



**DCC**  
DEPARTAMENTO DE CIENCIA  
DE LA COMPUTACIÓN

IIC2343

# Arquitectura de Computadores

Clase 13 - Resumen del Curso (I2)

Profesor: Germán Leandro Contreras Sagredo

## Propósito del material

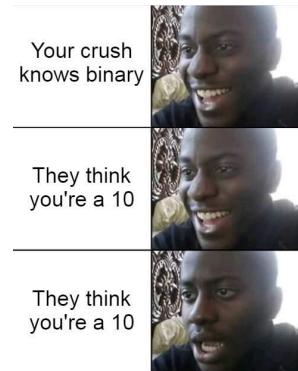
- El objetivo de este material es que puedan realizar un estudio **rápido** de los contenidos más relevantes del curso.
- La idea es que con este puedan identificar **qué contenidos les falta profundizar**, de ser el caso, se espera que no solo se queden con lo que se encuentra aquí, sino que además acudan a las clases y apuntes correspondientes.
- También se espera que sirva como guía rápida mientras estén desarrollando ejercicios de los compilados.



## Representaciones numéricas

- Contenido que encuentran en:

- **Clase 1 - Representaciones Numéricas I**
- **Clase 6 - Representaciones Numéricas II**
- **01 - Representaciones Numéricas Parte 1 - Números Enteros** (Apuntes)
- **02 - Representaciones Numéricas Parte 2 - Números Racionales** (Apuntes)



## Representaciones numéricas - Números enteros

- En este curso, nos enfocamos en la **representación posicional binaria** por su utilidad en circuitos digitales y, posteriormente, componentes del computador básico.
- **Fórmula general de transformación de bases**

$$\sum_{k=0}^{n-1} s_k \times b^k$$

$s$  = Símbolo (dígito).

$n$  = Cantidad de dígitos en la secuencia.

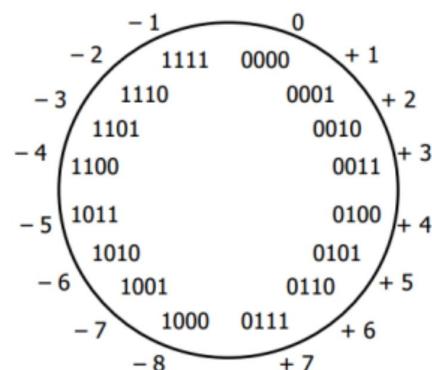
$b$  = Base numérica (o número de dígitos).

$k$  = Posición del dígito en la secuencia,  
siendo 0 la posición del extremo derecho.

## Representaciones numéricas - Números enteros

- **Complemento de 2:** Representación utilizada para números **enteros**. Se obtiene el complemento de 1 (complemento de cada dígito del número) y se suma una unidad al final. Esto asegura que  $x + C_2(x) = 0$  (el bit de *carry* restante se descarta).
- **Contras del complemento de 2**
  - Representación **desbalanceada** (un número negativo adicional).
  - **Overflow:** Operación cuyo resultado no es representable con la cantidad de bits disponible resultan en un valor erróneo.

$N$	0101
$-N$	1011
$N + (-N)$	0000



## Representaciones numéricicas - Números enteros

### ■ Conversión entre base binaria y hexadecimal

- **Hexadecimal a binario:** Cada dígito hexadecimal se representa en su valor binario con cuatro dígitos. Se concatenan los resultados para el valor final.

$$0x9F2 = \left\{ \begin{array}{l} 0x9 = 1001b \\ 0xF = 1111b \\ 0x2 = 0010b \end{array} \right\} \rightarrow 0x9F2 = 100111110010b$$

- **Binario a hexadecimal:** Se agrupan cuatro dígitos binarios y se representan en el valor hexadecimal. Se concatenan los resultados para el valor final.

$$100111110010b = (1001)(1111)(0010) = (0x9)(0xF)(0x2) = 0x9F2$$

\* Se puede hacer lo mismo entre la base binaria y la base octal, haciendo las conversiones con 3 dígitos en vez de 4.

## Representaciones numéricas - Números reales

- Se pueden representar en base binaria, pero **números finitos en base decimal pueden ser infinitos en base binaria**. Esto hace que la cantidad de bits para representar estos números sea significativa para reducir el error.
- **Representaciones de números reales en base binaria**
  - **Punto fijo:** Dados  $N$  bits, estos se dividen de forma **fija** para representar signo, parte entera y parte decimal.  
**Ej:**  $N = 8$  bits = 1 bit signo (s) + 3 bits parte entera (t) + 4 bits fracción (f)  
 $10,110_b = 0_s 010_t 0110_f$        $-1,0011_b = 1_s 001_t 0011_f$
  - **Punto flotante:** Dados  $N$  bits, los dividimos en un **significante/mantisa** y un **exponente**, ambos con signo.  
**Ej:**  $N = 8$  bits = 1 bit signo significante (ss) + 3 bits significante (s) + 1 bit signo exponente (se) + 3 bits exponente (e)  $\rightarrow -0,00011_b = -1,1 * 10^{-100} = 1_{ss} 110_s 1_{se} 100_e$

## Representaciones numéricas - Números reales

- Usamos punto flotante por un **mayor rango** (en perjuicio de la precisión).
  - Representación de punto flotante más utilizada: **Estándar IEEE754**
    - **Float (Single Precision Floating Point)**
      - 1 bit de signo.
      - 8 bits de exponente **desfasado** en 127 para obviar el bit de signo.
      - 23 bits de significante/mantisa **normalizado**: Tiene precisión de 24 bits ya que **asume que todo significante parte en 1**.
      - Exponente 00000000 reservado para representar el 0 (significante con solo bits de 0) y el exponente 11111111 para +-Infinity (significante con solo bits de 0) y NaN (significante con al menos un bit igual a 1).
      - $X = (-1)^{\text{signo}} * 1.\text{significante} * 10^{(\text{exponente} - 127)b}$

## Representaciones numéricas - Números reales

- Representación de punto flotante más utilizada: **Estándar IEEE754**
  - **Double (Double Precision Floating Point)**
    - 1 bit de signo.
    - 11 bits de exponente **desfasado** en 1023 para obviar el bit de signo.
    - 52 bits de significante/mantisa **normalizado**: Tiene precisión de 53 bits ya que **asume que todo significante parte en 1**.
    - Presenta un error de precisión mucho menor a float (error de float igual a  $10^{-23}$ ; error de double igual a  $10^{-52}$ ), pero requiere del doble de bits para ser representado.

# Representaciones numéricas - Números reales

## ■ Operaciones sobre números de punto flotante

### ○ Suma

1. Igualar exponentes. Se toma el número de menor exponente y este se incrementa hasta igualar al mayor. Al hacer esto, se mueve el punto decimal del significante.
2. Sumar los significantes.
3. Normalizar el resultado.
4. Pasar a formato float.

$$\begin{aligned}1,11 * 10^{11} + 1,11 * 10^1 \\= 1,11 * 10^{11} + 0,0111 * 10^{11} \\= (1,11 + 0,0111) * 10^{11} \\= 10,0011 * 10^{11} \\= 1,00011 * 10^{100}\end{aligned}$$

### ○ Multiplicación

1. Sumar exponentes **sin considerar el desfase dos veces**.
2. Multiplicar significantes. Se pueden multiplicar ignorando el punto decimal; luego, se vuelve a posicionar según la suma de las posiciones decimales de ambos operadores.
3. Normalizar el resultado.
4. Pasar a formato float.

$$\begin{aligned}1,11 * 10^{11} + 1,1 * 10^{11} \\= (1,11 * 1,1) * 10^{110} \\= 10,101 * 10^{110} \\= 1,0101 * 10^{111}\end{aligned}$$

## Representaciones numéricas - Números racionales

- Ejemplo de pérdida de precisión en floats:  $2^{30} + 5$

- Para sumarlos, igualamos la potencia del número menor a la del mayor:

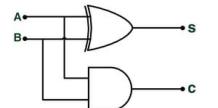
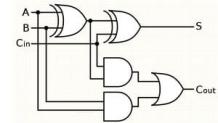
\* Estos bits se descartan porque la mantisa acepta hasta 23 bits.



## Operaciones aritméticas y lógicas

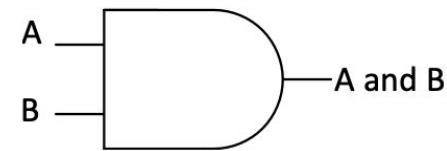
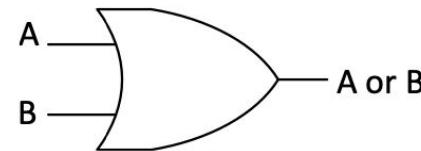
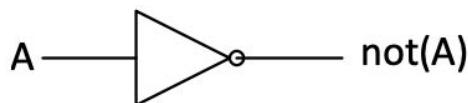
- Contenido que encuentran en:

- **Clase 2 - Operaciones Aritméticas y Lógicas**
- **03 - Operaciones Aritméticas y Lógicas**



## Operaciones aritméticas y lógicas - Compuertas lógicas

- **Compuertas lógicas:** Base de todos los componentes del computador básico: NOT, OR, AND.



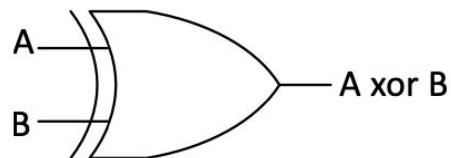
A	$\text{not}(A)$
1	0
0	1

A	B	$A \text{ or } B$
1	1	1
1	0	1
0	1	1
0	0	0

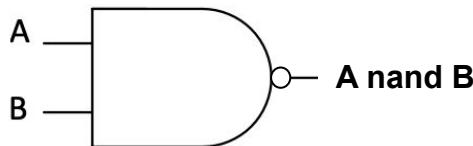
A	B	$A \text{ and } B$
1	1	1
1	0	0
0	1	0
0	0	0

## Operaciones aritméticas y lógicas - Compuertas lógicas

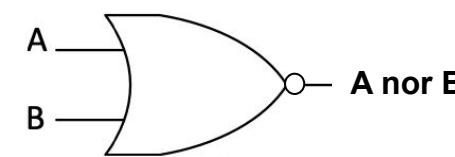
- **Compuertas lógicas:** Base de todos los componentes del computador básico: XOR, NAND, NOR, XNOR.



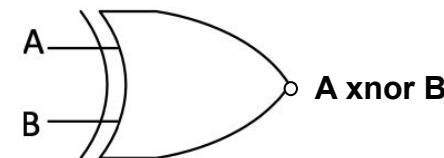
A	B	A xor B
1	1	0
1	0	1
0	1	1
0	0	0



A	B	A nand B
1	1	0
1	0	1
0	1	1
0	0	1



A	B	A nor B
1	1	0
1	0	0
0	1	0
0	0	1



A	B	A xnor B
1	1	1
1	0	0
0	1	0
0	0	1

## Operaciones aritméticas y lógicas - *Minterms* y *maxterms*

- Técnicas utilizadas para obtener la expresión lógica equivalente a una tabla de verdad.
- La expresión resultante se puede traducir directamente a un circuito eléctrico al utilizar solo conectivos lógicos AND, OR, NOT.
- Los *minterms* se basan en las condiciones (señales de entrada) que derivan en un valor de verdad igual a 1; los *maxterms* en las que derivan en un valor de verdad igual a 0. Conviene usar *minterms* cuando menos de la mitad de las filas de la tabla de verdad tienen *output* igual a 1; en otro caso, conviene usar *maxterms*. Si se tiene la misma cantidad de filas con *output* igual a 0 y 1, entonces se puede usar cualquiera de los dos.

