

# **66.70 Estructura del Computador**

- 1) Arquitectura de una computadora**
  
- 2) Arquitectura de un microprocesador**

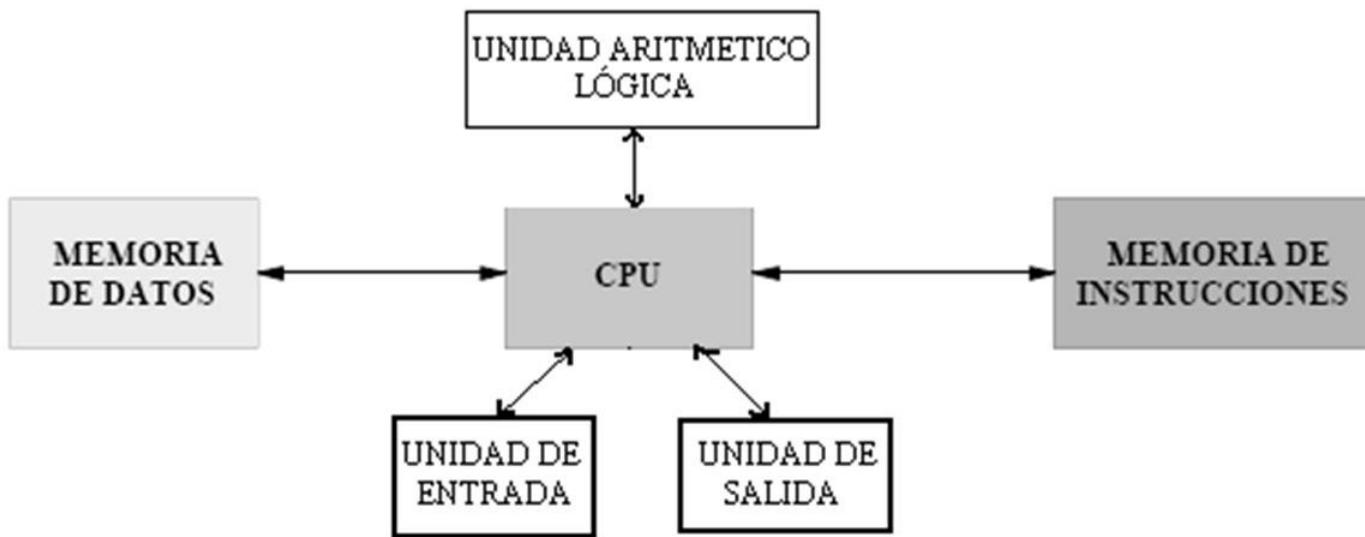
# **Arquitectura de una computadora**

# ¿Qué puede hacer una computadora?

- Las tareas que implementa pueden ser muy complejas
- Los modos de operación internos son sorprendentemente limitados

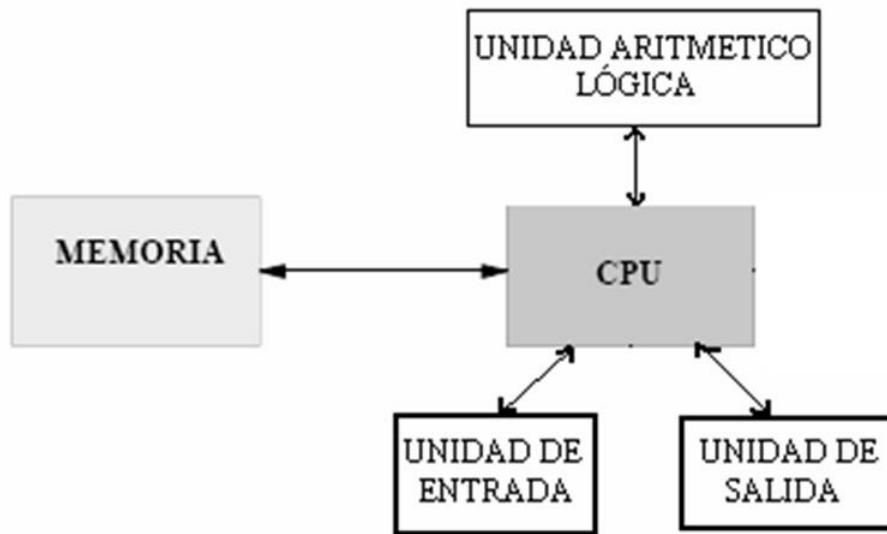
Esencialmente, lo que una computadora puede hacer es  
“ejecutar operaciones simples  
en una secuencia ordenada”

# Arquitectura Harvard



- ❖ Responde a la arquitectura de las primeras computadoras
- ❖ Se aplica actualmente en sistemas simples (p.e.: electrodomésticos)

# Arquitectura von Neumann



- ✓ Unifica el almacenamiento físico de datos y programa

# Arquitecturas von Neumann vs. Harvard

- ❖ Con la “arquitectura von Neumann” se logra modificar programas con la misma facilidad que modificar datos, permitiendo:
  - ✓ Fácil reprogramación, versatilidad característica de las computadoras actuales  
=> Compiladores, sistemas operativos, ...
  
- ❖ Alternativamente la “arquitectura Harvard” (datos y programa almacenados en ubicaciones físicas diferentes) es aplicada ventajosamente en sistemas electrónicos pequeños
  - ✓ Dispositivos automáticos que no requieren reprogramación (función única)
  - ✓ Pequeño volumen de memoria de datos y de programa

# El problema del conexionado

- ❖ ¿Cuáles y cuántos son los componentes externos con que debe conectarse el microprocesador?
- ❖ ¿Cuántos cables hacen falta?

