

# Estructura del programador

## 1 Sistemas de numeración

para convertir un n.º de base 2, 4, 8 o 16 a cualquier otra base → convierte poco a poco de binario y luego a otra

→ Pasar de base 10 a otra cualquiera: divido el número por la base a la que lo quiero pasar, hasta que el resto sea menor o igual al número de la base a pasar. Luego ordeno los restos obtenidos de atrás hacia delante, formando el número en la base.

Ej:

$$\begin{array}{r} 2712 \\ \hline 10 \end{array} \quad \begin{array}{r} 132 \\ \hline 10 \end{array}$$

$\frac{2}{10} \quad \frac{1}{10} \quad \frac{6}{10} \quad \frac{2}{10}$

$\frac{1}{10} \quad \frac{0}{10} \quad \frac{3}{10} \quad \frac{2}{10}$

$\frac{0}{10} \quad \frac{1}{10} \quad \frac{0}{10} \quad \frac{2}{10}$

$$11011_2$$

Ahora si el número a pasar tiene coma:

- con la parte entera se hace lo mismo
- con la parte decimal se utiliza el método de multiplicación: se multiplica  $0,5 \times 10^{-1}$  hasta completar los bits

Ej:  $23,53125_{10}$

$$\begin{array}{r} 2318 \\ \hline 10 \end{array}$$

$\frac{2}{10} \quad \frac{2}{10}$

$$\begin{array}{r} 0,53125 \\ \times 8 \\ \hline 4,25000 \\ 0,25000 \\ \times 8 \\ \hline 2 \end{array}$$

$$27,42_8$$

→ Pasar de una base cualquiera a base 10

Multiplico el número del que quiero pasar por el número de la base en la que está, estando este elevado a la posición del número a pasar.

Ej:  $6741 = 6 \times 8^3 + 7 \times 8^2 + 4 \times 8^1 + 1 \times 8^0 = 4441$

total = 2 bits → posiciones (de dir a izq)

Ahora si es un número con coma, luego de esta se utilizan posiciones negativas

Ej:  $27,42_8 = 2 \times 8^3 + 7 \times 8^2 + 4 \times 8^1 + 2 \times 8^0 = 23,53125_{10}$

→ Módulo:  $2^{\text{cantidad de bits}}$  ( $2^8 = 256$ ) → Con el módulo obtengo el rango [0; 255] (con números sin signo)

→ Complemento de un número: Un número + su complemento = 0

⇒ el complemento es lo que le falta a un número para llegar a su base

Ej: complemento de 5 en base 10 → 5  
complemento de 7 en base 8 → 1

Suma: sumo los símbolos, en el caso de que de peso del  $n^{\text{a}}$  de la base en la que estamos  $\Rightarrow$  se "de una vuelta" (comenzando con el cero, y le agregamos un 1 a la siguiente)

$$\begin{array}{r}
 + \quad \begin{array}{c} 1 \\ 1 \\ 1 \\ A \\ 4 \\ E \\ 9 \\ 4 \end{array} \\
 \hline
 \begin{array}{c} F \\ 3 \\ 9 \\ 5 \end{array}
 \end{array}$$

11 + 10  $\rightarrow$  se pone de 15  
 $\rightarrow 11 + 5 - 16$ , me sobran 5  
 para seguir sumando  
 $\Rightarrow$  arranco del 0 + 5  $\rightarrow$  5  
 y como retuve a empezar  
 desde el cero, le sumo 1  
 al próximo sumando

$$A + A + 1 = 1$$

$$A + 5 = 15 = F$$

Tengo  $E + 4 + 1$

$$E + 5 = 14 + 1 = 0$$

$$+ 3 = 3$$

y como me  
pasó sumo 1

COOL!  
en la  $0^{\text{a}}$   
debo descontar  
toda el  
arrastre  
en 12 pas  
significat

complemento a 1:  
convertir los 0's en 1's y  
los 1's en 0's.

complemento a 2:  
convertir los 0's en 1's y  
los 1's en 0's.  
luego se sumó 1.  
(a la derecha un espacio para dividir)

	Bin	Dec
0	0 0 0 0	0
1	0 0 0 1	1
2	0 0 1 0	2
3	0 0 1 1	3
4	0 1 0 0	4
5	0 1 0 1	5
6	0 1 1 0	6
7	0 1 1 1	7
8	1 0 0 0	8
9	1 0 0 1	9
A	1 0 1 0	10
B	1 0 1 1	11
C	1 1 0 0	12
D	1 1 0 1	13
E	1 1 1 0	14
F	1 1 1 1	15

TABLA PIOLA

Buscar Complemento: tengo que

por ejemplo

$$\begin{array}{r}
 1000 \\
 - 930 \\
 \hline
 7
 \end{array}$$

raro y  
complicado

06  $\rightarrow$  complemento a la  
base -1

más sencillo.

y luego le sumo 1  $\Rightarrow 6 + 1 = 7$

complemento

Desta: método para restar

$$A - B = C$$

$$A - B + r^n = C + r^n$$

$$A + r^n - B = C + r^n$$

$$A + (r^n - 1 - B) + 1 = C + r^n$$

complemento a la base -1

complemento a la base -1

yo conozco  
yo no conozco  
 $C + B$

$$A + C_{BB} = C + r^n$$

cont de dígitos  
para representar  
un n<sup>a</sup> de 10 dígitos  
en base do a binario  
 $2^n > 10^2$   
 $\log_2(2^n) > \log_2(10^2)$   
 $(\times \log_2(2)) > 2(\log_2(10))$   
 $\Rightarrow 1 > 6,6438$

Bonito: es el carry negado ( $B = \bar{C}$ )  
este es el flag con el que miro las (-)

Suma: Sumo los símbolos, en el caso de que sea parte del  $n^{\text{a}}$  de la base en la que estamos  $\Rightarrow$  se "de una vuelta" (comenzamos con el cero, y le agregamos un 1 a la siguiente suma)

$$\begin{array}{r} & \text{1} & \text{1} & \text{1} \\ & \text{A} & \text{4} & \text{F} \\ & \text{4} & \text{E} & \text{9} \\ + & \text{1} & \text{4} & \text{F} \\ \hline & \text{F} & \text{3} & \text{9} \\ & \downarrow & \downarrow & \downarrow \\ \text{A} + \text{A} + 1 & = & \text{1} & \text{1} \\ \text{A} + 5 = 15 - \text{F} & = & \text{1} & \text{1} \\ \hline & \text{---} & \text{---} & \text{---} \end{array}$$

$$11 + 10 \Rightarrow \text{se pone de } 16$$

$$\Rightarrow 11 + 5 = 16, \text{ me sobran } 5$$

para seguir sumando

$$\Rightarrow \text{arranque del } 0 + 5 = 5$$

y como regla al empezar

desde el cero, le sumo 1 al próximo sumando

$$\text{F} + 9 + 1 \Rightarrow (\text{F} + 1) + 9$$

$$= 0 + 9 = 9$$

Como se pasó una vez  $\Rightarrow$

le sumo un 1 al proximo

T complemento a 1

complemento de la base en la que estoy

COOL!  
en la que  
debo descontar  
tengo el  
arrancar  
en la pos  
siguiente

	Bin	Bin
0	0	000
1	0	001
2	0	010
3	0	011
4	0	100
5	0	101
6	0	110
7	0	111
8	1	000
9	1	001
A	1	010
B	1	011
C	1	100
D	1	101
E	1	110
F	1	111

TABLA PIOLA

Restar Complemento: tengo que restar.

$$\text{por ejemplo } \begin{array}{r} 100 \\ - 93 \\ \hline 7 \end{array} \quad \left\{ \begin{array}{l} \text{medio} \\ \text{raro y} \\ \text{complicado} \end{array} \right\} \Rightarrow \begin{array}{r} 99 \\ - 93 \\ \hline 06 \end{array}$$

Se pone el  $n^{\text{a}}$  anterior  
de esa base

complemento a la  
base - 1

más sencilla

$$y \text{ luego le sumo } 1 \Rightarrow 6 + 1 = 7$$

complemento

Resta: método para restar

$$A - B = C$$

$$A - B + r^n = C + r^n$$

$$A + r^n - B = C + r^n$$

$$A + (r^n - 1 - B) + 1 = C + r^n$$

complemento a la base - 1

complemento a la base - 1

yo conozco  
yo no conozco  
 $C_B$

$$A + C_{BB} = C + r^n$$

cont de dígitos  
para representar  
un  $n^{\text{a}}$  de 10 dígitos  
en base do a binario

$$2^x, 10^2$$

$$\log_2(2^x) \geq \log_2(10^2)$$

$$(x \log_2(2)), 2x \log_2(10)$$

$$\approx 6,6438$$

Resumen: es el carry negado ( $B = \bar{C}$ )  
este es el flag con el que miro las (-)

$$X = ?$$

$$\begin{array}{r} 934_{10} \\ - 768_{10} \\ \hline \end{array} = A$$

$$\begin{array}{r} 1000 \\ - 768 \\ \hline 232 \end{array} = B$$

IDEAS

- Busco el  $C_B - 1$
- (en binarios es negar todos los  $n_i$ )
- (también hacer  $120^\circ$ )
- Sumo 1 a  $C_B - 1 = C_B$
- el resultado le resto el  $n_i$  sigue activo
- y visto, tengo C

$\frac{999}{232} = C_B - 1$   
 $+ 1$   
 $232 = C_B$

$$\Rightarrow 934_{10}$$

$$+ 232_{10}$$

$$\hline 1166 = C$$

en el caso de los

buscar el  $C_B - 1 \rightarrow$  negar todos los simbolos

Método de Representación ( $n_i$  con signo) (son  $0$  o  $\pm 1$ ) (por ejemplo → complemento a 2)

Si comienza con un  $0 \rightarrow$  es  $+$   
 $1 \rightarrow$  es  $-$

Cómo genero un  $n_i$   $\oplus$   $\ominus$   $\otimes$   $\odot$

- 1) lo paso a binario
- 2) coloco un "0" a la izq (si ya lo tiene)  
si quiero  $\ominus$
- 3) Busco su  $C_B$

$$\text{Ej: } 7_{10} \rightarrow \begin{array}{r} 111_2 \\ 100_2 \\ 101_2 \\ 110_2 \\ 111_2 \end{array} \rightarrow 0111 = +7$$

$$-5_{10} \rightarrow \begin{array}{r} 101_2 \\ 110_2 \\ 101_2 \\ 110_2 \\ 101_2 \end{array} \rightarrow 101 = +5$$

complemento a 2  
 $C_B \rightarrow C_B - 1 + 1$

$$\begin{array}{r} 1010 \\ + 1 \\ \hline 1011 = -5 \end{array}$$

Como hallar en base 10 un  $n_i$   $\ominus$

- 1) lo negamos y le sumamos 1 (o si se buscanemos su  $C_B$ )
- 2) lo pasamos a base 10 y le agregamos el  $\ominus$  adelante

$$\text{Ej: } 11010110 \Rightarrow C_B: + \begin{array}{r} 00101001 \\ 1 \\ \hline 00101010 = -42 \end{array}$$

① fáciles, yo que

otros ejemplos

$$\begin{array}{r} EAF_{10} \\ - A F2_{10} \\ \hline C \end{array} = A$$

$$\begin{array}{r} 1000 \\ - AF2 \\ \hline 50D_{10} \end{array} = B$$

$$\begin{array}{r} 50E_{10} \\ C \end{array} = C$$

$$\begin{array}{r} A + C_B = C + r^n \\ \downarrow EAF \\ + 50E \\ \hline 135D = C + r^n \end{array}$$

$$\begin{array}{r} 1000 \\ - 1000 \\ \hline 35D_{10} \end{array} = C - A - B$$

suma

$$\begin{array}{r} 11111010 \\ + 10000001 \\ \hline 101111011 \end{array} \rightarrow -6$$

$$\begin{array}{r} 101111011 \\ - 101111011 \\ \hline C = 1 \end{array} \rightarrow \text{desborde}$$

carry = numero que te llevas hacia el lado de la operación

→ en las operaciones sin signo  $C = 0$  para que el resultado sea correcto. Si  $C = 1$  el valor se saldrá de rango

$$\begin{array}{r} 934_{10} = A \\ - 768_{10} = B \\ \hline \end{array} \quad \text{IDEAS} \quad \rightarrow \begin{array}{r} 1000 \\ - 768 \\ \hline 232 \end{array} \quad \begin{array}{r} 999 \\ - 768 \\ \hline 231 \end{array} \quad \text{c.p.s.}$$

$$\Rightarrow 934_{10} + 232_{10}$$

$$1166 = C + \underbrace{r^n}_{=1000} \quad \therefore C = 166$$

en el caso de los binarios es  $\oplus$  Focal, yo que buscar el  $C_p = 1$  es  $\rightarrow$  reemplazar todos los símbolos

Método de Representación ( $n$ : con signo) (son  $0$  o  $\pm 1$ ) (por ejemplo complemento a 2)

Si comienza con un  $0 \rightarrow$  es  $(+)$   
 Si comienza con un  $1 \rightarrow$  es  $(-)$

Cómo genero un  $n$ :  $\oplus$  &  $\ominus$

- 1) lo paso a binario
- 2) coloco un "0" a la izq (si ya lo tiene)  
 Si quero  $\ominus$
- 3) Busco su CB

$$\begin{array}{l} \text{Ej: } 7_{10} \rightarrow 111_2 \rightarrow 0111 = +7 \\ -5_{10} \rightarrow 101_2 \rightarrow 101 = +5 \quad \text{complemento a 2} \\ \quad \quad \quad \downarrow \\ \quad \quad \quad \text{C}_p \rightarrow \text{C}_p - 1 + 1 \end{array}$$

Como hallar en base 10 un  $n$ :  $\ominus$

- 1) lo negamos y le sumamos 1  
 (o sea buscamos su CB)
- 2) lo pasamos a base 10 y le agregamos el  $\ominus$  adelante

$$\begin{array}{r} 11010110 \Rightarrow \text{C}_p = + \\ \hline 00100001 \\ \hline 10101010 = -42 \end{array}$$

Otro ejemplo

$$\begin{array}{r} E4F_{16} \\ - AF2_{16} \\ \hline C \end{array}$$

$$\begin{array}{r} 1000 \\ - AF2 \\ \hline 50D_{16} \\ + 1 \\ \hline 50E_{16} \end{array} \quad \text{C.p.s.}$$

$$\begin{array}{r} A + C_p = C + r^n \\ \downarrow \\ E4F \\ + 50E \\ \hline 135D = C + r^n \\ \hline \begin{array}{r} 1000 \\ 35D_{16} \end{array} \quad C = A - B \end{array}$$

suma

$$\begin{array}{r} 11111010 \\ + 10000001 \\ \hline 101111011 \end{array} \quad \begin{array}{l} \rightarrow -6 \\ \rightarrow 127 \end{array}$$

$$\begin{array}{r} 101111011 \\ \downarrow \\ C = 1 \end{array} \quad \begin{array}{l} \rightarrow \text{exceso} \\ \rightarrow 8812 \end{array}$$

carry = numero que te llevas hacia afuera de la operación

en las operaciones sin signo  $C = 0$  para que el resultado sea correcto. Si  $C = 1$  el valor se fue de rango

$$\begin{array}{r}
 \begin{array}{c} 5 \\ + 0 \\ \hline 5 \end{array} \xrightarrow{\oplus} \\
 \begin{array}{c} 0 \\ + 0 \\ \hline 0 \end{array} \xrightarrow{\oplus} \\
 \hline \begin{array}{c} 10 \\ 0 \\ \hline 10 \end{array}
 \end{array} \quad \text{y en representación.}$$

Rango con signo  $2^8 - 256 \rightarrow [-128, 127]$

La suma de 2 positivos da  $\ominus$  ... overflow

**Overflow:** es el número que te llevas en los últimos dos pasos. Si ambos coinciden  $V=0$  y el resultado es correcto. Si son  $\neq \Rightarrow V=1$  se fue de rango el resultado.

→ Para hacer cuentas entre  $n$ : de  $f$  long  $10101100$  (8 bits) le agrego al de menor long los "0" restantes  $1100$  (4 bits) a su izq sólo si no es con signo!!  $00001100$ .