## Operaciones aritméticas y lógicas - *Minterms* y *maxterms*

- ***Minterms*:** Solo se consideran las filas cuyo valor de verdad sea 1. Las condiciones de cada fila de la tabla se conectan con conectivos AND; las condiciones falsas (iguales a 0) se niegan. La expresión de cada fila se conecta con conectivos OR.

$(\text{NOT}(A) \text{ AND } B \text{ AND } \text{NOT}(C)) \text{ OR } (A \text{ AND } \text{NOT}(B) \text{ AND } C) \text{ OR } (A \text{ AND } B \text{ AND } C)$

≡

$$(\bar{A} \cdot B \cdot \bar{C}) + (A \cdot \bar{B} \cdot C) + (A \cdot B \cdot C)$$

A	B	C	Output
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

## Operaciones aritméticas y lógicas - *Minterms y maxterms*

- **Maxterms:** Solo se consideran las filas cuyo valor de verdad sea 0. Las condiciones de una fila de la tabla se conectan con conectivos OR; las condiciones verdaderas (iguales a 1) se niegan. La expresión de cada fila se conecta con conectivos AND.

$$(A \text{ OR } B \text{ OR } C) \text{ AND } (A \text{ OR } B \text{ OR } \text{NOT}(C)) \text{ AND } (A \text{ OR } \text{NOT}(B) \text{ OR } \text{NOT}(C)) \\ \text{AND } (\text{NOT}(A) \text{ OR } B \text{ OR } C) \text{ AND } (\text{NOT}(A) \text{ OR } \text{NOT}(B) \text{ OR } C)$$

≡

$$(A+B+C) \cdot (A+B+\bar{C}) \cdot (A+\bar{B}+\bar{C}) \cdot (\bar{A}+B+C) \cdot (\bar{A}+\bar{B}+C)$$

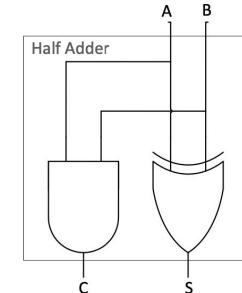
A	B	C	Output
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

\* También se puede construir usando el mismo formato de *minterms*, pero aplicando una negación al final y aplicando Ley De Morgan.

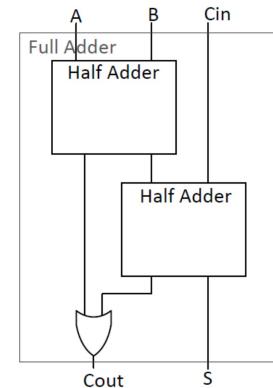
## Operaciones aritméticas y lógicas - Componentes aritméticas

- **Half-Adder:** “Medio sumador”, suma dos bits sin considerar bit de *carry*.

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



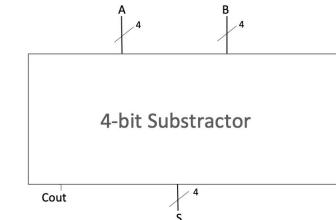
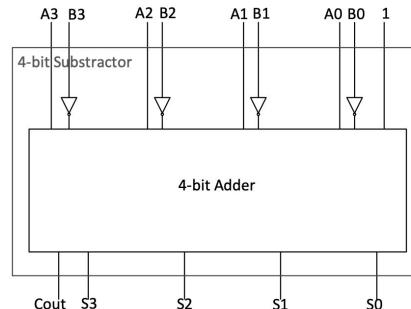
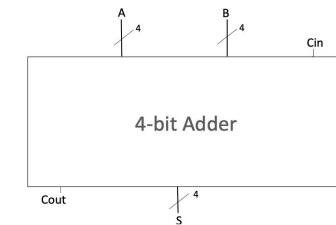
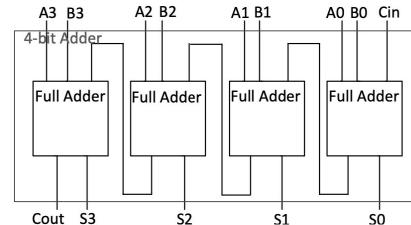
- **Full-Adder:** Sumador completo, suma dos bits y considera bit de *carry* (señal  $C_{in}$ ).



\* El *carry* de salida puede ser generado con la suma de  $A$  y  $B$  o con la suma entre dicho resultado y el *carry* de entrada, razón por la que  $C_{out}$  es la compuerta OR de estos dos resultados

## Operaciones aritméticas y lógicas - Componentes aritméticas

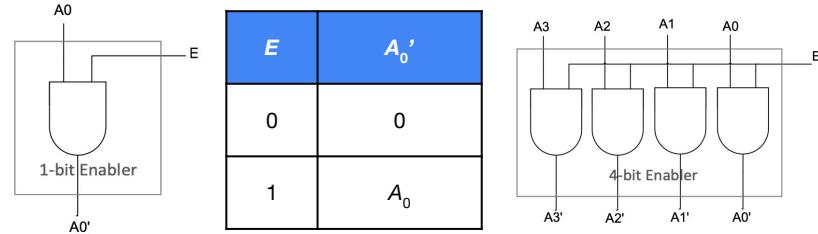
- **Sumador de 4 bits:** *Full-Adders* conectados a través de la señal  $C_{in}$ . Se extiende a más bits de la misma forma.
- **Restador de 4 bits:** Sumador de 4 bits pero con entrada *B* transformada a su complemento de dos.



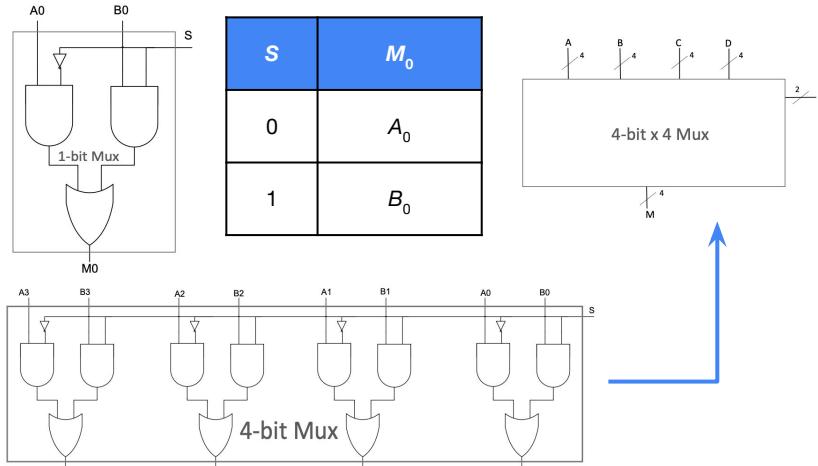
\* Notación de buses utilizada para facilitar la abstracción de la cantidad de señales.

## Operaciones aritméticas y lógicas - Componentes aritméticas

- **Enabler:** Componente que habilita o no el paso de una señal.



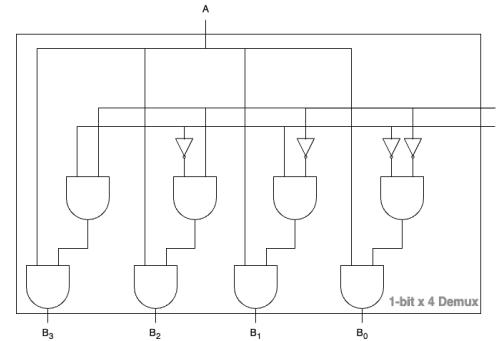
- **Multiplexor/Mux:** Componente que selecciona como salida una de un conjunto de señales de entrada.



## Operaciones aritméticas y lógicas - Componentes aritméticas

### ■ De-Multiplexor/Demux:

Componente que transmite una señal de entrada a una de múltiples salidas. En el resto de salidas no seleccionadas se transmite un 0.

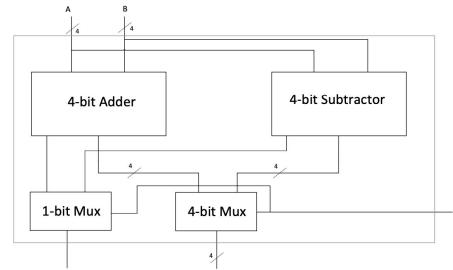


<i>S<sub>1</sub></i>	<i>S<sub>0</sub></i>	<i>Output</i>
0	0	$B_0 = A$
1	0	$B_1 = A$
0	1	$B_2 = A$
1	1	$B_3 = A$

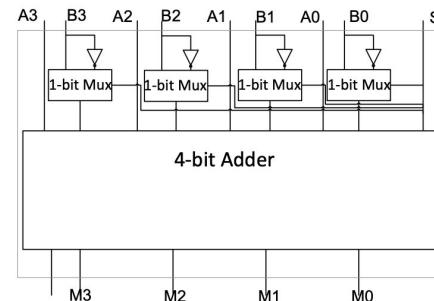
# Operaciones aritméticas y lógicas - Componentes aritméticas

## ■ Sumador-Restador

- No optimizado.
- Optimizado.



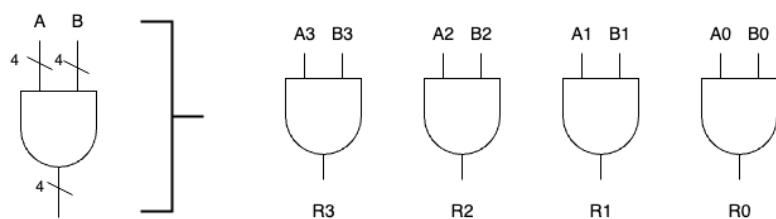
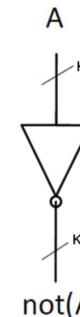
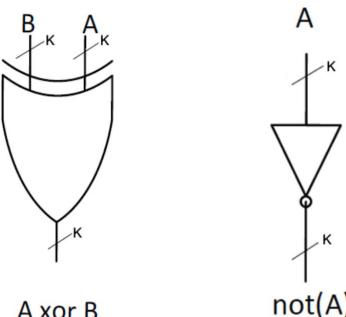
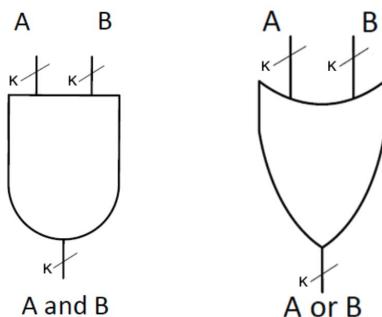
S	M	C <sub>out</sub>
0	A+B	A+B C <sub>out</sub>
1	A-B	A-B C <sub>out</sub>



S	M	Cout
0	A+B	A+B C <sub>out</sub>
1	A-B	A-B C <sub>out</sub>

## Operaciones aritméticas y lógicas - Componentes aritméticas

- **Operadores *bitwise*:** Componentes que extienden las operaciones de compuertas lógicas a señales de más de 1 bit.



