ARQUI

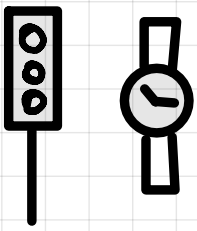
IIC 2343

• CLASE 1

• Correo: • yadran@.ing.puc.cl • yadran@uc.cl

7 Ago 23

• **Sistemas Embedidos**: • Estan dentro de otros Dispositivos.



• No son programables

• Procesadores no se comparan al de un celular o PC

• **PMD**: Dispositivos inalambricos PEQUEÑOS

↳ PC, Servidores y todos estos tecnologias comparten la mismo Base

• En este curso veremos como funcionan el HARDWARE

1) RECIBIR DATOS
2) ENVIAR DATOS
3) PROCESAN (IN → OUT)
4) ALMACENAN

} INTERNO

• **ARQUITECTURA VON-NEUMANN**:

- CPU
- UNIDAD DE CONTROL
- MEMORIA → Reg
- INPUT
- DATA PATH

- El Hardware de un COMPUTADOR → Solo Ejecuta instrucciones de **Bajo Nivel**

- **PROCESAMIENTO BASICO:**
  - Sumar 2 Num
  - Revisar un numero, comparar
  - Copiar Datos

⊗ ALU : Solo puede operar simple : • Sum • Restar

① **SEÑALES ELECTRICAS** : REPRESENTADOS EN 0 y 1

- COMANDO : RISC-V : 1001010100101010

↓

SUMAR 2 NUM

ADD (A, B) : ← TRADUCIR

ENSAMBLADOR

- **COMPILADOR**: TRADUCTORES DE Lenguaje de Alto nivel (Python, C)

C

⋮ → COMPILADOR → RISC-V ⋮ → ASSEMBLER

↓

0000 1400
1010 0111
1001 1000
⋮

- **TEMARIO**

1. NUMEROS (Enteros, punto flotantes: Representación, almacen y operaciones)
2. LOGICA DIGITAL, CIRCUITOS y ALU
3. CIRCUITOS SECUENCIALES, REGISTROS y MEMORIAS
4. COMPUTADOR BASICO: (componentes, instrucciones, datapath, ctrl flujo, subrutinas
5. CONTROL FLUJO
6. ARQUI. del SET de INSTRUCCIONES (ISA)
7. PIPELING
8. MEMORIA PRINCIPAL, CACHÉS, MEMORIA VIRTUAL
9. INPUT/OUTPUT: DISPOSITIVOS, CPU
10. PARALELISMO

- **Evaluaciones**

- 2 Interrogaciones:

I1:	14 OCT	20 %
I2:	6 DIC	20 %

- Proyecto: 40 % A lo largo del SEMESTRE
- 2 Actividades:

1:	11 SEPT	10 %
2:	23 oct	10 %

- **Exigencias**: $Prom(I_1 + I_2) \geq 3.7$

  Nota Proyecto ≥ 3.7

9 AGO 23

• CLASE 2 :

• **BINARIOS**: 0 y 1 es mucho mas facil de Distinguir en el mundo Electronico

* **Numeros Decimales** : Base 10

421_{10}

↳ $421 = 4 \cdot 10^2 + 2 \cdot 10^1 + 1 \cdot 10^0$

MAS Significativo — menos Significativo

* **Numeros Binarios** : Base 2

110100101_2

↳ $421 \Rightarrow 110100101$

$$1 \times 2^8 + 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

BIT MAS Significativo — BIT MENOS Significativo

• Los BITS son los ATOMOS de la COMPUTACIÓN

→ Mas significativo : n bits → el mult por 2^{n-1}

→ menos significativo : el mult por 2^0

* Cada valor tiene una UNICA REPRESENTACIÓN

- **LOGICA DIGITAL** : Permite manejar un computador
  → El Verdadero HARDWARE

- **NIVELES** :

  - LENGUAJES
  - ASSEMBLY
  - SO
  - ISA
  - MICROARQUITECTURA
  - LOGICA DIGITAL

- **Señal Electrica** : 0 - 0,5 Volts : 0_2
  1 - 1,5 Volts : 1_2

↓

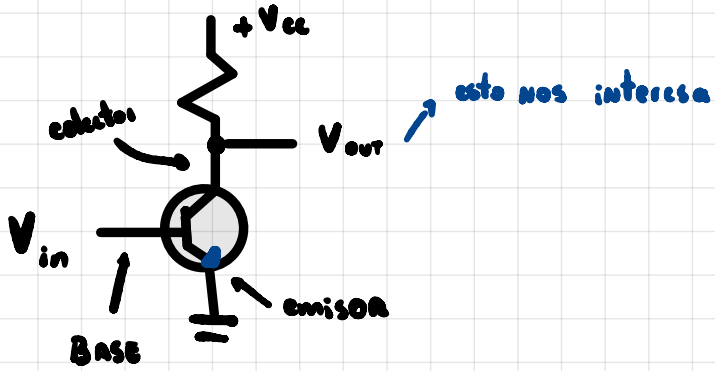
TRANSISTORES : Dispositivo SEMICONDUCTOR → INTERRUPTOR de Señales

- TRANSFERENCIA
- RESISTENCIA

} ELECTRICA

- TRANSISTOR MOSFET : El mas fabricado en el MUNDO

- **Esquema transistor :**



- Colector
- base
- emisor

1) V_{in} : voltaje menor : transistor OFF → Resist INFINITA → $V_{out} = 1$

2) V_{in} : voltaje excede : transistor ON → Alambre : → $V_{out} = 0$

↓

Actúa como **INVERSOR** : Compuerta Logica NOT

1) 1 TRANSISTOR : NOT

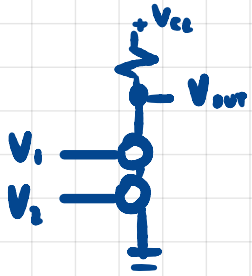
NOT

2) 2 TRANSISTORES : NAND : ambos abiertos

↳ EN SERIE

0	0	→ 1
0	1	→ 1
1	0	→ 1
1	1	→ 0

→

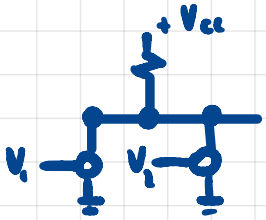


3) 2 TRANSISTORES : NOR :

↳ EN PARALELO

0	0	→ 1
0	1	→ 0
1	0	→ 0
1	1	→ 0

→

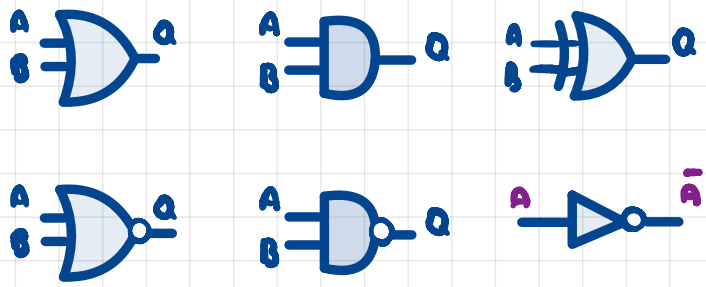


↓

- **COMPUERTAS FUNDAMENTAL** : NOT, NAND, NOR

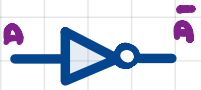
↳ El Computador usa BASICAMENTE, esas 3

- **SIMBOLOS COMPUERTAS** :



- **TABLAS DE VALORES**

① **NOT** :



A	NOT A
0	1
1	0

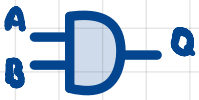
$\boxed{\bar{A}}$

② **OR** :

A B	OR
0 0	0
0 1	1
1 0	1
1 1	1

$\boxed{A + B = A \vee B}$

③ **AND** :



A B	AND
0 0	0
0 1	0
1 0	0
1 1	1

$\boxed{A \cdot B = A \wedge B}$

① **XOR** :

A B	XOR
0 0	0
0 1	1
1 0	1
1 1	0

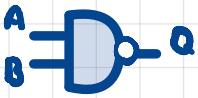
$\boxed{A \oplus B}$

② **NOR** :

A B	NOR
0 0	1
0 1	0
1 0	0
1 1	0

$\boxed{\overline{A + B} = \overline{A \vee B}}$

③ **NAND** :



A B	NAND
0 0	1
0 1	1
1 0	1
1 1	0

$\boxed{\overline{A \cdot B} = \overline{A \wedge B}}$

- ## COMPUERTAS BOOL

- NOT : $\bar{A}$
- OR : $A+B$
- AND : $A \cdot B$
- NAND : $\overline{A \cdot B}$
- NOR : $\overline{A+B}$
- XOR : $A \oplus B = (\bar{A} \cdot B) + (A \cdot \bar{B})$

- ## Algebra Boole

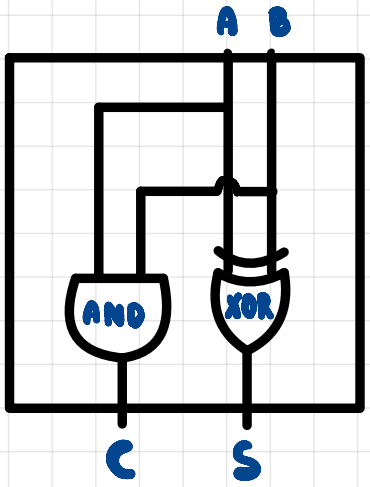
1. $A + 0 = 0$
2. $A + 1 = 1$, $A \cdot 0 = 0$
3. $A + \bar{A} = 1$, $A \cdot \bar{A} = 0$
4. $A + B = B + A$, $A \cdot B = B \cdot A$
5. $A + (B+C) = (A+B) + C$ y $A \cdot (B \cdot C) = (A \cdot B) \cdot C$
6. $A \cdot (B+C) = (A \cdot B) + (A \cdot C)$ y $A + (B \cdot C) = (A+B) \cdot (A+C)$
7. $\overline{A \cdot B} = \bar{A} + \bar{B}$, $\overline{A+B} = \bar{A} \cdot \bar{B}$

- ## SUMA DE BITS :

$$\begin{array}{r} {\scriptstyle 1\ 1\ 0} \\ 1\ 0\ 1\ 1 \\ +\ 0\ 0\ 1\ 0 \\ \hline 1\ 1\ 0\ 1 \end{array}$$

- $0 + 0 = 0$ carry = 0
- $0 + 1 = 1$ carry = 0
- $1 + 1 = 10 \rightarrow S = 0$ Carry = 1

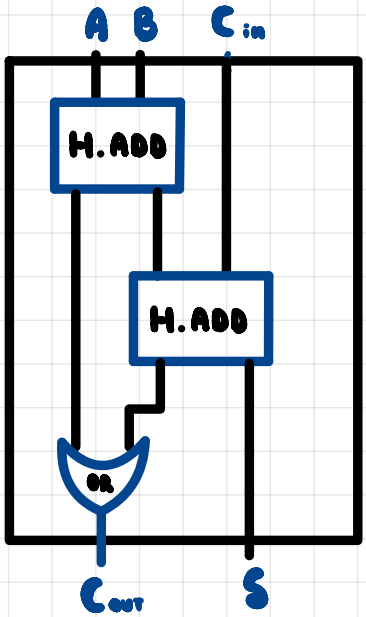
⊕ HALF - ADDER :



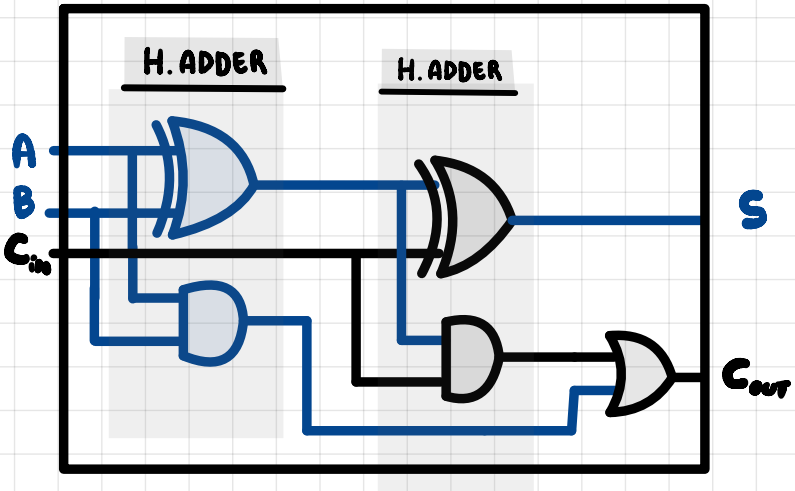
→

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0
		↓ AND	↓ XOR

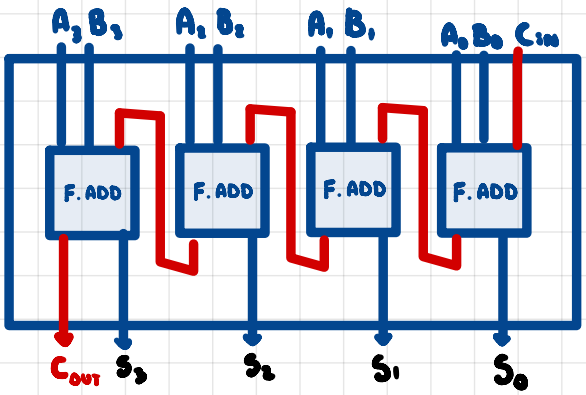
⊕ FULL - ADDER :



A	B	C	C_{OUT}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



⊕ SUMADOR 4 BITS



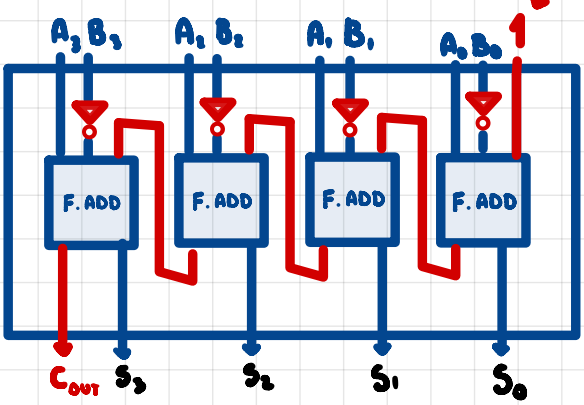

```
   1 1   ← C
   1 0 0 1 ← A
+  0 0 1 1 ← B
 ---------
   1 1 0 0 ← S
```

⊖ RESTADOR 4 BITS

Complemento + 1

2 + -1

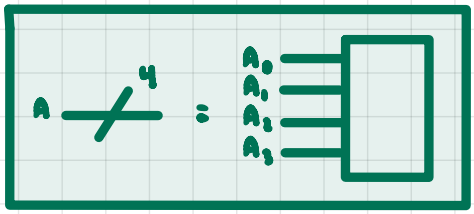
A + -B

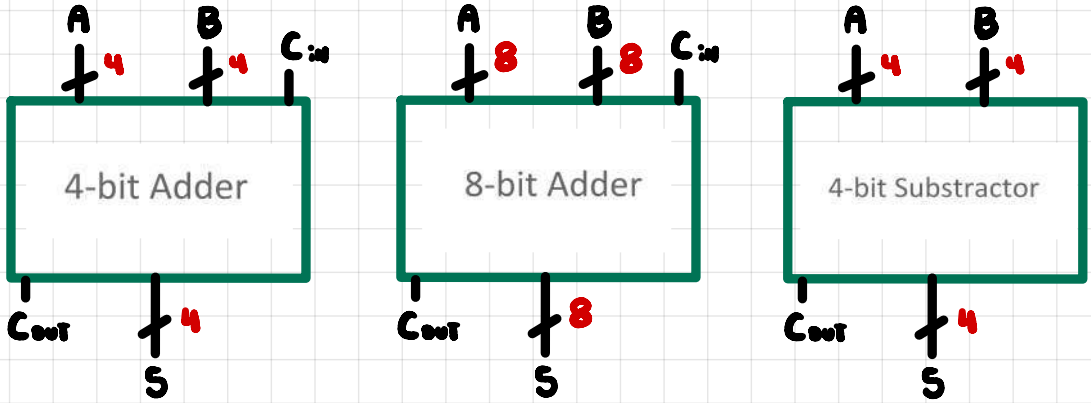



```
     1 1   1 ← C
2 :  0 0 1 0 ← A
-1 : 1 1 1 0 ← B'
     ---------
1    0 0 0 1 ← S
2 - 1 :  1
```

! ABSTRACCIÓN (De componentes)

- **BUS**: Conjunto de Cables (lineas de corriente)





CLASE 3

NUMEROS y ARITMETRICA

- **OVERFLOW**: Cuando se pasa del limite de Numeros

- **NUMEROS PRESICIÓN FINITA**: El Computador tiene una cantidad FIJA de BITS

↓

PROBLEMA DE MEMORIA

- 32 BITS : 2^{32} combinaciones posibles : $[0, 2^{32}-1]$

↓ $4.295.967.295_{10}$

NUM. NEGATIVOS : ① **SIGNO y MAGNITUD** : 421_{10} -421_{10}

0 110100101 1 110100101

② **COMPLEMENTO 2** : 0 120 = +
1 120 = −

32 BIT :

0000 0000 0000 0000 0000 0000 0000 0000_2	= 0_{10}	+
0000 0000 0000 0000 0000 0000 0000 0001_2	= 1_{10}	
0111 1111 1111 1111 1111 1111 1111 1110_2	= $2^{32}-2$	
0111 1111 1111 1111 1111 1111 1111 1111_2	= $2^{32}-1$	
1000 0000 0000 0000 0000 0000 0000 0000_2	= $-2.174.483.648$	−
1000 0000 0000 0000 0000 0000 0000 0001_2	= $-2....$	
1111 1111 1111 1111 1111 1111 1111 1110_2	= -2_{10}	
1111 1111 1111 1111 1111 1111 1111 1111_2	= -1_{10}	

NO TIENE REPRESENT EN +

- **REGLAS** : En Complemento 2

  ① Todos los negativos llevan un 1

  * No se le puede separar de la magnitud

  ↳ $1001_2 = -7_{10}$, pero $0001_2 \neq 7_{10}$

Representado :

$$(X_{31} \cdot -2^{31}) + (X_{30} \cdot 2^{30}) + (X_{29} \cdot 2 \ldots\ldots)$$

- **ARITMETRICA** : 2 POSITIVOS : NORMAL

  2 NEGATIVOS : DESCARTAR EL BIT QUE SOBRA

  8 BIT : 10 : 0000
  -10 :
  0

- **PROBLEMA DE OVERFLOW** :

$$\begin{array}{r} 0000\ 0010_2 \\ +\ 0111\ 1111 \\ \hline 1000\ 0001 \end{array}$$

↳ ⊖ ?? → NO SE DESCARTA

· El Computador nota : ☺ + ☺ = ⊖ ↓ OVERFLOW !

↓

DEJA DE EJECUTARSE ☹ + ☹ = ⊕ ↑

¡ OCURRE MENOS EN COMPUTADORES DE MAS BITS !

- EN COMPLEMENTO 2 : OCURREN OVERFLOW

⊕ PASAR A Comp 2 :

$Y + \bar{Y} = -1$

$\boxed{\bar{Y} + 1 = -Y}$

ej :

$$\begin{array}{r} 0001 \\ +\ 1110 \\ \hline 1111 \end{array} = -1$$

① INVERTIR

② SUMAR 1

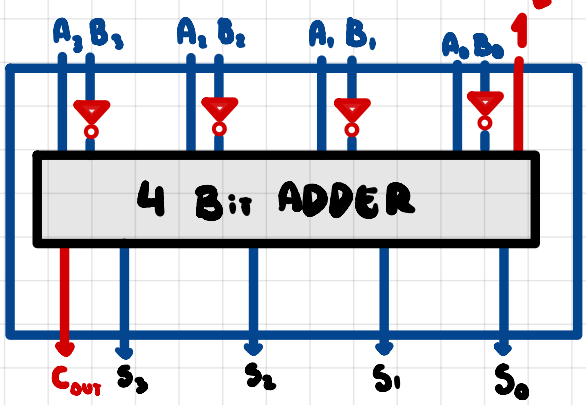
* De vuelta al PPT Logica DIGITAL :

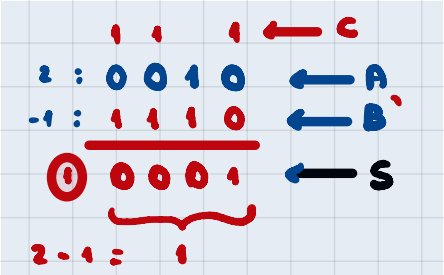
○ RESTADOR 4 BITS

Complemento + 1

2 + -1

A + -B





- **PROGRAMAMOS CIRCUITOS** : Para manejar los DATOS

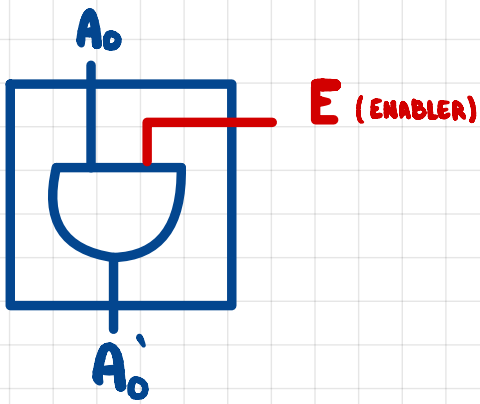
↳ Necesitaremos :
- HABILITAR / DESABILITAR
- SELECCIONAR

CLASE 4

• CIRCUITOS DE CONTROL

Salida

1 ENABLER : Da el control de HABILITAR o DESHABILITAR: S

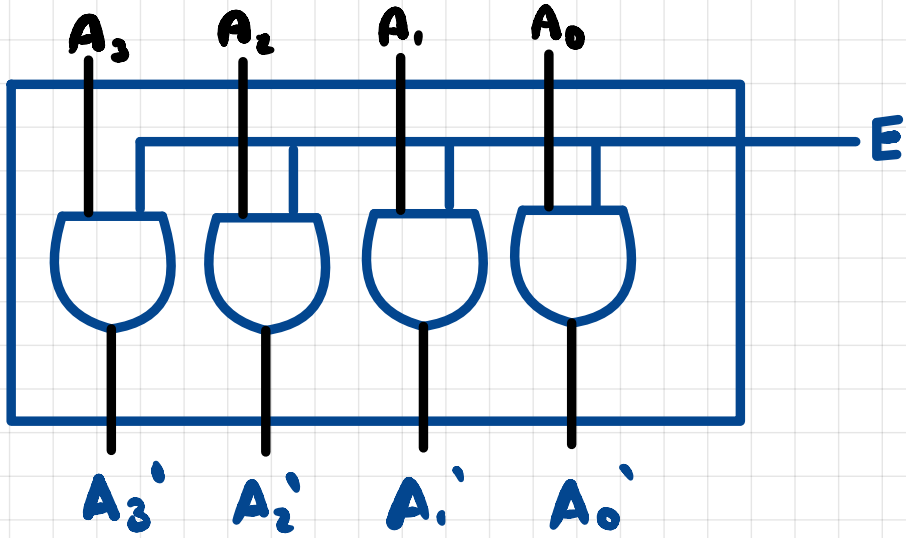


E	A_0'
0	0
1	A_0

1 HABILITA
0 DESHABILITA

• FUNCIONA COMO UN INPUT DE CONTROL : DEJA PASAR

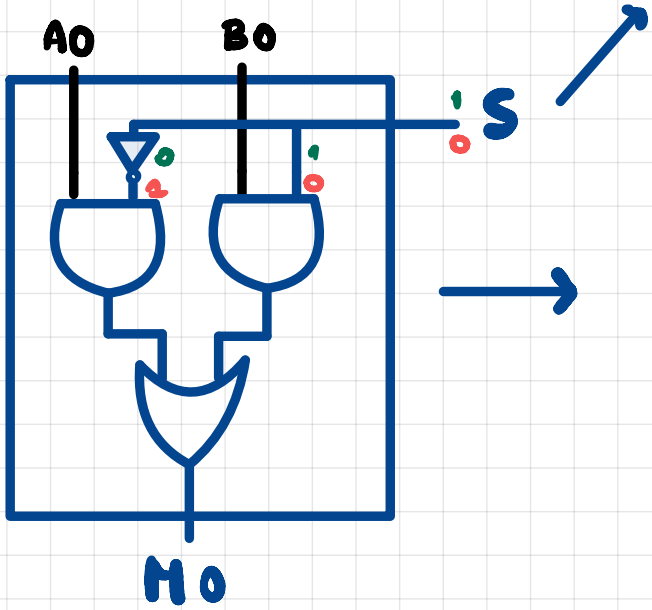
1.1 ENABLER 4 BIT



- **MUITIPLEXOR** (SELECTOR)

↳ Es un circuito con 2^n inputs : S : selecciona cual input

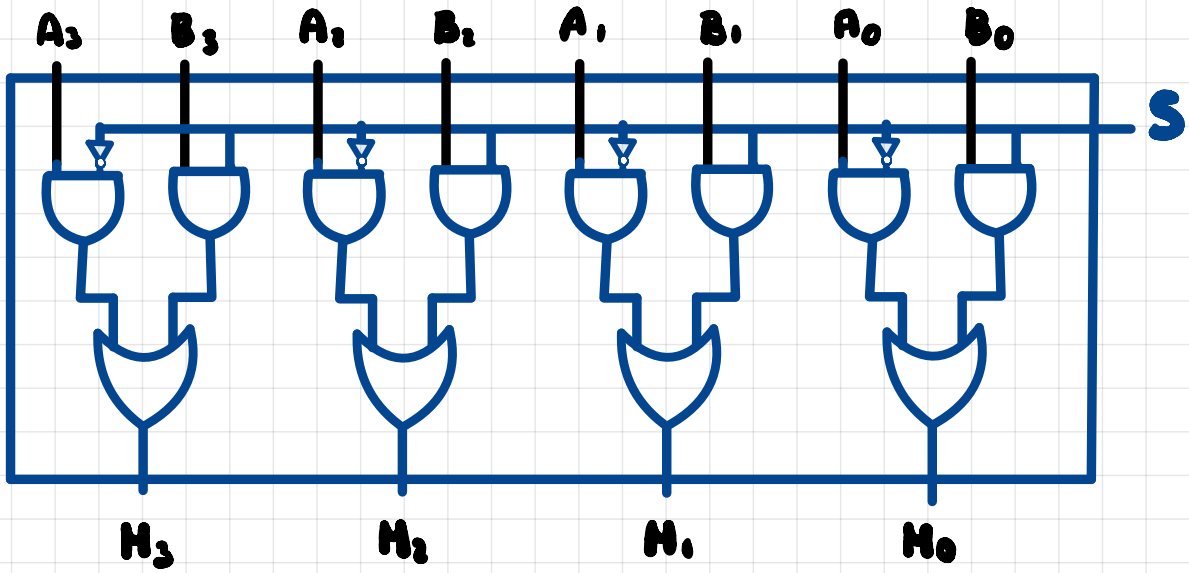
- **1-BIT MUX**



Señales de Control

S	M_0
0	A_0
1	B_0

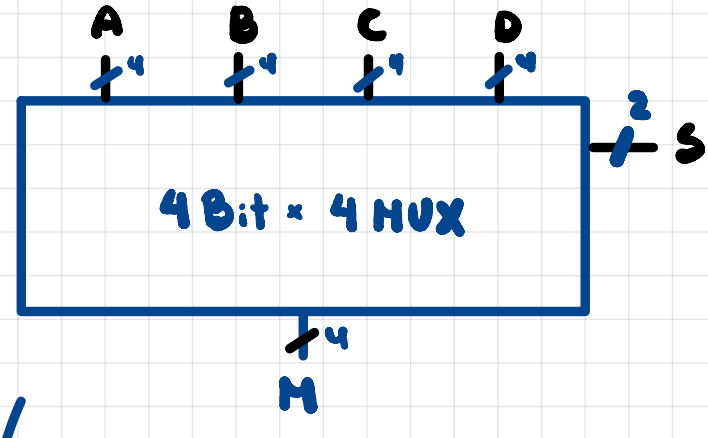
- **4-BIT MUX** (1 S)



$A_3 A_2 A_1 A_0$ } $S = 0 \longrightarrow M_3 M_2 M_1 M_0 = A_3 A_2 A_1 A_0$

$B_3 B_2 B_1 B_0$ } $S = 1 \longrightarrow M_3 M_2 M_1 M_0 = B_3 B_2 B_1 B_0$

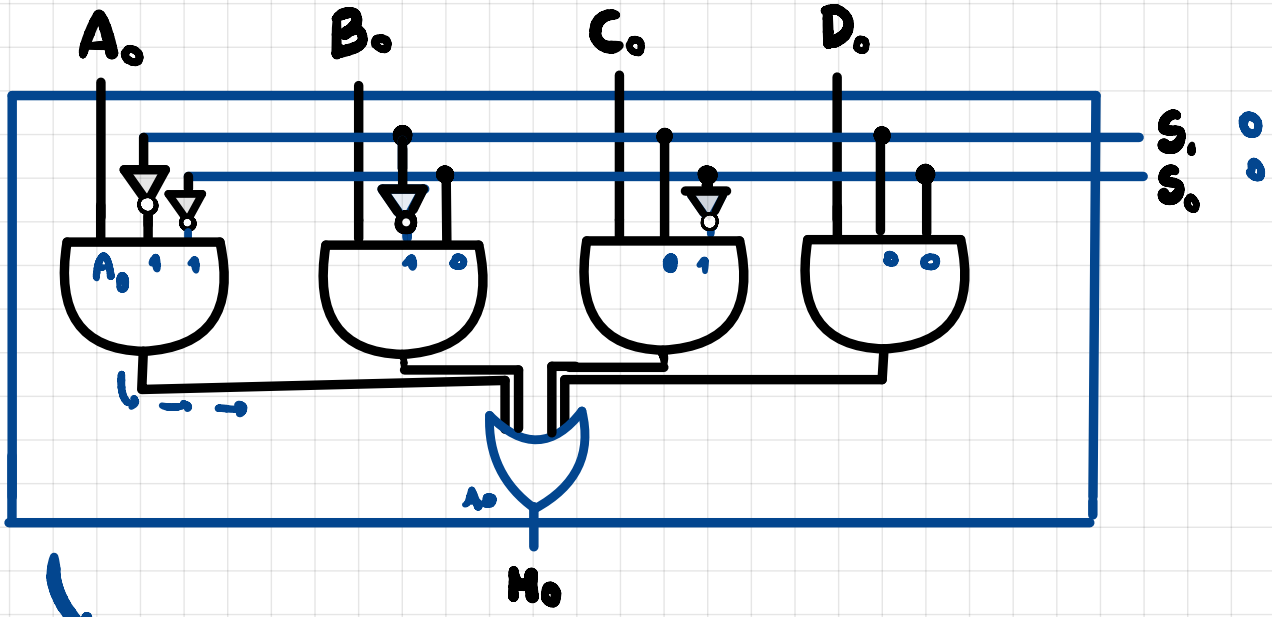
- **4-BIT × 4MUX** (2S) : $S_1 S_0$



S_1	S_0	M
0	0	A
0	1	B
1	0	C
1	1	D

$A_3 A_2 A_1 A_0$

Para el BIT 0 : A_0 , B_0 , C_0 , D_0



Solo 1 Pasa de los 4 :