↳ Si es un número con signo  $\ominus \Rightarrow$  le agrego  $1$  !!  
 (yo puse cuando le averigüé el complemento paralelo)

## PUNTO FLOTANTE

(norma IEEE-754)

Número representado:  $M \times b^{e_p}$

32 bits

Ejemplo:  $+6,023 \times 10^{-23}$  → número

1B | 4 | X

1 2 3 4 5 6 7 8 9 10

↳ 1bit para el signo de la mantisa (0 ó 1)

↳ X bits para la mantisa. → representación de la precisión

↳ y bits para el exponente (magnitud y signo) → representación del rango

signo	exponente	Mantisa
1B	exceso 127	23B
1B	exceso 1023	52B

= 32 Bits → simple precisión

= 64 Bits → doble precisión

Si aumentamos el exponente de 2 dígitos a 3

→ se está aumentando el rango (y se

disminuye la precisión)

$n = \oplus$  pequeño:  $\sim 1,2 \times 10^{-3}$

$n = \oplus$  pequeño:  $\sim 5,0 \times 10^{-8}$

$n = \oplus$  grande:  $\sim 3,4 \times 10^{38}$

$n = \oplus$  grande:

32B. Simple:  $(2^{-2}) \times 2^{32}$

→ 64B.

el más positivo con 4 dígitos:  $0111111$

el más negativo con 4 dígitos:  $1000 \rightarrow -8$

Si hago operaciones entre  $n = \oplus$  y  $\ominus$  nula da overflow si tengo  $\oplus$  y  $\ominus$ ,  $\oplus$  y  $\ominus$  nula da overflow

0101  
+ 0101  
-----  
1010 → da  $\ominus$   
→ hay overflow

## de Decimal a punto flotante

(3)

1) Paso a número a binario: -745,89 de base 10 a IEEE754 simple precisión

↳ a binario 0101101001,1110001111 (no se busca el 0) ya fue para eso

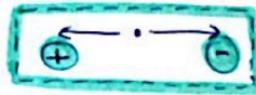
2) Normalizo el número:

De base 10, este bit es tipo el método de multiplicación  $0,89 \times 2 = 1,78$   
 $0,78 \times 2 = 1,56$   
 $0,56 \times 2 = 1,12$

0101101001,1110001111  $\times 2^9$  → 1,0111010011110001111  $\times 2^9$

Cerro la coma hacia el punto 3 más significativa  
(a su derecha pongo la coma)

↳ la coma puede ser corrida hacia la der o izq, según el número.



→ luego agrego ...  $\times 2^{\text{tantos bits que me mire}}$

3) Planteo signo, exponente y mantisa:

→ Signo: - → 1

→ Exponente: Se excede 127 + 9 = 136 → lo paso a binario: 10001000

→ Mantisa: 1,0111010011110001111

↳ nos quedamos con todo lo que se encuentre luego de la coma (yo que como el bit implícito se encuentra siempre, se sabe que esto no se guarda para no ocupar espacio)

4) Junto todo:

1 | 10001000 | 0111010011110001111

1 una coma punto  
poner los 23 bits  
de la mantisa

de flotante a decimal

11000100001110100110000111

1) Separo todo según signo(1), exponente(8bits) y mantisa(los bits restantes)

→ Signo (el 1er bit) → 1

→ exponente (los siguientes 8 bits) → 10001000,  $1 \cdot 2^8 + 0 \cdot 2^7 + \dots$

→ Mantisa (bits restantes) → 0111010011110001111, lo paso a 10 con el método de la

2) Le agrego a la mantisa el bit implícito → 1,0111010011110001111,1111

y lo piso a decimal

3) Piso el exponente: 10001000 → 136 → 136 - 127 = 9

4) Escribo todo junto:  $-1,0111010011110001111 \times 2^9 = -745,889$

(no me decido por el signo)

• el resultado de menor a lo que realmente es ya que los métodos no son exactos

## Suma entre dos punto flotante

1) Me fijo que tengan = exponente

↳ en caso de no tenerlo, se lleva el de menor exponente al mayor (para no perder precisión)

$$\begin{array}{r} 1,23 \times 2^5 \\ + 0,89 \times 2^4 \\ \hline \end{array}$$

↓

$$0,89 \times 2^4 = \frac{0,89 \times 2^6}{2} = 0,445 \times 2^5$$

(1) → cont. de movimientos (de 4 a 5)

base

$$\begin{array}{r} 39,36 \\ + 14,24 \\ \hline 53,60 \end{array}$$

$\times 2^5$

Caso para los mantisces de 5 cifras

$$\begin{array}{r} 0 1000\ 0100 \\ + 0 1000\ 0010 \\ \hline 0 1000\ 0110 \end{array}$$

001110101110001011000

↓  $\times 2^4$

Para hacer esta suma debo agregarle un cero a la mantisa del exponente más bajo, así llego a 5;

de 1100011010111000 pasa a tener 01100011010111000

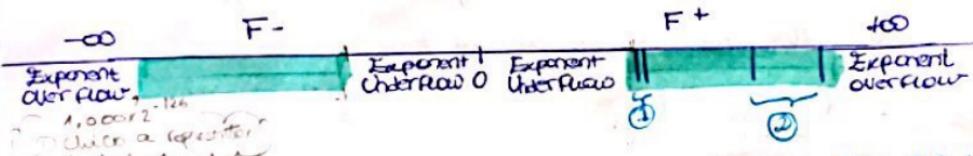
se suman solo las mantisces

$$\begin{array}{r} 0011101011 \\ + 0110001110 \\ \hline 1001111000 \end{array}$$

2) Sumo

$$\begin{array}{r} 1,23 \times 2^5 \\ + 0,445 \times 2^5 \\ \hline 1,675 \times 2^5 = 53,6 \end{array}$$

Rango representable .  $M_{\min} \cdot \text{base}^{exp_{\min}} < \text{Núm} < M_{\max} \cdot \text{base}^{exp_{\max}}$



- ① cuando el n° esté muy cerca de cero → la distancia entre ese y su anterior o posterior es MUY pequeña ( $2^{-126}$ )
- ② cuando el n° esté muy lejos del cero → la distancia entre n° y n° es muy alta ( $2^{126}$ )

Los distancias entre n° sucesivos son:

- punto fijo → siempre = .
- punto flotante → varía según la posición.

## Valores especiales

→ Cero: todos los n° de la mantisa y exp son cero, y el signo puede ser + o -.

→ Infinito: si en f me da un n° que no puedo representar, en neg de tener error tipo  $\infty$  [exp: todos 1's; mantisa: todos 0's]

→ NAN ("Not a Number")

gap = todos 1's , Maths  $\geq 0$ , us true space.

# Álgebra de Boole

(para los otros capítulos) circuitos de serie: ④ tiene corriente directa constante

## → Postulados de Huntington

1. Ley de equivalencia " $=$ ". Si  $a = b \Rightarrow b$  sustituye a a en cualquier expresión.
2. Regla de combinación " $+$ " si  $a$  y  $b$  están en  $\mathbb{N}(\text{conj}) \Rightarrow a + b$  está en  $\mathbb{N}$ . " $\cdot$ " si  $a$  y  $b$  están en  $\mathbb{N} \Rightarrow a \cdot b$  está en  $\mathbb{N}$ .
3. Existe un elemento en  $\mathbb{N}$  que es:

$0 \rightarrow$  de modo que para todo  $n$ :  $a + 0 = a$ .

$1 \rightarrow$  de modo que para todo  $n$ :  $a \cdot 1 = a$ .

### A. Comunitativa:

$$a + b = b + a$$

$$a \cdot b = b \cdot a$$

### B. Distributividad:

$$\begin{aligned} a(b+c) &= (ab)+(a \cdot c) \\ a+(b+c) &= (a+b) \cdot (a+c) \end{aligned}$$

### C. Existe un $n$ s.t.: (a complemento)

$$a \cdot \bar{a} = 0$$

$$a + \bar{a} = 1$$

### D. Existe en $\mathbb{N}$ al menos dos elementos no equivalentes entre sí.

## → Principio de dualidad

$+ \leftrightarrow \cdot$   
 si cambia todos los  $+$  por  $\cdot$  y viceversa  
 $0 \leftrightarrow 1$   
 y cambia los  $0$  por  $1$  y viceversa

todo sigue teniendo sentido

## → Definiciones

- Literal: una variable y/o su complemento.
- Término producto: conjunto de literales relacionados por el conectivo " $\cdot$ ".
- Término suma: conjunto de literales relacionados por el conectivo " $+$ ".
- Término normal: término producto o suma en el cual ningún literal aparece más de vez
- ↳  $F = A\bar{A} + \bar{B}C \times \Rightarrow F = \bar{A}\bar{B} + B\bar{C} \checkmark$
- Término canónico: término normal que contiene todos literales como variables

$$b) F(A, B, C) = \bar{A}\bar{B}C + A\bar{B}C \quad \text{suma de productos}$$

$$F(A, B, C) = (A+B+C)(A+\bar{B}+C) \quad \text{producto de sumas}$$

} ambos representan lo mismo

## Teoremas

- Idempotencia:  $a + a = a$ ;  $a \cdot a = a$
- Elemento absorbente:  $a + 1 = 1$ ;  $a \cdot 0 = 0$  no concuerda
- Absorción:  $a + (a \cdot b) = a$ ;  $a \cdot (a + b) = a$
- Asociatividad:  $a + (b + c) = (a + b) + c$ ;  $a \cdot (b \cdot c) = (a \cdot b) \cdot c$ .
- Complemento único: el elemento  $\bar{a}$  asociado a  $a$  es único.
- Invención:  $\bar{\bar{a}} = a$ .
- Leyes de Morgan:  $(a + b)' = a' \cdot b'$ ;  $(a \cdot b)' = a' + b'$ .

$$\begin{cases} 0' = 1 \\ 1' = 0 \end{cases}$$

$$\begin{cases} A + \bar{A} = 1 \\ A \cdot \bar{A} = 0 \end{cases}$$

→ cada función tiene una única tabla de verdad

⇒ Se puede representar una función lógica de los sig. maneras:

- Tabla de verdad.
- Expresión algebraica por suma de **miniterminos** (suma de productos).
- Expresión algebraica por producto de **maxiterminos** (producto de sumas).
- Suma de miniterminos de forma numérica.
- Producto de maxiterminos de forma numérica.

*ejemplo*  $\rightarrow F(A, B, C) = \sum_{\text{celdas}} \text{Miniterminos } (1, 3) + \prod_{\text{celdas}} \text{Maxiterminos } (0, 2, 3, 4, 6, 7)$

Tabla			
A	B	C	F
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	0
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	0

④ Tabla

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

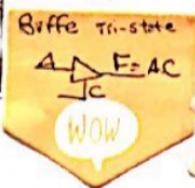
$$F(A, B, C) = (A + B + C) \cdot (A + \bar{B} + C) \cdot ($$

**miniterminos**

Tiene que tener igual cantidad de términos con unos (2)

**MÁXITERMINO**

Tiene que tener igual cantidad de términos sin de ceros (6)



cuando  $C = 0$ , la salida está desconectada ( $F = X$ )

## Compuestos

$$\begin{array}{c} A \\ B \end{array} \Rightarrow A + B$$

$$\begin{array}{c} A \\ B \end{array} \text{ AND } A \cdot B$$

$$\begin{array}{c} A \\ B \end{array} \text{ XOR } A \oplus B = \bar{A} \cdot B + A \cdot \bar{B}$$

OP EXCLUSIVA

$$\begin{array}{c} A \\ B \end{array} \text{ NOR } \overline{A + B} = \overline{AB}$$

$$\begin{array}{c} A \\ B \end{array} \text{ NAND } \overline{A \cdot B} = \overline{A + B}$$

$$\begin{array}{c} A \\ B \end{array} \Rightarrow \bar{A}$$

(or negada)  $\begin{array}{c} 1 \\ 0 \end{array} \Rightarrow 0$

(and negada)

(not NAND y NOR tiene su propia tabla)

0	1	0	1
1	0	1	0
0	1	0	1
1	1	0	0

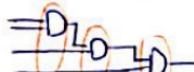
# Diseño de circuitos combinacionales

5

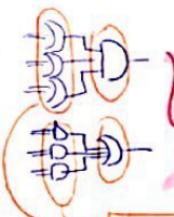
## Lógica combinacional

- Lógica de dos niveles
  - Suma de productos
  - Producto de sumas

## Lógica multinivel



para llegar al final  
hay que esperar los resultados  
de la compuerta, y solo  
por niveles



circuitos que  
no tienen memoria

Apenas cambia una variable  
el resultado de la función cambie  
sin importar los resultados anteriores

- ⊕ fácil de visualizar
- ⊕ rápido.
- ⊕ barato.

los resultados salen al mismo  
tiempo, no hay que esperar a  
resultados anteriores

La complejidad de una solución se  
puede medir → comp :  $\sum_{(compuertas)}^{(tipos de compuertas)}$

## Método de Karnaugh

$\bar{A} \bar{B}$	00	01	11	10
00	1	1	1	1
01	4	5	1	6
11	1	1	13	4
10	1	9	11	10

los extremos (los 4  
esquinas) de pueden  
juntar todos

$\bar{A} \bar{B}$	00	01	11	10
00	C			
01	0		0	
11		0	0	
10		0	0	

$(A+B+C)$        $(A'B'C')$

propone una manera alternativa de simplificación de circuitos lógicos  
Se agrupan en potencias de dos  $\rightarrow 2^1, 2^2, 2^3, 2^4$

Implicante primo: localizar todos los rectángulos más grandes posibles, agrupando todos los 1/0. → Si alguno de estos rectángulos contiene algún 1/0 que no está en ningún otro  $\Rightarrow$  es implicante primo esencial

Expresión mínima :  $\sum_{(con 1's en la tabla)} m_i$  (suma de productos) (NAND)

$$\bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}C\bar{D} + A\bar{B}\bar{C}\bar{D} + A\bar{B}C\bar{D} = \\ = \bar{A}\bar{B}\bar{D}(\bar{C}+C) + A\bar{B}\bar{D}(\bar{C}+C) = \bar{A}\bar{B}\bar{D} + A\bar{B}\bar{D} = \bar{B}\bar{D}$$

Expresión mínima:  $\prod_{(con 0's en la tabla)} M_i$  (producto de sumas) (NOR)

$$F(A, B, C, D) = \bar{B}\bar{D} + A\bar{B}\bar{C} + \bar{A}B\bar{D} + \bar{A}\bar{B}C$$

$$F(A, B, C, D) = (A+\bar{B}+D)(A+B+C)(B+C+\bar{D})$$

Adyacentes: dos términos en los que solo cambia una sola variable (recíndida)

Algoritmo de Quine-McCluskey → puede tener muchísimos miniterminos  
variables → Tenemos 4 variables → 4 dígitos binarios

1) Agrupo miniterminos según la cant de 1's.

2) Comparo grupos adyacentes.

3) Combinar miniterminos → implicantes.

4) Combinar implicantes en pasos sucesivos

5) Eliminar implicantes repetidos

Ejemplo (paso 1)

grupos 0	0 0000
grupos 1	1 0000
	2 0010
	3 0000
grupos 2	5 0101
	6 0110
	7 0100
	10 1001
restos	14 1110

(paso 2)

0, 1	000-
0, 2	00-
0, 8	-000
1, 5	0-01
1, 9	-001
2, 6	0-10
2, 10	-010
8, 9	100
8, 10	10-
5, 7	01-1
6, 17	011-
6, 14	-110
10, 14	1-10

ot que difiere  
una sola posición

resto (paso 3)

0, 1, 8, 9	-00-
0, 1, 2, 8, 10	-0-0-
0, 1, 2, 8, 10	-00-
0, 1, 2, 8, 10	-0-0-

(paso 4)

↓  
elimina  
los comb.  
repetidos

↓  
paso final  
suma tot.

$$F = \bar{a}c'd + a\bar{b}d + \\ + \bar{a}bc + b\bar{c}' + \\ + \bar{b}d' + cd'$$

Segunda parte

	0	1	2	5	6	7	E	10	11	14
(0, 1, 8, 9)	b'c'	X	X				X	X		
(0, 2, 8, 10)	b'd'	X		X			X		X	
(2, 6, 10, 14)	cd'			X	X			X	X	(X)
(1, 5)	a'c'd		X		X					
(5, 7)	a'b'd				X	X				
(6, 7)	a'b'c				X	X				

el  $cd'$  → es el único que agrega el 14 → es esencial!  
(y agrupa a todos los 11 en esa fila: 2, 6, 10, 11)  
→ los tomo para todos los demás opciones.

el  $b'c'$  → es el único que agrupa el 9 → es esencial!  
(y agrupa a todos los 11 de esa fila → 0, 1, 8, 9)  
→ los tomo de las demás opciones.

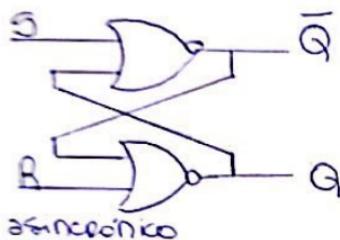
Sólo me quedan el 5 y 7: hay uno que los agrupa a ambos → F =  $cd' + b'c' + a'b'd$

as los flip-flops tienen los mismos estados.

