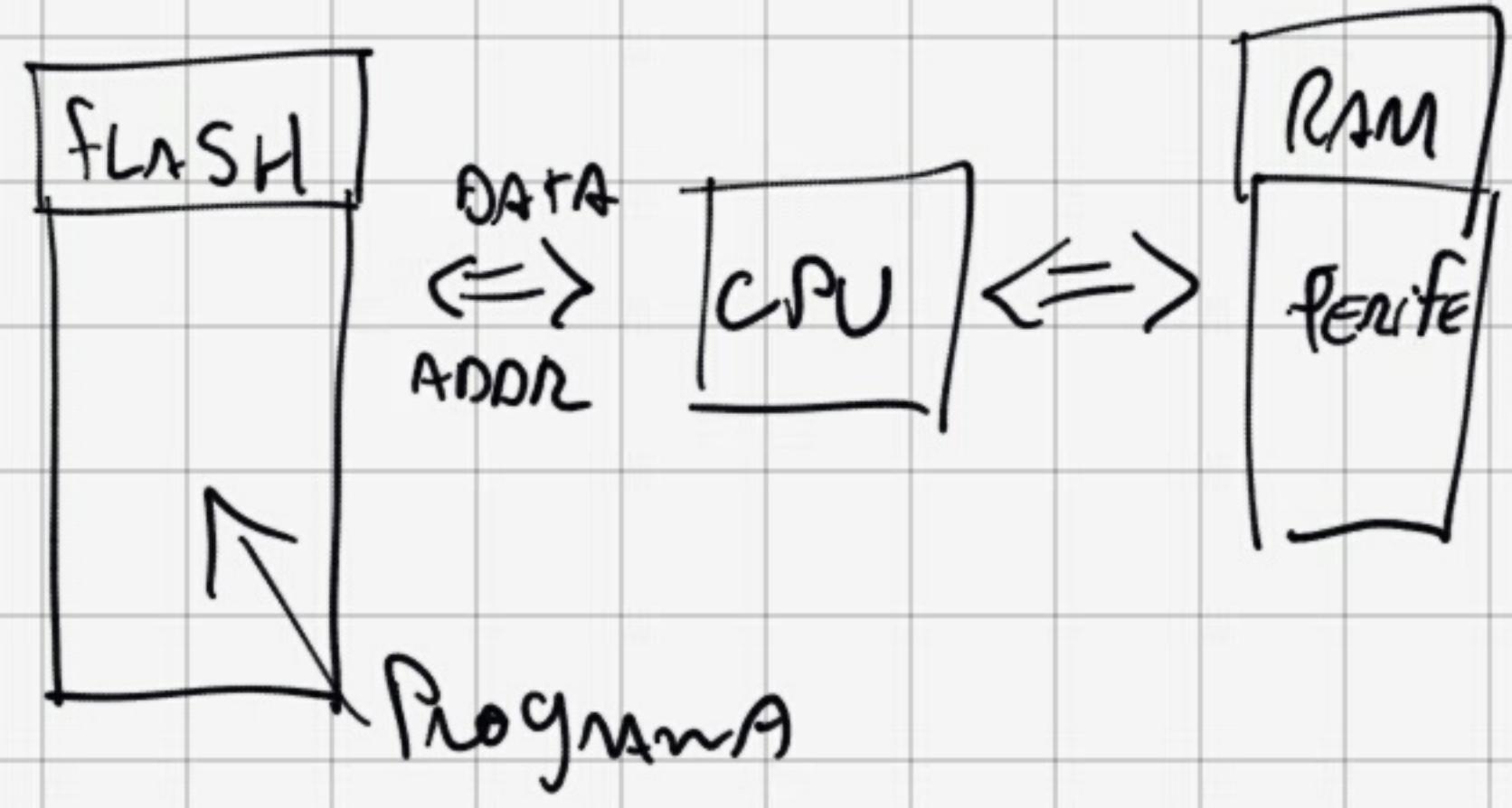
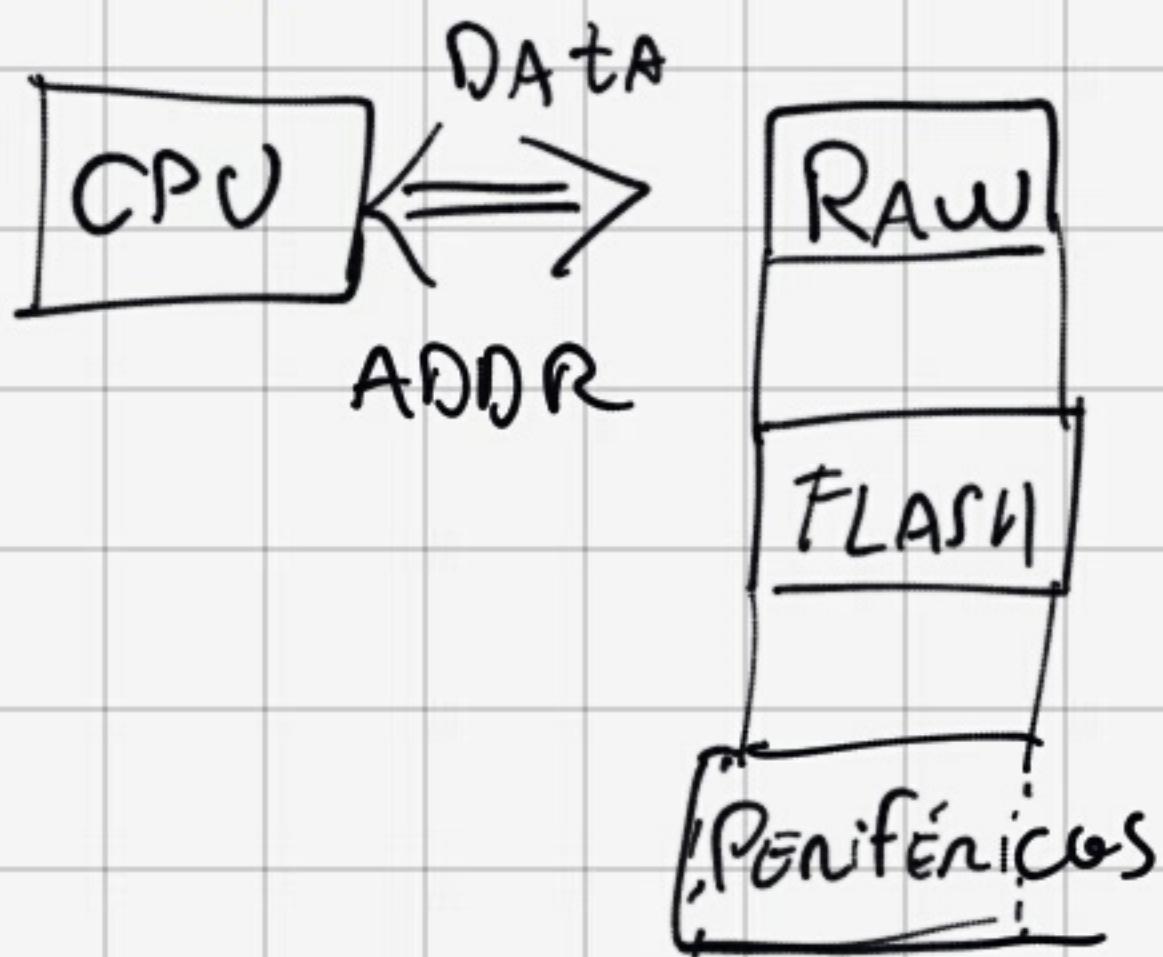
- MSP430, PIC
- Motorola 6800, Atmel
- Intel i3/i5/i7/i9
- Qualcomm Snapdragon (200, 400, 600, 800)

- Mediatek

- Função dedicada
- Memória volátil e não volátil no mesmo chip
- Clock  $\rightarrow$  10MHz  $\leftrightarrow$  100MHz
- MW

- Sist. operacional
- Armazenamento separado da CPU (HDD, SSD, ...)
- ~ 1GHz
- W (onibus de grandeza superior)

## Organização Von Newmann Vs Harvard



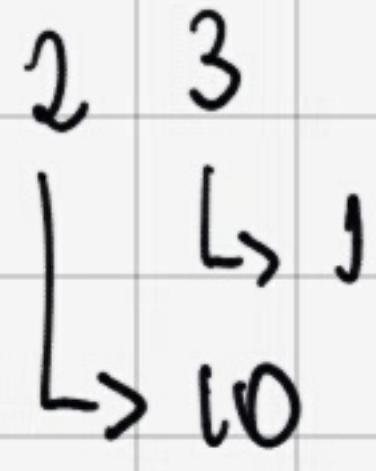
Quantos bits de endereçamento são necessários num MC com 512 kB de memória FLASH.

\* AVCA 2 → Assembly  
→ Livro texto → Sec 3.3 cap S - John Davis

- Representações Númericas
- Instruções Básicas de ASM.
- Fluxograma

## \* REPRESENTAÇÕES NÚMÉRICAS

- ↳ Binária.
- ↳ Hexadecimal



$$B = b_N \cdot 2^N + b_{N-1} \cdot 2^{N-1} + \dots + b_0 \cdot 2^0 ; \quad b \in \{0, 1\}$$

↑      ↑  
dígito    base

→ bases de pot de 2,  $N > 1$ .

$$\text{base} = 16 = 2^4$$

$$H = d_N \cdot 16^N + d_{N-1} \cdot 16^{N-1} + \dots + d_0 \cdot 16^0$$

↳ Na base 16 temos esses símbolos representativos:

$$\text{Símbolos: } [0-9, A, B, C, D, E, F]$$

↑      ↑      ↑  
10    11    12

$$\Rightarrow 0x3A4C \Rightarrow 3 \cdot 16^3 + 10 \cdot 16^2 + 4 \cdot 16 + 12$$

\* transformar hexa → Binário

Ex1

$$0x\overbrace{3A4C}^{\text{Suba byte}}$$

↓

$$0011|1010|0100|1100$$

) Ex2:

$$= 0xCAFE$$



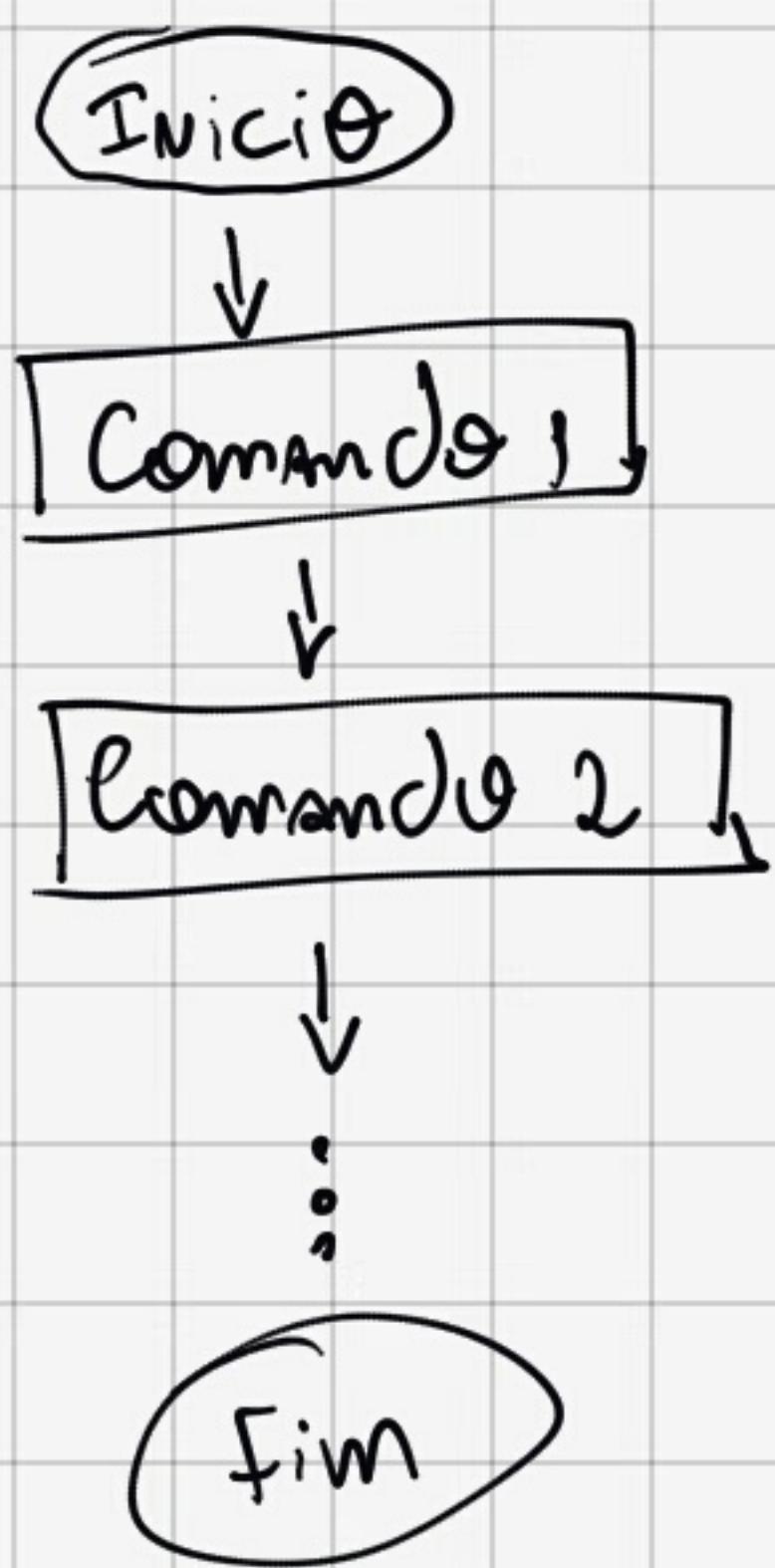
$$1100|1010|1111|1110$$

Agrupamentos:  
4 bits → Nibble  
8 bits → Byte  
16 bits → word  
32 bits → double (Double word)

"Antes de falar de Assembly Vamos falar de fluxogramma"

## \* Fluxogramma:

Ex: Receita de Bolacha



"(+) Ordem das instruções."

→ Pode ser usada qualquer linguagem:

- linguagem NATURAL (pt, en)
- C, ASM, C++, ...

(+) Documentação

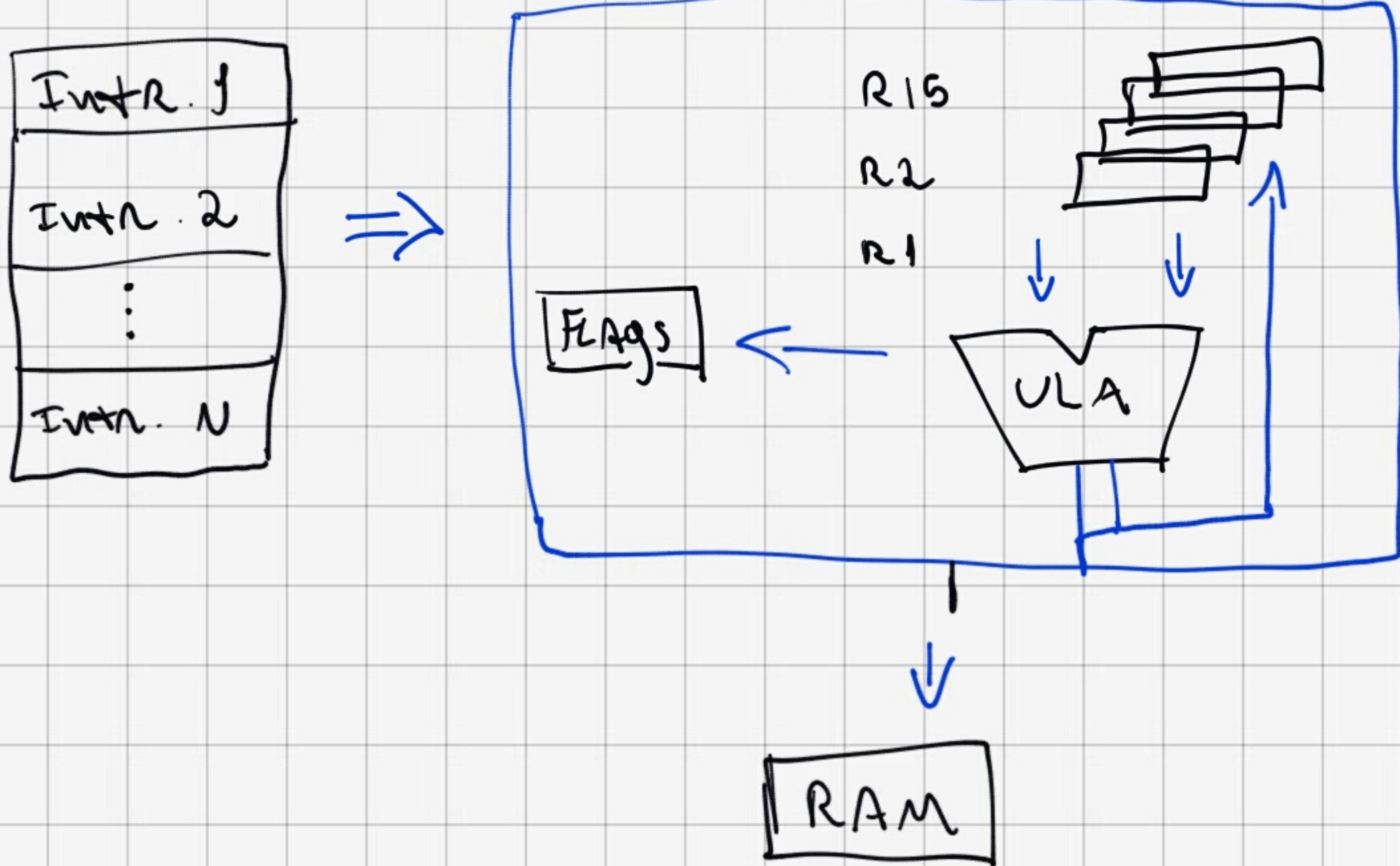
(+) Comunicação

(+) Fácil / Visualização de Algoritmos

(-) Necessita tradução p/ linguagem HLL (ASM)

(-) Algumas construções não têm equivalente direto

→ O fluxogramma retrata a execução sequencial de um processador (CPU)



## \* Assembly

Sintaxe Simplificada:

MOV: Copia

SRC  $\Rightarrow$  DST

MOV      SRC, DST  
Instrução      Operando

Ex: MOV R4, RS.

R4 0x1234  
RS 0xABC0

R4 0x1234  
RS 0x1234

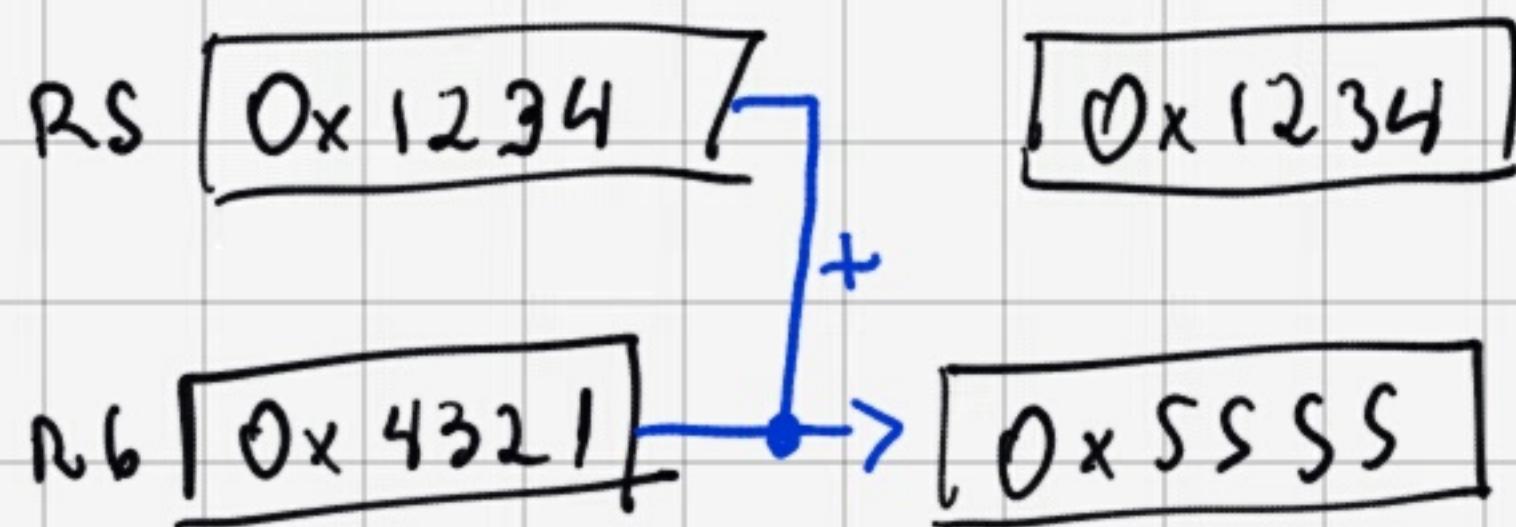
► Soma : ADD (Soma comutativa)

ADD      SRC      DST  
        RS , R6

$$R6 \leftarrow RS + R6$$

Antes

Depois



► Gutras operações,

Limpar Reg \*      CLR      RN

Mover      MOV      Ra, Rb

↳ Soma  
└ Subtração \*

ADD      Ra, Rb

Sub      Ra, Rb

Rotacões      RRA      RLA  
                  RRC      RLC      RN

Lógica      AND, BIC  
                  BIS, XOR, Ra, Rb

Inversão  
Negação      INV

► Sintaxe completa:

Coluna

1	3	21	45
LABEL:	Instr	OP1, OP2	j comentários
MAIN:	CLR	RS	; Limpar RS
	MOV	R6, R7	; Copia R6 em R7

Label : marcador da posição ou início de blocos de instruções

Instr : mnemônico da instrução MOV, ADD, etc...

OP1, OP2 : registros ou posições de memória R4, RS ...

Comentários: Sempre Iniciados com ponto-e-vírgula.  
Em Assembly, os comentários são importantíssimos pois  
o código se torna incompreensível sem eles.

Monitoria: Quarta-Feira 22/8 → 12h  
Local SALA X Do corredor de eletrônica ; Monitor: Victor.

\* Exercícios:

1. Colocar em R6 a soma de R6 com R5

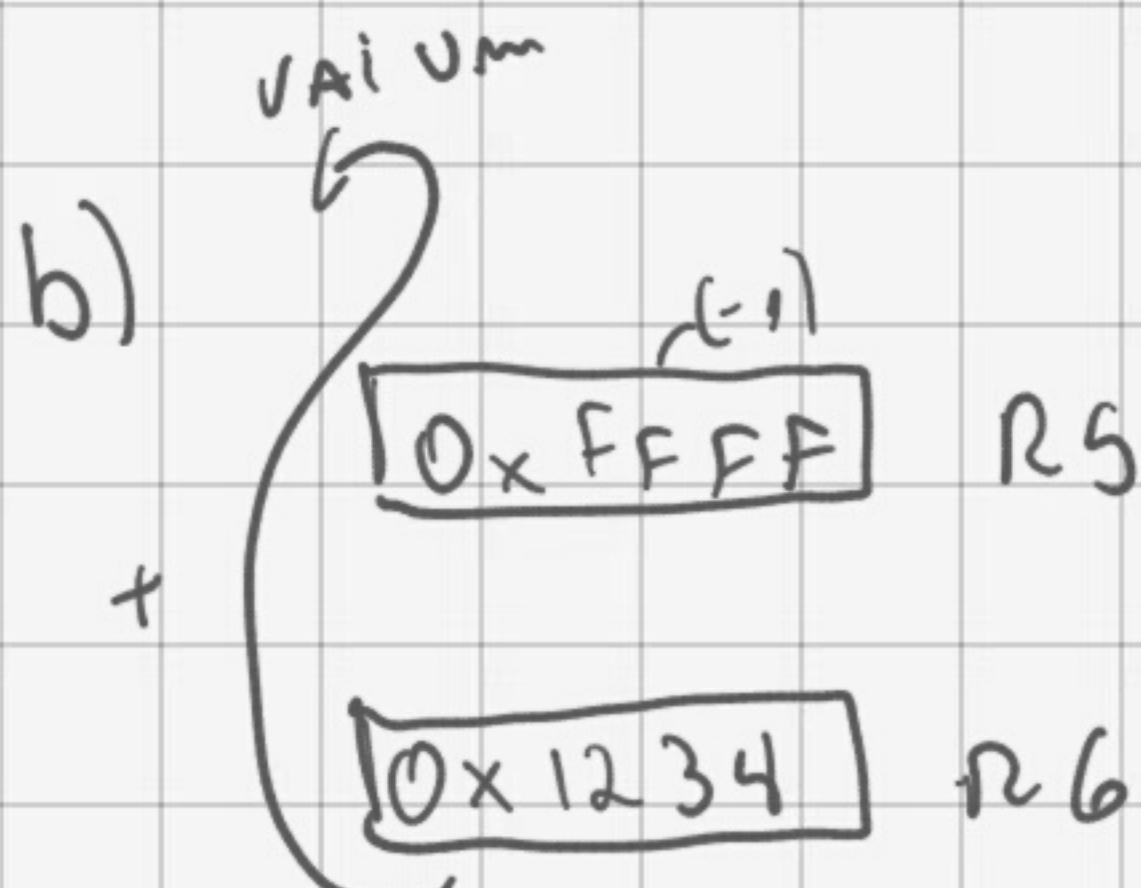
2.

a) Colocar em R7 a soma de R6 com R5 sem alterar R5/R6

b) E se a soma ultrapassar 16 bits?

1. ADD R5, R6 ;  $R6 \leftarrow R5 + R6$

2.a) MOV R5, R7 ;  $R7 \leftarrow R5 + R6$   
ADD R6, R7



$$R5 + R6 \Rightarrow [R7, R8]$$

MSW LSW

CLR R7  
MOV R5, R8  
ADD R6, R8  
ADDC #0, R7 ;  $R7 \leftarrow R7$

↳ Constante

$+0$   
 $r_c$

## XX Operações Bit a bit

### 10 Bit Set

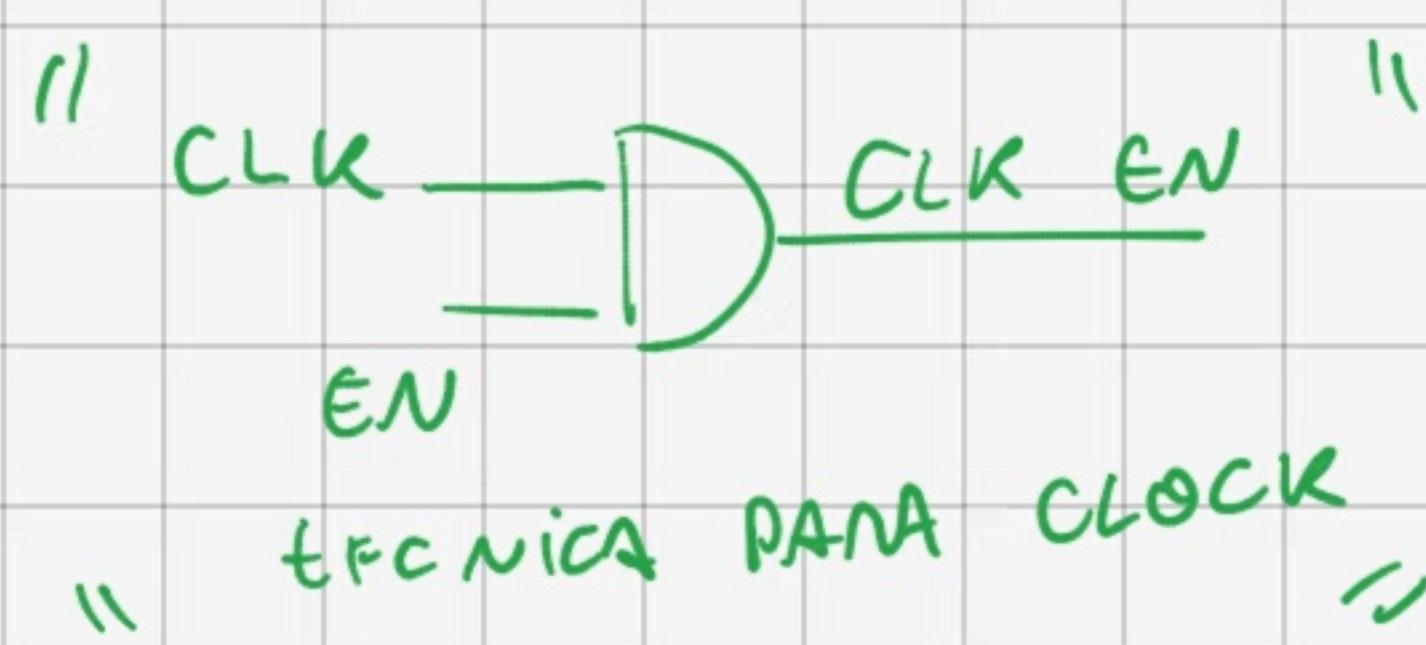
R5    0 0 0 0 1 1 1 1

(OR)

$$\begin{array}{r} 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ \hline 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \end{array} ?$$

BIS # 0x40, RS

BIS # BIT6, RS



$$A \quad B \quad S = A + B$$

Or

⇒ Só é 0 quando os dois forem 0"

A	B	S
0	0	0
0	1	1
1	0	1
1	1	1

$$f(A_1, \dots, A_n) = \sum_{i=1}^n A_i$$

### 10 Bit Clear:

0 0 0 0 1 1 1 1    RS  
 7 6 5 4 3 2 1 0  
 1 1 1 1 1 0 1 1    (AND)

0 0 0 0 1 0 1 1

BIC # BIT2, RS

AND Not (# BIT2), RS

# BIT2 = #01 + 04

$$A \quad B \quad S = A \cdot B$$

And

⇒ Só é 1 quando os dois forem 1"

A	B	S
0	0	0
0	1	0
1	0	0
1	1	1

$$f(A_1, \dots, A_n) = \prod_{i=1}^n A_i$$

### Bit toggle

0 0 0 0 1 1 1 1    (XOR)

0 1 0 0 0 1 0 0

0 1 0 0 1 0 1 1

XOR # (RS2 | BIT6), RS

XOR # 0x48, RS

$$A \quad B \quad S = A \oplus B$$

XOR (exclusivo)

$$f(A, B) = \overline{A} \cdot B + A \cdot \overline{B}$$

$$= A \oplus B$$

A	B	S
0	0	0
0	1	1
1	0	1
1	1	0

\* Exercícios: Subtrair um número de 48 bits  
Aprendendo por R7, de centro  
(também de 48 bits) Aprendendo por RS.

Dica: pergunto: o operando @  
↳ Subtração com borrow  
↳ SUBC

## \*Revisando AULA 2\*

Sintaxe Simplificada:

MOV SRC, DST  
Instrução Operações

Ex 3) MOV : Cópia SRC  $\Rightarrow$  DST

Ex:

Conigo  $\rightarrow$  MOV R4, RS

Antes

R4 0x1234



Depois

R4 0x1234

RS 0xABCD

RS 0x1234

Ex 2) Soma : ADD (Soma cumulativa)

$\nwarrow$  SRC  $\searrow$  DST

Conigo  $\rightarrow$  ADD RS, R6

"  $R6 \leftarrow RS + R6$ "

Antes

RS 0x1234

R6 0x4321

Depois

RS 0x1234

0x SSSS

## \* Outras operações

- Limpar Registro\*: CLR R (Número do reg.) ou nome
- MOVER : MOV Ra, Rb
- Soma Subtração\*: ADD Ra, Rb " Sub Ra, Rb
- Rotação: RRA RLA RN  
RRC RLC
- Lógica: AND, BIC Ra, Rb  
BIS, XOR
- Inversão Negação: INV

## Sintaxe Completa:

Label:	INTR	OP1, OP2	; Comandos
MAIN:	CLR	RS	; Limpar RS
	MOV	R6, R6	; Copiar R6 em R7

Exemplos:

1) ADD R6, RS ;  $R6 \leftarrow RS + R6$

2) a) MOV RS, R7 ;  $R7 \leftarrow RS + R6$   
 ADD R6, R7

b) CLR R7  
 MOV RS, R8  
 ADD R6, R8  
 ADDC #0, R7 ;  $R7 \leftarrow R7$   
*↳ constante*

Explicação:  
 } Variável (-)  
 } 0xFF  
 +  
 } 0x1234

RS  
 R6

$RS + R6 \Rightarrow [R7, R8]$   
 msb Lsb

## \* Operações Bit a bit:

Mas Antes, Vamos revisar alguns conceitos:

- $\text{HEXA} \rightarrow \text{BINARIO}$

→ OR

→ AND

→ XOR

\* Conversão Hex → Binário

1)  $0x3A4C$

2)  $0xCAFE$

1)  $0x3A4C$

$$\begin{array}{r} \swarrow \downarrow \searrow \\ 0011 / 1010 / 0100 / 1100 \end{array} = 111010001001100_2$$

$$= 2^{13} 2^{12} 2^{11} 2^9 2^8 2^7 2^6 2^5 2^4 2^3 2^2$$

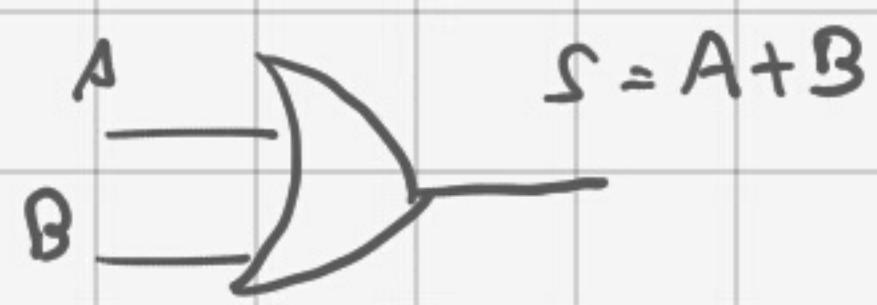
$$= 14924 \text{ (decimal)}$$

2)  $0xCAFE$

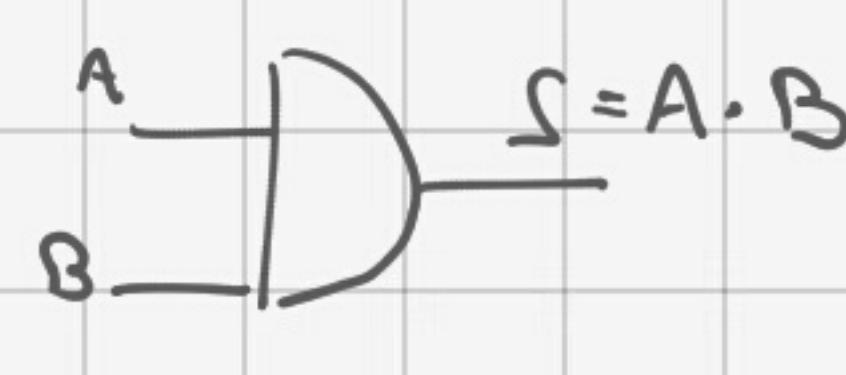
$$\begin{array}{r} \swarrow \downarrow \searrow \\ 1100 \quad 1010 \quad 1111 \quad 1110 \end{array} = \underbrace{1100}_{2^{14}} \underbrace{1010}_{2^{13}} \underbrace{1111}_{2^{12}} \underbrace{1110}_{2^{11}} = 51966$$

H	B
0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1
A	1 0 1 0
B	1 0 1 1
C	1 1 0 0
D	1 1 0 1
E	1 1 1 0
F	1 1 1 1

## \* Pontas Lógicas:



Or



AND



XOR (Exclusivo)

A	B	S
0	0	0
0	1	1
1	0	1
1	1	0

"

A	B	S
0	0	0
0	1	0
1	0	0
1	1	1

"

⇒ Só é 0 quando os dois forem 0"

A	B	S
0	0	0
0	1	1
1	0	1
1	1	0

⇒ é 0 quando as duas entradas forem iguais"

$$f(A_1, \dots, A_n) = \sum_{i=1}^n A_i$$

$$f(A_1, \dots, A_n) = \prod_{i=1}^n A_i$$

$$f(A, B) = \overline{A} \cdot B + A \cdot \overline{B}$$

$$= A \oplus B$$

\*OpenC3S Bit A Bit #

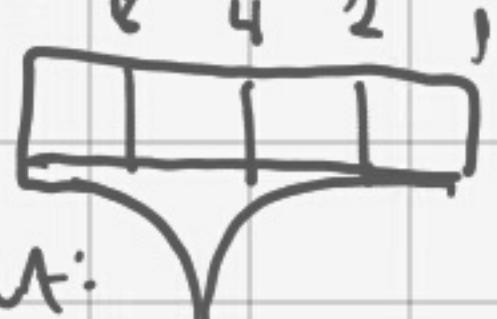
## \*\* Aula 3 : Assembly

"Programm IAR System"

- Modificações de ops.
- Acesso à memória
- controle de fluxo

### Recapitulando

Programa 3.a: Setando Bits usando MASCASAS .