- **DECODIFICADOR** : Selecciona una **DIRECCION DE MEMORIA**

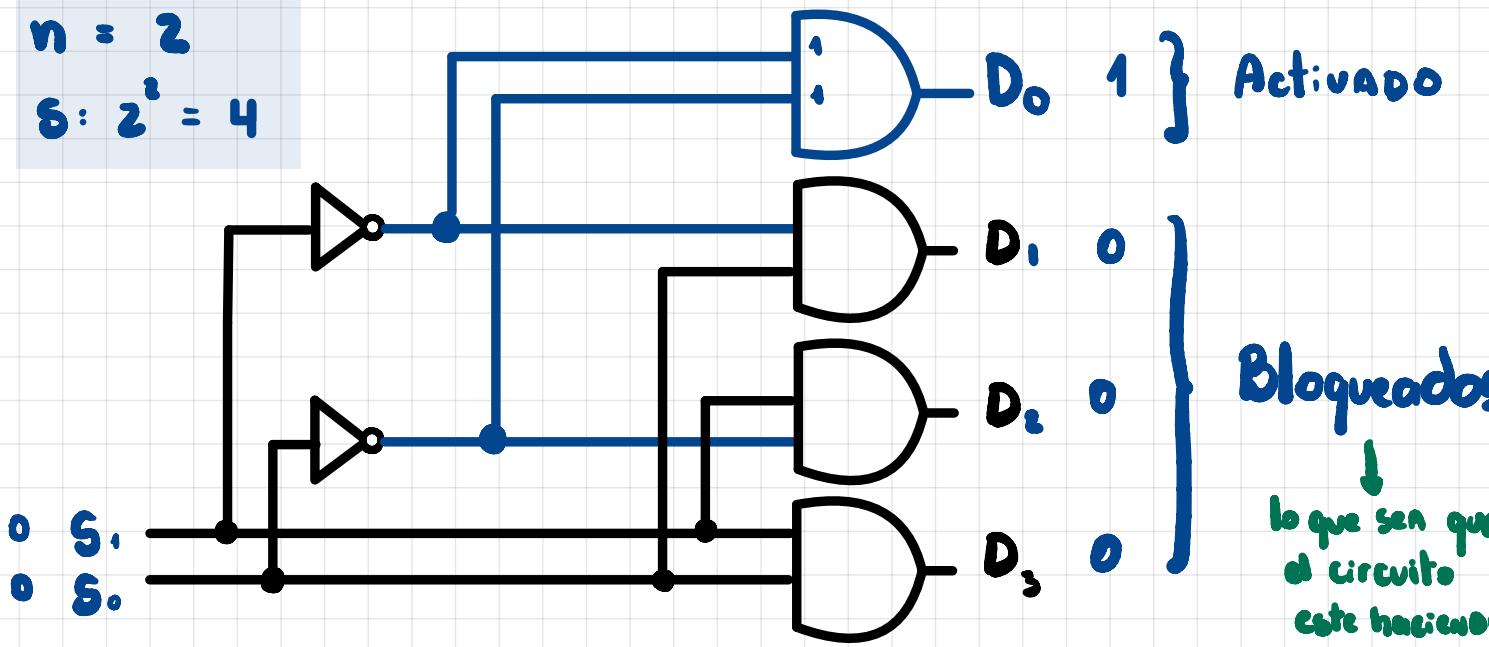
↳ Recibe : n bits ⟶ 1 en 2^n Lineas SALIDA

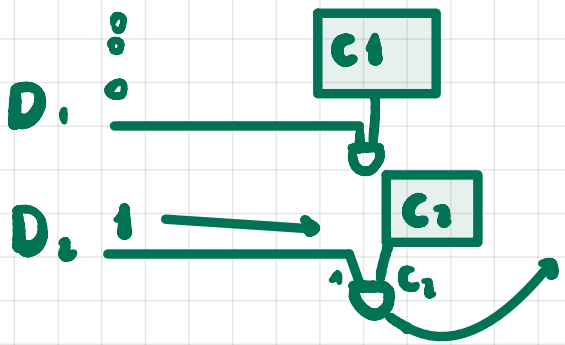
Decide cual de TODAS

4BITS ⟶ 1 entre 16

- $n = 2$

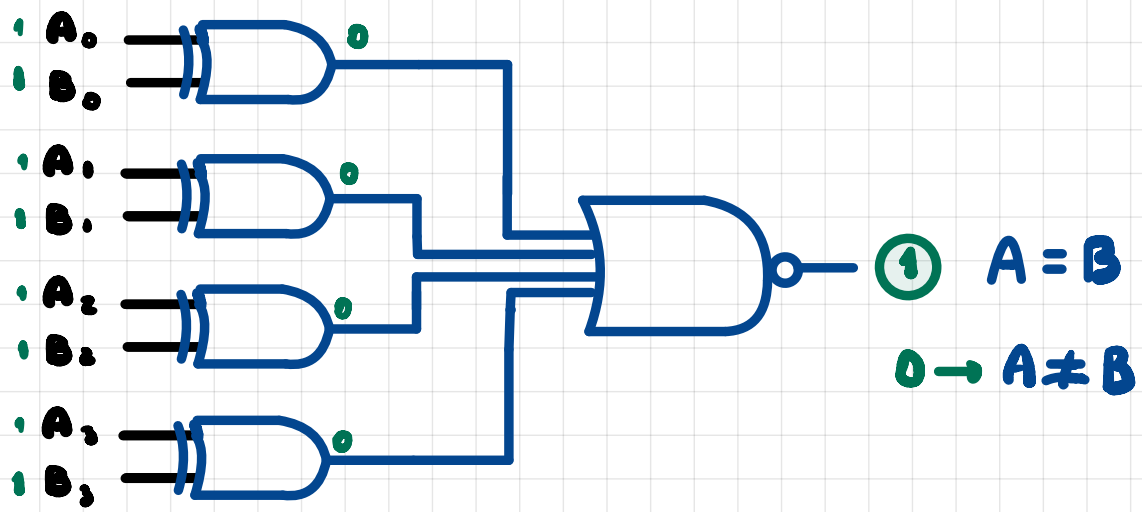
  $S: 2^2 = 4$



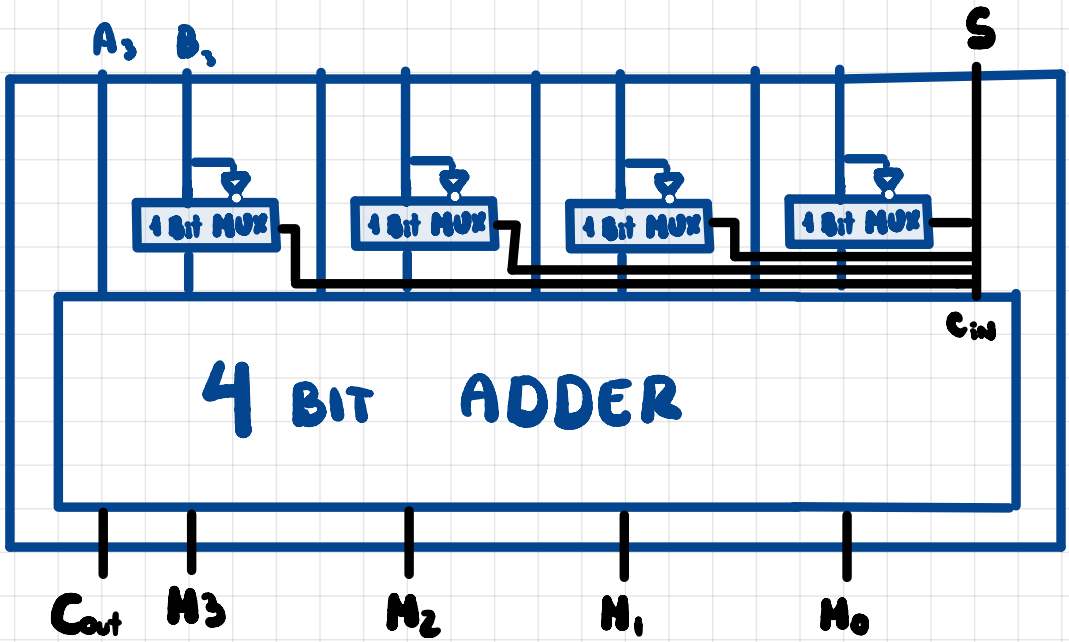


Se activo Circuito 2

- **COMPARADOR** : toma 2 Bits y los compara



- ## SUMADOR ó RESTADOR



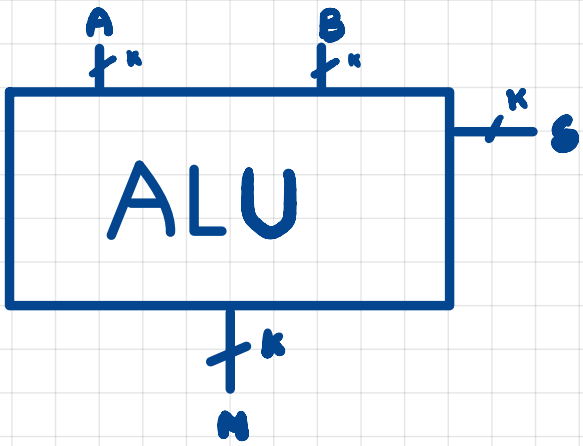
Aqui Check: OVERFLOW : $2\oplus$ y $M_3 = 1 \longrightarrow$ OVERFLOW ($\ominus$)

$2\ominus$ y $M_3 = 0 \longrightarrow$ OVERFLOW ($\oplus$)

→ S = 0 ⟶ B DIRECTO

S = 1 ⟶ B INVERTIDO

- **ALU** : Hace todas las **OPERACIONES**



K : BITS → A , B : INPUT
M : OUTPUT

S : INPUT DE CONTROL

(2^N operaciones DISTINTAS)

SUM/REST anterior : K = 4 BITS

→ Solo 2 operaciones

$= 2^N =$ **N = 1**

- **LA ALU** :
  1. SUMA
  2. RESTA
  3. NOT
  4. AND
  5. OR
  6. NOR

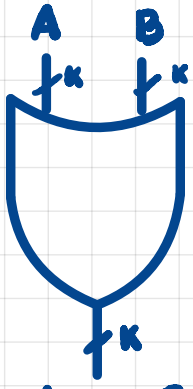
  → Son muy útiles

K = 8 BITS :

A :	0 1 1 0 0 1 0 1
B :	1 1 0 1 0 0 0 1

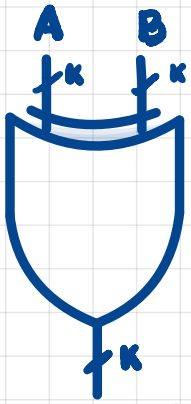
A AND B

01000001



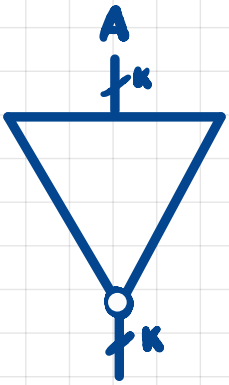
A OR B

11110101



A XOR B

10110100

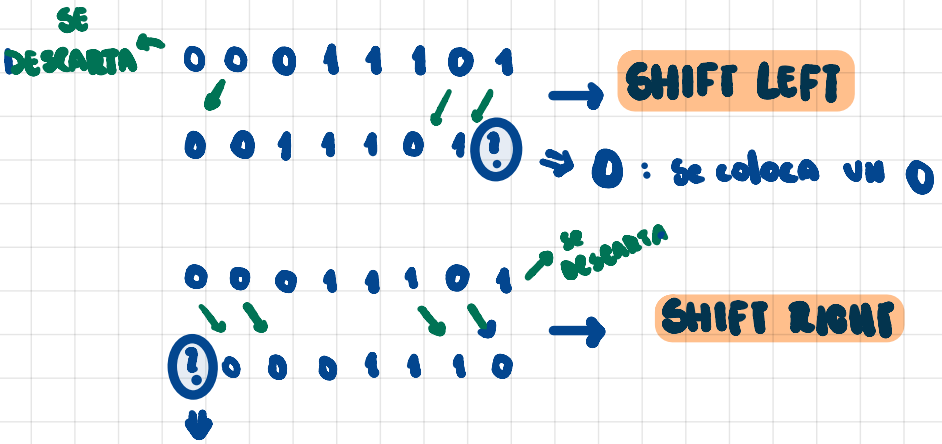


NOT(A)

10011010

* Shift se le hace al input A : COPIAR TODA LA PPT

- **SHIFT LEFT / RIGHT** : Desplazar un bit (operación de la ALU)

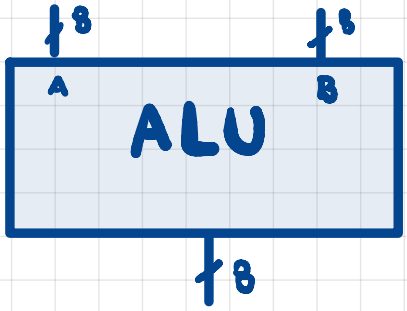


BIT VACÍO : Mismo valor que había originalmente

- En este caso : 0

- **SHIFT LEFT** : MUITIPICACIÓN x 2
- **SHIFT RIGHT** : DIVISIÓN ÷ 2

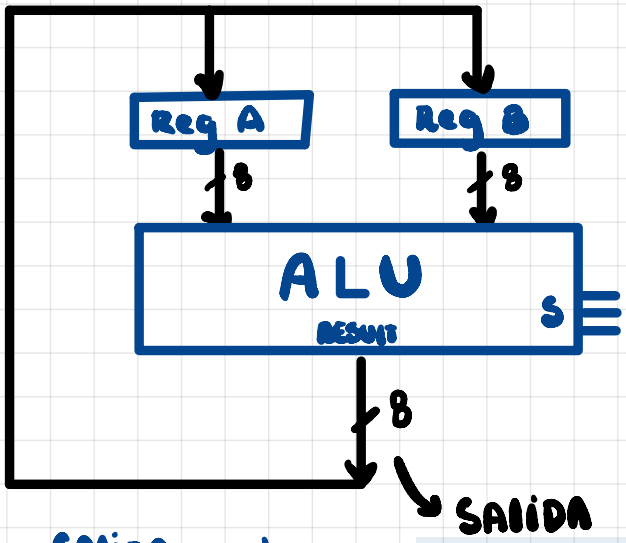
- **8 OPERACIONES BASICAS** : ALU



S2	S1	S0	RESULT
0	0	0	SUMA
0	0	1	RESTA
0	1	0	AND
0	1	1	OR
1	0	0	NOT
1	0	1	XOR
1	1	0	SHIFT L
1	1	1	SHIFT R

CLASE 5

REGISTROS:
- Son ubicaciones que almacenan valores
- Hechos a base de Flip-Flops



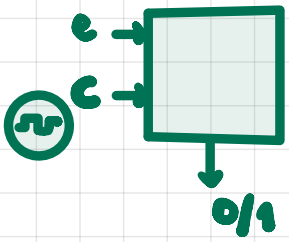
(PUEDE IR CAMBIANDO por ciclo)

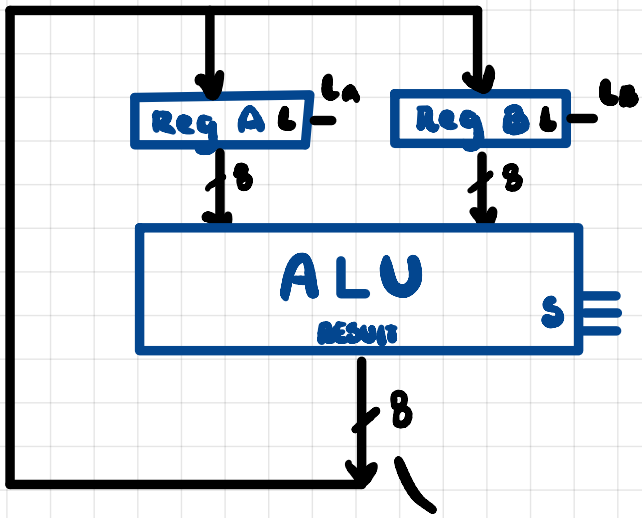
- SALIDA: va hacia los mismos Reg

→ Reg: 8 BITS : 8 Flip-Flops

↓

\# CADA LATCH guarda 1 BIT





- **INSTRUCCIONES 5 BITS**:
  - Para dirigir a los REG:

L_A	L_B	S_2	S_1	S_0	OPERACIÓN
1	0	0	0	0	A = A + B
0	1	0	0	0	B = A + B
1	0	0	0	1	A = A - B
0	1	0	0	1	B = A - B
1	0	0	1	0	A = A AND B
0	1	0	1	0	B = A AND B
1	0	0	1	1	A = A OR B
0	1	0	1	1	B = A OR B
1	0	1	0	0	A = NOT A
0	1	1	0	0	B = NOT B
1	0	1	0	1	A = A XOR B
0	1	1	0	1	B = A XOR B
1	0	1	1	0	A = SHIFT L(A)
0	1	1	1	0	B = SHIFT L(B)
1	0	1	1	1	A = SHIFT R(A)
0	1	1	1	1	B = SHIFT R(B)

- **INSTRUCCIÓN**: Cada combinación de valores de la señal

  0 1 0 0 0 : B = A+B

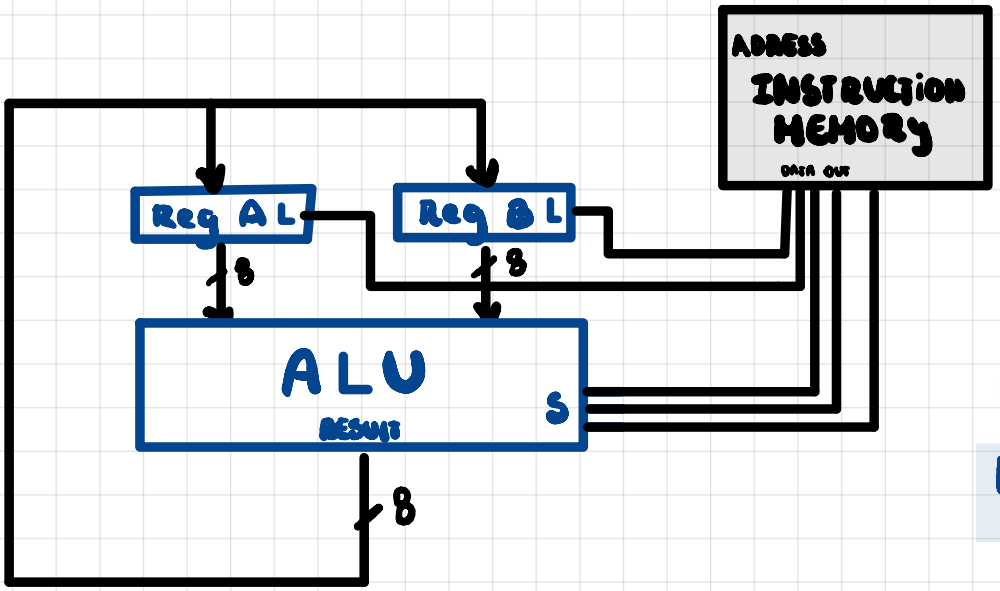
- **PROGRAMA**: Secuencia de instrucciones

  - FIBONACCI:

0 0 - - -		A B
1 0 0 0 0	A = A+B	0 1
0 1 0 0 0	B = A+B	
1 0 0 0 0	A = A+B	
0 1 0 0 0	B = A+B	
	⋮	

¿DE DONDE VIENEN?

- **MEMORIA DE INSTRUCCIONES**

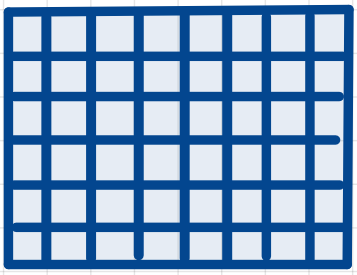


DIRECCION	INSTRUCC
0 0 0 0	10 000
0 0 0 1	01 000
0 0 1 0	10 000
0 0 1 1	01 000
0 1 0 0	10 000

DIRECCIONES: NO ALMACENADAS EN MEMORIA

Contenido en memoria

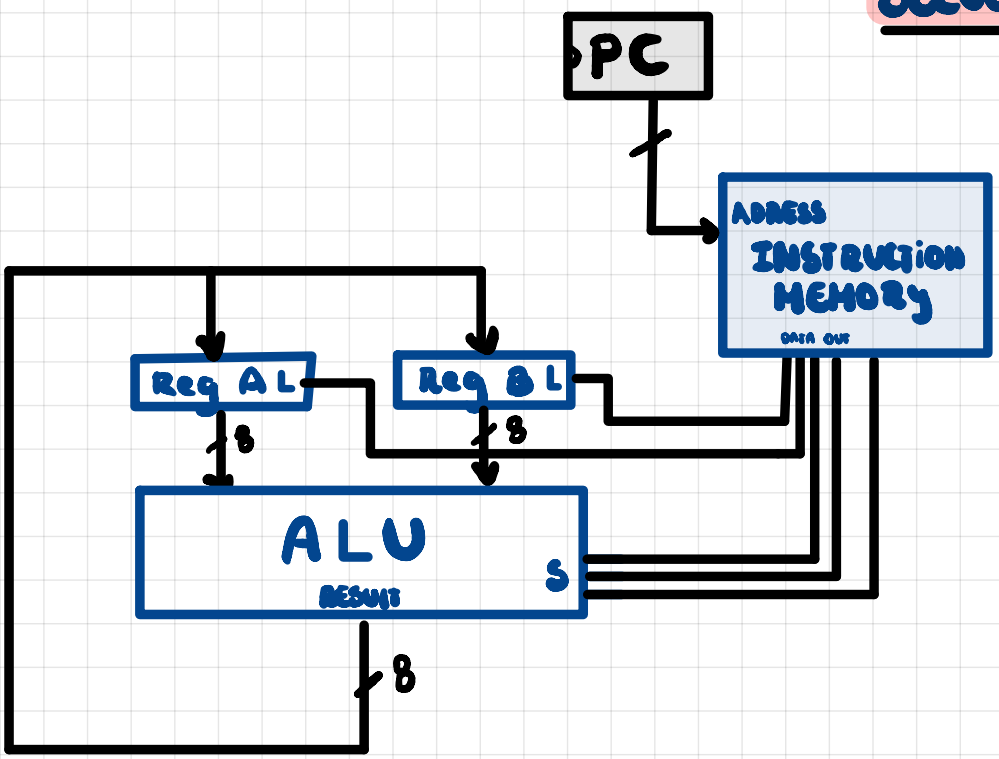
- **MEMORIA** : 1 GB : (2^{30} Byte)
  - Palabra: 4 Bytes (32 bit)

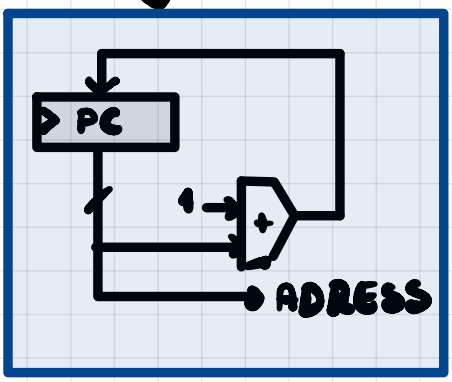


& COPIAR

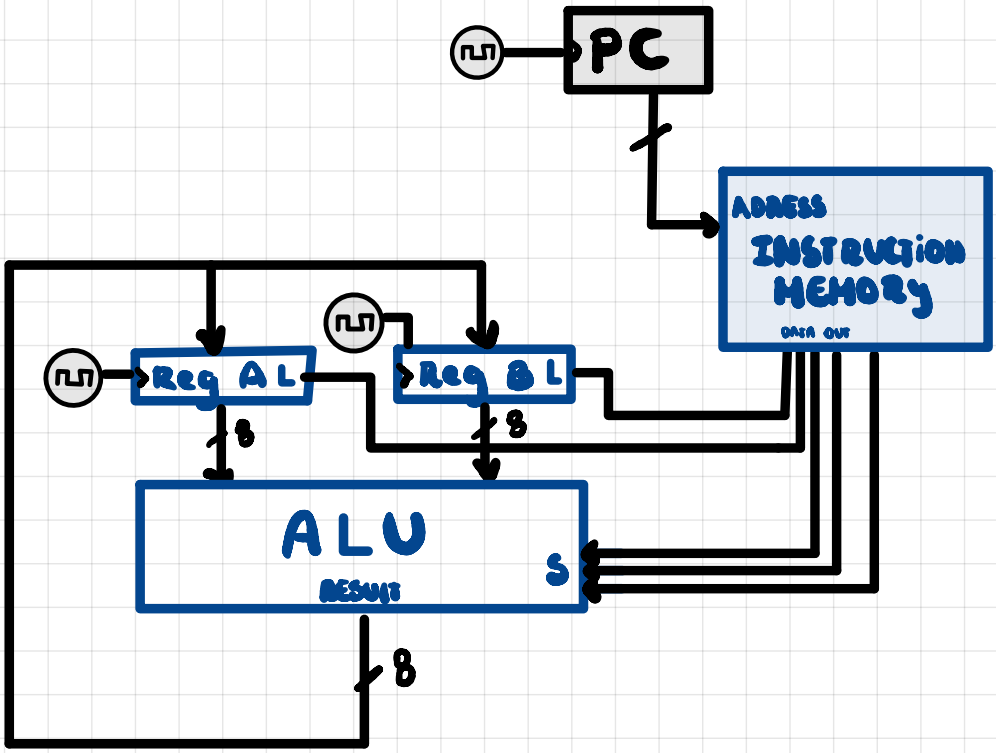
→ NECESITAMOS que la ejecución sea **secuencial**
- Necesitamos controlar cual es la Proxima INSTRUCCIÓN:

- **PROGRAM COUNTER** (PC): **Almacena** (Es un Reg) una **DIRECCIÓN DE MEMORIA**
  - **SECUENCIALMENTE SE AUMENTA 1**





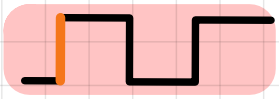
- **PC se actualiza**: despues de cada ciclo :



- **Se ejecutan** : SINCRONICAMENTE : Por eso necesitamos un **RELOJ (CLOCK)**

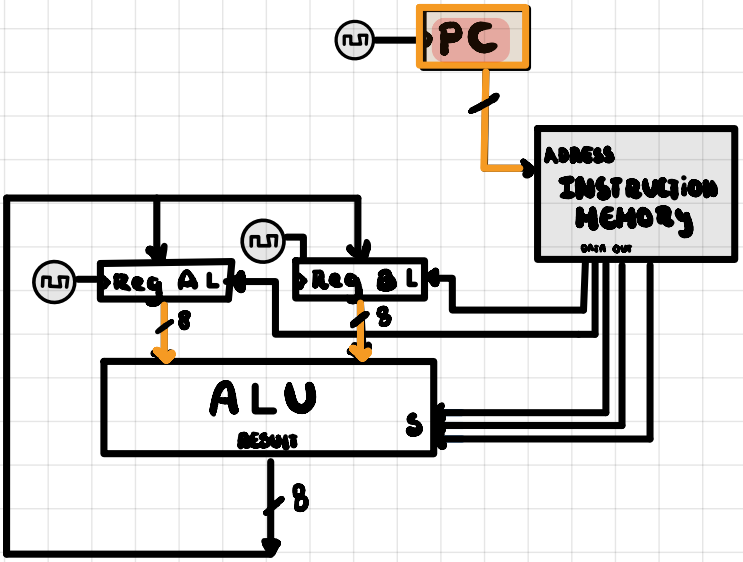
- **TODOS los REGISTROS** :
  - SE CONECTAN AL MISMO RELOJ
  - SE ACTUALIZAN SINCRONICAMENTE

① FLANCO DE SUBIDA :



1) PC ACTUALIZA SU VALOR → VA A ADRESS (I.MEM)

2) REG A/B ACTUALIZAN → VA A ENTRADA (ALU)

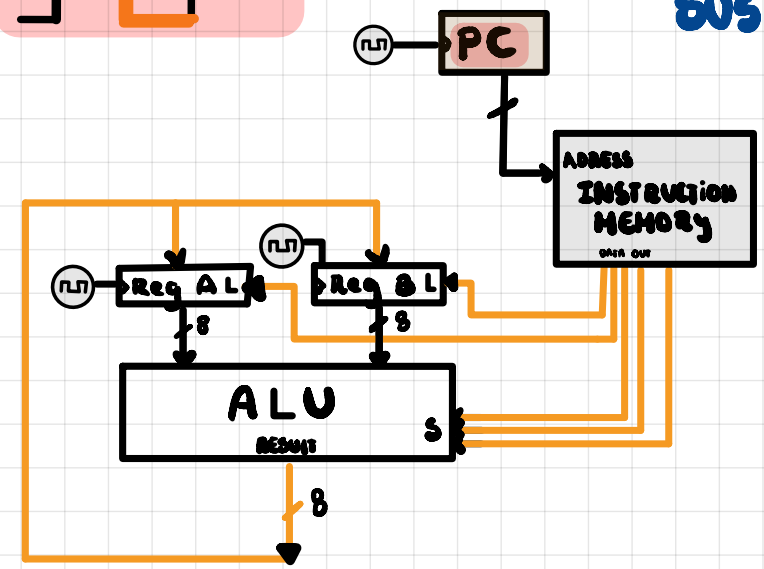


② DURANTE EL CICLO :

1) SE EJECUTA LA INSTRUCCIÓN

! LA ALU ACTUA SEGÚN SUS SEÑALES DE CONTROL (3)

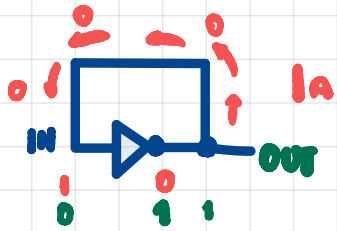
2) RESULTADOS SE PONEN EN LA ENTRADA REG A/B



③ → MISMO CICLO !!!