### 3. FLIP-FLOP (la memoria)

ASÍNCRONICO: tienen más entradas de control (llamativamente)

#### CIRCUITO BIESTABLE S-R



$$\text{Ecación característica: } Q^{NH} = S + \bar{R}Q^N$$

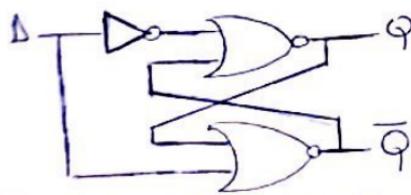
→ S=1 R=0 : comb que no hace nada

→ S=0 R=1 : produce un cambio cuando está en 1 (un pulso) y "enciende" a Q ; cuando S se pone en 0 : Q sigue encendido → permanece en 1

→ Si intenta tocar lo que S hace, ya que "coge" a Q, cada vez que se activa

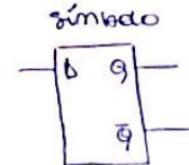
→ S=1, R=1 : "prohibido" no sucede

#### Flip Flop D asíncronico

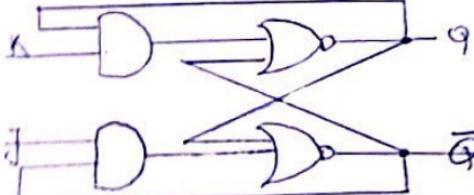


la salida copia } apunta al almacenamiento  
la entrada } de en nodo hasta que este cambiado → la dura  
una pulsación

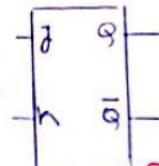
D	Q
0	0
1	1



#### Flip Flop JK asíncronico



J	K	Q <sup>NH</sup>
0	0	Q <sup>L</sup>
0	1	0
1	0	1
1	1	Q <sup>L</sup>

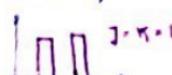


$$\text{Ecación característica: } Q^{NH} = J\bar{Q} + \bar{K}Q$$

mismo solamente para la mitad o una secuencia los resultados  
que no es que quiera

luego el SR  
que no se puebla  
también un 111

cuando  $J=K=1$  tiene  
se comporta como un



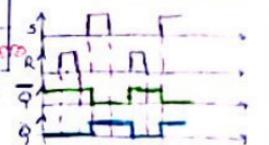
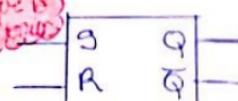
- Circuitos combinacionales: (6)
- en función de los entradas
- Circuitos secuenciales:
- la salida depende de lo actual, pero también de la historia

SR RESET

S	R	Q <sup>NH</sup>
0	0	Q <sup>N</sup>
0	1	0
1	0	1
1	1	-



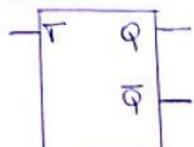
símbolo



PROBLEMA

que pasa

## FLIP FLOP T



$$\bar{Q} = T \oplus Q$$

T	$Q^{in}$	$Q^{out}$
0	Q	1
1	$\bar{Q}$	0

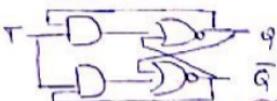
$$en T=1 \rightarrow f \leftarrow RT = 1$$

luego el JK, con sus entradas unidas

Si el estado actual es:  
 $\rightarrow 0 \rightarrow$  da 1.  
 $\rightarrow 1 \rightarrow$  da 0.

si  $T=0$  no combina nada.

Problema: para  $T=1$  oscila y no se para la frec.



SINCRÓNICO: además de las entradas de control posee una entrada por reloj.

o) Circuito de estados (o sea los flip-flop) se sincroniza a si mismo al aceptar cambios de sus entradas solo en instantes determinados.

conocemos como frecuencia a  $f = \frac{1}{T}$  periodo.

• El uso de una señal de sincronismo permite la eliminación de errores/retrasos por medio de la creación de un circuito bistable sincrónico, por ejemplo un SR sincrónico, incluye una entrada CLK como señal de sincronismo.

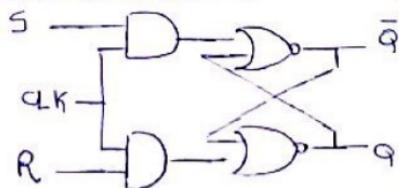
→ Las entradas S y R ya no pueden cambiar el estado del circuito hasta que no se reciba un nivel alto en CLK.

Por consiguiente, si los cambios en S y R se producen mientras CLK está inactivo (bajo), cuando este señal pase a 1 los nuevos estados de S y R, estables, se almacenarán en el flip-flop.

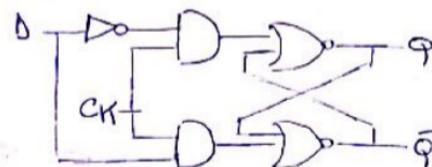
→ controlaremos los oscilaciones, entonces controlaremos la frecuencia, para que el pulso de alta dure lo suficiente.

el tiempo que requiere la señal S de recorrer para subir, caer y volver a subir.

## SR SINCRO NICO



## D sincrónico



D	Q
0	0
1	1

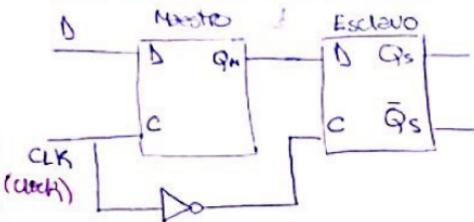
D	Q
0	0
1	1

permite aplicar un D a la misma tasa.

yo lo digo cuando tiene que quedar en el momento que pulso de reloj.

cuando se le usa en aplicaciones de redondeo, desde la salida hacia la entrada a través de otros circuitos, estos redondeos pueden provocar que el FF cambie sus estados más de una vez por cada reloj → en la frecuencia de que solo cambie 1 vez por alto de reloj, se sueltan el largo de redondeo con la estructura de MAESTRO-ESCLAVO.

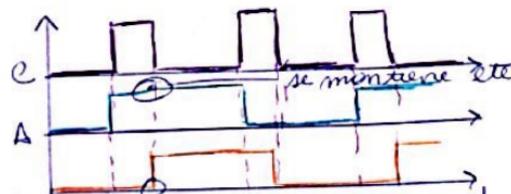
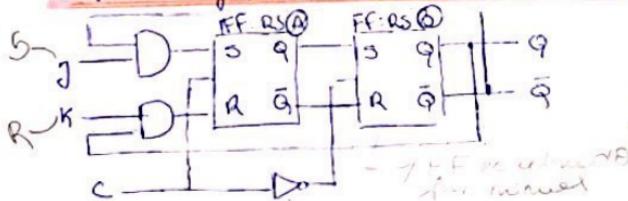
## Flip Flop D del tipo Maestro-Escávado



La entrada D se transferirá a la salida Q\_S del F.F. esclavo cuando la señal del reloj sube y regrese a bajar  $\rightarrow$  solo cuando tiene curva de subida del reloj. (de flanco ascendente L-1 a descendiente L-0)



## Flip Flop JK sincrónico del tipo Maestro-Escávado



Se combina con el reloj (C) negativo  
para B los niveles C=0  
se dice "actua".

actuar de reloj  $\rightarrow$  cuando sube

comparte un 2. Flip Flops encadenados donde el Segundo utiliza una señal de sincronismo que está negada con respecto a la que utiliza el primero.

El Flip-Flop Maestro cambia cuando la entrada principal de reloj está en su estado alto, pero el esclavo no puede cambiar hasta que esa entrada no vuelve a subir.

## o FLIP-FLOP:

- activado por NIVEL: puede controlar sus estados de forma continua cuando la señal del reloj está activa ( $A_{HO} \cdot 1; B_{AD} \cdot 0$ )
- activado por FLANCO: solo cambia en una transición ascendente o descendente de la señal del reloj (subida  $\nearrow$ ; bajada  $\searrow$ )

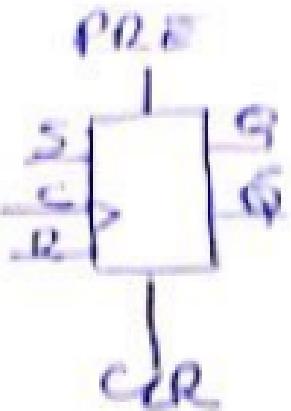
Tiempo controlado lo de los oscilaciones  
el B es una copia del A  $\rightarrow$  maestro.  
esclavo

! El combina R da:  
con los flancos:  
la salida que nombra  
es la del B  $\Rightarrow$   
es un flanco descendente  
ya que B combina  
cada vez que C  
desciende, ya que es el esclavo  
! La salida de A es de  
un flanco ascendente  
ya que es el maestro

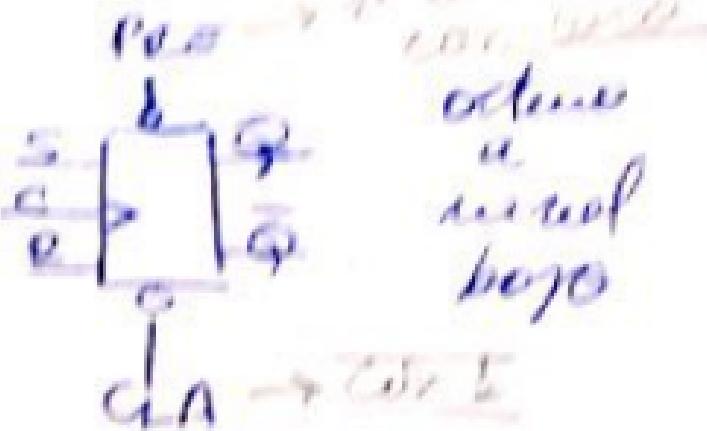
los estados memorizados tienen entradas asimétricas

Preset: se activa con 1

Clear: se activa con 0



salida  
a  
nivel  
alto



salida  
a  
nivel  
bajo

## 4.3 Registros y Contadores

→ Registros (almacenamiento de datos en los microprocesadores)

Capacidad de guardar información.

Están formados por **N FF** + lógica de compuertas.

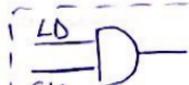
### → Almacenamiento

- entrada y salida en //.
- entrada y salida en serie.

### → Desplazamiento (multiplicación y división por desplazamiento)

- entrada en serie y salida en //.
- entrada en // y salida en serie.

- Siempre fijo
- Registros
- Registros están formados de almacénamiento (FF) y reparto de la complejidad.

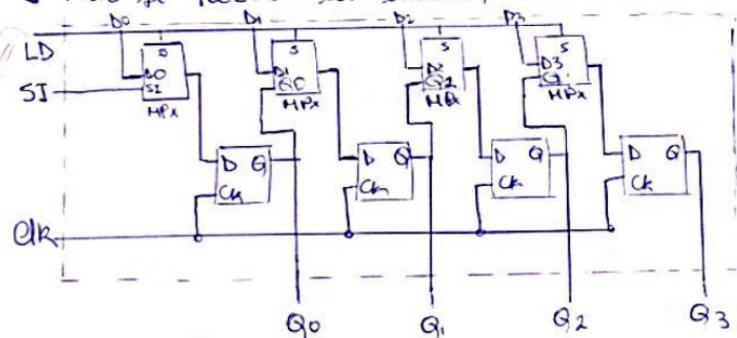


esto se utiliza para la multiplicación de 2 clock.

Si LD = 0, no sucede nada.

Si LD = 1 → depende del clock

Registro sincrónico: puedo cargar con un MUX si recibe la entrada en // o serie.



LD → carga en // 1 desplazamiento

en serie entra con 1 solo cable

serie

1 sola entrada: SI

1 entrada ck

4 salidas

iniciando total de 3

	Q <sub>0</sub>	Q <sub>1</sub>	Q <sub>2</sub>	Q <sub>3</sub>
0	0	0	0	0
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

desplazamiento

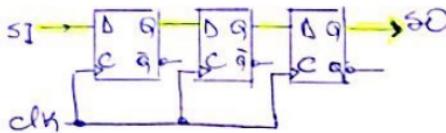
$$\begin{aligned} Q_0(t+1) &= S_2 \\ Q_1(t+1) &= Q_0(t) \\ Q_2(t+1) &= Q_1(t) \\ Q_3(t+1) &= Q_2(t) \end{aligned}$$

el último bit se pierde

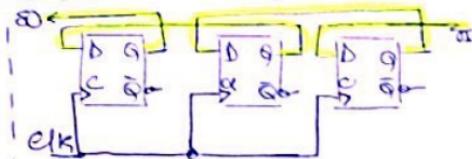
la suma temporalmente de 4 bits, crea neg que impreso 1 de va el último

PALABRA: conjunto de Bits?

## Dirección de desplazamiento fija (cableada)



va hacia la dirección de divide



va hacia la dirección de multiplicar

## CONTADORES

Círculo secuencial capaz de almacenar y contar pulsos de un CLK.

- contar cont. de veces que un evento sucede.
- medir tiempo.
- contar cont. de bits enviados y recibidos.

→ con cada ciclo del reloj la salida del contador se incrementa en 1

→ cuando al 1er estado una vez agotada la capacidad de cuenta

Modulos: cont de estados. ( $\text{bits} = 2^{\text{cant de FF}}$ )

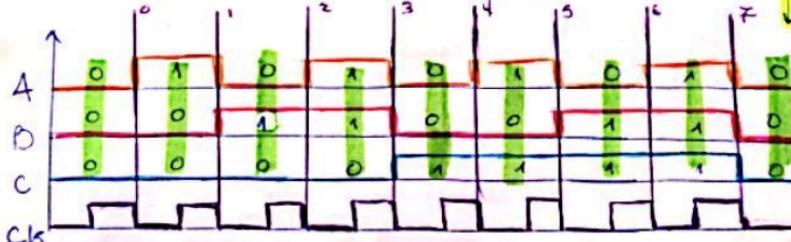
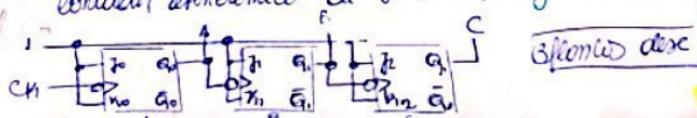
código de cuenta: forma en la que se seleccionan los estados con su representación (binaria u otro)

Diagrama de estados: como pase de un estado a otro según la entrada



Estado prohibido: no cumple un ciclo.

Inicialización: cuando le das energía puedes iniciar el conteo desde 10<sup>n</sup> = 2<sup>n</sup>.



por ejemplo:  
tiene 9 estados, si  
se te necesita  
el 10º se  
necesitan 4 FF  
⇒ tiene  $2^4 = 16$   
estados.  
señal 10<sup>9</sup> = 2<sup>9</sup>  
tengo 7 dígitos  
sin probabilidad

A combina  
con el clock  
descendente  
B combina 90  
que A desc  
y cambia  
en que B va

tempo de doble	0000	0001	0010
hacia dentro ↑	0011	0100	0101
	0110	1000	1001
	1010	1100	1101
	1110	1111	1111

el contador  
esta en  
binario

modulo 8 ( $2^3 - 8$ )

(invierte todo en 000)

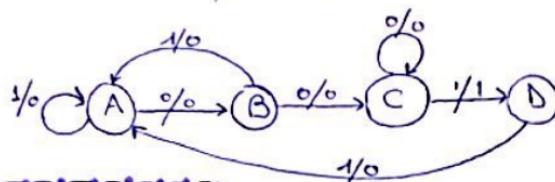
el Cn de B invierte la salida de A  
y el de C sigue la salida de B

los contactos sincrónicos son nulos  $\rightarrow$  funde cuando se enciende.  
 alguna compuerta + algo para hacer lo que quieras.

(4 líneas)

Diseñar una secuencia.