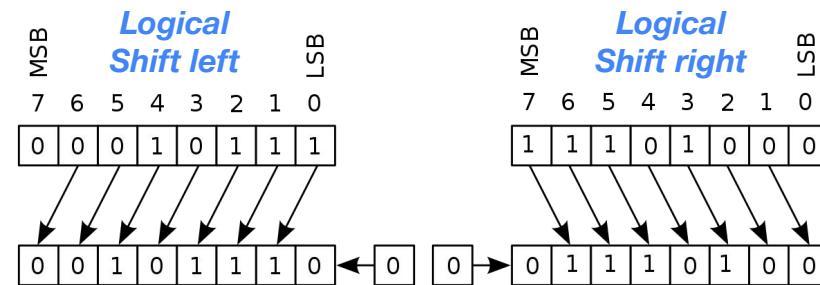
$$R = (A_3 \text{ AND } B_3)(A_2 \text{ AND } B_2)(A_1 \text{ AND } B_1)(A_0 \text{ AND } B_0) = R_3R_2R_1R_0$$

\* Ejemplo de construcción de componente *bitwise AND*.

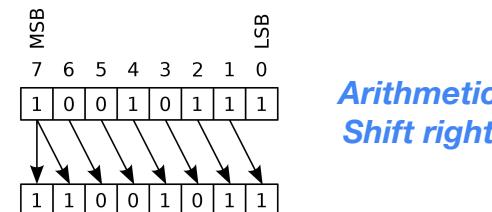
## Operaciones aritméticas y lógicas - Componentes aritméticas

- **Logical shifting components:** Componentes que desplazan los bits de un número a la derecha o a la izquierda, equivalente a dividir o multiplicar por 2 de forma respectiva.

- shift\_left(0100b) = 1000b
- shift\_left(1001b) = 0010b
- shift\_right(0100b) = 0010b
- shift\_right(1001b) = 0100b



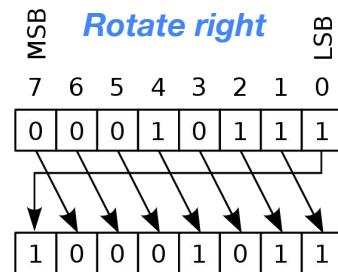
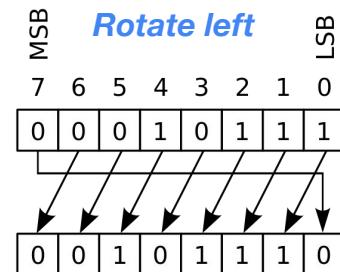
- **Arithmetic shift right:** Componente que realiza *shift right* manteniendo el signo (bit más significativo).
- shift\_arithmetic\_right(1100b) = 1110b
  - shift\_arithmetic\_right(1001b) = 1100b



## Operaciones aritméticas y lógicas - Componentes aritméticas

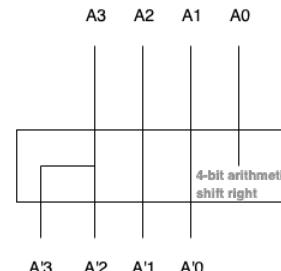
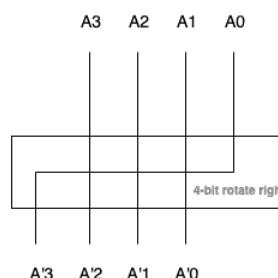
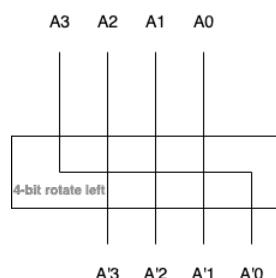
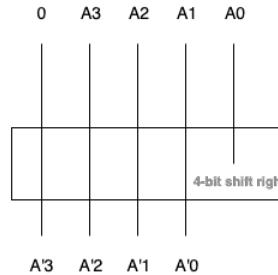
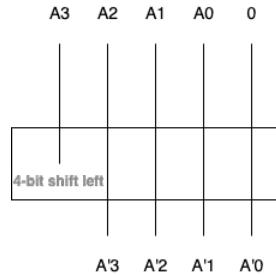
- ***Rotating components:*** Componentes que desplazan los bits de un número a la derecha o a la izquierda, pero que rotan el bit “descartado” al bit más o menos significativo respectivamente.

- $\text{rotate\_left}(1000\text{b}) = 0001\text{b}$
- $\text{rotate\_left}(0101\text{b}) = 1010\text{b}$
- $\text{rotate\_right}(0100\text{b}) = 0010\text{b}$
- $\text{rotate\_right}(0011\text{b}) = 1001\text{b}$



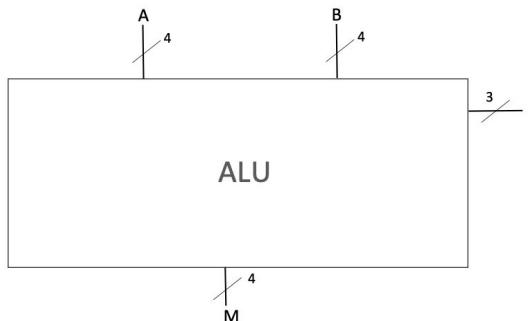
# Operaciones aritméticas y lógicas - Componentes aritméticas

## ■ Diagramas de componentes de *shift* y *rotate*



## Operaciones aritméticas y lógicas - Componentes aritméticas

- **Arithmetic Logic Unit (ALU):** Realiza las operaciones aritmético-lógicas con los componentes antes vistos y selecciona la operación mediante multiplexores (que reciben el resultado de todas las operaciones). Será la **unidad de ejecución del computador básico**.

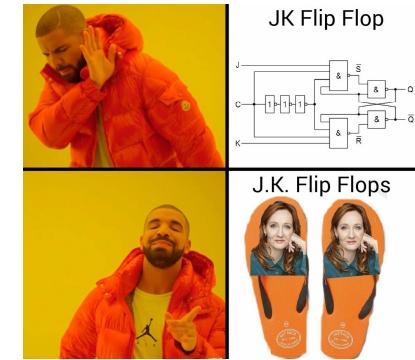


S2	S1	S0	M
0	0	0	Suma
0	0	1	Resta
0	1	0	And
0	1	1	Or
1	0	0	Not
1	0	1	Xor
1	1	0	Shift left
1	1	1	Shift right

\* Solo realiza *shifts* lógicos, no aritméticos.

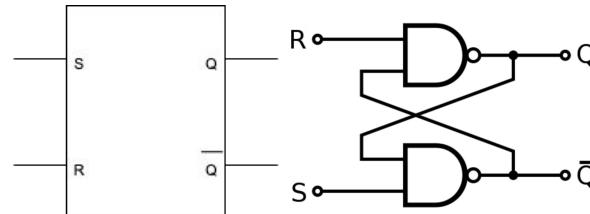
## Almacenamiento de datos

- Contenido que encuentran en:
  - **Clase 3 - Almacenamiento de Datos**
  - **04 - Almacenamiento de Datos (Apuntes)**



## Almacenamiento de datos - Latches

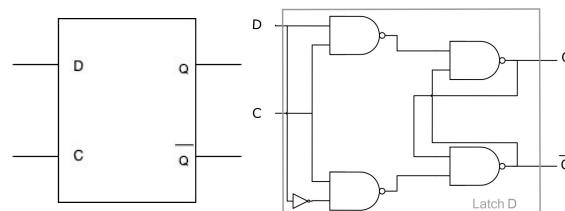
- **Latch RS:** Componente que puede almacenar un estado o cambiarlo mediante las señales  $R$ (eset) y  $S$ (et) a través de un **circuito secuencial**.



<b>S</b>	<b>R</b>	<b>Q(t+1)</b>
0	0	-
0	1	0
1	0	1
1	1	$Q(t)$

\* Notar el uso de compuertas NAND.

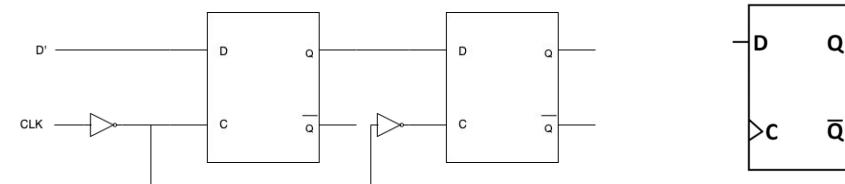
- **Latch D:** Componente que se construye sobre un Latch RS para permitir el almacenamiento de una señal  $D$  a partir de una señal de control  $C$ .



<b>C</b>	<b>D</b>	<b>Q(t+1)</b>
0	0	$Q(t)$
0	1	$Q(t)$
1	0	0
1	1	1

## Almacenamiento de datos - Flip-flops

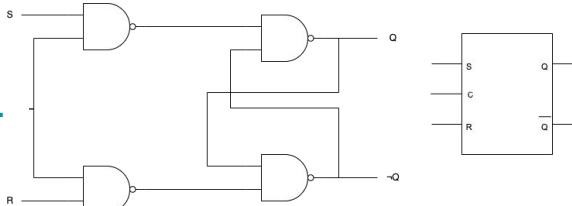
- **Flip-flop D:** Componente que permite guardar el estado anterior de una señal **en un instante dado**.



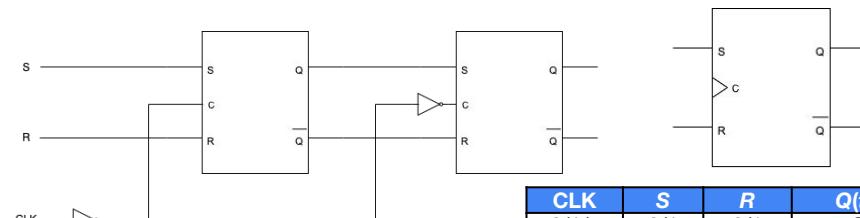
$\uparrow$  = Flanco de subida (CLK de 0 a 1).  
 $\downarrow$  = Flanco de bajada (CLK de 1 a 0).

CLK	D	$Q(t+1)$
0/1/↓	0/1	$Q(t)$
↑	0	0
↑	1	1

- **Flip-flop RS:** Flip-flop construido con latches RS que incluyen señal de control C.



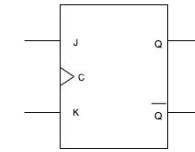
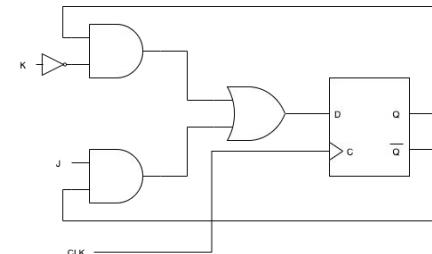
\* Latch RS con señal de control C.