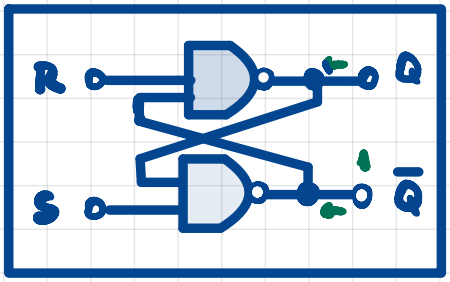
CLASE 6

- **Memoria**: necesitamos CIRCUITO que recuerde información
  - 1) **Retardo**: Primera idea para



la compuerta no es instantanea

- ② **LATCH - SR**: Se basa en el **RETARDO DE PROPAGACIÓN**



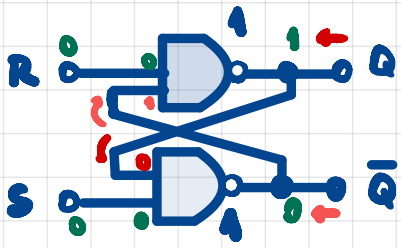
Reset R	Set S	Q^{T+1}
0	0	-
0	1	1
1	0	0
1	1	Q^T

→ sig tiempo

→ lo que HABÍA

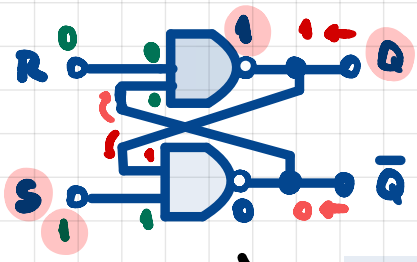
R y S no pueden ser 0: No es Aceptable

Caso 1: S:0 R:0

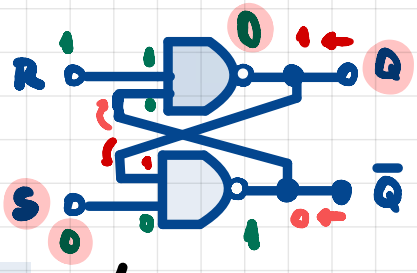


S: R = 0, S = 0 → Q = $\bar{Q}$ = 1

Caso 2: S:1 R:0



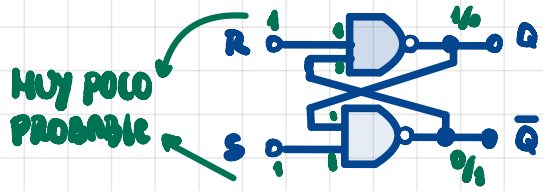
Caso 3: S:0 R:1

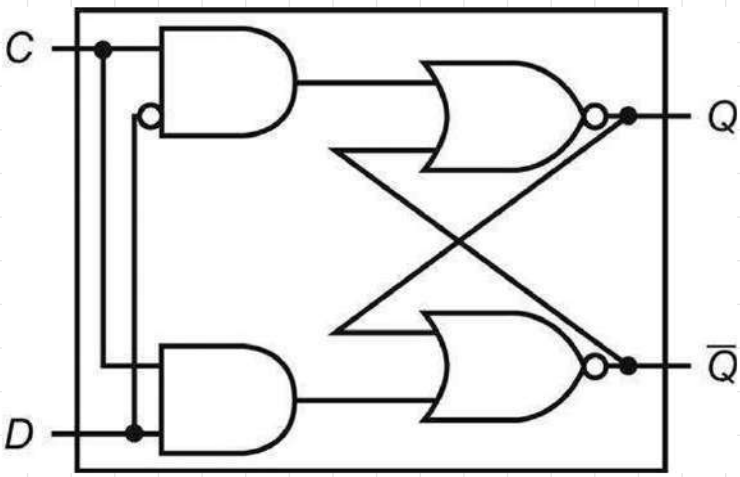


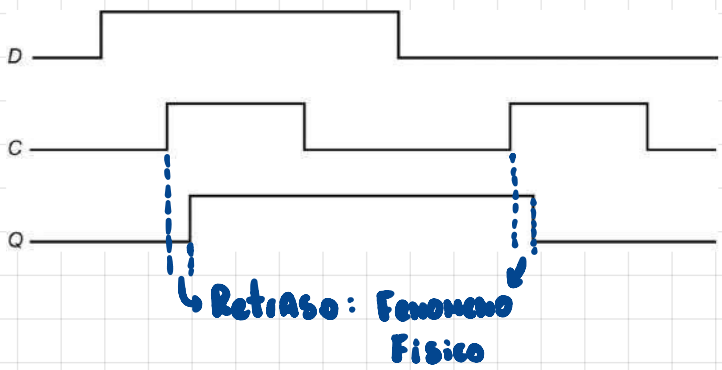
S = Q
si S ≠ R

Caso 4: S:1 R:1 → **NO DETERMINISTA**

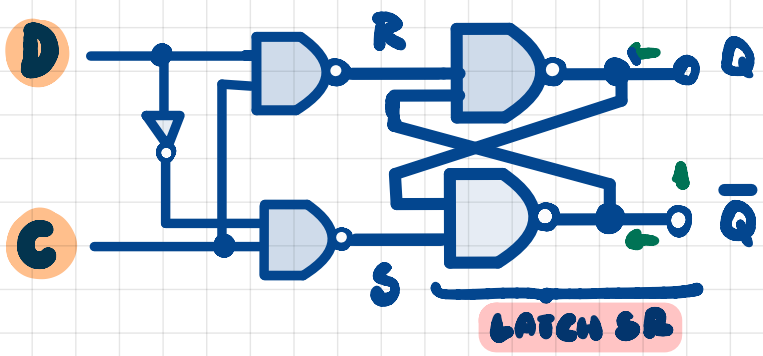
R y S = 1 → siempre retorna lo que HAY







- **LATCH - D** : (CONTROLADO POR el **CLOCK**) : EVITA $\begin{matrix} 0:R \\ 0:S \end{matrix}$
- Para que Latch S-R cambie de ESTADO (solo en instantes especificos) R y S vengan de NAND → Conectado al CLOCK

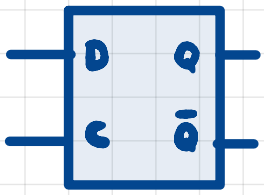


C	D	Q^{t+1}
0	0/1	Q^t → Lo que Había
1	0	0
1	1	1

- CLOCK = 1 → LATCH "se carga" con D

C:0 { D:0 → Q^t ; D:0 → Q^t

- **LATCH D** :



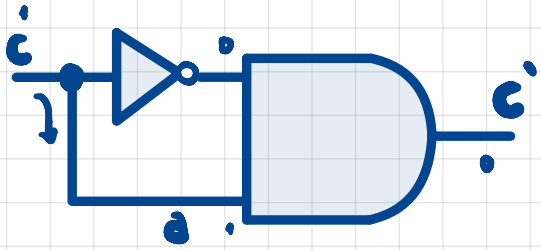
D : DATO INPUT

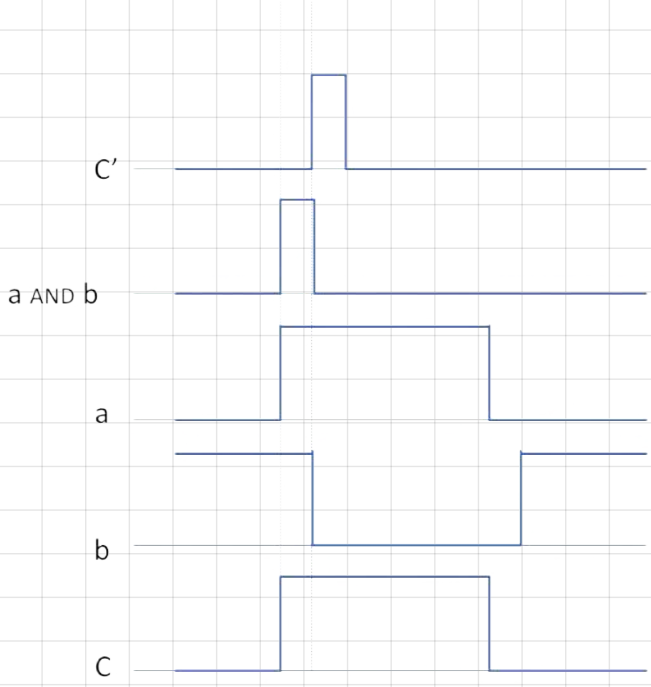
C : CLOCK (CONTROL)

Q : ESTADO OUTPUT

⊛ Necesitamos circuitos que puedan LEER el valor de una SEÑAL en un instante particular en el tiempo y ALMACENARLO

- **ESTRECHAR NIVEL ALTO DE C** : USAREMOS **GENERADORES DE PULSOS**

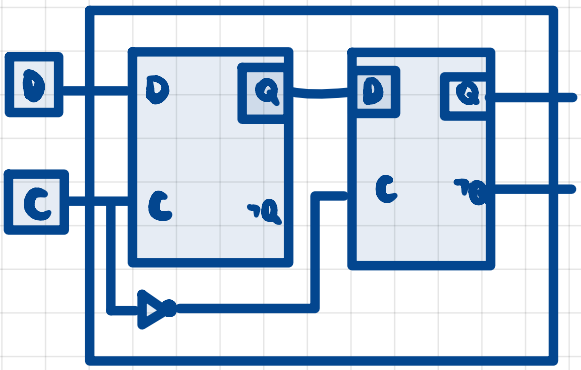




- Con esto llegamos al **Flip-Flop** !!

- Almacenamos D en el Latch DURANTE LA TRANSICIÓN DE C : 0 → 1 = FLANCO DE SUBIDA

- **Flip - Flop D** : 2 LATCHES D + 1 INVERSOR

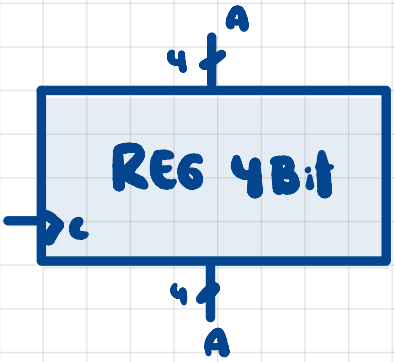


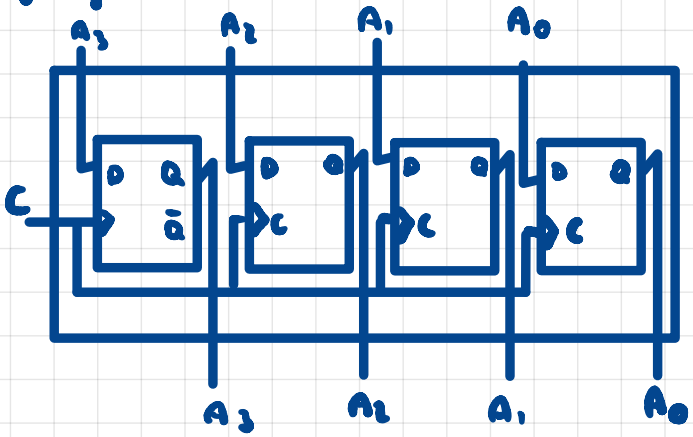
- C cambia 1 → 0 (FLANCO BAJADA)

(otros cire en subida) Q ALMACENA D

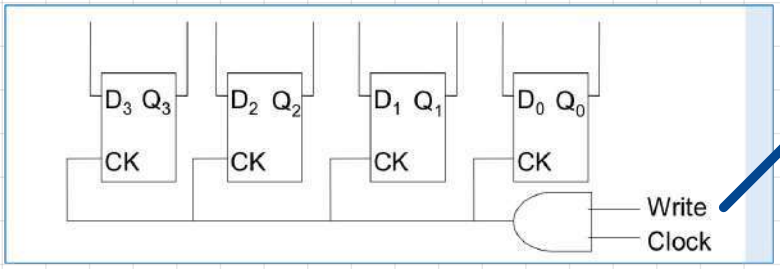
C	D	Q^{T+1}
0/1/↑	0/1	
↓	0	0
↓	1	1

REGISTRO : ARREGLO DE Flip flops



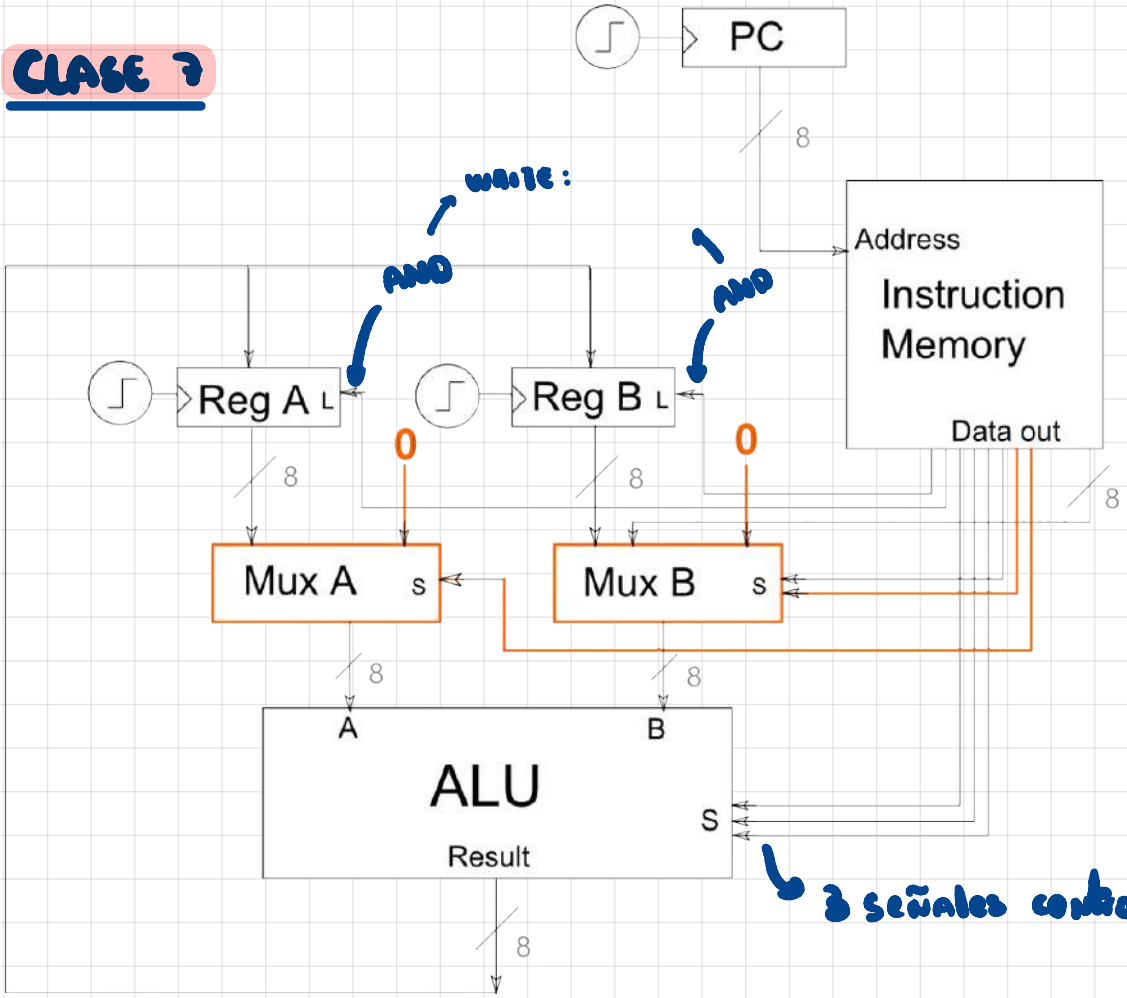


WRITE : Controla selectivamente el CLOCK
↳ enable



Para Escribir en Reg : WRITE = 1

- **CLASE 7**



3 señales control: 8 Operaciones

- Por ahora, cada instrucción: 16 Bits:
  1. 8 Bit Señales de Control
  2. 8 Bit Para el Literal
- Mux A y Mux B: Selecciona si a la ALU le entregamos un input del Reg o del Literal → CONSTANTE
- PC: Funciona solo (Se suma 1 automaticamente)

• INSTRUCCIONES DE CONTROL

Lit=4

La	Lb	Sa0	Sb0	Sb1	Sop2	Sop1	Sop0	Operación
1	0	1	0	0	0	0	0	A=B
0	1	0	1	1	0	0	0	B=A
1	0	0	0	1	0	0	0	A=Lit
0	1	0	0	1	0	0	0	B=Lit
1	0	0	0	0	0	0	0	A=A+B
0	1	0	0	0	0	0	0	B=A+B
1	0	0	0	1	0	0	0	A=A+Lit
1	0	0	0	0	0	0	1	A=A-B
0	1	0	0	0	0	0	1	B=A-B
1	0	0	0	1	0	0	1	A=A-Lit
1	0	0	0	0	0	1	0	A=A and B
0	1	0	0	0	0	1	0	B=A and B
1	0	0	0	1	0	1	0	A=A and Lit
1	0	0	0	0	0	1	1	A=A or B
0	1	0	0	0	0	1	1	B=A or B
1	0	0	0	1	0	1	1	A=A or Lit
1	0	0	0	0	1	0	0	A=notA
0	1	0	0	0	1	0	0	B=notA
1	0	0	0	1	1	0	0	A=notLit
1	0	0	0	0	1	0	1	A=A xor B
0	1	0	0	0	1	0	1	B=A xor B
1	0	0	0	1	1	0	1	A=A xor Lit
1	0	0	0	0	1	1	0	A=shift left A
0	1	0	0	0	1	1	0	B=shift left A
1	0	0	0	1	1	1	0	A=shift left Lit
1	0	0	0	0	1	1	1	A=shift right A
0	1	0	0	0	1	1	1	B=shift right A
1	0	0	0	1	1	1	1	A=shift right Lit

28 INSTRUCCIONES

A CADA INSTRUCCIÓN VAMOS A DARLE UN largo de 6 BITS

HAY 8 SEÑALES DE

• INSTRUCCIONES DE 16 BIT

implica que las INSTRUCCIONES DE MEMORIA DEBE Estar formada por Palabras de 16 BIT

→ Cada palabra de memoria se compone de 16 Flip Flops

- **OPCODE** : Enumeración correlativa de cada instrucción !

	Opcode	La	Lb	Sa0	Sb0	Sb1	Sop2	Sop1	Sop0	Operación
0 ←	000000	1	0	1	0	0	0	0	0	A=B
	000001	0	1	0	1	1	0	0	0	B=A
	000010	1	0	0	0	1	0	0	0	A=Lit
	000011	0	1	0	0	1	0	0	0	B=Lit
	000100	1	0	0	0	0	0	0	0	A=A+B
	000101	0	1	0	0	0	0	0	0	B=A+B
	000110	1	0	0	0	1	0	0	0	A=A+Lit
	000111	1	0	0	0	0	0	0	1	A=A-B
	001000	0	1	0	0	0	0	0	1	B=A-B
	001001	1	0	0	0	1	0	0	1	A=A-Lit
	001010	1	0	0	0	0	0	1	0	A=A and B
	001011	0	1	0	0	0	0	1	0	B=A and B
	001100	1	0	0	0	1	0	1	0	A=A and Lit
	001101	1	0	0	0	0	0	1	1	A=A or B
	001110	0	1	0	0	0	0	1	1	B=A or B
	001111	1	0	0	0	1	0	1	1	A=A or Lit
	010000	1	0	0	0	0	1	0	0	A=notA
	010001	0	1	0	0	0	1	0	0	B=notA
	010010	1	0	0	0	1	1	0	0	A=notLit
	010011	1	0	0	0	0	1	0	1	A=A xor B
	010100	0	1	0	0	0	1	0	1	B=A xor B
	010101	1	0	0	0	1	1	0	1	A=A xor Lit
	010110	1	0	0	0	0	1	1	0	A=shift left A
	010111	0	1	0	0	0	1	1	0	B=shift left A
	011000	1	0	0	0	1	1	1	0	A=shift left Lit
	011001	1	0	0	0	0	1	1	1	A=shift right A
	011010	0	1	0	0	0	1	1	1	B=shift right A
28 ←	011011	1	0	0	0	1	1	1	1	A=shift right Lit

(6) A = A + 6

↓

Ahora Escribimos : 000110 00000110

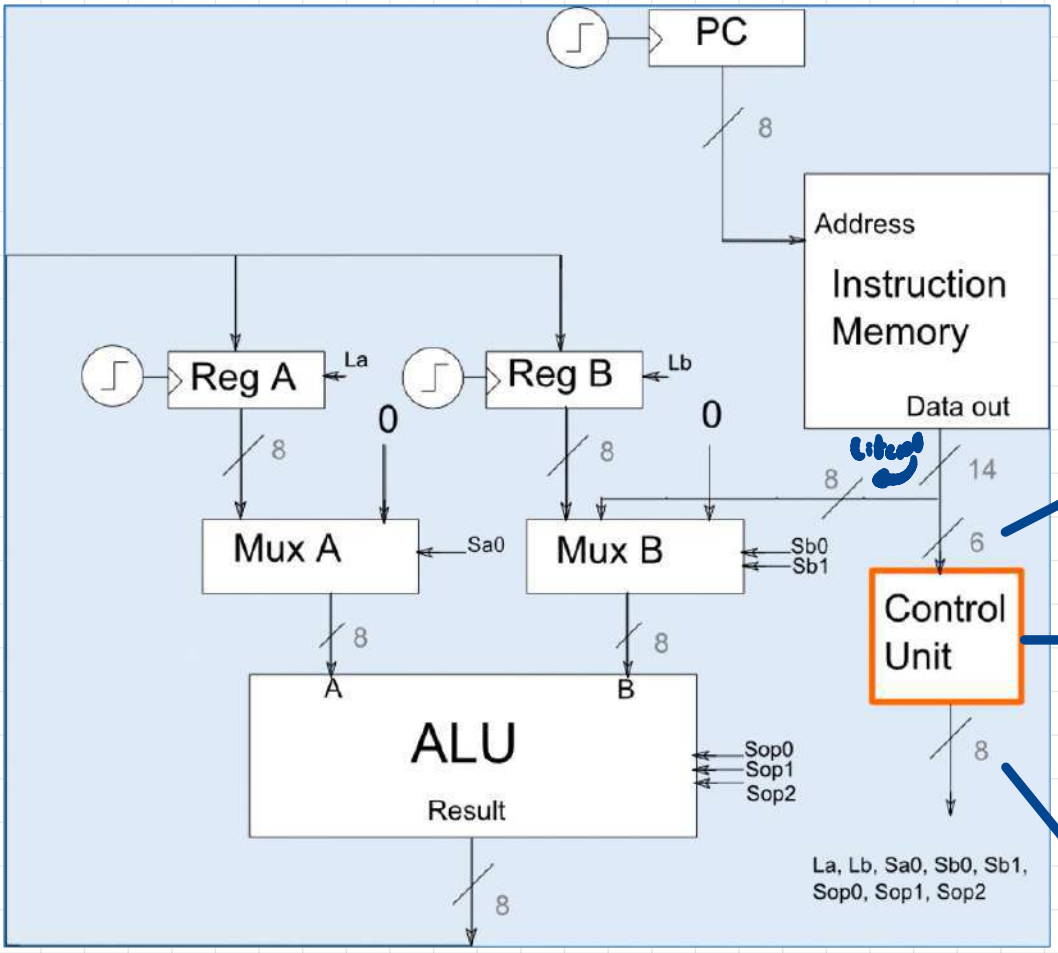
↓ OPCODE : 6 BIT

↓ Literal : 8 BIT

* Debe haber alguien que traduzca los OPCODES

↓

UNIDAD DE CONTROL



6 inputs: 6 BIT OPCODE

Traduce los OPCODE a los SEÑALES DE CONTROL

8 out: 8 señales de control

puede crecer!

mas instrucciones

CONTROL UNIT: CIRCUITO DIGITAL

· El lenguaje de maquina es dificil de entender:

↓

Necesitamos un Nivel INTERMEDIO

↳ **ASSEMBLY NOS AYUDA!**: Versión Simplificada

↳ El ENSAMBLADOR NOS TRADUCE a lenguaje Maquina!

Instrucción	Operandos	Opcode	La	Lb	Sa0	Sb0	Sb1	Sop2	Sop1	Sop0	Operación
MOV	A,B	000000	1	0	1	0	0	0	0	0	A=B
	B,A	000001	0	1	0	1	1	0	0	0	B=A
	A,Lit	000010	1	0	0	0	1	0	0	0	A=Lit
	B,Lit	000011	0	1	0	0	1	0	0	0	B=Lit
ADD	A,B	000100	1	0	0	0	0	0	0	0	A=A+B
	B,A	000101	0	1	0	0	0	0	0	0	B=A+B
	A,Lit	000110	1	0	0	0	1	0	0	0	A=A+Lit
SUB	A,B	000111	1	0	0	0	0	0	0	1	A=A-B
	B,A	001000	0	1	0	0	0	0	0	1	B=A-B
	A,Lit	001001	1	0	0	0	1	0	0	1	A=A-Lit
AND	A,B	001010	1	0	0	0	0	0	1	0	A=A and B
	B,A	001011	0	1	0	0	0	0	1	0	B=A and B
	A,Lit	001100	1	0	0	0	1	0	1	0	A=A and Lit
OR	A,B	001101	1	0	0	0	0	0	1	1	A=A or B
	B,A	001110	0	1	0	0	0	0	1	1	B=A or B
	A,Lit	001111	1	0	0	0	1	0	1	1	A=A or Lit
NOT	A,A	010000	1	0	0	0	0	1	0	0	A=notA
	B,A	010001	0	1	0	0	0	1	0	0	B=notA
	A,Lit	010010	1	0	0	0	1	1	0	0	A=notLit
XOR	A,A	010011	1	0	0	0	0	1	0	1	A=A xor B
	B,A	010100	0	1	0	0	0	1	0	1	B=A xor B
	A,Lit	010101	1	0	0	0	1	1	0	1	A=A xor Lit
SHL	A,A	010110	1	0	0	0	0	1	1	0	A=shift left A
	B,A	010111	0	1	0	0	0	1	1	0	B=shift left A
	A,Lit	011000	1	0	0	0	1	1	1	0	A=shift left Lit
SHR	A,A	011001	1	0	0	0	0	1	1	1	A=shift right A
	B,A	011010	0	1	0	0	0	1	1	1	B=shift right A
	A,Lit	011011	1	0	0	0	1	1	1	1	A=shift right Lit

LENGUAJE ASSEMBLY

- El computador hasta ahora trabaja con numeros $\mathbb{Z}$ (enteros)

0 → 255 -1 → 256

- Necesitamos trabajar con numeros $\mathbb{R}$ (Reales)

NUM REALES :

3,14159....

¿Como escribo 3.155.760.000 > 2^{32}

num mas grande posible

- **REPRESENTACIÓN APROXIMADA :**

① NOTACIÓN CIENTIFICA

Notación científica : un único dígito a la izquierda del punto decimal

$0.000000001 = 0.0001 \times 10^{-5} = 0.01 \times 10^{-7} = 1.0 \times 10^{-9}$

$3{,}155{,}760{,}000 = 3.15576 \times 10^{9} = 0.315576 \times 10^{10} = 0.0315576 \times 10^{11}$

: MANTIZA

: EXPONENTE

un mismo número puede representarse de *varias maneras distintas* en notación científica

★ Podemos usar INFINITAS REPRESENTACIONES !

Eligiremos la unica que tiene un entero DELANTE del pto.

Not. Cientif. NORMALIZADA : 1 sola Representación

$0.000000001 = \mathbf{1.0 \times 10^{-9}}$

$3{,}155{,}760{,}000 = \mathbf{3.15576 \times 10^{9}}$

Los números reales también se pueden representar en base 2, generalizando la representación posicional de números enteros:

los exponentes de la base 2 al lado derecho del punto decimal son negativos

$$3 \times 10^0 \quad 6 \times 10^{-1} \quad 6 \times 10^{-2} \quad 4 \times 10^{-3}$$

$$3\ .\ 6\ 6\ 4\ \ldots_{10} = 1\ 1\ .\ 1\ 0\ 1\ 0\ 1\ 0\ 1_2$$

$$1 \times 2^1 \quad 1 \times 2^0 \quad 1 \times 2^{-1} \quad 0 \times 2^{-2} \quad 1 \times 2^{-3} \quad 0 \times 2^{-4} \quad 1 \times 2^{-5} \quad 0 \times 2^{-6} \quad 1 \times 2^{-7}$$

Y también podemos escribir números (reales) binarios en notación científica normalizada

... usando **2** como base del exponente:

$$11.1010101 = 11.1010101 \times 2^0 = \mathbf{1.11010101} \times \mathbf{2}^1$$

exponente

Siempre hay un 1 en Binario

no es necesario escribirlo

el HARDWARE lo incluye

• ARITMETRICA COMPUTACIONAL

⊙ PUNTO FIOTANTE : Punto Binario

$$1.\underbrace{XXXXXXXX}_{\text{se omite}} \cdot 2^{\underbrace{YYY}_{\text{se omite}}}$$

En C : Este tipo de dato se llama **FIOAT**

- OVERFIOW : Numero muy grande
- UNDERFIOW : Numero muy pequeño

↓

Solución : APROXIMAR A 0

• AHORA USAREMOS **SIGNO MAGNITUD** :

$$\pm \underbrace{1.}_{\text{Fijo}}XXXXXXXX \ldots \times \underbrace{2}_{\text{Fijo}}^{YYY}$$

¿Como expresamos los exponentes negativos

32 BITS :

1	10000001	01000000000000000000000
1 BIT :	8 BIT :	23 BIT :
SIGNO	Exponente	FRACCION

$$\boxed{(-1)^{\text{signo}} \times \text{fracción} \times 2^{\text{exponente}}}$$

5 PROPIEDADES

1) GRAN RANGO DE NUMEROS REPRESENTABLES

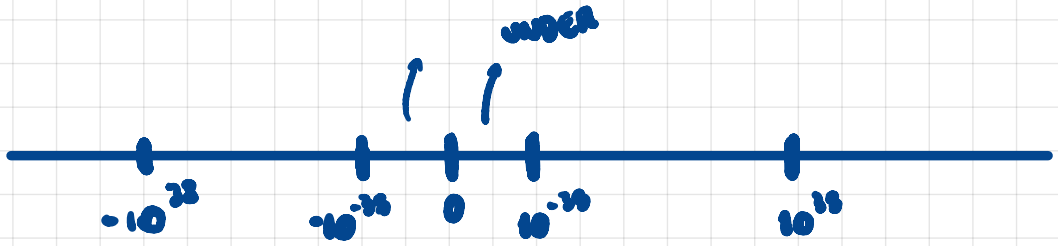
aprox: $\pm 1{,}7 \times 10^{-38}$ hasta $\pm 1{,}7 \times 10^{38}$

2) Buena Presición: $2^{-22} \approx$ 1 en 4 Millones

3) OVERFLOW → UNDERFLOW

OVERFLOW ↓ exponente demasiado grande para 8 BIT

UNDERFLOW ↓ fracción muy pequeña ↓ 0 : BUENA APROXIMACIÓN



CLASE 8 : REP PUNTO FIOT EN IEEE 754 (32 BITS)

$$(-1)^{\text{signo}} \times (1 + \text{fracción}) \times 2^{\text{exponente} - 127}$$

11000000101000000000000000000000

| 1 | 10000000 | 01000000000000000000000 |

SIGNO $\begin{pmatrix} +0 \\ -1 \end{pmatrix}$ — EXPONENTE $(e + 127)$ — MANTISA

- $-7{,}125_{10} = -111.001_2 = -1{,}11001_2 \times 2^{2}$: $\boxed{1 \quad 10000001 \quad 11001\,000000\ldots 0}$

  $2 + 127$

  $129 = 10000001$

- El 0 : $+0$: 0 00000000 00000000000000000000000

  -0 : 1 00000000 00000000000000000000000

- El $\infty +$: 0 11111111 00000000000000000000000

  El $\infty -$: 1 11111111 00000000000000000000000

- NaN (Not a number) : X 11111111 XXXXXXXXXX..... X

  $\downarrow$

  $\neq 0$

• SUMA DE PUNTOFIOTANTE (*: Igualar exponentes)

* exp pequeño → agrandarlo!

$$9{,}999 \times 10^{1} + 1{,}610 \cdot 10^{-1}$$

$$\downarrow$$

$$9{,}999 \times 10^{1} + 0{,}01610 \cdot 10^{1}$$

¿Como ajustar exponente? → SHR del significante

$$9{,}999 \times 10^{1} + 0{,}01610 \cdot 10^{1} = 10{,}015 \times 10^{1}$$

$$= 1{,}0015 \times 10^{2}$$

e $1{,}0010_b \times 2^{5} + 1{,}01_b \times 2^{-1}$

1 Igualar exp: $1{,}001 \times 2^{5} + 0{,}00000101 \times 2^{5}$ → SHR 6 veces!

$1{,}01 \times 2^{-1}$) SHR
$0{,}101 \times 2$
$0{,}0101 \times 2^{1}$) SHR
$0{,}00101 \times 2^{1}$) SHR

$$1{,}00100101 \times 2^{5}$$

↓
Ajustar para 1 Digito solo a la coma

• REDONDEO:

$$1{,}0015 \times 10^{2} =$$

$$1{,}002 \times 10^{2}$$

* Al NORMALIZAR: + o − el exp: Revisar si da OVER o UNDERFLOW
* Al REDONDEAR: Revisar si Resultado sigue NORMALIZADO

• Algoritmo suma pto flotante

1. comparar los exponentes; “shift” el número más pequeño a la derecha hasta que su exponente sea igual al exponente más grande
2. sumar los significantes **(MANTISAS)**
3. normalizar la suma, ya sea “shifting” a la derecha e incrementado el exponente, o “shifting” a la izquierda y decrementando el exponente

   ¿“overflow” o “underflow”? → *excepción*
4. redondear el significante al número apropiado de bits

   ¿está normalizado? → terminar

   volver al paso 3

→ Todo esto lo realiza **FPU** no la ALU !!