- 1º Hacer un diagrama de estados con lo pedido.  
por ejemplo: quiero que se detecte una secuencia 001 con tal FF



Si aparece un 1 antes de que entén los dos ceros se vuelve a empezar

entrada / salida  
 Estado 0 : espera que entre la secuencia  
 Estado 1 : finaliza la secuencia

- 2º Se le pone un valor a los estados (el ejer)

$$A: 00 ; B: 01 ; C: 10 ; D: 11$$

Tenemos 4 estados  $\Rightarrow$  necesitamos 2bit  $\Rightarrow$  2FF ( $2^2 = 4$ )

FF  
 estados (bits)

- 3º Tabla de Transición

Entrada FF <sub>1</sub> , Entrada FF <sub>0</sub>		E	Q <sub>1in</sub>	Q <sub>0in</sub>	salida FF <sub>1</sub>	salida FF <sub>0</sub>	FF <sub>1</sub>	FF <sub>0</sub>
Q <sub>1in</sub>	Q <sub>0in</sub>	0	0	0	0	0	X	0
0	0	1	0	0	0	0	X	0
0	1	0	1	0	0	0	1	0
0	1	1	0	0	0	X	0	1
1	0	0	1	0	0	0	X	0
1	0	1	1	1	1	0	X	0
1	1	0	0	1	0	1	0	X
1	1	1	0	0	0	1	0	1

#### 4- Tablas de Karnaugh (de c/salida y FF)

E	Q <sub>0</sub>	Q <sub>1</sub>	Q <sub>2</sub>	Q <sub>3</sub>	Q <sub>4</sub>
0	0	0	0	0	0
1	0	0	0	0	1

$$S = Q_1 \bar{Q}_0 E$$

E	Q <sub>0</sub>	Q <sub>1</sub>	Q <sub>2</sub>	Q <sub>3</sub>	Q <sub>4</sub>
0	X	0	1	0	0
1	X	X	1	0	0

$$R_1, G, G_0$$

E	Q <sub>0</sub>	Q <sub>1</sub>	Q <sub>2</sub>	Q <sub>3</sub>	Q <sub>4</sub>
0	0	0	1	0	X
1	0	0	0	0	X

$$S_1, Q_1 \bar{Q}_0 E$$

E	Q <sub>0</sub>	Q <sub>1</sub>	Q <sub>2</sub>	Q <sub>3</sub>	Q <sub>4</sub>
0	0	0	1	0	X
1	X	1	1	0	0

$$R_0, \bar{Q}_1 \bar{Q}_0 + Q_0 E$$

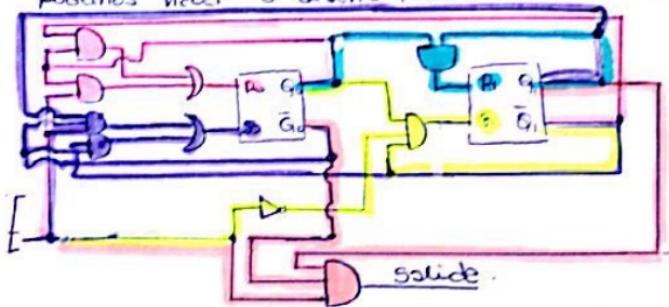
E	Q <sub>0</sub>	Q <sub>1</sub>	Q <sub>2</sub>	Q <sub>3</sub>	Q <sub>4</sub>
0	1	0	0	X	0
1	0	0	0	0	1

$$Q_0, \bar{Q}_1 \bar{Q}_0 E + Q_1 Q_0 E$$

→ secuencia de los estados: 00000 → 00001 → 00010 → 00011 → 00100 → 00101 → 00110 → 00111 → 01000 → 01001 → 01010 → 01011 → 01100 → 01101 → 01110 → 01111 → 10000 → ...  
los estados son los que cumplen el diseño.

#### 5- Dibujar el circuito final

Teniendo todos los tablas y ecuaciones (gracias a Karnaugh), podemos hacer el diseño final del circuito.



en el FF del bit menos significativo, ante la derecha

#### Algunas tablas de transición

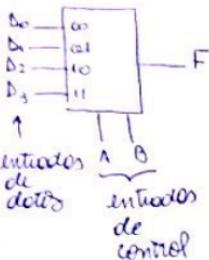
Q <sub>n</sub>	Q <sub>n+1</sub>	S, R	J, K	D	T
0	0	0, X	0, X	0	0
0	1	1, 0	1, X	1	1
1	0	0, 1	X, 1	0	1
1	1	X, 0	X, 0	1	0

(+) fácil de hacer  
D y T a la vez  
de diseño, y  
JK & SR son más  
complicados

(-) la complejidad del  
circuito es alta  
y complicado el JK  
y complicado el  
el T & D

que pasa de un estado a otro

Multiplexor (MUX) → elemento que conecta una cantidad de entradas a una sola salida única.



A	B	F
0	0	D <sub>0</sub>
0	1	D <sub>1</sub>
1	0	D <sub>2</sub>
1	1	D <sub>3</sub>

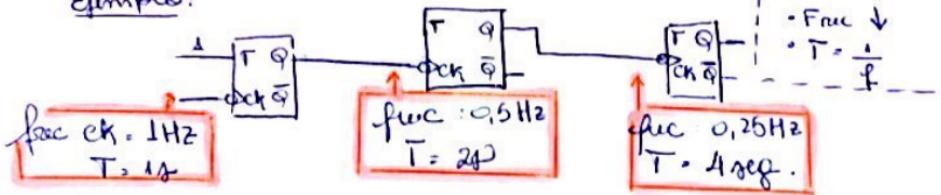
Multiplexor con 4 entradas y 2 salidas. La salida F adopta el valor correspondiente a la entrada de datos seleccionada por los líneas de control A y B.  
(si AB = 00 el valor de salida D<sub>0</sub>)

→ cuando se diseñan circuitos con MUX se los considera como un bloque funcional, que puede representarse como una "caja negra".

→ los contadores y FF NO pueden aumentar la frec.

→ Si se arranca con  $X\text{Hz}$  en el circuito → le entra  $X\text{Hz}$  al 1º FF pero luego se reduce a la mitad para entrar al 2º FF con  $\frac{X}{2}\text{Hz}$ , y luego se reduce a la mitad de nuevo, y entra al 3º FF con  $\frac{X}{2} \cdot \frac{1}{2}\text{Hz}$ , y así sucesivamente.

Ejemplo:



# 5. ISA (Arquitectura del set de instrucciones)

## Accesos a memoria RAM

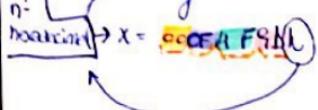
Forma una estructura de datos organizada tipo tabla  
(conjunto de filas encolumnadas ( $\text{c/fila} = 1 \text{ byte}$ ))

Cada renglón de tabla es identificado por su "dirección".

Cada dato es agrupado FISICAMENTE de a 8 bits (= 1 byte).

## Guardar palabras de varios bytes

que me guardo la variable  $x$  en RAM:



- cada 2 hexadecimales
  - byte + byte.
  - cada dígito son 4 bits
- el byte  $\Leftrightarrow$  signifi. en dirección  
 $\Leftrightarrow$  base en memoria.

## Espacio direccional

Si tengo 32 bits en el procesador  $\Rightarrow$  puedo guardar  $2^{32}$  direcciones (4GB).

Si tengo 64 bits en el procesador  $\Rightarrow$  puedo guardar  $2^{64}$  direcciones.

## Aplicación

Local memory

1 word
2 words
4 words
8 words
16 words
32 words
64 words
128 words
256 words

X →

00

0F

4F

9F

...

high memory

↓

low memory

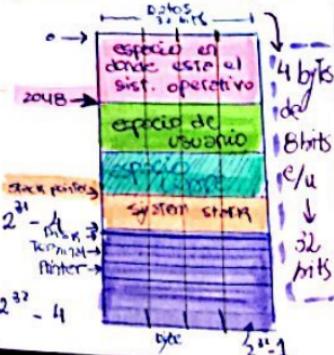
- Si tengo memoria estática, las variables y el programa se guardan durante el tiempo de uso del programa (mientras se ejecuta).
- Si tengo memoria dinámica (stack) se utiliza y se libera la memoria a medida que la aplicación lo pide. } muy duradera

Mapa de memoria de un sistema  $\rightarrow$  representa la asignación de espacio de direcciones con arquitectura ARC (A Risc computer)

- Memoria 32 bits direcciones por byte  $\rightarrow$  espacio de direcciones  $2^{32}$
- Endianamiento del tipo Big-Endian
- Todas las instrucciones ocupan 32 bits
- Hay 32 registros de 32 bits.

Representación en donde tengo 4 bytes por fila (uno al lado del otro) para eso la ult. posición es  $2^{32}-4$

espacio dentro (4)  
fuera (dato tipo de la computadora (loja, puas, guante, monedas))  
dispositivos de entrada y salida (teclado, monitor, etc.)  
I/O space



→ A lo largo de la ejecución el contenido del stack ha cambiado  
↳ a medida que va creciendo el pointer del stack va  
aumentando (ya que crece hacia arriba)

## Algunas instrucciones ARC

memoria → pasadas a memoria

reg. → Id.

→ St

→ Sethi

al min. → andcc  
un de → orcc  
tux → orcc  
opear → orcc  
des desf → orcc  
salir → orcc  
en un → srl

registo → add

→ add

→ call

→ jmpl

→ be

→ breq

→ bes

→ ltrs

→ ba

[...cc → especifican que luego de una operación deben actualizarse los adyacentes de concatenación (2, 1, 2, 0).]

lee en la memoria ram y pasa el dato al registro (carga)  
pasa un dato de un registro a la memoria ram (descarga)

carga los 22 bits más significativos de un registro  
para ni muy grandes

lógica and → bit del resultado es 1 si los bits correspondientes de ambos operandos vale 1, sino es 0

lógica or → bit del resultado es 1 si al menos uno de los 16 bits correspondientes de los operandos es 1, sino es 0

lógica nor → complemento a dos // 16 bits o su inverso // 16 bits de los operandos

lógica de desplazamiento (corrimiento a la derecha) → 1, sino es 1

suma

permite implementar una función

vuelve a la función que lo llama

si es = a cero // Salto por igual (polarización)

si es negativo .

si es carry (arrastre)

si es overflow (desborde)

Igo to // Salto incondicional .

cc: código  
de condición  
trueno  
o falso

## Registros accesibles al programador

- el registro %r0 : es a cero → puede quitarlo lo que quieras pero siempre me da de ceros/falso
- el registro %r14 : %rip stack pointer → apunta a la pila de ejecución
- el registro %r15 : %rr → guarda el valor del call para el jump
- el registro %psr : guarda los flags
- el registro %pc : guarda la ejecución del momento de obtenerlos (no tiene acceso)
- el resto de los registros se pueden usar todos para lo que queremos! (Igo del r0 al r15)

## Códigos de condición de ARC

L(2) → resultado cero

F(1) → resultado negativo

E(2) → resultado a la salida de la unidad aritmética lógica de 32 bits

(v) → Desbordamiento, se activa cuando la operación aritmética es demasiado grande como para ser manejada

## Sintaxis

lab.1: addec begin end org dw8 addc andcc orcc xorcc notcc complementos

etiquetas

función

(o pun-

tay en

estos repetidos

y letras

en serie

en este

repetido

## Directivas del ensamblador

- • equ Declara constantes → traduce lo que compila en 0's y 1's  
empieza el programa.) (no tiene nulla por que no con el procesador)
  - • begin
  - • end Termina el programa.
  - • org la proxima linea se almacen en donde indica → .org 2008
  - • dub define un espacio de memoria • dw8 25 → reserva 25 bytes para lo sup.
  - • if condicional if
  - • endif termina el condicional.
  - • global externo
  - • Ejemplos
- lo ponemos a disposición de otros módulos → cuando sue los función no acceda en otro
- uso una función que no aparece en ese prog., sino en otro módulo. → se nombre al comienzo del bucle

## en C:

```
int main(void)
{
    long x=15;
    long y=9;
    long z=0;
    z=x+y;
    Return(0);
}
```

## en ensamblador:

### puntitos

begin → localiza todo lo que sigue a partir de 2008

org 2008 → carga la variable x en %r1

ld [x] → carga la variable y en %r2

ld [y] → suma lo que hay en r1 y r2 y lo pone en r3

add %r1, %r2, %r3 → pasa el dato guardado en r3 a la var.

st %r3, [z] → vuelve a la prox pos de donde fue llamado (return)

jmp l %r1+4 → si no pongo el +4 iría a donde me llamaron y se crearía un loop

X: 15 } declaración de variables

Y: 9 }

Z: 0 }

end

entre operación y operación hay un espacio de 4 bytes

## Declaración de array

data: arrayA: dw8 5

reserva 5 lugares de 4bytes en total por lo tanto su contenido

representa

251-10181-517

→ Nunca sabemos si los registros que vamos a usar estén ya con contenido o los inicielizamos en cero.

andcc %r3, %r0; %r3 ! pone en 0 el r3

andcc %r1, %r1, %r0 → da como resultado r1 es cero → si no el r1 no es cero → para controlar que sea nula ponemos "be"

① Pregunta → Al ejecutar cuando el resultado de una operación es diferente de cero

si en vez de guardar un valor como variable la guardo como constante  $\Rightarrow$  no tengo que ir a buscarlo a memoria RAM  $\therefore$  es rápido el programa. (eficiente)

Opciones de ld lee memoria (los datos NO están en memoria)

ld [x], %r<sub>1</sub>  
ld (%r<sub>1</sub>, %r<sub>2</sub>)  
ld (%r<sub>1</sub>, %r<sub>3</sub>)  
ld (%r<sub>1</sub>, [Array], %r<sub>2</sub>)

largo un arreglo  $\rightarrow$  Array ->  $\frac{10 \cdot 3 \cdot 4 \cdot 3}{4 \cdot 8 \cdot 12 \cdot 16} \rightarrow$  largo del arreglo en bytes  
 $r_1 = 20 \rightarrow$  largo del arreglo  
quiero acceder al contenido del arreglo  
 $\Rightarrow$  ld %r<sub>1</sub>, [arreglo], %r<sub>2</sub>  
largo en  $r_2 = -5$ , o sea el contenido del arreglo en la pos. de r<sub>1</sub>.

Subrutinas  $\rightarrow$  invocamos código como una instrucción (función o procedimiento) NO tiene uso global y extern.

Convenciones de llamada (métodos para pasar argumentos hacia y desde la rutina invocada)

- colocar los argumentos en los registros.
- colocar los argumentos en una pila del tipo LIFO (último)
- colocar los argumentos por apelida reservada en memoria (no es dinámico)

### Ejemplo

! Programa que suma dos n:

begin

ei call org 2048

tiene { los parámetros }  
el 1.1.1.1 ld [2], %r<sub>1</sub> } que se pasa  
de 12 . . . . . ld [2], %r<sub>2</sub> } bien lo que  
fincall sbr-add } nombre  
que llamo st %r<sub>3</sub>, [2] } antes del call  
a esta } y luego de diferencia que se  
se pasa sbr-add %r14, %r6 } pasen  
el call } en las celdas  
de sbr-add !

! sume el cont de r<sub>1</sub> y r<sub>2</sub>

! devolver el resultado r<sub>3</sub>.

sbr-add : addcc %r<sub>1</sub>, %r<sub>2</sub>, %r<sub>3</sub>

jmpl %r<sub>3</sub>, %r<sub>0</sub>

x: 15

y: 9

z: 0

end

multis

! Prog que suma dos números

begin

ld [x], %r<sub>1</sub>

ld (%r<sub>1</sub>, %r<sub>2</sub>)

addcc %r<sub>14</sub>, -4, %r<sub>14</sub>

st %r<sub>1</sub>, %r<sub>14</sub>

addcc %r<sub>14</sub>, -4, %r<sub>14</sub>

st %r<sub>2</sub>, %r<sub>14</sub>

call sbr-add

ld %r<sub>14</sub>, %r<sub>3</sub>

addcc %r<sub>14</sub>, 4, %r<sub>14</sub>

st %r<sub>3</sub>, [2]

jmpl %r<sub>15</sub>, 4, %r<sub>0</sub>

habilita largo

en el stack  
(en donde está la  
resto de pos.)

guarda el dato  
en el stack

guarda el resultado  
en r<sub>3</sub>

guarda el valor  
de r<sub>3</sub> en Z

este rompe/ciclo  
deben ejecutar  
la del add 1.1.1.  
en la pila de  
cambios y hasta  
1000 ejecuciones  
del stack (lo ejecuta)

guarda en r<sub>14</sub>  
y la suma de  
r<sub>6</sub> y r<sub>7</sub>

guarda en r<sub>14</sub>  
y la suma de  
r<sub>6</sub> y r<sub>7</sub>

pasa a ult. valor de  
stack con r<sub>14</sub>

return

! subrutina sbr-add

sbr-add : ld %r<sub>14</sub>, %r<sub>0</sub>

addcc %r<sub>14</sub>, 4, %r<sub>14</sub>

ld %r<sub>14</sub>, %r<sub>6</sub>

addcc %r<sub>6</sub>, %r<sub>7</sub>, %r<sub>14</sub>

st %r<sub>14</sub>, %r<sub>14</sub>

jmpl %r<sub>15</sub>, 4, %r<sub>0</sub>

Si en vez de pasar constante  $\rightarrow$  no tiene el programa.

## Opciones de ld

- ld [x], %r1 → Carga la variable x en %r1,
- ld %r1, %r2 → dirección de memoria, dirección de lectura
- ld %r1, %r2; !r2 → suma los contenidos de los registros
- ld %r1, [Array]; !r2 → el resultado de la suma se guarda en r2
- ld %r1, [Array]; !r2 → se suma r1 a la dirección Array
- ld %r1, [Array]; !r2 → el resultado se guarda acc.

Subrutinas → invocamos código como una instrucción (función o procedimiento). No tiene etiqueta global y externa.

Convenciones de llamada (métodos para pasar argumentos hacia y desde la rutina invocada)

- ↳ Colocar los argumentos en los registros.
- ↳ Colocar los argumentos en una pila del tipo Local (dinámica).
- ↳ Colocar los argumentos en una pila del tipo Global (estática).