CLK	S	R	$Q(t+1)$
0/1/↓	0/1	0/1	$Q(t)$
↑	0	0	$Q(t)$
↑	0	1	0
↑	1	0	1
↑	1	1	-

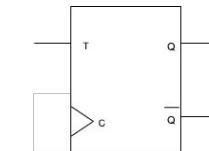
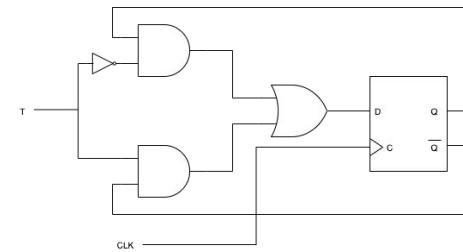
## Almacenamiento de datos - Flip-flops

- **Flip-flop JK:** Similar al flip-flop RS, pero sin estado inválido.



CLK	J	K	$Q(t+1)$
0/1/↓	0/1	0/1	$Q(t)$
↑	0	0	$Q(t)$
↑	0	1	0
↑	1	0	1
↑	1	1	$\neg Q(t)$

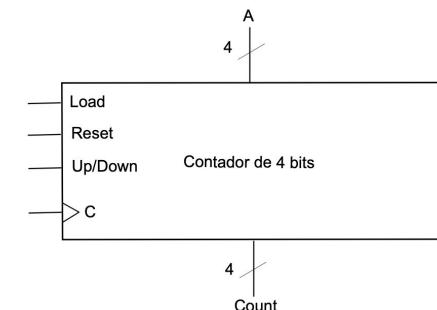
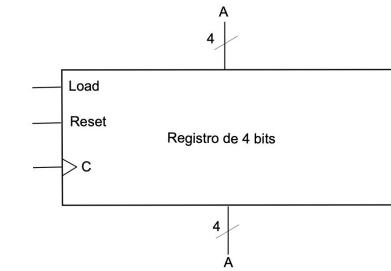
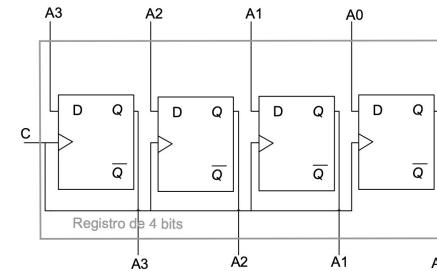
- **Flip-flop T:** Flip-flop que a partir de una señal  $T(\text{oggle})$  invierte el estado.



CLK	T	$Q(t+1)$
0/1/↓	0/1	$Q(t)$
↑	0	$Q(t)$
↑	1	$\neg Q(t)$

## Almacenamiento de datos - Registros y contadores

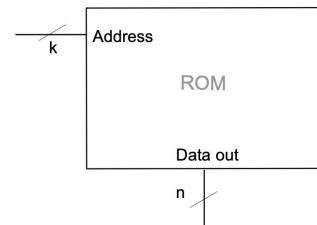
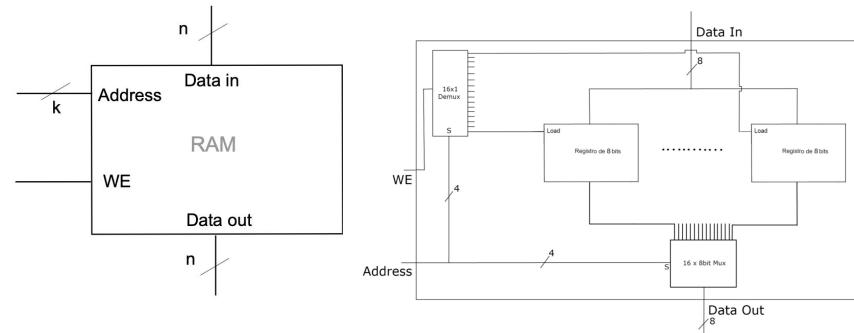
- **Registro:** Conjunto de flip-flops que almacenan cada bit de un valor numérico. Posee señales *Reset* para dejar el valor en 0 y *Load* para cargar una señal de entrada.
- **Contador:** Registro con señales de incremento/decremento para aumentar o disminuir en una unidad el valor almacenado.



\* De ahora en adelante, todo componente que en su interior posea flip-flops debe estar conectado al *clock* del sistema para asegurar sincronización.

## Almacenamiento de datos - Memorias

- **RAM:** Extensión de los registros para leer o escribir **palabras de memoria** (generalmente de 8 bits - 1 byte). Si se poseen  $k$  bits de direccionamiento, la memoria posee  $2^k$  palabras de memorias ( $2^k$  bytes).
- **ROM:** Memoria de “solo lectura” (dependiendo del caso, pero pueden asumirla así).



\* El uso del Demux es fundamental para el funcionamiento de la memoria RAM y la propagación de la señal WE (Write Enable) solo al registro que corresponda.

## Almacenamiento de datos - Tipos de dato

- **Variables:** Valor que puede cambiar durante la ejecución de un programa y cuyo tipo de dato incide en la forma en que se almacena.

Tipo de dato	Codificación	Interpretación	#Bits de representación
char	base 2 sin signo	carácter o entero positivo	8
signed char	base 2 con signo en complemento de 2	entero positivo o negativo	8
short	base 2 con signo en complemento de 2	entero positivo o negativo	16
unsigned short	base 2 sin signo	entero positivo	16
int	base 2 con signo en complemento de 2	entero positivo o negativo	32
unsigned int	base 2 sin signo	entero positivo	32
long	base 2 con signo en complemento de 2	entero positivo o negativo	64
unsigned long	base 2 sin signo	entero positivo	64
long long	base 2 con signo en complemento de 2	entero positivo o negativo	128
unsigned long long	base 2 sin signo	entero positivo	128
float	punto flotante de precisión simple	Racionales y casos especiales	32
double	punto flotante de precisión doble	Racionales y casos especiales	64
long double	punto flotante de precisión cuádruple	Racionales y casos especiales	128

## Almacenamiento de datos - *Endianness*

- ***Endianness***: Orden en el que se almacenan la secuencia de un dato.
  - ***BigEndian***: La palabra más significativa de la secuencia se almacena en la dirección **menor**.
  - ***LittleEndian***: La palabra menos significativa de la secuencia se almacena en la dirección **menor**.

\* Ejemplos de *endianness* para el almacenamiento del dato 0000111101010101, que se separa en los bytes 00001111 y 01010101

Dirección de memoria (hexa)	Palabra almacenada ( <i>big endian</i> )
0x00	00001111
0x01	01010101
0x02	-

Dirección de memoria (hexa)	Palabra almacenada ( <i>little endian</i> )
0x00	01010101
0x01	00001111
0x02	-

# Almacenamiento de datos - Arreglos y matrices

- **Arreglo:** Tipo de dato que además posee un **largo** y una **dirección de memoria de inicio**.
- **Matriz:** Arreglo de arreglos. Se puede almacenar según la convención de **filas** o **columnas**.

```
let variables: number[] = [1, 3, 5, 7];
```

Dirección de memoria (hexa)	Palabra almacenada
0x00	00000001
0x01	00000011
0x02	00000101
0x03	00000111
0x04	-

```
let variablesMatrix: number[][] = [
  [1, 3, 5],
  [2, 4, 6]
];
```

Dirección de memoria (hexa)	Palabra almacenada	Dirección de memoria (hexa)	Palabra almacenada
0x00	00000001	0x00	00000001
0x01	00000011	0x01	00000010
0x02	00000101	0x02	00000011
0x03	00000100	0x03	00000100
0x04	00000100	0x04	00000101
0x05	00000110	0x05	00000110

**Direccionamiento en convención por filas:**  $dir(\text{matriz}[i,j]) = dir(\text{matriz}) + i * sizeof(\text{matriz}[i,j]) * \#columnas + j * sizeof(\text{matriz}[i,j])$

**Direccionamiento en convención por columnas:**  $dir(\text{matriz}[i,j]) = dir(\text{matriz}) + j * sizeof(\text{matriz}[i,j]) * \#filas + i * sizeof(\text{matriz}[i,j])$

## Computador básico

### ■ Contenido que encuentran en:

- **Clase 4 - Programabilidad**
- **Clase 5 - Saltos y Subrutinas**
- **Clase 7 - Arquitecturas de Computadores**
- **05 - Programabilidad** (Apuntes)
- **06 - Saltos y Subrutinas** (Apuntes)
- **07 - Arquitecturas de Computadores** (Apuntes)

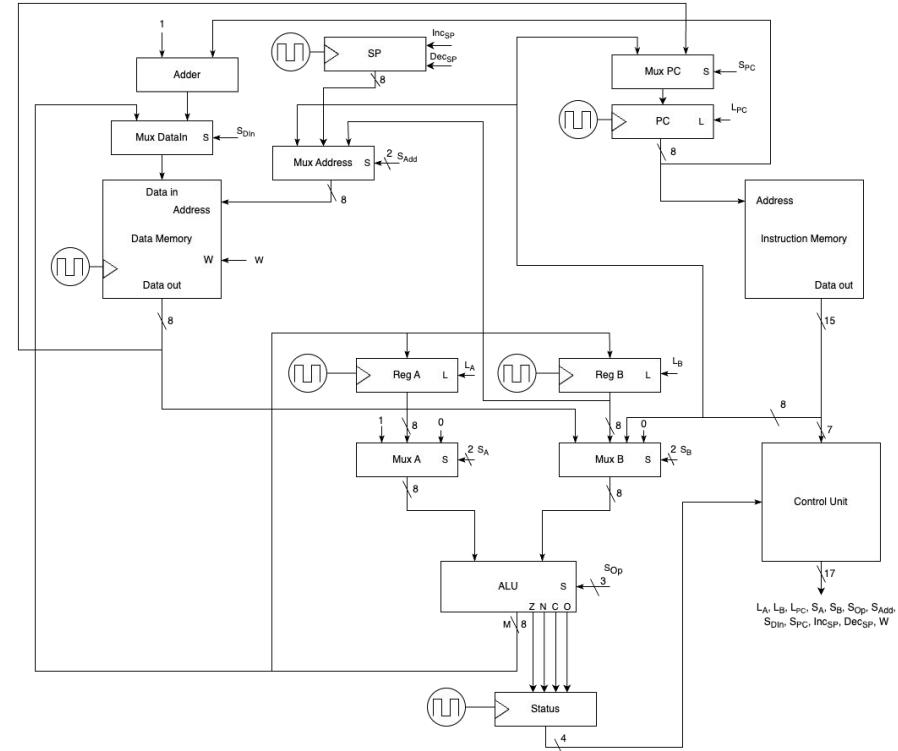
When your friend only writes programs in assembly language