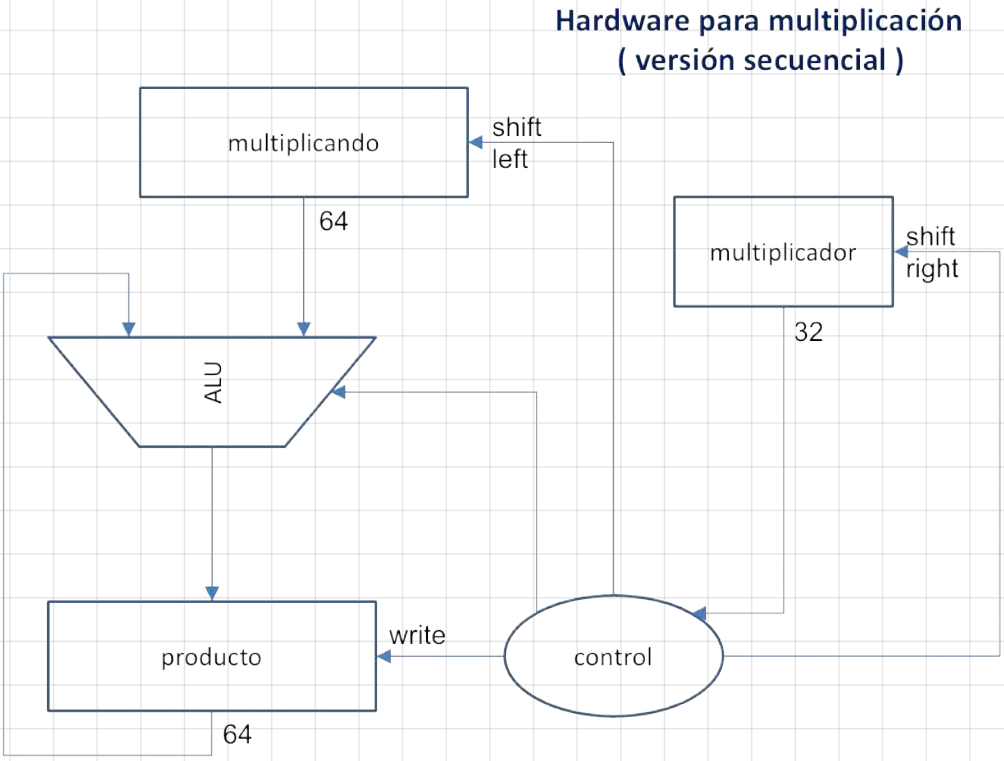
• MULTIPLICACIÓN

```
  1 2 3   x   4 5
  -------
  6 1 5   →   123 x 5
4 9 2 0   →   123 x 40
-------
5 5 3 5
```

• BASE 2:

```
        1 0 1 0   x   1 0 0 1
        -------
        1 0 1 0       1010 x 1
      0 0 0 0         1010 x 0
    0 0 0 0           1010 x 0
  + 1 0 1 0           1010 x 1
  -----------------
  = 1 0 1 1 0 1 0
```

• HARDWARE MULTIPLICACIÓN



• Por cada bit del multiplicador es una iteración

• ALGORITMO :

Algoritmo de multiplicación de números enteros de 32 bits sin signo, usando el hardware anterior

Partir

1. Examinar el dígito de más a la derecha del multiplicador

1a. Si es 1, sumar el multiplicando al producto y poner el resultado en el registro del producto

2. Desplazar (*shift*) el registro del multiplicando a la izquierda un bit

3. Desplazar (*shift*) el registro del multiplicador a la derecha un bit

4. Verificar si los pasos 1, 2 y 3 se han repetido 32 veces

4a. Si aún faltan repeticiones, ir a 1

Terminar

• ALGORITMO PTO. FLOTANTE

Algoritmo de multiplicación de punto flotante

1. sumar los exponentes desfasados y restar el desfase a la suma para obtener el nuevo exponente desfasado
2. multiplicar los significantes
3. normalizar el producto si es necesario, "shifting" a la derecha e incrementado el exponente

   ¿"overflow" o "underflow"? → excepción
4. redondear el significante al número apropiado de bits

   ¿está normalizado? → ir al paso 5

   volver al paso 3
5. poner el signo del producto en positivo si los signos de los operandos son iguales; en negativo, en caso contrario

   terminar

e $1{,}000 \times 2^{-1} \times 1{,}110 \times 2^{-2}$

1 SUMAMOS EXPONENTES : $-1 + -2 = -3$

• CON DESFASE $(-1+127) + (-2+127) - 127 = -3 + 127 = 124$

2 MULTIPLICAMOS SIGNIFICANTES : PROD EN 4 BITS

$1{,}000 \times 1.110 = 1.110000$

↓

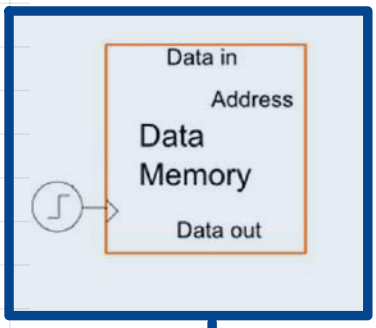
$1{,}110000 \times 2^{-3}$

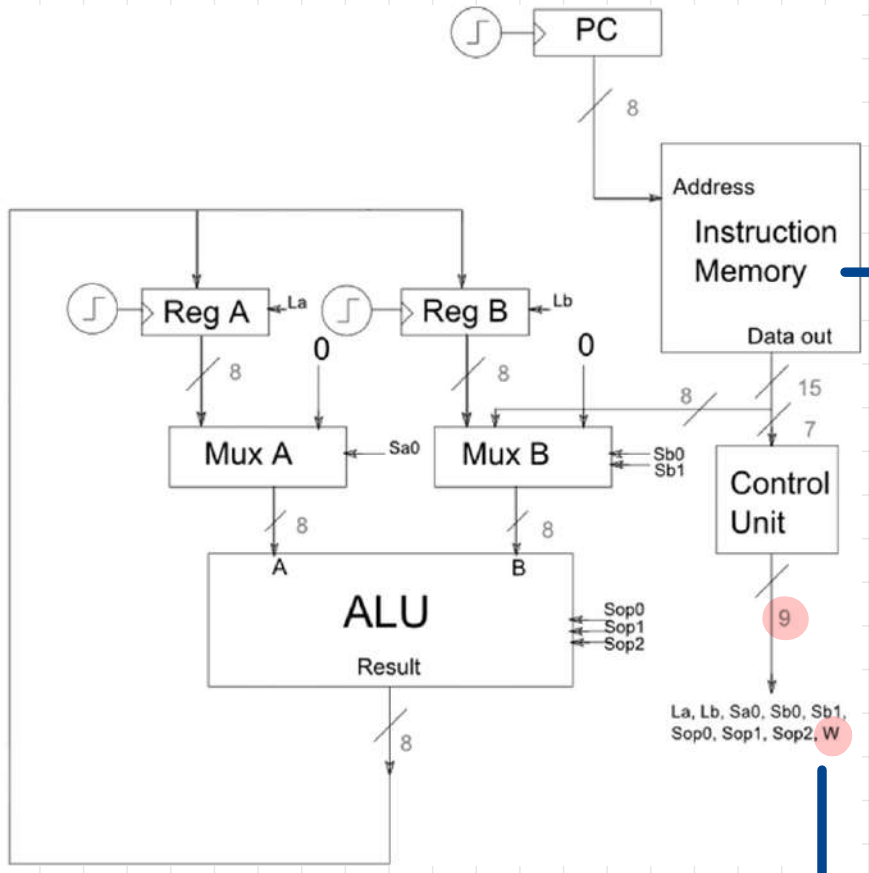
→ $1{,}110 \times 2^{-3}$

... COPIAR .... SOBRE REDONDEOS

• CLASE 9

→ QUEREMOS UNA MEMORIA PARA MANEJAR DATOS
(y no solo instrucciones)





→ LEE y SACA DATOS

↓

→ 8 + 1
9 BIT

Similar a memoria: instrucción
- ADRES
- DATA OUT
- DATA IN → guarda valores
  - Podemos ESCRIBIR
    LEER

DATOS: ✓
INSTRUCCIONES: ✗

Favorece que solo trabajar con REG

NUEVA SEÑAL CONTROL

↓

W → ESCRIBIR: 1

W → LEER: 0

- **NUESTROS NUEVOS CODIGOS EN ASSEMBLY**

Dirección	Label	Instrucción/Dato
	DATA:	
0x00	var0	Dato 0
0x01	var1	Dato 1
0x02	var2	Dato 2
0x03		Dato 3
0x04		Dato 4
	CODE:	
0x00		Instrucción 0
0x01		Instrucción 1
0x02		Instrucción 2
0x03		Instrucción 3
0x04		Instrucción 4

Pueden ir a parar a

Ahora tenemos:

NOMBRE DE VARIABLE

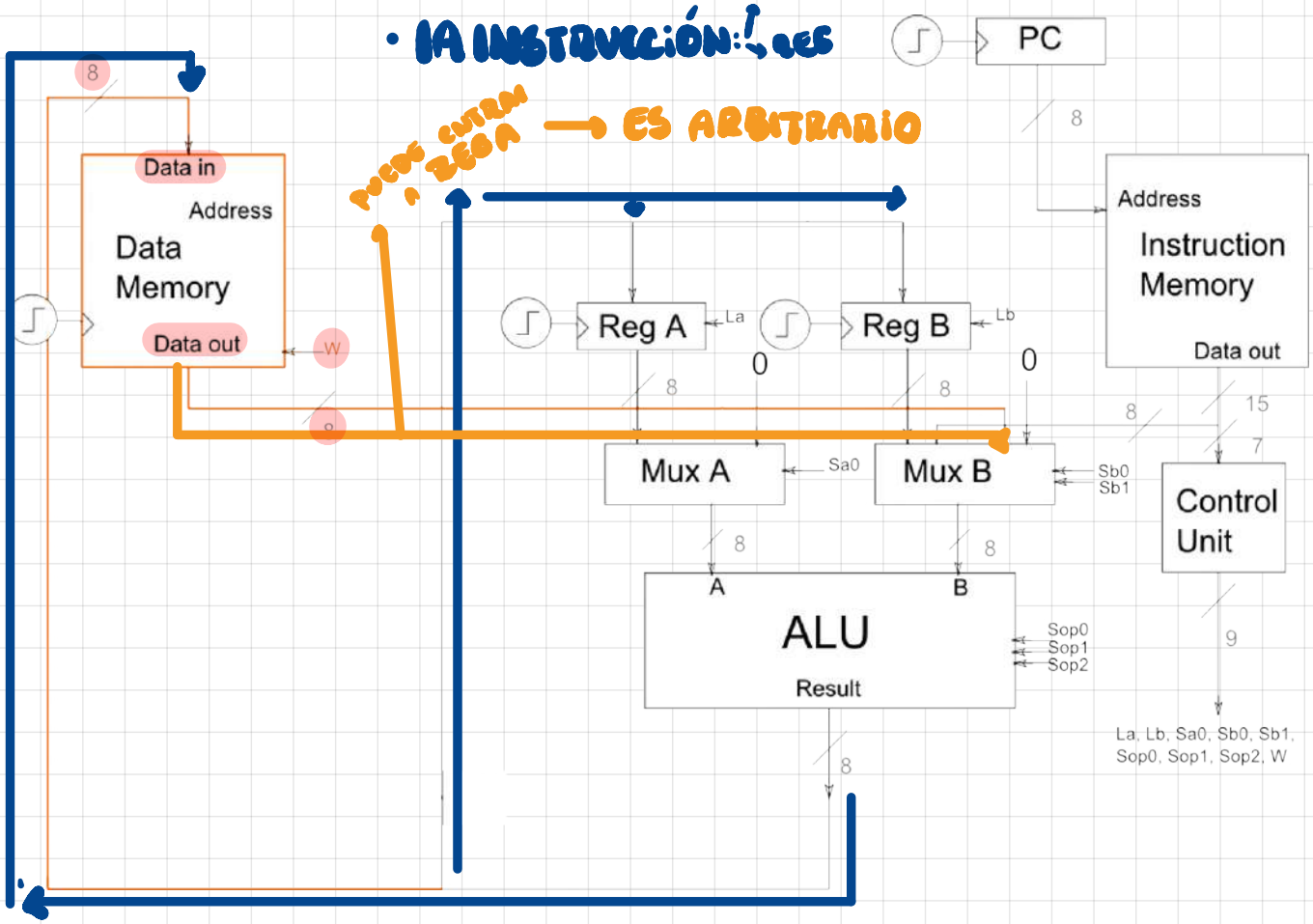
El assembler traduce a la **DIRECCIÓN EN MEMORIA**

VAR1 $\xrightarrow{\text{ASSEMBLER}}$ DIRECCIÓN MEMORIA

- **NUEVO CIRCUITO (W)**

- LA INSTRUCCIÓN: ¿D.MEM? → ES

PUEDE CAMBIAR A ZERO → ES ARBITRARIO



- **SET DE INSTRUCCIONES ACTUALIZADO**: + 32 INSTRUCCIONES a los 28 antes

ACTUAL: 60 INSTRUCCIÓN

Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
MOV	A,(Dir)	A=Mem[Dir]		MOV A,(var1)
	B,(Dir)	B=Mem[Dir]		MOV B,(var2)
	(Dir),A	Mem[Dir]=A		MOV (var1),A
	(Dir),B	Mem[Dir]=B		MOV (var2),B
	A,(B)	A=Mem[B]		-
	B,(B)	B=Mem[B]		-
	(B),A	Mem[B]=A		-
ADD	A,(Dir)	A=A+Mem[Dir]		ADD A,(var1)
	A,(B)	A=A+Mem[B]		-
	(Dir)	Mem[Dir]=A+B		ADD (var1)
SUB	A,(Dir)	A=A-Mem[Dir]		SUB A,var1
	A,(B)	A=A-Mem[B]		-
	(Dir)	Mem[Dir]=A-B		SUB (var1)
AND	A,(Dir)	A=A and Mem[Dir]		AND A,(var1)
	A,(B)	A=A and Mem[B]		-
	(Dir)	Mem[Dir]=A and B		-
OR	A,(Dir)	A=A or Mem[Dir]		OR A,(var1)
	A,(B)	A=A or Mem[B]		-
	(Dir)	Mem[Dir]=A or B		OR (var1)
NOT	A,(Dir)	A=notMem[Dir]		NOT A,(var1)
	A,(B)	A=notMem[B]		-
	(Dir)	Mem[Dir]=not A		NOT (var1)
XOR	A,(Dir)	A=A xor Mem[Dir]		XOR A,(var1)
	A,(B)	A=A xor Mem[B]		-
	(Dir)	Mem[Dir]=A xor B		XOR (var1)
SHL	A,(Dir)	A=shift left Mem[Dir]		SHL A,(var1)
	A,(B)	A=shiflt left Mem[B]		-
	(Dir)	Mem[Dir]=shift left A		SHL (var1)
SHR	A,(Dir)	A=shift right Mem[Dir]		SHR A,(var1)
	A,(B)	A=shiflt right Mem[B]		-
	(Dir)	Mem[Dir]=shift right A		SHR(var1)
INC	B	B=B+1		-

- **INSTRUCCIONES ADICIONALES**:
  - (INSTRUCCIÓN + OPERANDOS)

- MEMORIAS Y REGISTROS INTERCAMBIAN VALORES

XOR A,(DIR)

REG A XOR DIRECCIÓN MEMORIA

- DIR: POR AHORA lo USAMOS EN el Literal
  - de la instrucción

NOT A,(B)

(B) → lo vamos a interpretar como DIRECCIÓN

REG(B) = 10100110 → NO VALIDO:

- **DIRECCIONAMIENTO**:

1. **DIRECTO**: LA DIRECCION EN MEMORIA ESTA EXPLICITAMENTE EN el Literal

- El OPERANDO SE especifica como (LABEL)

2. **MOV A, (VAR1)**: GUARDAR EN REG A el CONTENIDO DE VAR1 → DIR

3. **INDIRECTO**: EN VEZ DE DIR MEMORIA: VOY AL REG
   ↓
   REG B

2. DIRECTO: VIGUÑA. M. 387...
   INDIRECTO: CAMPUS SAN J.

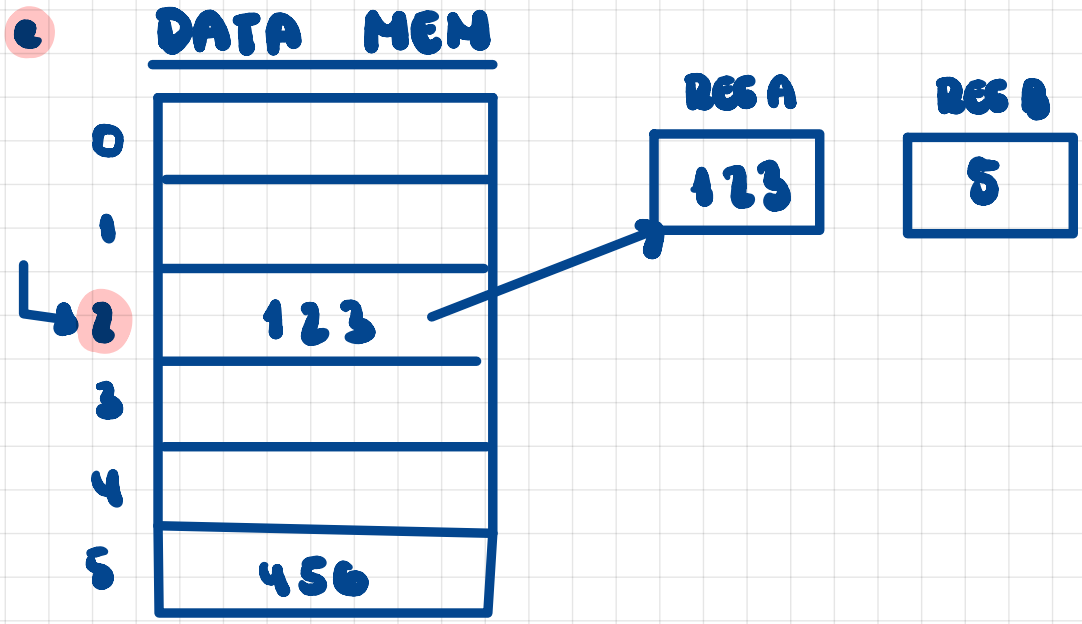
- El LITERAL: 3 INTERPRETACIONES:

DEPENDE DE LA INSTRUCCIÓN DE COMO LA INTERPRETO

1.
2. PATRON BITS → se le opera
3. **DIR MEMORIA**: 0 - 255
   ↳ solo positivos

NO SE PUEDEN SUMAR DIR DE MEMORIA

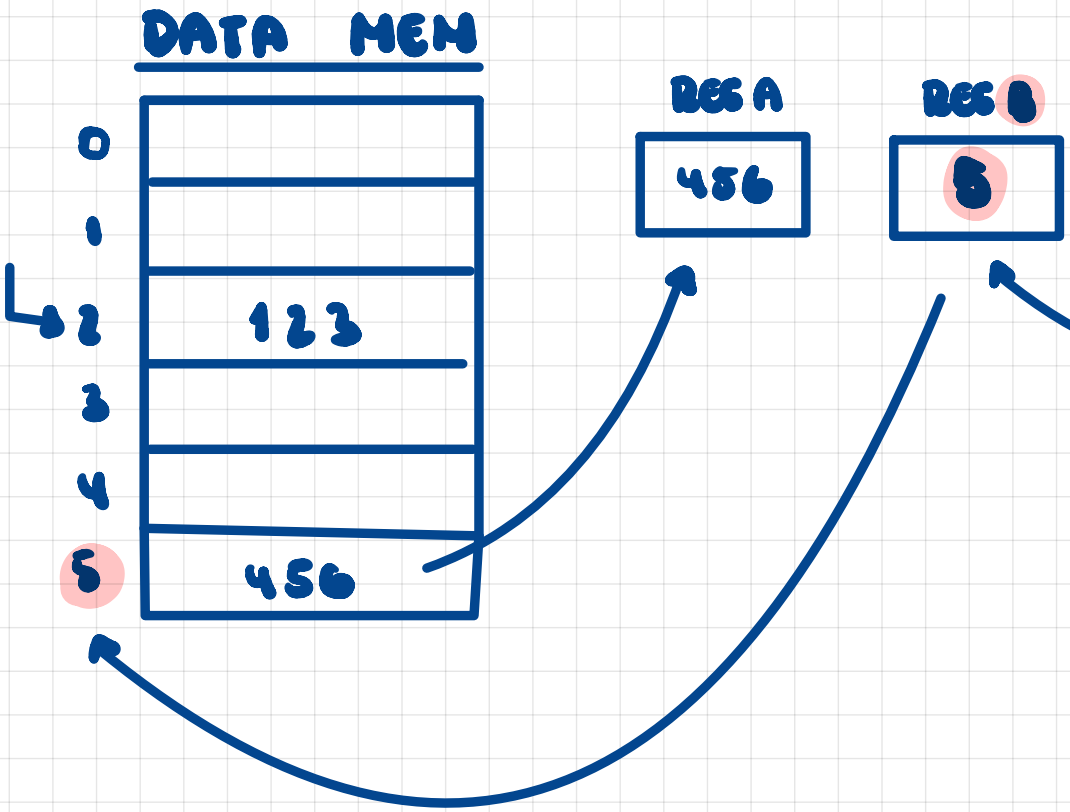
c



MOV A, (2)

↓

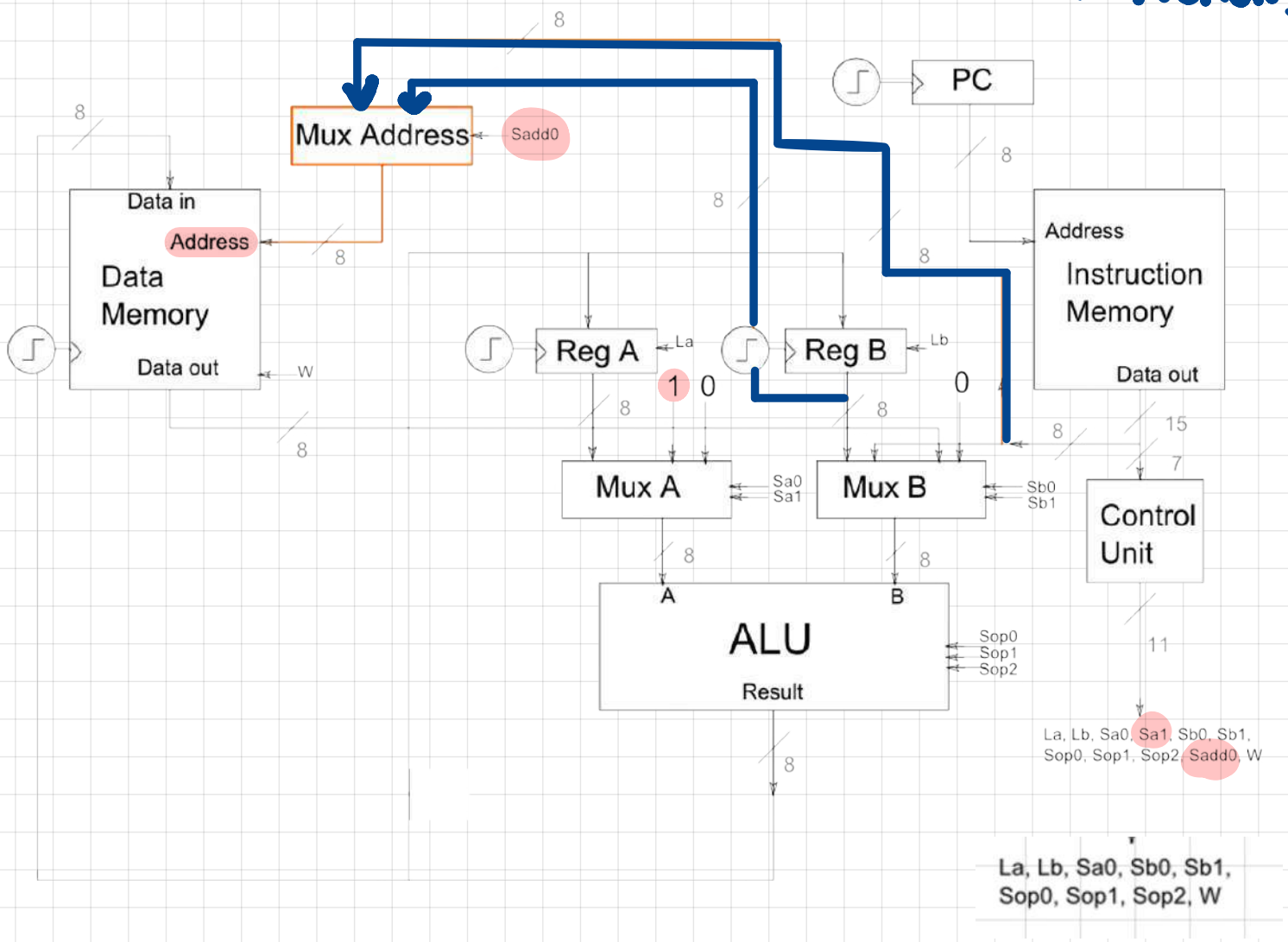
DIRECTO



MOV A, (B)

INDIRECTO

- **MUX ADDRES**: 2 INPUT 1 OUTPUT → ADRESS (DATA MEMORY)




```
DATA:
a       17
b       54
suma    —no inicializado

CODE_1: —versión 1
        MOV A,(a)
        MOV B,(b)
        ADD A,B
        MOV (suma),A

CODE_2: —versión 2
        MOV A,(a)
        MOV B,(b)
        ADD (suma)
```

- **ASSEMBLER y LOADER**: Nos facilitan TODO!