### Ejemplo

! Programa que suma dos nrs:

begin

el call .org 2048

lne

el 7f11 ld [r1],%r1 ;(los parámetros)  
de lo .ld [r2],%r2 ;que se pose  
fueran ;tén los parámetros  
de lo st %r2,[r2] ;entre los coll  
2 este y luego se devuelva el resultado  
Se pasa ;jmpl %r1,%r0 ;para el  
el call ;subroutine sbr-add

! sume el contenido de r1 y r2

! devolver el resultado r3

sbr-add : addcc %r1,%r2,%r3

jmpl %r3,%r0

x: 15

y: 9

z: 0

salir

! Programa que suma dos números

begin

.org 2048

ld [r1],%r1

ld [r2],%r2

addcc %r1,%r2,-4,%r4

st %r1,%r4

addcc %r4,-4,%r4

st %r2,%r4

call sbr-add

ld %r4,%r3

addcc %r4,4,%r4

st %r3,[r2]

jmpl %r3,%r0

! Subroutine sbr-add

sbr-add : ld %r1,%r0

addcc %r1,%r0,4,%r4

ld %r1,%r0

addcc %r0,%r0,%r0

st %r0,%r4

jmpl %r0,%r0

! Subroutine sbr-add  
en el stack  
(en donde está la  
resto de los )

guardar el dato  
en el stack

guardar el resto  
en r3

guardar el resultado  
de r3 en r2

! Subroutine sbr-add  
diseño de la  
pila del stack  
en el caso de  
una función

! Subroutine sbr-add  
guardar en r0

! Subroutine sbr-add  
la suma de  
r0 + r1

! Subroutine sbr-add  
stack anterior

x: 15  
y: 9  
z: 0

salir

- Marcos → bloques de instrucciones que cuando se ejecuta se "copia y pega".  
 → no es instrucción de ensamblador.

### ! Pega con macros

- begin

- org 2048

- :

- ld [x], %r1

- ld [y], %r2

- push %r1

- push %r2

- call Starr-add

- pop %r3

- st %r3 [z]

- :

↓ Sube los datos al stack  
 cada vez que aparece  
 push se repite este cod:

macro add1  
 addcc %r1,%r2,%r3  
 ret  
 endmacro

↓ lo mismo pero borra el resto del stack, y devolviendo la pila

ld %r1,%r3  
 addcc %r1,%r2,%r3

### Macro

- Se accede en tiempo de ensamblado (compilación)

- En tiempo de ejecución es mucho más rápido ya que no pega saltos ni vueltas.

- Su código es repetido la cant. de veces que es llamado, → ocupa mucha más espacio.

### Subrutina

- Se accede por un call en tiempo de ejecución y termina con jmpf.

- su código de máquina está en un lugar específico y único en memoria por lo cual no ocupa más espacio que ese.

- su tiempo de ejecución es más lento a causa de los saltos y returns.

### ! Suma dos numeros con macro

- begin

- macro macr-add Reg1, Reg2, Reg3.  
 add Reg1, Reg2, Reg3

- end macro

- org 2048

- ld [x], %r1

- ld [y], %r2

- macr-add %r1, %r2, %r3

- st %r3, [z]

- jmpf %r1 + 4, %r0

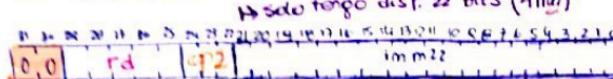
los registros siempre se mencionan en función de su contenido, nunca en términos de una dirección, → no hay necesidad de encerrar entre llaves las referencias a los registros.

# CÓDIGO DE MÁQUINA

Todos las instrucciones ocupan 32 bits

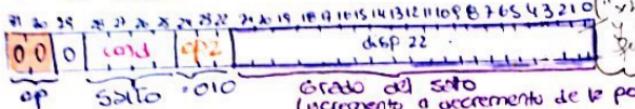
No Todos las instrucciones poseen el mismo formato  
Formatos de instrucción

## • sethi



op n de registro 100 (voy x posiciones hacia atrás, ej)  
de destino

## • BRANCH



op Salto = 010 Grado del salto  
Incremento o decremento de la pcp

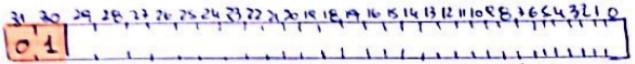
Jugando  
posibles

•  $2^{24}$  saltos posibles (16 Mb)  $\rightarrow$

• este valor lo pone el ensamblador  $\rightarrow$  op2

1. En se fija que  
no de cambio  
2. CS  $\rightarrow$  [00] : op  
(aparte pcse).  
2. Es fijo si  
es sethi o  
branch segun  
el op2 (ajustar  
el anio).  
Juego, depende  
de lo que es, mib  
er rd o and  
mido en pcp.  
bits de 4 bits  
4 x disp 22  
2. shift  
left.

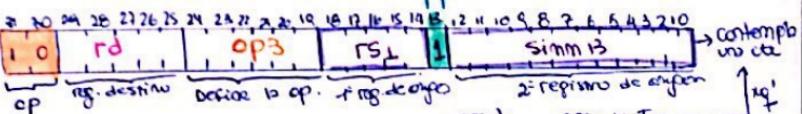
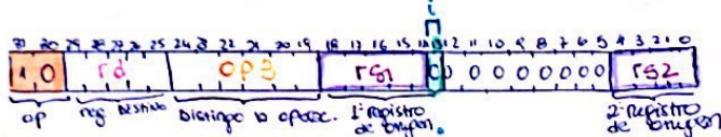
## • call



op  $\rightarrow$  desplazamiento a la subrutina  
(dirección a la cual se pone saltar)

• constante de 32 bits  $2^{30} \times 4 = 2^{32} \rightarrow$  puede saltar  
a cualquier  
lugar medida en memoria

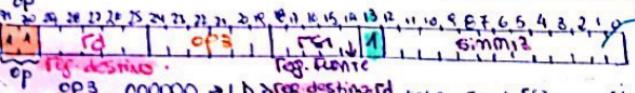
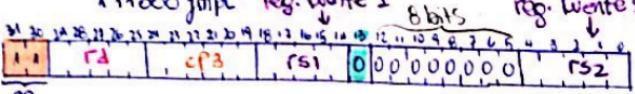
## • Funciones aritméticas



op3:  
01000 addcc  
01001 andcc  
010010 orcc  
010100 andcc  
100110 srl  
110000 jmpl

Reg. Fuente 1 8 bits Reg. Fuente 2  
si: 1 = 1 los 13 bits menos significativos  
se interpretan como conjunto  
1 = 0  $\rightarrow$  No es una operación inmediata

## • Memoria



op3:

00000  $\rightarrow$  LB  $\rightarrow$  reg destino  
000100  $\rightarrow$  ST  $\rightarrow$  reg origen

en este caso se  
usaron 2  
variables

Lugar de  
la cte

Formato de datos en ARC (hay 12 formatos de datos dif)

↳ Se agrupan en 3 tipos: entero signado, entero sin signo y punto flotante. Dentro de estos tipos, los tamaños admisibles son byte (8 bits), media palabra (16 bits), palabra (32 bits), palabra rotulada (32 bits, de los cuales los dos bits menos significativos forman un rótulo o referencia, en tanto que los 20 bits restantes forman el valor), palabra doble (64 bits) y palabra cuadriplic (128).

Modos de direccionamiento → de que manera se direccionan los datos (14)  
(en una instrucción de assembler la dir. donde están los datos)

Modos (opciones que tengo en cuenta a los operando)

• Inmediato: constante incluida en la instrucción, es el nº de registro. El registro al que apunta es inmediato.

Por Registro: el registro tiene el dato.

Directo o absoluto: dir. de memoria, incluida en la instrucción. Es el dirección del memoria.

Indirecto: dir. de memoria donde está el puntero del dato (lento) → en computación

Indirecto por registro: el registro tiene el puntero del dato.

Indirecto por registro: Un registro de la dir. inicial el otro un incremento (arrays).

Set de instrucciones = Conjunto de instrucciones + Registros disponibles

↳ Características:

- Tamaño de instrucciones (tamaño del nº de registro)
- tipo de operaciones admitidas
- tipos de operandos y de resultados
- formas de indicar la ubicación de los datos → modo de direccionamiento.

Set de instrucciones

### Aritmética/Lógica

ADD  
SUB  
ADDCC  
SUBCC  
AND  
AND N → AND Negado  
ANDCC  
ANDNCC  
OR  
ORN → OR Negado  
ORCC  
ORNCC  
XOR  
XNOR  
XORCC  
XNORCC  
Sethi



Se utiliza para guardar en los registros 22 bits

• q: las instrucciones son todos de 22 bits ⇒ ya ocupa espacio sobre que instrucción es, y queda menos espacio para las otras o variables

### Control

BRA → branch always  
BNE → si es ≠ de cero.  
BE → si es = a cero.  
BO → si es mayor (branch if greater)  
BLE → si es menor o =.  
BGE → si es mayor o =.  
BL → si es menor.  
BGT → si es mayor } N = sin signo  
BLT → si es menor o = } N = sin signo  
BCC → carry clear (carry = 0)  
BCS → carry SET (carry = 1) → da carry  
BF0 → si es positivo.  
BNEG → si es negativo.  
BNC → overflow clear (overflow = 0)  
BVG → overflow set (overflow = 1) → da overflow  
call  
jmp

- U → Unsigned → No signado.
- los que comienzan con "B" son saltos condicionales (versiones de IF's)

## Desplazamiento

SRL → lógico

SRA → aritmético

SLL → a izquierda

si signado

$$1\ 000 \rightarrow -8$$

Agrego el 0 adelante

$$0\ 1000$$

Busco su CB

$$CB = 20111$$

$$+ \quad \underline{1}$$

$$CB = \frac{1}{4}000 \rightarrow -8$$

significado

desplaza un registro hacia  
la derecha y carga ceros  
en los bits de significado

a derecha

desplaza un registro hacia  
la derecha y almacena una  
como del bit de significativo  
que contiene el signo en los  
bits vacíos (extensión del signo)

Ahora si hago un  
desplazamiento a derecha  
extiendo encontré con  
el -4

Supuestamente  $1111 = -8$

SRL

$$\begin{array}{r} 1111\ 0000 \\ 0000\ 1111 \end{array} \rightarrow \text{NO es } -4$$

SRA  $\begin{array}{r} 00001111 \end{array} \rightarrow \text{es } -4 \rightarrow \text{copia el signo que tiene } (-)$

## Acceso a memoria

ST → carga a registro 32 bits

STB → same con los 8 bits menos  
significativos (de 1 byte)

STH → half → guarda los 16 bits  
menos significativos

LD → carga una variable

LSB → copia 1 byte.

LSH → signados

LWB → no signados

## RISC VS CISC

Arquitectura del set de instrucciones

RISC → Reduced instruction set computer.

CISC → complex instruction set computer.

Set de instrucciones de µP



- ✓ circuito interno
- ✓ software de aplicaciones
- ✓ Performance

RISC → filosofía de diseño

(U.C. reduce en memoria de datos) → ciclo de reloj corto compleja una instrucción errores de memoria en lectura/escritura o como argumento de operaciones registradas

④ Cont de registros disponibles

Ahorro de instr. redund.

### Reduced vs. complex

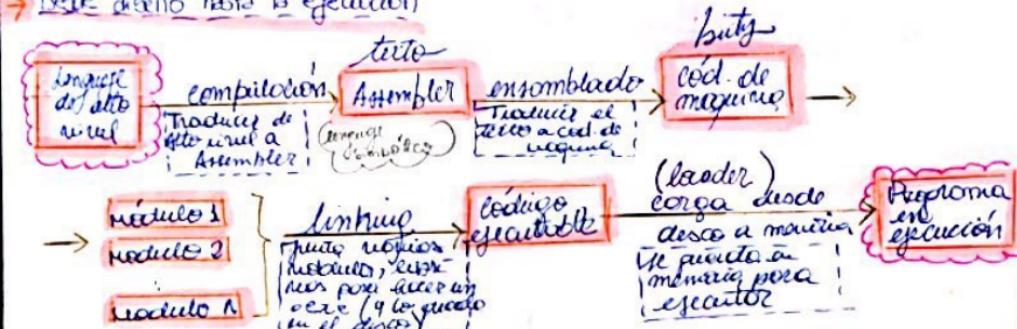
- Dec → decrementa 1 registro o memoria  $\Rightarrow$  Restar 1
- NOT → complemento a 1  $\Rightarrow$  NOR (o sea negamos bit a bit)
- Neg → complemento a 2  $\Rightarrow$  NOR + 1 (negamos todo y sumamos 1)
- Test op1, op2 → AND que solo altera flags  $\Rightarrow$  los resto y veo los flags (saber cuál es mayor en los flags)
- Push → Subir a la pila  $\Rightarrow$  con un SP → incrementar el tamaño de la pila y guardar algo allí.
- Pop → Bajar de la pila  $\Rightarrow$  con un SP → guardamos el valor y decrementamos el tamaño de la pila.
- IN, OUT → acceso a periféricos  $\Rightarrow$  toma los periféricos como dir de mem
- MOV reg, reg<sub>2</sub> → copia lo que está en el reg1 en el reg2  $\Rightarrow$  OR 0

RISC → Ahorro de instrucciones redundantes.

- ④ cont de registros disponibles.
- Dispositivos mapeados en memoria o en mapa indep.
- Ciclo de reloj para completar una instrucción.
- Accesos a memoria a lectura/escritura o como argumento de operaciones.

### 3 El lenguaje y la máquina

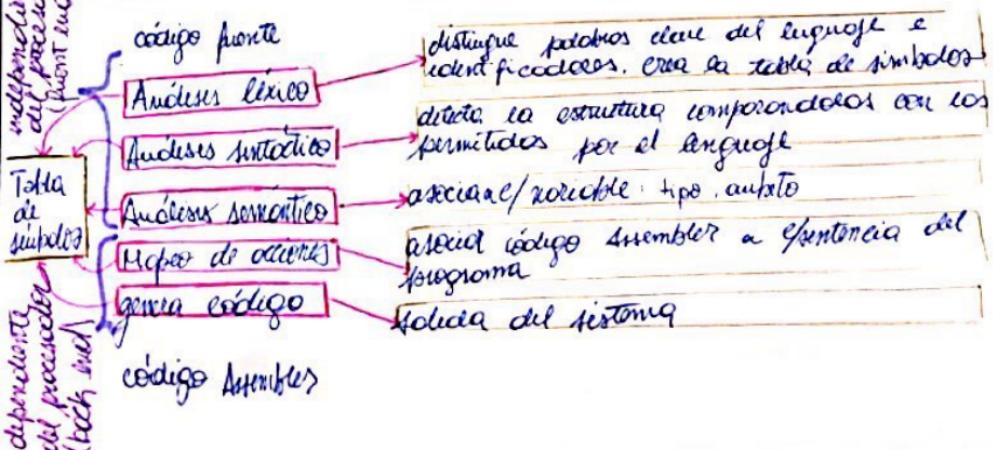
→ Desde diseño hasta la ejecución



### Compiladores (Analiza el Texto)

- Compilador de **una sola pasada** (se lee linea, lo que hace, luego a la sig y se lo que hace, hasta ver todos los lineas y al final genera el Assembler) o de **pasadas múltiples** (en cada pasada hace cosas diferentes).
- **Compilador incremental** (genera el código instrucción por instrucción cuando el usuario aprieta una tecla, se va hasta el final ... [python, Node.js]).
- **Cross-compilador** (genera cod para una maq. diferente).

### Proceso de compilación



código fuente

ejemplo:

Tiempo Total = Precio + 5

→ no sabe lo que  
es "total" ni  
"precio"  
→ lo pone

Añadiremos lógica

comparar los símbolos horarios

del compilador (fijo si los precios)

se sincronizan al tiempo de preparación

Añadiremos sintáctico

asigna a id1 = "expresión"

→ recorre sentencias (un "nuevo" "expresión" = id2 → etc)

selecciona y estructura los precios

Añadiremos semántico

tabla de símbolos  
atributos a símbolo

si llueve hasta acá → interpreta lo que el programador  
quiere hacer

A partir de ahora lleva a cabo lo que se espera!

Mapeo de Acciones

- código Assembler  
- localización de los símbolos  
- uso de registros

Tiempo una linea?

de código → lo lleva

a Assembler como 1 linea

o (P), y luego con todos

los líneas ya pasadas a

el código requerido

generación del código

sólida del sistema;

→ Ensamblador (relación 1)

Mapeo de Acciones

Estructuras de control de flujo del programa

Dato statement	Dato ítem
if (a > b) then c else d;	en Asembler: a>b jez... ;... código de if b... ;... código de else d... ;... código de else fini ;...
while (a == b) a = a + 1;	na add a, 1 test a, b jne a, b ;... y summa b a en memoria
for (a = 1; a <= b; a = a + 1)	test add a, 1 jle a, b add a, 1 ;...

obtención del código de máquina

• begin

• op 2048

main : ld [x], %r1	! load x into %r1.
ld [y], %r2	! load y into %r2
addc %r1, %r2, %r3	! %r3 ← %r1, %r2
st %r3, %r0	! store %r3 into %r0
jmpl %r1+4, %r0	! Retorno.