# Computador básico - Microarquitectura

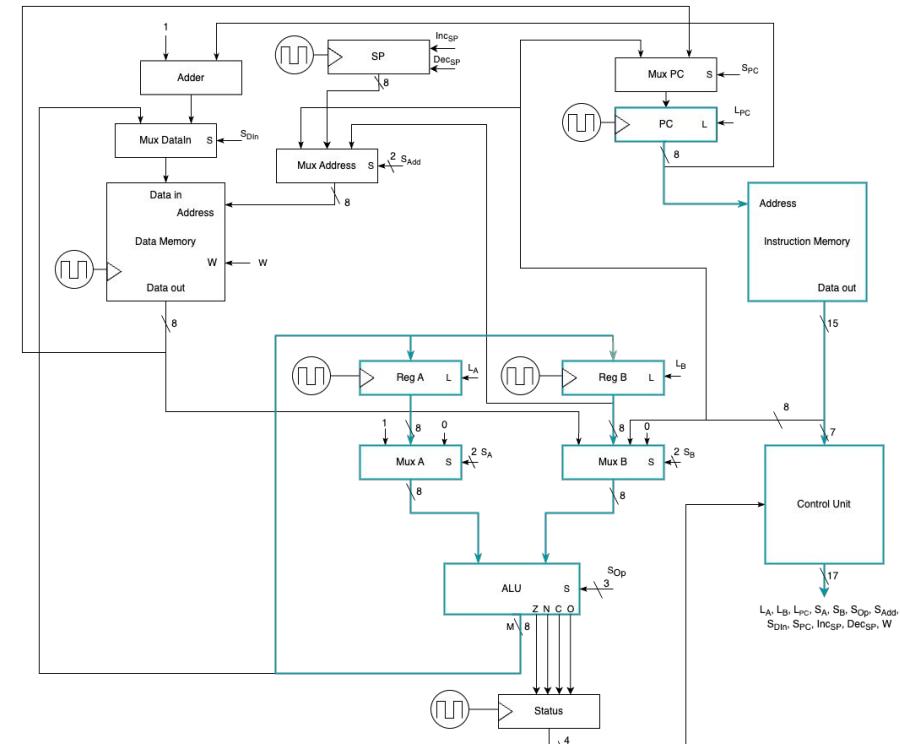
## ■ Diagrama del computador básico

Diagrama completo. Iremos destacando las conexiones y componentes relevantes que habilitan cada funcionalidad.



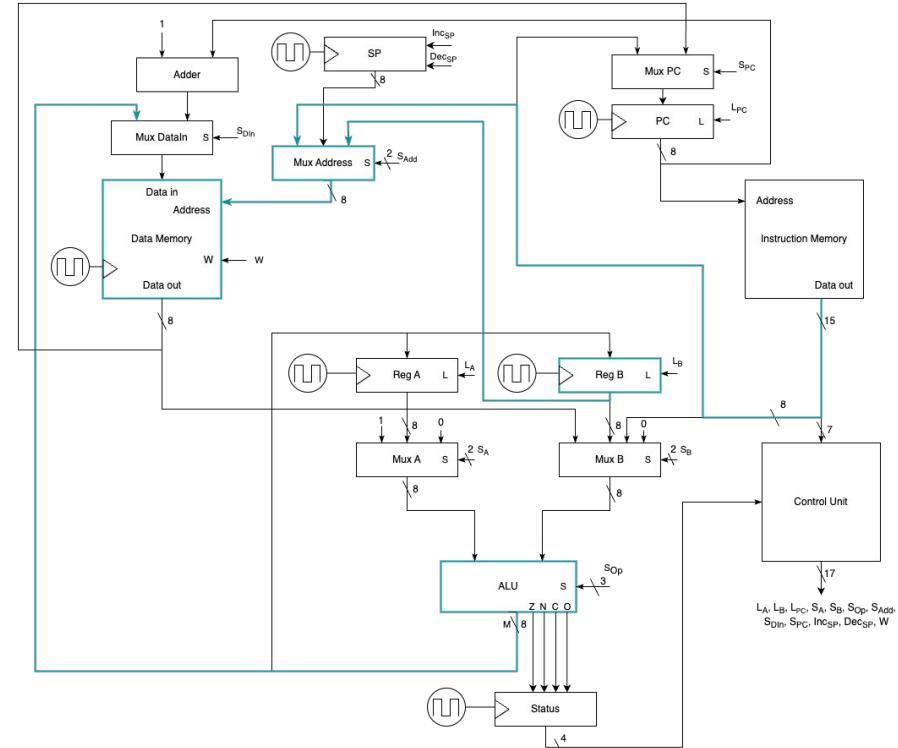
# Computador básico - Microarquitectura

- Contador PC (*Program Counter*) permite ejecutar de forma secuencial un programa almacenado en la memoria ROM *Instruction Memory*. Por cada flanco de subida incrementa su valor y, así, cambia la instrucción en ejecución.
- *Control Unit* decodifica el *opcode* de la instrucción en señales de control que ejecutan la operación deseada.
- Registros A y B permiten realizar operaciones en la ALU y almacenar el resultado temporalmente. Muxes A y B permiten ampliar las operaciones a realizar.



## Computador básico - Microarquitectura

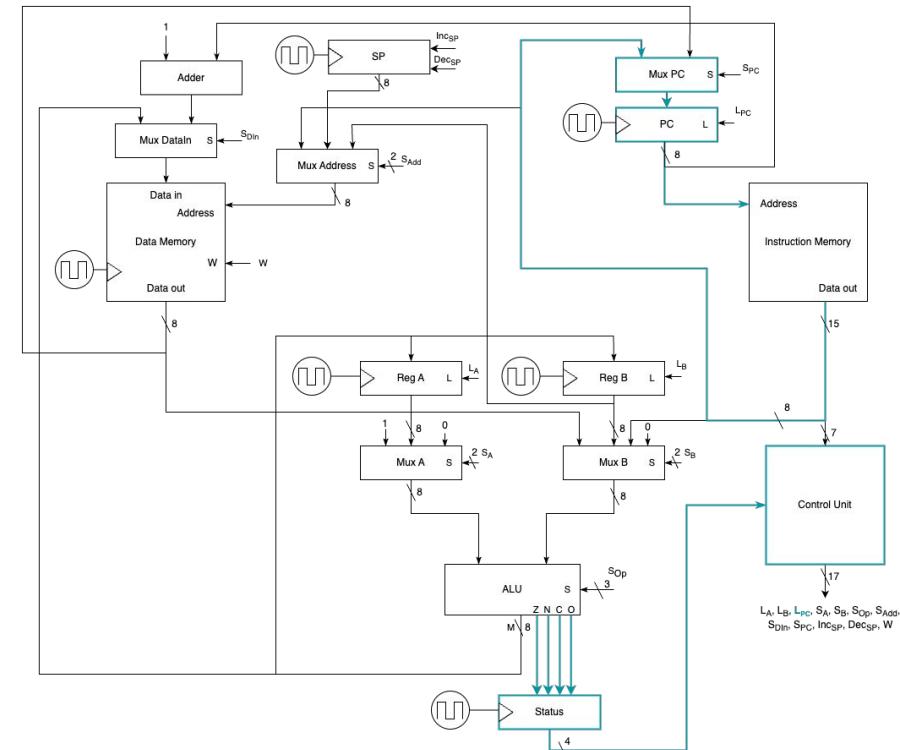
- Valor literal (numérico) asociado a la instrucción puede ser utilizado para realizar más operaciones a través del Mux *B*.
- Los resultados de las operaciones se pueden almacenar en la memoria RAM *Data Memory* (donde también se almacenan las variables del programa).
- La dirección donde se lee o escribe un dato puede ser dada por el literal mismo (direcciónamiento directo) o por el valor del registro *B* (direcciónamiento indirecto).



## Computador básico - Microarquitectura

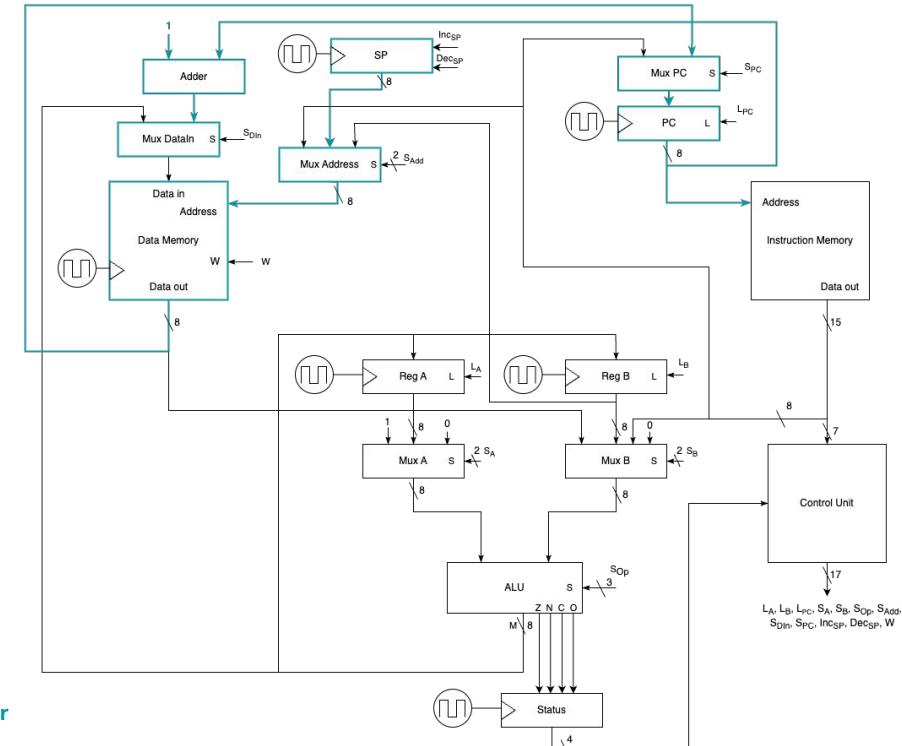
- La conexión entre el literal y el contador PC (a través de su Mux) permite la ejecución de **instrucciones de salto**, cargando en el contador la dirección de memoria de la instrucción deseada.
- Si se quiere realizar un salto **condicional**, se hace uso de las *flags* de estado que otorgan información de la instrucción anterior ( $Z =$ cero;  $N =$ negativo;  $C =$ carry;  $O =$ overflow). Si se ejecuta una instrucción de salto que cumple la condición, la *Control Unit* habilita la señal  $L_{PC}$ .

\* Usamos la instrucción CMP antes del salto para tener certeza del estado esperado, pero en estricto rigor se pueden realizar saltos condicionales siempre.



# Computador básico - Microarquitectura

- Contador SP (*Stack Pointer*) permite añadir **memoria de stack** (ingreso de elementos desde la última dirección de la memoria).
- A través de la memoria de stack podemos implementar **subrutinas**. La conexión del PC y la *Data Memory* permite almacenar la dirección de retorno ( $PC+1$ ) en el *stack* y cargarla de vuelta a través de la conexión entre la salida de la *Data Memory* y PC a través de su Mux.



\* Por construcción, SP siempre apunta una dirección sobre el tope. Por ende, para leer el valor del tope es necesario invertir una iteración en decrementarlo en una unidad.

## Computador básico - ISA

- ***Instruction Set Architecture:*** Indica la **forma** en la que se escriben los programas de una arquitectura y las operaciones que soporta, indicando a su vez las señales requeridas para ello, su formato, etc.