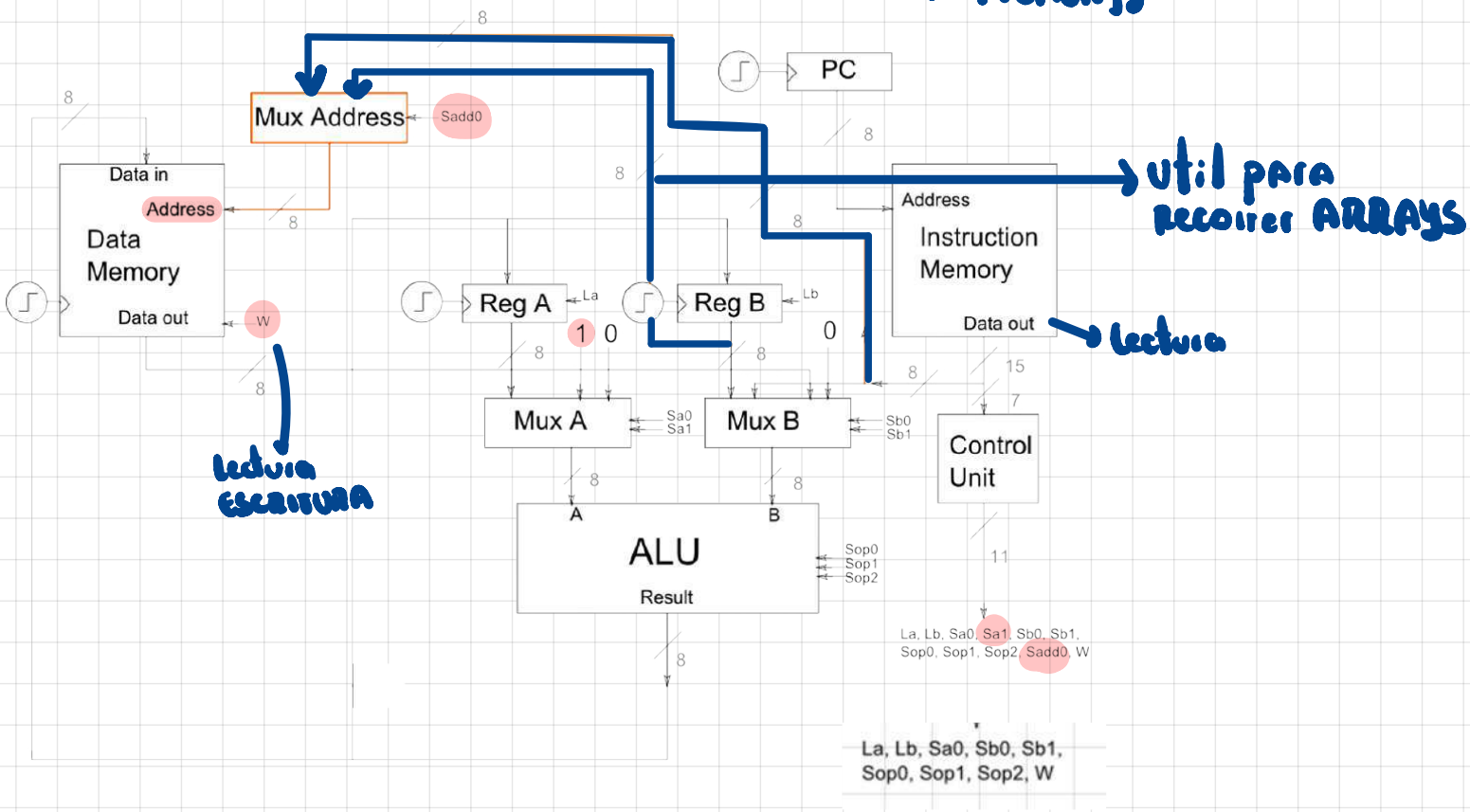
SUMA A + B → QUEDA EN REG(A)

EN (SUMA) QUEDA LO DE REG(A)

SUMAR Reg A + Reg B → GUARDAR EN (SUMA)

- **CLASE 10**

- **MUX ADDRES**: 2 INPUT 1 OUTPUT → ADRESS (DATA MEMORY)



Compilador → Assembler

↳ genera nombres: Direcciones indirectas

- **CONTROL DE FLUJO** : → CONDICIONES ← if, else, while

A = B ? → A - B = 0

1 SALTOS

a) JMP : SALTA

b) JMP CONDICIONAL : Salta si Algo

```
if x = 0 :
    do something
else :
    otra cosa
....
```

→

```
CMP A,0
JNE else
....
....
...
JMP end
else: ....
      ....
      ....
end: ...
```

JNE → JUMP NOT EQUAL

else → LOADER convierte la instrucción a una dirección

JMP end → salta al final

end: → ETIQUETAS

2 LOOPS

```
while i > 0
    .....
    i = i - 1
```

↳ necesario que termine

→

```
while : CMP A,0
        JLE end
        ...
        ...
        ...
        SUB A,1
        JMP while
end : .....
```

JLE → Jump Less or Equal

A < 0

↓

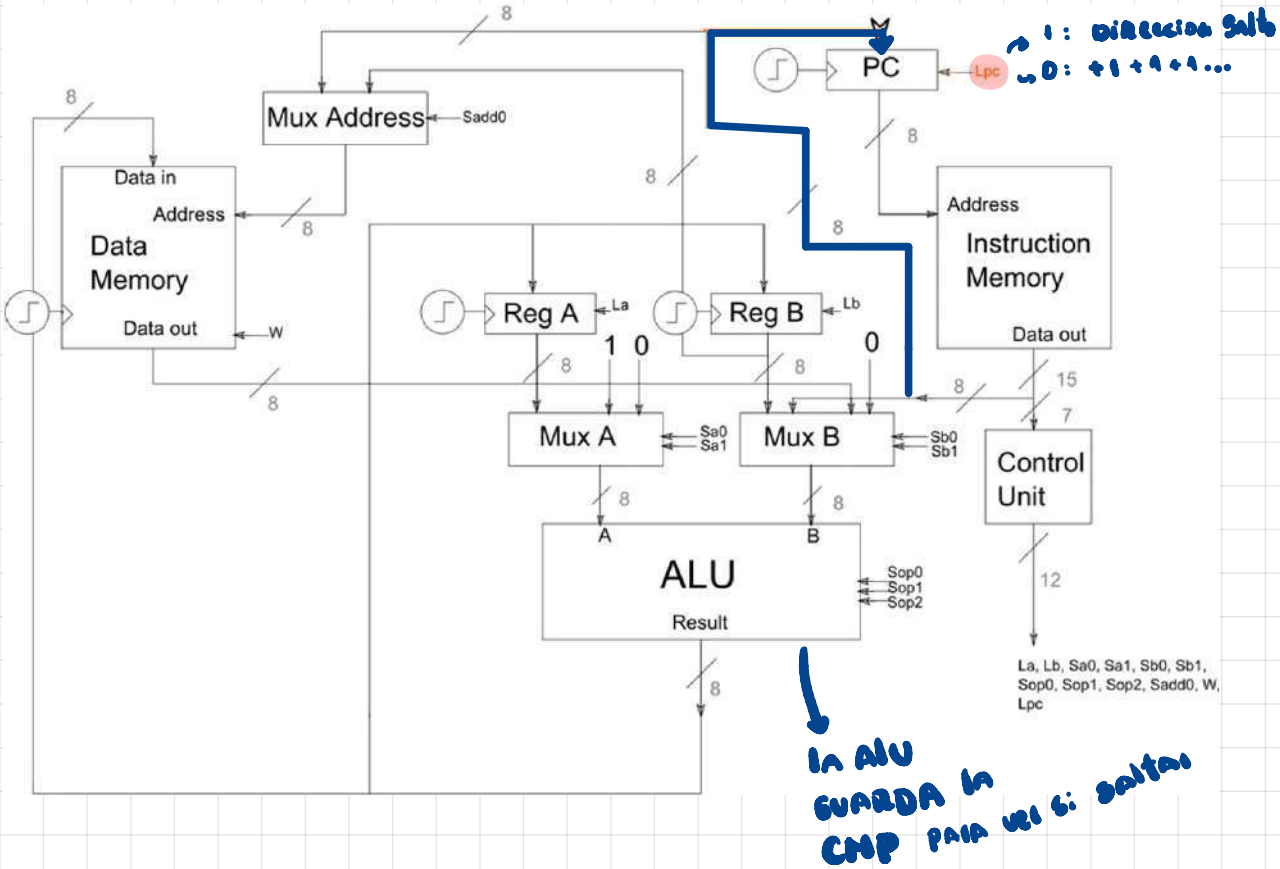
-1

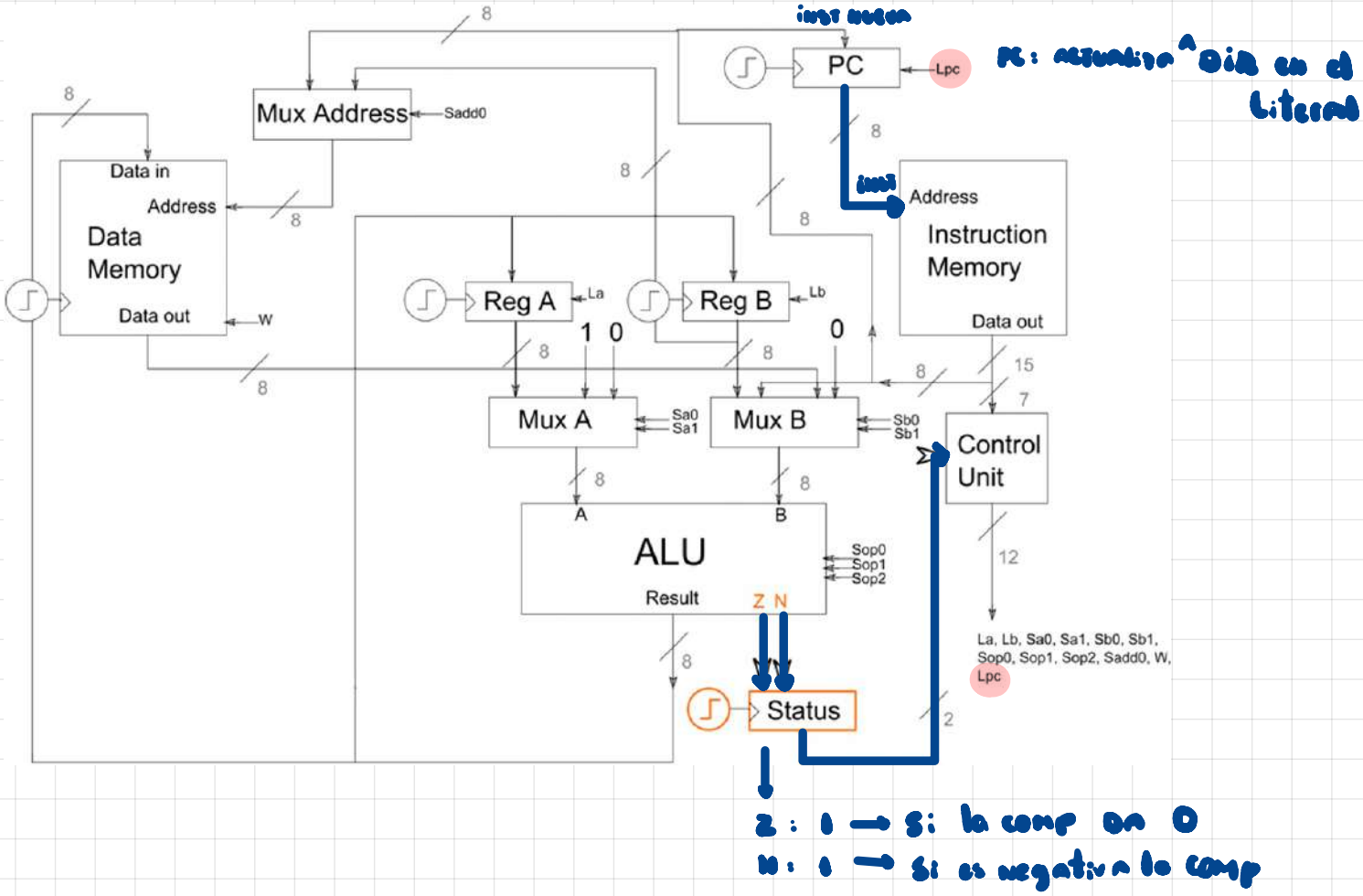
-1

-1

-1 < 0 ✓ → JUMP END

- **PC** : ACTUALIZAMOS PARA DARLE DIRECCIÓN EN SALTOS





- ## NUEVAS INSTRUCCIONES

Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
CMP	A,B	A-B		
	A,Lit	A-Lit		CMP A,0
1 JEQ =	Dir	PC = Dir	Z=1	JEQ label
2 JNE ≠	Dir	PC = Dir	Z=0	JNE label
3 JGT >	Dir	PC = Dir	N=0 y Z=0	JGT label
4 JLT <	Dir	PC = Dir	N=1	JLT label
5 JGE ≥	Dir	PC = Dir	N=0	JGE label
6 JLE ≤	Dir	PC = Dir	Z=1 o N=1	JLE label

Ocho nuevas instrucciones de nuestro lenguaje *assembly* **(que ya totaliza 69 instrucciones ⇒ opcodes de 7 bits)**:

- **CMP** toma dos operandos y los compara por la vía de restarlos

  ... el resultado queda codificado en los bits ***N*** (¿es negativo?) y ***Z*** (¿es cero?) del registro ***Status***
- las instrucciones **J*xx*** tienen un solo operando: la dirección de una instrucción en la *Instruction Memory* (especificada por una etiqueta, o *label*)

  ... esa dirección (*Dir*) es escrita en el registro *PC* si y sólo si se cumple la condición codificada en términos de los bits *N* y/o *Z*

N : ¿ES NEG!

Z : ¿ES 0!

1 **CMP** : 2 OPERANDOS y los RESTA

2 **JXX** : DIRECCIÓN (LABEL) : IR A la fuerza a label

- ## MULT 2 NUM

```
DATA:
a    15        —multiplicando
b    7         —multiplicador
prod           —producto, no inicializado

CODE:  —multiplicación, sin signo
    MOV A,0
    MOV (prod),A  —inicializamos prod = 0
    MOV B,0       —B va a ser el contador de repeticiones
    MOV A,(b)     —A = número de repeticiones (multiplicador)
loop:  CMP A,B        —comparamos
    JEQ end       —si son iguales, terminamos (saltamos a end)
    MOV A,(a)     —tomamos el multiplicando
    ADD A,(prod)  —se lo sumamos una vez más a prod
    MOV (prod),A  —y actualizamos el valor de prod
    INC B         —incrementamos el contador de repeticiones
    MOV A,(b)     —A = número de repeticiones
    JMP loop      —repetimos el ciclo
end:
```

- **CLASE 11**

- **SUBRUTINAS** (FUNCIONES, METODOS, PROCEDIMIENTOS)

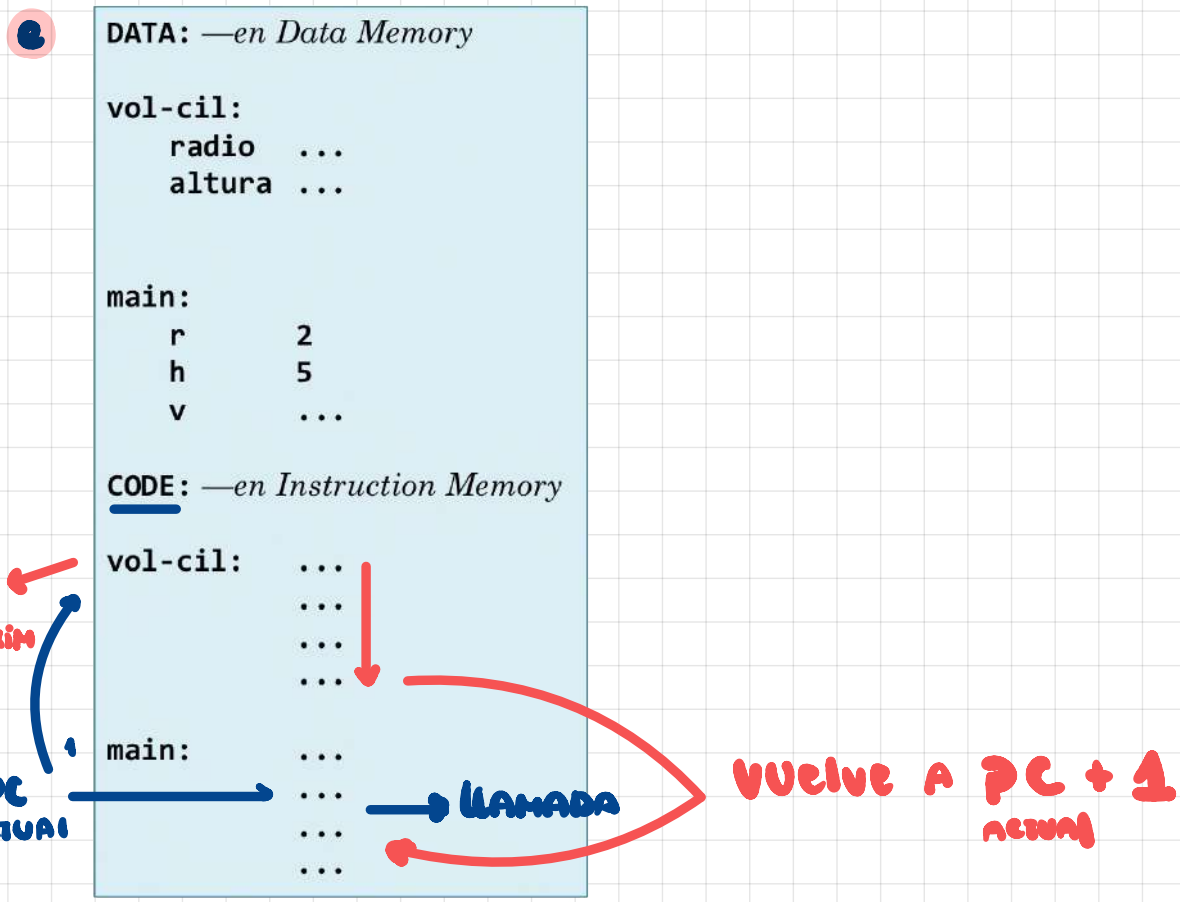
★ Facilidad para PROGRAMADORES

```
main:                    —programa principal
    r = …                —se asigna un valor a esta variable
    h = …                —se asigna un valor a esta variable
    v = vol_cil(r, h)
    print("El volumen del cilindro es", v, "cm3")

vol_cil(radio, altura):          —funcion o subrutina
    return PI*radio*radio*altura
```

LLAMADA: → PARAMETROS REALES

→ PARAMETROS FORMALES

↳ C



2) Sólo que justo antes, es necesario "pasarle" a la función **vol_cil** los valores que deben tomar los parámetros formales **radio** y **altura**

... es decir, los valores que en ese momento tienen las variables **r** y **h** (los parámetros reales)

→ hay que copiar los valores de los parámetros reales en las variables de la *Data Memory* que representan los parámetros formales (variables **radio** y **altura** a las que la función tiene acceso)

... mediante instrucciones adicionales en el **main**

Además, como la función "retorna" un valor, hay que reservar espacio en la *Data Memory* para este "valor de retorno": **retval**

VALOR DE RETORNO

```
DATA: —en Data Memory

vol-cil:
    radio   2
    altura  5
    retval  ...

main:
    r       2
    h       5
    v       ...

CODE: —en Instruction Memory

vol-cil:    ...
            ...
            ...
            ...

main:       ...
            ...
            ...
            ...
```

para el valor de retorno

a través de los registros

3 Ahora recien se ejecuta la FUNCIÓN

3) Una vez que los parámetros han sido "pasados", se ejecuta el código correspondiente a la función

... y justo antes de terminar la ejecución de la función, se calcula el valor de retorno y se almacena en la variable **retval**

```
DATA: —en Data Memory

vol-cil:
    radio   2
    altura  5
    retval  ... 63

main:
    r       2
    h       5
    v       ... 63

CODE: —en Instruction Memory

vol-cil:    ...
            ...
            ...
            ...

main:       ...
            ...
            ...
            ...
```

PODEMOS PASARLO

PC actual

PC actual

4 VOLVER DE LA SUBRUTINA AL PC anterior + 1

4) Finalmente, al terminar la ejecución de la función, hay que "pasar de vuelta", o "retornar", el valor calculado por la función:

- usando nuevamente la *Data Memory*

... y reanudar la ejecución del programa **main** en el punto en que fue suspendida:

- retomando el valor original del registro *PC* más 1
- → este valor debió haber quedado guardado en alguna parte antes de que se empezara a ejecutar la función

→ necesitamos una nueva instrucción —que hay que agregar a la función

TERMINO LA EJECUCIÓN
↳ VUELVA AL MAIN

```
DATA: —en Data Memory

vol-cil:
    radio   2
    altura  5
    retval  63

main:
    r       2
    h       5
    v       ...

CODE: —en Instruction Memory

vol-cil:    ...
            ...
            ...
            ...
main:       ...
            ...
            ...
            ...
```

a través de los registros

PC actual

próximo *PC*

¿MAS FUNCIONALIDAD? → MAS COMPONENTES

- VARIABLES? → Requiere de MEMORIA (DATA MEMORY) + CONEXIONES + INSTRUCCIONES
- 2 if! while? → Reg status + conexiones + inst
- ! SUBRUTINAS? → NUEVAS CONEXIONES e INSTRUCCIONES

INVENTAMOS INSTRUCCIÓN? → GUARDAR (PC) + 1
↳ RETORNARLO

1. CALL
1. RETURN

- **NUEVA CONEXION!**

VUELTA AL PC

JUMP

Mux : 1 ALU RESULT
1 PC + 1

... pero para manejar el valor del registro *PC* correctamente necesitamos **hardware adicional**:

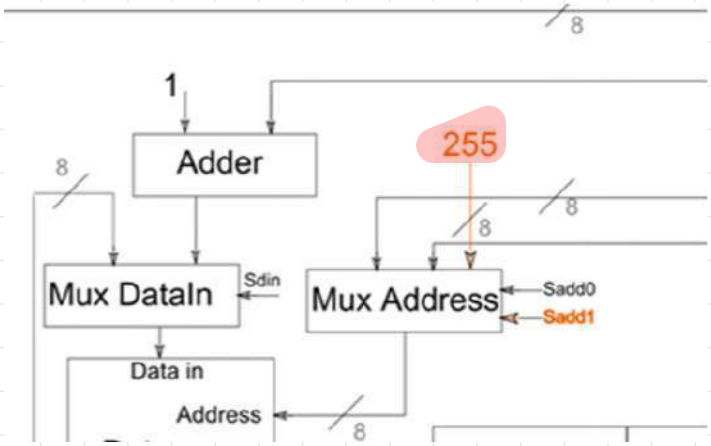
- conectamos la salida *Data out* de la memoria al registro *PC*

  ... y la salida del registro *PC* (más 1) a la entrada *Data in* de la memoria
- agregamos dos nuevos multiplexores

¿CUAL DIRECCIÓN EN MEMORIA? PARA PC

Usamos la dirección 255 —una dirección fija— de la *Data Memory* para almacenar el valor *PC* + 1

- el multiplexor *Address* tiene ahora dos señales de control



NUEVAS INSTRUCCIONES

1. **CALL DIR**: 1 ALMACENA $\boxed{PC + 1}$ en dress 255 en Mem Datos — mem [255]

   2 SALTA a DIRECCIÓN DIR $\boxed{PC \leftarrow DIR}$

2. **RET**: CARGA PC con Mem [255] (PC+1) antiguo

   → ES LA ULTIMA INSTRUCCIÓN de una FUNCIÓN

e

```
        DATA:
...
128     var1
129     var2
...
255

        CODE:
...
20      main:  MOV A,5
21             MOV B,2
22             MOV (var1),A
23             MOV (var2),B
24             CALL func1
25             ...
...
55      func1: MOV A,(var1)
56             MOV B,(var2)
57             ADD A,B
58             MOV (var1),A
59             RET
```

→ Entrega Parametros

→ llamada + mem [255] = PC + 1

→ Resultado

→ Fin y vuelta a PC + 1 (25)

• CLASE 12 : PROBLEMA

e

```
        CODE:
...
20      main:  MOV A,5
21             MOV B,2
22             MOV (var1),A
23             MOV (var2),B
24             CALL func1
25             ...
...
55      func1: MOV A,(var1)
56             MOV B,(var2)
57             ADD A,B
58             MOV (var1),A
59             CALL func2
60             RET
...
77      func2: MOV A,(var1)
78             MOV B,(var2)
79             ADD A,B
80             RET
```

PC+1

Pero, ¿qué pasa en esta caso?

main llama a **func1** → en **Mem**[255] queda almacenada la dirección de retorno 25

... pero antes de que **func1** termine (ejecute **RET**), la propia **func1** llama a su vez a la función **func2**

Entonces, la dirección de retorno que hay en **Mem**[255] en ese instante —la dirección 25— es sustituida por la dirección 60, del código de **func1**

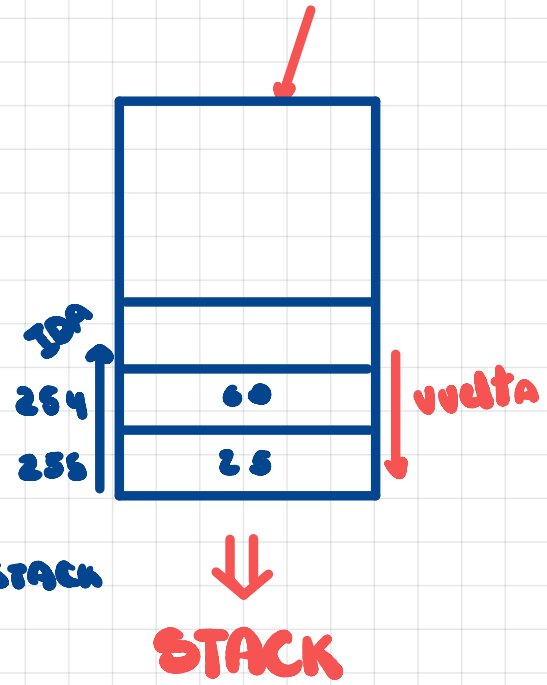
... y la ejecución nunca vuelve al **main**

→ NECESITAMOS LLAMADAS ANIDADAS

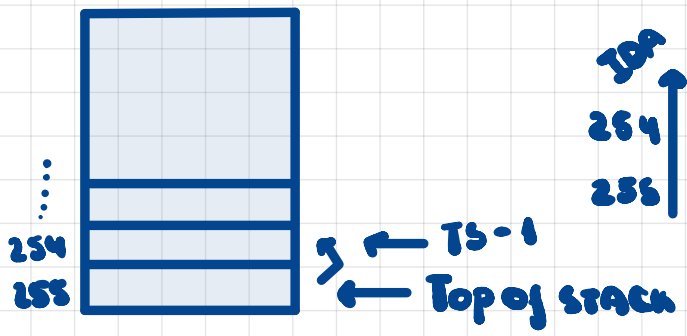
↓

Necesitamos + ESPACIO

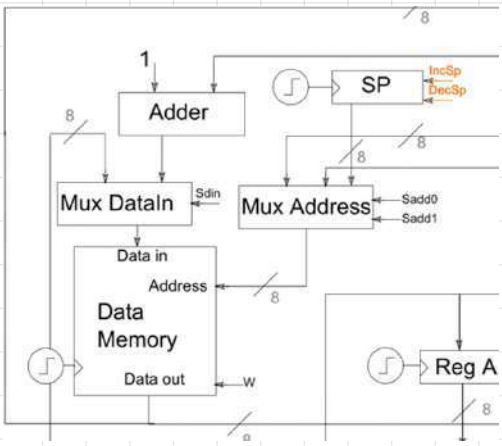
⚠ : Se perdio el 25

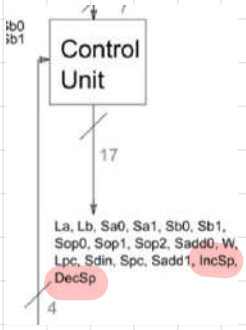


• STACK



• REG SP : ENTREGA DONDE VA el TOPE DEL STACK





En el caso de llamadas anidadas o recursivas, las sucesivas direcciones de retorno deben manejarse en un **stack**:

- el (nuevo) **registro *SP*** contiene el *stack pointer*

  ... y puede ser incrementado o decrementado

• INSTRUCCIÓN

Para estar seguros:

- el registro ***PC*** almacena direcciones de la *Instruction Memory*
- el registro ***SP*** almacena direcciones de la *Data Memory*

CALL (3 EFECTOS) → 1 ciclo

Así, al llamar a una función, debemos (en un ciclo del reloj):

- guardar ***PC***+1 en la *Data Memory* en la dirección contenida en ***SP***
- decrementar en 1 el valor de ***SP***
- guardar la dirección de (la primera instrucción de) la función en ***PC***

RET (2 EFECTOS)

... y, luego, al retornar desde la función (en dos ciclos del reloj):

- incrementar en 1 el valor de ***SP*** → CAMBIAR PUNTERO del STACK (pointer)
- guardar en ***PC*** la dirección almacenada en la dirección ***SP*** (ya incrementado) de la *Data Memory*

2 CICLOS: al usar el mismo Registro → No se puede hacer en paralelo

e

```
        CODE:
...
20      main:  MOV A,5
21             MOV B,2
22             MOV (var1),A
23             MOV (var2),B
24             CALL func1
25             ...
...
55      func1: MOV A,(var1)
56             MOV B,(var2)
57             ADD A,B
58             MOV (var1),A
59             CALL func2
60             RET
...
77      func2: MOV A,(var1)
78             MOV B,(var2)
79             ADD A,B
80             RET
```

Tomemos la ejecución justo antes de que **main** ejecute **CALL func1**

PC = 22, ***SP*** = 255
PC = 23, ***SP*** = 255

Ahora **main** hace la llamada **CALL func1**

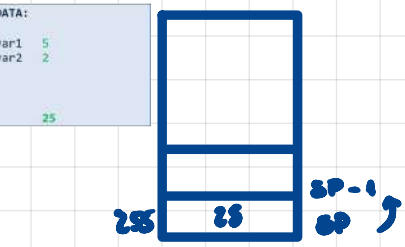
PC = 24, ***SP*** = 255

... que produce los tres efectos descritos en la diap. 20:

Mem[***SP***] ← ***PC***+1 (= 25)
SP ← ***SP***–1 (= 254)
PC ← ***dir*** (= 55)

DATA:

...		
128	var1	5
129	var2	2
...		
253		
254		
255		25



PC = 55

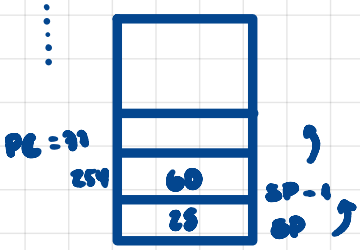
SP = 255
SP = 254

Un poco después **func1** hace la llamada **CALL func2**

PC = 59, ***SP*** = 254

... que produce los tres efectos descritos en la diap. 20:

Mem[***SP***] ← ***PC***+1 (= 60)
SP ← ***SP***–1 (= 253)
PC ← ***dir*** (= 77)



DATA:

...		
128	var1	7
129	var2	2
...		
253		
254		60
255		25

- **RETURN** : STACK pointer ++ (VUELTA : 2 ciclos (no paralelos)

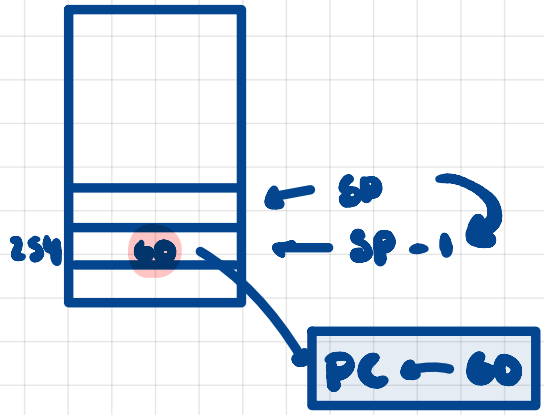
```
CODE:
...
20      main:  MOV A,5
21             MOV B,2
22             MOV (var1),A
23             MOV (var2),B
24             CALL func1
25             ...
...
55      func1: MOV A,(var1)
56             MOV B,(var2)
57             ADD A,B
58             MOV (var1),A
59             CALL func2
60             RET
...
77      func2: MOV A,(var1)
78             MOV B,(var2)
79             ADD A,B
80             RET
```

Un poco después **func2** ejecuta **RET**

PC = 80, ***SP*** = 253

... que produce los dos efectos descritos en la diap. 20:

SP ← ***SP***+1 (= 254)
PC ← Mem[***SP***] (= 60)



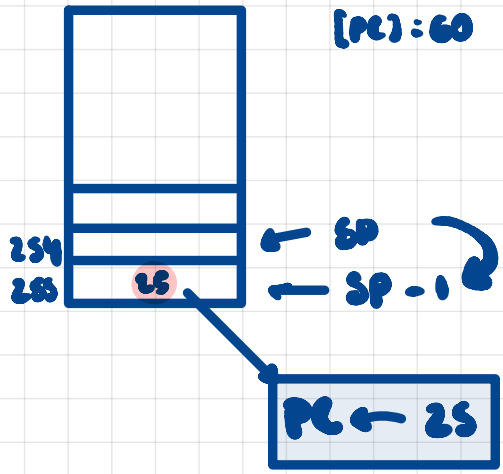
[PC] : 60

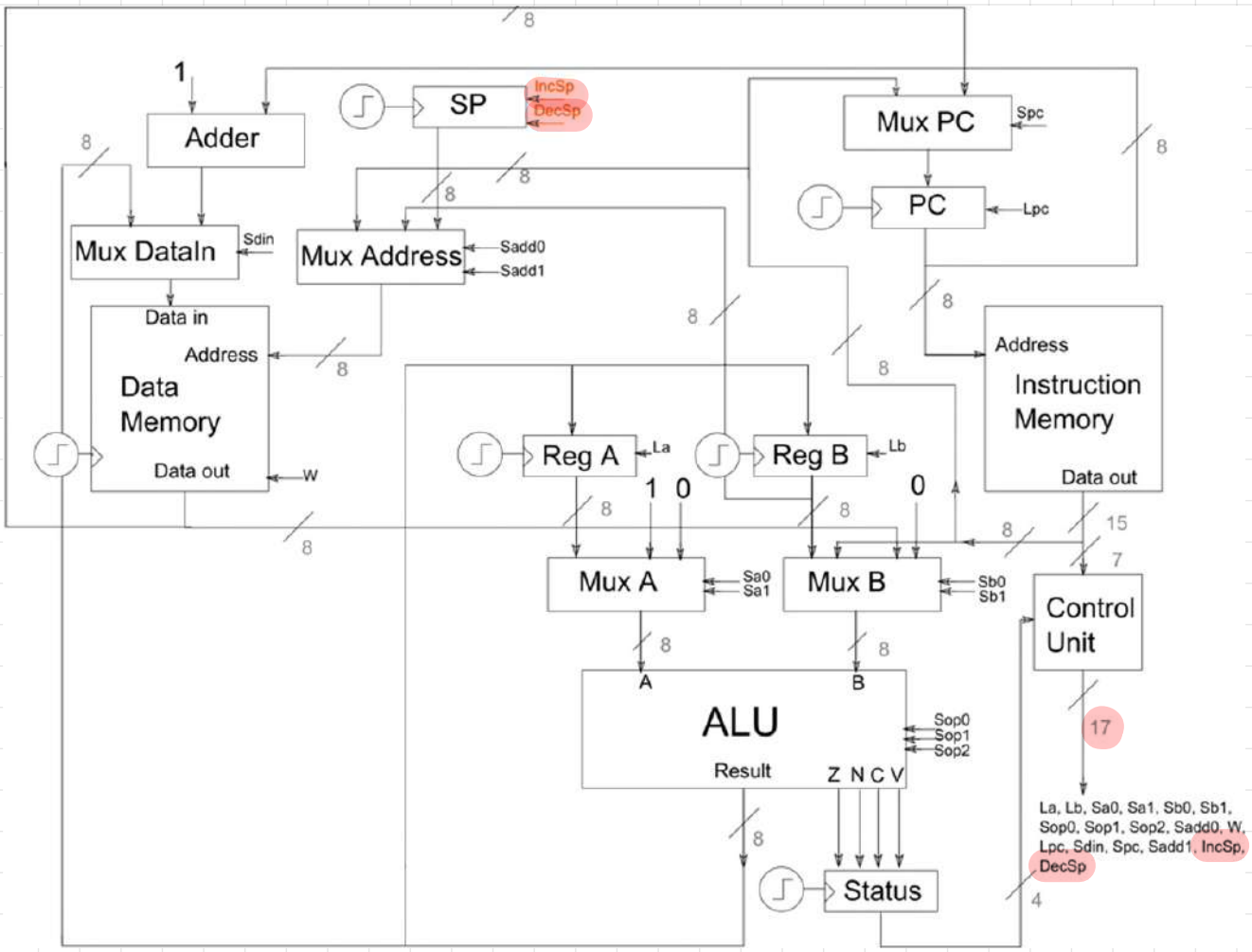
Y luego **func1** ejecuta **RET**

PC = 60, ***SP*** = 254

... que produce los dos efectos descritos en la diap. 20:

SP ← ***SP***+1 (= 255)
PC ← Mem[***SP***] (= 25)





• ¿QUE PASO CON LOS REGISTROS?