2048

2052

2056

2060

2064

• Representación simbólica para direcciones y etc

• Definir la ubicación de variables en memoria

• Variables inicializadas antes de ejecución

• Provee cierto grado de eficiencia en tiempo de ensamblado

• Utilizar variables declaradas en otros módulos

• Unidad Macros

assembler y cod. de máquina)

ofrece al programador

- Representación simbólica para direcciones y datos
- Definir la ubicación de variables en memoria.
- Variables inicializadas antes de ejecución
- Provee cierto grado de aritmética en tiempo de ensamblado
- Utilizar variables declaradas en otros módulos
- Utilizar macros.

2048

2052

2056

2060

2064

x: 15 2068  
y: 9 2072  
z: 0 2076

• end.

acceso a memoria

⇒ ld [x], %r1 → formato de → (1,1) 000001 | 000000 | 00000 | 1 | 00000000000000000000000000000000

con este formato siempre se "suman" cosas → en vez de nulos como ld [x], r1

el ensamblador lo ve como

ld r0, [x], r1 suma 0 a x para quitarlo en el reg. 1 = 2

rd op rd op<sup>3</sup> rs1 i simm3  
registro destino  
rs1 = 1  
me sumo  
quedo en r0, etc.  
en r0, etc.  
una variable  
posición de la etc

sigue una etc  
2068

⇒ ld [y], %r2 → formato de acc. → (1,1) 000002 | 00000 | 00000 | 1 | 01000000000000000000000000000000

registro  
etc a destino  
memoria r2

sigue una etc 2072

le sigue el r0 a la etc  
posición de la etc y

### Proceso de ensamblado en 2 pasadas

- Preproceso: expansión de macros ⇒ registros definiciones, reemplazar
- Primer pasada
  - Detecta identificadores y les asigna una pos de memoria
  - crea el tablo de símbolos. → 

Simbolo	Valor	global/externo
:	:	tablo de simbolos
- Segunda pasada
  - cada instrucción es pasada a cod. de máquina
  - cada identificador es reemplazado por su ubicación en memoria según la tabla de símbolos.
  - cada linea es procesada completamente antes de avanzar a la sig.
  - genera el cod. objeto y el listado.

↳ binario (.bin)

cod. de máquina  
op rd op<sup>3</sup> rs1 i ...

### Localización del programa en memoria

- En general no se sabe donde va a ser cargado el program, ya que la localización 2048 puede estar ocupada. ⇒ K debe hacer un cod. relocable
- En la tabla de símbolos se marca si la dirección es

Si hoy una  
Mañana cambia  
que especifica  
el código =>  
se reemplaza  
todo por los  
instrucciones  
específicas, y  
luego arranca  
el resto de ensambladores (pasadas)

relocalizable, o si es una dirección externa.

↓  
esto lo hace el ensamblador → por "R" como reubicable.

↓  
el int. op. es el responsable de guardar los funciones

Símbolo	Valor	Global/externo	Reubicable
---------	-------	----------------	------------

→ El linker une f subrutinos, módulos → estos sufren corriente (la tabla de símbolos debe actualizarse)

- ↳ Relocaliza los módulos combinando y recolocando las direcciones internas a cada uno para reflejar su nueva localización.
- ↳ Define en el módulo a cargar la dirección de la primera instrucción a ser ejecutada ("main").
- ↳ Resuelve referencias de memoria externa al módulo

↳ En un módulo

- ↳ Símbolos locales
- ↳ Hay ...
- ↳ Símbolos globales

- global → declara un símbolo global (puede usar)
- extern → utiliza un símbolo declarado en otro módulo

los  
directivos  
NO tienen  
significado  
en la memoria  
RAM  
solo se  
carga el  
programa  
para su  
ejecución

Ensamblador: marca direcciones como relocalizables o absolutas

Linker: Redefine las direcciones relocalizables a partir de la nueva dirección de origen

### Símbolos relocalizables

→ Símbolos relocalizables:

↳ main

→ NO relocalizables: di  
↳ nroables : tut

Main program		Subroutine library	
begin		begin	
.org 2048		CNE	eqz 1
extern sub		CNE	org 2048
main: .d	100	global sub	
call sub		sub: .global k23, k20, k21	
k1: .addk k23 + 4, k20		addk k23, CNE, k21	
x: 105		jmpk k23 + 4, k20	
y: 92		end	
.end			



Symbol	Value	Global External	Reloc. static
sub	-	External	No
main	2048	No	Yes
x	2048	No	Yes
y	2048	No	Yes

Main Program

Subroutine Library

↳ debe relocalizar todos los símbolos relocalizables.

relocable, o si es una dirección absoluta.

↓  
esto lo hace el ensamblador → por "R" como reubicable.

el lnt. op. es el responsable de guardar los funciones

Símbolo	Valor	global/externo	Definible

→ El linker une f subrutinos, módulos → estos sufren unión módula (la tabla de símbolos debe actualizarse)

↳ Relocaliza los módulos combinandoles y rectificando las direcciones internas a cada uno para reflejar su nueva localización.

→ Define en el módulo a cargar la dirección de la primera instrucción a ser ejecutada ("main").

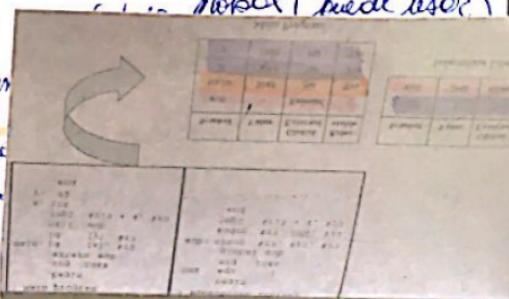
→ Resuelve referencias de memoria externa al módulo

↳ En un módulo símbolos locales  
hay ...      ↗ símbolos globales

• global → declara

• EXTERN → utiliza

... global (que lo necesita)



los directivos  
NO tienen!  
representación RAM  
cuando se corre el  
progr. para ejecución

Ensamblador: marca dirección  
Linker: Redefine las direcciones  
nueva dirección

→ Módulos relocables: pos. de memoria relativas a un org(x)

→ NO Relocables: direcciones de entrada/salida  
• variables  
• rutinas del sistema

→ Carga del programa en memoria

loader → busca el archivo .exe en el disco y lo carga en memoria principal, para luego ejecutarlo.

↳ debe relocate todos los símbolos relocables.

# Creación de la tabla de símbolos

(Se completa en la primera pasada)

(1)

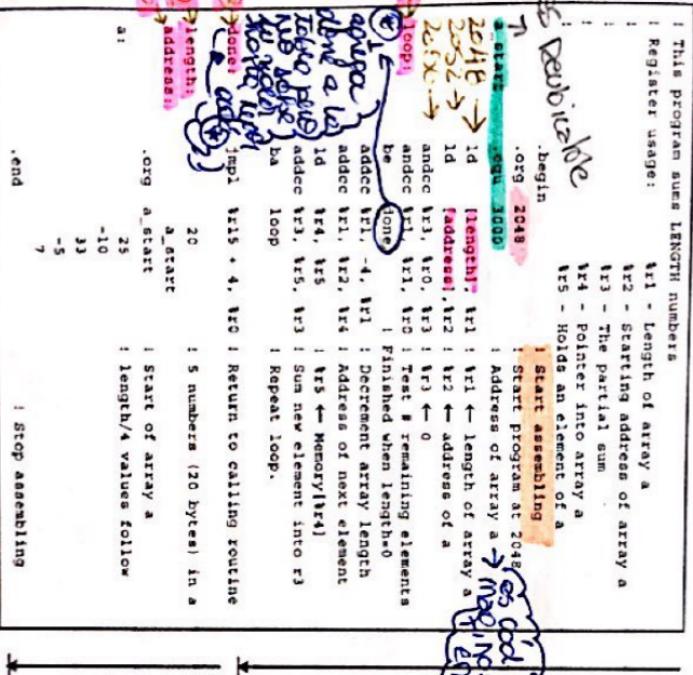
Símbolo	Valor
a_start	3000
length	-
address	-

La 1er pasada declara la dirección de memoria del código de máquina.

1er pasada

(2)

Símbolo	Valor
a_start	3000
length	2092
address	2096
loop	2060
done	2048



Contador de posición (análogo a un **program counter** en tiempo de ensamblado), si al final de la 1<sup>ra</sup> pasada quedan rotos los sin definir → existe un end en el propio archivo. Incrementado por cada instrucción según su tamaño (en ARC 4 bytes)

Muchos! No son variables!

No reconoce 'length' → se agrega al lo tabla más tarde con una nueva línea a donde fue declarado.

1 → en la 1<sup>ra</sup> pasada el ensamblador establece los rotulos No definidos y termina

1 → en la 1<sup>ra</sup> pasada el ensamblador establece los rotulos No definidos y termina

# Archivos creados por el ensamblador

(Se completa en la segunda pasada) genera el código objeto

## Salidas

### 1) Archivo de texto (listado)

### 2) Archivo con código objeto

- binario (latin)
- uso de numeros
- código dividido al final de cada linea

Firma parte del archivo de listado

Símbolo	Valor
a_start	3000
length	2092
address	2096
loop	3012
done	2088

Location counter	Instruction	Object code
.begin		
a_start .equ 3000	.org 2048	10000100 00000000 00101000 00101000
2048	ld [length], tr1	10000110 00000000 00101000 00101000
2052	id [address], tr2	10000110 10001000 11000000 00000000
2056	andcc tr3, tr0, tr3	10000000 10001000 01000000 00000001
2060 loop:	andcc tr1, tr1, tr0	10000000 10000000 00000000 00000010
2064	be done	00000010 10000000 00000000 00000010
2068	addcc tr1, -4, tr1	10000010 10000000 01111111 11111100
2072	addcc tr1, tr2, tr4	10001000 10000000 01000000 00000010
2076	ld tr4, tr5	10001010 00000001 00000000 00000000
2080	ba loop	00010000 10111111 11111111 11110111
2084	addcc tr3, tr5, tr3	10000010 10000000 01111111 11111100
2088 done: jmpl tr15+4, tr0		10000011 11000011 11100000 00000000
2092 length: .org		00000000 00000000 00000000 00000000
2096 address: .end		00000000 00000000 00001011 10111000
3000 a: 25		00000000 00000000 00000000 00011001
3004 -10		111111111111111111110110
3008 33		00000000 00000000 00000000 00100001
3012 -5		111111111111111111110111
3016 7		00000000 00000000 00000000 00000011

## Tabla de símbolos

## Listado

al final el ensamblador le debe agregar al módulo trazado:

- nombre y tamaño del módulo.
- dirección del símbolo de comienzo.
- información cerca de los símbolos globales y externos.
- información cerca de las rutinas de biblioteca o las que el módulo hace parte.
- información cerca de las rutinas que debe cargarse en memoria.
- los valores de cualquier constante que debe cargarse en memoria.
- los símbolos redefinibles se definen con "R".
- información de revisación. →

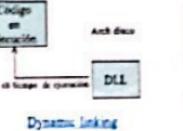
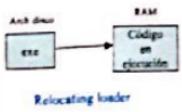
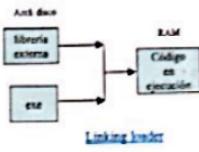
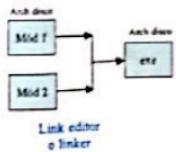
→ Link-editor: → linkea para generar un .exe en tiempo de desarrollo.

→ Relocating loader: → linkea en tiempo de carga.  
que es la cosa que el linker solo hace y no recibe y no

→ Linking loader: → loader que hace un proceso de buscas porque relocaliza.  
objetos? (objetos es módulos)  
carga los módulos en memoria y los ejecuta  
en memoria y llaman a ejecución  
→ linkea en tiempo de carga.  
→ Relocaliza, busca bibliotecas, linkea y carga el paquete en memoria para su ejecución.  
→ Busca referencias externas y resuelve desplazamientos de carga en memoria.  
Al momento de carga no tengo todas las bibliotecas, solo están declaradas → pero las necesito en memoria para ejecutar el programa → el loader carga a memoria TODAS las bibliotecas para que sea todo ejecutado.

→ Linking loader dinámico: → linkea en tiempo de ejecución.  
objetivos:  
no compila todo lo que no usa → carga rutinas solo cuando se ejecutan  
muestra al final los resultados → usa dynamic link libraries (dll).  
El sistema operativo quiere usar una función, no está cargada → la busca → la linkea → la ejecuta.

### Linking & loading



memoria  
bytes int float double

# ARQUITECTURA y Micro ARQUITECTURA

(19)

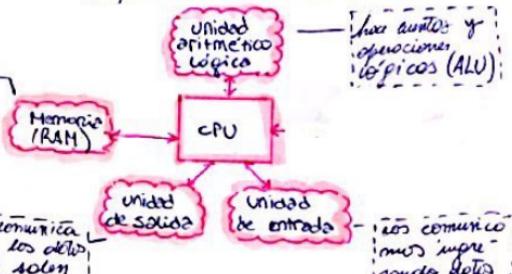
## Arquitectura de una computadora.

- lo que puede hacer una computadora es "ejecutar operaciones simples en una secuencia ordenada". (fue cálculo, y ejecuta algoritmos, que son pocos a seguir aritméticos y matemáticos)

la arquitectura  
interior, la del  
Harvard, no  
funciona S.O.  
• siempre  
es el mismo  
programa,  
no cambia  
nada. (p.e.:  
un ladrillo  
siempre  
funciona.)

no distingue entre  
memoria de datos  
(en donde están los  
números que usan  
para decirle de  
números y si quieren  
a medida que se  
ejecuta el programa),  
y memoria de  
instrucciones (dile  
que el programa  
que el dato  
que poner a seguir)

## Arquitectura von Neumann.

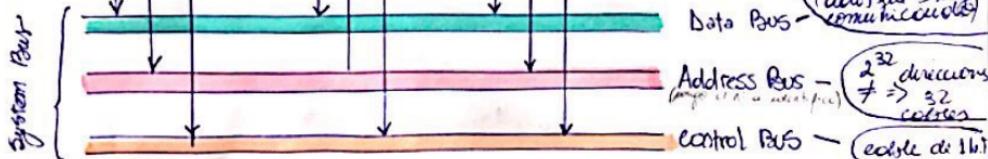
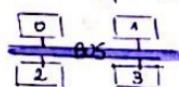


funciones y  
operaciones  
lógicas (ALU)

los programan los tiene como datos.

• podemos modificar programación  
sin tener que cambiar datos.

→ Corección de estructura tipo BUS → puente visto: reducir el n.º de cables  
necesarios para la comunicación entre  
los componentes, realizando comunicaciones a través de 3 solo cada de  
datos.



• hay  $2^3$  cables  
⇒ tienen 32 adresas  
(dato) que establecen  
comunicación

Address Bus - (longitud 32 bits)  
 $\neq \Rightarrow 32$  direcciones

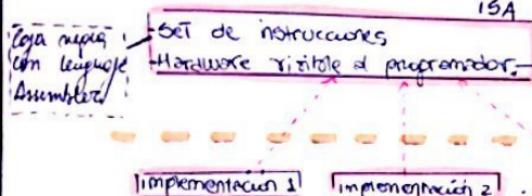
Control Bus - (cada bit 1bit)

Dirección: 32 bits  
Datos: 32 bits

Si el CPU quiere acceder a la memoria 2048, lo monolita al Pys (conjunto de cables). El n.º se recibe tanto por todos los posiciones de memoria (1-1) y por todos los dispositivos de entrada y salida (1-0). Pero hay 1 solo elemento que se identifica como 2048 ⇒ el elemento 2048 pone su dato en el Pys de 1 dato.

# Microarquitectura (Arquitectura del microprocesador)

La idea es integrar el CPU con la ALU. Los componentes de datos la tecnología que conocemos entran en un solo CHIP.

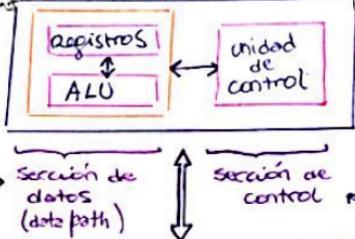


Registros que el programador tiene que usar

- Ejecutar un programa significa (ciclo de búsqueda - ejecución)
- Buscar en memoria la próxima instrucción a ser ejecutada.
  - Decodificar el código de operación de esa instrucción (CPU)
  - Ejecutar esa instrucción.
  - Volver a 1.