Instrucción	Operandos	Opcode	Condición	L <sub>PC</sub>	L <sub>A</sub>	L <sub>B</sub>	S <sub>A0,1</sub>	S <sub>B0,1</sub>	S <sub>Op0,1,2</sub>	S <sub>Add0,1</sub>	S <sub>DIn</sub>	S <sub>PC</sub>	W	Inc <sub>SP</sub>	Dec <sub>SP</sub>
MOV	A,B	0000000		0	1	0	ZERO	B	ADD	-	-	-	0	0	0
	B,A	0000001		0	0	1	A	ZERO	ADD	-	-	-	0	0	0
	A,Lit	0000010		0	1	0	ZERO	LIT	ADD	-	-	-	0	0	0
	B,Lit	0000011		0	0	1	ZERO	LIT	ADD	-	-	-	0	0	0
	A,(Dir)	0000100		0	1	0	ZERO	DOUT	ADD	LIT	-	-	0	0	0
	B,(Dir)	0000101		0	0	1	ZERO	DOUT	ADD	LIT	-	-	0	0	0
	(Dir),A	0000110		0	0	0	A	ZERO	ADD	LIT	ALU	-	1	0	0
	(Dir),B	0000111		0	0	0	ZERO	B	ADD	LIT	ALU	-	1	0	0
	A,(B)	0001000		0	1	0	ZERO	DOUT	ADD	B	-	-	0	0	0
	B,(B)	0001001		0	0	1	ZERO	DOUT	ADD	B	-	-	0	0	0
	(B),A	0001010		0	1	0	A	ZERO	ADD	B	ALU	-	1	0	0

Parte 1 de la ISA del computador básico

# Computador básico - ISA

ADD	A,B	0001011	0	1	0	A	B	ADD	-	-	-	0	0	0
	B,A	0001100	0	0	1	A	B	ADD	-	-	-	0	0	0
	A,Lit	0001101	0	1	0	A	LIT	ADD	-	-	-	0	0	0
	A,(Dir)	0001110	0	1	0	A	DOUT	ADD	LIT	-	-	0	0	0
	A,(B)	0001111	0	1	0	A	DOUT	ADD	B	-	-	0	0	0
	(Dir)	0010000	0	0	0	A	B	ADD	LIT	ALU	-	1	0	0
SUB	A,B	0010001	0	1	0	A	B	SUB	-	-	-	0	0	0
	B,A	0010010	0	0	1	A	B	SUB	-	-	-	0	0	0
	A,Lit	0010011	0	1	0	A	LIT	SUB	-	-	-	0	0	0
	A,(Dir)	0010100	0	1	0	A	DOUT	SUB	LIT	-	-	0	0	0
	A,(B)	0010101	0	1	0	A	DOUT	SUB	B	-	-	0	0	0
	(Dir)	0010110	0	0	0	A	B	SUB	LIT	ALU	-	1	0	0
AND	A,B	0010111	0	1	0	A	B	AND	-	-	-	0	0	0
	B,A	0011000	0	0	1	A	B	AND	-	-	-	0	0	0
	A,Lit	0011001	0	1	0	A	LIT	AND	-	-	-	0	0	0
	A,(Dir)	0011010	0	1	0	A	DOUT	AND	LIT	-	-	0	0	0
	A,(B)	0011011	0	1	0	A	DOUT	AND	B	-	-	0	0	0
	(Dir)	0011100	0	0	0	A	B	AND	LIT	ALU	-	1	0	0
OR	A,B	0011101	0	1	0	A	B	OR	-	-	-	0	0	0
	B,A	0011110	0	0	1	A	B	OR	-	-	-	0	0	0
	A,Lit	0011111	0	1	0	A	LIT	OR	-	-	-	0	0	0
	A,(Dir)	0100000	0	1	0	A	DOUT	OR	LIT	-	-	0	0	0
	A,(B)	0100001	0	1	0	A	DOUT	OR	B	-	-	0	0	0
	(Dir)	0100010	0	0	0	A	B	OR	LIT	ALU	-	1	0	0

Parte 2 de la ISA del computador básico

# Computador básico - ISA

NOT	A,A	0100011	0	1	0	A	-	NOT	-	-	-	0	0	0
	B,A	0100100	0	0	1	A	-	NOT	-	-	-	0	0	0
	(Dir)	0100101	0	0	0	A	B	NOT	LIT	ALU	-	1	0	0
XOR	A,B	0100110	0	1	0	A	B	XOR	-	-	-	0	0	0
	B,A	0100111	0	0	1	A	B	XOR	-	-	-	0	0	0
	A,Lit	0101000	0	1	0	A	LIT	XOR	-	-	-	0	0	0
	A,(Dir)	0101001	0	1	0	A	DOUT	XOR	LIT	-	-	0	0	0
	A,(B)	0101010	0	1	0	A	DOUT	XOR	B	-	-	0	0	0
	(Dir)	0101011	0	0	0	A	B	XOR	LIT	ALU	-	1	0	0
SHL	A,A	0101100	0	1	0	A	-	SHL	-	-	-	0	0	0
	B,A	0101101	0	0	1	A	-	SHL	-	-	-	0	0	0
	(Dir)	0101110	0	0	0	A	B	SHL	LIT	ALU	-	1	0	0
SHR	A,A	0101111	0	1	0	A	-	SHR	-	-	-	0	0	0
	B,A	0110000	0	0	1	A	-	SHR	-	-	-	0	0	0
	(Dir)	0110001	0	0	0	A	B	SHR	LIT	ALU	-	1	0	0
INC	B	0110010	0	0	1	ONE	B	ADD	-	-	-	0	0	0
	(B)	0110011	0	0	1	ONE	DOUT	ADD	B	ALU	-	1	0	0
	(Dir)	0110100	0	0	1	ONE	DOUT	ADD	LIT	ALU	-	1	0	0

Parte 3 de la ISA del computador básico

# Computador básico - ISA

CMP	A,B	0110101	0 0 0	A	B	SUB	-	-	-	0	0	0
	A,Lit	0110110	0 0 0	A	LIT	SUB	-	-	-	0	0	0
	A,(Dir)	0110111	0 0 0	A	DOUT	SUB	LIT	-	-	0	0	0
	A,(B)	0111000	0 0 0	A	DOUT	SUB	B	-	-	0	0	0
JMP	Dir	0111001	1 0 0	-	-	-	-	-	LIT	0	0	0
JEQ	Dir	0111010	Z=1	1 0 0	-	-	-	-	LIT	0	0	0
JNE	Dir	0111011	Z=0	1 0 0	-	-	-	-	LIT	0	0	0
JGT	Dir	0111100	N=0 y Z=0	1 0 0	-	-	-	-	LIT	0	0	0
JLT	Dir	0111101	N=1	1 0 0	-	-	-	-	LIT	0	0	0
JGE	Dir	0111110	N=0	1 0 0	-	-	-	-	LIT	0	0	0
JLE	Dir	0111111	N=1 o Z=1	1 0 0	-	-	-	-	LIT	0	0	0
JCR	Dir	1000000	C=1	1 0 0	-	-	-	-	LIT	0	0	0
JOV	Dir	1000001	V=1	1 0 0	-	-	-	-	LIT	0	0	0
CALL	Dir	1000010	1 0 0	-	-	-	SP	PC	LIT	1	0	1
RET		1000011	0 0 0	-	-	-	-	-	-	0	1	0
		1000100	1 0 0	-	-	-	SP	-	DOUT	0	0	0
PUSH	A	1000101	0 0 0	A	ZERO	ADD	SP	ALU	-	1	0	1
	B	1000110	0 0 0	ZERO	B	ADD	SP	ALU	-	1	0	1
POP	A	1000011	0 0 0	-	-	-	-	-	-	0	1	0
		1000111	0 1 0	ZERO	DOUT	ADD	SP	-	-	0	0	0
POP	B	1000011	0 0 0	-	-	-	-	-	-	0	1	0
		1001000	0 0 1	ZERO	DOUT	ADD	SP	-	-	0	0	0
NOP		1001001	0 0 0	-	-	-	-	-	-	0	0	0

Parte 4 de la ISA del computador básico

## Computador básico - ISA

- Ejemplo de código con la ISA del computador básico. Cabe destacar que se definen dos segmentos: DATA para las variables (incluyendo arreglos) y CODE para las instrucciones.
- El **Assembler** es el programa encargado de transformar este código en lenguaje de máquina (binario) y de asegurar que existan instrucciones que almacenen las variables en memoria, así como también *mapear* correctamente las direcciones de los *labels* a los literales correctos.

DATA:

```
var 1  
arr 1  
      3  
      5
```

CODE:

```
MOV B,arr  
MOV A,(B)  
SHL A,A  
MOV (B),A  
INC B  
MOV A,(B)  
SHL A,A  
MOV (B),A  
INC B  
MOV A,(B)  
SHL A,A  
MOV (B),A
```

# Computador básico - Clasificaciones

## ■ Paradigmas según ISA

- **RISC:** *Reduced Instruction Set Computer.* Instrucciones pequeñas y simples. Su diseño permite simplificar el *hardware*, poniendo énfasis en el *software*.
- **CISC:** *Complex Instruction Set Computer.* Muchas instrucciones y con complejidad alta. Énfasis en un *hardware* más complejo para poder ejecutarlas.

## ■ Paradigmas según memoria

- **Harvard:** Memoria de datos separada de la memoria de instrucciones.
- **Von Neumann:** Datos e instrucciones en una única unidad de memoria.

\* Bajo estos paradigmas, el computador básico presenta una microarquitectura Harvard y una ISA RISC.

## Arquitectura RISC-V

- Contenido que encuentran en:
  - Clase 8 - Arquitectura RISC-V



# Arquitectura RISC-V - Instrucciones

## ■ Instrucciones más utilizadas: Operaciones lógico-aritméticas

Mnemotecnia	Instrucción	Tipo	Descripción
ADD rd, rs1, rs2	Adición	R	$rd \leftarrow rs1 + rs2$
SUB rd, rs1, rs2	Sustracción	R	$rd \leftarrow rs1 - rs2$
ADDI rd, rs1, imm12	Adición de literal	I	$rd \leftarrow rs1 + imm12$
AND/OR/XOR rd, rs1, rs2	Operación AND/OR/XOR	R	$rd \leftarrow rs1 \& /\^ rs2$
ANDI/ORI/XORI rd, rs1, imm12	Operación AND/OR/XOR con literal	I	$rd \leftarrow rs1 \& /\^ imm12$
SLL/SRL rd, rs1, rs2	Operación <i>shift left/right</i> lógico	R	$rd \leftarrow rs1 <>/> rs2$
SRA rd, rs1, rs2	Operación <i>shift right</i> aritmético	R	$rd \leftarrow rs1 >> rs2$
SLLI/SRLI rd, rs1, shamt	Operación <i>shift left/right</i> lógico con literal	I	$rd \leftarrow rs1 <>/> shamt$
SRAI rd, rs1, shamt	Operación <i>shift right</i> aritmético con literal	I	$rd \leftarrow rs1 >> shamt$

\* *shamt* o *shift amount* es la cantidad de *shifts* a realizar y se codifica como un entero a partir de los 5 bits menos significativos del literal (*imm12[4:0]*).