Finalmente

... ¿qué pasa en este caso con los contenidos de los registros *A* y B?

PC = 21, ***A*** = 5, ***B*** = ...
PC = 22, ***A*** = 5, ***B*** = 3
PC = 23 ⇒ ***PC*** = 55, ***A*** = 5, ***B*** = 3
PC = 55, ***A*** = 5, ***B*** = 4
PC = 56, ***A*** = 9, ***B*** = 4
PC = 57 ⇒ ***PC*** = 24, ***A*** = 9, ***B*** = 4
PC = 24, ***A*** = 13, ***B*** = 4

Pero, ¿es este el resultado esperado por `main`?

... también tenemos que guardar, en el mismo stack, los valores que los registros *A* y *B* tienen al momento de llamar a una función

... para que la función llamada pueda usar libremente estos registros

... y el `main` o la función que hizo la llamada pueda recuperar esos valores una vez que la llamada vuelve

+ 2 INSTRUCCIONES

STACK POINTER

1. **PUSH**
   a. ALMACENA EN MEM[SP] lo que hay en Reg,
   b. DECREMENTA SP

2. **POP**:
   a. INCREMENTA SP
   b. CARGA EN Reg lo que hay en MEM[SP]

Entonces, en lugar del código de la izquierda, escribimos el código de la derecha

```
CODE:
...
20      main: ...
21          MOV A,5
22          MOV B,3
23          CALL func
24          ADD A,B
25          ...
...
55      func: INC B
56          ADD A,B
57          RET
```

```
CODE:
...
20      main: ...
21          MOV A,5
22          MOV B,3
23          PUSH A
24          PUSH B
25          CALL func
26          POP B
27          POP A
28          ADD A,B
29          ...
...
55      func: INC B
56          ADD A,B
57          RET
```

¡Ojo!: los **POP**s deben ejecutarse en el orden inverso a los **PUSH**s

En definitiva, para permitir el uso de funciones, hemos agregado 6 instrucciones a nuestro *assembly*:

Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
CALL	Dir	Mem[SP] = PC + 1, SP- -, PC = Dir		CALL func
RET		SP++		-
		PC = Mem[SP]		-
PUSH	A	Mem[SP] = A, SP- - → SP - 1		-
PUSH	B	Mem[SP] = B, SP- -		-
POP	A	SP++		-
		A = Mem[SP]		-
POP	B	SP++ → SP + 1		-
		B = Mem[SP]		-

Instrucción	Operandos	Opcode	Condition	Lpc	La	Lb	Sa0,1	Sb0,1	Sop0,1,2	Sadd0,1	Sdin0	Spc0	W	IncSp	DecSp
XOR	A,B	0101000		0	1	0	A	B	XOR	-	-	-	0	0	0
	B,A	0101001		0	0	1	A	B	XOR	-	-	-	0	0	0
	A,Lit	0101010		0	1	0	A	LIT	XOR	-	-	-	0	0	0
	A,(Dir)	0101011		0	1	0	A	DOUT	XOR	LIT	-	-	0	0	0
	A,(B)	0101100		0	1	0	A	DOUT	XOR	B	-	-	0	0	0
	(Dir)	0101101		0	0	0	A	B	XOR	LIT	ALU	-	1	0	0
SHL	A,A	0101110		0	1	0	A	-	SHL	-	-	-	0	0	0
	B,A	0101111		0	0	1	A	-	SHL	-	-	-	0	0	0
	(Dir)	0110011		0	0	0	A	B	SHL	LIT	ALU	-	1	0	0
SHR	A,A	0110100		0	1	0	A	-	SHR	-	-	-	0	0	0
	B,A	0110101		0	0	1	A	-	SHR	-	-	-	0	0	0
	(Dir)	0111001		0	0	0	A	B	SHR	LIT	ALU	-	1	0	0
INC	B	0111010		0	0	1	ONE	B	ADD	-	-	-	0	0	0
CMP	A,B	0111011		0	0	0	A	B	SUB	-	-	-	0	0	0
	A,Lit	0111100		0	0	0	A	LIT	SUB	-	-	-	0	0	0
JMP	Dir	0111101		1	0	0	-	-	-	-	-	LIT	0	0	0
JEQ	Dir	0111110	Z=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JNE	Dir	0111111	Z=0	1	0	0	-	-	-	-	-	LIT	0	0	0
JGT	Dir	1000000	N=0 y Z=0	1	0	0	-	-	-	-	-	LIT	0	0	0
JLT	Dir	1000001	N=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JGE	Dir	1000010	N=0	1	0	0	-	-	-	-	-	LIT	0	0	0
JLE	Dir	1000011	N=1 o Z=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JCR	Dir	1000100	C=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JOV	Dir	1000101	V=1	1	0	0	-	-	-	-	-	LIT	0	0	0
CALL	Dir	1000101		1	0	0	-	-	-	SP	PC	LIT	1	0	1
RET		1000110		0	0	0	-	-	-	-	-	-	0	1	0
		1000111		1	0	0	-	-	-	SP	-	DOUT	0	0	0
PUSH	A	1001000		0	0	0	A	ZERO	ADD	SP	ALU	-	1	0	1
PUSH	B	1001001		0	0	0	ZERO	B	ADD	SP	ALU	-	1	0	1
POP	A	1001010		0	1	0	-	-	-	-	-	-	0	1	0
		1001011		0	1	0	ZERO	DOUT	ADD	SP	ALU	-	0	0	0
POP	B	1001100		0	0	1	-	-	-	-	-	-	0	1	0
		1001101		0	0	1	ZERO	DOUT	ADD	SP	ALU	-	0	0	0

- **CLASE 13**

```
main
    W = 3
    L = 4
    h = 5
    v = volumen(w, 1, h)
    print ("El volumen es:", v)

volumen (ancho, largo, alto):
    a = area (ancho, la'rgo)
    return area*alto

area(lado1, lado2):
    return lado1*lado2
```

```
MOV A,(W)
MOV (SP),A
SP --

MOV A,(L)
MOV (SP),A
SP --

MOV A,(h)
MOV (SP),A
SP --
PUSH A
PUSH B
```

PROTOCOLO LLAMADA

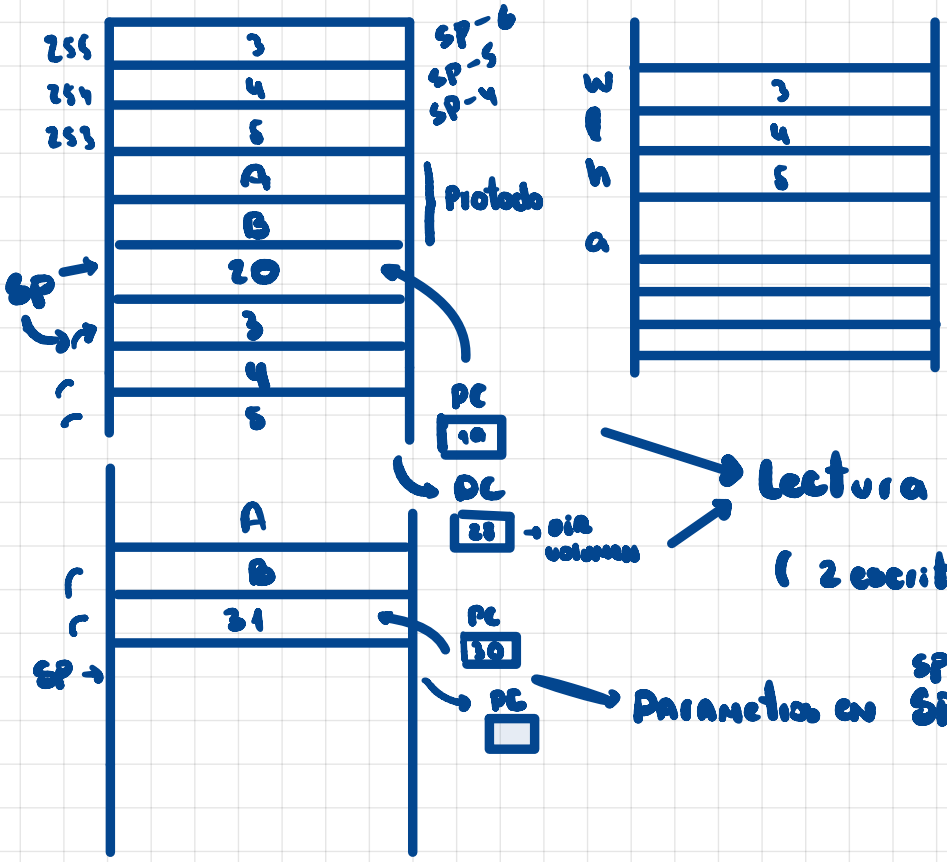
1. Pasar parametros
2. Guardar los Reg
3. CALL

PC : 19
PC + 1 → SP
SP --

CALL volumen :

CALL AREA

DATA MEM



Lectura + escritura : 1 Ciclo

(2 escrituras : 2 ciclos)

Parametros en SP-5 SP-4

```
MOV A,(SP-4)
MOV B,(SP-5)
ADD A,B      : 12 x0
MOV (SP),A
SP --
```

Se termino la ejecución → hay que devolverse

· **CLASE 14**: Repaso!

· CLASE 15: RISC-V (Lenguaje Assembly)

1: **INSTRUCCIONES de 3 operandos**: USA 3 REG

→ add a, b, c : b+c → se guarda en a

(b+c)

→ sub a, b, c : a ← b-c

2 Sumar 4 num :

```
add a, b, c
add a, a, d
add a, a, e
```

P.ej., ¿qué haría el compilador en el siguiente caso?

f ← (g+h) - (i+j)

Respuesta:

```
add t0, g, h
add t1, i, j
sub f, t0, t1
```

2 **32 REGISTROS**: en Risc-V : x0, ..., x31 de 32 bits c/u

2 P.ej., si las variables **f**, **g**, **h**, **i** y **j** están asignadas a los registros **x19**, **x20**, **x21**, **x22** y **x23**

... y usamos los registros **x5** y **x6** para almacenar los resultados intermedios

... entonces el código compilado para **f ← (g + h) - (i + j)** es

```
add x5, x20, x21
add x6, x22, x23
sub x19, x5, x6
```

* **addi**: (suma inmediata): op 3 es la constante

2 add: x22, x22, 4 : x22 ← x22 + 4

3 **INSTRUCCION DE MEMORIA**: Solo hay 2 instrucciones que leen y cargan la memoria

1 lw (load word)

2 sw (store word)

La memoria puede verse como un arreglo muy grande de celdas o casillas, en que la dirección (de memoria) actúa como índice:

- **lw** y **sw** deben especificar la dirección de la celda involucrada
- ... lo hacen de modo indirecto, por una constante y un registro → la suma de la constante más el contenido del registro

(const + Reg) → eso va a la MEMORIA

- **MEMORIA RISC-V** : 32 BITS : dir 0 → dir $2^{32} - 4$

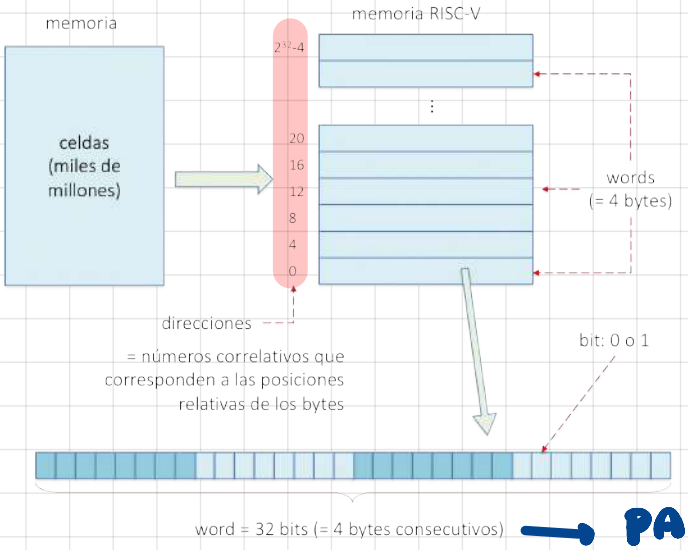
La memoria, en la práctica en todos los computadores, está organizada en bytes (8 bits consecutivos)

... en que cada byte tiene una dirección: 0, 1, 2, ...

Los computadores modernos, sin embargo, operan sobre ***palabras*** (***words***) de 4 bytes consecutivos (32 bits),

... por lo que las direcciones de estas palabras son 0, 4, 8, ...

... es decir, la dirección correspondiente al primer byte de la palabra



→ **PALABRAS** : 4 byte

`lw` : copia datos desde la memoria a un registro (*load word*) → **CARGA UN REGISTRO**

P.ej., ¿cómo traduce el compilador la sentencia **g** ← **h** + **A[8]** en C?

- el compilador coloca el arreglo **A** en memoria → sabe cuál es la dirección de memoria de **A[0]** (el primer elemento de A) y coloca esta dirección en un registro, llamado el *registro base*; p.ej., **x22**
- el compilador asocia variables con registros → supongamos que las variables **g** y **h** están asociadas a los registros **x20** y **x21**

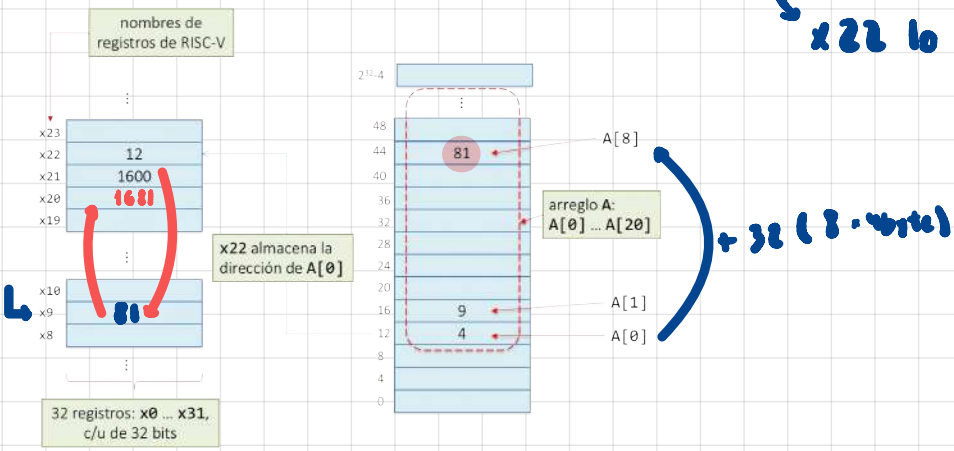
Entonces **g** ← **h** + **A[8]** es traducida por el compilador así:

8 palabras más

```
lw   x9, 32(x22)
add  x20, x21, x9
```

—dos instrucciones en assembly:
—... primero, copia A[8] *al registro* x9
—... luego, suma y pone el resultado en g

(x22) → MEM[dir] → x22 lo que haya



sw: copia datos desde un registro a la memoria (*store word*): → CARGAR EN la MEMORIA

A[12] ← h + A[8] —*sentencia en C, Python, etc.*

De nuevo, **x22** contiene la dirección de **A[0]** y **x21** está asociado a **h**:

```
lw  x9, 32(x22)
add x9, x21, x9
sw  x9, 48(x22)
```

—*tres instrucciones en assembly:*

—... *copia (load)* A[8] *al registro* x9

—... *suma* h *a* A[8]

—... *copia (store) el registro* x9 *a* A[12]

12·4 → PALABRAS

! TENEMOS las instrucciones de siempre: AND, OR, XOR, NOT, *shift left*, *shift right* y *shift right arithmetic*

- **SHIFTS**:

Los *shifts* mueven todos los bits de una palabra a la izquierda o a la derecha, llenando los bits que quedan "vacíos" con 0s:

P.ej., si **x19** contiene

0000 0000 0000 0000 0000 0000 0000 1001_2 (= 9_{10})

... y ejecutamos la instrucción

`slli x11,x19,4` —*shift left logical immediate*

... → *shift left* de 4 posiciones y dejamos el resultado en **x11**:

0000 0000 0000 0000 0000 0000 1001 0000_2 (= 144_{10})

SHL de 4 en binario == MULT 16 en decimal

El assembly también tiene la instrucción *shift right logical immediate* (**srli**)

... y las versiones en que la cantidad de bits desplazados es especificada en un registro y no como una constante:

- *shift left logical* (**sll**) y *shift right logical* (**srl**) → los normales

En el caso de *shift right*, el lenguaje tiene además las instrucciones en que los bits que quedan "vacíos" a la izquierda se llenan con copias del bit de signo original (y no necesariamente con 0s):

- *shift right arithmetic immediate* (**srai**) y *shift right arithmetic* (**sra**)

- **AND/OR**: bit a bit
  - and x9, x10, x11: a ← b&c

2 SALTOS:

* No hay saltos incondicionales: jmp label
  = beq x0, x0, label

BRANCH

Todo lenguaje de programación

... incluyendo los lenguajes de máquina y los lenguajes *assembly*

... deben tener instrucciones para controlar la ejecución condicional de otras instrucciones:

- dependiendo de los datos de entrada y de los valores creados durante la ejecución del programa,
  ... se ejecutan unas instrucciones u otras

1. *branch if equal*
   beq *registro1, registro2, L1*
   " ir a la instrucción etiquetada *L1* si el valor en *registro1* **es igual** al valor en *registro2* "

2. *branch if not equal*
   bne *registro1, registro2, L1*
   " ir a la instrucción etiquetada L1 si el valor en *registro1* **no es igual** al valor en *registro2* "

3. *branch if less than*
   blt *registro1, registro2, L1*
   " ir a la instrucción etiquetada *L1* si el valor en *registro1* **es menor que** el valor en *registro2* "

También están las instruccciones **bltu**, **bge** y **bgeu**

branch reg 1, reg 2, dir
↓
Literal

→ versión unsigned (para negativos)

4.

```
if (i = j):
    f ← g + h
else:
    f ← g - h
```

si las variables **f**, **g**, **h**, **i** y **j** corresponden a los registros **x19** a **x23**

```
        bne x22, x23, Else
        add x19, x20, x21
        beq x0, x0, Exit
Else:   sub x19, x20, x21
Exit:
```

beq x0,x0,Exit es en la práctica un branch incondicional (**x0** es siempre igual a **x0**) a la instrucción etiquetada **Exit**

} Branch incondicional

El *assembler* (ensamblador) libera al compilador y al programador de tener que calcular las direcciones (de memoria) para los *branches*

* Revisar lw → x0, sw → x0 } problematicos

5.

```
Loop:  slli   x10, x22, 2        —multiplicación i*4
       add    x10, x10, x25      —suma i + save[0]
       lw     x9, 0(x10)         —carga save[i] en x9
       bne    x9, x24, Exit      —el test del loop
       addi   x22, x22, 1        —incremento de i
       beq    x0, x0, Loop       —volvemos al principio
Exit:
```

· CLASE 16 : Funciones → Subrutinas

Una ***función*** (*procedimiento* o *subrutina*) es una herramienta que los programadores usamos para estructurar un programa:

- los programas son más fáciles de entender
- el código puede ser reusado

Los *parámetros* de la función actúan como una interfaz entre la función y el resto del programa y datos:

- pasan valores
- devuelven resultados

6 pasos

Funciones en RISC-V

Al ejecutar una función, el programa debe seguir estos seis pasos:

1. poner los parámetros en un lugar donde la función tenga acceso a ellos
2. transferir el control a la función
3. obtener la memoria necesaria para la ejecución de la función
4. ejecutar la tarea correspondiente
5. poner el resultado en un lugar donde el programa que hizo la llamada tenga acceso a ese resultado → x17 → convención
6. devolver el control al punto de origen, ya que una función puede ser llamada desde varios puntos en un programa

Los registros son el lugar de más rápido acceso a datos en un computador

En RISC-V hay 32 registros

... al llamar a una función, los usamos así:

- **x10 – x17**: ocho registros para pasar parámetros o retornar valores
- **x1**: registro para la dirección de retorno

`jal x1, direcciónDelaFunción`

... salta a la dirección *direcciónDelaFunción*

... y simultáneamente guarda la dirección de la siguiente instrucción —la **dirección de retorno**— en el registro **x1**

`jalr x0, 0(x1)`

... salta a la dirección almacenada en **x1** (**x0** está *hardwired* con el valor 0)

1. **Registros** : x10 - x17

   Se usan para parametros → convención

2. **JAL** : (Jump and Link):
   1. x1 ← PC + 4
   2. Saltar al Label

3. Resultado : en x17 ← Result :→ convención

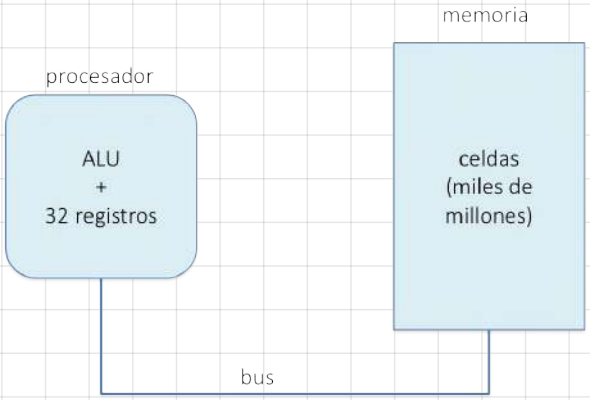
6. **JALr** devuelve PC+4

   JALR x0 , 0(x1)

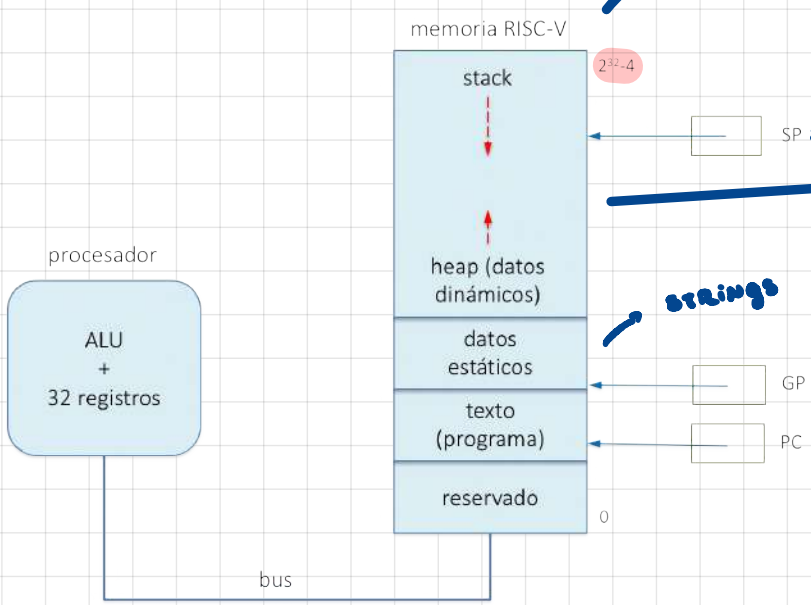
   0(x1): 0 → A[0], x1 → PC+4

   volver al punto partida

• MEMORIA RISC-V



Si se cruza SP con heap: STACK overflow



DATOS que APARECEN Y DESAPARECEN a lo largo del program

strings

Global pointer : en algun registro

El programa que hace la llamada primero pone los valores de los parámetros en los registros **x10** ... **x17** (sólo los que necesite)

... y luego ejecuta **jal x1, *F*** para guardar la dirección de retorno en **x1** y saltar (al código correspondiente) a la función ***F*** :

- por supuesto, el registro **PC** —*program counter*— tiene la dirección de la instrucción que está siendo ejecutada en este momento

  ... por lo que **jal** pone realmente **PC** + 4 en **x1**

La función ***F*** entonces hace sus cálculos (o lo que sea)

... luego pone los resultados en los mismos registros **x10** ... **x17**

... y finalmente devuelve el control al programa que hizo la llamada ejecutando **jalr x0, 0(x1)**

e

```
int xmpl(int g, int h, int i, int j):
    int f
    f ← (g+h) - (i+j)
    return f
```

```
addi    sp, sp, -12
sw      x5, 8(sp)
sw      x6, 4(sp)
sw      x20, 0(sp)

add     x5, x10, x11
add     x6, x12, x13
sub     x20, x5, x6

addi    x10, x20, 0

lw      x20, 0(sp)
lw      x6, 4(sp)
lw      x5, 8(sp)
addi    sp, sp, 12

jalr    x0, 0(x1)
```

Primero guardamos en el stack los valores de los registros **x5**, **x6** y **x20** —tres palabras (12 bytes)— para poder usarlos durante la ejecución de la función

→ Respaldo en el stack

Luego, calculamos el valor de la expresión y los guardamos en **x20**

... y para retornarlo lo guardamos en **x10** —un registro de parámetro

Antes de volver al punto donde se produjo la llamada, restauramos los valores de los tres registros guardados en el stack

... y finalmente retornamos

durante la llamada

Los parámetros **g**, **h**, **i** y **j** corresponden a los registros **x10**, **x11**, **x12** y **x13**

La variable **f** corresponde al registro **x20**

Además, para calcular el valor de **f**, usamos dos registros temporales: **x5** y **x6**

lo pasamos a x10 "Por convencion" (le sumamos 0 y lo guardo en x10)

Si al llamar a una función el compilador necesita más registros que los ocho registros **x10** ... **x17**, entonces

... todos los registros adicionales que sean usados deben ser finalmente restaurados a los valores que tenían antes de que se produjera la llamada —recuperar sus valores originales:

- → los valores originales de esos registros deben ser guardados en la memoria mientras dura la llamada a la función

La estructura de datos para esto es un ***stack*** (una cola del tipo "el último en entrar es el primero en salir", o LIFO)

... y un puntero —el ***stack pointer*** o **sp**, almacenado en el registro **x2**— a la dirección más recientemente reservada en el stack para estos registros adicionales

No siempre es necesario guardar en el stack los valores de los registros adicionales

→ TOMA tiempo: 2 ciclos, y hay datos que no son TAN RELEVANTES!

... p.ej., cuando no representan valores de variables del programa, sino resultados intermedios:

- en RISC-V, los registros **x5** a **x7** y **x28** a **x31** son registros que guardan valores temporales que no son preservados durante una llamada a una función

→ VALORES de reg: NO IMPORTAN si se sobreescriben, porque son temporales y no son importantes

- ... mientras que los registros **x8**, **x9**, y **x18** a **x27** son registros que si son usados, entonces deben ser preservados durante una llamada a una función

Esto significa que en el ej. anterior podemos ahorrarnos los dos *loads* y los dos *stores* de **x5** y **x6**

VALORES IMPORTANTES: ¡GUARDAR EN el STACK!