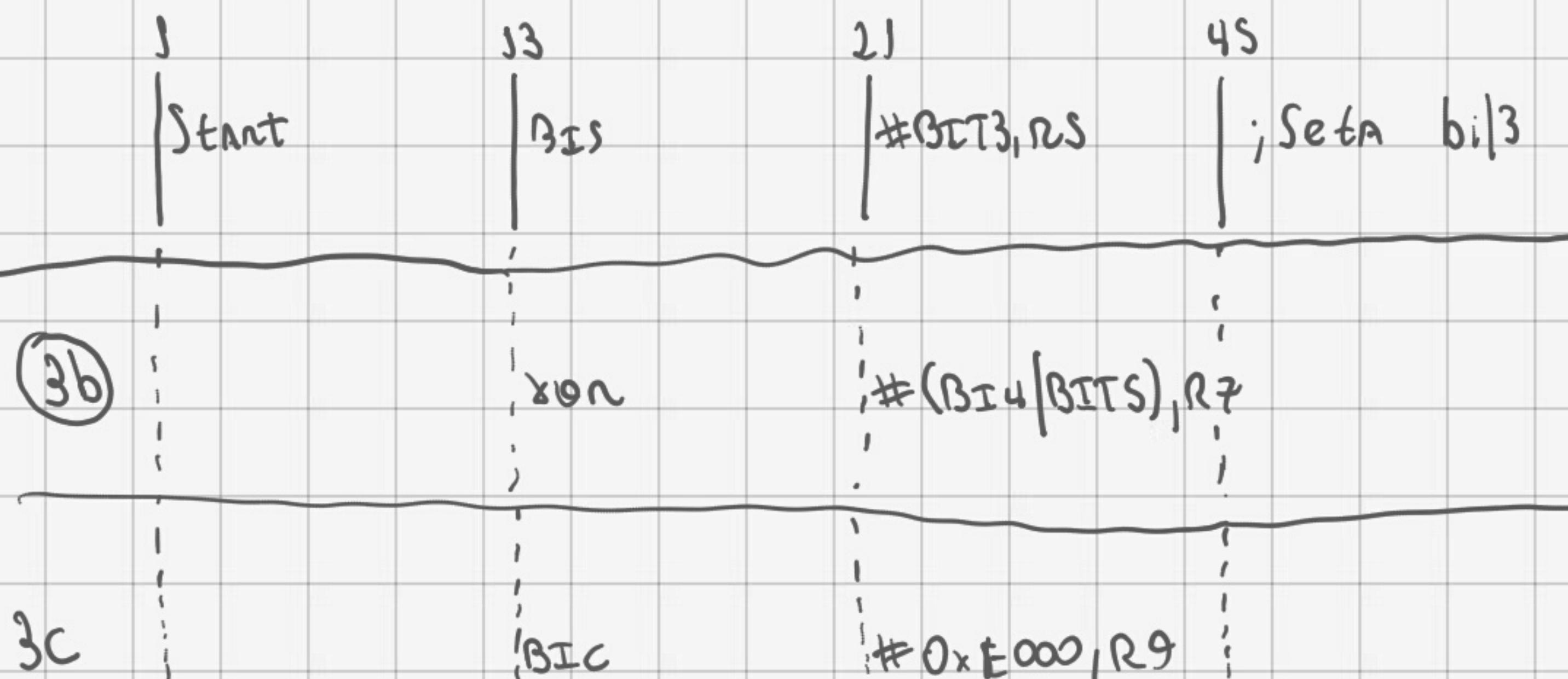


criando uma MASCARA:

BIT 0	. Set	0x0001
BIT 1	. Set	0x0002
BIT 2	. Set	0x0004
BIT 3	. Set	0x0008
BIT 4	. Set	0x0010
:		
BIT 15	. Set	0x8000

Setando 11 Bits

BIS, BIC, XOR, AND, INV



► Instruções do MSP430 tem 2 modos

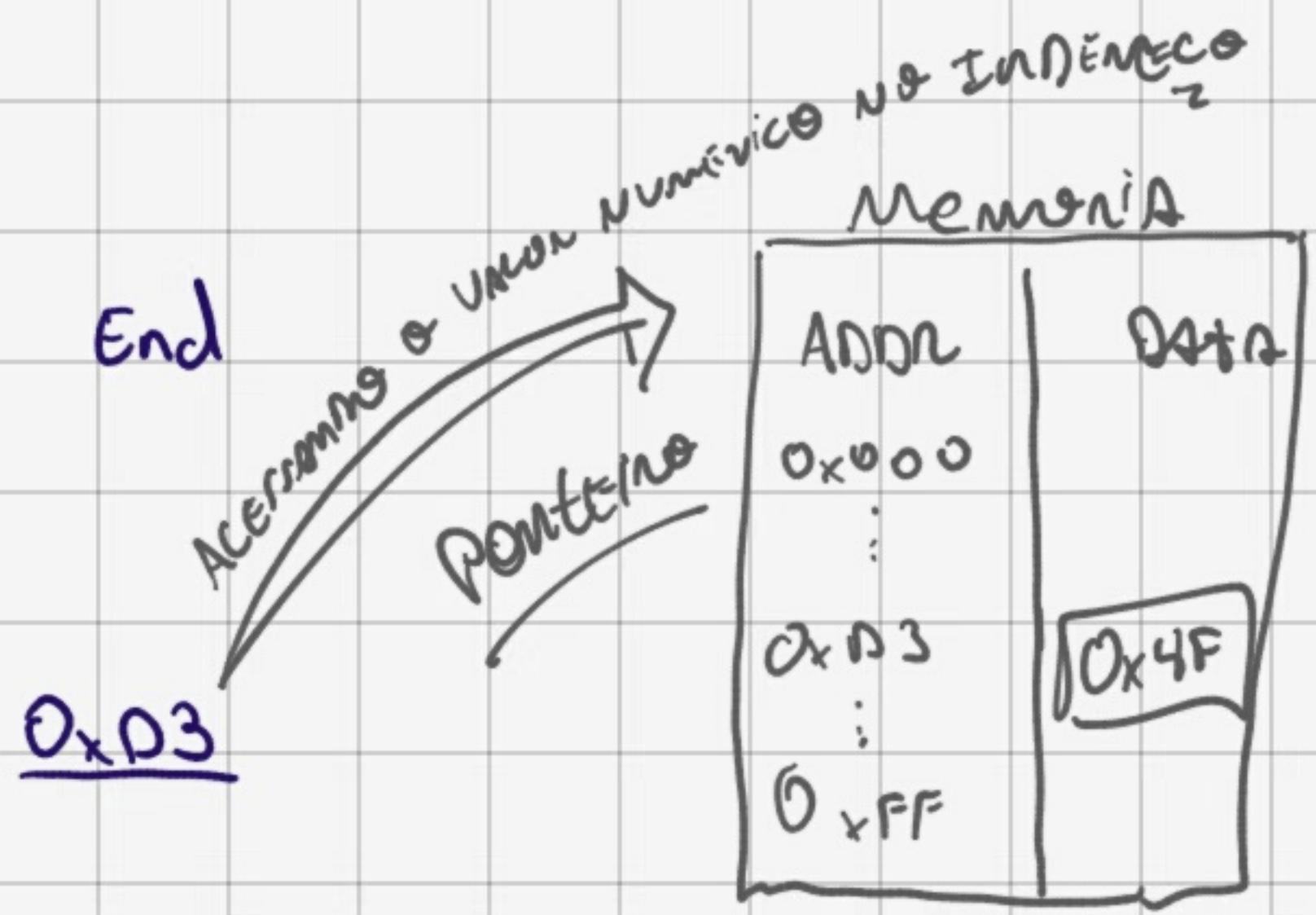
MOV .B → Operações com 8 bits  
↳ Byte

MOV .W → Operações com 16 bits  
↳ word

MOV .X → Operações em 20 bits : fun no grupo do curso.  
↳ Address

## ► Interpretação do conteúdo dosregs.

	VALOR	$\star\star(\text{den})$	$\star\star(\text{s})$	ASCII	End
R5	0x41	65	65	'A'	
R6	0xFF	255	-1		
R7	0xD3	211	-45		
R8	0x65	101	101	'e'	



## ► Operadores de acesso à memória:

### - Operador @

Antes:

R7	0x2F
R8	0x15

Depois:

R7	0x34
R8	0xF8

Mem	
ADDR	DATA
15h	24h
2F h	4Fh
34h	E3h

MOV #0x34, R7

ADD @R7, R8  
E3h

⚠ Atenção: O operador @ só pode ser usado na fonte (SRC)

- ADD @R7, R8 ✓
- ADD R7, @R8 X
- ADD @R7, @R8 X

## ► Estrutura de dados:

1. Vectors

2. Strings

→ Vectors: J: convenção de programação conhece o tamanho do vetor.

$$V = [1, 2, 3] \Rightarrow$$

MEM	
ADDR	DATA
:	
0x14	J
0x15	2
0x16	3
0x17	S
:	?

vector de 3 elem. V →

Mem	
ADDR	DATA
30h	0x12
31h	0x34

BIT mais significativo ↑      BIT MENOS significativo ↑

### VETORES

1º Cenário: O número de ELEMENTOS faz do vetor

VETOR de 4 ELEMENTOS

Mem	
ADDR	DATA
0x14	0x04
0x15	0xA
0x16	0xB
0x17	0xC
0x18	0xD
:	:

RS | 0x0014 → (meta - data) cabeça

### Exercício:

Realizar a soma comutativa de VETOR apontadas por RS, que contém 3 ELEMENTOS  
∴ VETOR de Bytes

RS | 0x14

```

MOV.B    @RS, R8
ADD.B    #J, RS
ADD.B    @RS, R8
(INCREMENTO) INC.B   R5
ADD.B    @RS, R8
  
```

⇒ USO da operação @Rm + (Pós-Incremento) : J realiza a operação

pós INCREMENTO

2º Incrementa o registro de endereço.

```

MOV.B    @RS+, R8
ADD.B    @RS+, R8
ADD.B    @RS, R8
  
```

⇒ VETOR 16 bits

"O Início é sempre P&N"

MEMÓRIA	
ADDR	DATA
0x14	0x34
0x15	0x12
0x16	0xCD
0x17	0xAB

Little endian

VETOR de 2

ELEMENTOS

$$V = [0x1234, 0xABCD]$$

MOV.W @RS+, R8

ADD.W @RS, R8

+2

► String: (tabela ASCII)

$0x41 \rightarrow 'A'$

$0x42 \rightarrow 'B'$

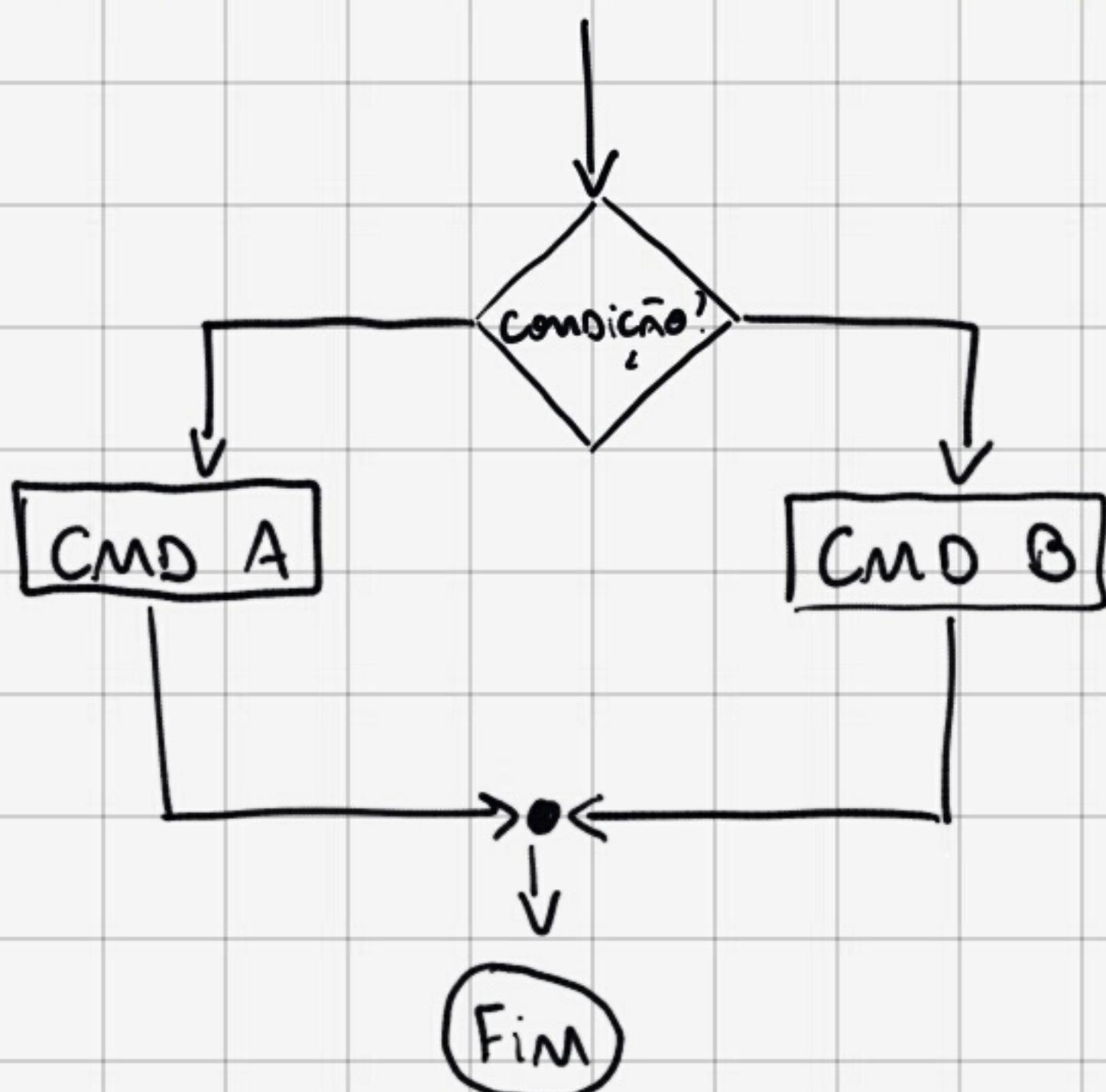
$0x00 \rightarrow '\backslash 0' \rightarrow \text{fim}$

MEMÓRIA	
ADDN	DATA
0x14	0x41
0x15	0x42
0x16	0x61
0x17	0x61
0x18	0x00

'A'  
'U'  
'I'  
'a'  
'\0' → fim

"AULA"

\* Estrutura de Controle de fluxo:



⇒ Instruções que permitem controle de fluxo: Saltos  
 → Argumentos de saltos são Labels

JMP: Salto Incondicional

JEQ, JZ JC JN	JNEQ, JNZ JNC
---------------------	------------------

→ JMP: Série de Fibonacci

$$DN = DN-1 + DN-2$$

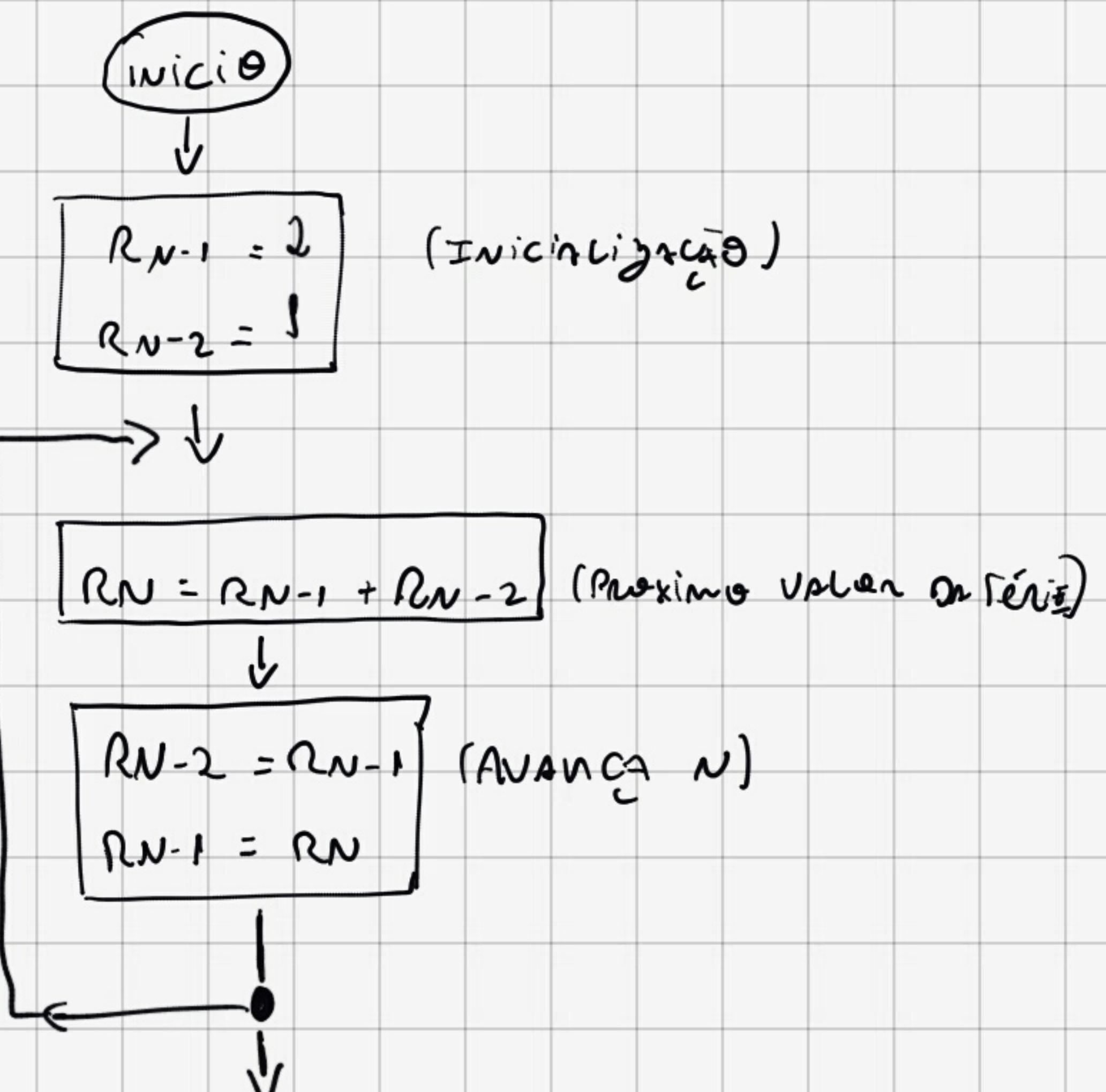
$$S = [1, 2]$$

$$F = [1, 2, 3, 5, 8, 13, \dots]$$

$$RN-2 : RS$$

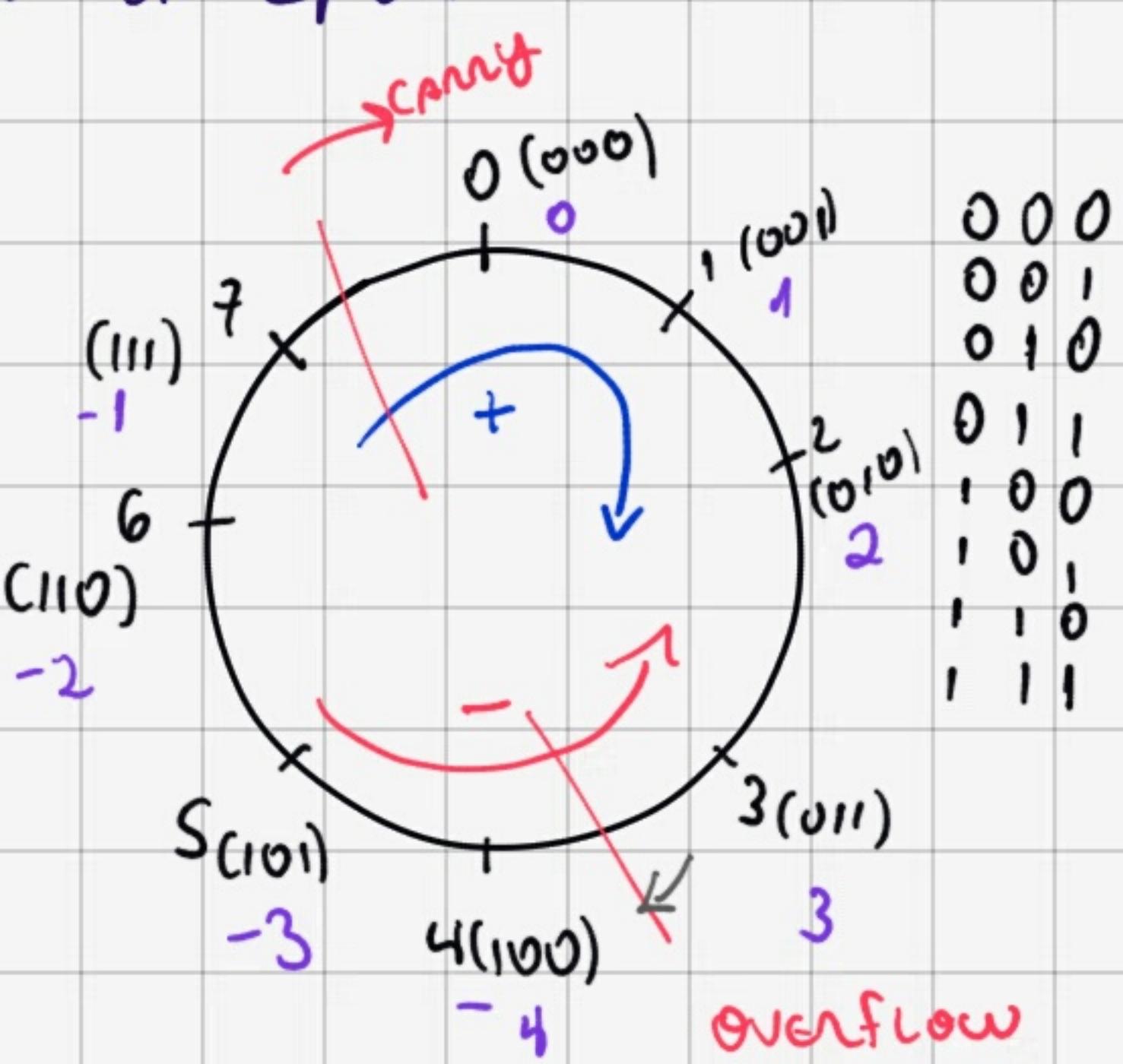
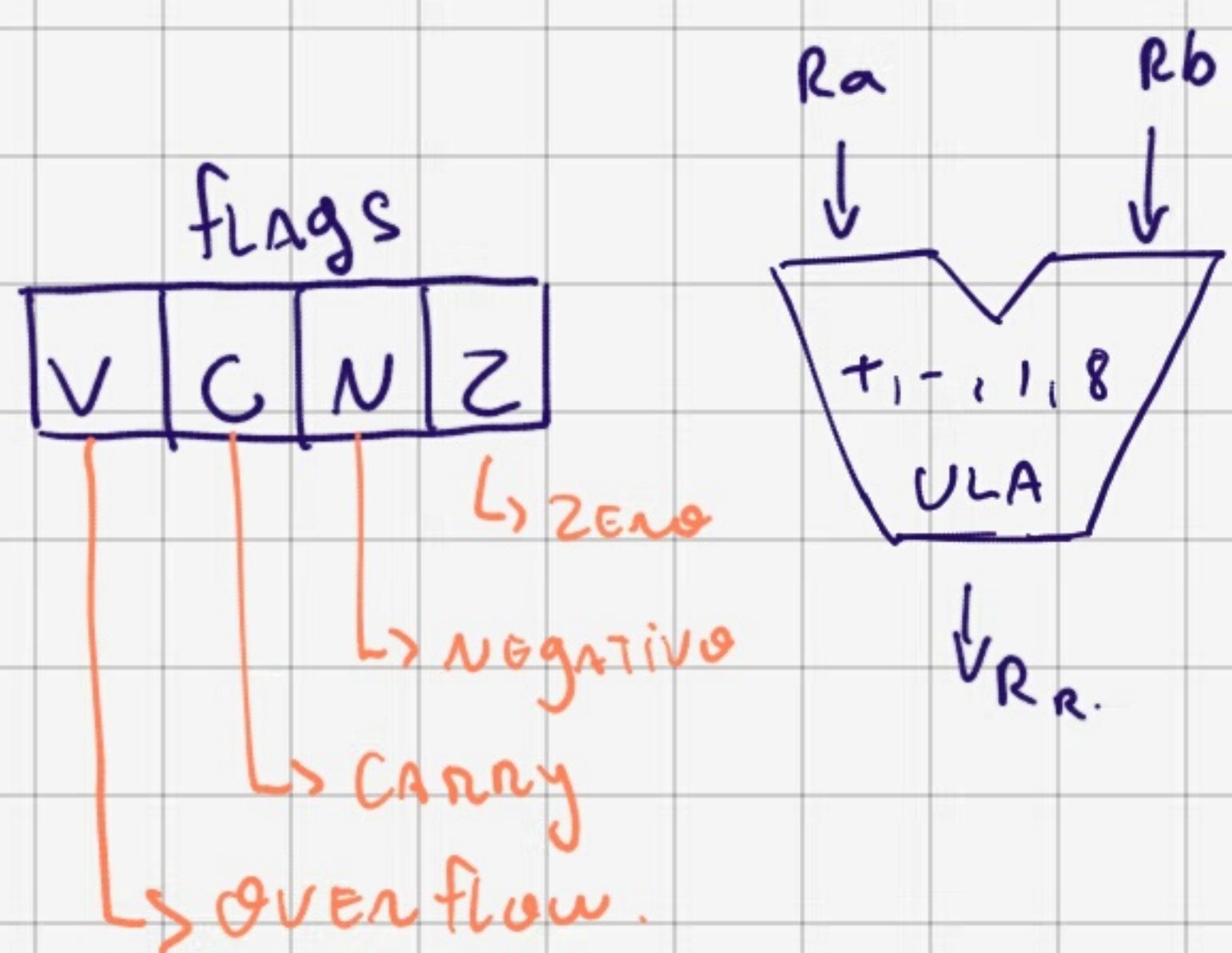
$$RN-1 : RB$$

$$RN : RZ$$

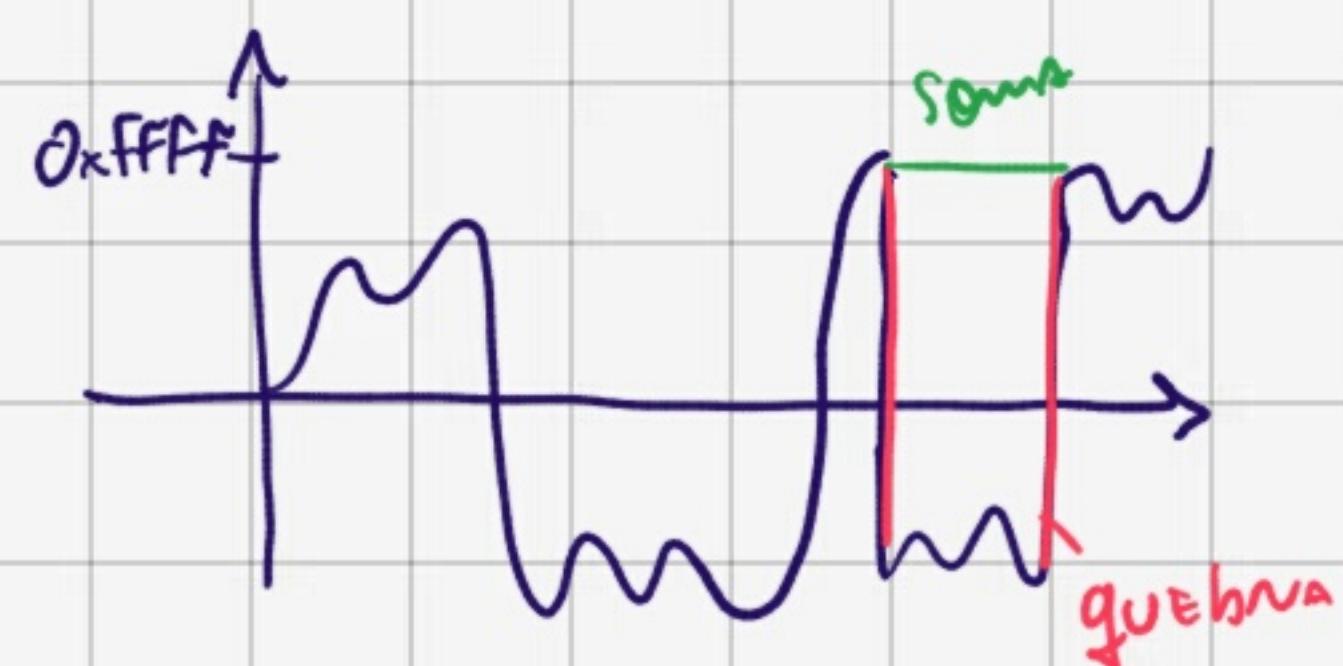


	13	21	4S	
Start				
	MOV.W #J, RS			; Inicialização
	MOV.W #2, R6			; S = [1, 2].
Loop:				
	MOV.W RS, R7			; calcula θ
	ADD.W R6, R7			; proximo valor
Next:				
	MOV.W R6, RS			; avança N.
	MOV.W R7, R6			
Fim:				
	JMP Loop			

→ Sintese condicionais utilizando flags da CPU:



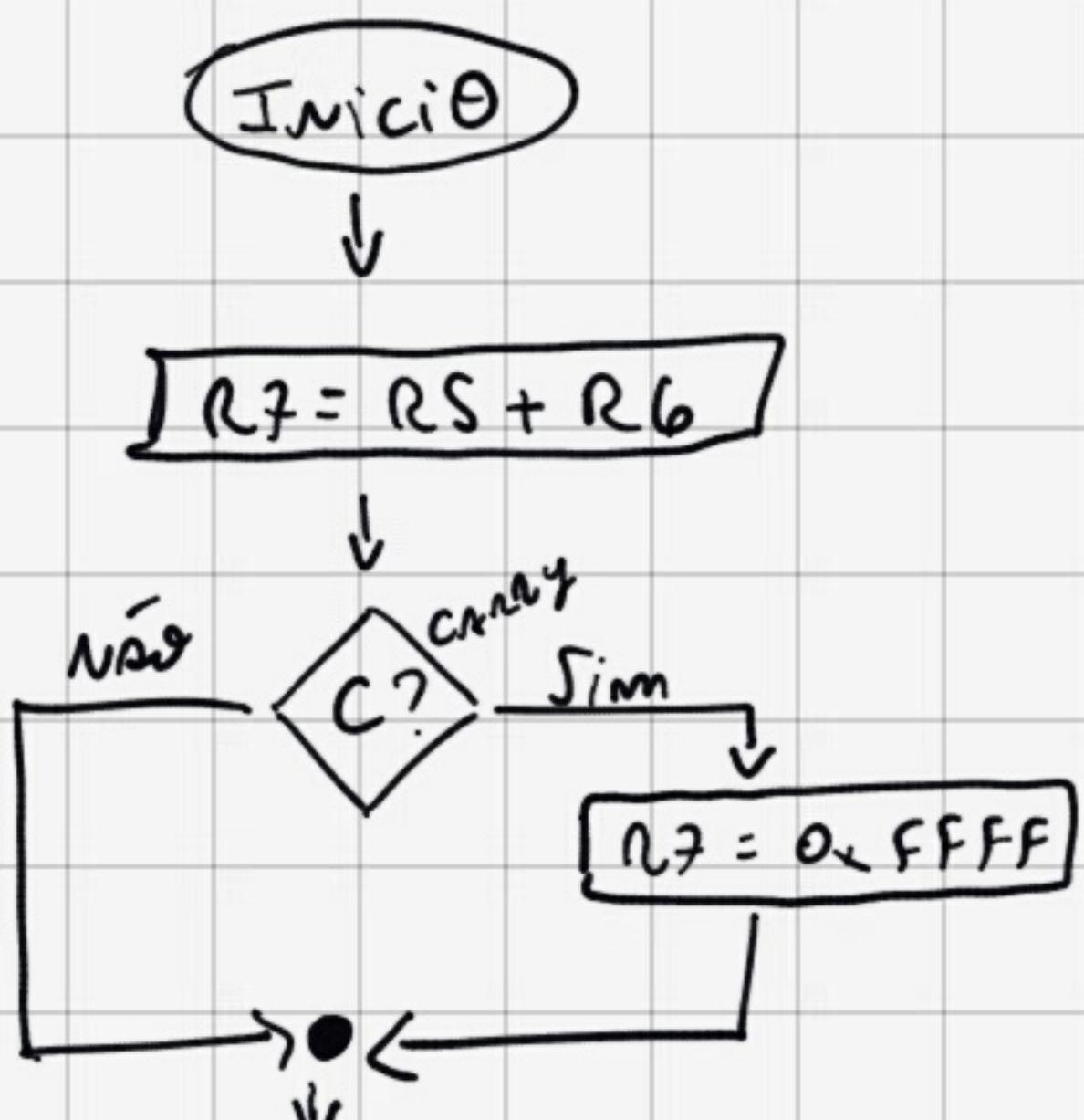
Ex: Soma c/ Saturação



"A soma de 2 números de sinal diferentes, nunca ocorre overflow!"

12h monitorizar

→ Somar R5 com R6, se houver carry, saturar em 0xFFFF:



	13	21	4S	
Start:				
	MOV.W R6, R7			; Soma
	ADD.W RS, R7			
	JNC			Fim
	MOV.W #FFFF, R7			; Se houver carry, saturar

## # Laboratório : Módulo 0/

### Ensaios

PE1:

```
MOV.B #3, RS ; Coloca o valor 3 em RS
MOV.B #4, R6 ; Coloca o valor 4 em R6
ADD.B RS, R6 ; Operação R6 = RS + R6
JMP $ ; Trata execução em um laço infinito
NOP ; Nenhuma operação.
```

Ensaios 3 : RS é carregado 0xFFFF

PE3:

```
MOV.B #0xFFFF, RS ; Coloca um valor hex. em RS
MOV.B #0x4321, R6 ; Coloca um valor hex em R6
ADD.B RS, R6 ; operação R6 = RS + R6
JMP $ ; Trata execução em um laço
NOP
```

Ensaios 4 :

PE4:

```
CLR RS ; Zera RS
MOV #4, RS ; Coloca 4 em RS
```

Loop:

```
CALL #SUBROT ; Chamar Subrotina "SUBROT"
DEC R6 ; Decrementar R6
JNZ LOOP ; Se diferente de zero, ir para Loop
NOP ; Nenhuma operação
JMP $ ; Trata execu. Em um caso de repetição
NOP ; Nenhuma operação
```

SUBROT:

```
ADD #1, RS ; Soma 1 em RS
ADD #1, RS ; Soma 1 em RS
RET ; Retorna
```

⇒ SUBROT : Soma 1 ao RS 2 vezes

→ O numero de vezes que a Subrotina é chamada é ditado pelo valor de R6 : R6 é o contador do loop

1) Mode Usando FS ; 2) Faça Soft Reset (CTRL + SHIFT + R) use F6  
\* Ler as próximas 4 páginas no próximo.

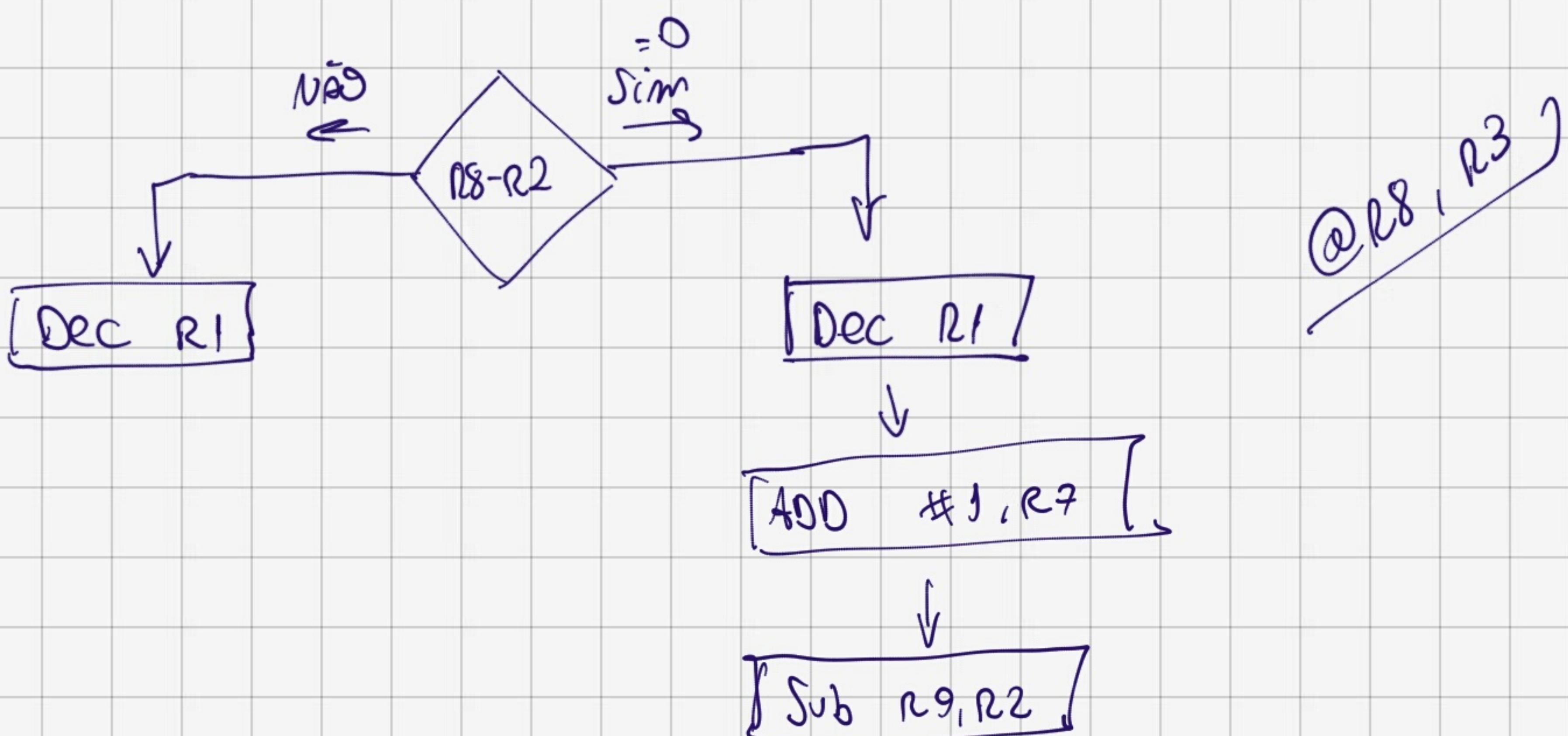
Ensaio S:

PES:      MOV      #1, RS  
              MOV      #4, R6

Loop:      RLA      RS ; Desloca 1 bit para Esqueras  
              DEC      R6 ; Decrementar R6  
              JNZ      ; Se diferente de zero, ir para loop  
              NOP      ;  
              JMP      \$ ; Iniciar Execucao num Loop Infinito  
              NOP      ;

12 → R1    0 → R7    R2 → \$1

R8 - R2  
Se for 0  
R7++      Se n for, R7 = DEC R7  
R9 = R2



## \* AULA 4: Assembly Avançado

- Testes
- Modos de engrenamento
- Comparação de números
- Pilha
- Subrotinas

{ Anteriormente: }

Instruções:

I {  
MOV  
ADD [C]  
SUB [C]  
BI [S/C/T]  
AND, XOR  
DADD

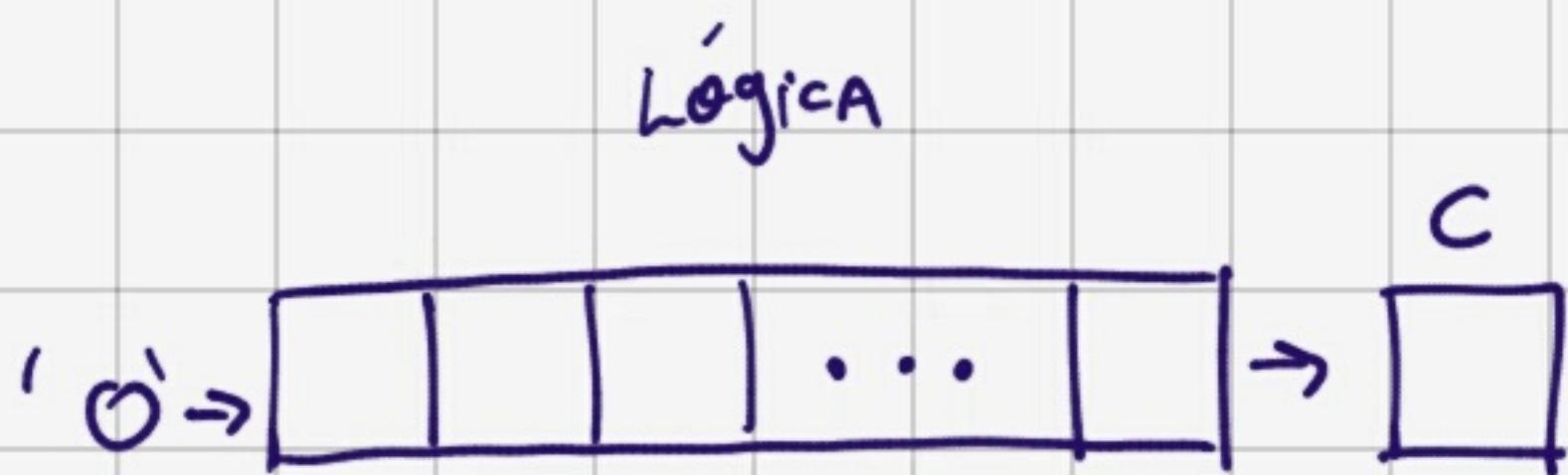
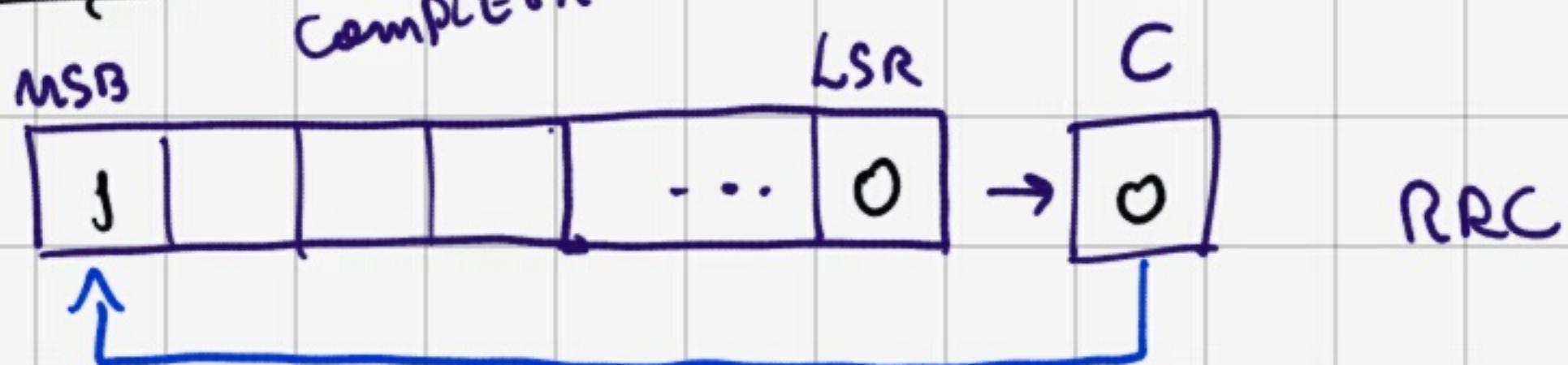
II {  
R [R/L][A/C]  
SWPB  
PUSH, POP, CALL, RET  
RETI

Saltos {  
JMP  
JC, JN, JZ  
JNC, JNZ  
JGE, JL  
JHS, JLO

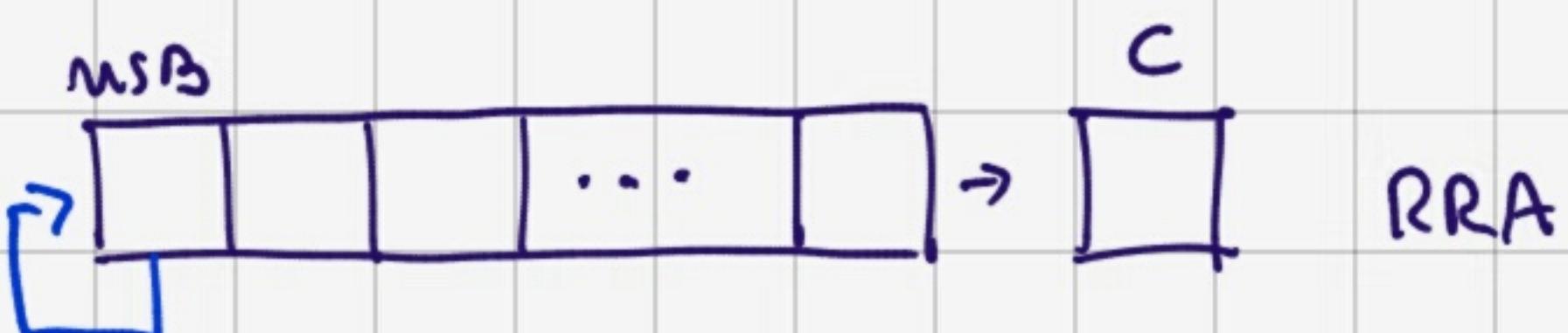
BIT #BIT3, RS  $\rightsquigarrow$  res. vai p/ o carry

DADD #0x39, R7 ; R7 = 0x03  
0x42

Rotações:



Aritmética



## Comparações de números

A > B ?