## Arquitectura RISC-V - Instrucciones

- **Instrucciones más utilizadas:** Operaciones de carga, almacenamiento y subrutinas

Mnemotecnia	Instrucción	Tipo	Descripción
LW rd, imm12(rs1)	Cargar word (32 bits)	I	$rd \leftarrow \text{mem}[rs1 + imm12]$
SW rs2, imm12(rs1)	Almacenar word (32 bits)	S	$rs2 \rightarrow \text{mem}[rs1 + imm12]$
BEQ rs1, rs2, imm12	Salto con condición “igual”	B	if $rs1 == rs2$ : PC $\leftarrow$ PC + imm12
BNE rs1, rs2, imm12	Salto con condición “distinto”	B	if $rs1 != rs2$ : PC $\leftarrow$ PC + imm12
BGE rs1, rs2, imm12	Salto con condición “mayor o igual”	B	if $rs1 >= rs2$ : PC $\leftarrow$ PC + imm12
BLT rs1, rs2, imm12	Salto con condición “menor”	B	if $rs1 < rs2$ : PC $\leftarrow$ PC + imm12
JAL rd, imm20	Salto incondicional con “enlace”	J	$rd \leftarrow \text{PC}+4$ ; PC $\leftarrow$ PC + imm20
JALR rd, imm12(rs1)	Salto incondicional con “enlace” a registro	I	$rd \leftarrow \text{PC}+4$ ; PC $\leftarrow rs1 + imm12$

\* En general, en vez de JAL y JALR trataremos de usar pseudo-instrucciones al ser más intuitivas. LW también se puede usar para leer una dirección de memoria a partir de su *label*.

# Arquitectura RISC-V - Instrucciones

## ■ Instrucciones más utilizadas: Pseudo-instrucciones

Mnemotecnia	Instrucción	Instrucción(es) base
LI rd, imm12	Cargar literal en registro que utiliza $\leq 12$ bits	ADDI rd, zero, imm12
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 $rs1 \leq rs2$	BGE rs2, rs1, offset
BEQZ rs1, offset	Salto si $rs1 = 0$	BEQ rs1, zero, offset
BNEZ rs1, offset	Salto si $rs1 \neq 0$	BNE rs1, zero, offset
J offset	Salto incondicional	JAL zero, offset
CALL offset12	Llamado a subrutina (dirección $\leq 12$ bits)	JALR ra, ra, offset12
RET	Retorno de la subrutina	JALR zero, 0(ra)

## Arquitectura RISC-V - Directivas

### ■ Directivas básicas para correr código en Assembly RISC-V

Directiva de Assembler	Descripción
.text	Segmento de texto (código).
.data	Sección de datos global.
.globl sym	Label sym se vuelve global.
.word w1, w2, ..., wN	Almacena N valores de 32 bits en palabras de memoria sucesivas.
.byte w1, w2, ..., wN	Almacena N valores de 8 bits en bytes de memoria sucesivos.

### ■ ECALLs para imprimir enteros y terminar programa

```
.globl main
.text
main:
    li a0, 11          # a0 = 11
    li a7, 1           # a7 = 1 (PrintInt)
    ecall              # Imprime 11 en consola
    li a7, 10          # a7 = 10 (Exit)
    ecall              # Termina el programa
```

## Arquitectura RISC-V - Convención

### ■ Registros relevantes y encargado de respaldo en subrutinas

Registro(s)	Mnemotecnia ABI	Descripción	Encargado de respaldo
x0	zero	Registro cero. Almacena este valor y <b>no cambia</b> . Ignora las escrituras.	-
x1	ra	<b>Return Address</b> . Almacena la dirección de retorno de las subrutinas.	Caller
x2	sp	<b>Stack Pointer</b> , apunta al último elemento almacenado.	Callee
x5-x7, x28-x31	t0-t6	Registros temporales. Pierden su valor entre llamados de subrutinas.	Caller
x8-x9, x18-x27	s0-s11	Registros guardados ( <i>saved</i> ). Preservan su valor entre llamados de subrutinas.	Callee
x10-x17	a0-a7	Registros para argumentos de subrutinas.	Caller
x10-x11	a0-a1	Si bien son de argumentos de subrutinas, también se utilizan para almacenar valores de retorno.	Caller

# Arquitectura RISC-V - Convención

## ■ Ejemplo de respaldo de registros t\*

En este caso, los registros  $t_0-t_1$  son respaldados por el *caller* porque son modificados por la subrutina y por definición son *caller-saved*.

\* En estricto rigor es necesario respaldar el registro *ra* solo para llamadas de subrutinas anidadas, pero por convención lo respaldamos siempre ante un *call* dado que nuestro código podría llamarse desde otro archivo, por lo que respaldar *ra* asegura que este llamado siempre funcione.

\* Este código almacena un segundo arreglo cuyos elementos poseen el valor duplicado de los elementos de *arr*.

```
.data
len: .word 5
arr: .word 198, 137, 42, 63, 175
.text
start:
    li $0, 4
    lw $1, len
    li $0, 0
    la $1, arr
    mul $2, $0, $1
    add $2, $2, $1
    while:
        lw $0, 0($1)
        addi $3, $0, -12
        sw $0, 0($3)
        sw $1, 4($3)
        sw $2, 8($3)
        call double_give_next_person
        lw $0, 0($3)
        lw $1, 4($3)
        lw $2, 8($3)
        addi $3, $3, 12
        sw $0, 0($2)
        addi $0, $0, 1
        beq $0, $1, end
        add $1, $1, $0
        add $2, $2, $0
        j while
double_give_next_person:
    mv $0, $0
    add $0, $0, $0
    ret
end:
    li $7, 10
    ecall

# s0 = 4 bytes por dirección
# s1 = largo del arreglo
# Contador (i)
# t1 = dirección del arreglo (initialmente arr[0])
# t2 = s0 * s1 = bytes que ocupa el arreglo de entrada
# t2 += t1 = primera dirección que podemos usar de .data
# a0 = arr[i]

# Respaldamos t0, t1 y ra (caller-saved). t1 se respalda
# aunque no se use porque call lo modifica con la
# dirección de retorno en RARS

# Recuperamos t0, t1 y ra y restauramos el stack

# out[i] = a0
# t0 += 1
# Termina cuando se recorre todo el arreglo
# t1 += s0 = dirección arr[i+1]
# t2 += s0 = dirección out[i+1]

# a0 = a0 + t0 = 2 * a0
```

# Arquitectura RISC-V - Convención

## ■ Ejemplo de respaldo de registros s\*

Si en el mismo código invertimos el uso de registros s\* por registros t\*, entonces los debemos respaldar en la subrutina ya que son *callee-saved* (lo que asegura que su valor persista entre llamados).

\* Es el mismo código anterior. Estratégicamente no usamos t1 porque call modifica su valor en RARS y necesitaría respaldo.

```
.data
len: .word 5
arr: .word 198, 137, 42, 63, 175
.text
start:
    li t0, 4
    lw t2, len
    li s0, 0
    la s1, arr
    mul s2, t0, t2
    add s2, s2, s1
    while:
        lw a0, 0($1)
        addi sp, sp, -4
        sw ra, 0(sp)
        call double_give_next_person
        lw ra, 0(sp)
        addi sp, sp, 4
        sw a0, 0($2)
        addi s0, s0, 1
        beq s0, t2, end
        add s1, s1, t0
        add s2, s2, t0
        j while
double_give_next_person:
    addi sp, sp, -4
    sw s0, 0(sp)          # Respaldamos s0 (callee-saved)
    mv s0, a0
    add a0, a0, s0
    lw s0, 0(sp)
    addi sp, sp, 4
    ret
end:
    li a7, 10
    ecall
```

# s0 = 4 bytes por dirección  
# s1 = largo del arreglo  
# Contador (i)  
# t1 = dirección del arreglo (initialmente arr[0])  
# t2 = s0 \* s1 = bytes que ocupa el arreglo de entrada  
# t2 += t1 = primera dirección que podemos usar de .data  
  
# a0 = arr[i]  
  
# Respaldamos ra (caller-saved)  
  
# Recuperamos ra y restauramos el stack  
  
# out[i] = a0  
# s0 += 1  
# Termina cuando se recorre todo el arreglo  
# s1 += t0 = dirección arr[i+1]  
# s2 += t0 = dirección out[i+1]

# Arquitectura RISC-V - Convención

## Ejemplo de respaldo de función recursiva

En este ejemplo, es importante notar que la función actúa como *caller* y como *callee*.

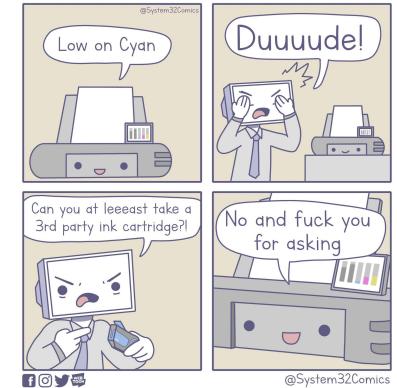
```
.data
N: .word 4           # N = Argumento de factorial. Calcularemos N! = 4

.text
main:
    addi sp, sp, -4      # Reservamos 4 bytes en el stack
    sw ra, 0(sp)        # Respaldamos ra
    la t0, N             # Dirección de memoria de N
    lw t0, 0(t0)          # Valor de N
    add a0, zero, t0      # Argumento 0 = valor de N
    call factorial        # factorial(N)
    li a7, 1              # Llamada de sistema: print int
    ecall                 # Valor en consola: 24 (a0, valor de retorno)
    lw ra, 0(sp)          # Restauramos ra
    addi sp, sp, 4          # Restauramos el stack
    li a7, 10             # Llamada de sistema: exit
    ecall

factorial:
    addi sp, sp, -8      # Reservamos 8 bytes en el stack
    sw ra, 0(sp)        # Respaldamos ra
    sw a0, 4(sp)          # Respaldamos N
    blez a0, factorial_zero # if (N > 0){
    addi a0, a0, -1      #   N -= 1
    call factorial        #   (N-1)! = factorial(N-1)
    lw t0, 4(sp)          #   Recuperamos N
    mul a0, a0, t0          #   N! = N * (N-1)! = N * factorial(N-1)
    j factorial_end       # }
factorial_zero:
    li a0, 1              # else {
    factorial_end:
    lw ra, 0(sp)          #   N! = 1
    addi sp, sp, 8          # Restauramos solo ra, a0 ahora posee el retorno
                           # Restauramos el stack
ret
```

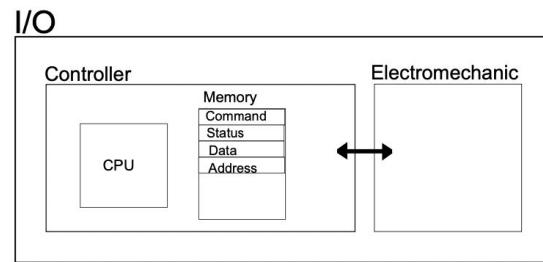
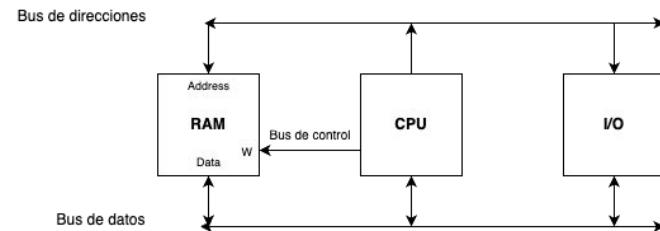
## Dispositivos I/O

- Contenido que encuentran en:
  - **Clase 9 - Comunicación de CPU y Memoria con I/O**
  - **09 - Comunicación de CPU y Memoria con I/O (Apuntes)**



## Dispositivos I/O - Diagrama general

- **Conectividad:** Los dispositivos I/O comparten el bus de datos de la memoria y la CPU y se puede interactuar con ellos con el bus de direcciones.
- **Composición:** Son como un computador, pero con tareas específicas para interactuar con sus piezas electromecánicas. Los registros más importantes son los de **estado** y **comandos** para interactuar con ellos.