• RECURSIÓN : importante respaldar en el stack lo importante

```
int factorial(int n):
    if n < 1:
        return 1
    else:
        return n*factorial(n-1)
```

```
factorial:
    addi    sp, sp, -8
    sw      x1, 4(sp)
    sw      x10, 0(sp)

    addi    x5, x10, -1
    bge     x5, x0, L1

    addi    x10, x0, 1
    addi    sp, sp, 8
    jalr    x0, 0(x1)

L1:
    addi    x10, x10, -1
    jal     x1, factorial

    addi    x6, x10, 0
    lw      x10, 0(sp)
    lw      x1, 4(sp)
    addi    sp, sp, 8

    mul     x10, x10, x6

    jalr    x0, 0(x1)
```

La función es recursiva → es necesario guardar en el stack los valores de los registros **x10** (el parámetro **n**) y **x1** (la dirección de retorno), para poder recuperarlos al volver a la llamada anterior

Comparamos **n** con 1 (en realidad, **n**–1 con 0), y vamos al label **L1** si **n** ≥ 1

Si **n** < 1, entonces **factorial** devuelve 1, a través del registro **x10**

Si **n** ≥ 1, **n** es decrementado y **factorial** es llamado recursivamente

Empezamos a "salir" de (esta llamada de) la función

... ponemos en **x10** el producto de **x10** y **x6**

... finalmente, **factorial** salta de nuevo a la dirección de retorno

LLAMADA RECURSIVA

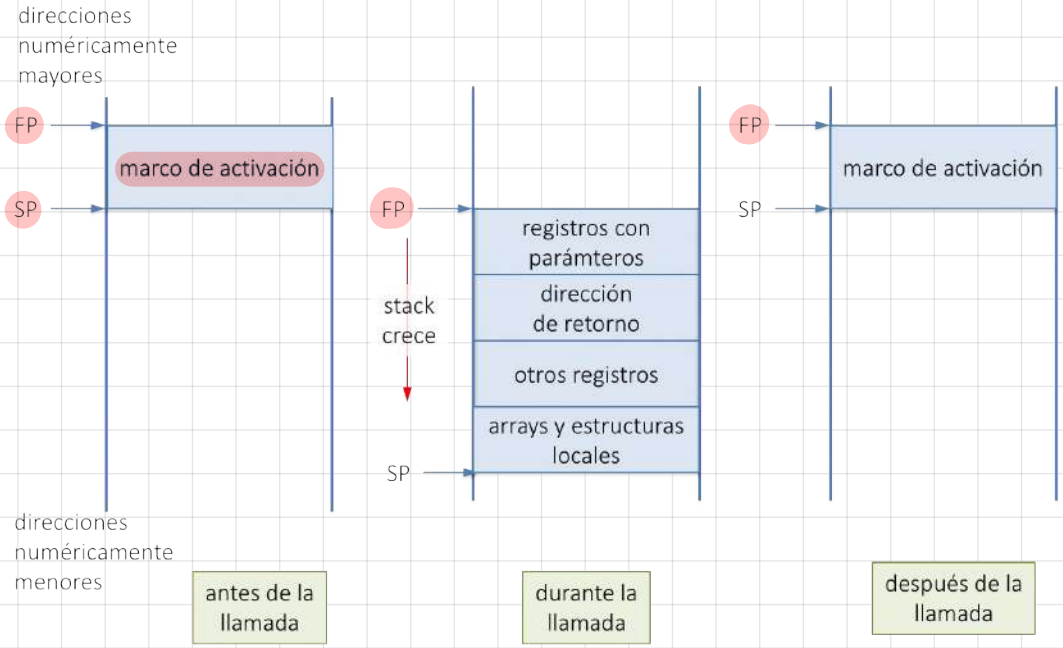
CUANDO SE DEVUELVE RETORNA DEL STACK

MULTIPLICAR 2 REG

• PUNTERO NUEVO : FP ← es donde estaba stack pointer

CONVENIENCIA

El stack, en la memoria



• FP ← SP

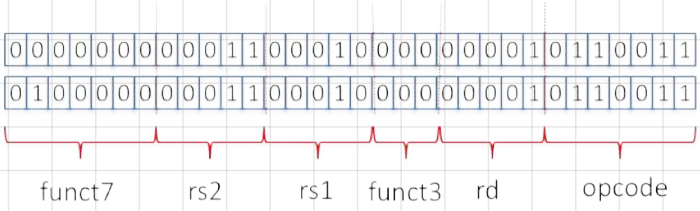
SP ← SP --

• RESUMEN DE LOS REGISTROS :(CONVENCIÓN)

nombre	número	uso	¿al stack en *call*?
x0	0	valor constante 0	n.a.
x1 (ra)	1	dirección de retorno	sí
x2 (sp)	2	stack pointer	sí
x3 (gp)	3	puntero global (al área de datos estáticos)	sí
x4 (tp)	4	puntero de *threads*	sí
x5-x7	5-7	valores temporales	no
x8 (fp)-x9	8-9	saved	sí
x10-x17	10-17	argumentos/resultados	no
x18-x27	18-27	saved	sí
x28-x31	28-31	valores temporales	no

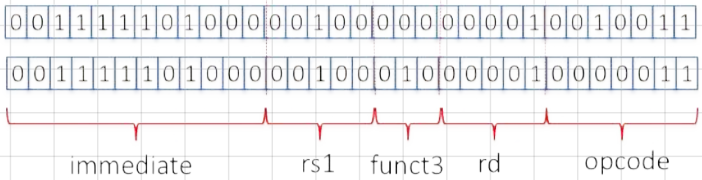
• FORMATO DE INSTRUCCIONES (RISC-V)

R-type (arithmetic)



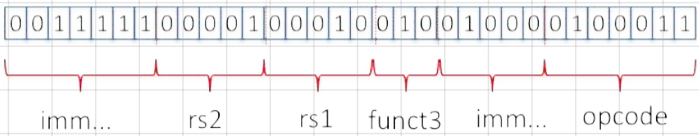

```
add x1,x2,x3
sub x1,x2,x3
```

I-type (loads & immediata)




```
addi x1,x2,1000
lw x1,1000(x2)
```

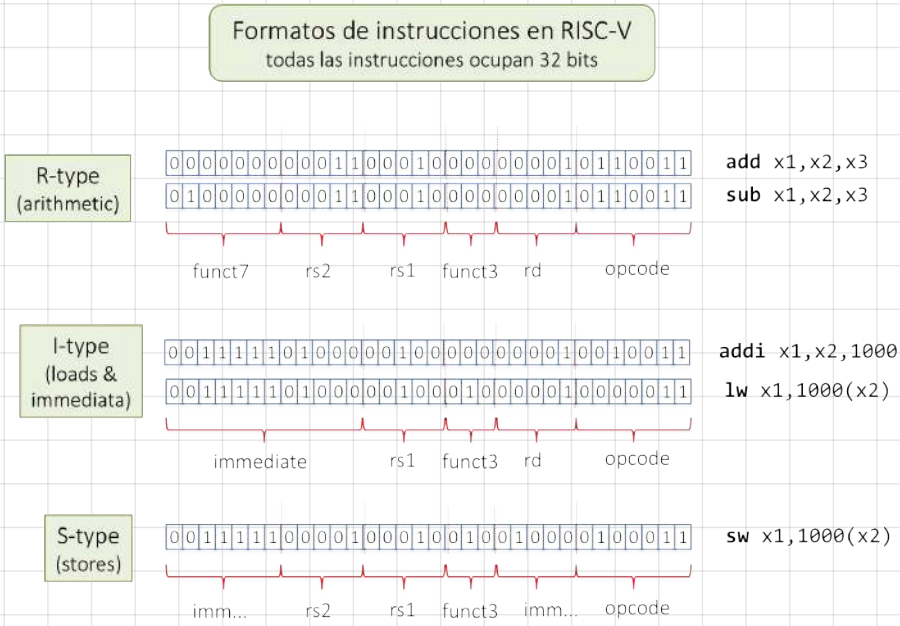
S-type (stores)



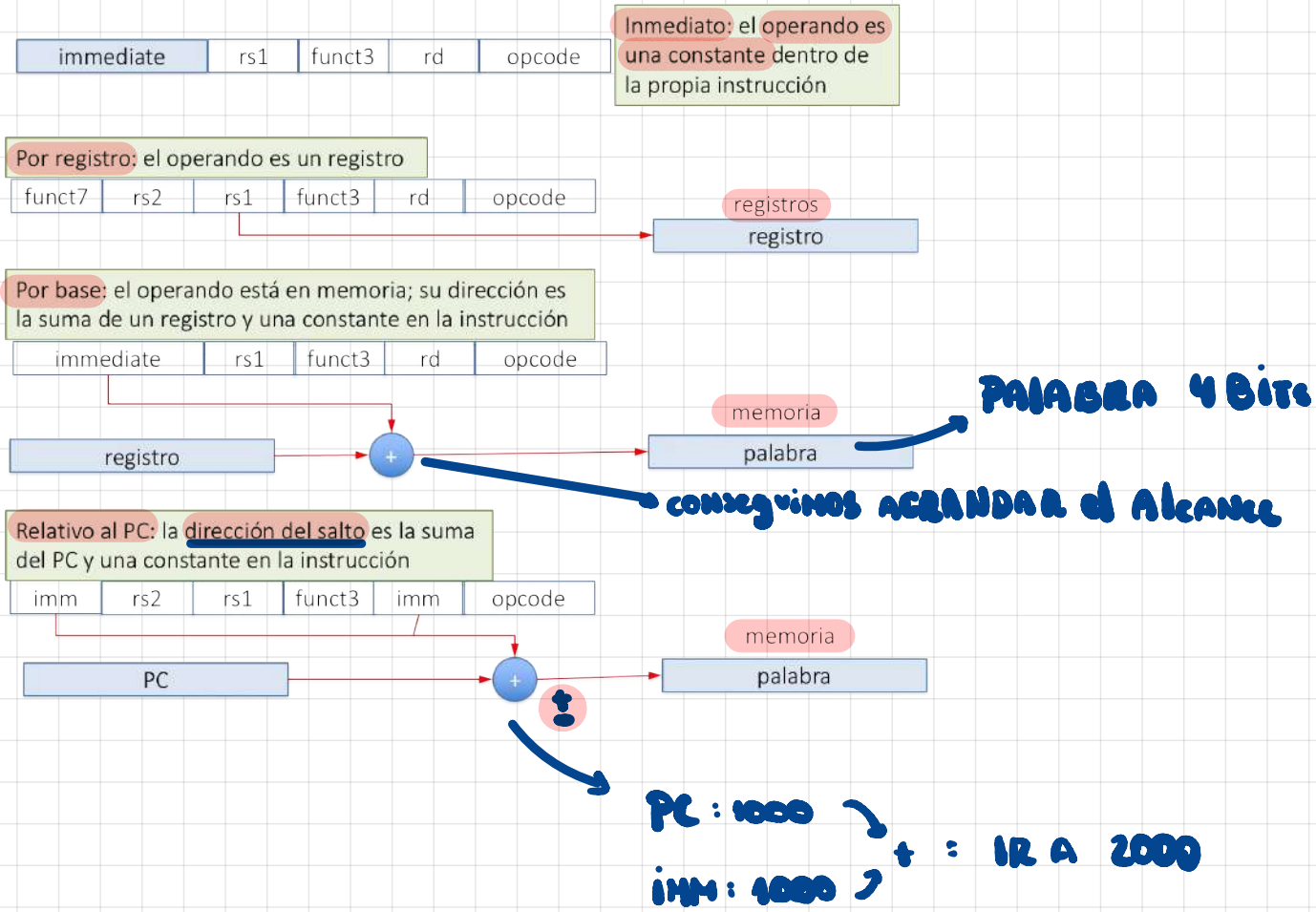

```
sw x1,1000(x2)
```

★ TODAS SON DE 32 BITS !

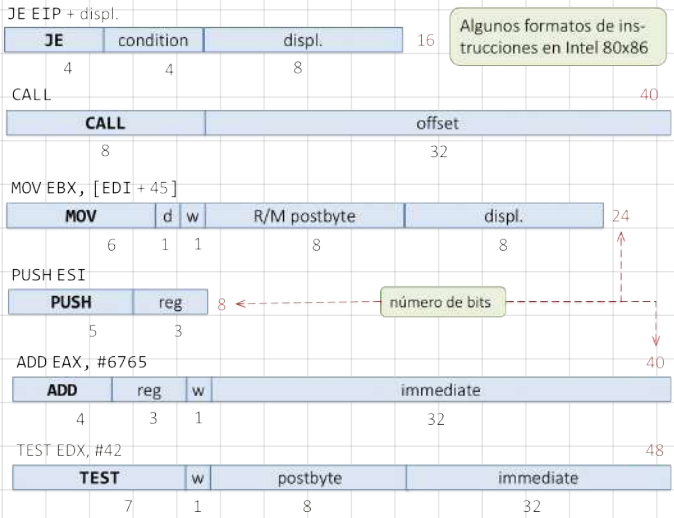
• CLASE 17



• DIRECCIONAMIENTO



EN x86



Es mas complicado para temas de HARDWARE

---

swap intercambia los valores de dos localidades en memoria: **v[k]** y **v[k+1]**

```
void swap(int v[], int k):
    int temp
    temp ← v[k]
    v[k] ← v[k+1]
    v[k+1] ← temp
```

```
swap:
    slli   x6, x11, 2
    add    x6, x10, x6
    lw     x5, 0(x6)
    lw     x7, 4(x6)
    sw     x7, 0(x6)
    sw     x5, 4(x6)
    jalr   x0, 0(x1)
```

Al traducir a assembly:
- asignar registros a las variables
- producir el código para el cuerpo de **swap**
- preservar los registros durante la llamada a **swap** (en este caso, no hay)

swap tiene dos parámetros, **v** y **k**, que usan los registros **x10** y **x11**
La variable local **temp** usa **x5**

Las direcciones de memoria son direcciones de bytes
Las direcciones de palabras consecutivas están separadas por 4—olvidarlo es un error común en programación en assembly

31

- INTERCAMBIAR VALORES
- TRAGA A LOS REGISTROS
- VOLVER
- ES UNA SUBRUTINAS
- v[0]
- v[k] → x6 esta DIR
- INT: x4

Los parametros están en: x10 - x17

- Swap(int v[0], int k)
  - v[0] → x10
  - k → x11

JALR x0, 0(x1): Volver a la direc de Retorno

- **EN RESUMEN**:

$v[k] \xrightarrow{DIR} x6 \xrightarrow{CONT} x5$

$v[k+1] \xrightarrow{DIR} x6+4 \xrightarrow{CONT} x7$

Los devuelve Cruzados

BUBBLESORT on RISC-V

```
void swap(int v[], int k):
    int temp
    temp ← v[k]
    v[k] ← v[k+1]
    v[k+1] ← temp
```

```
swap:
    slli   x6, x11, 2
    add    x6, x10, x6
    lw     x5, 0(x6)
    lw     x7, 4(x6)
    sw     x7, 0(x6)
    sw     x5, 4(x6)
    jalr   x0, 0(x1)
```

swap intercambia los valores de dos localidades en memoria: **v[k]** y **v[k+1]**

Al traducir a assembly:
- asignar registros a las variables
- producir el código para el cuerpo de **swap**
- preservar los registros durante la llamada a **swap** (en este caso, no hay)

swap tiene dos parámetros, **v** y **k**, que usan los registros **x10** y **x11**

La variable local **temp** usa **x5**

Las direcciones de memoria son direcciones de bytes

Las direcciones de palabras consecutivas están separadas por 4—olvidarlo es un error común en programación en assembly

```
void sort(int v[], int n):
    int i,j
    for (i=0, i<n, i+=1):
        for (j=i-1, j≥0 &&
            v[j]>v[j+1], j-=1):
            swap(v,j)
```

```
sort:
    addi    sp, sp, -20
    sw      x1, 16(sp)
    sw      x22, 12(sp)
    sw      x21, 8(sp)
    sw      x20, 4(sp)
    sw      x19, 0(sp)

    .
    .
    .

exit1:
    lw      x19, 0(sp)
    lw      x20, 4(sp)
    lw      x21, 8(sp)
    lw      x22,12(sp)
    lw      x1, 16(sp)
    addi    sp, sp, 20

    jalr    x0, 0(x1)
```

Ordena un arreglo de enteros usando *bubble* o *exchange sort*, muy simple aunque no muy rápido

Salvamos los registros: **i** en **x19**, **j** en **x20**

Ya vamos a ver el uso de **x21** y **x22**

El cuerpo de la función (próx. diapo.)

Restauramos los registros

Finalmente, volvemos

```
void sort(int v[], int n):
    int i,j
    for (i=0, i<n, i+=1):
        for (j=i-1, j≥0 &&
            v[j]>v[j+1], j-=1):
            swap(v,j)
```

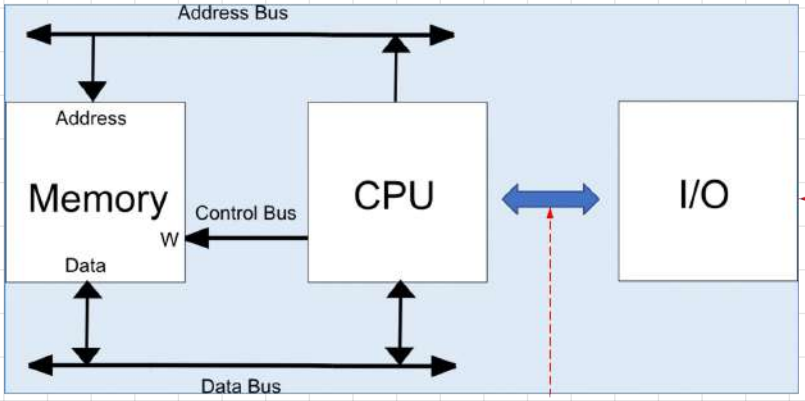
```
    addi    x21, x10, 0
    addi    x22, x11, 0
    addi    x19, x0, 0
forext
    bge     x19, x22, exit1
    addi    x20, x19, -1
forint
    blt     x20, x0, exit2
    slli    x5, x20, 2
    add     x5, x211, x5
    lw      x6, 0(x5)
    lw      x7, 4(x5)
    ble     x6, x7, exit2
    addi    x10, x21, 0
    addi    x11, x20, 0
    jal     x1, swap
    addi    x20, x20, -1
    jal     x0, forint

exit2:
    addi    x19, x19, 1
    jal     x0, forext
```

La llamada a **swap** necesita **x10** y **x11** para los parámetros de **swap**

- **CLASE 41**:

- **INPUT / OUTPUT**: MISIÓN PRINCIPAL → INTERCAMBIAR DATOS !

Un sistema computacional tiene tres componentes principales: CPU, memoria, y **dispositivos de input/output** (I/O)

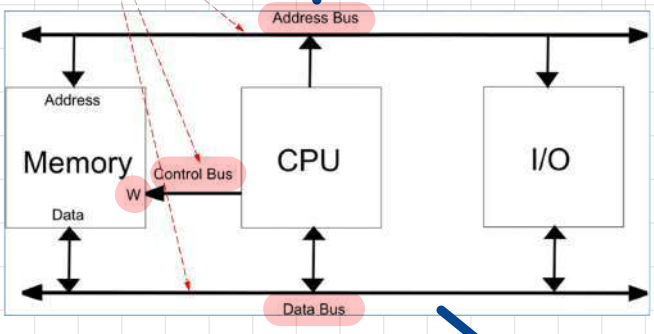


P.ej., teclado, monitor, impresora, disco duro, *modem*, mouse, cámara, audífonos, micrófono, sensores varios, manejador de DVDs, linterna, ...

Una CPU interactúa con los dispositivos de I/O por dos razones:
- controlar el dispositivo
- intercambiar datos con el dispositivo

Todos los dispositivos de I/O se comunican con la CPU y la memoria a través de **buses**

COMUNICACION: BUSES



Un **bus** es una colección de alambres paralelos:
- transmiten **datos**, **señales de control**, y **direcciones**
- son externos a la CPU (la CPU también tiene buses internos)
- en la práctica, un mismo bus tiene líneas de control, datos y dirección
- un computador tiene varios buses (CPU-memoria, CPU-I/O alta velocidad, CPU-I/O menor velocidad)

La conexión entre la CPU y un dispositivo de I/O usa el **mismo modelo que la conexión entre CPU y memoria**:
- en una operación de I/O, la CPU envía primero una dirección, y luego envía o recibe datos

LOS BUSES SE VAN A ALAIGAI, LA CPU SE COMUNICARA POR AHÍ MISMO CON LOS DISPOSITIVOS I/O

- **OPERACIONES I/O**: ¡"la CPU ve a los I/O como si fuera una memoria"!

1. **FETCH (load, RESET)**: (RECIBIR INFO): (INPUT)

   ir a buscarla y traerla

Fetch (*load, read*):

1. Usa las **líneas de control** para obtener acceso al bus
2. Coloca una dirección en las **líneas de dirección**
3. Usa las **líneas de control** para especificar que se trata de una operación fetch
4. Examina las **líneas de control** para esperar a que la operación se complete
5. Lee los valores de las **líneas de datos**
6. "Setea" las **líneas de control** para permitir que alguien más use el bus

1. **PIDE ACCESO** AL BUS (UN CONTROLADOR lo maneja)
2. **USA EL BUS DIRECCIONES** (especificar el dispositivo)
3. **ESPECIFICA QUE OPERACIÓN** (I/O) → **BUS CONTROL**: FETCH (IN)
4. **ESPERAR TERMINE LA OPERACIÓN** → **BUS CONTROL**
5. **HACE LA OPERACIÓN** (LEE VALORES)
6. "SETEA" líneas control → **LIBERAR EL BUS**!

2. **STORE (WRITE)**: lo mismo pero para (OUTPUT)

Store (*write*):

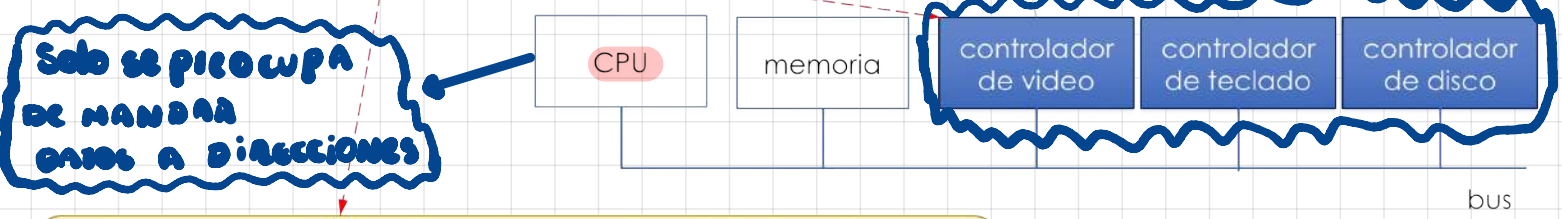
1. Usa las **líneas de control** para obtener acceso al bus
2. Coloca una dirección en las **líneas de dirección**
3. Coloca un valor en las **líneas de datos**
4. Usa las **líneas de control** para especificar que se trata de una operación store
5. Examina las **líneas de control** para esperar a que la operación se complete
6. "Setea" las **líneas de control** para permitir que alguien más use el bus

- **CONTROLADORES DEL DISPOSITIVO** (DRIVERS VERSIÓN HARDWARE)

→ INTERPRETAN LAS SEÑALES DE LA CPU, Y MANEJAN LOS DISPOSITIVOS!

Todo dispoditivo de I/O tiene dos partes:

- el dispositivo propiamente dicho, formado por elementos electromecánicos que realizan las operaciones de interacción
- un **controlador**



SOLO SE PRECUPA DE MANDAR DATOS A DIRECCIONES

Chip incluido en la placa madre (*motherboard*) del procesador, o tarjeta con circuito impreso que puede insertarse en un *slot* del bus:

- puede incluir un procesador embebido programado —el trabajo que hace el controlador es complejo
- tiene **registros**: la CPU escribe en los registros para pedir al dispositivo que envíe o acepte datos, que se encienda o se apague, etc.; la CPU lee los registros para conocer el estado del dispositivo, si está listo para recibir una solicitud, etc.
- tiene un **buffer** de datos, que puede ser leído o escrito por la CPU

El software que hay que instalar en el sistema operativo para poder darle instrucciones al controlador de un dispositivo de I/O particular se llama el **driver del dispositivo**

↓

DRIVERS

- **REGISTROS**: los dispositivos tienen Reg donde la CPU guardara datos

- **CONTROLADOR DISCO (DRIVE) → ¡NO CONFUNDIR!**

P.ej., si un programa necesita datos desde el disco:

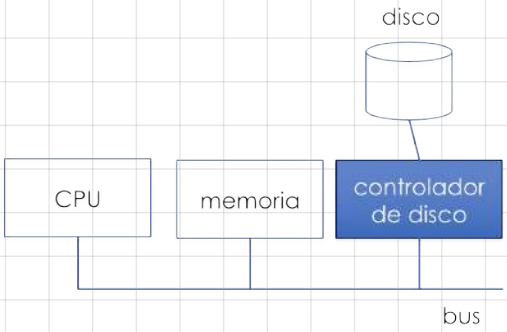
- la CPU —mediante la ejecución de instrucciones del sistema operativo— le da una instrucción al controlador del disco

  ...y el controlador le da instrucciones a su vez al control electromecánico del disco (que se conoce como el *drive* del disco)

- cuando el *drive* ha encontrado la pista y sector del disco apropiados, envía los 512 bytes del sector al controlador, en la forma de un *stream* de bits
- el controlador (re)agrupa los bits en bytes, usando el buffer interno

  ... verifica el *checksum* del bloque de bytes → EVITAR ERRORES

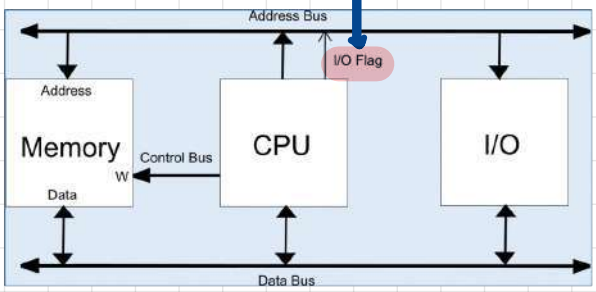
  ... y finalmente escribe el bloque en la memoria



- **PUERTOS I/O → Se le asignan NUMEROS para identificar REGISTROS**

Para indicar que una dirección es la dirección de un registro de un controlador (y no de una celda de memoria), se usa una señal de control: **I/O Flag**, una línea del bus

: líneas de control



A cada registro se le asigna un número de **puerto de I/O**

El conjunto de todos los puertos forman el **espacio de puertos de I/O**, al que solo tiene acceso el sistema operativo

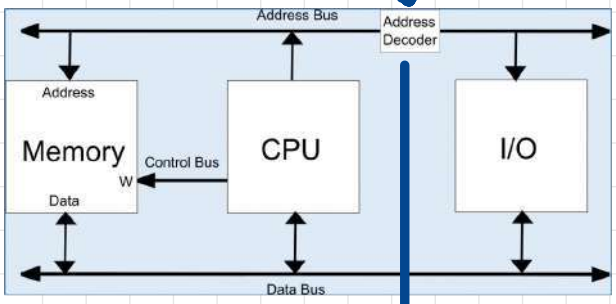
Los espacios de direcciones para la memoria y para I/O son diferentes

La CPU usa instrucciones especiales, p.ej.:

- **`IN reg,port`** : lee el registro **port** del controlador y guarda el valor en el registro **reg** de la CPU
- **`OUT port,reg`** : escribe el valor del registro **reg** de la CPU en el registro **port** del controlador

- **NUEVAS INSTRUCCIONES**:
  1. IN reg, port :
  2. OUT port, reg :

• MEMORY-MAPPED I/O (OTRA FORMA DE IDENTIFICAR LOS REGISTROS)



Los registros de los controladores son parte del mismo espacio de direcciones de la memoria del computador

A cada registro se le asigna una **dirección de memoria única**, que no está asignada a ninguna celda de memoria

... p.ej., las direcciones asignadas a los registros están en la parte alta de la numeración (el extremo 0xFFFF)

La CPU usa **instrucciones normales de acceso a memoria**

(Un "detalle" para más adelante: hay que evitar que el contenido de un registro de un controlador sea traído a la cache y que todos los accesos subsecuentes sean al valor que está en la cache)

Es necesario ocupar un *Address Decoder*: pieza de hardware que "vigila" las líneas de dirección del bus para determinar si la dirección corresponde a una celda de memoria o a un registro de un controlador

clave!

P.ej., un tablero con 16 luces conectado a un bus de 32 bits debe responder a cinco comandos:

- encender el display
- apagar el display
- encender la i-ésima luz
- apagar la i-ésima luz
- definir el brillo del display

Se eligen las siguientes direcciones, no usadas por otros dispositivos

dirección	operación	significado
1024 (a 1027)	store	valor ≠ 0 enciende el display; valor = 0, lo apaga
1024 (a 1027)	fetch	devuelve 0 si el display está apagado; devuelve ≠ 0 si está encendido
1028 (a 1031)	store	cambia el brillo según los 4 bits de datos menos significativos: valores de 0 a 15
1032 (a 1035)	store	los 16 bits menos significativos controlan c/u de las luces: 0 apaga, 1 enciende

```
if (address = 1024 && op = store && data ≠ 0):
    encender_display
elseif (address = 1024 && op = store && data = 0):
    apagar_display
```

Aunque la operación es *store*, el dispositivo—en este caso, el tablero—no es una memoria y no almacena datos para usarlos más adelante

En cambio, la interfaz del dispositivo contiene un circuito que compara los bits de la dirección especificada en la operación con las direcciones asignadas al dispositivo

... y, si coinciden, entonces activa otro circuito que implementa el *fetch* o *store* correspondiente

... p.ej. arriba vemos la versión en software (para facilitar su comprensión) del primer *store* de la tabla

¿COMO OPERAMOS EN LA PRACTICA?

↓

3 FORMAS

- **3 FORMAS CONTROL (CPU - I/O)** (I/O PROGRAM / I/O control / DMA / interrupciones)

1 I/O PROGRAMADO (busy waiting)

Es el más simple → INEFICIENTE!