$A - B \left\{ \begin{array}{l} \text{RESULTADO Positivo} \Rightarrow A > B \\ \text{RESULTADO NEGATIVO} \Rightarrow A < B \\ \text{ZERO} \Rightarrow A = B \end{array} \right.$

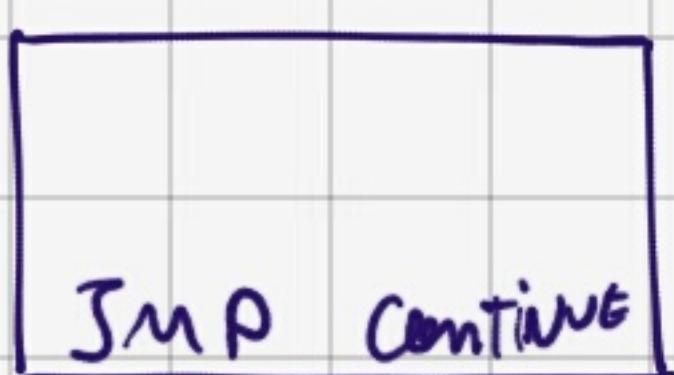
Sub.w      R4, RS  
 CMP.w      R4, RS → NEF

### Números Sem Sinal:

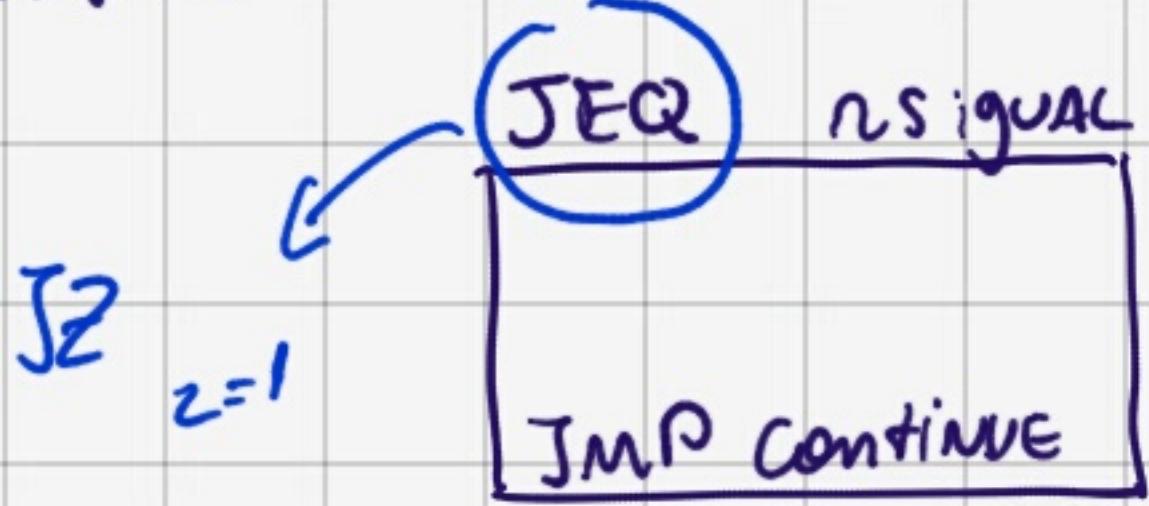
JHS, JLO

→ CMP #0,20,RS  
JHS RS maior

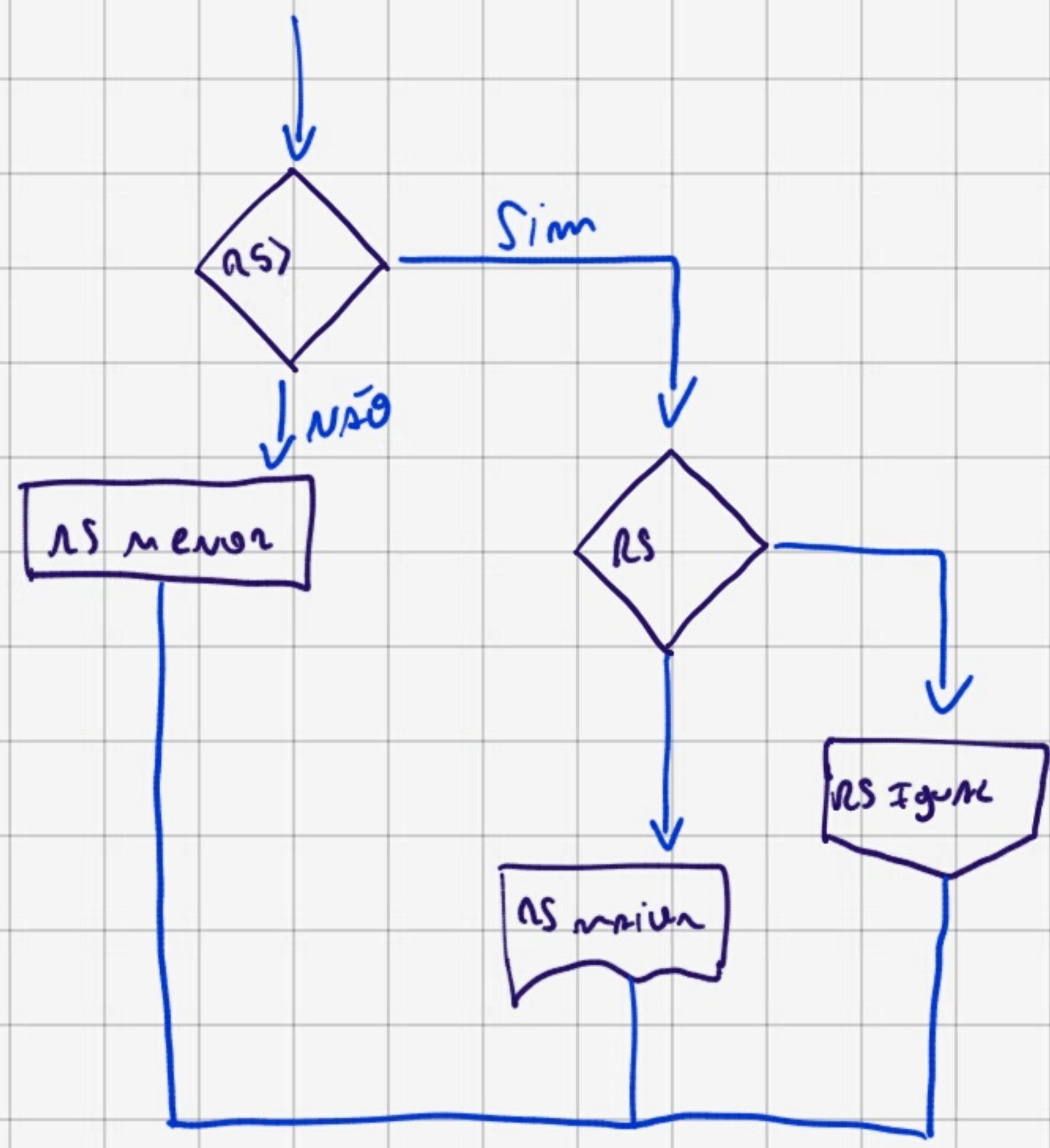
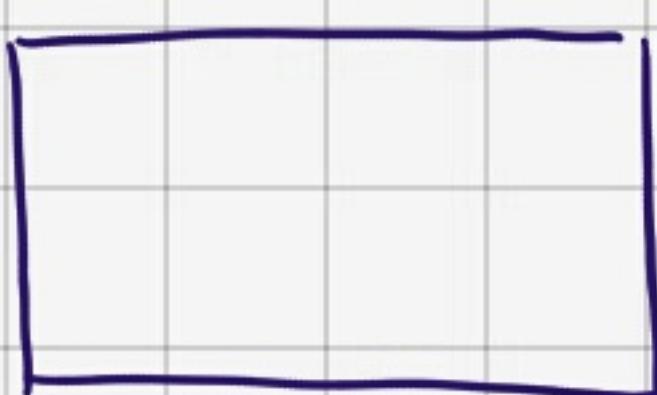
RS menor:



RS MAIOR:



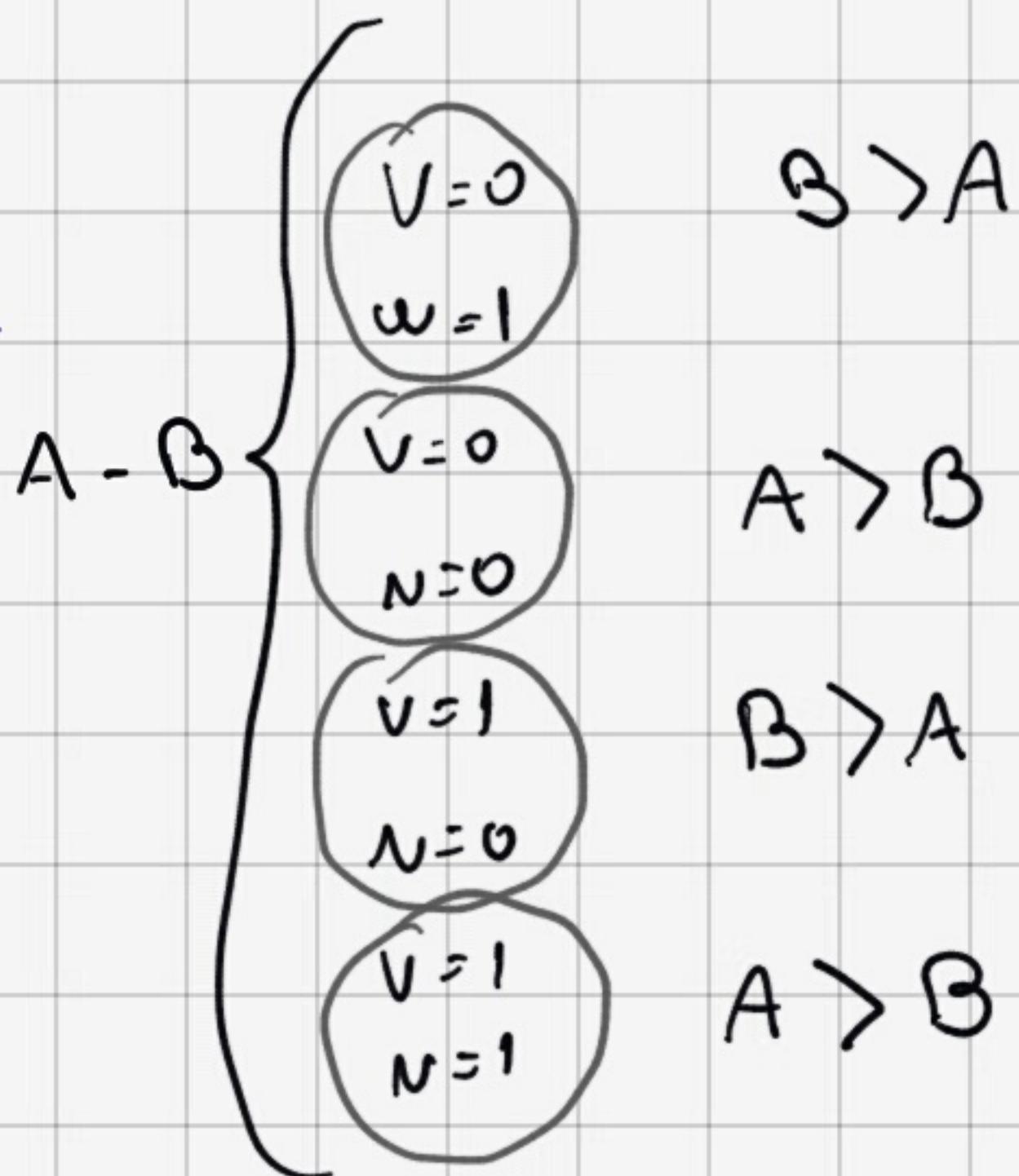
RS IGUAL:



### Para Números Com Sinal:

JGE → jump if Greater or Equal

JL → jump if Less.  
↳ verifica a flag V/N



Exemplo: Comparar R5 com R6 se RS for maior  
Dividir R6 por 2 até RS ficar menor.

R5 e R6 são números sem sinal

Loop:

CMP.W

JHS

R6, RS *referência de comparação*

fim

ns memori:

RRA

JMP

R6 ;  $R6 \leftarrow R6/2$

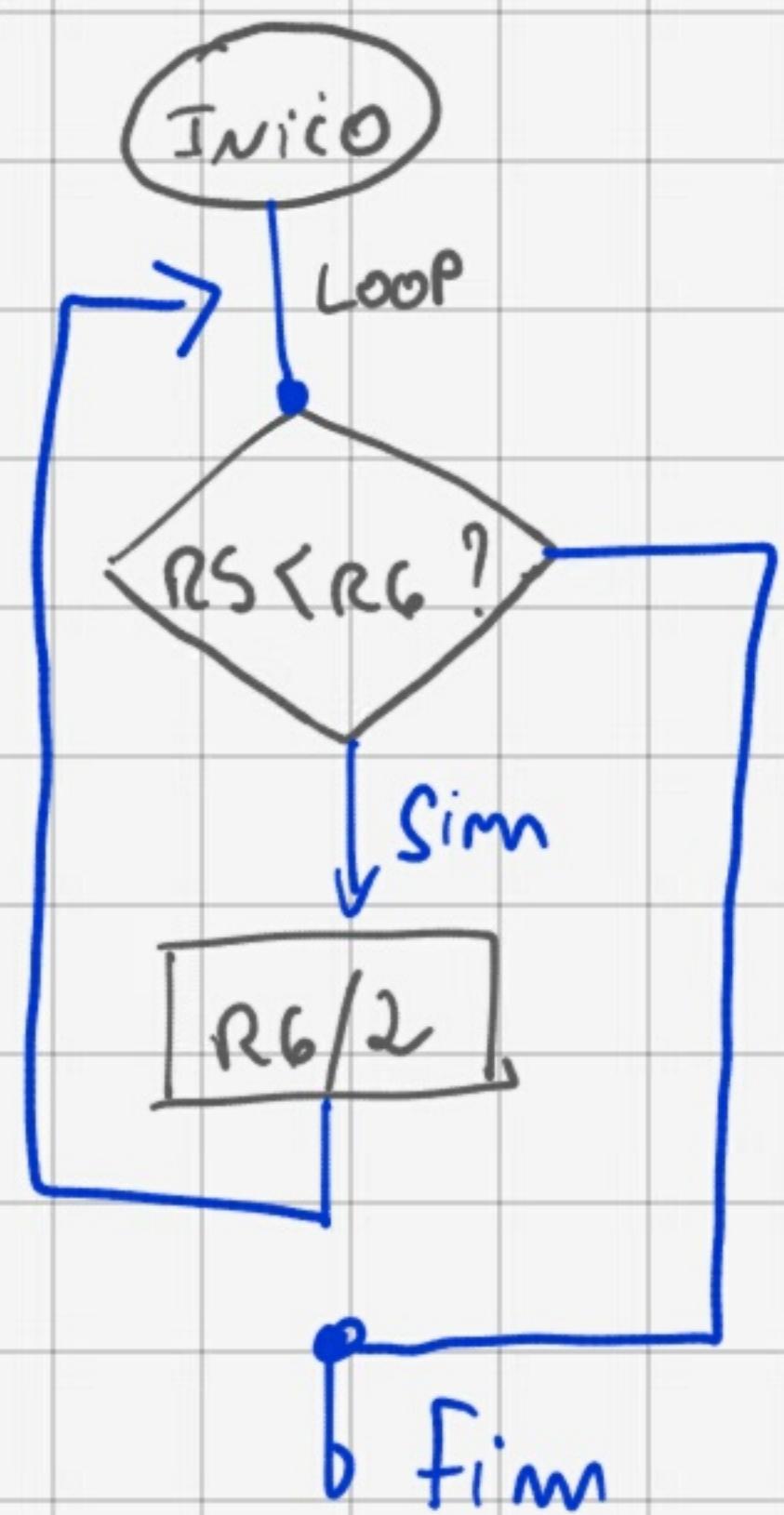
loop

fim:

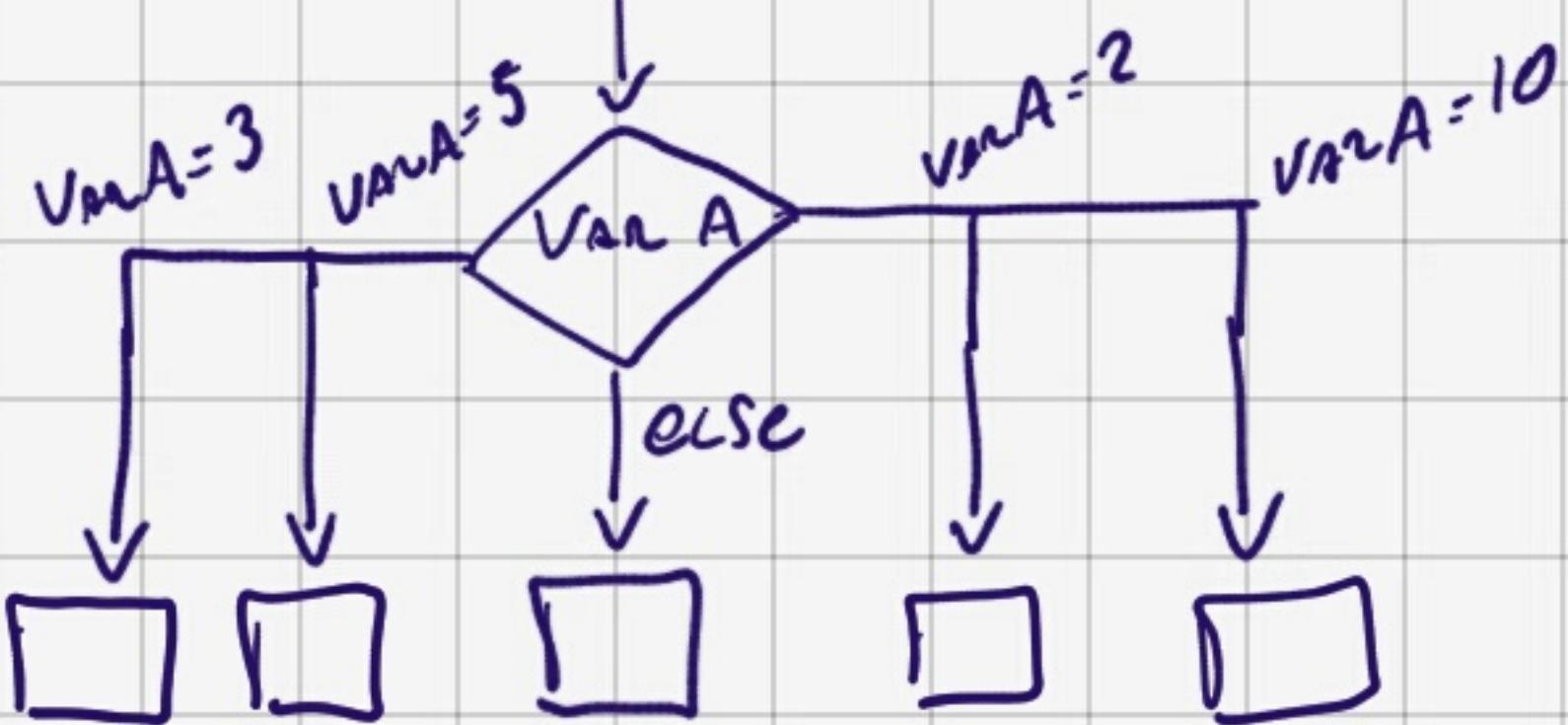
→

JMP

\$



## Multiphas Escritoras



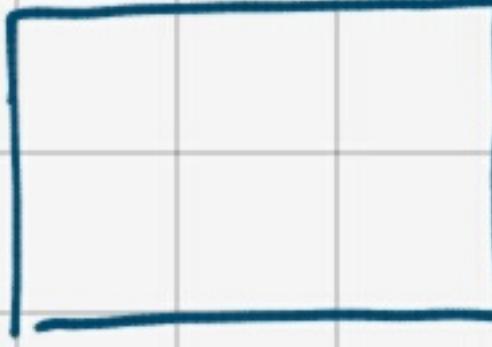
⇒ [ ADD  
JMP  
JMP  
JMP  
JMP ]  
VarA, PC  
VarA zero  
VarA dois  
VarA quatro  
:  
eise

"ESTRUTURA DE DADOS"  
"em Interrupções"

VarA zero :



VarA dois :



	ADDR	DATA
B	2400h	16
B	2401h	0
W	2402h	8
	2403h	3fh



VarA memoria.RAM | 0x2400

• dat

Little Endian:

ADD N → LS Byte

ADD N+1 → MS Byte

- bytes 16, 0

- words 0x3f08

## \* Modos de Endereçamento

- Registro → MOV RS, R6
- Simbólico → ADD VarA, PC *VarA, PC* *REG* *simb*
- Absoluto → ADD \$VarA, PC
- INDEXADO → MOV O(RS), R6
- INDIRETO → MOV @RS, R6
- INDIRETO C/AUTO-INCREMENTO → mov @RS+, R6
- INMEDIATO (constante) → ADD #1, RS

RS	12h
ADDR	DATA
10h	S2
11h	37
12h	8A
13h	
14h	AB
15h	74
16h	CD

## \* Pilha (last in, first out: LIFO)

R6 → Stack Pointer (SP)

HOS → Head of STACK

TOS → top of STACK

R6 [000F]

R7 [3421]

RAM	
ADDR	DATA
43FC	0Fh <i>→ SP</i>
43FD	(xxh)
43FE	21h
43FF	34h <i>---&gt; SP (anterior)</i>
4400	

POP.B R6

POP.B R7

PUSH R6 → SP-2

PUSH.B #0x0F → SP-2

*↳ PUSH*

Não zero! (é o único comando que não zero)

Exercício: (resposta no Cap 6 do IDE grava)

Como as instruções CALL e RET usam a PILHA??

0	1	2	3	4	S
5	7	2	3	9	1

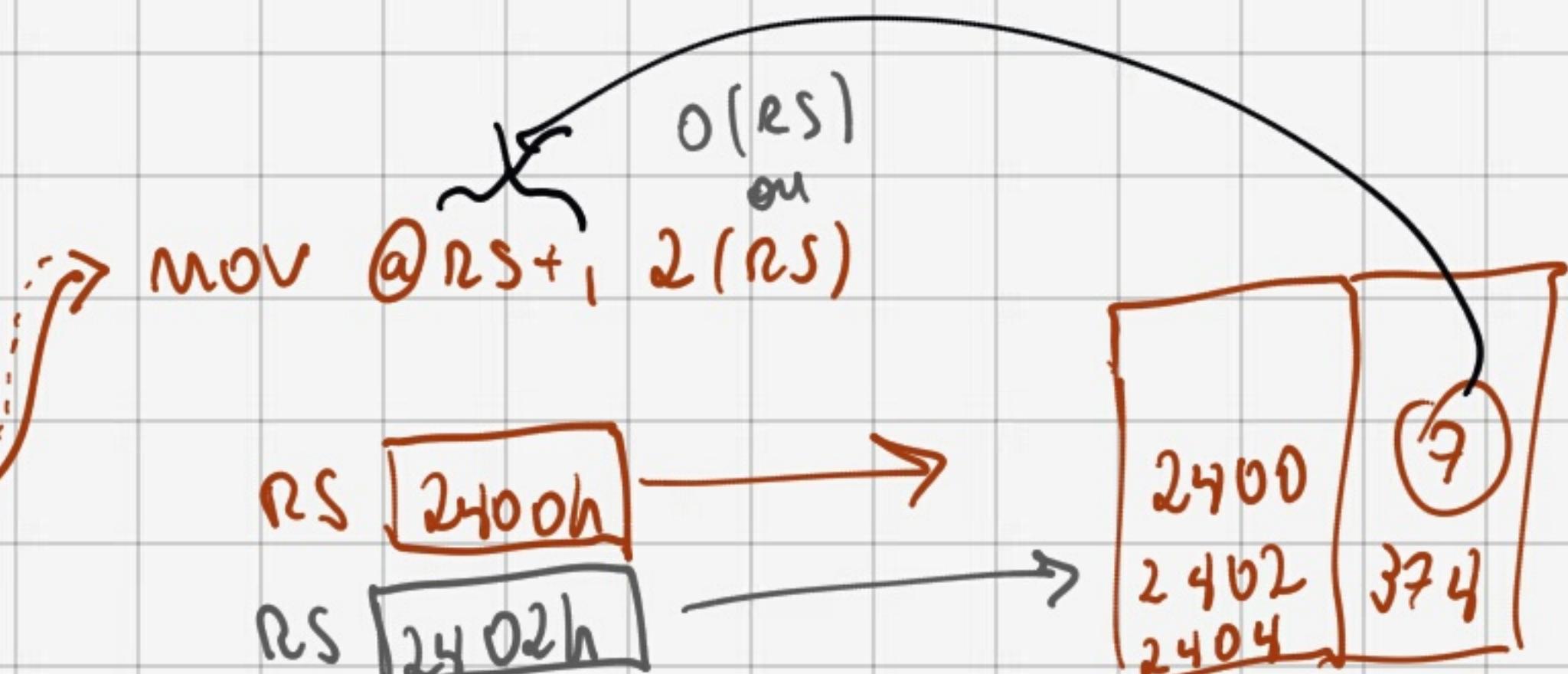
MOV #Vector, RS  
 CALL #onDenA  
 Jmp J  
 NOP

onDenA MOV @RS, 0x04

.data  
 .byte 5, 7, 2, 3, 9, 1

• teste

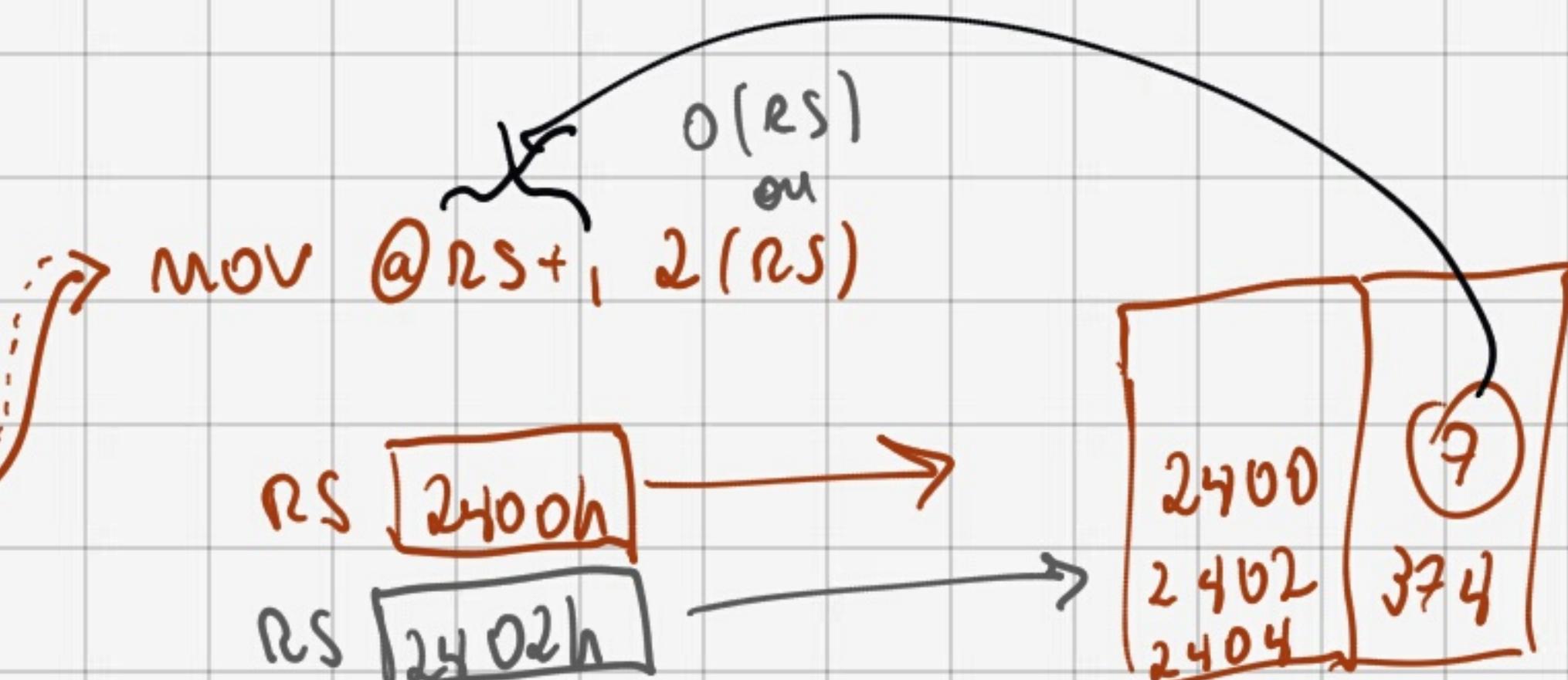
MOV.W #Vector, RS  
 MOV.W @RS+, R6  
 MOV.W @R6, R7



vector: .word 1000|0000|0001|0000  
 (BITF/BIT4), 0xFOFO, 1024  
 IJJJ  $2^{10} = 1K$   
 0x8030

Problema 1:

MOV.W #Vector, RS  
 MOV.W @RS+, R6



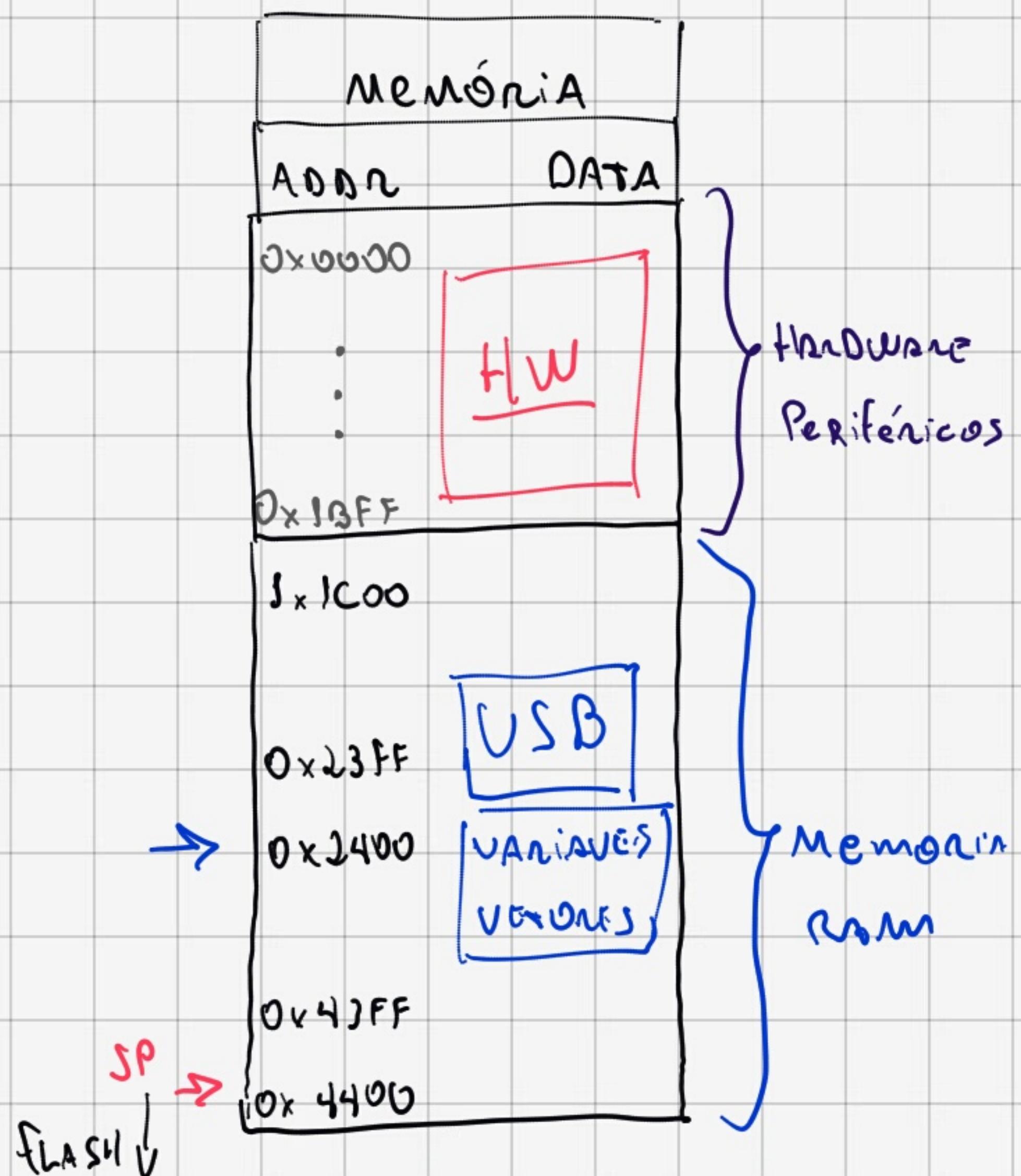
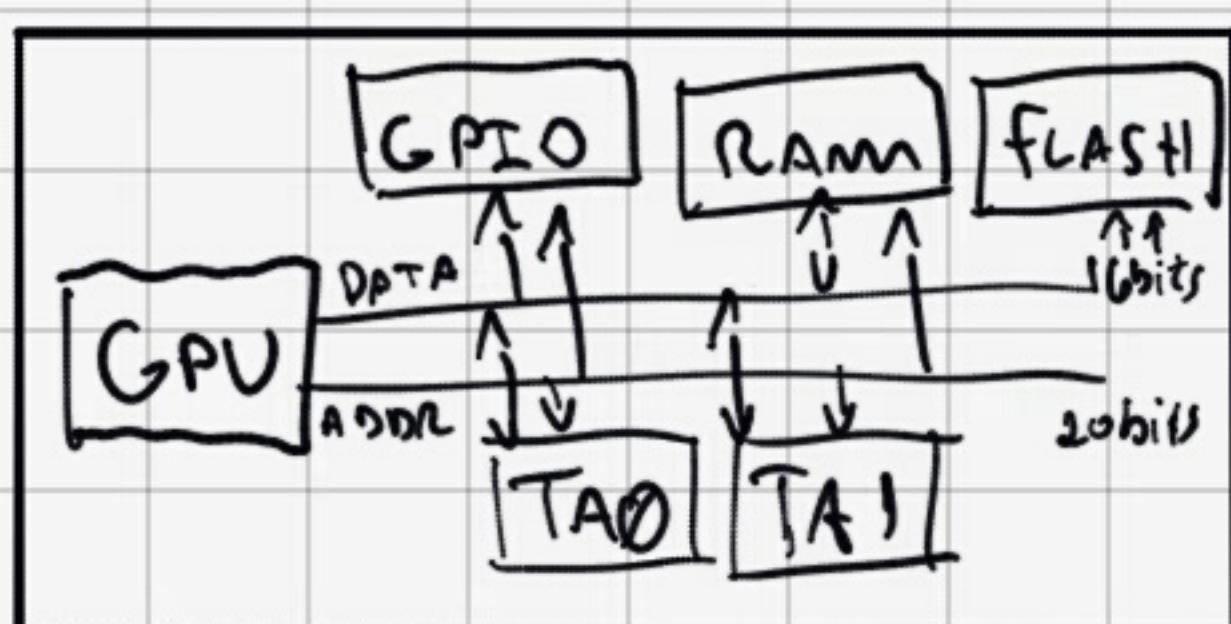
J2, 56h, 49h, 43h, 54h, 4Fh, S2h, 44h, 41h, 4Eh, 49h, 4Sh, 4Ch  
 V I C T O R D A N I E L

⇒ Acesso ao vetor ⇒ memory Browser ⇒ Executa o indexado de inicio  
 ⇒ "8-bit hex - C style" (0x02400)

## X AULA 5: Arquitetura do MSP430

- Organização da Mem
- Subrotina
- Registros Especiais
- Formato das Instruções

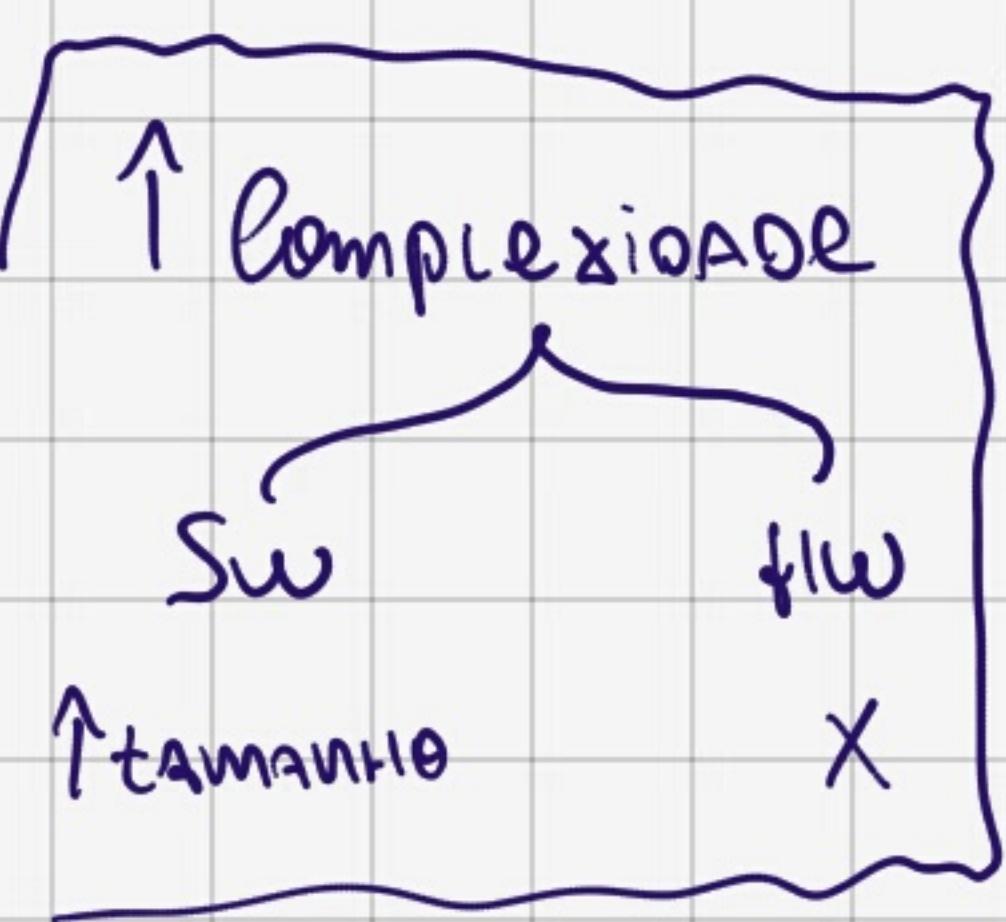
### X Organização da Mem.



#Subrotinas: • Permitem a reutilização do código.

### 1) Instruções

- CALL → RUSTI (PC+2)
- RET\* → BR 8ADDR



RET: Instrução emulada

RET = POP PC

main:

```

MOV.W    # VETOR, RS
CALL
JWP
    
```

# Subrot → X  
\$ → X  
JWP → X

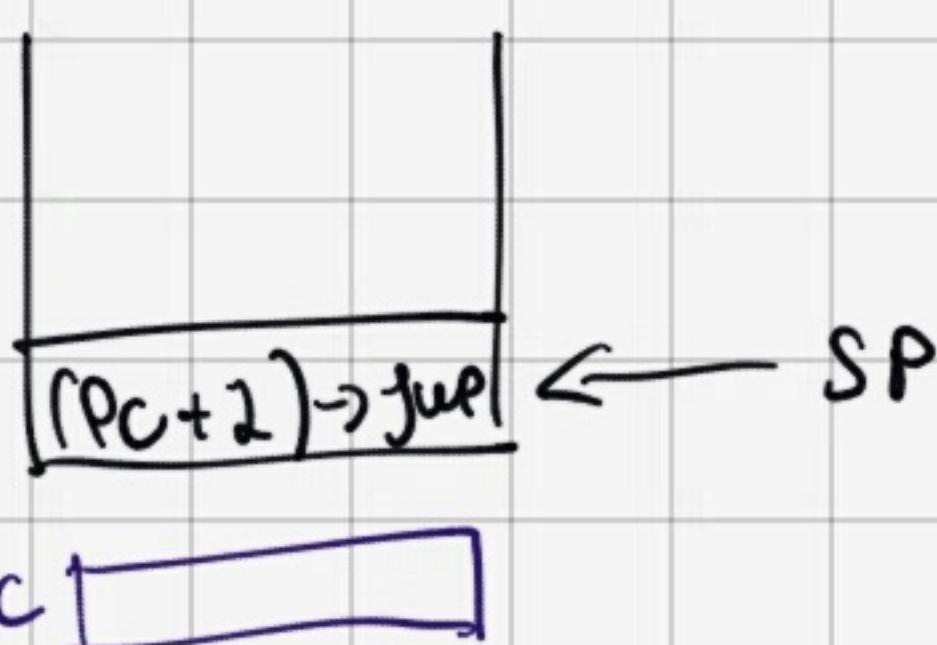
Subrot: → BIC  
(PC) → :  
②      :  
RET

# BIT3, RS → ✓

"Só realiza saltos  
no 16 bits"

→ Não realiza saltos para fora da Subrotina

Pilha:



## \* Exercício: Multiplicar 2 números:

$$n = \text{mult} 8(a, b)$$

$a, b$ : bytes de entrada S/Sinal

$n$ : resultado em - Bits

$$\hookrightarrow a \times b = n$$

(n bits)      (n bits)      (n+n bits)

$$2 \times 5 \left\{ \begin{array}{l} S+S = \\ 2+2+2+2+2 = \end{array} \right.$$

## \* Passagem De Parâmetros:

Convenções

→ Por Registro

→ Por posições pré-definidas no memo.

→ Pela Pilha

⇒ Por posições pré-definidas ...	<table border="0"> <tr> <td>VAN A</td><td>.Set</td><td>0x2400</td></tr> <tr> <td>VAN B</td><td>.Set</td><td>0x2402</td></tr> <tr> <td>RES 16</td><td>.Set</td><td>0x2404</td></tr> </table>	VAN A	.Set	0x2400	VAN B	.Set	0x2402	RES 16	.Set	0x2404
VAN A	.Set	0x2400								
VAN B	.Set	0x2402								
RES 16	.Set	0x2404								

## \* Pela Pilha:

→ PUSH1 Arg. 2  
 PUSH1 Arg. 2  
 PALL #mult 8



## Mult 8:

MOV.B	4(SP), RS	$\downarrow^a$
MOV.B	2(SP), R6, b	
CLR	R9	
CMP.B	R6, R5	
JMP	mult8-ameron	

tos  
(SP) → PC + 2(010)  
 arg 2 = b  
 arg 1 = a

## Mult8-ameron:

MOV.B	RS, R8
MOV.B	R6, R7
JMP	_____

R7: Iterador

R8: ACC

mult8 - answer:

MOV.B	R6, R8
MOV.B	R5, R7

## multiloop:

ADD.W R8, R9 ; R9 = R9 + R8  
 DEC R7 ; R9 += R8  
 JNZ mult8-Loop  
 MOV.W R9, 4(SP)  
 MOV.W 0(SP), 2(SP)  
 INCQ.W SP  
 RET  
 MOV.W @SP+, 0(SP)

Register Substitution:  
 TOS → PC+2 (0:0)  
 (SP) → RETurn

Return Statement:  
 PC+2 (0:0)  
 RETurn

# \* Registros Especiais

→ MSP430 Rosjui 16 registros de ujo genérico, entre tanto 4 son especiales.