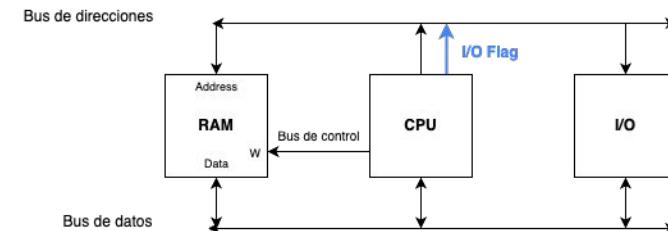
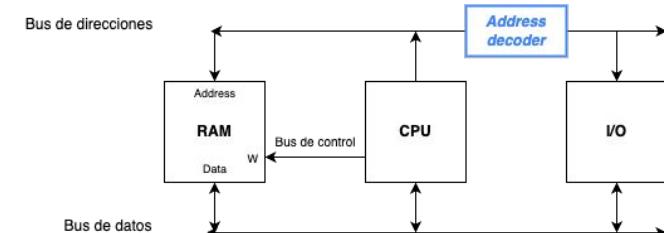
\* Con el registro de estado podemos obtener información sobre el dispositivo, mientras que con el registro de comando podemos interactuar con él para que realice ciertas tareas. Los registros de datos y direcciones se utilizan para dispositivos de almacenamiento, de forma que puedan transferir o recibir datos de la memoria del computador.

## Dispositivos I/O - Comunicación con CPU

- **Memory mapped:** Se reservan  $N$  direcciones de memoria que apuntan a registros de comando/estado de dispositivos I/O. Estos son detectados por el **address decoder**.

Problema: **memory barrier** (se pierde el uso de las direcciones reservadas en la RAM).

- **Port mapped:** Se agregan instrucciones a la ISA para leer (IN Reg, Port) o escribir (OUT Port, Reg) los valores de los registros de comando/estado de los dispositivos I/O a partir de puertos asociados a estos. Se añade una *flag* adicional para poder interpretar bien el puerto como un registro de un dispositivo.



# Dispositivos I/O - Comunicación con CPU

## ■ Ejemplo de interacción con I/O

A continuación, se mostrará un ejemplo de código en RISC-V que interactúa con dispositivos *memory mapped* utilizados por un tren con las siguientes tabla de direcciones, estados, comandos y dirección base 0x10030000:

Offset	Nombre	Descripción
0x00	dir_ise	Contiene la dirección ISR del manejo de alerta del tren
0x04	reg_tr	Registro de comandos del tren
0x08	sensor_te	Registro de estado de sensor de temperatura
0x0C	sensor_pr	Registro de estado de sensor de protuberancia
0x18	sensor_dr	Registro de estado de sensor de derrame
0x20	rad_cl	Registro de comandos de la radio

Nombre	Comando o Estado	Valor
reg_tr	Notificar locomotora	255
reg_tr	Freno de emergencia	127
sensor_te	Temperatura alta	255
sensor_te	Temperatura baja	1
sensor_te	Temperatura normal	127
sensor_pr	Sobredimensión del tren o protuberancias	4
sensor_pr	Rieles libres de obstáculos	2
sensor_dr	Existencia de derrame	3
sensor_dr	Sin derrame de fluidos en el tren	1
rad_cl	Mandar reporte de sensores	1

# Dispositivos I/O - Comunicación con CPU

En esta ISR, se notifica a la locomotora de una emergencia y se activa el freno del tren si se detecta una protuberancia en los rieles usando las direcciones de los registros señalados en la diapositiva anterior. Lo importante es notar cómo se revisan los estados y se realizan comandos leyendo/escribiendo datos en las direcciones mapeadas de los dispositivos.

```
isr:
    addi sp, sp, -12           # Registros s* son callee-saved. También respaldamos ra por llamada anidada
    sw $0, 0(sp)
    sw $1, 4(sp)
    sw ra, 8(sp)
    call sens_pr_sub
    mv a0, t0                  # s0 = estado sensor_pr
    li s1, 0                   # s1 = cantidad de estados no deseados
    li t0, 4                   # Revisión de protuberancias
    beq s0, t0, emergency_action # Protocolo de emergencia si hay protuberancias
    j isr_end
emergency_action:
    call rad_cl_sub
    call reg_tr_sub
    j isr_end
sens_pr_sub:
    li t0, 0x1003000C          # t0 = dirección sensor_pr
    lw a0, 0(t0)                # a0 = valor sensor_pr
    ret
rad_cl_sub:
    li t0, 0x10030020          # t0 = dirección rad_cl
    li t1, 1                   # t1 = comando para enviar reporte de sensores
    sw t1, 0(t0)
    ret
reg_tr_sub:
    li t0, 0x10030004          # t0 = dirección reg_tr
    li t1, 255                 # t1 = comando para notificar a la locomotora
    sw t1, 0(t0)
    li t1, 127                 # t1 = comando para activar el freno de emergencia
    sw t1, 0(t0)
    ret
isr_end:
    lw s0, 0(sp)               # Registros s* y ra se restauran
    lw s1, 4(sp)
    lw ra, 8(sp)
    addi sp, sp, -12           # Se restablece el stack
    mret                       # Retorno de una excepción para RISC-V (usado para ISRs)
```

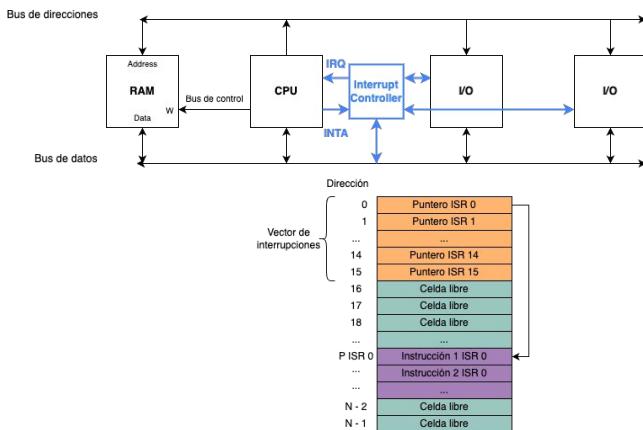
# Dispositivos I/O - Modos de comunicación

- **Polling:** La CPU consulta con cierta frecuencia el estado de un dispositivo para saber si debe realizar una acción.  
**Ej:** Programa en RISC-V que espera a que una impresora prenda para imprimir.

```
.text
main:
    li t0, 112
    lw t1, 0(t0)          # t0 = 0x70 = Registro de estado de una impresora
    li s0, 1
    li s1, 3
    beq t1, s0, print    # Estado 1 = Prendida.
    beq t1, s1, waitTurnOn # Estado 3 = Prendiendo.
    j end
waitTurnOn:
    lw t1, 0(t0)
    beq t1, s0, print
    j waitTurnOn          # Si todavía no está prendida, repetimos la pregunta.
print:
    ...
...
```

- **Interrupciones:** Los dispositivos I/O solicitan una “interrupción” que gatilla una **Interrupt Subroutine** o **ISR** para realizar una acción. Esta solicitud la envían como una señal **Interrupt Request** o **IRQ** al **controlador de interrupciones**, que se encarga de solicitar la atención de la CPU. Si es atendida, la CPU envía la señal **Interrupt Acknowledge** o **INTA** y el controlador le envía de vuelta el **identificador del dispositivo** para acceder a la dirección de su ISR desde el **vector de interrupciones**.

La CPU requiere de una **Interrupt Flag** o **IF** para saber cuándo está atendiendo una interrupción (**IF = 1** → CPU disponible para atender interrupciones).



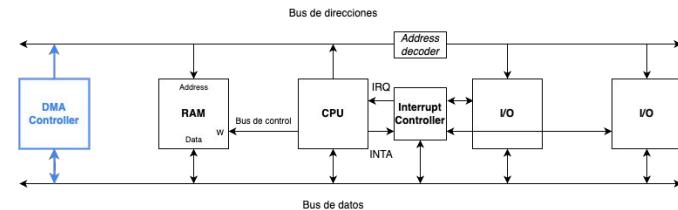
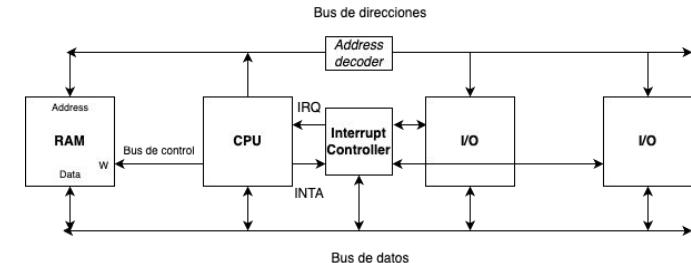
# Dispositivos I/O - Modos de comunicación

## ■ Tipos de interrupción

- **Excepciones:** Condiciones de error al ejecutar una instrucción atendidas por el sistema operativo a través de ISR es específicas: ***exception handlers***. **Ejemplos:** división por cero, *stack overflow*.
- **Traps:** Llamadas explícitas al sistema operativo que gatillan interrupciones que le ceden el control a este para que realice una acción. **Ejemplo en RISC-V:** `ecall`

## Dispositivos I/O - Transferencia de datos

- **Programmed I/O (PIO):** La CPU se encarga de gestionar la transferencia de datos a través de subrutinas que solo tienen ese fin. Cuenta con la desventaja de que los dispositivos I/O solo necesitan acceder a la memoria de datos (RAM), no requieren de procesamiento en la CPU.
- **Direct Memory Access (DMA):** Controlador con acceso al bus de datos que se encarga de gestionar la transferencia de datos entre la RAM y los dispositivos I/O. Solo notifica a la CPU del inicio y del término de la transferencia, permitiendo que esta ejecute otras tareas.



\* El controlador DMA solo puede interactuar con dispositivos I/O *memory mapped*.



**DCC**  
DEPARTAMENTO DE CIENCIA  
DE LA COMPUTACIÓN

IIC2343

# Arquitectura de Computadores

Clase 13 - Resumen del Curso (I2)

Profesor: Germán Leandro Contreras Sagredo