→ tiempo 2.75 lo que apunta PC  
Interpreta la instrucción (decod.) → todo se hace  
en forma de bits → la linea de la MDR realiza los operadores, control, lógicos

## Procesador

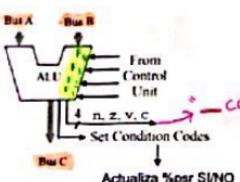


realiza la ejecución de todos los estados de las instrucciones (ciclo Fetch)

## TRAYECTO DE DATOS

- hay 32 registros posibles conectados a la ALU mediante buses. Así se puede elegir 1 registro de todos los que hay ya que la ALU tiene 1 entrada de operandos "A".

- La ALU puede realizar 16 operaciones sobre los buses A y B, y el resultado de 32 bits se coloca en el bus C, a menos que se quede bloqueada por el MUX como consecuencia de colocar en él una palabra proveniente de la RAM.



desplaza el contenido del RAM hacia la derecha en una cantidad especificada por el bus B. Se introducen 0's en los bits que quedan libres.

Recupera los 32 bits de información de los 32 bits y los lleva a los 32 bits que poseen en '0'.

produce un efecto de escritura de Apaga sobre los 50 bits significativos.

desplaza en 6 lugares hacia la derecha 32 operaciones del RAM, separadas en 5 bits. Se siguen los bits libres. El bit de dirección se compone

$F_1 F_2 F_3 F_4$	Operation	Changes Condition Codes
0 0 0 0	ANDCC (A, B)	yes
0 0 0 1	ORCC (A, B)	yes
0 0 1 0	NORCC (A, B)	yes
0 0 1 1	ADCC (A, B)	yes
0 1 0 0	SRL (A, B)	no
0 1 0 1	AND (A, B)	no
0 1 1 0	OR (A, B)	no
0 1 1 1	NOR (A, B)	no
1 0 0 0	ADD (A, B)	no
1 0 0 1	LSHIFT2 (A)	no
1 0 1 0	LSHIFT10 (A)	no
1 0 1 1	SIMM13 (A)	no
1 1 0 0	SEXT13 (A)	no
1 1 0 1	INC (A)	no
1 1 1 0	INCPC (A)	no
1 1 1 1	RSHIFT5 (A)	no

Todos los bits de salida de la ALU se crean una de 32 bits. Si el primero es 1, se crean 31 bits más. Si el primero es 0, se crean 31 bits más. Los 16 bits de los 32 bits de salida de la ALU se crean de acuerdo a los 16 bits de la dirección de memoria. Los 16 bits de la dirección de memoria se crean de acuerdo a los 16 bits de la dirección de memoria.

ejecución  
instrucción



controla la ejecución de todos los instrucciones (ciclo Fetch)

la ALU solo sume alimento de datos que vienen del registro  $\Rightarrow$  si quiero usar una este tipo:  
add %r1, 25, %r2  
el 25 si o sí lo guarda en un registro en el último momento

extensión del signo de los 13 bits → significado de la potencia

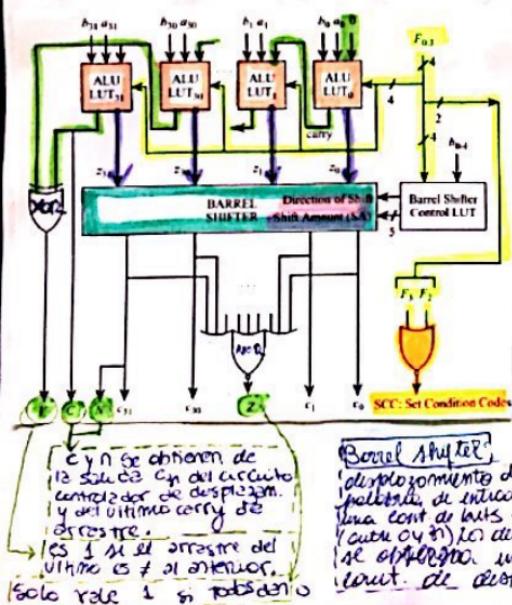
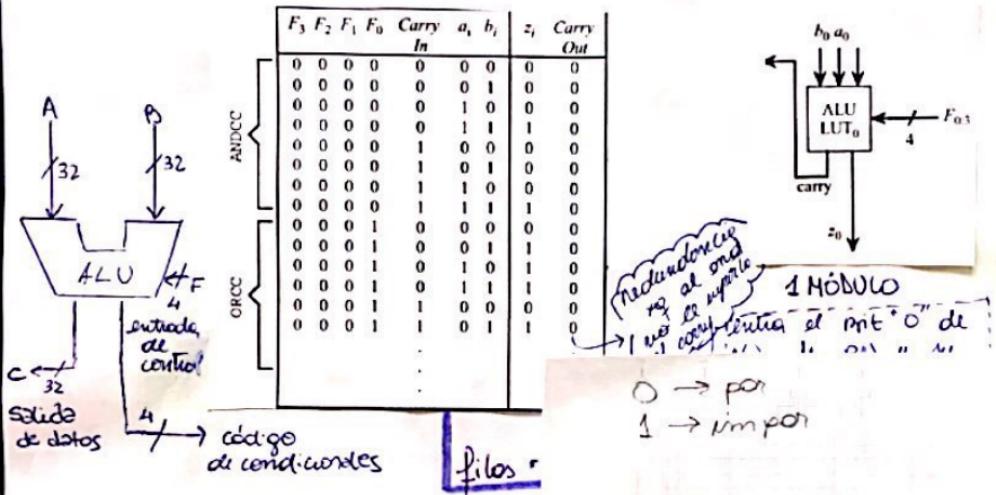
# → ALU

Es un circuito combinacional  
(si componen los estados combinan los resultados)

función: Mover datos y hacer operaciones aritméticas/lógicas  
¿cuales?

los operaciones binarias de ALU

Implementación: con 32 módulos de 1 bit puede obtenerse una alu de 32 bits.

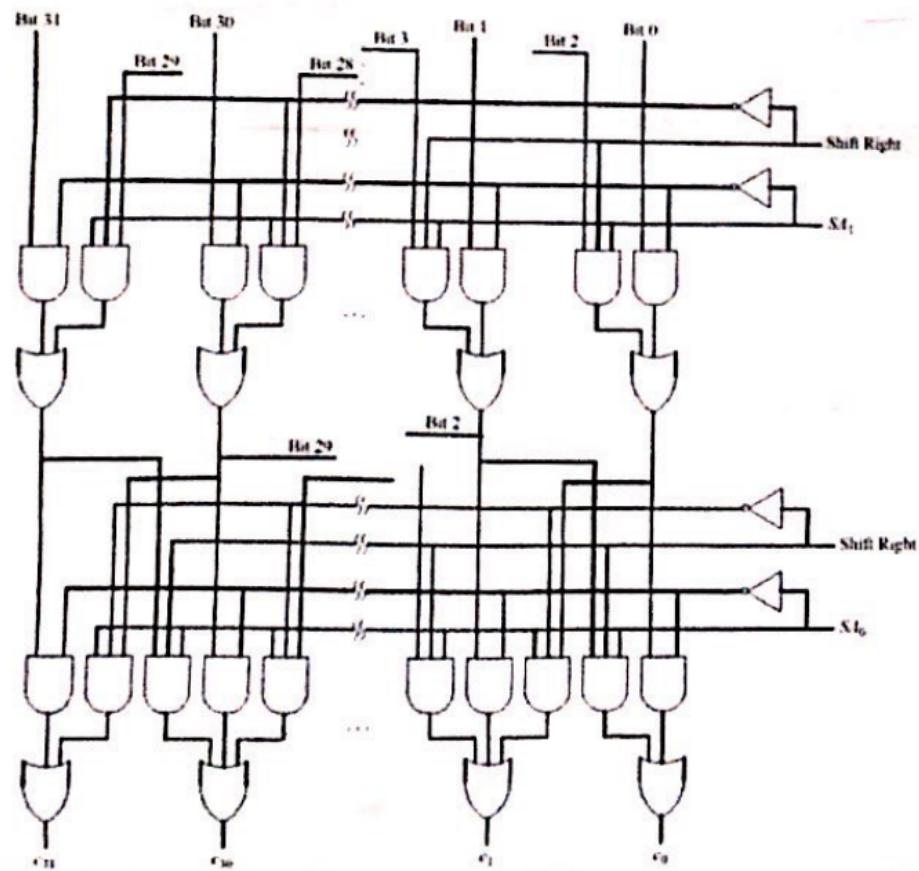


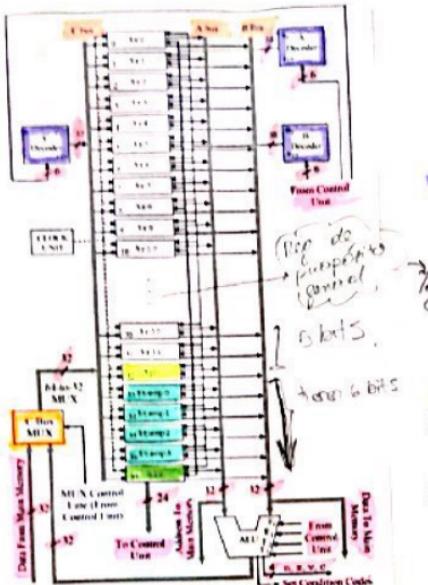
- = los 4 bits que indican qué operación (del 0 al 15) se debe resolver.
- = Carry (por si hacemos una suma).
- = Desplaza la cant de bits indicados.
- = indica el sentido (izq o der).
- = cant de posiciones a mover.
- = resultado de la operación.

**Barrel Shifter:** desplazamiento de la palabra de inicio en una cant de bits arbitraria. Una cant de bits arbitraria ( $0 \leq k \leq 15$ ) los desplazamientos se producen por niveles, y se aplica un bit  $f$  de le entrada que indica el rango de desplazamientos.

■ Saber si la operación termina en "cc" o no.

# Desplazador rápido (Barrel Shifter)





■ = es el contador que apunta a la dirección a ser leída en la RAM (salida va al bus de datos). (call, jump, etc: cuando entra dato la dirección / PC).

■ = decodificadores. Simplifican la elección de registros. La entrada de bits selecciona un único registro para c/u de los buses. En la entrada del decodificador/registro se coloca el n° de registro deseado.

■ = contiene la instrucción a ejecutar (no es acceso directo).

■ = NO son accesibles por el usuario. Se usan para interpretar el conjunto de inst. (son registros temporales para operaciones).

*Vale quería que el contenido del reg. 12 diera la clave de todos los registros menores al de 12, pongo un 5 en ese, y luego, esa linea, se conecta al ALU*

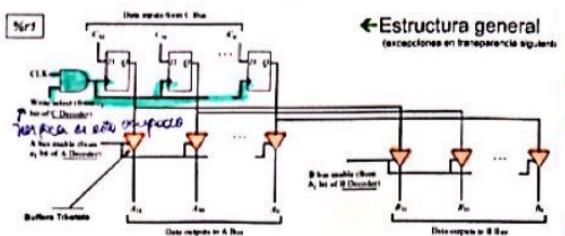
- Albytes  $\rightarrow$  32 bits. ( $3 \text{ bits} = 8 \text{ bits} \times 4$ )
- 6 bits  $\rightarrow$  32 reg + 32 reg =  $64 \text{ reg}$

$$2^{6 \text{ bits}} = 64$$

LA ALU genera los códigos de conexión. En el caso de tener flags se genera una señal SCC que le indica a %PSR que debe actualizar sus códigos de conexión.

## → Los registros

### Estructura interna de los registros



\* Los Flip-Flops son tipo D por flanco descendente

- Todos son implementados con circuitos biestables D activados por flanco negativo.
- todos adoptan un formato similar.
- Todos tienen 32 bits de ancho.
- El registro cero (%r0) siempre es '0' y no puede ser modificado. Este no tiene entrada de bus C ni desde el decodificador  $\Rightarrow$  no requiere FF.
- El registro no %r1 tiene salidas adicionales que corresponden a RS1, RS2, SP/OP2, EQ3 y el bit 13.
- El contador / PC solo puede contener valores que sean múltiplos de 4, por lo que los 2 bits C significan '0' siempre.
- Los índices mayores a 37 no corresponden a ningún reg.  $\Rightarrow$  pueden utilizarse cuando no se requiere una reg. en bus.

- los buffers si están en 1 permite leer lo que hay en A o B
- este AND permite y se asegura que negro solo cuando la sección de control lo determina

## → La sección de control

La figura muestra la totalidad de la arquitectura microprogramada de ARC. Ilustra el trayecto de datos, la unidad de control y las conexiones entre ellos.

**Memoria de Lectura (ROM)** está en el corazón de la unidad de control, de 2048 palabras de 41 bits. Esta contiene líneas que deben controlarse para implementar cada instrucción. Se considera memoria de control.

→ Cada palabra de 41 bits es una **Microinstrucción**. La unidad de control es la responsable de la búsqueda y ejecución de las microinstrucciones.

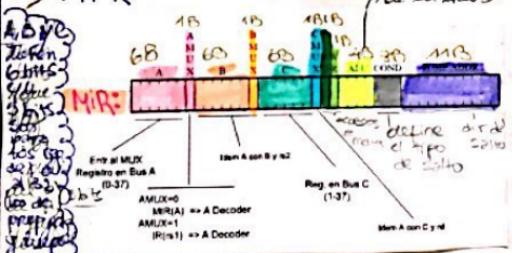
La ejecución de microinstrucciones se controla a través del registrador de instrucciones de microprograma **MIR**, del registrador de estado /<sub>PSR</sub> y de un mecanismo que permite determinar la próxima microinstrucción a ejecutar (formado por **CSL** y **MUX** de  $\overline{SL}$  de memoria de control). Se recalcula en **Ciclo de salto**. [Unidad de saltos de control]

Inicialmente se coloca la microinstrucción de la dirección o de la ROM en el MIR para ser ejecutada.

Luego se selecciona la siguiente microinstrucción de alguna de las entradas **NEXT** o **JUMP** sobre la base de lo que dice el campo **cond** del MIR.

Luego de colocar la palabra en el registro MIR, el trayecto de datos realiza las operaciones requeridas por los valores que adopten los diferentes campos del mismo registro.

## → MIR



- Campo **RD**: Determina si se tiene que leer ( $rd=1$ ) o no ( $rd=0$ ) en memoria.
- Tamb. controla al MUX C de 41 a 32 bits del BUS C, y determina si el BUS C se copia desde la memoria ( $rd=1$ ) o desde  $\overline{RD}$  ( $rd=0$ )
- Campo **WF**: Determina si se debe escribir ( $WF=1$ ) o no ( $WF=0$ ) en memoria.
- Campo **ALU**: Determina cual de los 16 operaciones va a ejecutarse. No hay forma de programarla. Si se desea ver se ejecuta una acción que no modifica nada.

- **Campo A**: Determina cual es el reg. cuyo contenido debe colocarse sobre el Bus A (parámetro de la operación).
- **Campo AHUX**: Selecciona si el decodif. A obtiene su entrada desde el campo A (**AHUX = 0**) o desde el campo **RS1** de MIR (**AHUX = 1**). → Si 0, **RS1** → **Bus A**
- **Campo B**: Determina cual es el reg. cuyo contenido debe colocarse sobre el Bus B.
- **Campo BHUX**: Determina si el decodif. obtiene sus entradas desde el campo B (**BHUX = 0**) o desde el campo **RS2** del y. ir (**B HUX = 1**).
- **Campo C**: determina en qué registro se almacena el dato transferido a través del **BUS C**.
- **Campo CHUX**: Elije si la entrada del decodificador C se obtiene desde el campo C (**CHUX = 0**) o desde el **rd** del y. ir (**CHUX = 1**).

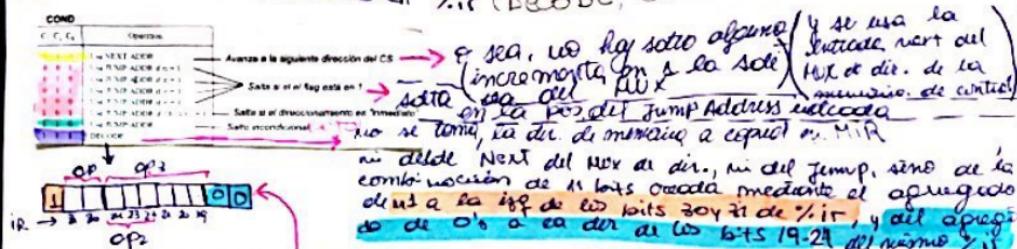
- campo jump addr. Son 11 bits de dirección para poder acceder a cualquiera de las posiciones de la memoria de control.

para otras operaciones  
esa dir de memoria se  
llama del PC A, el destino  
se ingresa en memoria bus B.  
y la otra se establece en  
el bus de control que se  
llama en este caso se  
llama en este caso se

$\Rightarrow$  Rd y WR No pueden estar a la vez con 1

- campo C013: hace que el microcontrolador cierre siguiente. ya sea desde:

- la posición sig. en ROM (NEXT)
- posición indicada en el jump address
- lo almacenado en %IR (decode, como si fuera 1)



## DURANTE LA DECODIFICACIÓN

SE ADOPTA ESTE VALOR

cuando el MUX

Cuando pongo R [registro(r<sub>i</sub>) ó temp 0 ó ir ó PC ó dir de memoria] MUX → 0  
Cuando pongo R [r<sub>si</sub> ó Rd ó un n {0, 1, ...}] ; MUX → 1

El Registro %IR contiene la instrucción que está siendo ejecutada. Los campos op, op1, op2 y op3 son los de esa instrucción.

cuando el reg. de instrucción

## Temporización

Se acomodan las secciones maestras



Tiempo de establecimiento de las secciones esclavas de los registros. Realizar operaciones aritméticas. Establecimiento de las banderas n, z, v, c

Microarquitectura opera con 2 fases

común las secc  
nreg maestras de los  
registros.

lo hace en la parte  
maestra se transf.  
a la parte esclava  
en el estado bajo se  
hace en el alto.