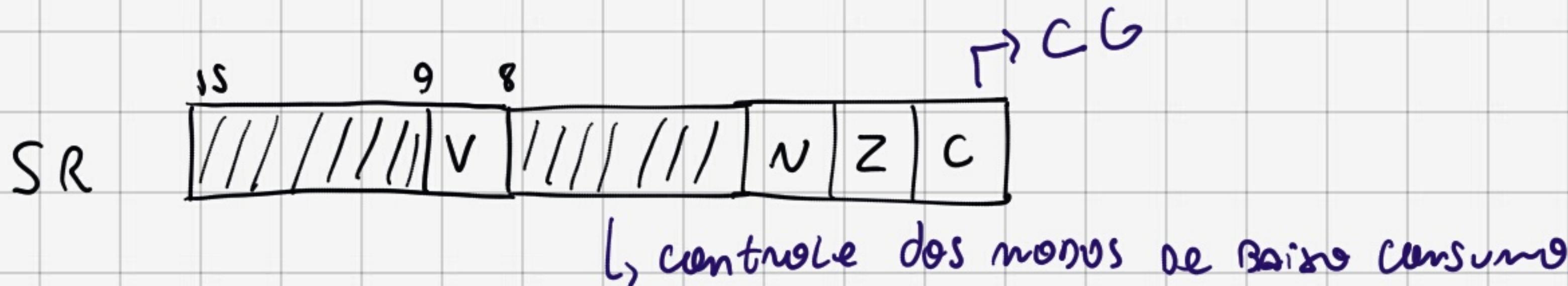
The diagram illustrates the mapping of four registers (R0, R1, R2, R3) to specific program components:

- R0 → PC**: Register R0 maps to the Program Counter, which is currently at **0**.
- R1 → SP**: Register R1 maps to the Stack Pointer, which is currently at **0**.
- R2 → SR**: Register R2 maps to the Status register.
- R3 → CG**: Register R3 maps to the Constant Generator.

→ na instrução CALL → guarda PC (16 bits)

∴ PC, SP Sempre Incrementa de 2

↳ INSTRUÇÃO de 16 bits



# GG: Geraden de constante

CONSOLIDATION

- Clear (C)
  - Incremento (+1) (+2)
  - Decremento (-2) (+2)
  - MOV
  - ADD

→ Emulação no RISC

CLR  $\Rightarrow$  MOV #0, Rm

INC → ADD #1, Rm

\*tarefa: 1) Ler o cap. do user's Guide sobre o mpy32

2) ESCREVA uma tabela com todos os valores de constantes geradas pelo CG → 2 em quais modos de ensinamento ELAS SÃO geradas.

teste Ordena:

```
MOV.b #VETOR, R5  
CALL #Ordena  
JMP $  
MOF
```

COLoca Em R6 O VALOR

contendo no indice 1

o VETOR (O tamanho do VETOR)

Ordena:

```
MOV.b $1, RS  
MOV.b VETOR(RS), R6
```

0	1					
12	V	i	o	.	.	L

$$R6 \boxed{0x56} = V$$

12, 56h, 49h, 43h, S4h, 4Fh, S2h, 44h, 41h, 4Eh, 49h, 4Sh, 4Ch  
V I C T O R D A N I E L

⇒ Acesso ao VETOR ⇒ Memory Browser ⇒ Encontra o indice de inicio  
⇒ "8-bit hex - C style"

JHS JLO → Sem Sinal (C)

JGE/JL → Com Sinal (N XOR V)

## \* AULA 6:

- Detalhes da Arquitetura do MSP430
- Periféricos dedicados
  - ↳ MPY 32
  - ↳ GPIO

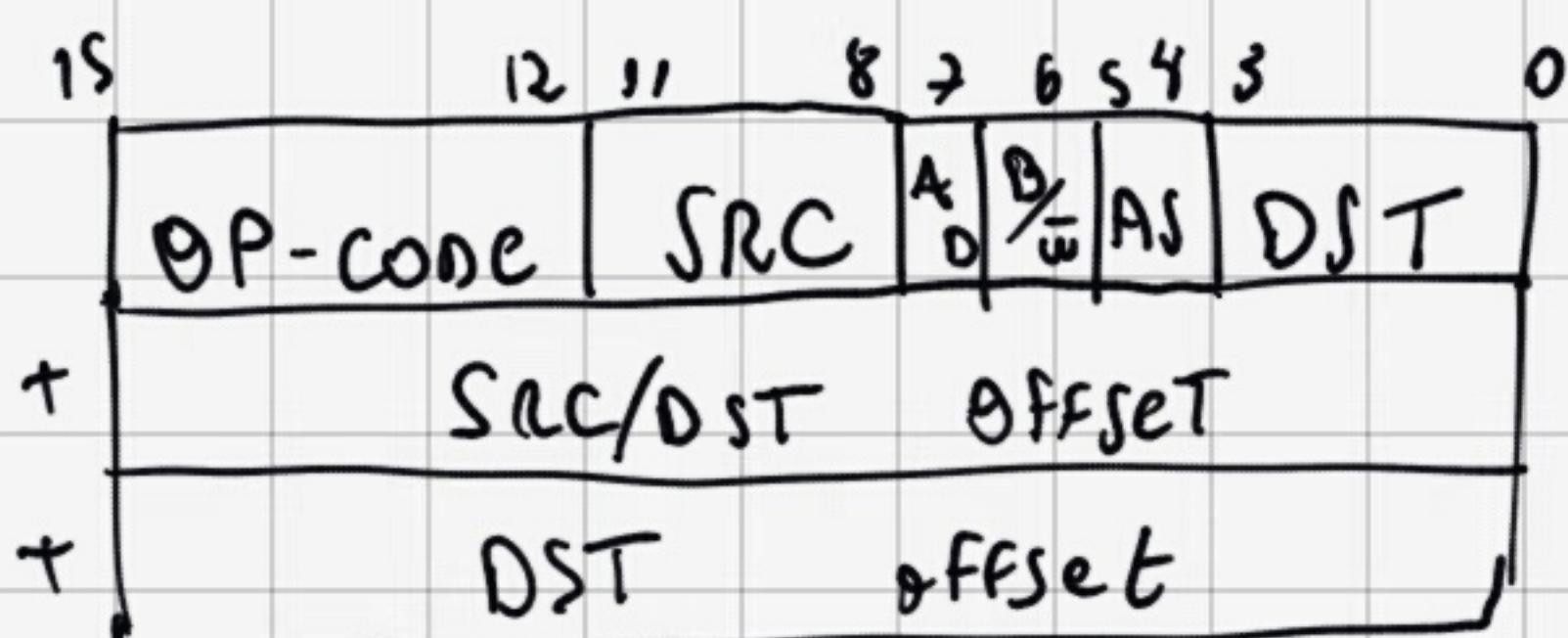
} . Gerador de cts  
 } . formato das INST  
 } . Pipeline

## \* formato das instruções

→ formato I (dois operandos)

Ex: MOVW @RS, R7

$R = 1$   
 $B/W = 0$



16-bits

16-bits

16-bits

0x4SE7

AD	B/w	AS
1	1	10

0x0003

A   
 0 - register  
 1 - indexado (Simbólico Absoluto)  
 2 - Indireto (@)  
 3 - C/Auto-Incremento

## \* modos de Indexação:

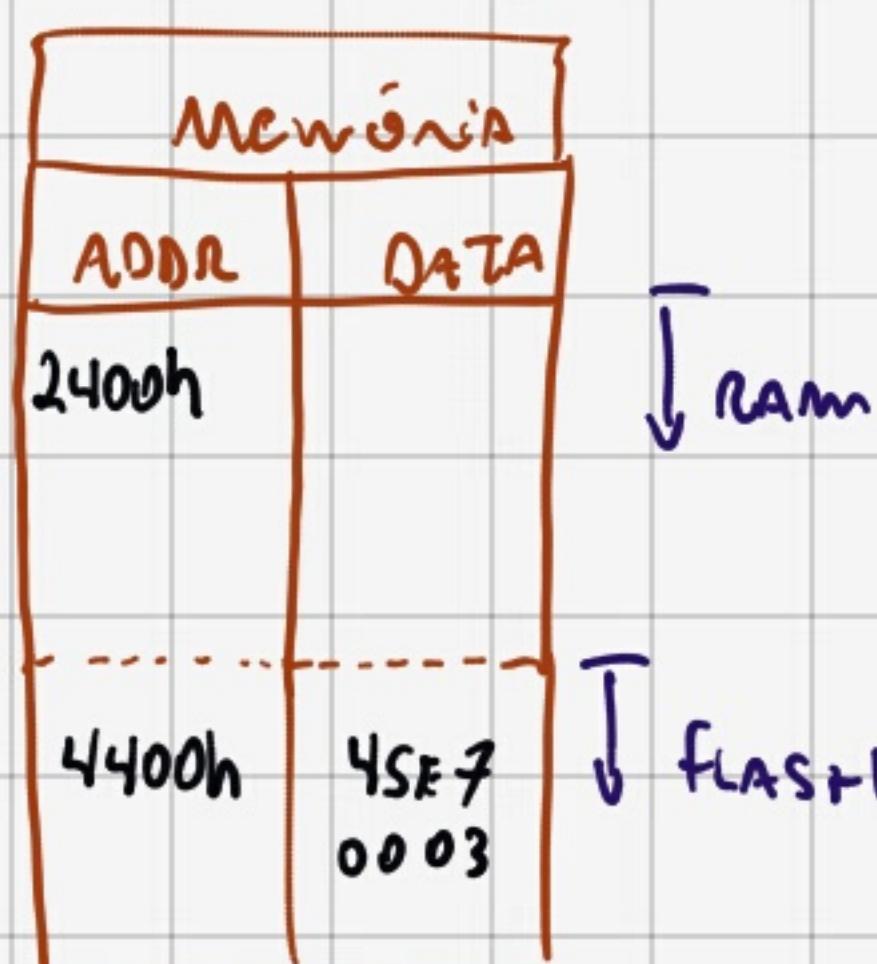
AS	Ad	modo de end.	Sintaxe	R2	R3
00	0	Register	Rn	SR	0
01	1	Indexado/Abs/Simb	01(Rn)/8var/vídeo	0	+1
10	-	Indireto	@Rn	+4	-2
11	-	Ind. c/auto-incr	@Rn+	+8	-1

\* Absolute MOVW &varA, R5  
"0x2408(R2)"

"9 wordands de 2 em 2"

.data  
 vtor .word 3, 1, 2, 3  
 varA .word 373  
 (2408h)

## \* Simbólico



RAM

FLASH

→ movw var, R...

## \*Immediato (Constante)

INC.w NS  
ADD.w #1, RS  
@PC+, RS

OP-code	SAC	AD	g <sub>RS</sub>	A <sub>S</sub>
5	0	0 0	11	5
0	0	0		1

0x5035

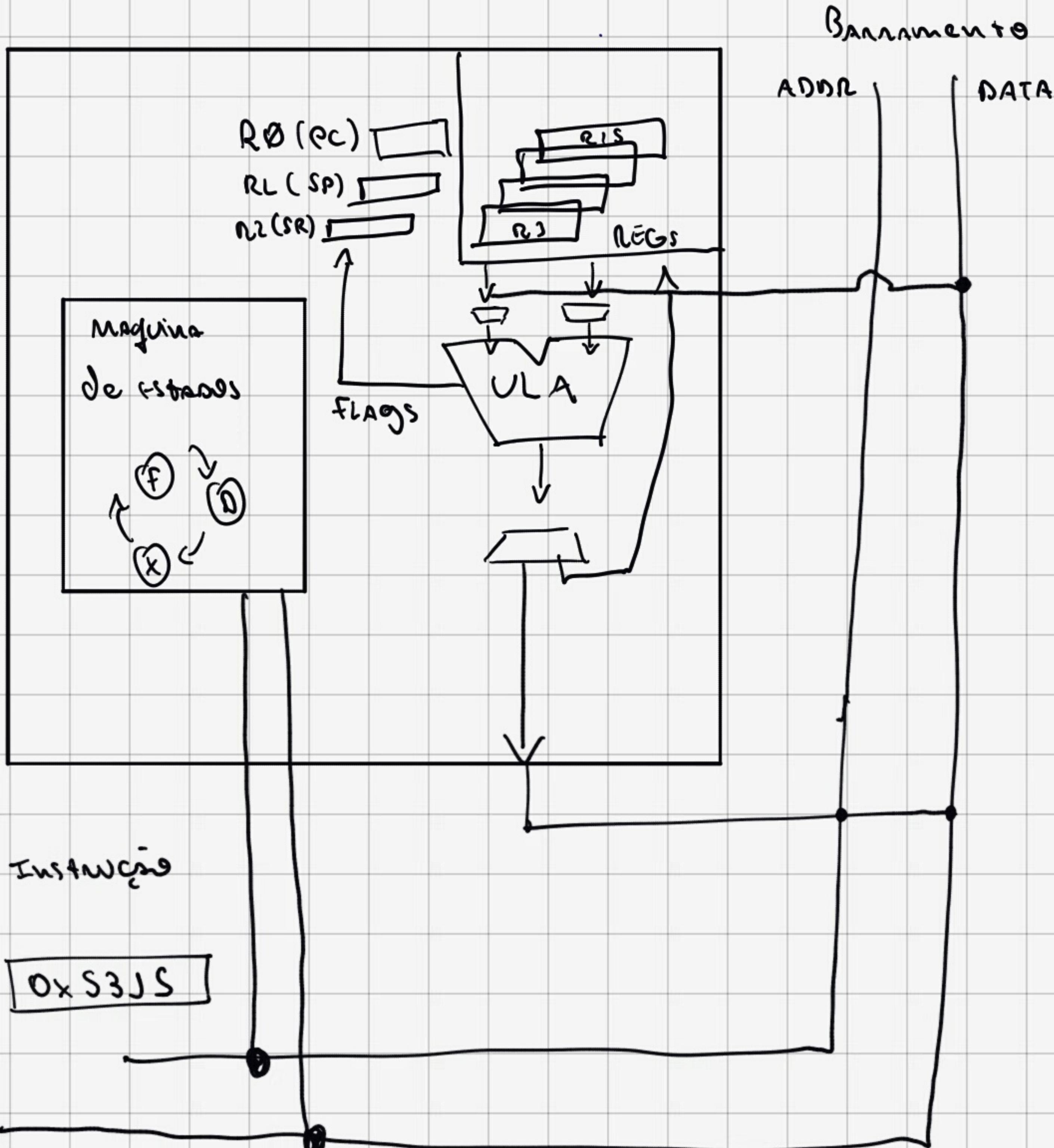
0x0001

ADD.w #1, RS

5	3	0	0	0 1	A <sub>S</sub>	5
5	3	0	0	0 1		5

0x5315

## \*funcionamento da CPU



### 1. Fetch (busca)

→ Puxa a instrução da memória para futura decodificação

### 2. Decode

→ Decodifica a instrução e prepara as entradas da ULA para Execução

### 3. Execute

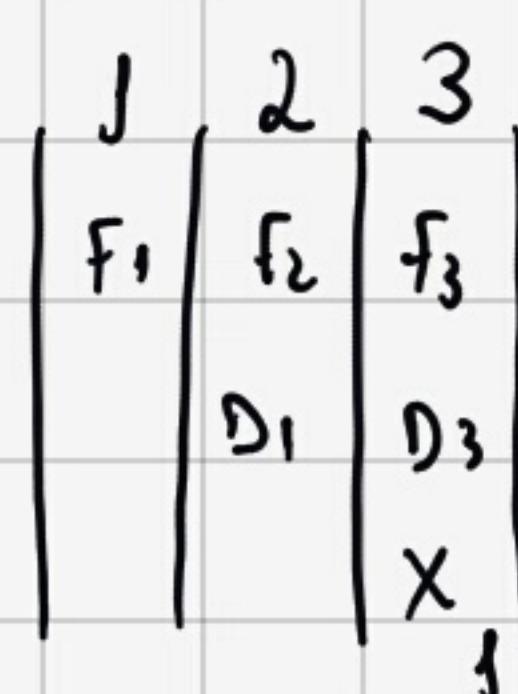
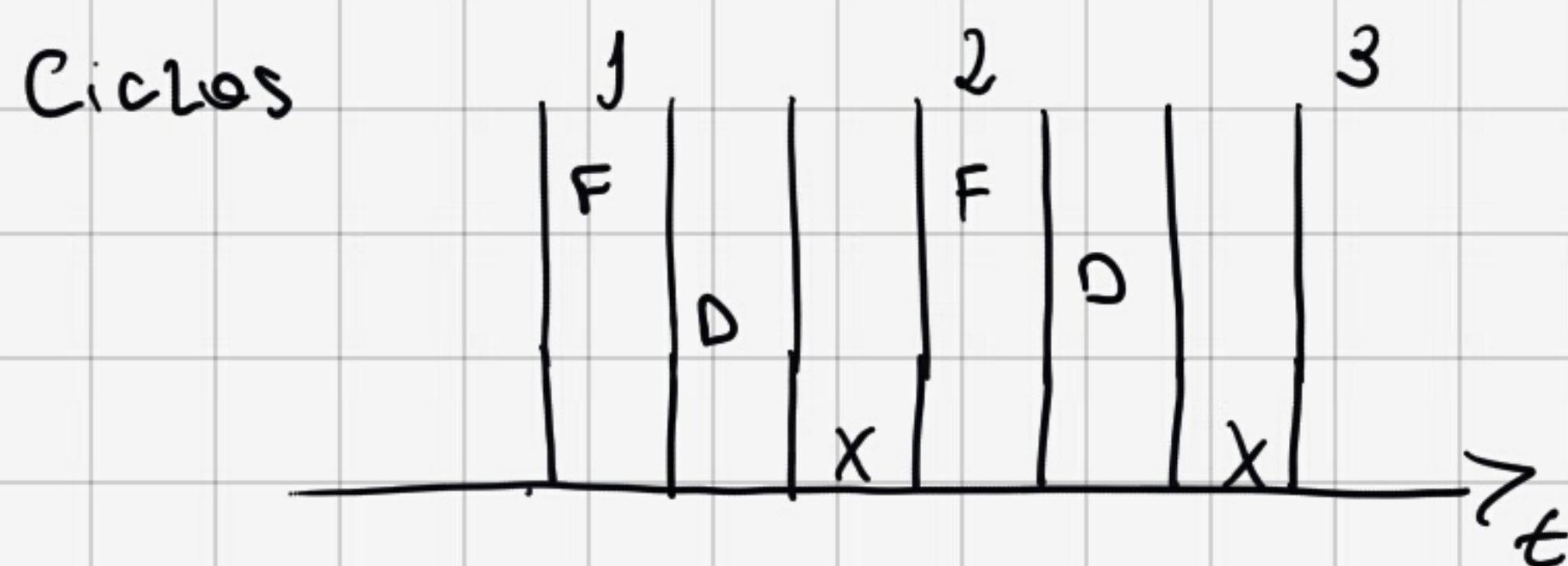
→ Realiza a operação

### 4. Save

→ Salva o resultado

## \* Pipeline

⇒ Crom Pipeline



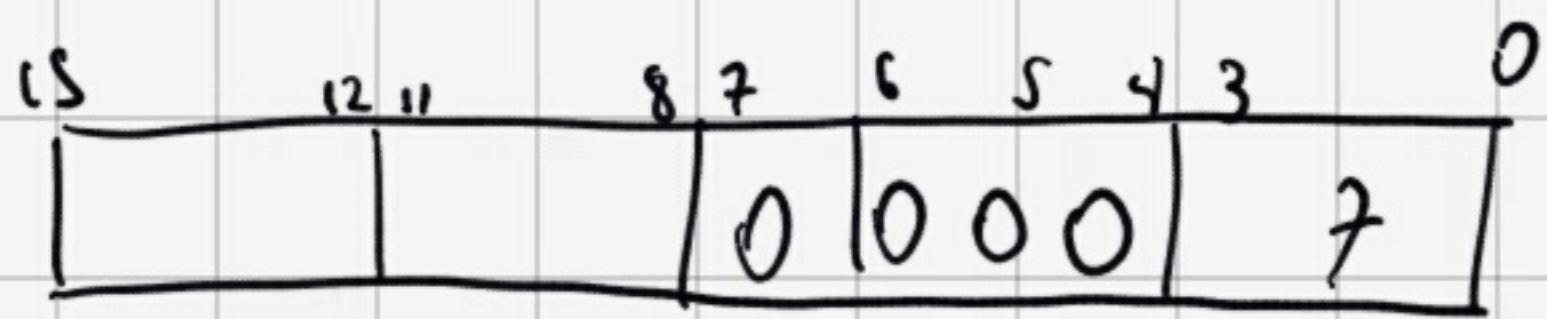
## \* Uso de buffer - no pipeline

- ① MOV R6, R7 → F<sub>1</sub>  
- ② ADD #34, R7 → F<sub>2</sub>, D<sub>1</sub>  
③ JC label → F<sub>3</sub>, D<sub>2</sub>, X<sub>1</sub>  
④ NOP → F<sub>4a</sub>, D<sub>3</sub>, X<sub>2</sub>  
⑤ NOP → F<sub>4b</sub>, D<sub>4a</sub>, X<sub>3</sub>
- ↓  
(S<sub>a</sub>)  
(S<sub>b</sub>)

## Pós-fetch:

- ① MOV R6, R7 → F<sub>1</sub>  
② ADD #34, R7 → F<sub>2</sub>, D<sub>1</sub>  
③ JC label → F<sub>3</sub>, D<sub>2</sub>, X<sub>1</sub>  
④<sub>a</sub> RRA R7 → F<sub>4a</sub>, D<sub>3</sub>, X<sub>2</sub>  
⑤<sub>b</sub> RLA R7 → F<sub>4b</sub>

## \*Formato 2 (1 operando)



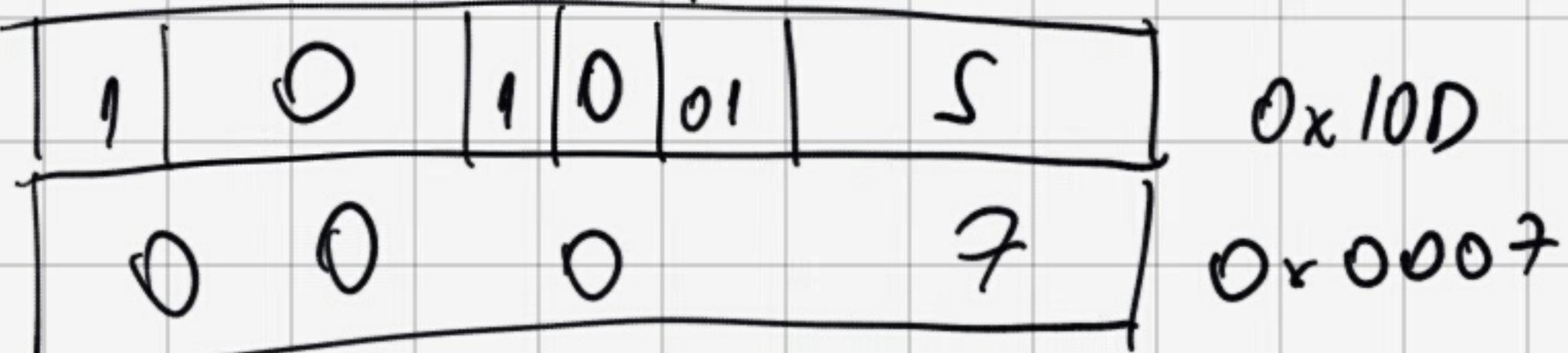
0x0107

SWPB.W

7 (RS)

%W

AD



0x10D

0x0007

## \*Saltos



ADD

# J4, R6

JC

label

RNC

R6

MOV

RS, R7

label

CLR

R8

## \*GPIO<sub>L</sub>:

## \* Aula 7, Periféricos

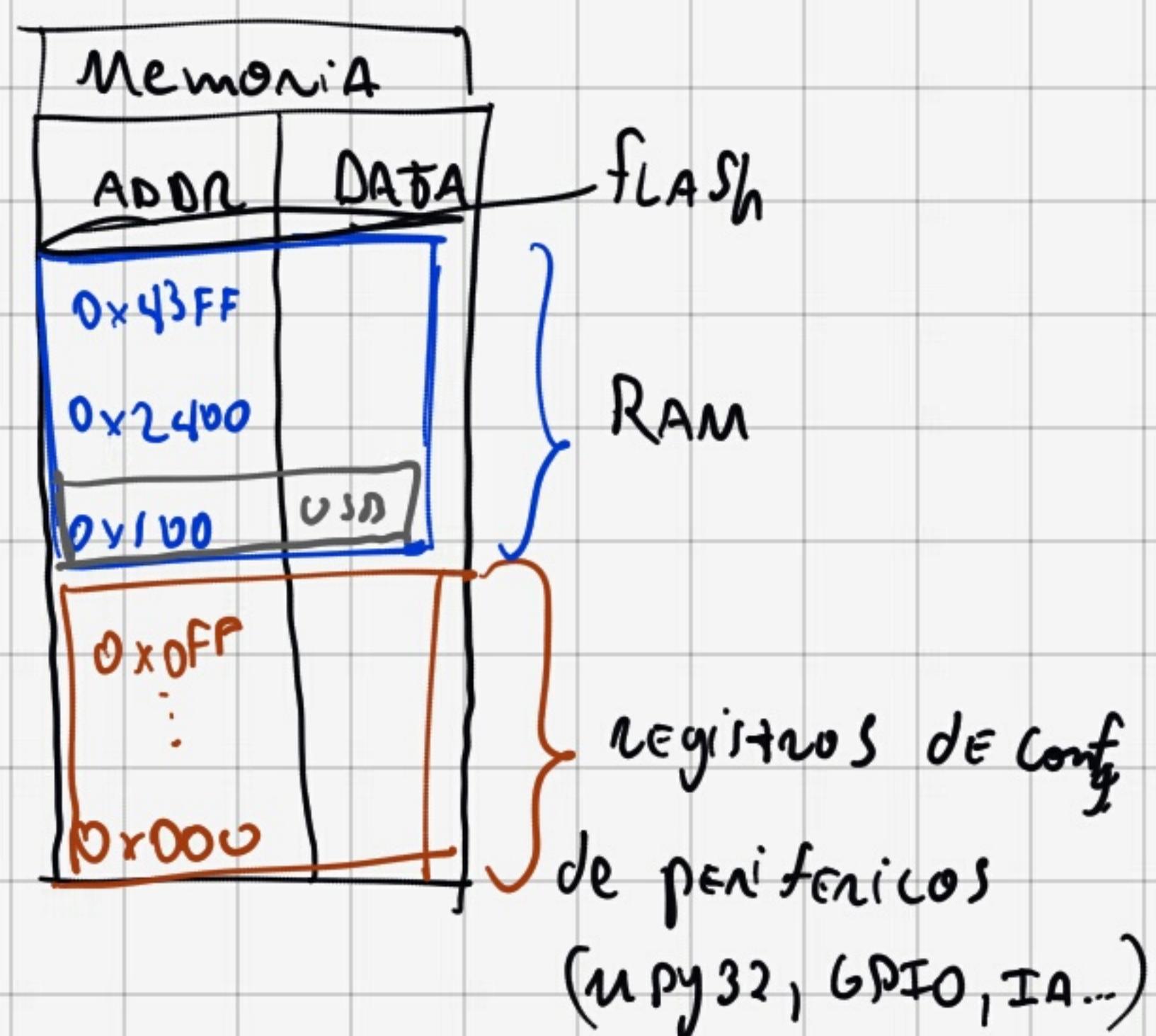
- Multiplicador Dedicado

- GPIO

- ↳ Config

- ↳ Debounce

- Exercícios

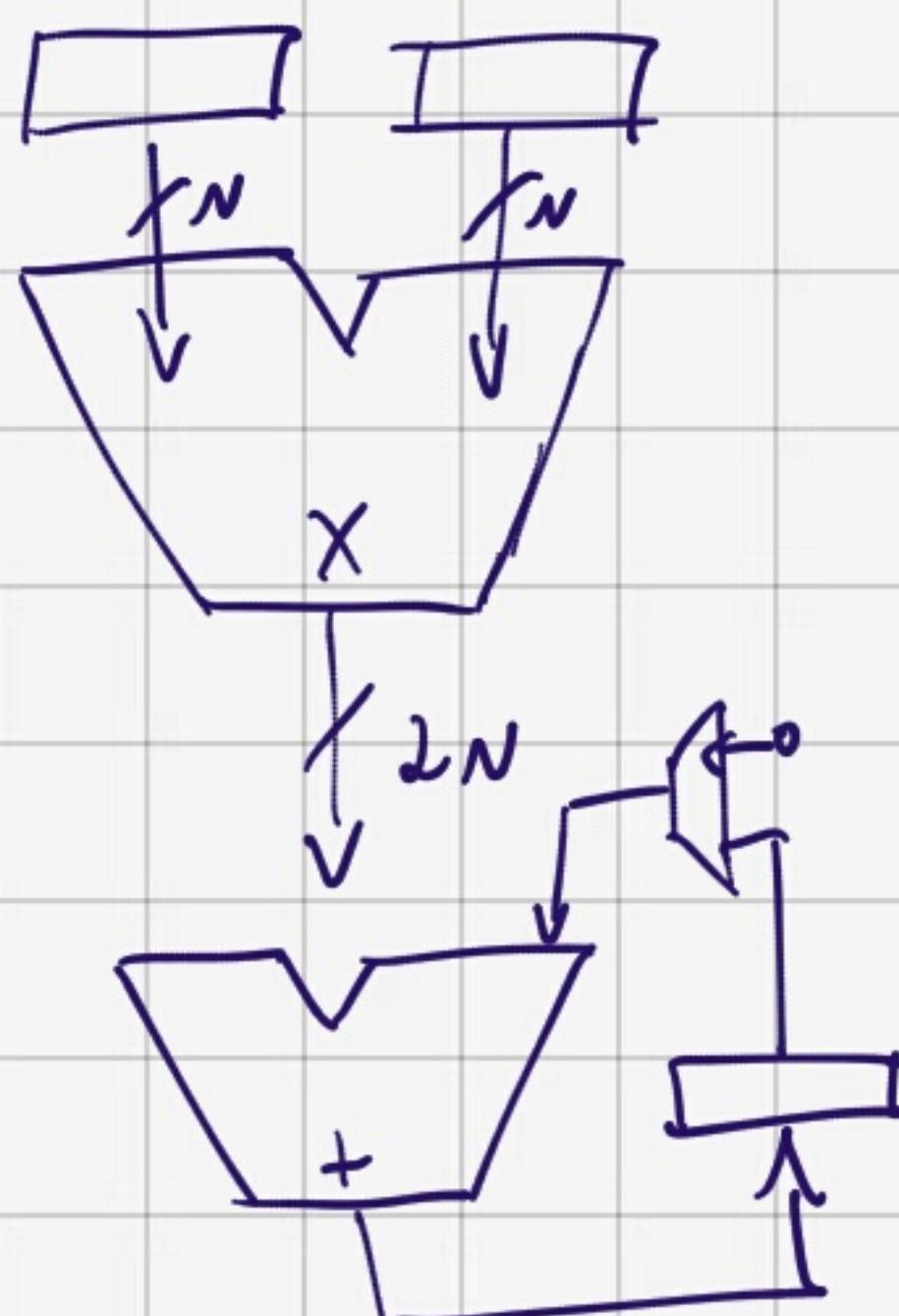


## \* Multiplicador Dedicado:

$$S = \sum_{i=0}^n a_i \cdot b$$

$$\Rightarrow S = a_1 \cdot b_1 + a_2 \cdot b_2 + \dots$$

↗ Mult      ↗ Mult  
 ↗ Soma



$$\Rightarrow P = A \cdot B ; \quad A, B = 32 \text{ bits} \rightarrow A_{31-16} - 16 \text{ bits MS word}$$

$A_{15-0} - 16 \text{ bits LS word}$

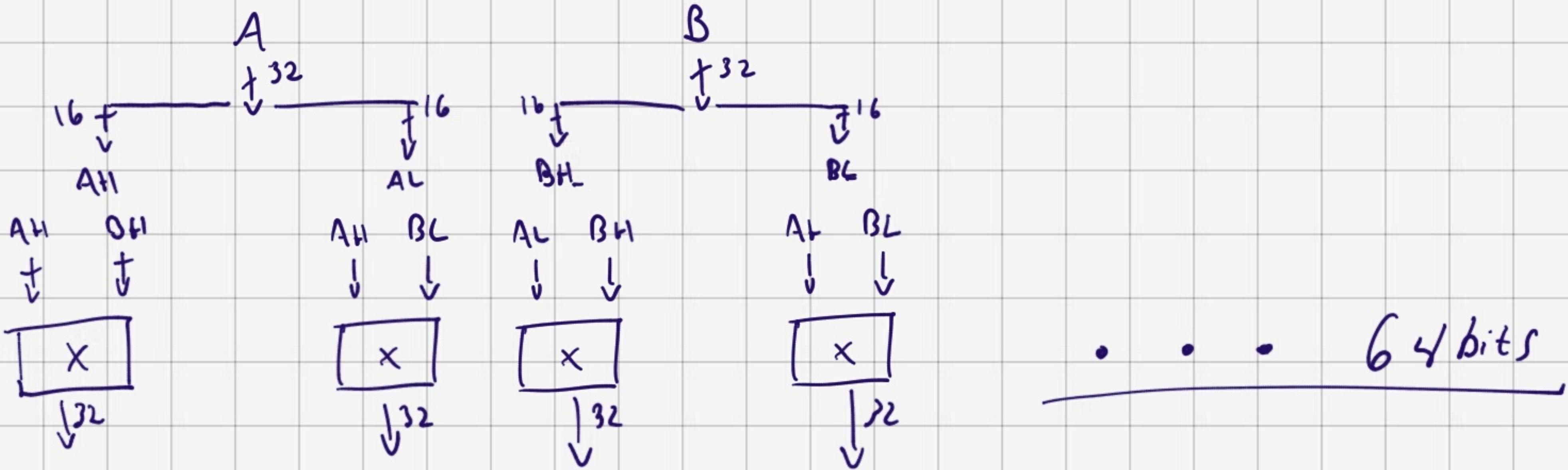
$$\Rightarrow P = (A_{31} \cdot 2^{16} + A_{15}) \cdot (B_{31} \cdot 2^{16} + B_{15})$$

$$P = \frac{A_{31} B_{31}}{P_3} \cdot 2^{32} + \left( \frac{A_{31} \cdot B_{15}}{P_2} + \frac{B_{31} \cdot A_{15}}{P_1} \right) \cdot 2^{16} + \frac{A_{15} B_{15}}{P_0}$$

$$(P_{311} \cdot 2^{16} + P_{31L}) \cdot 2^{32} + (P_{211} \cdot 2^{16} + P_{21L} + P_{111} \cdot 2^{16} + P_{11L}) \cdot 2^{16} + P_{011} \cdot 2^{16} + P_{01L}$$

$$P_{311} \cdot 2^{48} + (P_{31L} + P_{211} + P_{111}) \cdot 2^{32} + (P_{21L} + P_{11L} + P_{011}) \cdot 2^{16} + P_{01L}$$

↑  
carry



## \* NO MSP430 |

\* multiplicación sum final

```
MOV #0x0003, &MPY  
MOV #0x0004, &OP2  
MOV &RESLO, RS
```

" 3 . 4 = 12"  
" 12 -> RS "

\* multiplicación con final

```
MOV # -J, &MPYS  
MOV # 2, &OP2  
MOV &RESLO, RS  
MOV &RESTHI, R6
```

► Ejemplo: multiplicar Vector1 x Vector2

{ RS  
Vector1  
• Byte 3, 4, 3, 2  
Vector2  
• Byte 3, 1, 2, 3  
R6  
 $v = [a \ b \ c]$

dsr  
inc RS  
inc R6  
mov.b @RS+, &MPY  
mov.b @R6+, &OP2  
RESLO → ACUMULA

Loop:

```
mov.b @RS+, &MAC  
mov.b @R6+, &OP2  
dec.b R10  
JNZ Loop
```

Mov.w &RESLO, R7

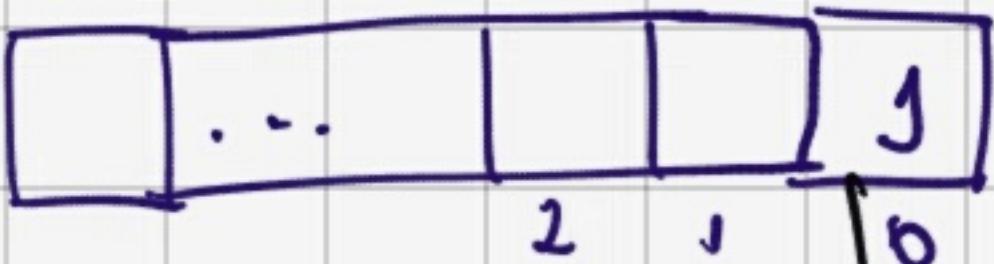
RESULTADO DA  
MULTIPLICACIÓN

## GPIO: General Purpose Input/Output

MSP430

fora do chip

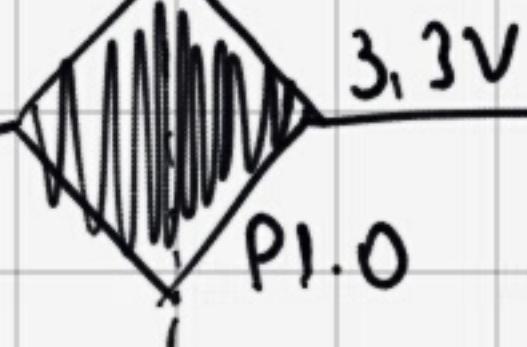
P1OUT



X mov.b #BIT0, &P1OUT

BIS.b

#BIT0, &P1OUT



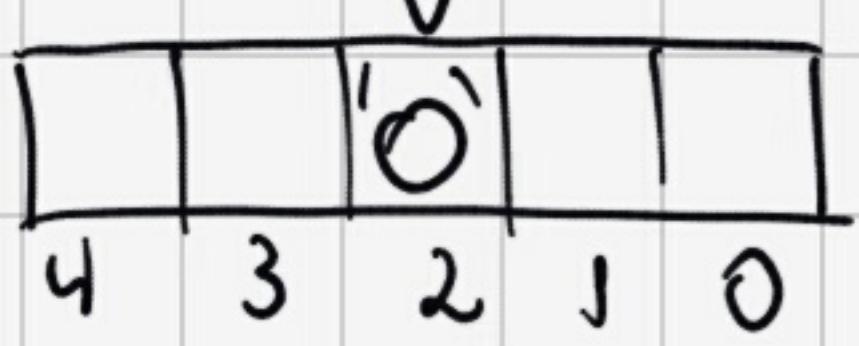
escrever 1 → liga  
escrever 0 → apaga

↓ ↗ liga

OUTPUT

P1.0

P1IN



∴ read-only

Buffer with Schmitt

P1.1

P1.2

3.3V

0V

"Informações"  
"não DATA'S"

bit.b #BIT2, &P1IN → resultado vai p/ o carry  
(Bit test)

JC

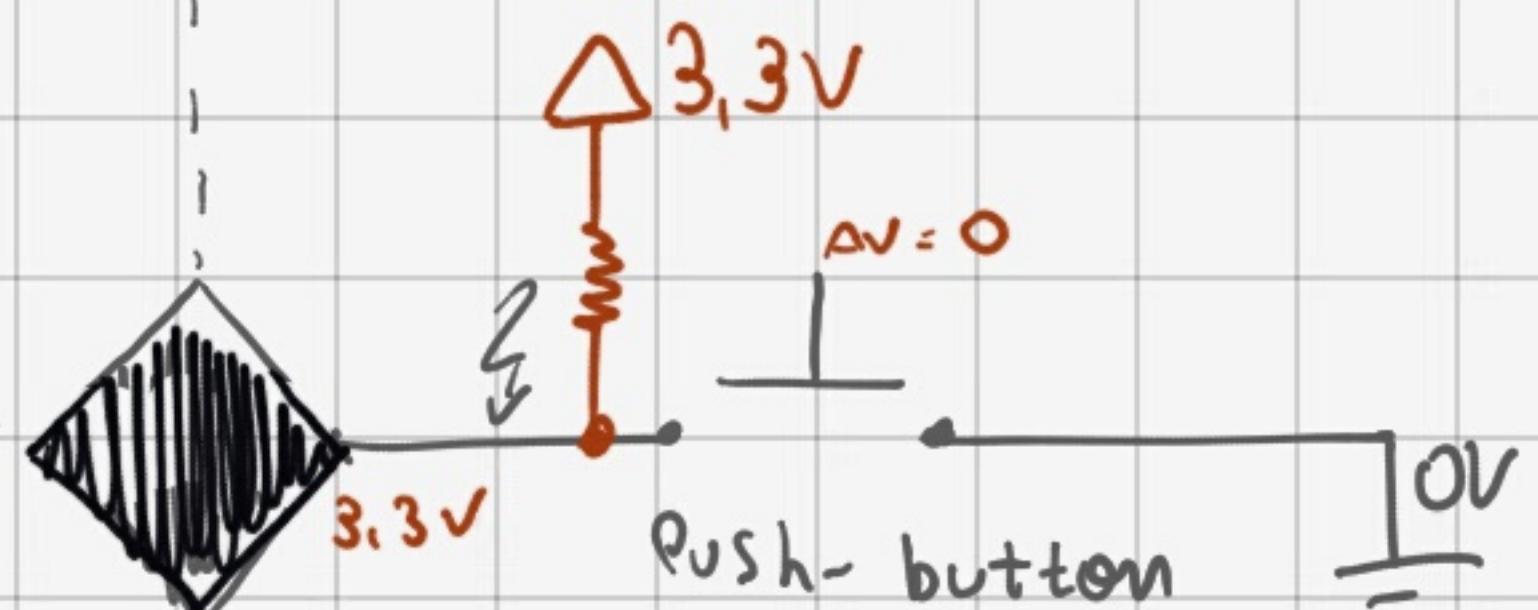
P1b2 Setado

P1.b2 zero:

P1DIR

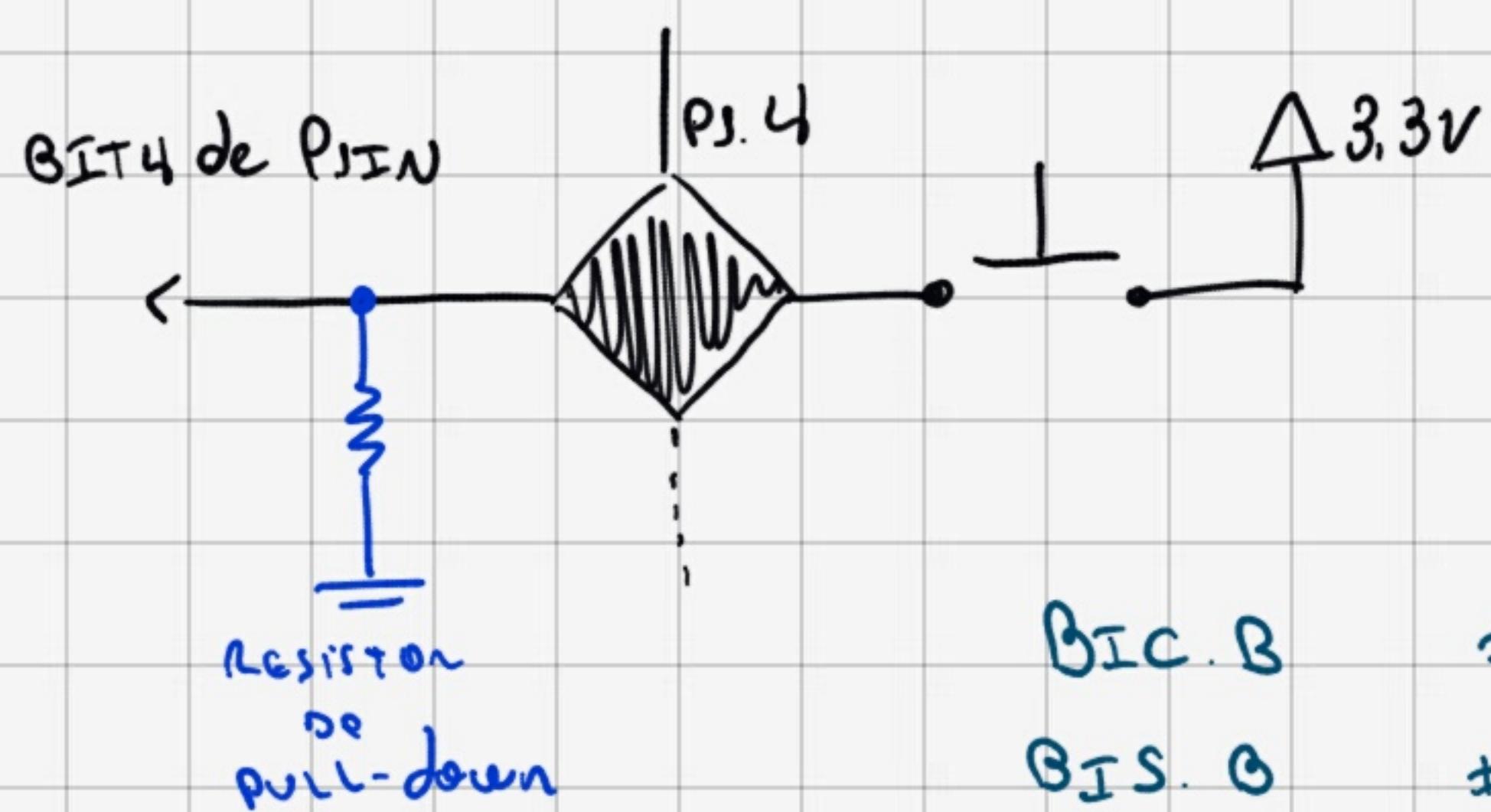
P1OUT

P1IN

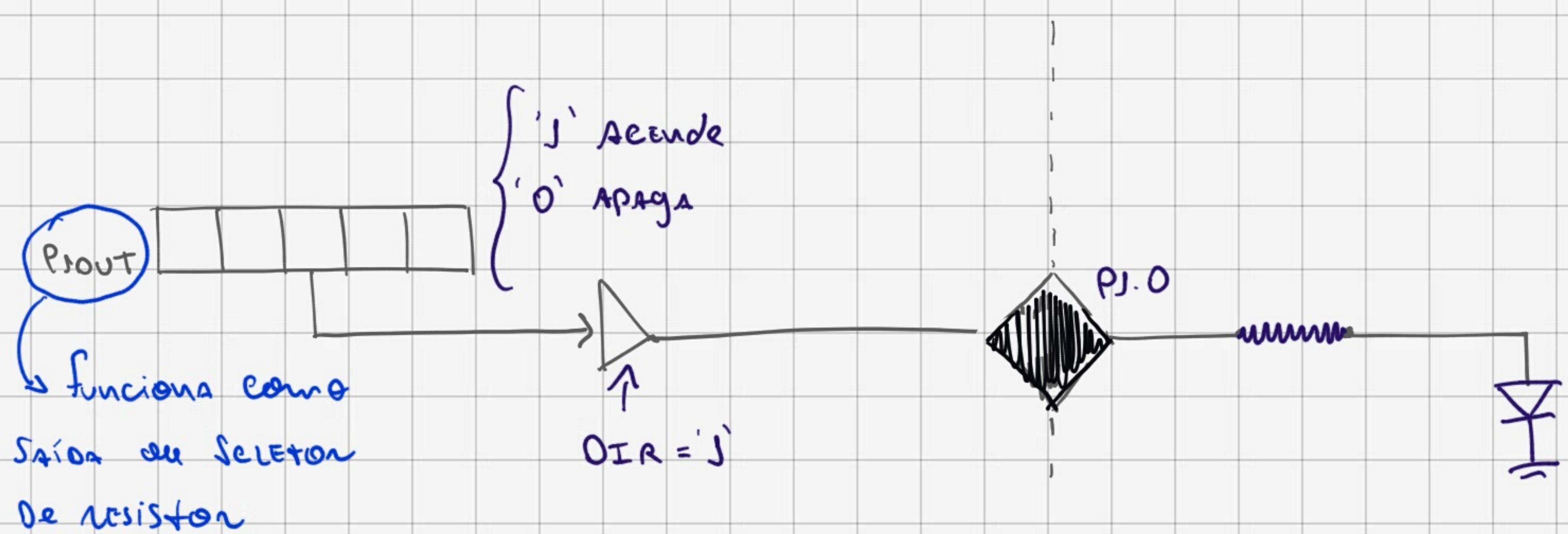


⇒ Quando não pressionado o pino "flutua", não temos uma tensão definida.  
⇒ Soluções:  
Resistor de pull-up/down

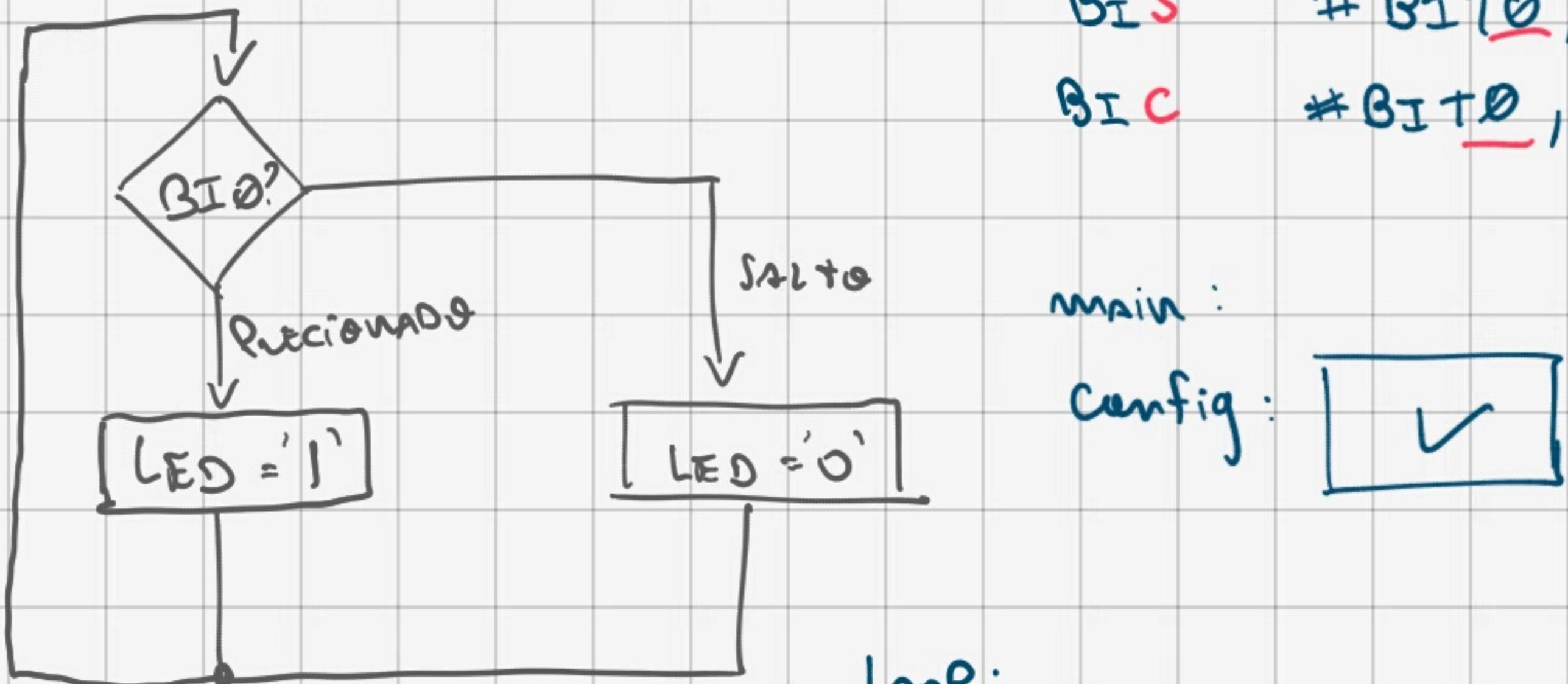
## Exemplo: PULL-DOWN



BIC.B      #BIT4, &PJDIR ; Pj.4 é entraõ  
 BIS.0      #BIT4, &PJREN ; habilita resistor  
 BIC.B      #BIT4, &PJOUT ; res. de pull-down



BIS      #BIT0, &PJ DIR ; PJ0 é saída  
 BIC      #BIT0, &PJ OUT ; LED apaga



BIT.B      #BIT 0, &PJ IN  
 JC            Soltado

Boton      { '0' → Precionado  
 '1' → Soltado

Precionado:

BIS.B      #BIT0, &PJOUT  
 JMP          Continue

Soltado:

BIC.B      #BIT0, &PJOUT

Continue:

JMP          Loop

## # Prova } :

- ⇒ Arquitetura RISC
- ⇒ Assembly
- ⇒ GPIO

## # Livros:

- Programação de Sistemas Embutidos  
Rodrigo M. A. Almeida

- ⇒ Microcontroladores MSP430: teoria e prática  
Fábio Perinha

\* Arq. Risc: (Reduced Instruction Set code - risc) MSP430

→ Apesar de o MSP430 ter uma Arq. von-Neumann, → Arq. do tipo Von-Neumann  
ele utiliza de um set de instruções (RISC).  
São 27 instruções físicas (core instructions).  
Com o uso dos registradores que geram constantes (R2 e R3) é possível  
enviar mais 24 instruções, totalizando 51 instruções

→ As instruções são de 3 formatos (formato 1, formato 2, jump)

- Dual-operand → Dois operandos → fonte e destino
- Single-operand → Apenas um operando → fonte ou destino
- jump → Saltos

⇒ todos as instruções de operando simples ou duplos possuem de 16 ou 32 bits. ⇒ Nomenclaturas

- Src: O operador fonte é definido por As e S-reg;
- dst: O operador destino é definido por Ad e D-reg
- As: Determina qual modo de endereçamento utilizado para instrução, especificando quem é o registrador fonte.

- S-reg: Quem das 16 registradores diretos à CPU, é utilizado como fonte
- Ad: Determina qual modo de endereçamento utilizado pela instrução, especificando quem é o registrador destino.
- D-reg: Quem das 16 registradores diretos à CPU, é utilizado como destino
- B/W: Indica se a instrução é byte(1) ou word(0)

## # Questão 3 PVJ - Anterior # Consulta a tabela (montagem)

TABLOUIM:      BIT      #B13, RS ;(3=inst)      RRC      RS ;(2=inst)      JNC      RANDNUMBER

XOR.B      R6, R5

RANDNUMBER:      NET

(1) ➤ Primeira instrução:  
 0x80 3 S  
 0x0004

Ad	B/W	AS
0	0	JJ

*destino*

(2) ➤ Segunda Instrução

Resposta:

0x80 3 S	0x0004	0x000S	0x280J	0xE64S	0x4130
Ad	B/W	AS			
0	J	JJ			

" ⇒ 001|010|00000000 J"  
 Instrução → 0x280J

## \* Exercício da lista: Consulta tabela (Desmontagem)

8077	SUB.B # 0x0030 , R7	Ad      B      AS
0030	CMP.B # 10 , R7	0   J   JJ
9077	SL      LJ	; +2
000A	CALL # 0x404E	
3802	ADD      R7 , R6	
J200		
404E		
S700		

LJ: ADD

Bit mais significativo  
 0x128 : CALL!

B = 1011

## Exercício GPIO : Questão 24 Lista

P4.7

P1.0

4PENTADO  $\Rightarrow$  0

BIS.B #BIT7, &P4DIR

S2 - P1.J  $\Rightarrow$  Entrada

BIS.B #BIT0, &P1DIR

LEO - P1.0  $\Rightarrow$  Saída

BIS.B #BIT7, &P4OUT

BIC.B #BITJ, &P1DIR

BIS.B #BIT0, &P1OUT

BIS.B #BITJ, &PIREN

BIS.B #BITJ, &P1OUT

BIS.B #BIT0, &P1DIR

MAIN:

BIT.B #BITJ, &PIIN

JC SOLTO

N SOLTO: BIS.B #BIT0, &P1OUT

JMP CONT

SOLTO: BIC.B #BIT0, &P1OUT

CONT: JMP MAIN

## \* Aula 9: GPIO ++ / C

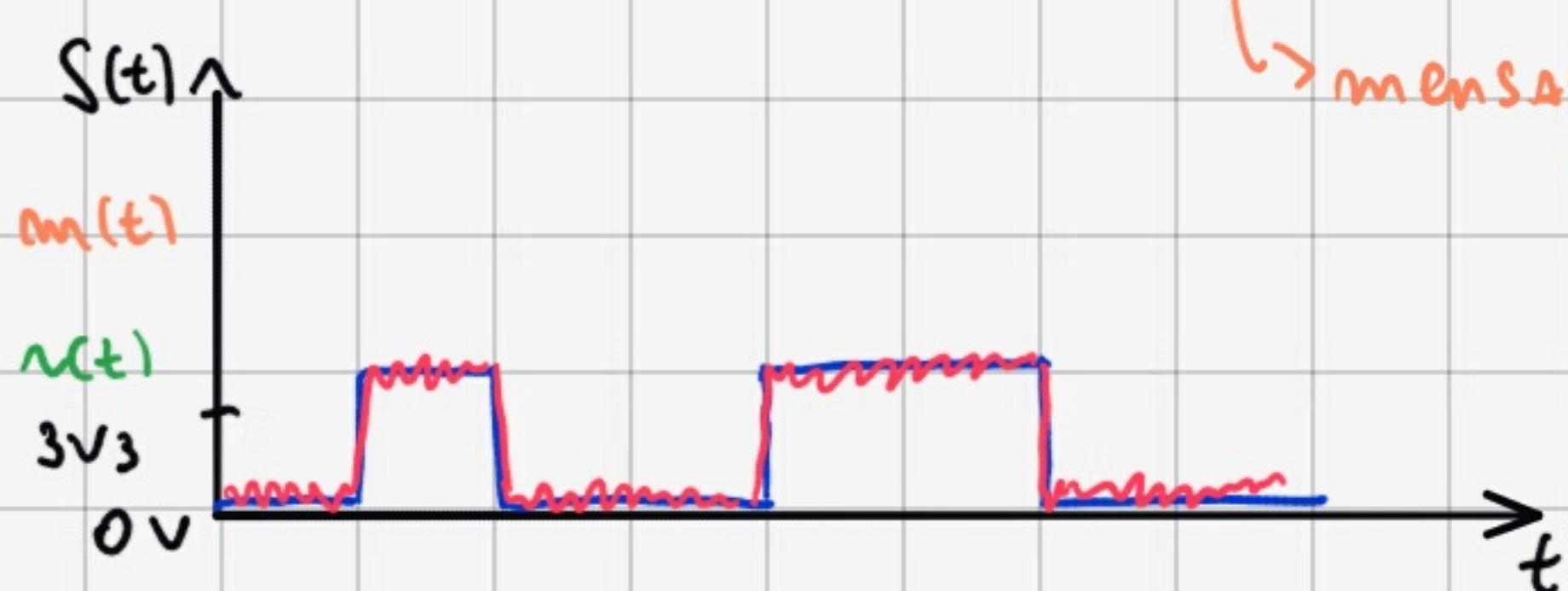
- AVISOS
- GPIO
  - ↳ Nebotes
  - ↳ ruídos
- leitura de ENTRADA
- C: INTRODUÇÃO

### 10 Aspectos Análogicos de pinos digitais:

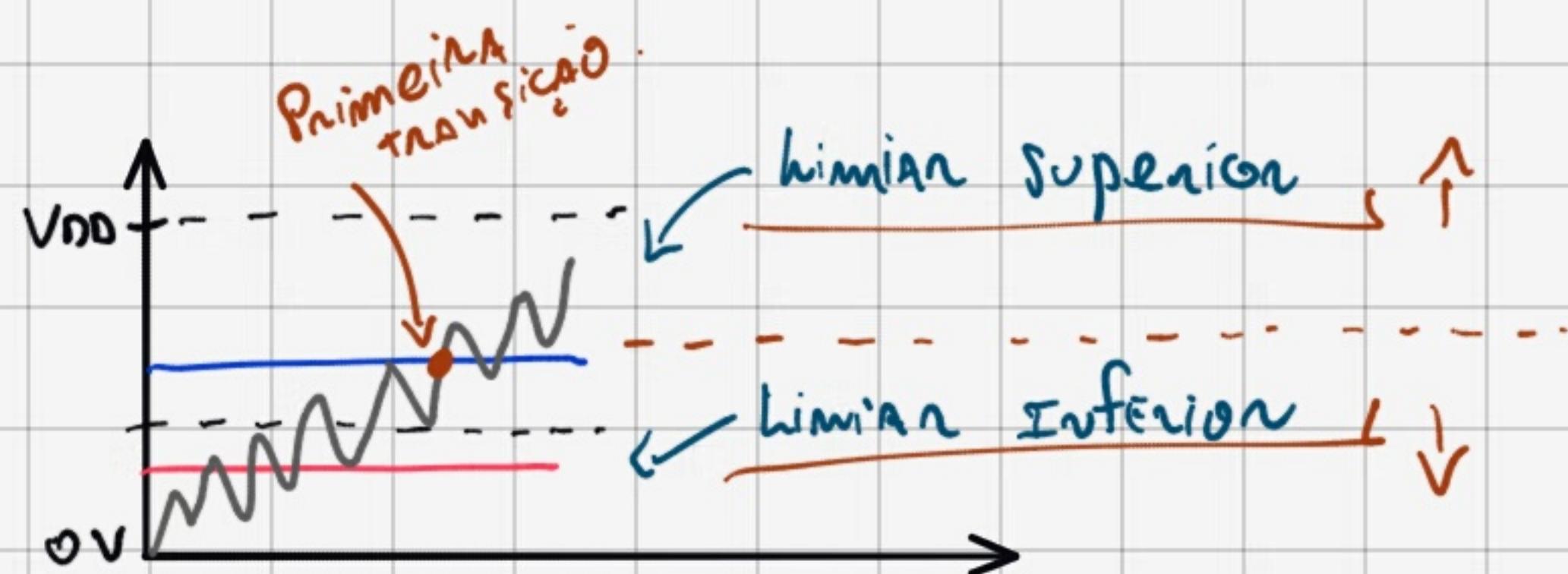
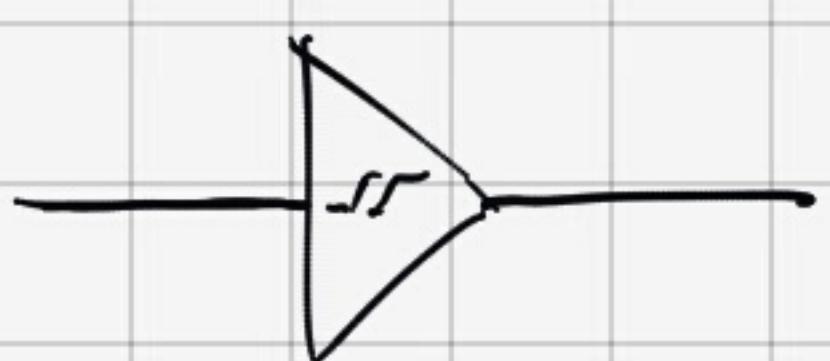
#### \* Ruídos:

$$S(t) = m(t) + n(t)$$

↳ variável aleatória  
↳ mensagem



#### ↳ Schmitt trigger:



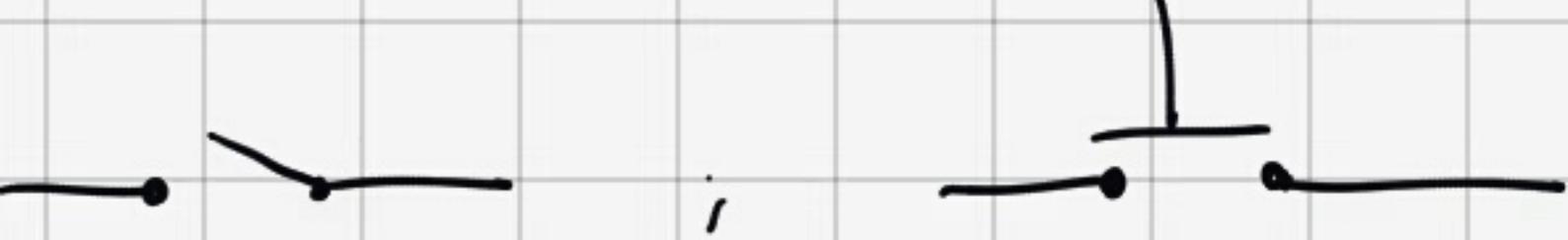
#### ⇒ Não-determinismo de ENTRADA:



- Quando a CHAVE não faz contato não podemos determinar a tensão do pino

#### ⇒ ALGUMAS CHAVES:

- Single Pole, Single throw (SPST)

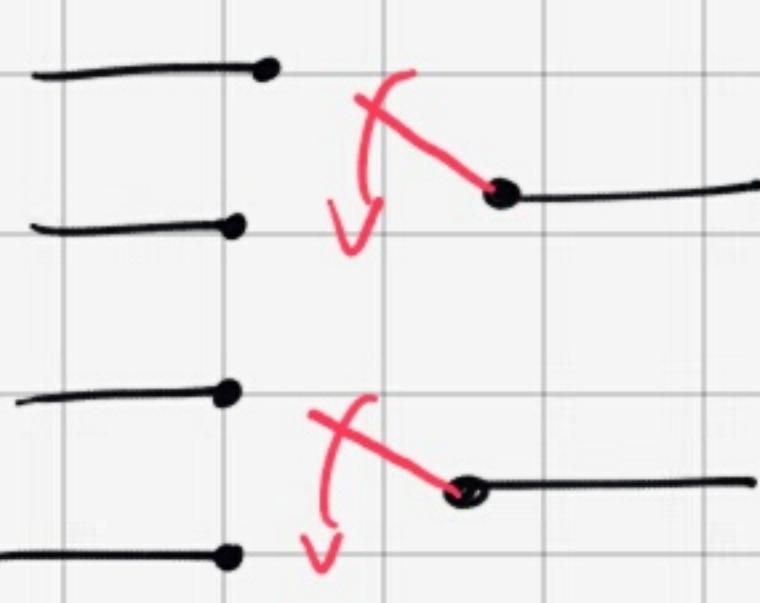


- Single Pole, Double throw

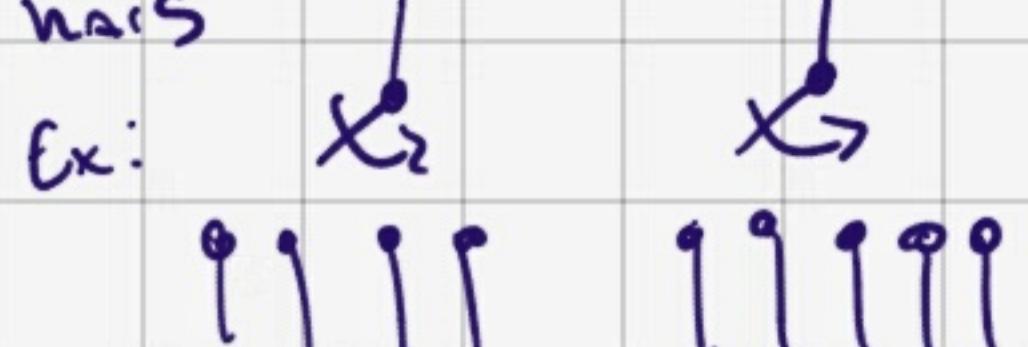


Obs: Pole → n. de aguçamentos de CHAVE.

- Double Pole, Double throw

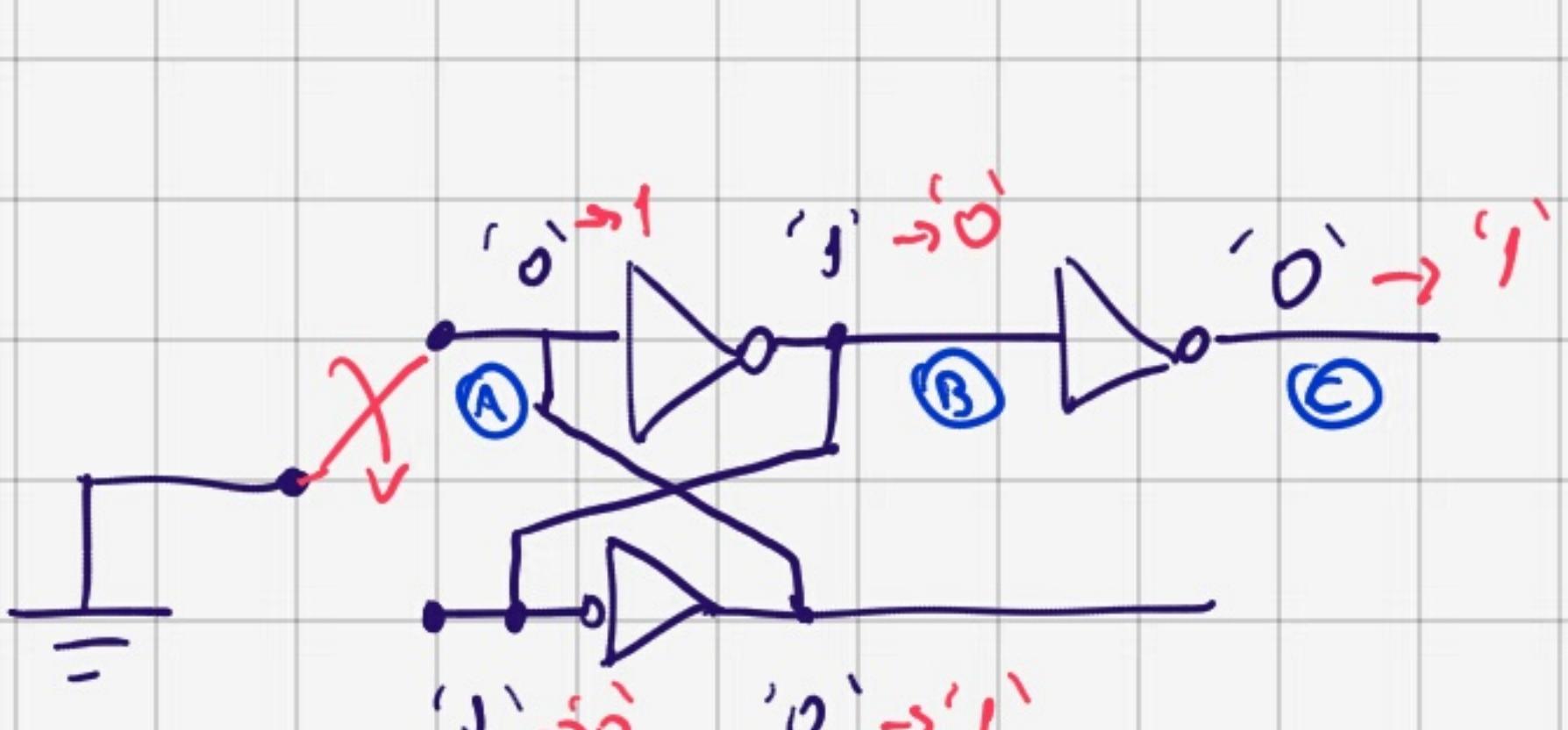
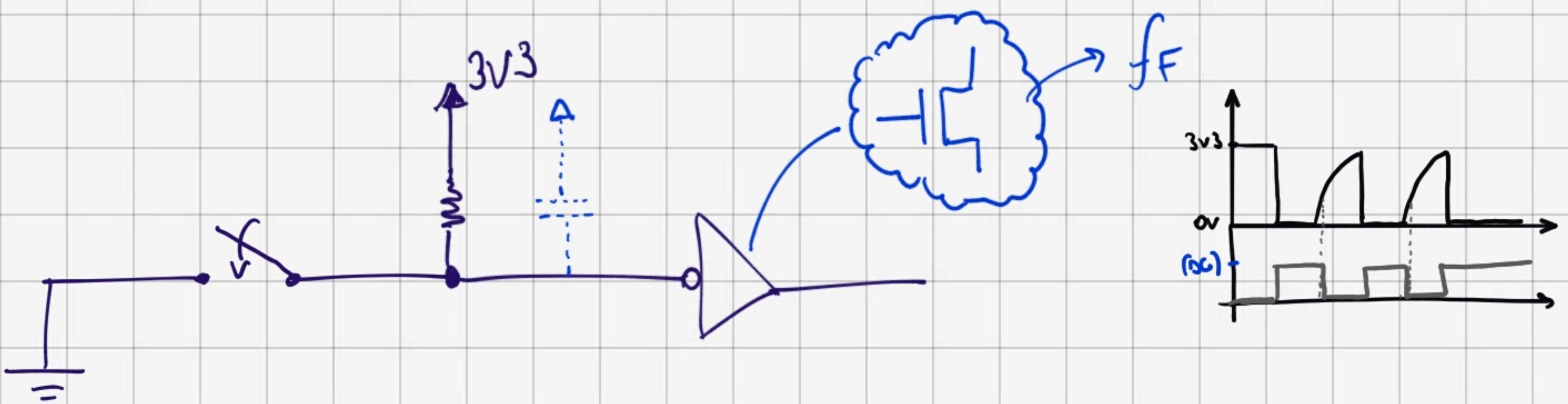


throws: n. de terminais



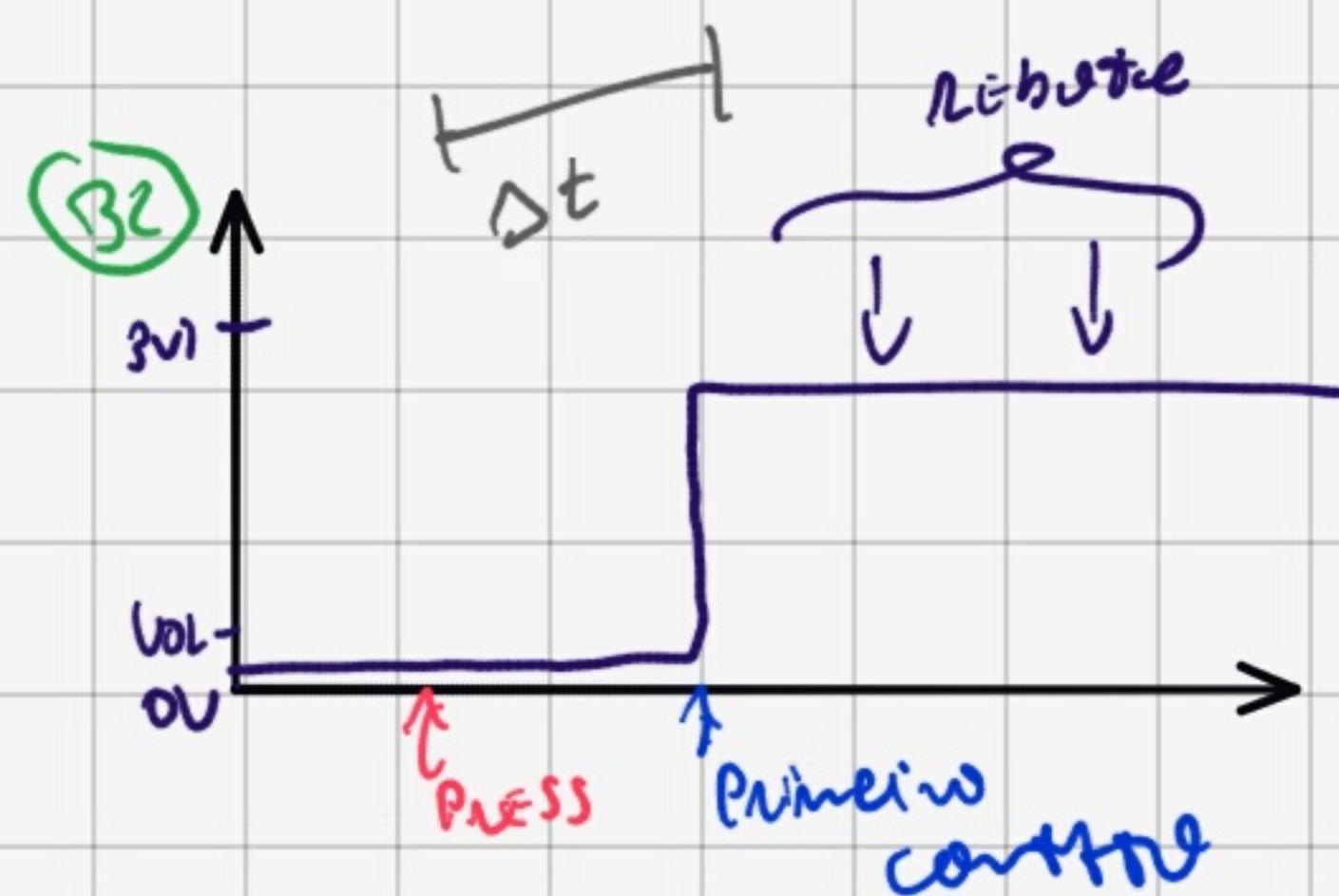
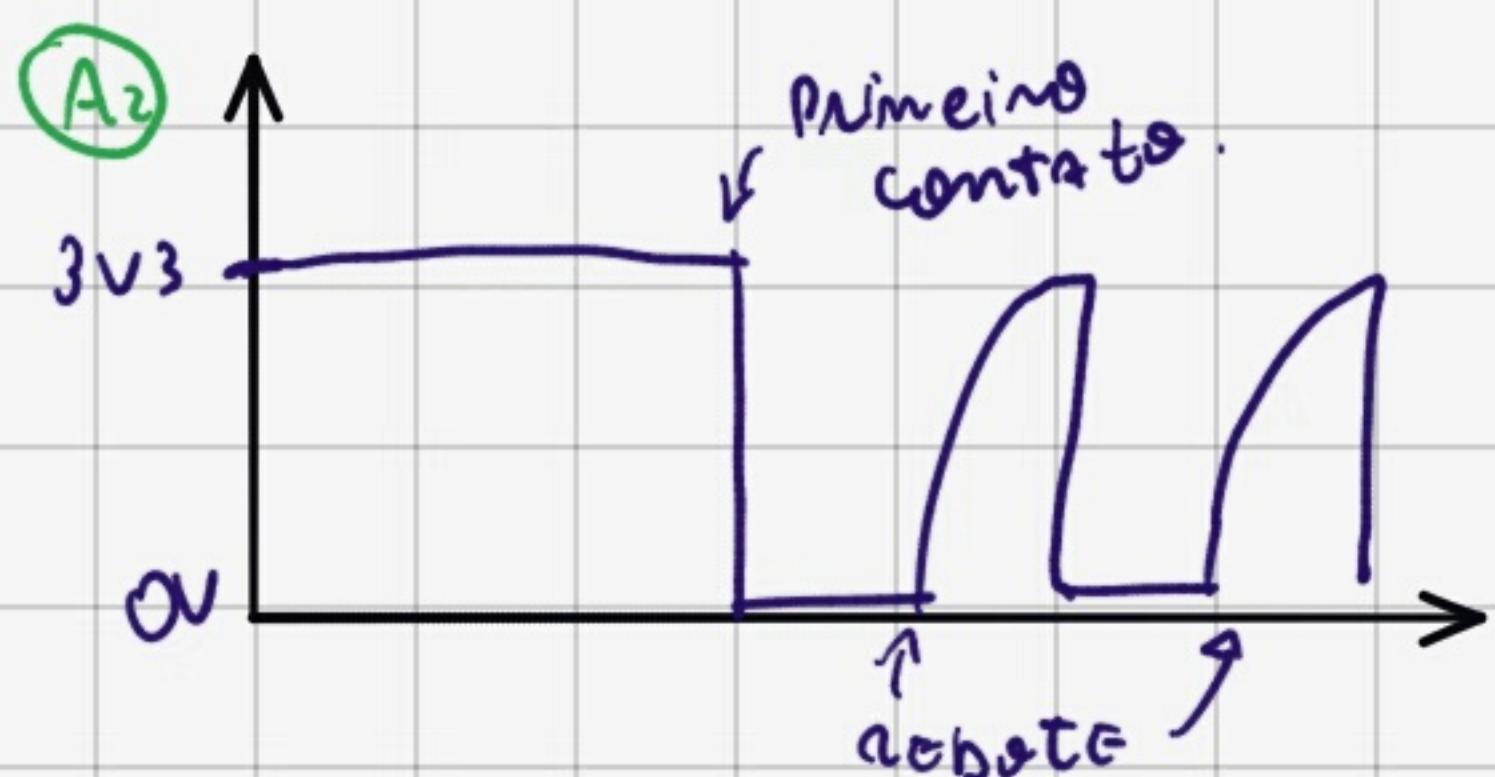
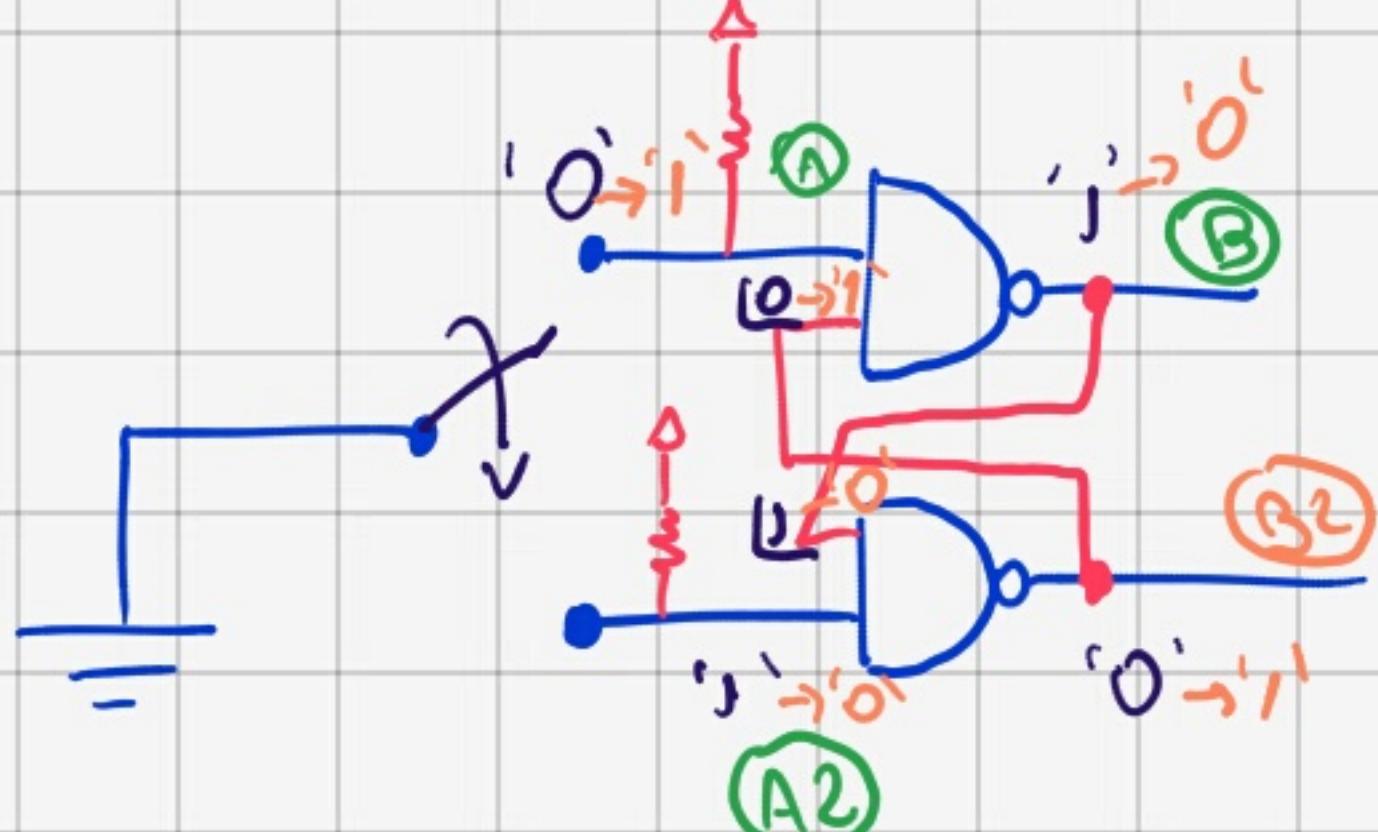
## \* Rebotes:

- trepicadores, Rebotes, ou "bounce" em inglês



\* Solução para baixar consumo de energia ("Evitar o 'Rebote'")  $\rightarrow$  USANDO NANDS:

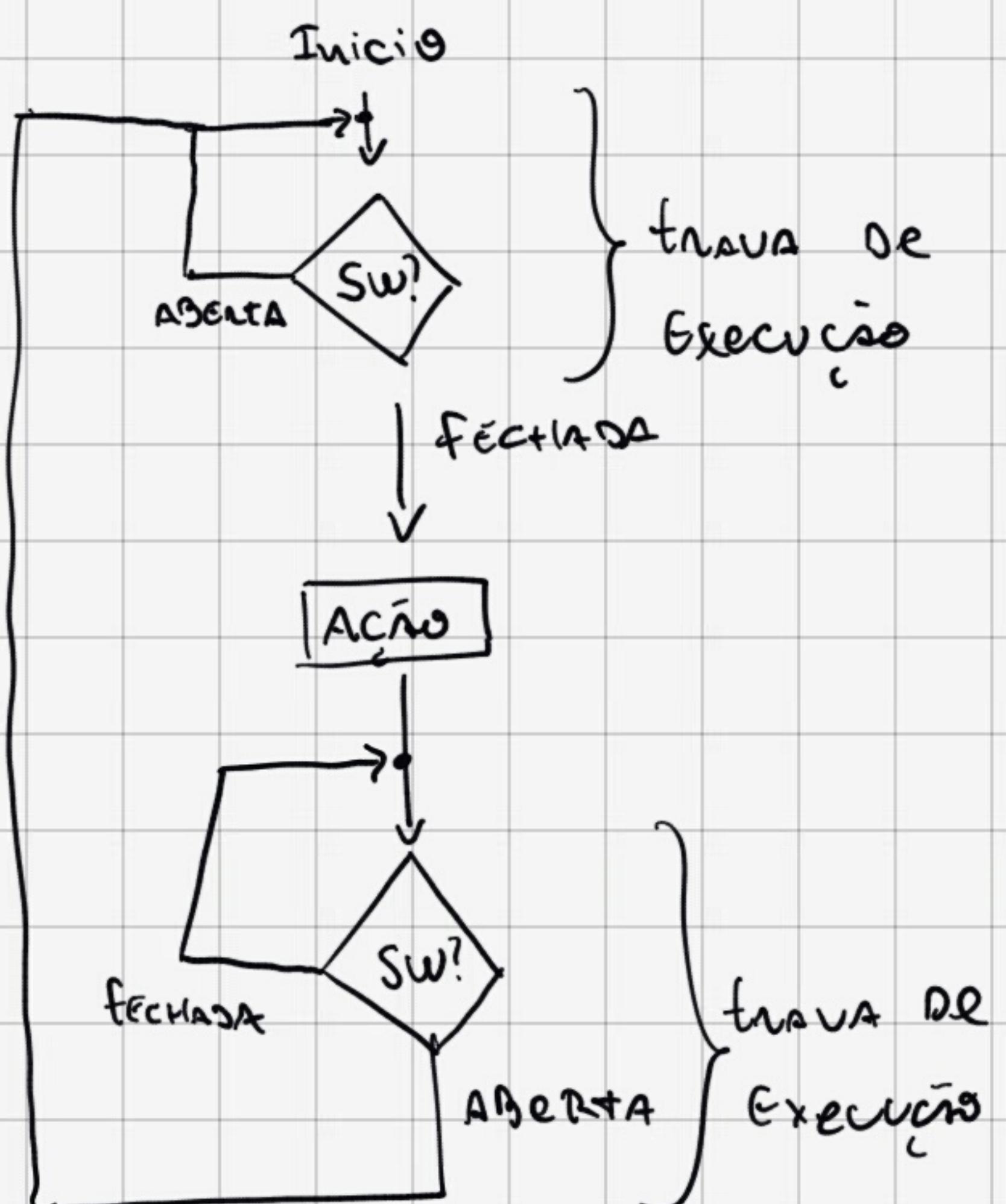
NANDs:



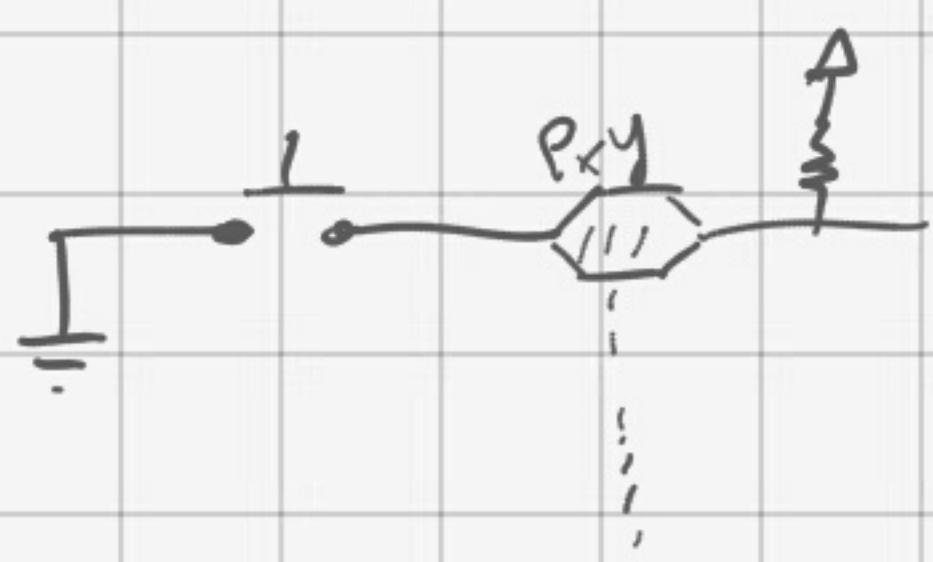
$\Delta t$ : tempo de resposta do circuito digital

$$\Delta t < T_{rebote}$$

## \* Lemocão de Rebotes por Software



Cm ASM:

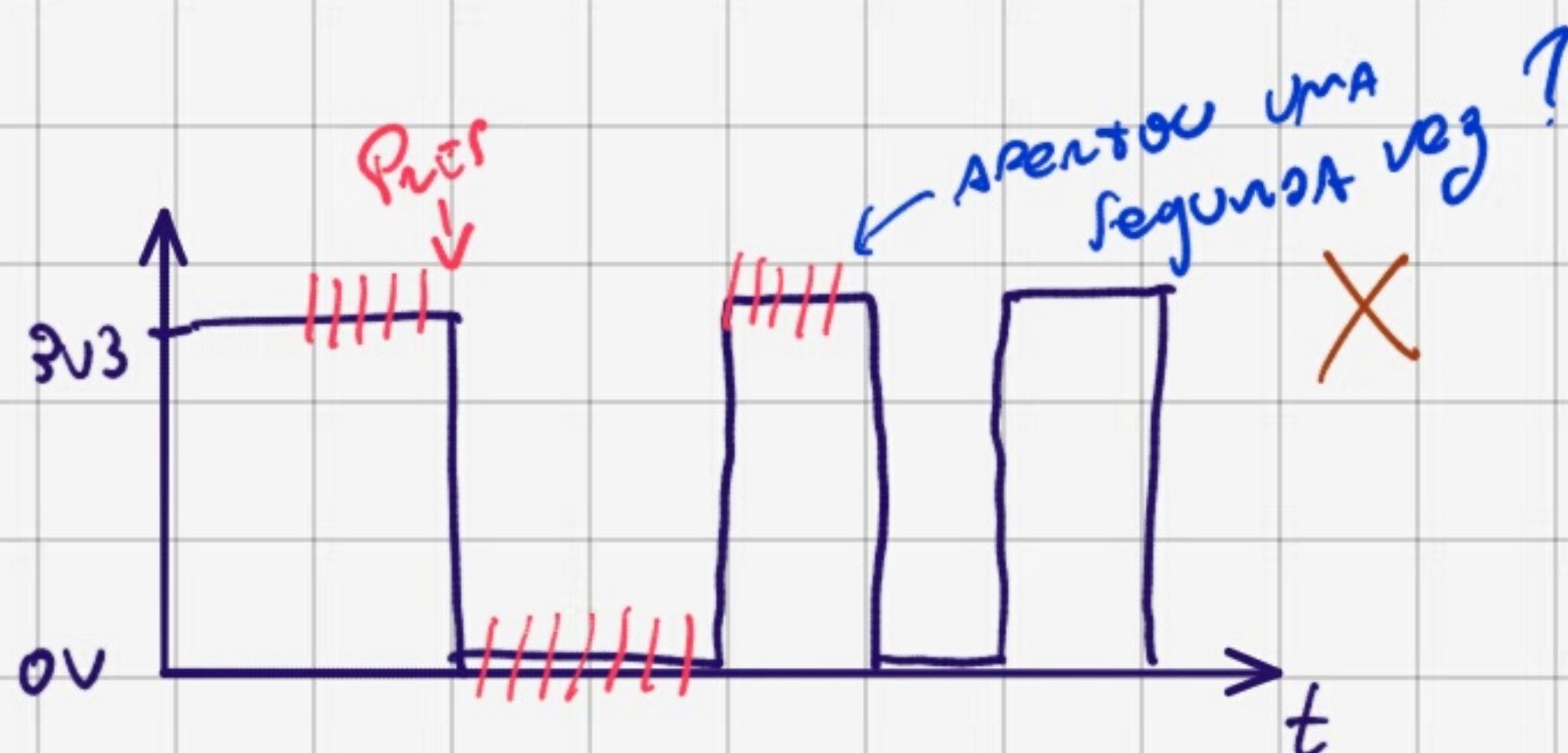


Loop:

trava1: BIT.B #BITY, & Px IN  
JC trava1

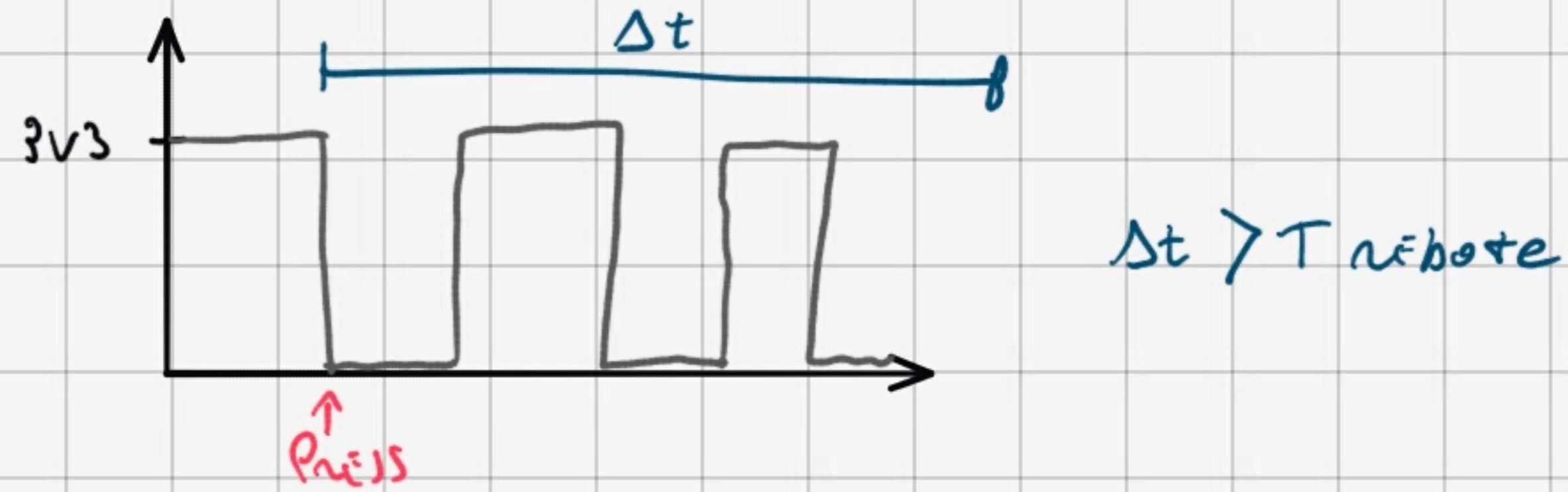
Acção:

trava2: BIT.B #BITY, & Px IN  
JNC trava2  
SMP Loop



Rebote "engata" o processador  
indicando que o usuário soltou a  
chave, o que não é verdade.

∴ USE DO ATUAIS INTENSAIS



\* Como determinar  $\Delta t$ ? ∴ Empiricamente

→ Se  $\Delta t$  é pequeno

+ Sistema reativo

- Pouca durabilidade

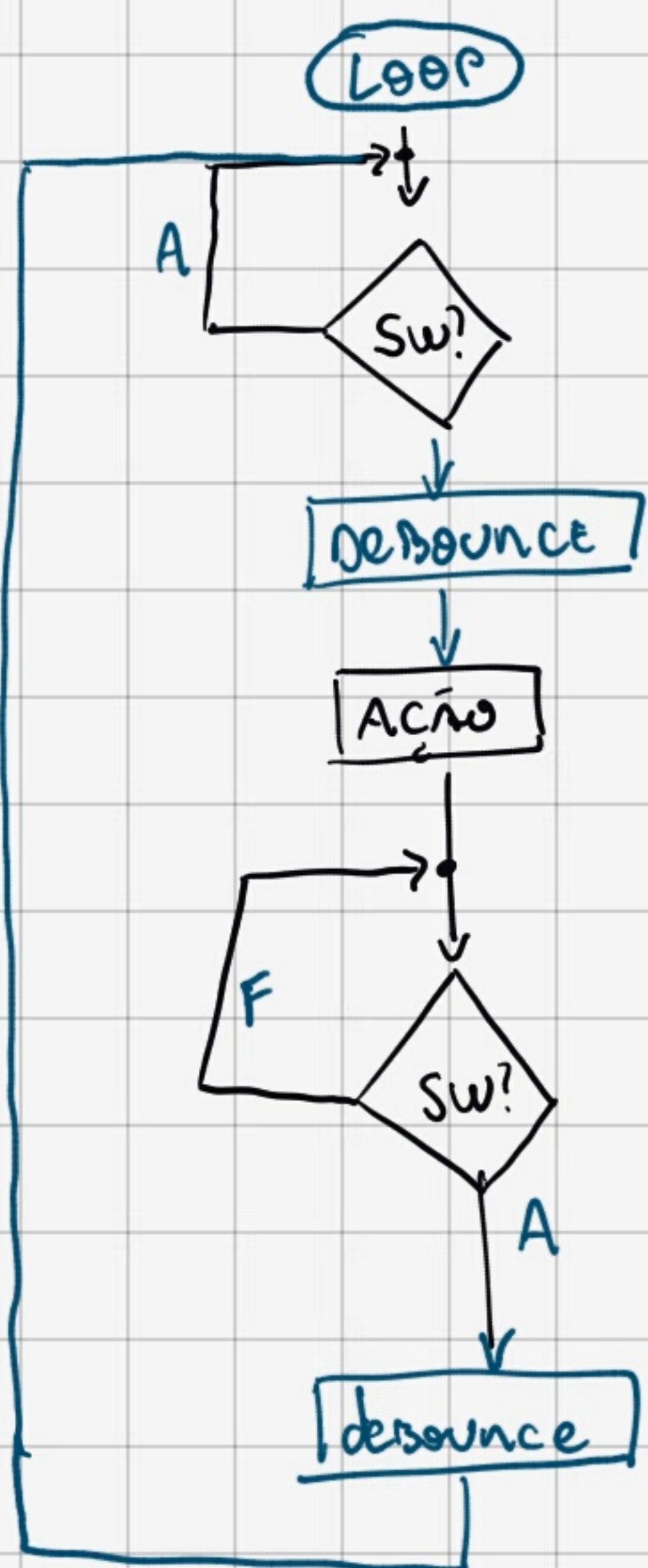
(Trebote ↑ c/ o envolvimento da CHAVE)

→ Se  $\Delta t$  é grande

+ Prolonga duração tempo

- menos reativo

⇒ Configura inicial  $\Rightarrow \sim 100\text{ms}$



" pg 141 user-Gui "

debounce:

MOV.W #1000, R5

deb-Loop:

3 batidas  
declock

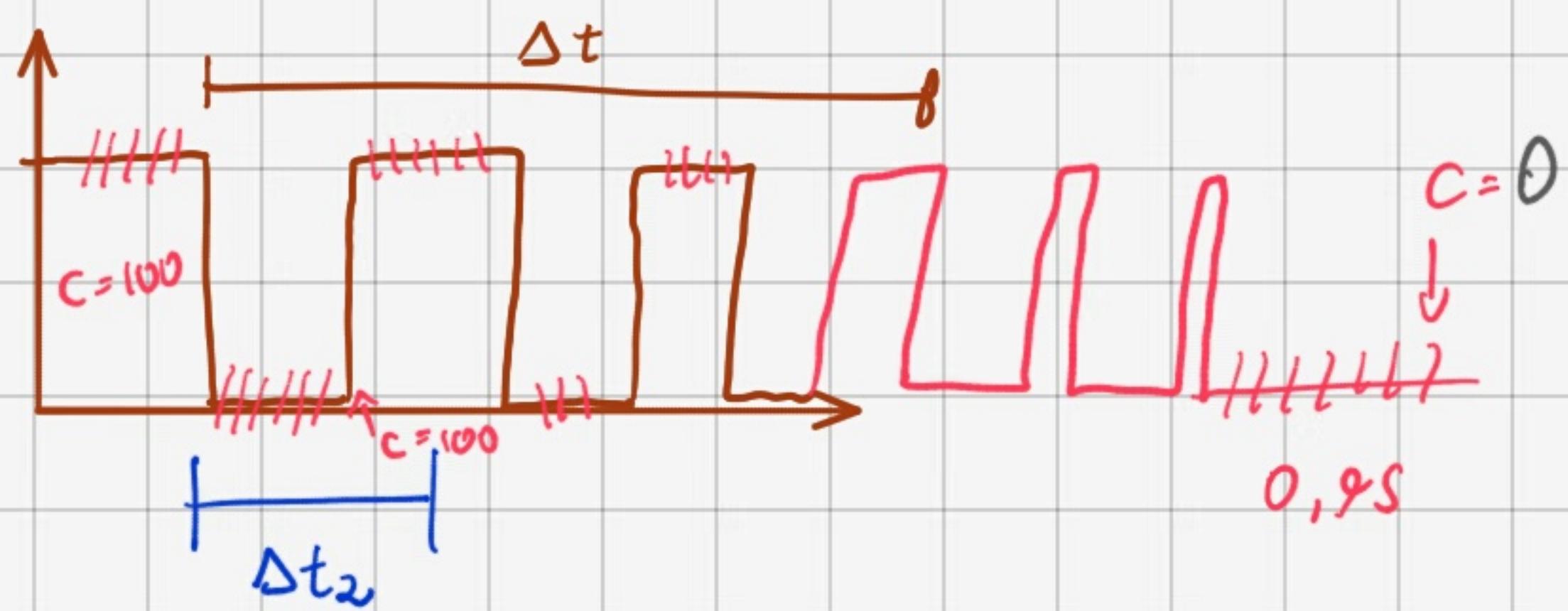
RS  
deb-Loop  
RET

(+1)  
(+2)

MCLK @ 10KHz

$$t_{gatito} = 3000 \times \frac{1}{10.000} = \frac{3}{10} s = 300ms$$

► Melhorar a reatividade reduzindo  $\Delta t$



Verif. Sw:

BIT.B #BITy, &PxIN  
JC Verif. Sw

Mov.W #100, RS

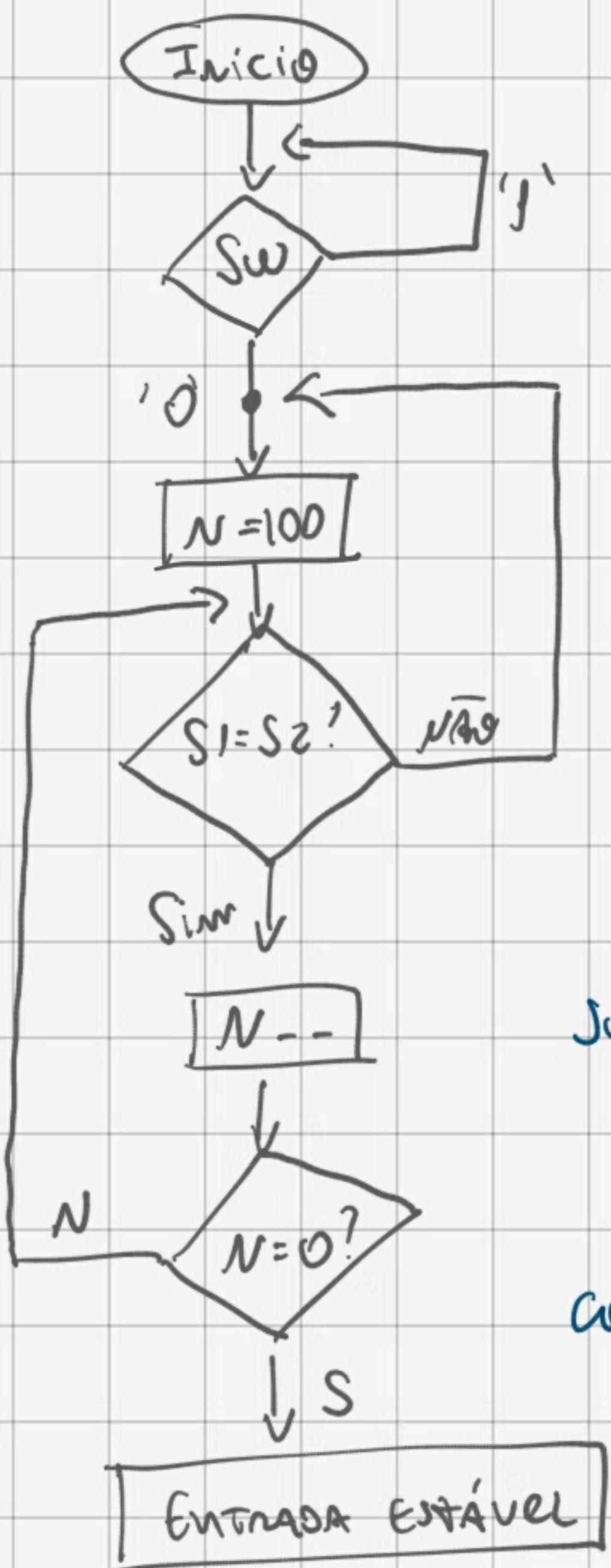
BIT.B #BITy, &PxIN

JC

get sw

get sw

get sw → R7



Press:

Mov.B #1, R6  
JMP continue

Cmp R6, R7

JNE loop

Dec RS

Mov R7, R6  
Jmp get sw R7

JW SOLTO:

Mov.B #0, R6

continue:

• • •

## \* AULA 3:

→ Anúncios:

- minicurso IoT (OPTA)
- Palestras IoT OPTA → Hoje às 14h no Auditório
- Laboratório: Módulo 2 Adiado de 1 Semana

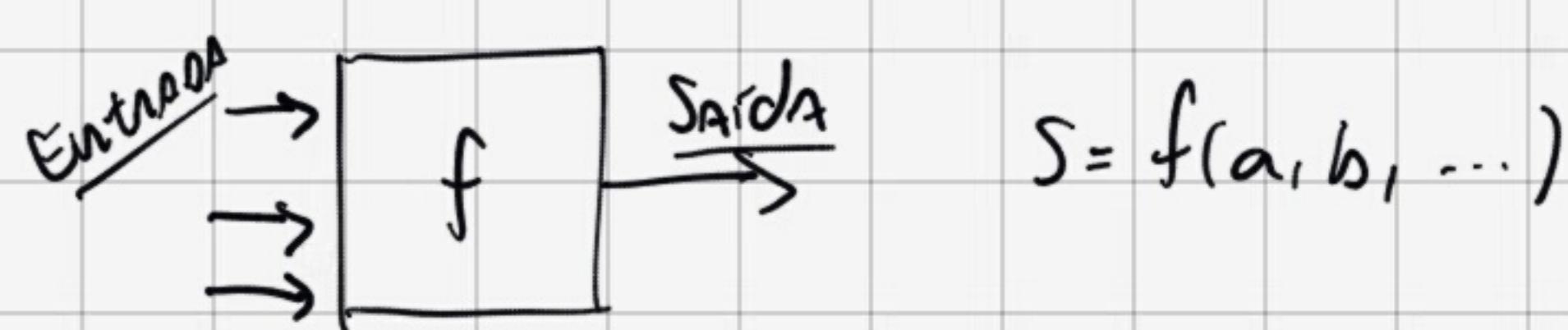
## \* AULA: → Funções

→ Macros

→ Controle de fluxo

→ Intr. A Interrupções

\* Funções: Encapsulam funcionalidade repetitiva.



Em C, uma função tem N entradas e APENAS 1 retorno.

Em ASM, uma Subrotina tem N entradas e N saídas.

\* Passagem de parâmetros:

→ Por valor:

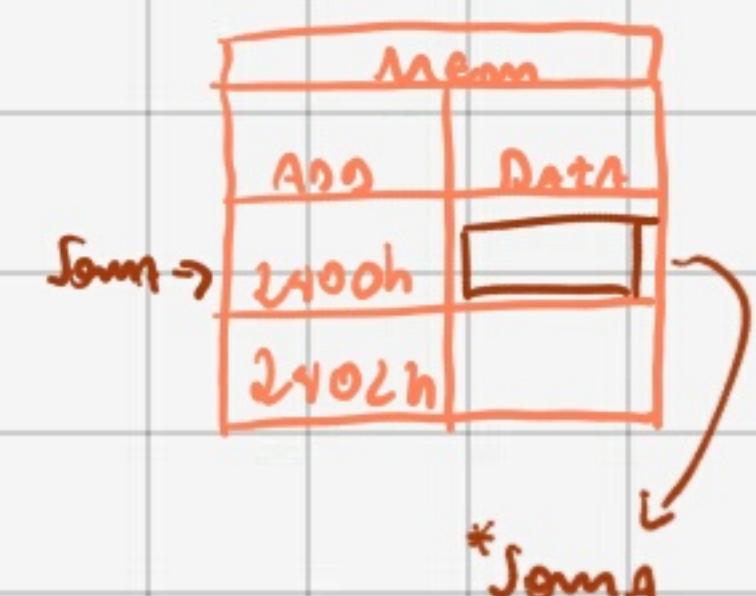
```
Unit16-t Soma(Unit8-t a, Unit8-t b){  
    Unit16-t S;  
    S = a + b; → em "return (a + b);"  
    return S;  
}
```

→ Por referência:

```
Unit16-t SomaAcc(Unit8-t a){  
    STACt unit16-t Soma = 0;  
    Soma += a;  
    return Soma;  
}
```

```
Void SomaAcc(Unit16-t *Soma Unit8-t a){  
    (*Soma) += a;  
}
```

Referência da  
Soma /  
↓



→ VSO da função C/passagem de parâmetro por referência:

main(){

```
    Unit16-t Soma = 0;  
    SomaAcc(&Soma, 5);  
    SomaAcc(&Soma, 10);
```

&: Operador que obtém o endereço da variável  
\*: Operador que diferencia o end. → busca o valor.

MACROS: → Declarações com tratamento e substituição de strings.

\* Define PI 3.14159

a = 2 \* PI \* n; → a = 2 \* 3.14159 \* n

\* Define Soma(a,b) a + b

S = Soma(3,2);

↓ substituição de strings OK

S = 3 + 2;

S = Soma(S, 7) \* 10;

S = (S + 7) \* 10

S = 75!

"use (|||) para  
evitar esse erro"

"Som macro"  
P1OUT |= BIT0; // Acende o led  
P1OUT &= N(BIT0); // Apaga

Com MACRO:

\* Define Port Set (Port, Pin)

Port##(Port, Pin) Out | = Bit##(Pin)

PortSet(P1, 0);

P1OUT |= BIT0;

→ Port config In VP (P2, 1);

Ex. defina a MACRO ACIMA

" PxDin &= N(BITY); "

PxRen |= BITY;

PxDout |= BITY; "

\* Define port configInVP (port, pin)

" " "

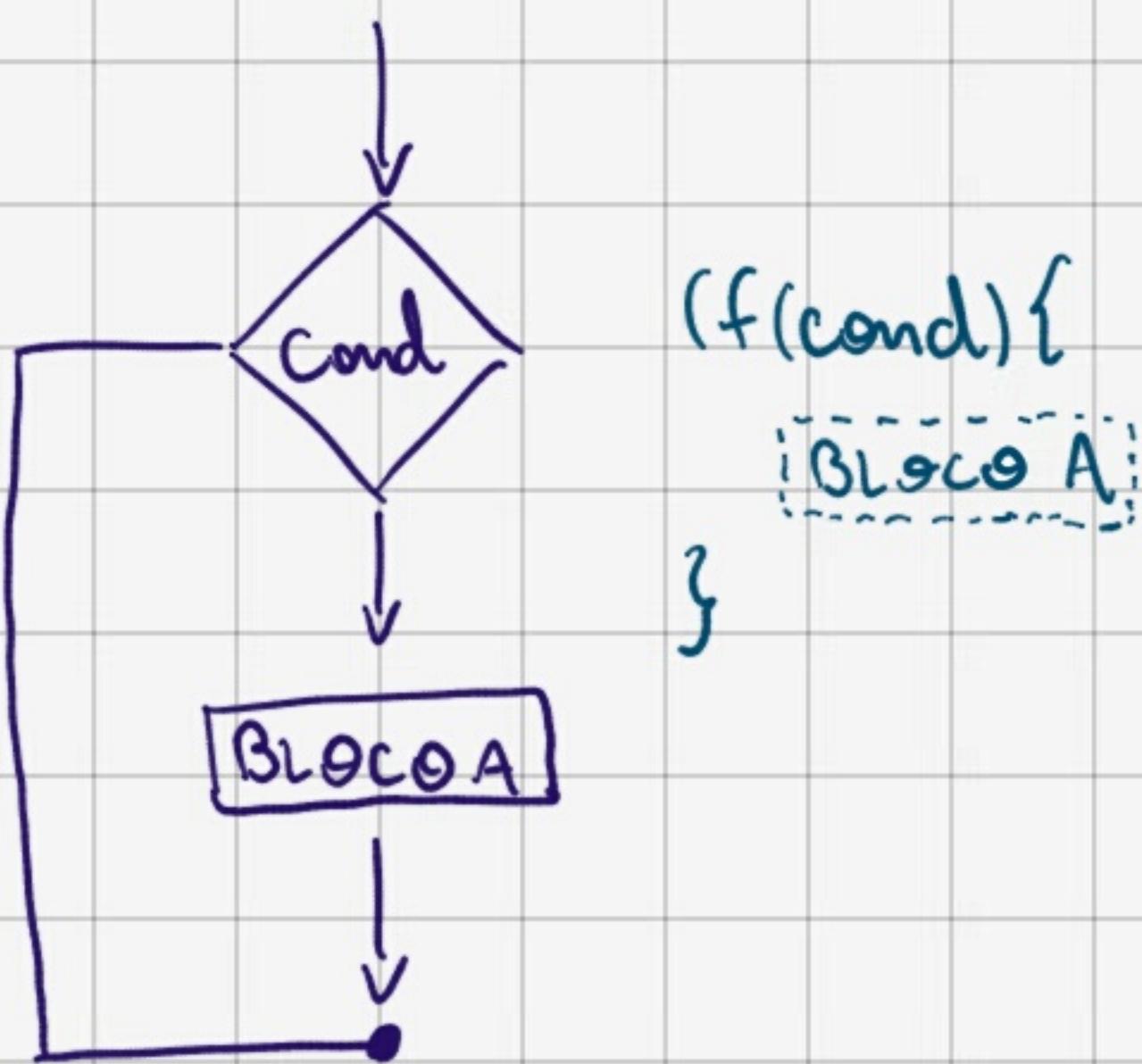
Port##(Dir) D = N(Bit##(Pin)); \

Port##(Ren) R = Bit##(Pin);

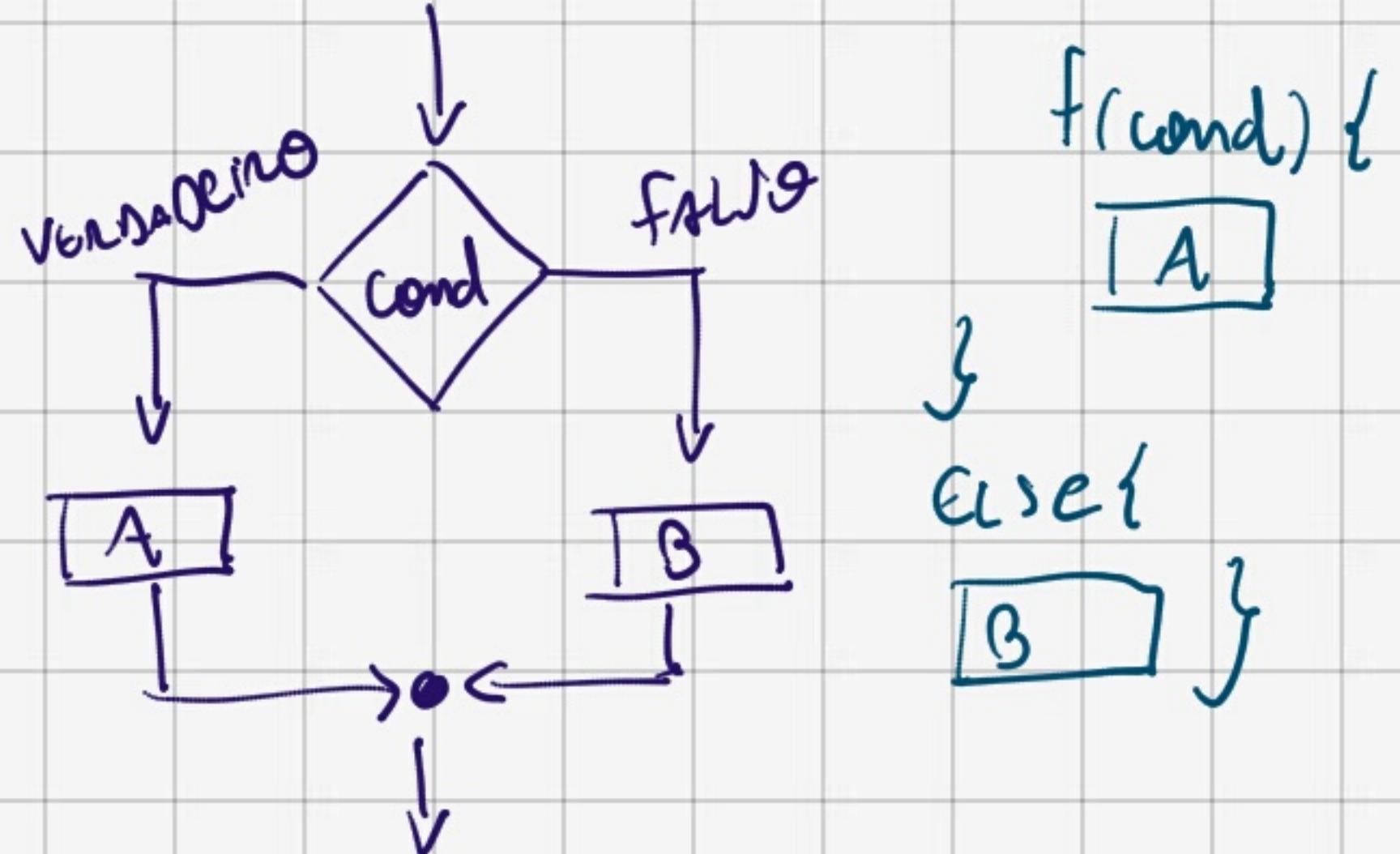
Port##(Out) O = Bit##(Pin) → Sem ponto e vírgula.

## \* Controle de fluxo: (Estruturas)

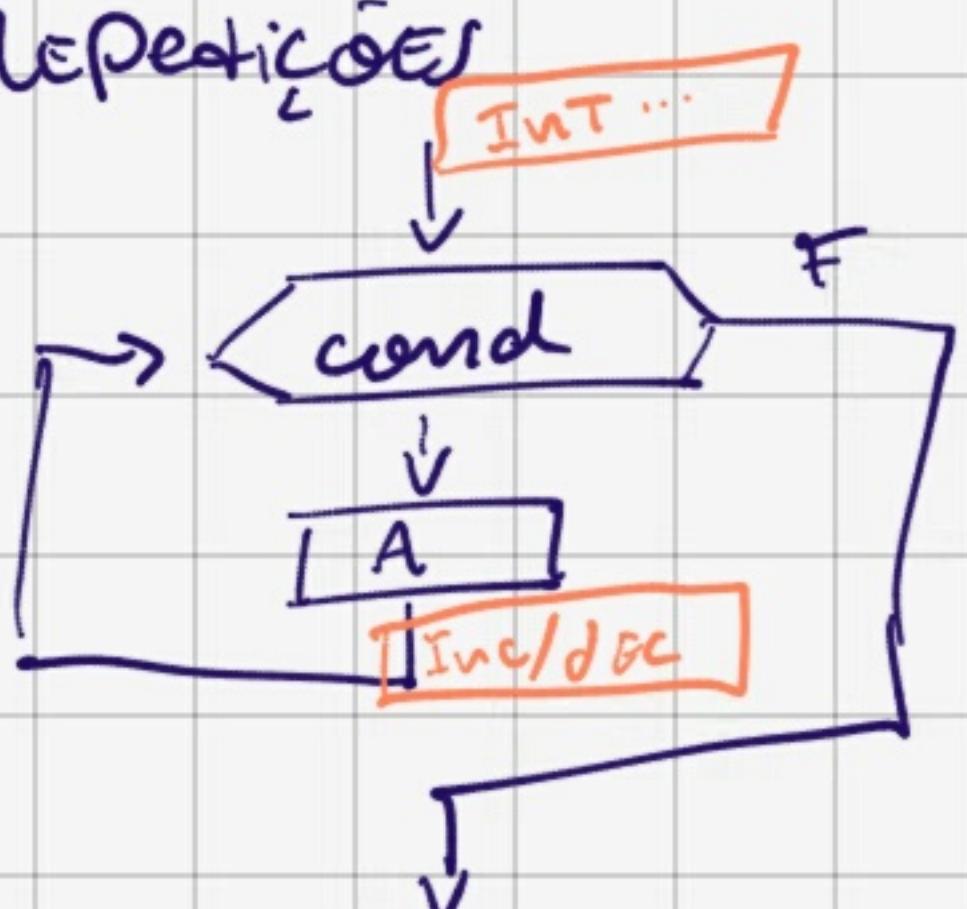
⇒ ALTERNATIVA SIMPLES:



⇒ ALTERNATIVA COMPOSTA:



## ► Repetições



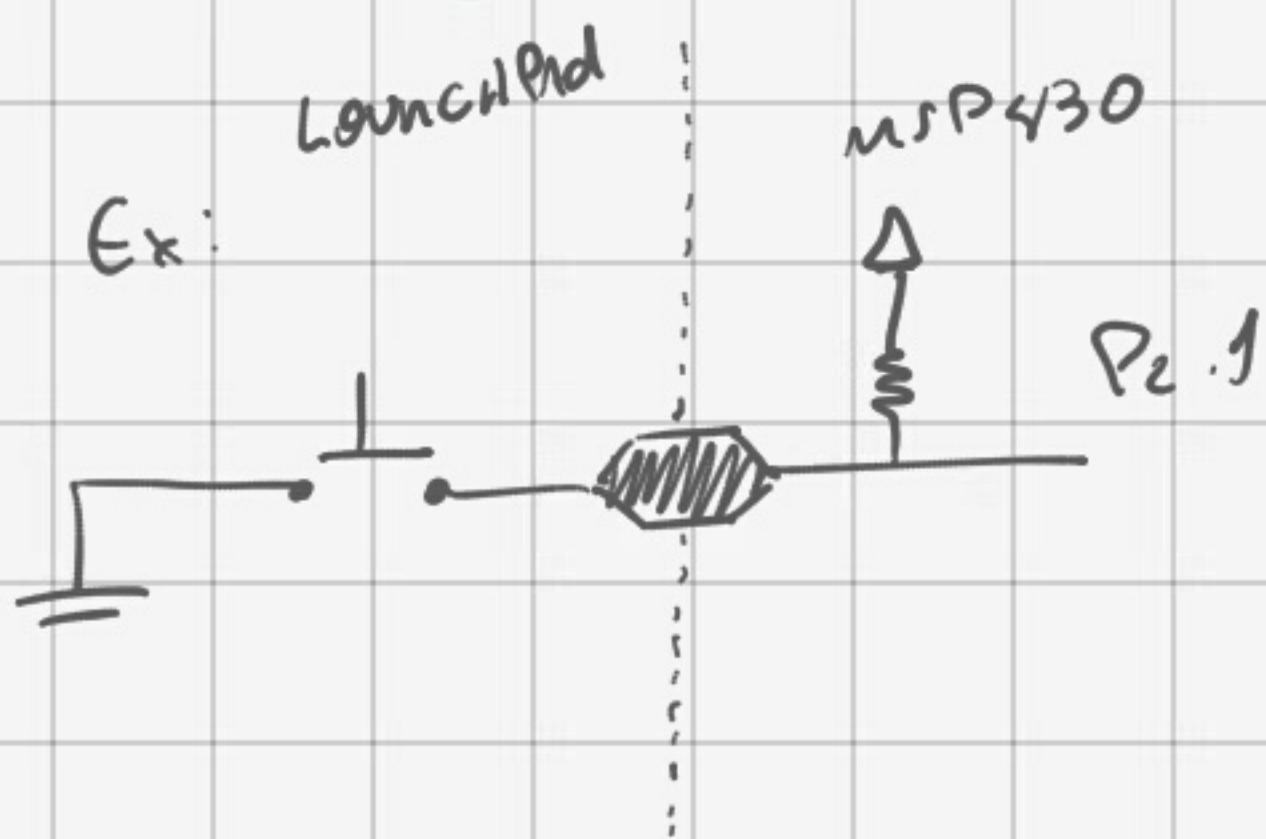
while (cond) {

}  
    | A |  
    | INT | i = 0 ; i < 10 ; i++ | cond | Inc/dec |  
    | A | }  
    | } |

INC Rn	DEC Rn
X CMP # Cte, Rn	= JNZ loop V
JEQ loop	

↳ RECOMENDAÇÕES: USE SEMPRE  
DECREMENTO AO INVÉS DE  
INCREMENTO NOSSOS CONTROLES  
DE FLUXO.