→ ESPERA SIN HACER OTRAS COSAS A QUE TERMINE LA OPERACIÓN

La CPU tiene una instrucción de input (o fetch) y una instrucción de output (o store):

- cada instrucción especifica un dispositivo
- se transfiere un solo carácter entre un registro fijo de la CPU y el dispositivo
- la CPU ejecuta una secuencia fija de instrucciones por cada carácter escrito o leído (enviado al o recibido desde el dispositivo, respectivamente)

Usado en microprocesadores en sistemas embebidos o sistemas de tiempo real, en que la CPU hace todo el trabajo

- ¿Recibiste?
- ¿Recibiste?
- ¿Recibiste?
- ¿Recibiste? → Si → PASA A la sigte OP

- P.ej., supongamos un display cuyo controlador tiene cuatro registros, cada uno con una dirección (ya sea *I/O ports* o *memory-mapped I/O*):
  - dos registros de input (status y datos del teclado)
    ... y dos registros de output (status y datos de la pantalla)
  - el bit 7 del registro de status del teclado se pone en 1 cada vez que "llega" un nuevo carácter (el usuario oprimió una tecla)
    ... la CPU **lee repetidamente este registro** hasta que el bit 7 esté en 1
    ... entonces lee el registro de datos del teclado (que almacena el carácter)
  - similarmente, el bit 7 del registro de status de la pantalla se pone en 1 cada vez que ésta está lista para aceptar un carácter
    ... la CPU **lee repetidamente este registro** hasta que el bit 7 esté en 1
    ... entonces escribe un carácter en el registro de datos de la pantalla

- ESTUDIO 12

0 RISC - V

1 REGISTROS

0. x0 (Zero) : Almacena un 0
1. x1 (ra) : Return Adress (Dir de RET de Subrutinas)
2. x2 (sp) : STACK POINTER
3. x3 (gp) : Glob pointer
4. x4 (tp) : Thread pointer
5. x5 - x7, x28 - x31 (t0 - t6) : Reg Temporales (pierden valores entre Subrutinas)
6. x8 - x9, x18 - x27 (s0 - s11) : Reg CONSTANTES (mantienen valor)
7. x10 - x17 (a0 - a7) : Reg Argumentos Subrutinas
8. x10 - x11 (a0 - a1) : se utilizan tambien para VALORES RETORNO

2 DIRECTIVA

- .text (codigo)
- .data (datos globales)

- .word w1, w2, ..., wN N VALORES de 32 bits
- .byte w1, w2, ..., wN N VALORES de 8 bits

3 INSTRUCCIONES

Mnemotecnia	Instrucción	Tipo	Descripción
ADD rd, rs1, rs2	Adición	R	rd ← rs1 + rs2
SUB rd, rs1, rs2	Sustracción	R	rd ← rs1 - rs2
ADDI rd, rs1, imm12	Adición de literal	I	rd ← rs1 + imm12

add t_0, t_1, t_2

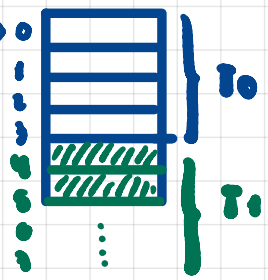
- No existe SUB LIT → add $t_0, t_0, -N$

Mnemotecnia	Instrucción	Tipo	Descripción
AND rd, rs1, rs2	Operación AND	R	rd ← rs1 & rs2
OR rd, rs1, rs2	Operación OR	R	rd ← rs1 | rs2
XOR rd, rs1, rs2	Operación XOR	R	rd ← rs1 ^ rs2
ANDI rd, rs1, imm12	Operación AND con literal	I	rd ← rs1 & imm12
ORI rd, rs1, imm12	Operación OR con literal	I	rd ← rs1 | imm12
XORI rd, rs1, imm12	Operación XOR con literal	I	rd ← rs1 ^ imm12

- **LECTURA Y ESCRITURA**

Mnemotecnia	Instrucción	Tipo	Descripción
LW rd, imm12(rs1)	**Cargar *word* (32 bits)**	**I**	**rd ← mem[rs1 + imm12]**
LH rd, imm12(rs1)	Cargar *half word* (16 bits)	I	rd ← mem[rs1 + imm12]
LB rd, imm12(rs1)	Cargar byte (8 bits)	I	rd ← mem[rs1 + imm12]
LWU rd, imm12(rs1)	Cargar *word* sin signo (32 bits)	I	rd ← mem[rs1 + imm12]
LHU rd, imm12(rs1)	Cargar *half word* sin signo (16 bits)	I	rd ← mem[rs1 + imm12]
LBU rd, imm12(rs1)	Cargar byte sin signo (8 bits)	I	rd ← mem[rs1 + imm12]
SW rs2, imm12(rs1)	**Almacenar *word* (32 bits)**	**S**	**rs2 → mem[rs1 + imm12]**
SH rs2, imm12(rs1)	Almacenar *half word* (16 bits)	S	rs2(15:0) → mem[rs1 + imm12]
SB rs2, imm12(rs1)	Almacenar byte (8 bits)	S	rs2(7:0) → mem[rs1 + imm12]

- **lw (Load word)** : Cargar palabra (32 BITS) en →



- lw t_0, 0(SP) → t_0 = MEM[SP + 0]
  [SP : SP+3]
- lw t_1, 4(SP) → t_1 = MEM[SP + 4]
  [SP+4 : SP+7]

- **sw (STORE word)** : ESCRIBE palabra (32 BITS) EN MEM

  - lw t_0, 0(SP) → MEM[SP : SP+3] = t_0
    0(DIR)

- **SALTOS**

Mnemotecnia	Instrucción	Tipo	Descripción
BEQ rs1, rs2, imm12	Salto con condición "igual"	B	if rs1 == rs2: PC ← PC + imm12
BNE rs1, rs2, imm12	Salto con condición "distinto"	B	if rs1 != rs2: PC ← PC + imm12
BGE rs1, rs2, imm12	Salto con condición "mayor o igual"	B	if rs1 >= rs2: PC ← PC + imm12
BGEU rs1, rs2, imm12	Salto con condición "mayor o igual" sin signo	B	if rs1 >= rs2: PC ← PC + imm12
BLT rs1, rs2, imm12	Salto con condición "menor"	B	if rs1 < rs2: PC ← PC + imm12
BLTU rs1, rs2, imm12	Salto con condición "menor" sin signo	B	if rs1 < rs2: PC ← PC + imm12
CALL ← JAL rd, imm20	Salto incondicional con "enlace"	J	rd ← PC+4; PC ← PC + imm20
RET ← JALR rd, imm12(rs1)	Salto incondicional con "enlace" a registro	I	rd ← PC+4; PC ← rs1 + imm12

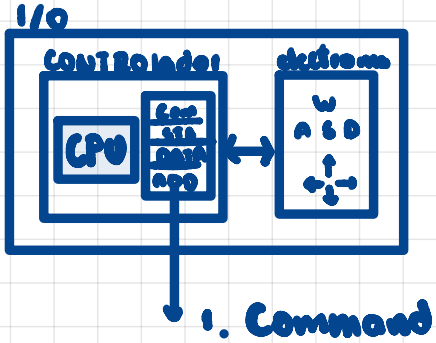
ra

Resumen de instrucciones - Pseudo-instrucciones 1/2

Mnemotecnia	Instrucción	Instrucción(es) base
LI rd, imm12	Cargar literal en registro que utiliza ≤ 12 bits	ADDI rd, zero, imm12
LI rd, imm	Cargar literal en registro que utiliza > 12 bits	LUI rd, imm[31:12]; ADDI rd, rd, imm[11:0]
LA rd, sym	Cargar dirección en registro	AUIPC rd, sym[31:12]; ADDI rd, rd, sym[11:0]
MV rd, rs	Copiar registro	ADDI rd, rs, 0
NOT rd, rs	Complemento de 1	XORI rd, rs, -1
NEG rd, rs	Complemento de 2	SUB rd, zero, rs
BGT rs1, rs2, offset	Salto si rs1 > rs2	BLT rs2, rs1, offset
BLE rs1, rs2, offset	Salto si rs1 ≤ rs2	BGE rs2, rs1, offset

- **ESTUDIO 12**

1 DISPOSITIVOS I/O



1. Command
2. Status
3. Data
4. Adress

1. **Elementos electromagnéticos**

↳ operan con interacciones

2. **Controladores**

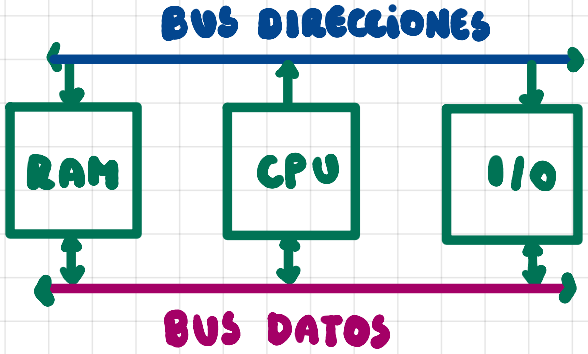
↳ controlan a los elem Elect
↳ comunicación con el COMPUTADOR

3. **Driver** : software por el cual el compu solicita comandos al CONTROLADOR

- **Buffer** : Espacio de Memoria de los I/O

1. Reg. Comando : Computador escribe instrucciones a ejecutar
2. Reg. Estados : info al compu
3. Reg. Datos : Datos al compu o al I/O
4. Reg. Dirección : Direcciones ↳ (Disco duro)

- **Esquema**



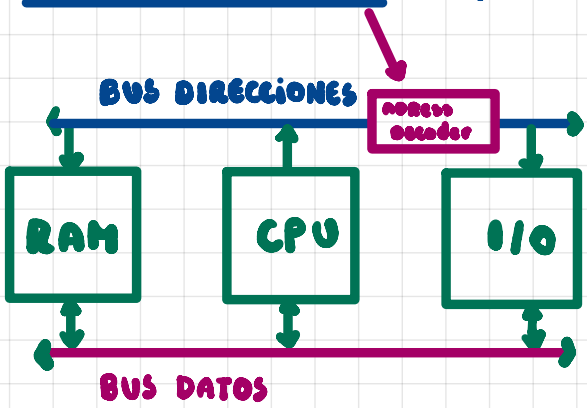
- **3 Tipos comunicación**

1. **Comunic. comandos** : CPU → I/O ↗ CPU escribe en REG COMAN ↳ ordena una **operación**
2. **Comunic. estados** : I/O → CPU ↗ I/O actualiza REG ESTADO ↳ **CPU lo lee!**
3. **Transf. datos** : MEM ⟷ I/O ↘

- **Tipos de TRANSF. DATOS**

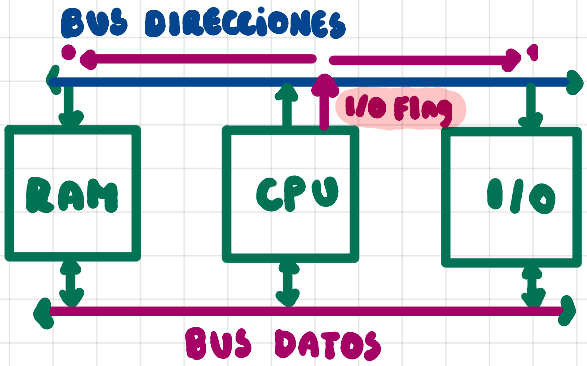
1. **MEMORY MAPPED I/O** : (Reutilizar la ISA)

- Reserva espacios en MEMORIA para "MAPEAR" reg y buffer del disp I/O
- **ADRESS DECODER** : Opera sobre la DIRECCION → la TRANSFORMA a la dir "REAL" del DISPOSITIVO



- **Desventaja** : Pierdo espacio en MEMORIA

2. **PORT I/O** : (Nuevas instrucciones) → Puertos + 1 Flag



1. in Reg, port → Leer
2. OUT port, reg → Escribir

- **Desventaja** : Modificar Arquitect.

Ejemplo I/O + RISC V

1. Prender impresora
2. Si imprime → Terminar proceso
3. Si no imprime → Imprime :
   1. hoja color
   2. Tamaño oficio
   3. Vertical

Direcciones :

Dirección de memoria	Dirección I/O
0x70	Registro de estado impresora.
0x71	Registro de comando impresora.

Especificaciones :

Valor registro	Estado
0x00	Impresora apagada.
0x01	Impresora prendida.
0x02	Impresora imprimiendo.

Valor registro	Modo de color
0x00	Blanco y negro.
0x01	Color.

Valor registro	Comando
0x00	Apagar impresora.
0x01	Prender impresora.
0x02	Imprimir.

Valor registro	Tamaño
0x00	Carta.
0x01	Oficio.

Valor registro	Orientación
0x00	Vertical.
0x01	Horizontal.

Assembly

```
DATA:                   ;Se podrían guardar las direcciones como variables,
CODE:                   ;pero usaremos directamente los literales.
  main:
    MOV B,112           ;B = 0x70 = Registro de estado
    MOV A,(B)
    CMP A,2
    JEQ end             ;Está imprimiendo, no hacemos nada.
    CMP A,1
    JEQ print           ;Está prendida, imprimimos.
  turnOn:               ;Está apagada, la prendemos e imprimimos.
    MOV B,113           ;B = 0x71 = Registro de comando.
    MOV A,1
    MOV (B),A           ;Prender impresora, luego imprimir.
  print:
    MOV A,0
    OUT 2,A             ;Orientación vertical
    MOV A,1
    OUT 0,A             ;Impresión a color
    OUT 1,A             ;Tamaño oficio
    MOV B,113           ;B = 0x71 = Registro de comando
    MOV A,2
    MOV (B),A           ;Imprimir
  end:
```

RISC - V :

Direcciones (PALABRAS) :

Dirección de memoria	Dirección I/O
0x70	Registro de estado impresora.
0x74	Registro de comando impresora.
0x78	Registro de modo de color impresora.
0x7C	Registro de tamaño de papel impresora.
0x80	Registro de orientación de impresión impresora.

```
.text                   # Se podrían guardar las direcciones como variables,
  main:                 # pero usaremos directamente los literales.
    li s0, 112          # s0 = 0x70 = Registro de estado
    li s1, 2            # s1 = Estado "imprimiendo"
    li s2, 1            # s2 = Estado "prendida"
    li s3, 1            # s3 = Comando "prender"
    li s4, 1            # s4 = Impresión a color
    li s5, 1            # s5 = Tamaño oficio
    li s6, 0            # s6 = Orientación vertical
    li s7, 2            # s7 = Comando "imprimir"
    lw t0, 0(s0)        # t0 = Valor de registro de estado
    beq t0, s1, end     # Está imprimiendo, no hacemos nada.
    beq t0, s2, print   # Está prendida, imprimimos.
  # Está apagada, la prendemos e imprimimos.
  turnOn:               # Prender impresora, luego imprimir.
    sw s3, 4(s0)        # s0 + 4 = 0x74 = Registro de comando.
  print:
    sw s4, 8(s0)        # s0 + 8  = 0x78 = Registro de modo de color.
    sw s5, 12(s0)       # s0 + 12 = 0x7C = Registro de tamaño de papel.
    sw s6, 16(s0)       # s0 + 16 = 0x80 = Registro de orientación impresión.
    sw s7, 4(s0)        # Imprimir.
  end:
```

• COMUNICACIÓN I/O → CPU

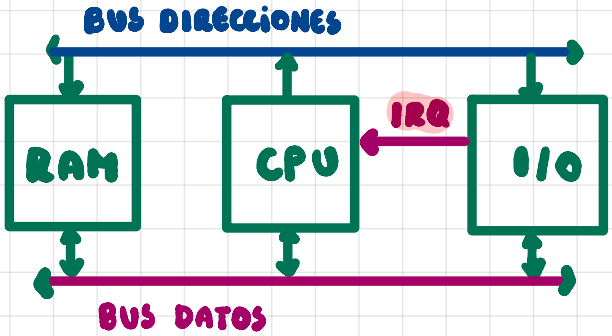
1. **POLLING** la cpu **CONSULTA** reiteradamente **REG. ESTADOS**
   - Ventaja: No modificar ARQUI
   - **Desventajas**: Muy ineficiente para muchos Dispositivos

```
turnOn:                 ;Está apagada, la prendemos e imprimimos.
  MOV B,113             ;B = 0x71 = Registro de comando.
  MOV A,1
  MOV (B),A             ;Prender impresora, luego imprimir.
waitTurningOn:
  MOV B,112             ;B = 0x70 = Registro de estado
  MOV A,(B)
  CMP A,1
  JNE waitTurningOn     ;Si todavía no está prendida, repetimos la pregunta.
print:
  MOV A,0
  OUT 2,A               ;Orientación vertical
  MOV A,1
  OUT 0,A               ;Impresión a color
  OUT 1,A               ;Orientación horizontal
  MOV B,113             ;B = 0x71 = Registro de comando
  MOV A,2
  MOV (B),A             ;Imprimir
end:
```

→ Preguntar reiteradamente

```
.text                       # Se podrían guardar las direcciones como variables,
  main:                     # pero usaremos directamente los literales.
    li s0, 112              # s0 = 0x70 = Registro de estado
    li s1, 2                # s1 = Estado "imprimiendo"
    li s2, 1                # s2 = Estado "prendida"
    li s3, 3                # s3 = Estado "prendiendo"
    li s4, 1                # s4 = Comando "prender"
    li s5, 1                # s5 = Impresión a color
    li s6, 1                # s6 = Tamaño oficio
    li s7, 0                # s7 = Orientación vertical
    li s8, 2                # s8 = Comando "imprimir"
    lw t0, 0(s0)            # t0 = Valor de registro de estado
    beq t0, s1, end         # Está imprimiendo, no hacemos nada.
    beq t0, s2, print       # Está prendida, imprimimos.
    beq t0, s3, waitTurningOn # Está prendida, imprimimos.
  # Está apagada, la prendemos y esperamos a que prenda para imprimir.
  turnOn:                   # Prender impresora, luego esperar a que prenda.
    sw s4, 4(s0)            # s0 + 4 = 0x74 = Registro de comando.
  waitTurningOn:
    lw t0, 0(s0)
    beq t0, s3, waitTurningOn # Si todavía no está prendida, repetimos la pregunta.
  print:
    sw s5, 8(s0)            # s0 + 8  = 0x78 = Registro de modo de color.
    sw s6, 12(s0)           # s0 + 12 = 0x7C = Registro de tamaño de papel.
    sw s7, 16(s0)           # s0 + 16 = 0x80 = Registro de orientación impresión.
    sw s8, 4(s0)            # Imprimir.
  end:
```

2. **INTERRUPCIONES**: los Dispositivos AVISAN a la CPU



- **IRQ**: señal
- **STATUS** (flags): **IF** → indica si la CPU puede atender interrupciones
- **ISR**: Subrutina ejecutada por la interrupción

• Pasos de interrupciones

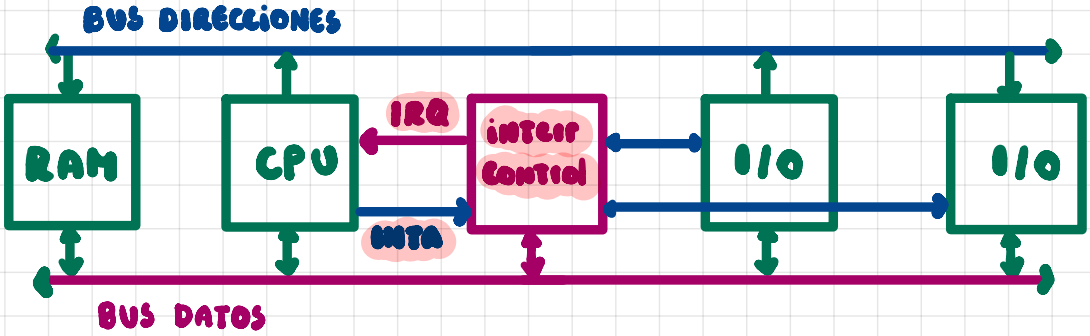
Pasos a seguir después de una interrupción

1. Dispositivo interrumpe (IRQ = 1, transmitida a la CPU).
2. CPU termina la ejecución de la instrucción actual y guarda sus *condition codes* (registro *Status*) en el *stack*.
3. CPU revisa si el *flag* de interrupciones está activo (IF = 1). Si no es el caso, saltamos al paso 11.
4. CPU deshabilita la atención de más interrupciones (IF = 0).
5. CPU llama a la ISR asociada al dispositivo (`CALL`).
6. ISR respalda el estado actual de la CPU (x86: `PUSHA` respalda en el *stack* todos los registros de propósito general).
7. ISR ejecuta su código.
8. ISR devuelve el estado previo a la CPU (x86: `POPA` restaura desde el *stack* todos los registros de propósito general).
9. ISR retorna (x86: `IRET`).
10. CPU habilita la atención de interrupciones (IF = 1).
11. CPU recupera *condition codes* desde el *stack* y retoma su ejecución.

* En el paso 3, la revisión del *flag* IF modifica los *condition codes* al hacer una comparación en la ALU. Por ese motivo es importante respaldarlos antes.

• Hardware adicional

1. **Controlador interrup**: Recibe las solicitudes de los I/O
2. **INTA**: Señal enviada de la CPU → Controlador con el ID del I/O



• Vector interrupciones

! Controlador

1. Reg. Comando y Estado
2. **IRR**: reg de interrup en **ESPERA**
3. **InSR**: reg de interrup en **ATENCIÓN**
4. **IMR**: reg de int. **IGNORADAS**
5. Circuito de **Prioridades** de int

- **Tipos Interrupciones**

1. **Excepciones** : • Error al ejecutar una instrucción
   • Atendidos por CPU (S.O) a traves de ISR's
   - Stack overflow

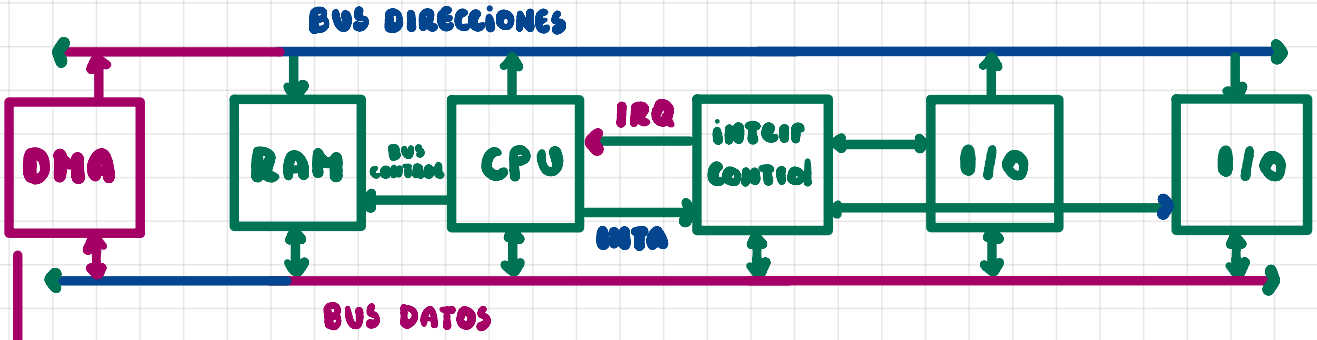
2. **Traps (software)** : Interr explicitos para ceder control al SO
   - ecall en RISC-V

- **Transferencia Datos** (Memoria y I/O)

1. **Programed I/O** (PIO): Transf Data a Nivel de Código
   - **Desventaja** : la CPU interrumpe Ejecución para Transmitir datos (MUCHO TIEMPO)

Solución

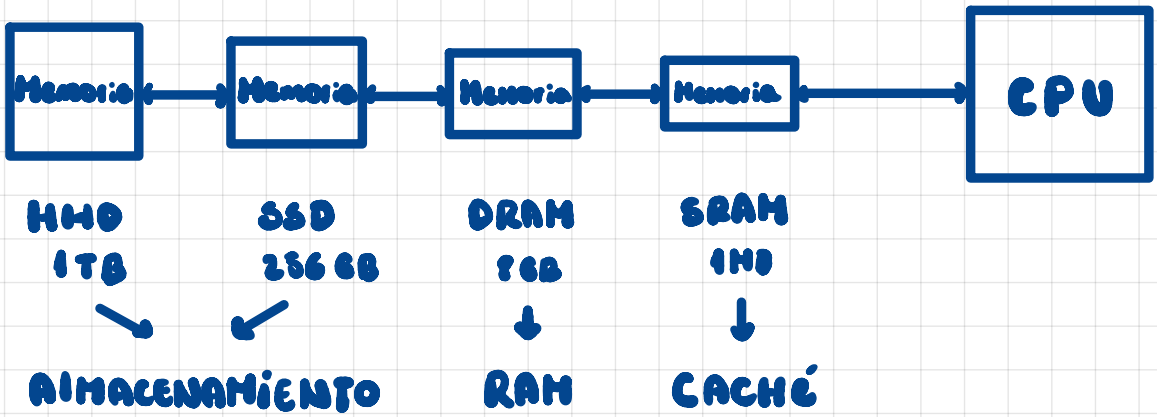
- **DMA** : (Direct Memory Access)



1. Encargado de la TRANSFERENCIA DE DATA
2. Ejecuta programa TRANSF → AVISA a la CPU cuando termina
   ↓
   por una INTERRUPCIÓN

2 MEMORIAS

1 JERARQUÍAS DE MEMORIA



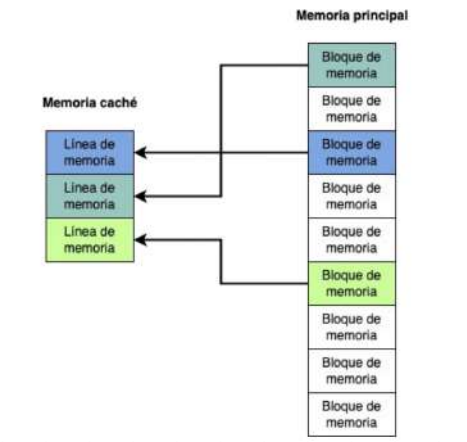
· EVALUACIÓN

1. **HIT** : Busqueda **EXITOSA** de un DATO en la CACHÉ
2. **MISS** : Busqueda **FALLIDA** de un DATO en la CACHÉ

3. **HIT-RATE** : #(hits) ÷ #(Accesos) ⟶ %
4. **MISS-RATE** : 1 - HR

5. **HIT-TIME** : Tiempo en buscar (encuentre o no)
6. **MISS-PENALTY** : Tiempo luego de un MISS, de COPIAR un bloque de MEMORIA siguiente en jerarquía y acceder al DATO Buscado.

- **MEMORIA CACHÉ**
  - 1er Nivel de Jerarquía y la CPU no sabe de su EXISTENCIA
  - Manejada por el **CONTROLADOR DE CACHÉ**
  - Se divide en:
    1. **LINEAS** (físicas) de Memoria
    2. **BLOQUES** (Lógica) de Memoria
    - Cada LINEA: **BIT DE VALIDEZ** indica si el contenido es valido
    - **Esquema**:



- ASIGNACIÓN de BLOQUES de memoria a las LINEAS de la CACHÉ

- **CONTROLADOR CACHÉ**:
  - **2 Funciones Principales**:

1. **MECANISMOS ACCESO A DATOS**: forma en que CONTROLADOR ASOCIA BLOQUES a lineas de CACHÉ
   - **FUNCIONES CORRESPONDENCIA**
   - **Politicas Reemplazo** (localidad temporal)

2. **POLITICAS ESCRITURA**: forma en que CONTROLADOR ACTUALIZA UN DATO EN LA MEMORIA PRINCIPAL si la CPU ESCRIBE

- **Funciones Correspondencia** (asocia un Bloque con una Linea)
  - Bloque → Memoria
  - Linea → Caché

- **TAG** : id "unico" determina si
  - el **DATO** de una direccion de la **MEM. Principal** se encuentre en la **CACHÉ**
  - Se obtiene con Bits de (Dir) Memoria

- **Datos importantes**

1. **Tamaño memoria** → nos da n° bits de DIRECCIÓN
   - para mapear bien : 2^{10} Bytes → 10 Bits !
2. **Cantidad lineas CACHE** → **id Lineas** (4 lineas) $\overset{\log_2(\ )}{=}$ 2 Bits
   - 8 lineas = 3 bits
3. **Tamaño lineas CACHE** → **OFFSET** : Bits menos signif.

- **3 Tipos de F. correspondencia**

1. **DIRECTLY MAPPED**
   - Cada **BLOQUE** se asocia a 1 sola **LÍNEA** de **CACHÉ**
   - Cada **LINEA** posee un **INDICE**

INDICE : n° bloques % n° Linea

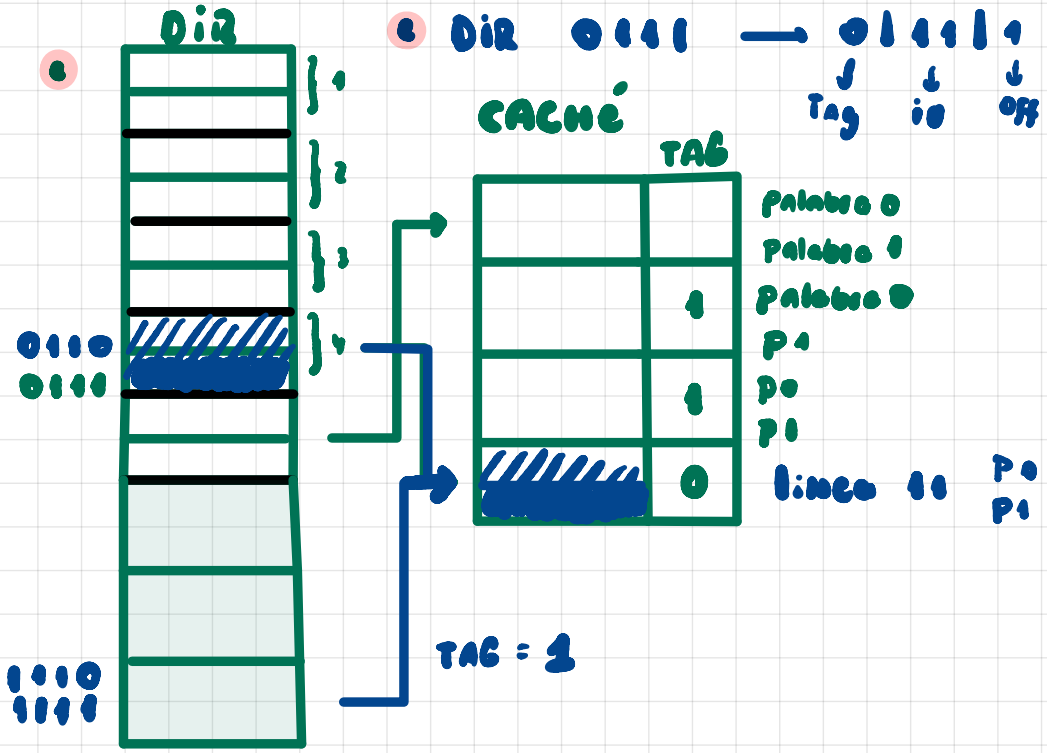
% → Modulo

- 7 bloques 4 lineas

0 → 0 ← 4
1 → 1 ← 5
2 → 2 ← 6
3 → 3

- **TAG | ID linea | offset**

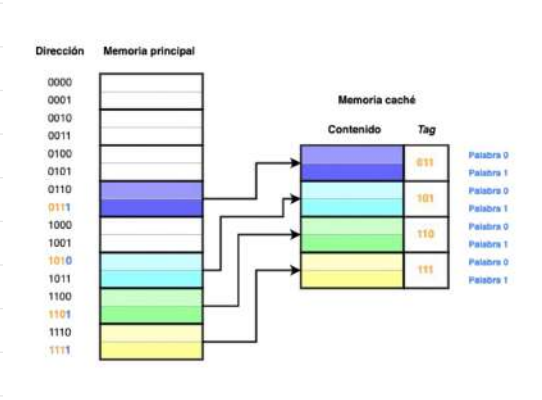
TAG → Sobrante
ID linea → linea cache
offset → Tamaño linea } cuantas divisiones



FULLY - ASOCIATIVE

- Un bloque se ASOCIA a cualquier LINEA CACHÉ
- BITS DIRECCIÓN

TAG | offset



dir 7 : 0 1 1 | 1

TAG → PUEDE ENCONTRARSE EN CUALQUIER LINEA

offset

DESVENTAJA: Recorrer toda la CACHÉ para buscar DATO

3 n-WAY ASSOCIATIVE

- La caché se agrupa en CONJ DE N LINEAS (NO N CONJ)
- UN Bloque se puede asociar a CUALQUIER LINEA (f. asociativa)
  - ↳ Dentro del CONJ (Directa)
- DIR DE BITS

TAG | ID CONJ | offset

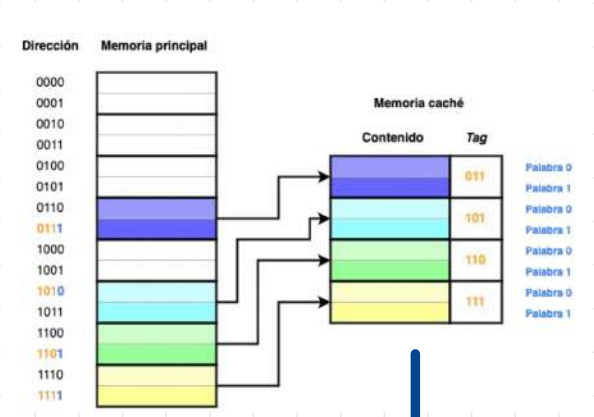
e.g. : 0111

- offset (Palabras en una línea)
- ID CONJ

n° CONJ = n° LINEAS ÷ N

- N way association

n° CONJ = 2 ⇒ 1 BIT



4 LINEAS ÷ 2 way

= 2 CONJ