Las secciones maestras de los registros se cargan con el flanco creciente  
Las secciones esclavas de los registros se cargan en el flanco de caída

Figura 6.14 • Relaciones de tiempo para el funcionamiento de los registros.

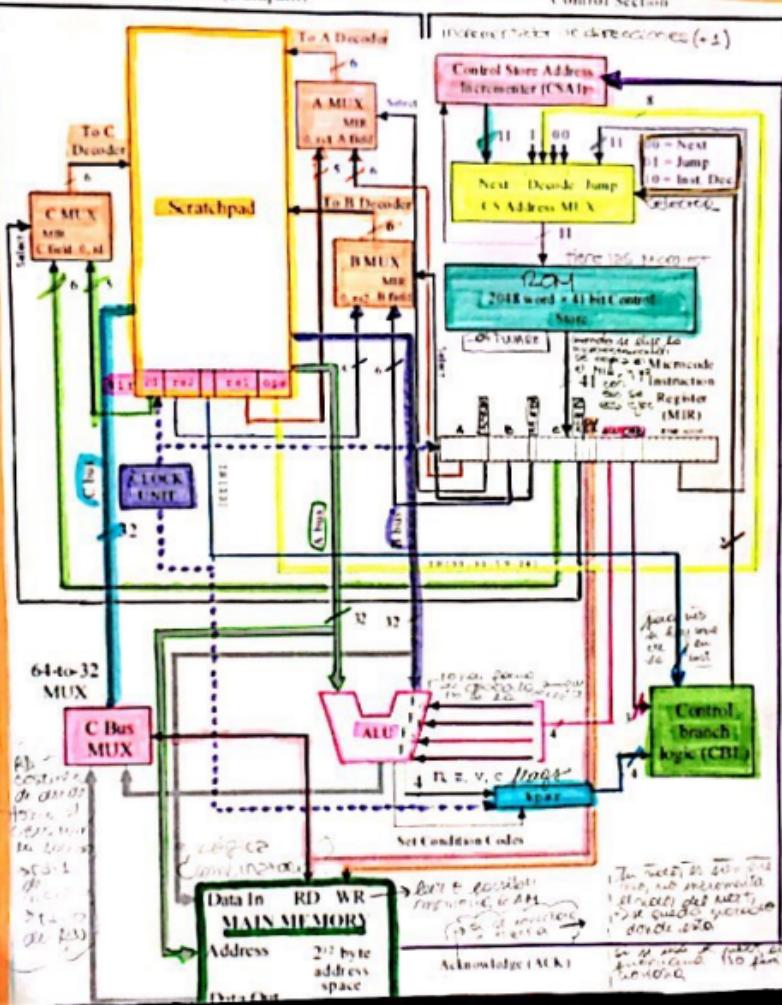
- de campo COND  $\rightarrow$  CBL  
3 bits - indican el salto
- de %PSR  $\rightarrow$  CBL - 4 bits  
de %IR  $\rightarrow$  CBL - 1 bit  
los 6 flags para poder resolver los jumps.
- de RAM  $\rightarrow$  CSAI  
Alfabeto de 6 bits  
Manda una señal de OK cuando se termina de R.D o W.R (reg. señales)
- de CBL  $\rightarrow$  MUX - 2 bits

Si dicen que x bit están conectados permanentemente a la tensión d alimentación d específico que ese bit es 4 siempre  
Is true, fija que es 1  
lo que:  $y = 1 \text{ (an)}?$

En operaciones de lectura y escritura  $\rightarrow$  la zona de dirección del PSR (A)  
 $\rightarrow$  el dato que se imprime es del tipo R  
 $\rightarrow$  Adelante de datos saldrá de memoria  $\rightarrow$  RUC

se recuerda la próxima instrucción de reloj  $\rightarrow$  NO hay que pausar futuras instrucciones

MUX9, deciden de donde a provenir su señal del campo C o del



- **Campo de registros (0-31), PC, (temp 0-3), IF**
- es el CSAI, que incrementa en 1 lo dir de memoria cuando el campo COND = 000
- es la memoria de control (ROM) que contiene 2048 microinstrucciones de 41 bits
- ALU, Realiza 1 de 16 operaciones sobre los buses A y B, y C
- %PSR, contiene los resultados de los flags de condición.
- es el MIR, controla la ejecución de las microinst. junto con %PSR, CBL y MUX de control
- memoria principal (RAM) contiene instrucciones de 32 bits direccionaladas (31 bits)
- es el CBL, se encarga de hacer los saltos condicionales obtenidos en el campo COND
- es el MUX del RUC, decide si tempr la info de la alu o de la RAM
- Mux de la ROM, decide que dir de memoria le envía a la ROM puede provenir de next, decode o jump
- es el clock, las microinst operan sobre un ciclo de 2 fases:  
F0  $\rightarrow$  combinan las 8 series mestres de los registros  
F0  $\rightarrow$  al mismo tiempo en lo mismo

- De campo COND  $\rightarrow$  CBL  
3 B - indican el salto
- De %PSR  $\rightarrow$  CBL - HB  
De %IR  $\rightarrow$  CBL - LB  
los 6 flags para poder resolver los jumps.
- De RAM  $\rightarrow$  CSAI  
Acknowledge  
Manda una señal de OK cuando se termina de RDO WR (xq' demora 0)
- De CBL  $\rightarrow$  MUX - 2B  
selecciona cual de las 3 entradas sale (según COND)
- De jump ADDRESS  $\rightarrow$  MUX (1B)  
si se produce un salto se manda la dirección
- Campos op's  $\rightarrow$  MUX dir - 8B  
en el caso de elegir una salida decodificada se usan esos campos.
- CSAI  $\rightarrow$  CS Add. MUX  
(entrada)
- De MIR  $\rightarrow$  ALU - 4 Bits  
La microinst. destinada a resolver x LA ALU
- MIR  $\rightarrow$  A MUX 6 Bits  
RS1  $\rightarrow$  A MUX 6B + 0  
son las entradas
- MIR  $\rightarrow$  WR Permite escribir en RAM, si es lo ejecutado
- HIR  $\rightarrow$  MUX's Select  
Determina que entrada pasa por los MUX(A,B,C)
- MIR  $\rightarrow$  B MUX - 6 B  
RS2  $\rightarrow$  B MUX 0 + 5B
- MIR  $\rightarrow$  C MUX - 6 B  
RD  $\rightarrow$  C MUX - 0 + 5B
- HIR  $\rightarrow$  RAM  
MIP  $\rightarrow$  CRUS MUX

Para poder leer se dirige a la RAM, y controla la 12 salida del MUX CRUS

## Dirección Sentencias operativas

Comentario	
R[rd] ← AND[R[pc], R[rd]];	/ Leer una instrucción de ARC desde memoria principal
R[rd] ← DCCD;	/ Salir (256 posibilidades) finalizando el código de operación
1157: R[rd] ← LSHP10(R[rd]); GOTO 2047;	/ Copiar el campo imm22 en el registro de destino
// end3	
1200: R[rd] ← AND[R[rd], R[rd]];	/ Guardar 0pc en R[15]
1201: R[temp0] ← ADD[R[rd], R[rd]];	/ Desplazar el campo dsgp30 a la izquierda
1202: R[rd] ← ADD[R[rd], R[temp0]];	/ Desplazar hacia la derecha
1203: R[pc] ← ADD[R[pc], R[rd]];	/ Salir a ejecutarse
GO TO 0;	
// add3	
1600: IF R[rd][13] THEN GOTO 1603;	/ El segundo operando origen está en modo inmediato?
1601: R[rd] ← ADDCC(R[rd]), R[rd]];	/ Resolver ADDCC sobre registros origen
GO TO 2047;	
1603: R[Tempo] ← SEXT13(R[rd]);	/ Obtener el campo sines13 con extensión de signo
1605: R[rd] ← ADDCC(R[rd]), R[Tempo]];	/ Resolver ADDCC sobre operando origen en registro/sinal13
GO TO 2047;	
1606: R[Tempo] ← SMM13(R[rd]);	/ Obtener el campo dsign13
1607: R[rd] ← ANDCC(R[rd]), R[Tempo]];	/ Resolver ANDCC sobre operando origen en registro/sinal13
GO TO 2047;	
// end3	
1608: IF R[rd][13] THEN GOTO 1606;	/ El segundo operando origen está en modo inmediato?
1609: R[rd] ← ANDCC(R[rd]), R[rd]];	/ Resolver ANDCC sobre registros origen
GO TO 2047;	
1610: R[Tempo] ← SMM13(R[rd]);	/ Obtener el campo dsign13
1611: R[rd] ← ORCC(R[rd]), R[Tempo]];	/ Resolver ORCC sobre operando origen en registro/sinal13
GO TO 2047;	
// end3	
1624: IF R[rd][13] THEN GOTO 1626;	/ El segundo operando origen está en modo inmediato?
1625: R[rd] ← PORCC(R[rd]), R[rd]];	/ Resolver PORCC sobre registros origen
GO TO 2047;	
1626: R[Tempo] ← SMM13(R[rd]);	/ Obtener el campo dsign13
1627: R[rd] ← NORCC(R[rd]), R[Tempo]];	/ Resolver NORCC sobre operando origen en registro/sinal13
GO TO 2047;	
// end3	
1688: IF R[rd][13] THEN GOTO 1690;	/ El segundo operando origen está en modo inmediato?
1689: R[rd] ← SRL(R[rd]), R[rd]];	/ Resolver SRL sobre registros origen
GO TO 2047;	
1690: R[Tempo] ← SMM13(R[rd]);	/ Obtener el campo dsign13
1691: R[rd] ← SRL(R[rd]), R[Tempo]];	/ Resolver SRL sobre operando origen en registro/sinal13
GO TO 2047;	
// end3	
1744: IF R[rd][13] THEN GOTO 1747;	/ El segundo operando origen está en modo inmediato?
1745: R[pc] ← ADD(R[rd]), R[rd]];	/ Resolver ADD sobre operando origen en registro/sinal13
GOTO 0;	
1746: R[Tempo] ← SEXT13(R[rd]);	/ Obtener el campo sines13 con extensión de signo
1747: R[pc] ← ADD(R[rd]), R[Tempo]];	/ Resolver ADD sobre operando origen en registro/sinal13
GO TO 0;	
// end3	
1792: R[Tempo] ← ADD(R[rd]), R[rd]];	/ Calcular dirección origen
IF R[rd][13] THEN GOTO 1794;	→ Si el 2º bit pertenece a una direc.
1793: R[rd] ← AND(R[Tempo], R[Tempo]);	/ Calcular dirección origen sobre el bits A y C que son el 1º y 2º bit de la dirección
READ; GOTO 2047;	Al resto del dato se le da el resto de los bits
1794: R[Tempo] ← SEXT13(R[rd]);	/ Obtener el campo sines13 para la dirección origen
1795: R[Tempo] ← ADD(R[rd]), R[Tempo]];	/ Calcular dirección de origen
GO TO 2047;	
// end3	
1808: R[Tempo] ← ADD(R[rd]), R[rd]];	/ Calcular dirección de destino
IF R[rd][13] THEN GOTO 1819;	
1809: R[rd] ← RSHP15(R[rd]); GOTO 40;	/ Mover el campo rd hacia la posición del campo rd
40: R[rd] ← RSHP15(R[rd]);	/ desplazándolo 25 bits a la derecha
41: R[rd] ← RSHP15(R[rd]);	
42: R[rd] ← RSHP15(R[rd]);	
43: R[rd] ← RSHP15(R[rd]);	
44: R[rd] ← ADD(R[Tempo], R[rd]);	/ Colocar la dirección de destino sobre el bits A y
WHRD; GOTO 2047;	/ el operando sobre el bits B
1810: R[Tempo] ← SEXT13(R[rd]);	/ Obtener el campo sines13 para calcular la dirección de destino
1811: R[Tempo] ← ADD(R[rd]), R[Tempo]];	/ Calcular dirección de destino
GO TO 1809;	
// instrucciones de salto: bz, be, beo, bne, beaq	
1818: GOTO 2;	/ Árbol de decodificación para saltos
2: R[Tempo] ← LSHP10(R[rd]);	/ Extender el signo de los 22 bits menores
3: R[Tempo] ← RSHP15(R[Tempo]);	/ significativas de 9 bytes, desplazando
4: R[Tempo] ← RSHP15(R[Tempo]);	7 primeros 10 bits a la derecha, luego 10 bits a la derecha
5: R[rd] ← RSHP15(R[rd]);	/ La extensión de signo se realiza a través de RSHP15
6: R[rd] ← RSHP15(R[rd]);	/ Mover el campo COND (R13) utilizando las veces RSHP15
7: R[rd] ← RSHP15(R[rd]);	/ La extensión de signo no afecta la operación
8: R[rd] ← RSHP15(R[rd]);	/ La instrucción es bz?
IF R[rd][13] THEN GOTO 18;	
18: R[rd] ← ADD(R[rd]), R[rd]];	
IF R[rd][13] THEN GOTO 19;	/ La instrucción no es bz?
19: R[rd] ← ADD(R[rd]), R[rd]];	
20: IF R[rd][13] THEN GOTO 21;	/ Ejecutar bz
21: R[rd] ← ADD(R[rd]), R[rd]];	
GO TO 2047;	/ El valor indicado por bz no se ejecuta
22: R[pc] ← ADD(R[pc], R[Tempo]];	/ Ejecutar el salto → El salto es siempre una cantidad de 8 bits de memoria
GO TO 0;	/ Es bz?
23: IF R[rd][13] THEN GOTO 16;	/ Ejecutar bz
16: R[pc] ← ADD(R[rd]), R[rd]];	/ El valor indicado por bz no se ejecuta
24: C THEN GOTO bz+4;	/ Es bz?
GO TO 2047;	/ Ejecutar bz
25: R[rd] ← ADD(R[rd]), R[rd]];	/ El valor indicado por bz no se ejecuta
26: GOTO 2047;	/ Ejecutar bz
27: IF R[rd][13] THEN GOTO 19;	/ El valor indicado por bz no se ejecuta
19: GOTO 2047;	/ Ejecutar bz
28: IF R[rd][13] THEN GOTO bz+4;	/ El valor indicado por bz no se ejecuta
26: GOTO 2047;	/ Ejecutar bz
29: R[pc] ← RSHP15(R[rd]); GOTO 0;	/ Incrementar 8 bits y ejecutar bz

## MICRO INSTRUCCIONES

+ estas 3 líneas se ejecutan en todos los microinstrucciones

1/000:

F102: [128] R[rd] ← AND(R[rd], R[pc]);  
281: R[Tempo] ← LSHP15(R[rd]);  
128: R[Tempo] ← ADD(R[rd], R[Tempo]);

GO TO 0;

D: R[rd] ← AND(R[rd], R[pc]); read; se carga el contenido de memoria en el R[rd] y se guarda otra lectura de memoria (dato para pág 84 Read) y la instrucción se ejecuta de nuevo porque el código almacenado en el R[rd] se lee, → cuando el código descodifica la pág 84 linea pr. el cod almacenado en R[rd]

pueden leer una pág 84 y leer en memoria temporal y lo guardan en R[rd]

La memoria es de 8 bits.  
Cada instrucción ocupa 32 bits (4 bytes)

R[Tempo] → no tiene ea tipo de cuentas bytes soltos, sino cuentas sucesivas.  
Como cada instrucción es de 4 bytes

R[Tempo] lo debes multiplicar x 4.

⇒ 11: R[Tempo] ← LSHP15(R[Tempo]) / 4 R[rd] Añade temporalmente los bits de bytes a soltos.

12: Al valor que tiene por lo tanto es el resultado de los bytes soltos.

13: Al valor que tiene por lo tanto es el resultado de los bytes soltos.

RSVA → Bus de dir. a memoria.

BUS P → Bus de datos a memoria.

BUS C → Bus de datos desde memoria.