## \* Instrução à Interrupções: Amostrar pinos de entrada

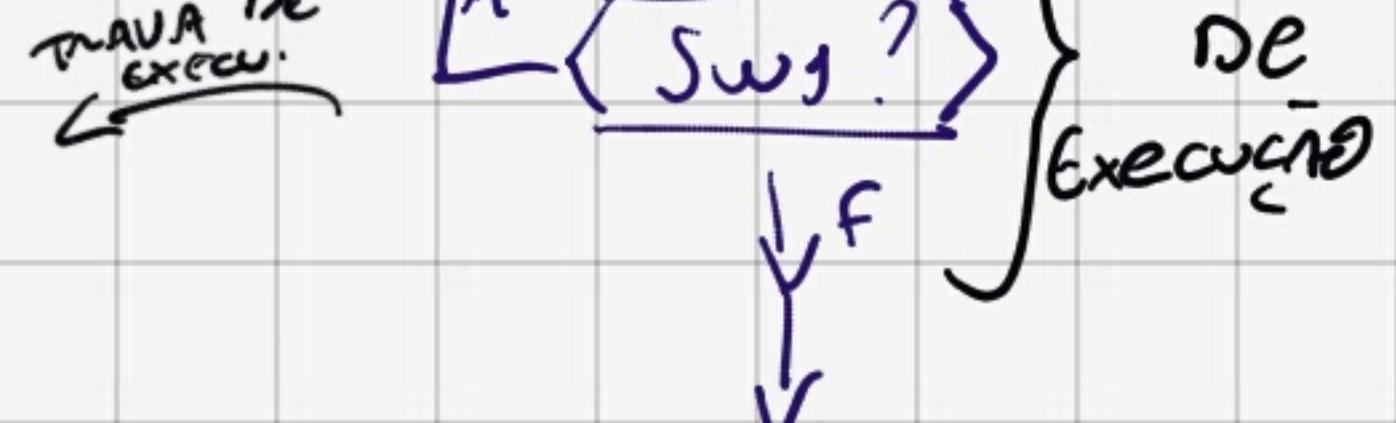


while (P2IN & BIT1);

$\begin{array}{c} \text{BIT1} \\ \text{0x02} \\ \text{0x00} \end{array}$

7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0

0010



↳ while (! (P2IN & BIT1)) i  
    ou

while ((P2IN & BIT1) == 0x00);

! → INVERSOR LÓGICO

0 → 1
1 → 0
2 → 0
-1 → 0

~ → INVERSOR BIT A BIT

0001 → 1110
0100 → 1011

## \* Interrupções:

O processo de amostragem é rápido, porém trava a CPU durante uma verificação.

∴ Interrupções permitem que uma ação seja mapeada a um evento.

### Pooling:

- já chegaram?
- já chegamos?
- :

### Interrupções:

- me avise quando chegarmos

⇒ Para casa ⇐

Pesquisar: o que são → flags de Interrupções

- habilitações de INTs
- vetor de Interrupções
- rotina de tratamento de uma INT.
- tabela de inserção de rotinas de tratamento de INT.

## # Interrupções (Cont.)

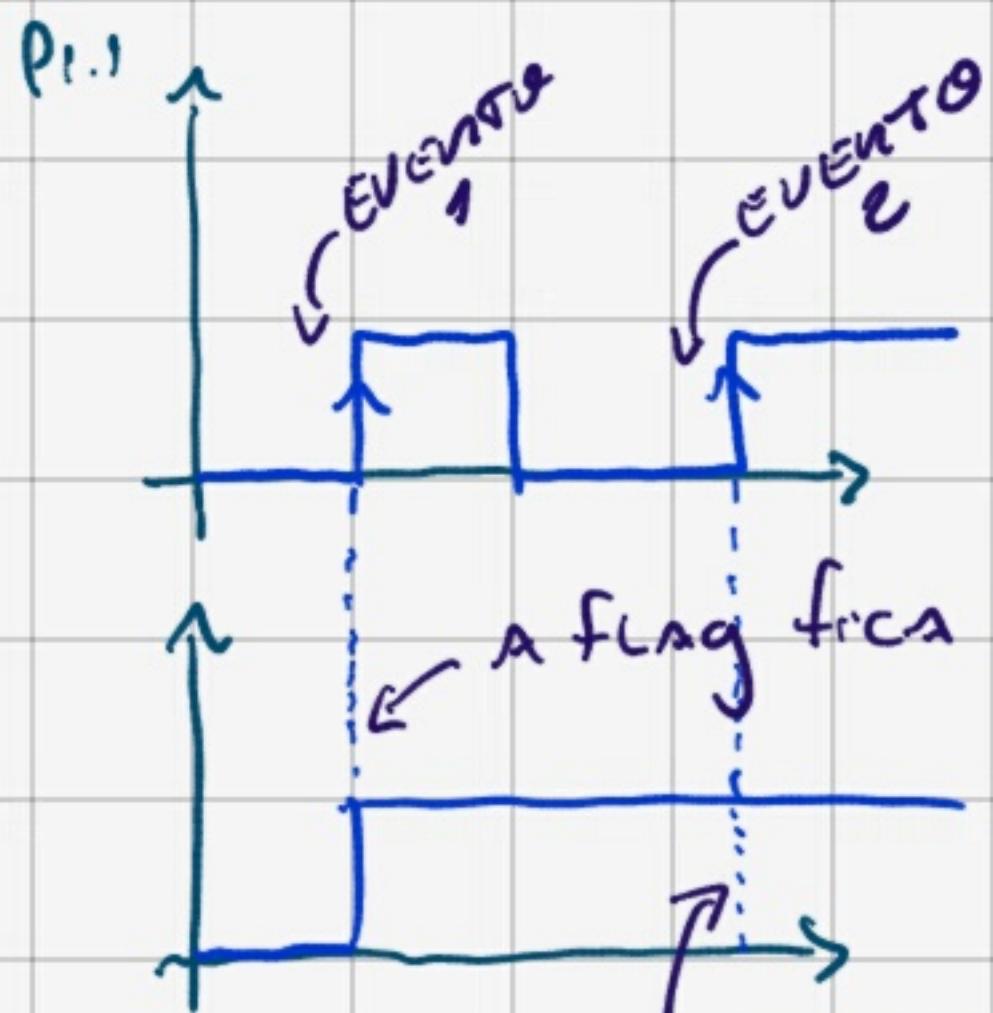
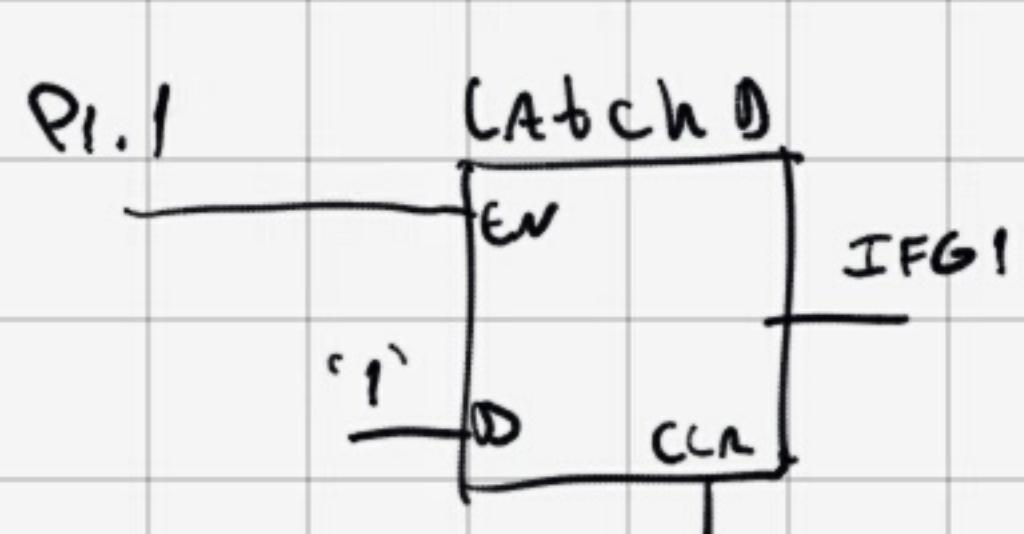
⇒ Conceitos importantes

- Interrupt flag (IFG)
- Interrupt Enable (IE)
- Interrupt Service Routine (ISR)
- Tabelas de endereços das ISRs
- Interrupções dedicadas vs Agrupadas
- VETOR ...

⇒ Interrupções...

## XX Interrupt FLAG:

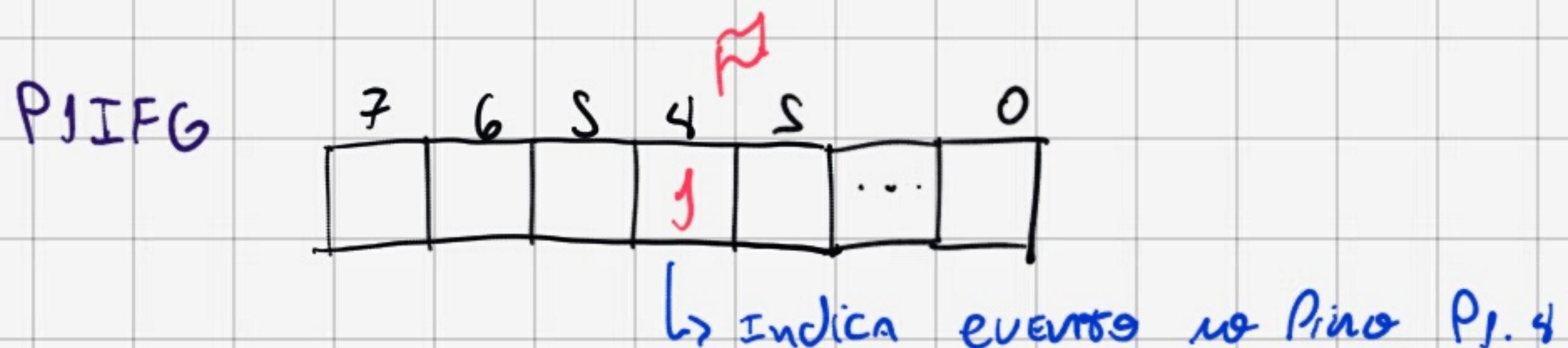
∴ Bits de registros que indicam o acontecimento de um evento de trânsito.



⇒ A flag fica "Setada" até o programa tratar a INT  
⇒ Segundo evento é perdido se a INT não for tratada a tempo.

⇒ Leitura das IFGs é feita

Em registro que agrupam diferentes "Eventos"

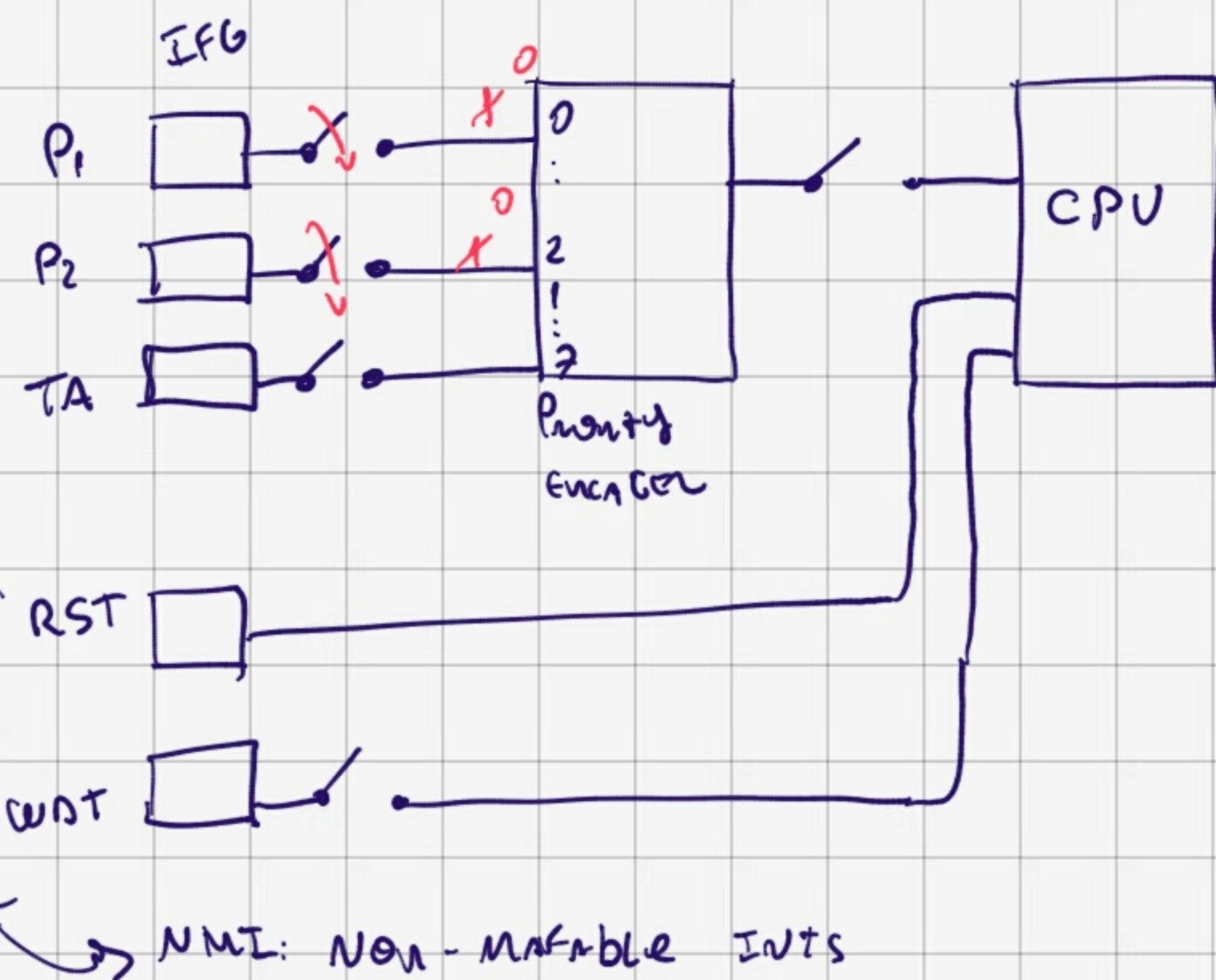


Ex: `while (! (P1IFG & BIT4)) ; // Aguarda a ocorrência de evento em P1.4`



`P1IFG &=~ (BIT4)`

## \* Interrupt Enable:



## \* Status Registers:

SR	8	7	6	5	4	3	2	1	0
	V	SCG	AC OFF	CPU OFF	CPU OFF	GTF	N	Z	C

↑

ASM: Bis.W R1T3, SR

C: —— Enable-Interrupt();

► Como é tratada uma INT?

→ IFG = 1 (por hardware)

1. NÃO HÁ PASSAGEM DE PARÂMETROS
2. OBRIGATÓRIO SALVAR O CONTEXTO
3. (PC+2) É SALVO NA PILHA (SR) É SALVO NA PILHA
4. SR=0 : IMPEDE QUE OUTRAS INTs INTERROMPAM A CPU ENQUANTO UMA INT ESTIVER SENDO TRATADA.
5. ISR É EXECUTADA : DEVE SER CURTA!

(↳ rotina DE TRATAMENTO DE INT

↳ IFG = 0

6. RESTAURA O CONTEXTO

7. RETORNA DA INT (RETJ)

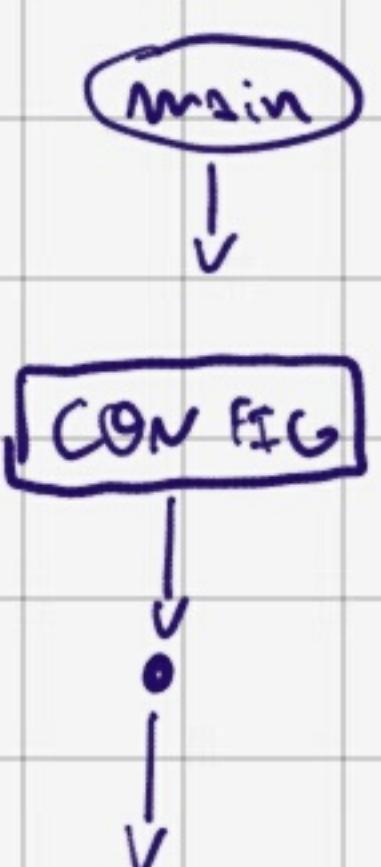
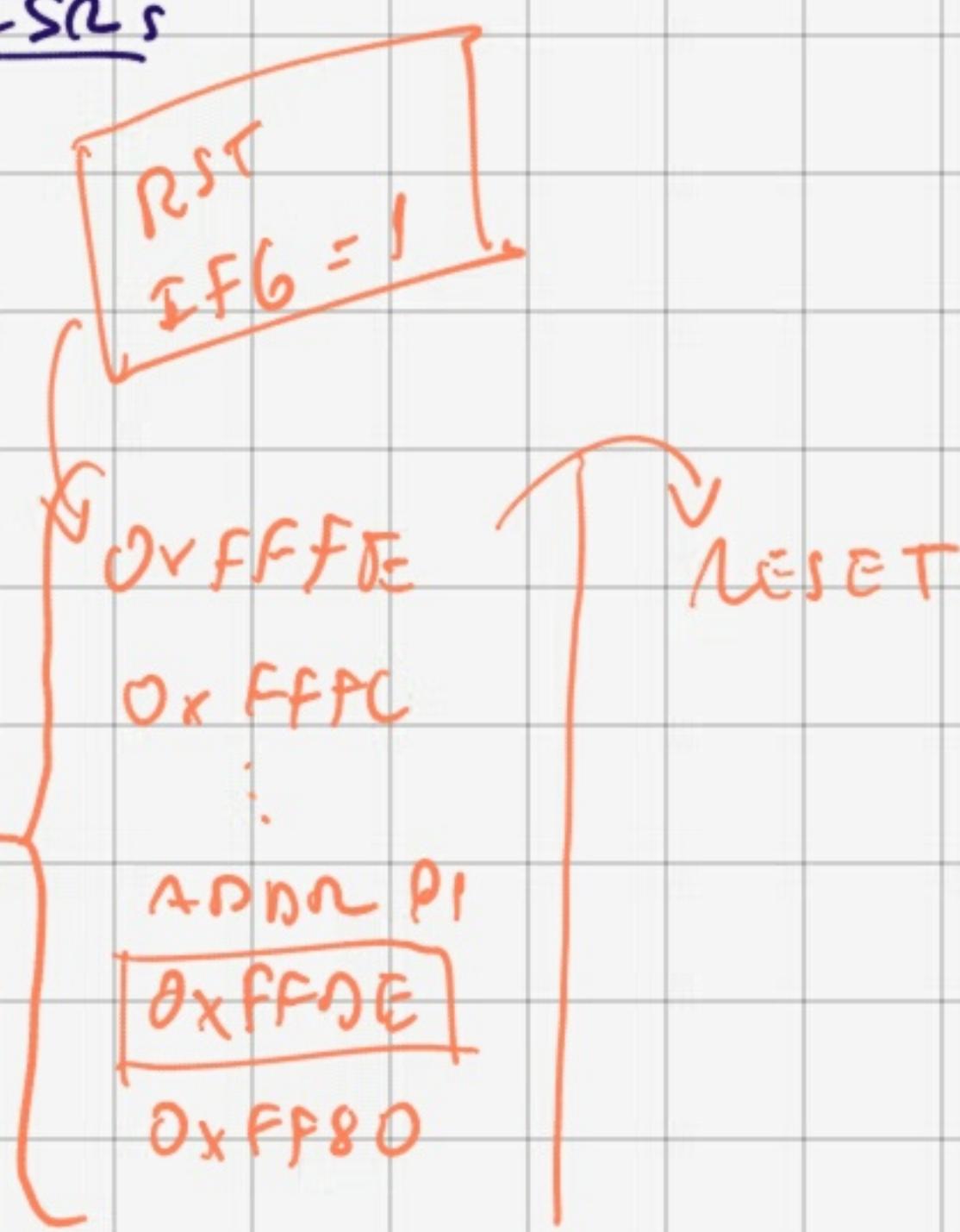
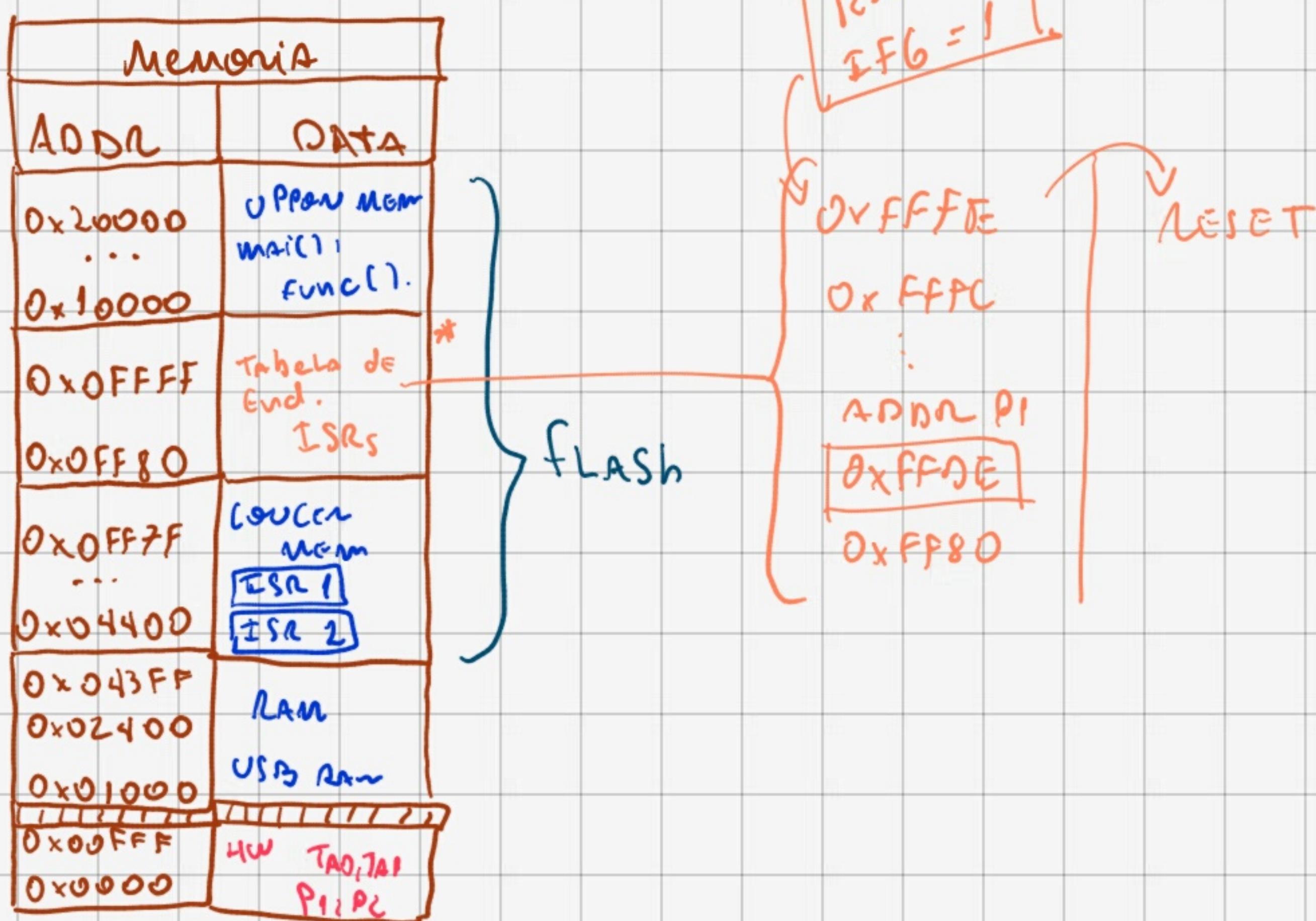
NET → POP PC →

→ MOV @SP+, PC

RETJ → TOS → SR

TOS → PC

## \* Tabela de Endereços das ISRs



## \* Exemplo completo:

Detectar o pressionar do botão J11 por INT.

```
void config() {
    P2DIR = ~BIT1; // Input
    P2REN = BIT1; // habilita resistor
    P2OUT = BIT1; // Pull-up

    PJIE = BIT1; // habilita INT
    PJIES = BIT1; // flanco de descida
    PJIFG = 0;

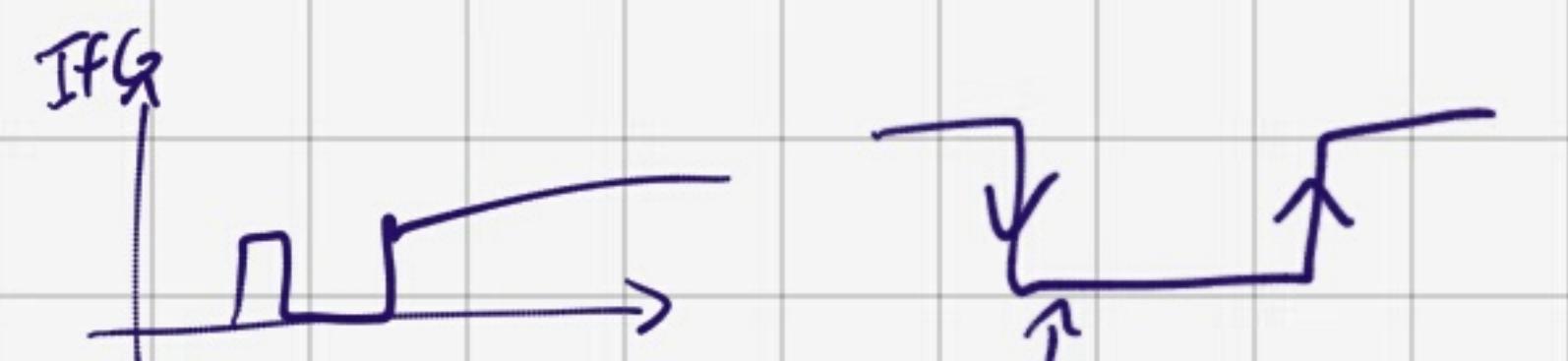
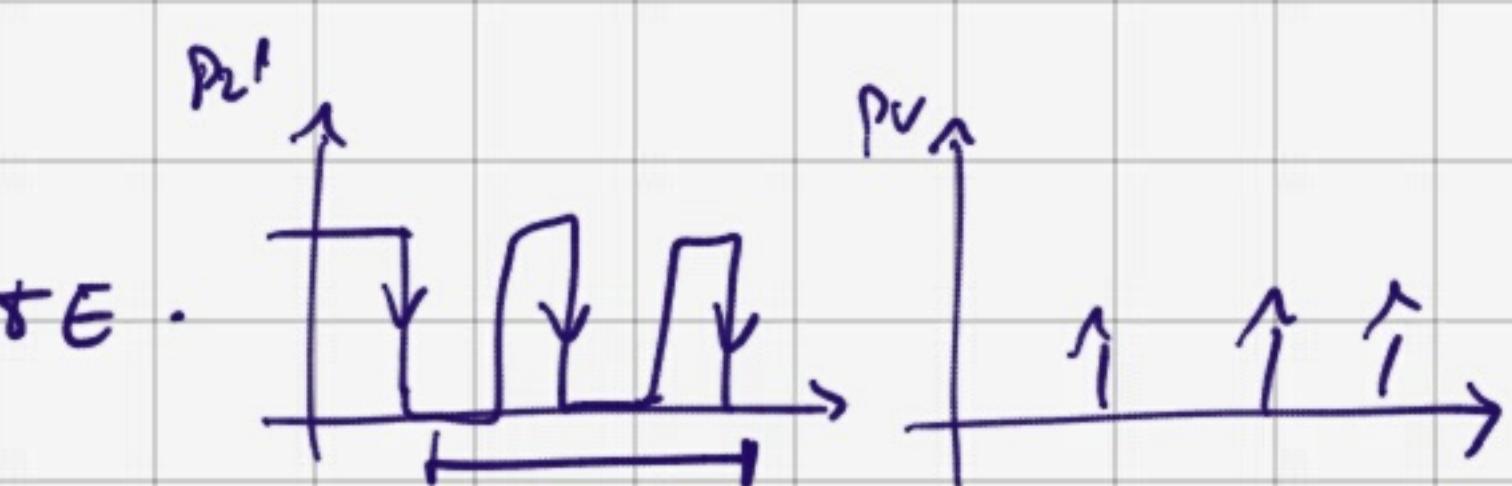
    /* enable interrupt(); */
}
```

```
void main() {
    config();
    while (1) {
    }
}
```

\*\* Programa Vector = P0T2 - Vector  
- interrupt void P2ISR() { ← End \*

```
debounced → [ACAO]
P2IFG &= ~BIT1;
}
```

⇒ Incluir uma rotina para resolver o rebote.



P2IES = BIT2;

if (P2IES & BIT1) {

[APONTAR para o Botão]

} ELSE {

[Soltar o Botão]

}

## \* MÓDULO 2: temporizadores e Interrupções

### \* Configuração de I/O:

// uso da forma compacta para configurar a porta P1.0

```
P1DIR |= BIT0; //
```

// Criando um loop (apenas para fins didáticos, não é necessário)

```
while(1){
```

```
    P1OUT ^= BIT0; // Altera o bit P1.0 (LED VERMELHO)
```

```
}
```

BASICO

Programma  $\Rightarrow$  Acender o led vermelho P1.0 e apagá-lo

### \* include <msp430.h>

```
int main() {
```

```
    WOTCTL = WOTPW | WOTHOLD;
```

// Configurar pino P1.0

```
P1DIR |= BIT0;
```

// Loop

```
while(1){
```

// toggle XON

```
P1OUT ^= BIT0; // Altera o bit P1.0 se 1  $\leq$  0
```

```
}
```

```
} return 0;
```

Programa Basico 2  $\Rightarrow$  Configurar P4.7 (Led Verde)  $\rightarrow$  com pull-down

\* include <msp430.h>

```
int main () {  
    WOTCTL = WOTPW | WOTHOLD;
```

// config. porta P4.7 com Pull-Down

```
P4DIR |= BIT7; // Zera bit 7 no P4DIR  
P4REN &= ~BIT7; // Ativa o resistor  
P4OUT &= ~BIT7; // Seleciona o resistor Pull-down
```

// Loop

```
while (1) {
```

```
    P4OUT ^= BIT7;
```

```
}
```

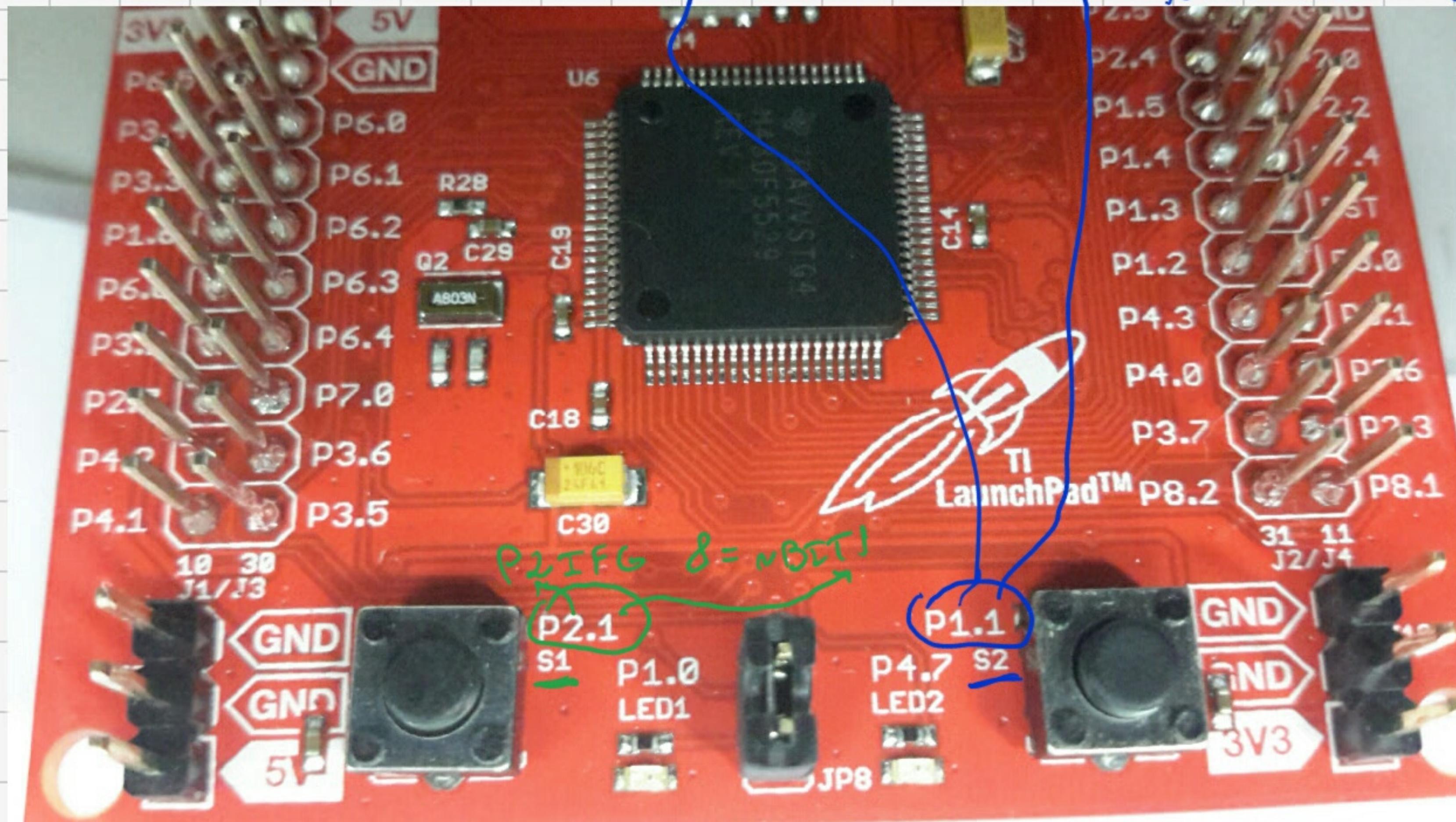
```
return 0;
```

$\Rightarrow$  Usando os botões da placa

" P2IFG &= ~BIT1; // (Botão S1) "

P1IFG &= ~BIT1; // (Botão S2)

NO CASO DO S2



## Ejercicio 1: (modulo 2)

LED verde P4.7 Ascenso gira Botão S1 é precionado //

```
int main(void) {  
    // Desliga côn de guarda  
    // Conf. entradas
```

```
P4DIR |= BIT7; // Zera led verde (configura Pino)  
P2DIR &= ~BIT1; // "Zera" botão S1 (configura S1)  
P2REN |= BIT1; // habilita a resistor pull up/pull down  
P2OUT |= BIT1 // Ativa pull down
```

// loop

```
while(1){
```

```
    if((P2IN & BIT1)==0){ // verifica se o botão S1  
        // está precionado
```

```
        P4OUT |= BIT7; // Liga o led verde
```

```
}
```

```
    else{
```

```
        P4OUT &= ~BIT7;
```

```
}
```

```
}
```

```
return 0;
```

```
}
```

## Ex:2) Ruido de chave mecânica

Altere o estado do led vermelho tanta vez que o botão S1  
fique precionado. (Não remova os rebites)

→ mesmo código anterior, com algumas alterações

• • •

```
P1DIR |= BIT0;  
P1DIR &= ~BIT1;  
• • •
```

```
while(1){
```

```
    if((P2IN & BIT1)==0){
```

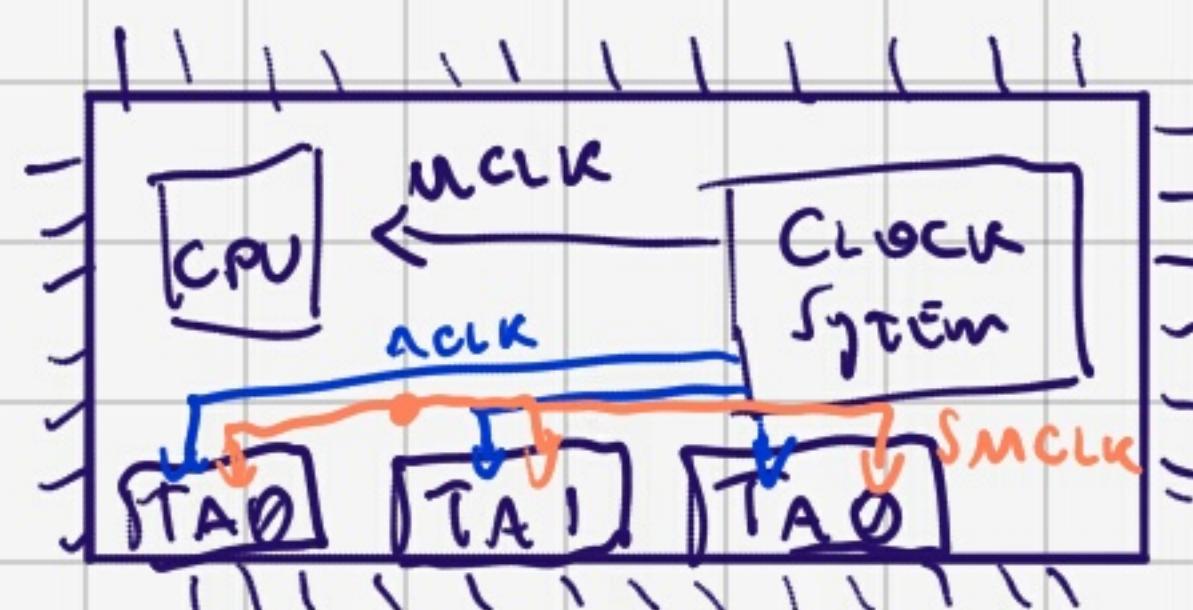
```
        P1OUT |= BIT0 // Ativa o
```

```
// sem o else
```

ESTADO DO LED

Ex 3) Memória de rebotos

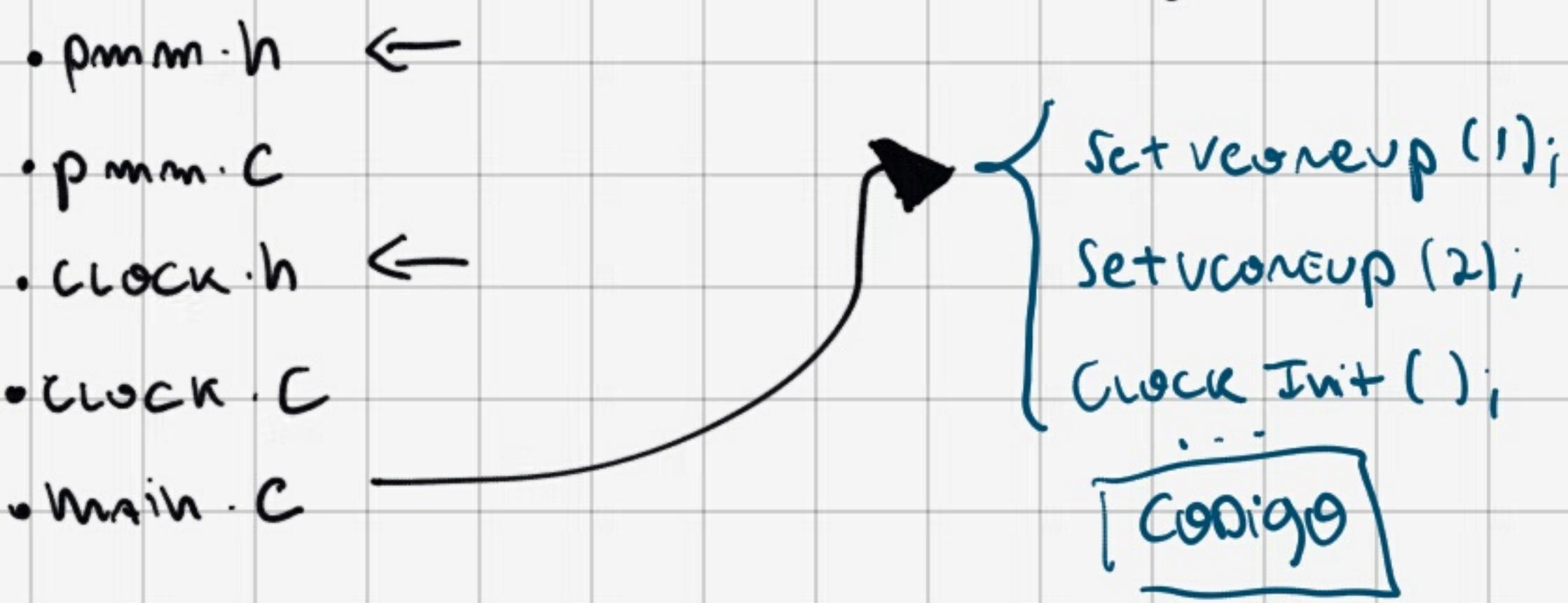
- \* temporizadores (Timers)
- contadores
- referências de tempo
- comparadores



⇒ Referências de tempo:

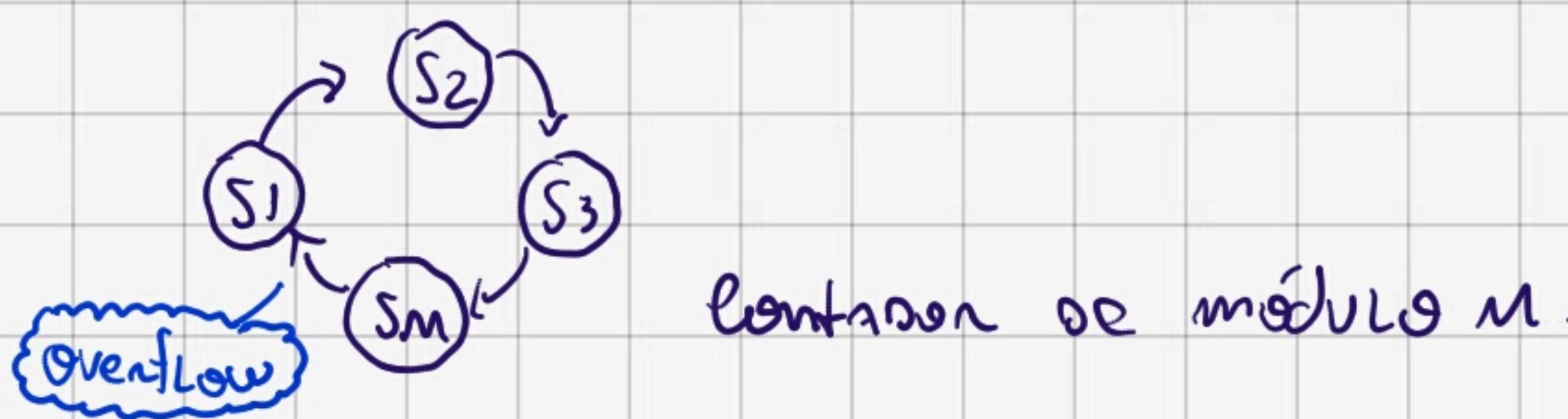
- MCLK: master CLOCK : 16 MHz
- ACLK: Auxiliar CLOCK : 32768 Hz
- SMCLK: Submaster CLOCK : 1 MHz

↳ Baixar o template Setclock.zip



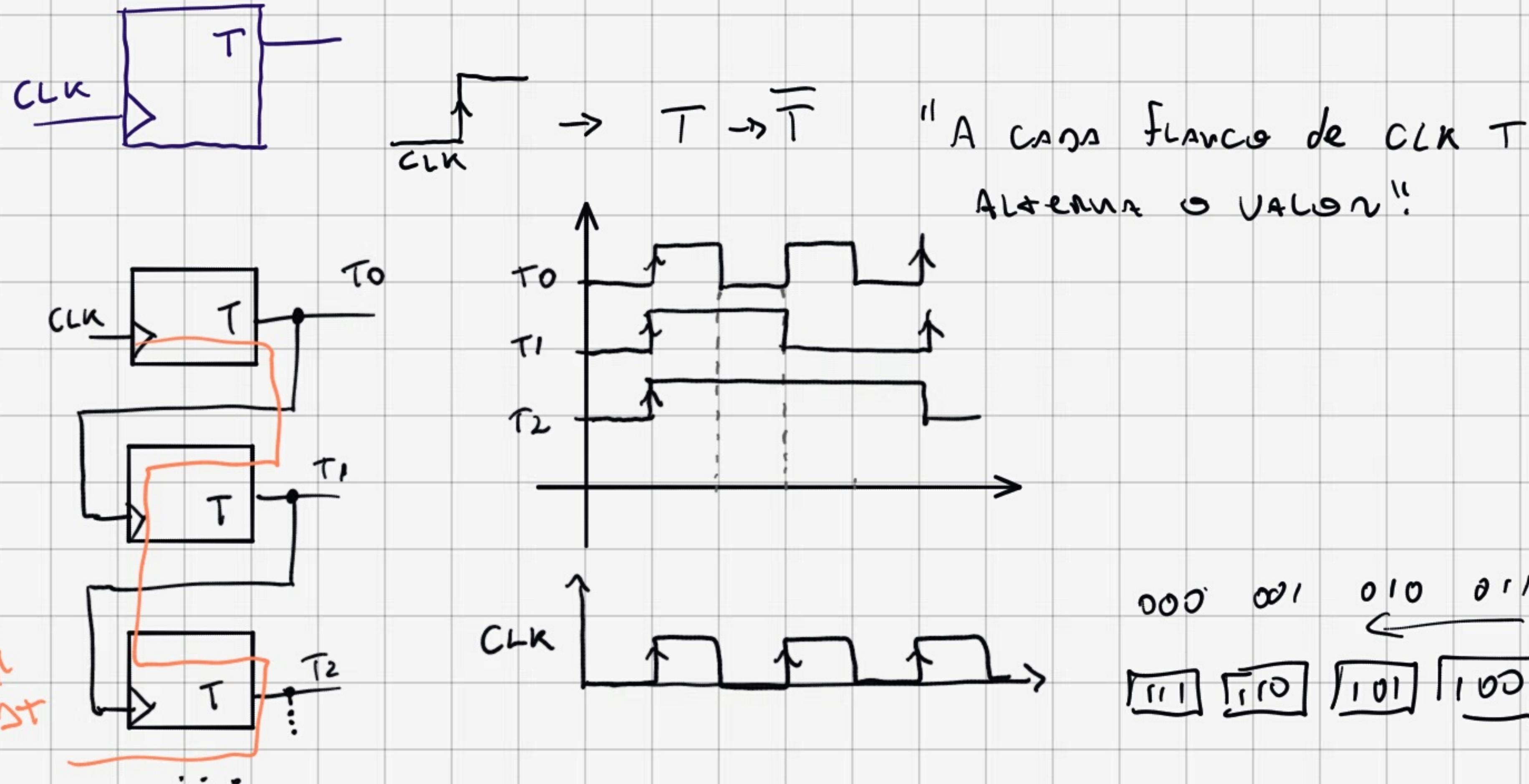
⇒ Contadores:

Máquinas de Estados Ciclicos

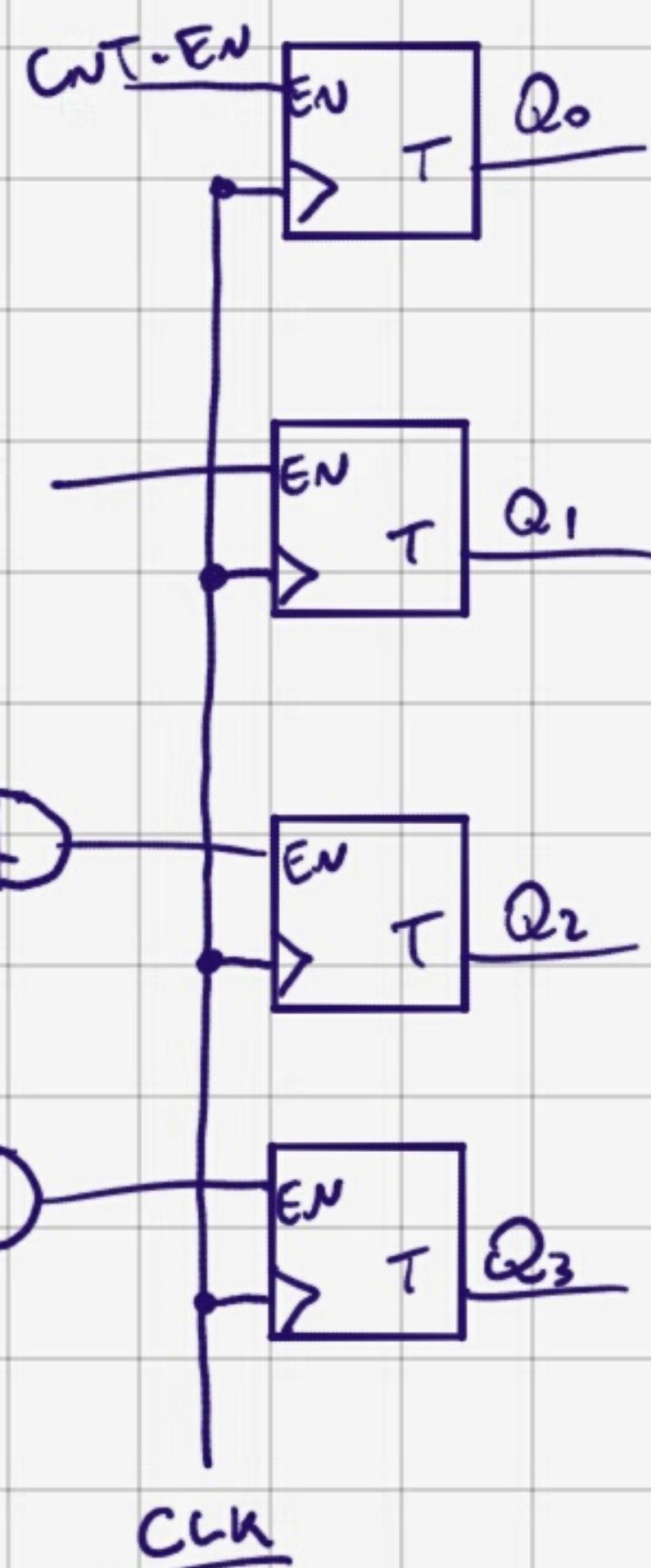
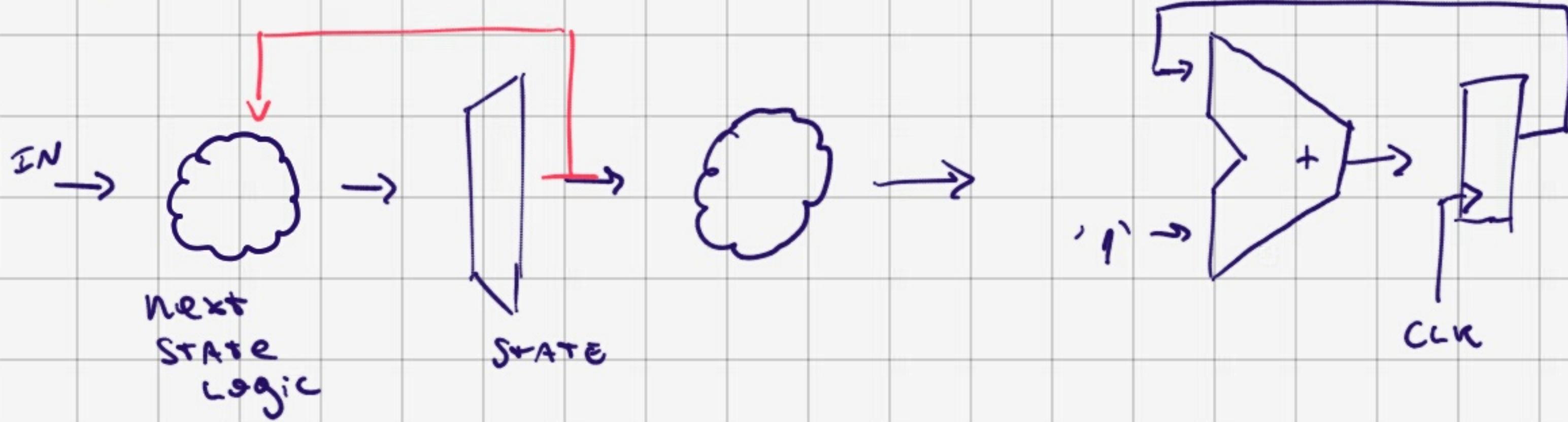


Contador de módulo m.

→ flip-flop toggle



## Design em FSM:



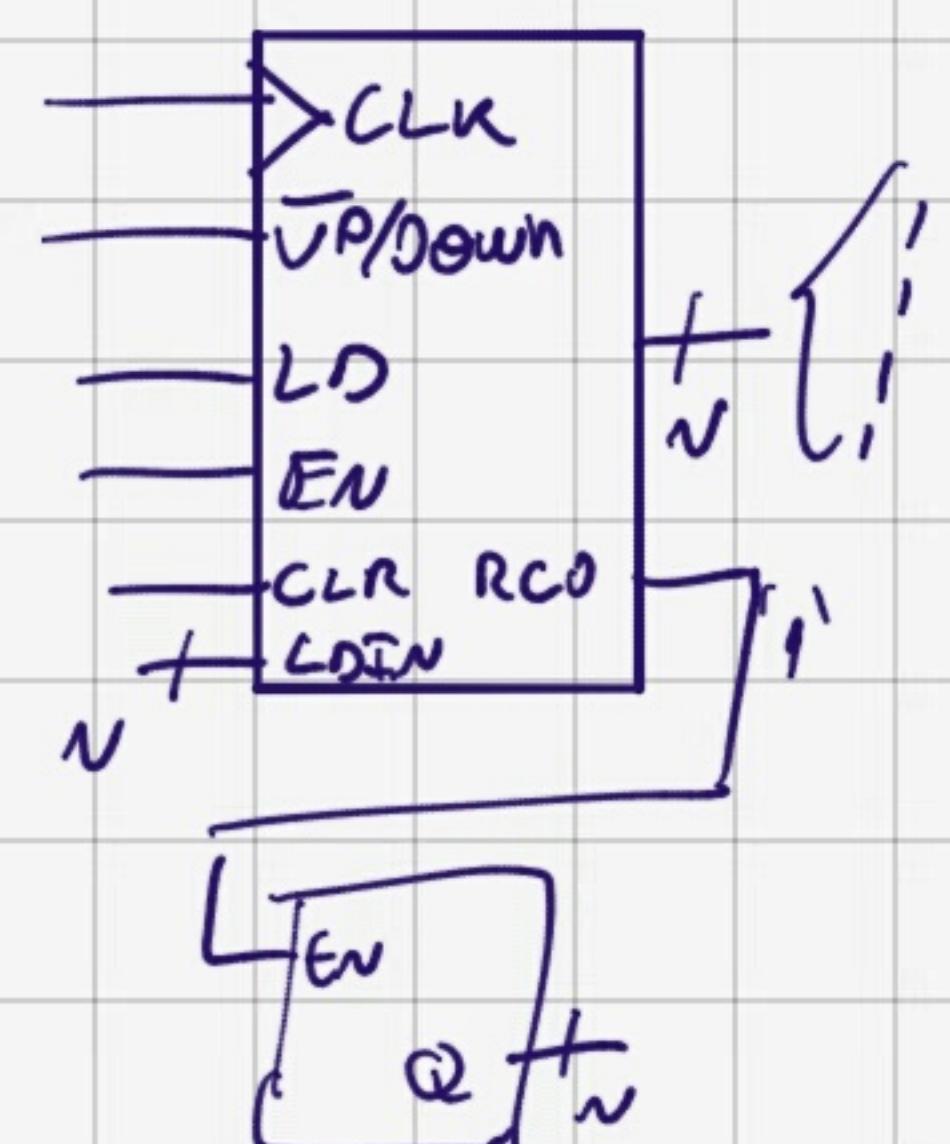
Contagem binária

000
001
010
011
100
101
110
111

1000

Contadores comerciais

74163, 74169



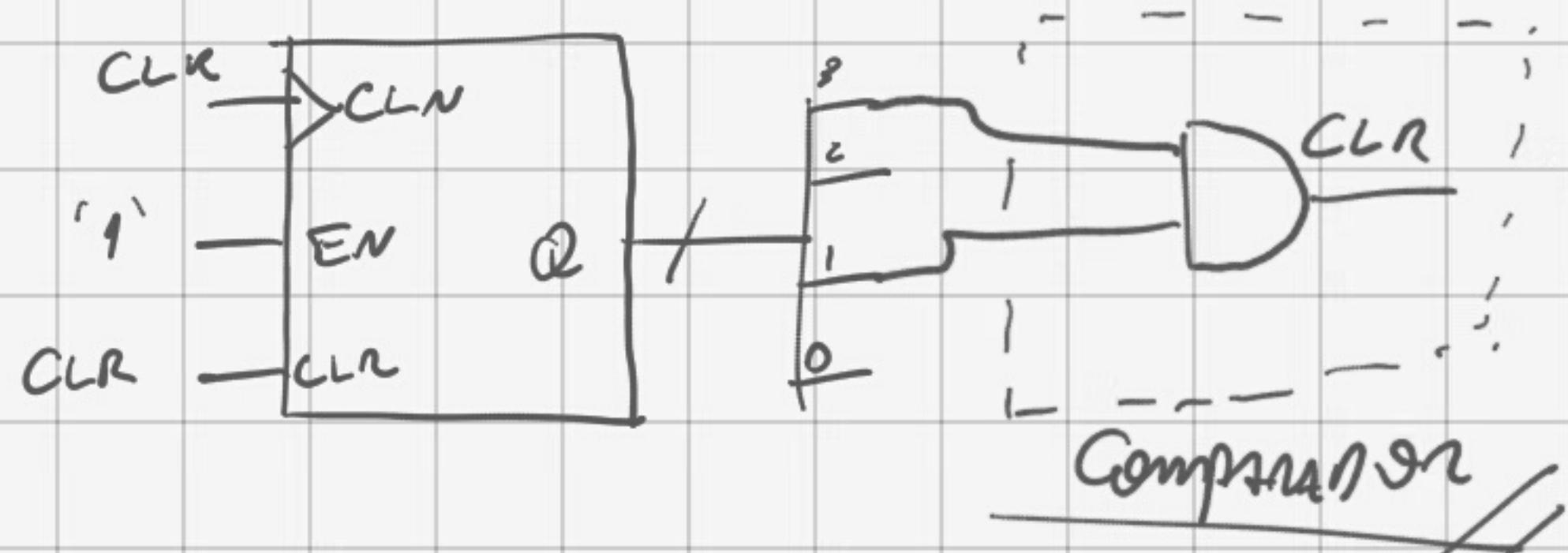
Ex: Usando contadores comerciais, crie um contador de  $[0-10]$ .

→ Como é representado  $(10)_{10}$  em binário?

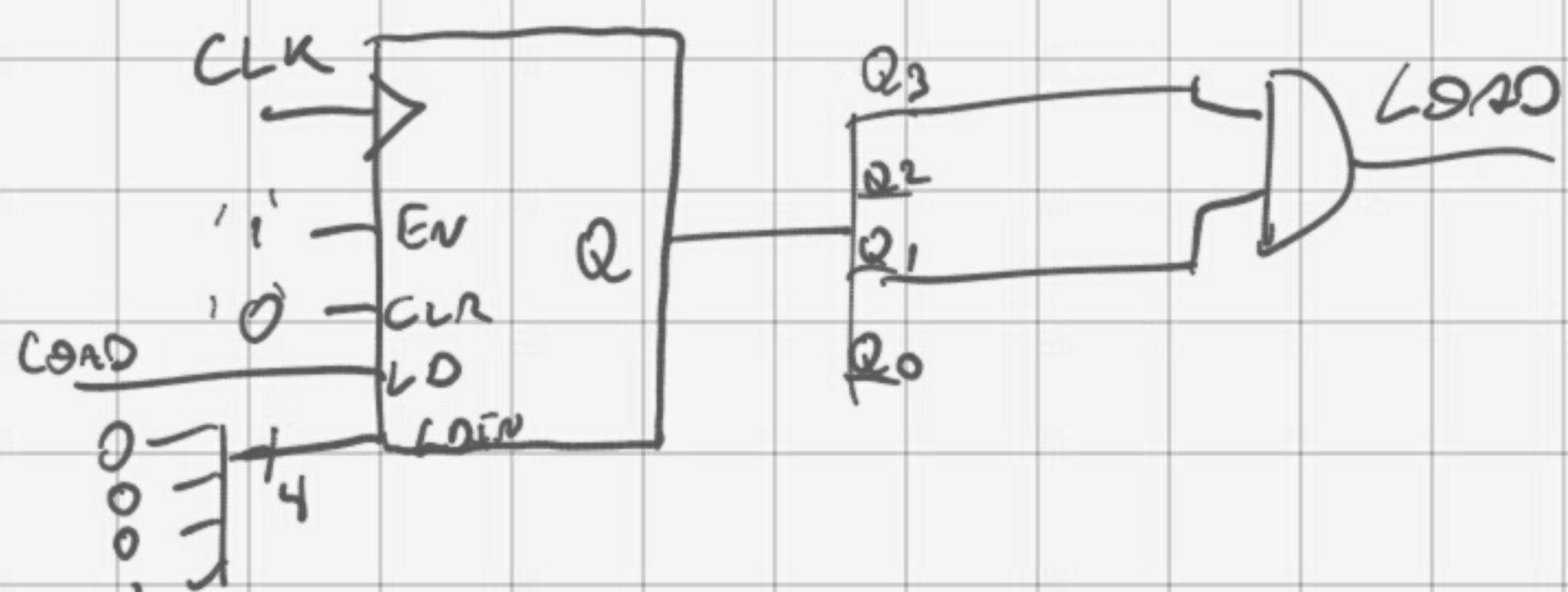
→ Como usar essa condição ( $Q=10$ ) para reiniciar a contagem?

$$Q_3 Q_2 Q_1 Q_0 = (10)_{10} = (1010)_2$$

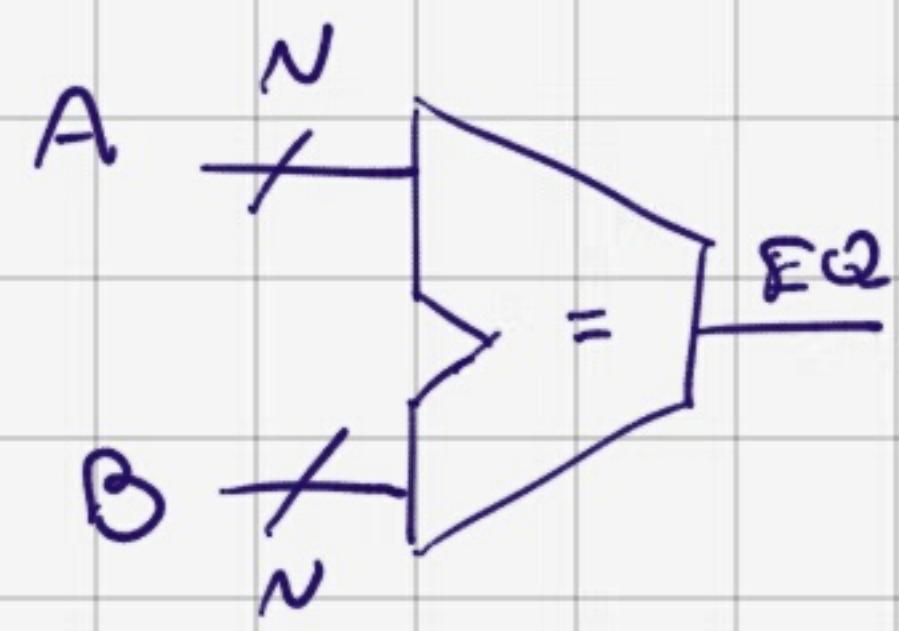
$$(Q_3 \text{ e } Q_1) = '1'$$



Ex 2: Contar [1-10]



→ Comparador de uso geral:

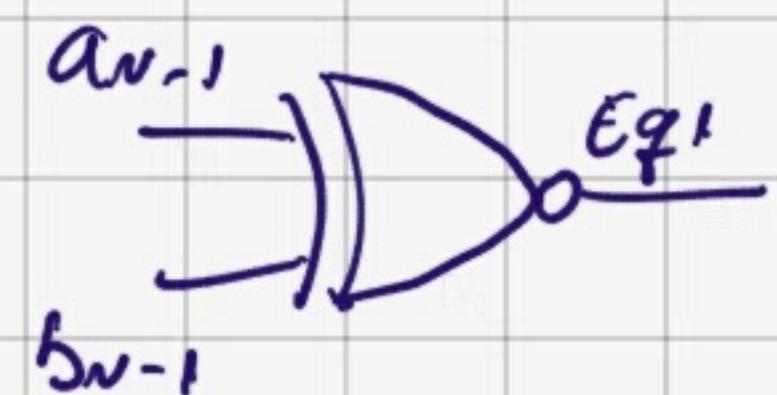
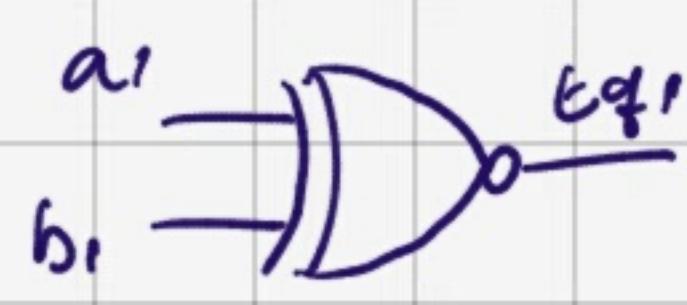
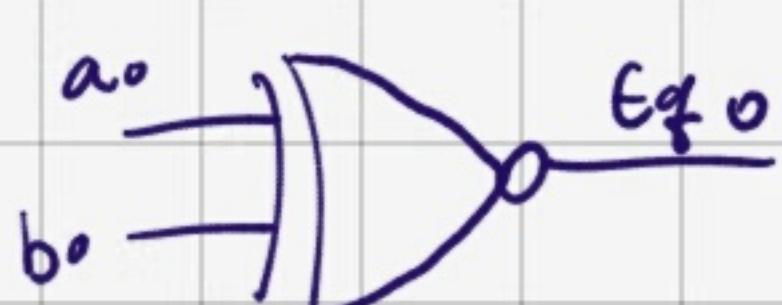
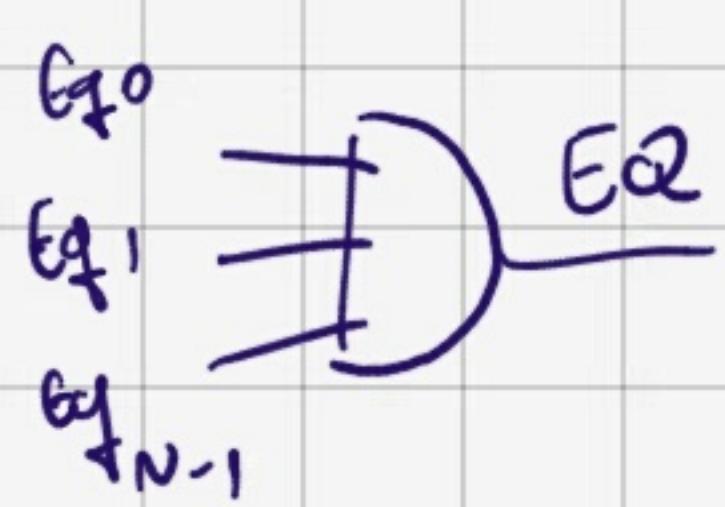


$$EQ = 1 \text{ quando } A = B$$

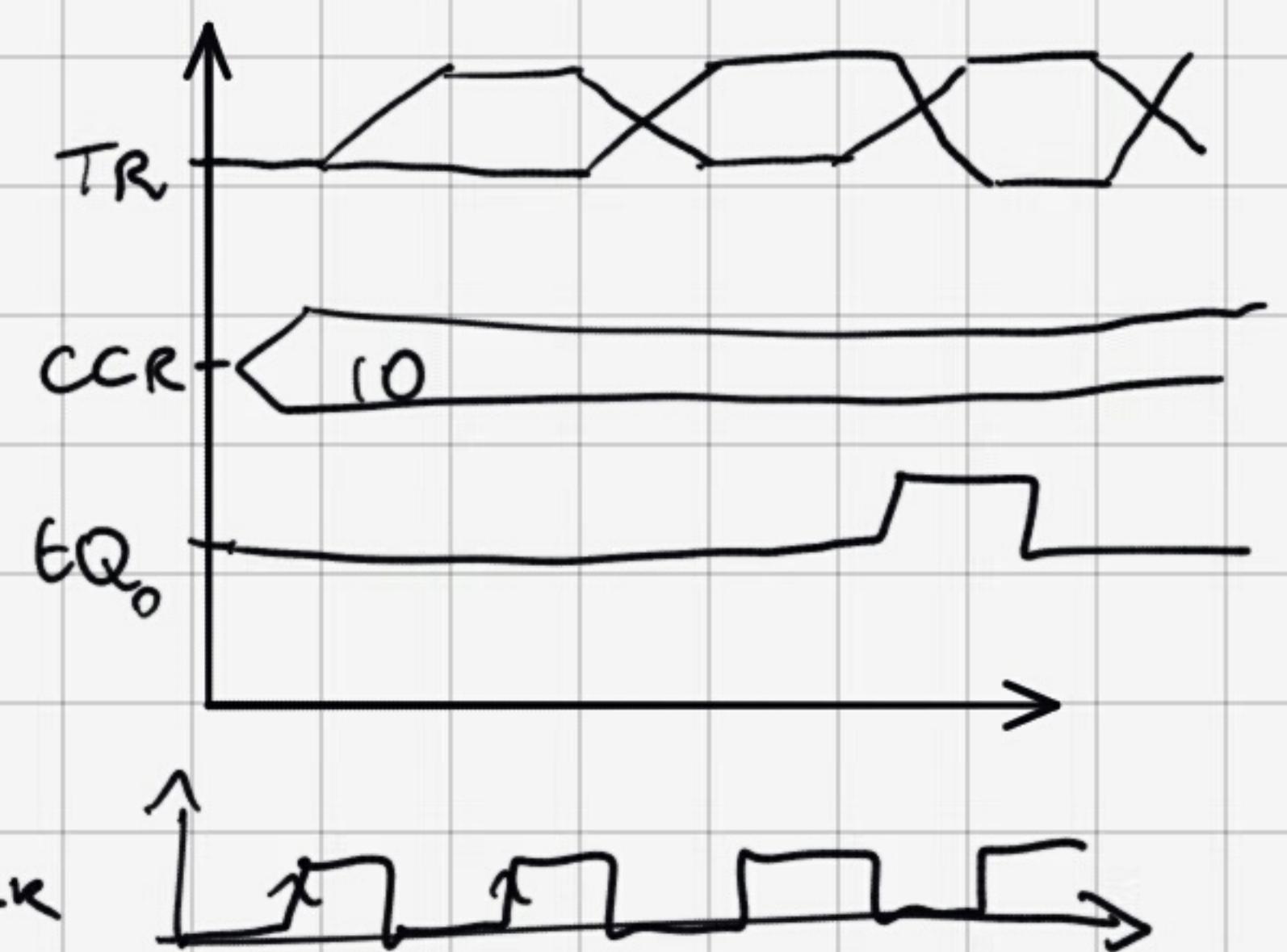
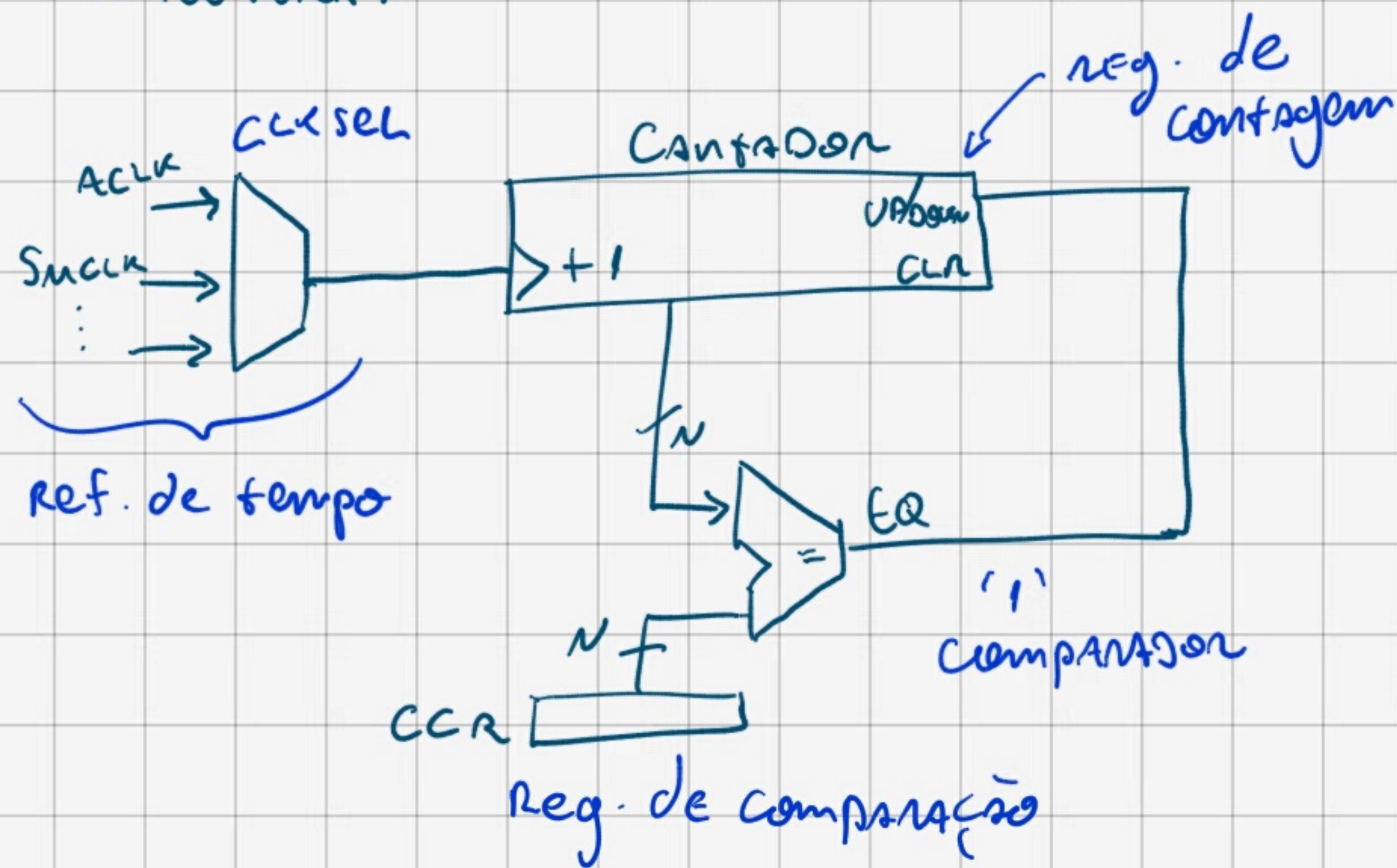
a	b	$x_{0n}$	$x_{Nn}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

$$A = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{N-1} \end{bmatrix}$$

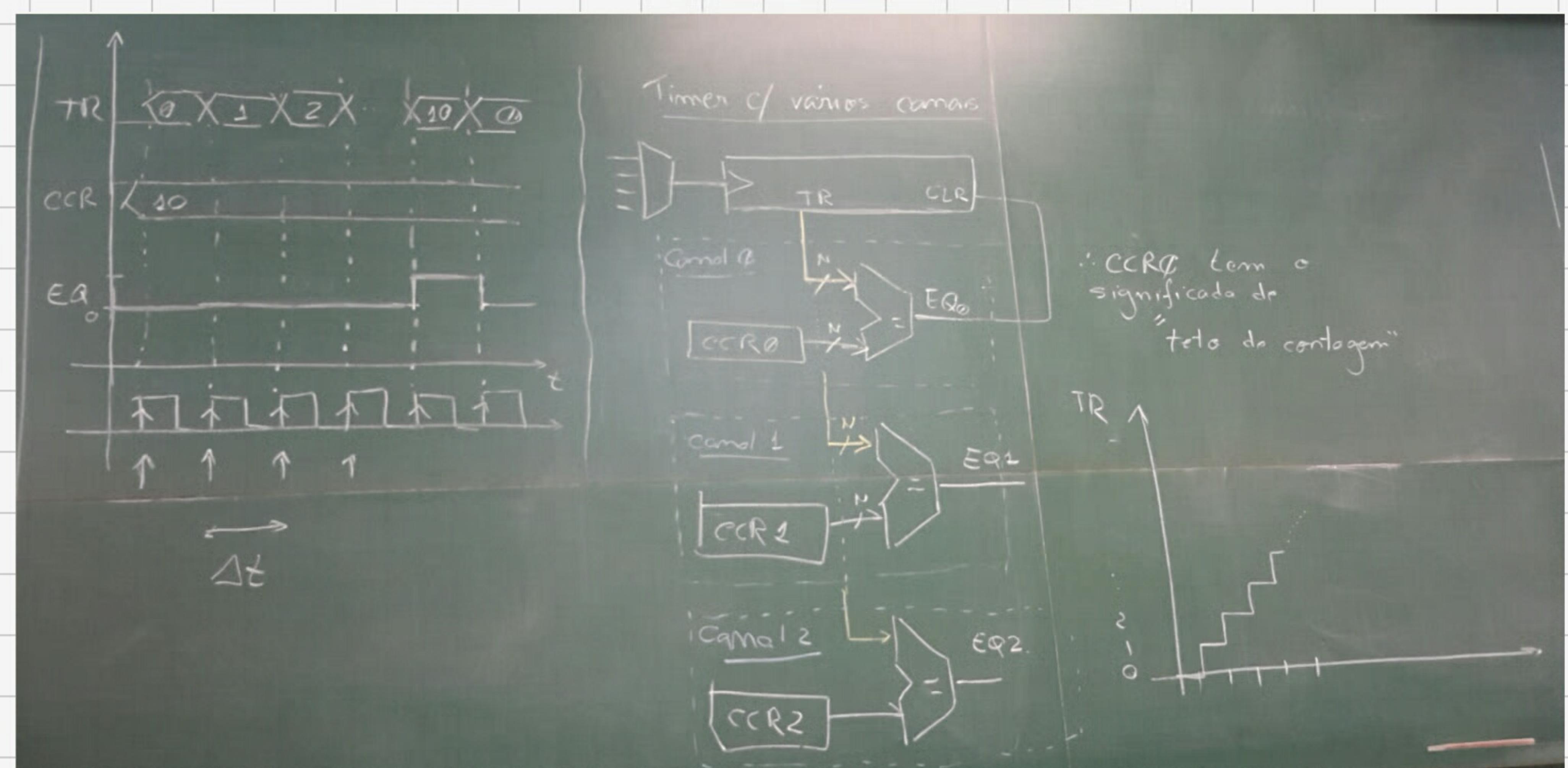
$$B = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{N-1} \end{bmatrix}$$

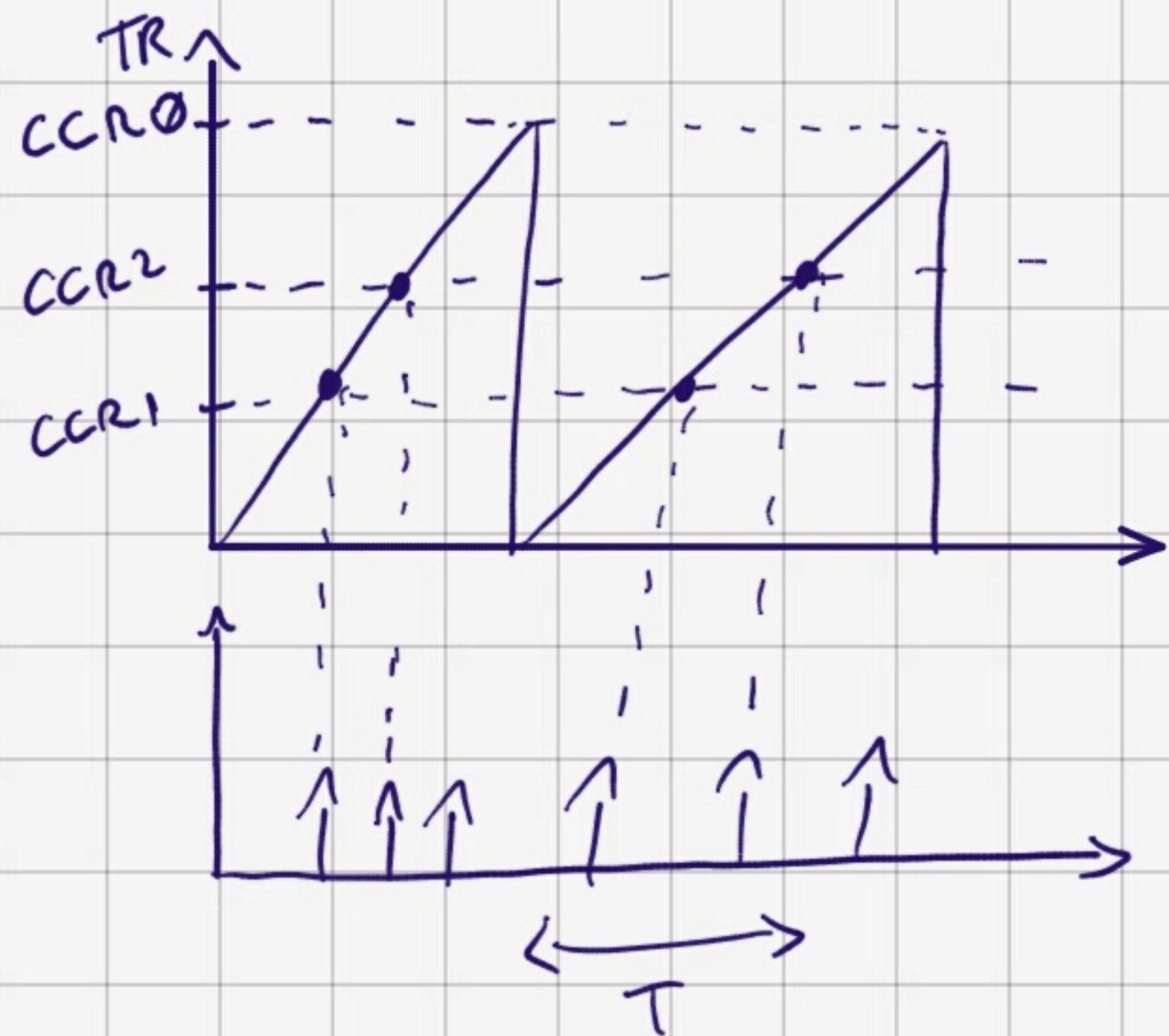


< Entradas >



timers e/ vários canais:





"Canal 0 tem CMA ISR só p/ele"

"Canais 1 → N Estão agrupados  
numa ISR"

### \* Config. Específica do MSP430

TA0, TA1, TA2

↳ Contador: TA0R

TA0CTL

Comparador: 

TA0CCR0
TA0CCTL0

 CANAL 0

TA0CCR1
TA0CCTL1

 CANAL 1

TA0CCR2
TA0CCTL2

 CANAL 2

TAXCCRY

TAXCCTLY

↑

bit0 → IFG

### \* Modos de Contagem:

MC -- HOLD → 0

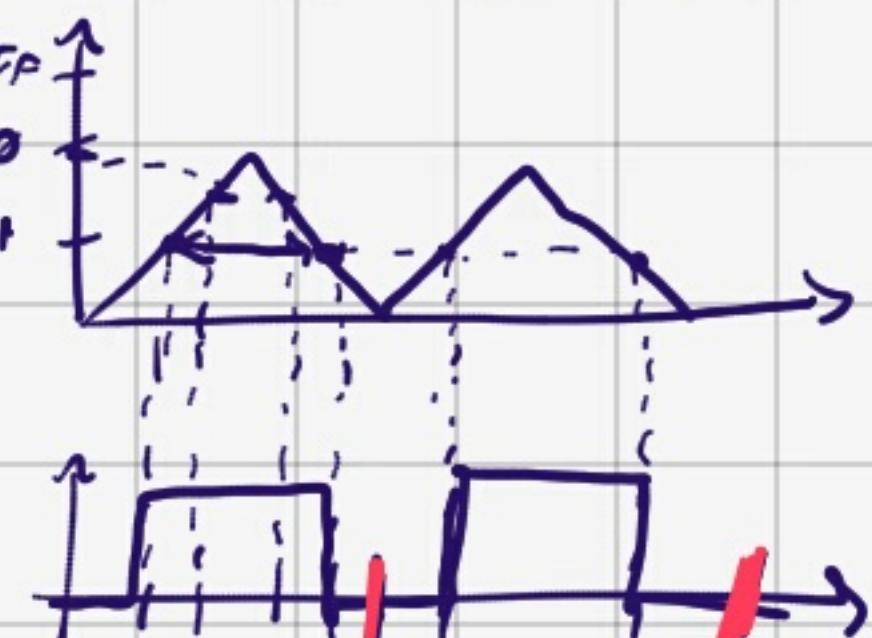
MC -- UP → 1



MC -- CONTINUOUS → 2



MC -- UPDOWN → 3



Configuração de Timer : [0 - 10]

```
ConfigTimer() {  
    // Configuração  
    TA0CTL = TASSEL_2 + ACLK + MC_0 + UP + TACLR;
```

// Comparadores

```
TA0CCR0 = 10;
```

```
}
```

```
main() {
```

```
    configTimer();
```

```
    while(1) {
```

```
        while (!(TA0CCTL0 & TAIFG));
```

```
        P1OUT |= BIT0;
```

```
        TA0CCTL0 &= ~TAIFG;
```

```
}
```