Buscando una solución:

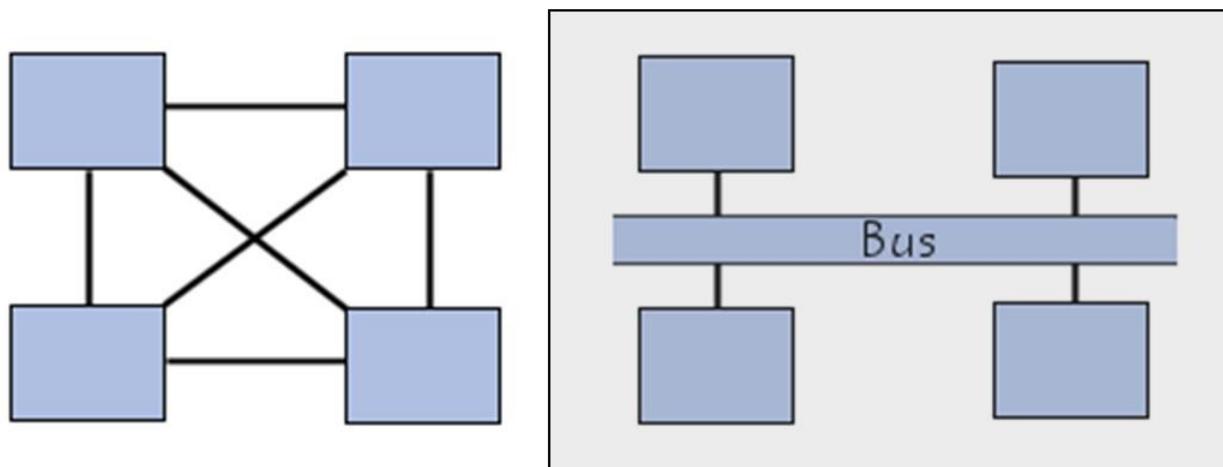
- ❖ ¿Deben estar conectados todos al mismo tiempo?

**Estructura tipo “BUS”**



# Conexionado con estructuras tipo BUS

Persigue el propósito de reducir el número de rutas necesarias para la comunicación entre los distintos componentes, realizando las comunicaciones a través de un solo canal de datos



- Bus →
- Líneas de datos
  - Líneas de direcciones
  - Líneas control

# Conexionado con estructuras tipo BUS

Un bus es un sistema de comunicación para la transferencia de datos entre componentes electrónicos

Permite conectar

- componentes de una computadora
- varias computadoras entre sí
- componentes electrónicos dentro de un microprocesador

Componentes

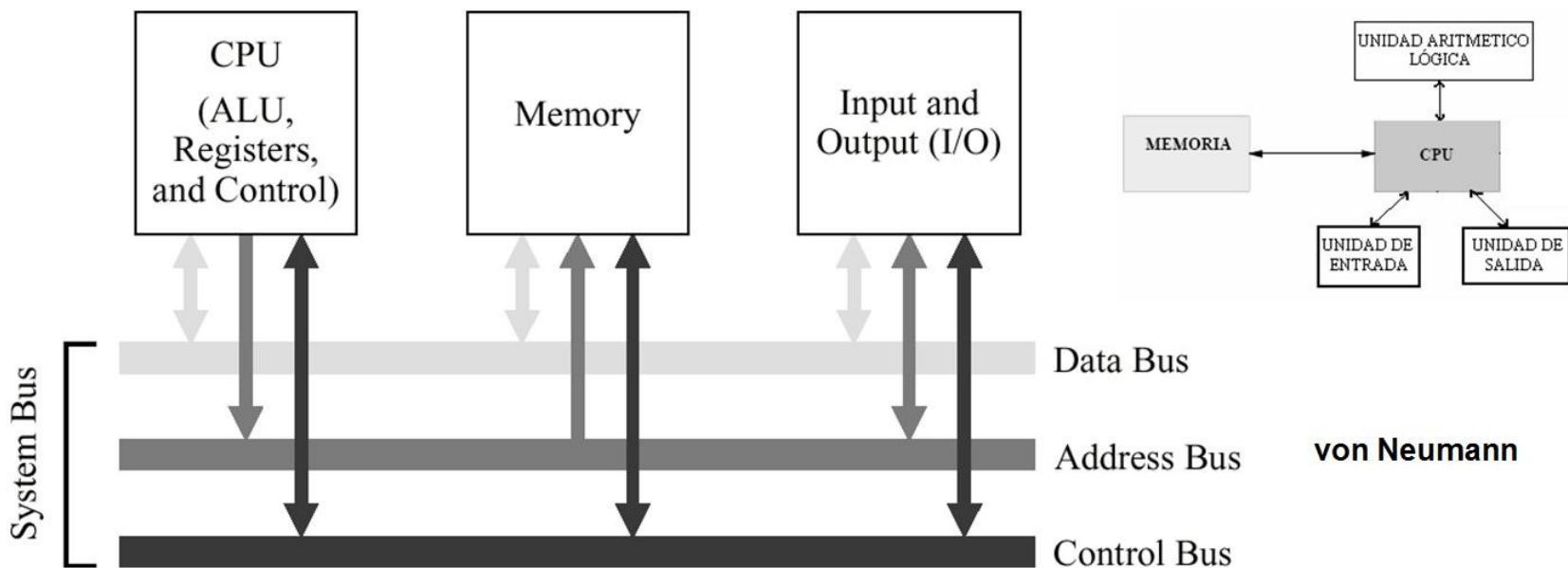
- hardware: cables, fibra óptica, circuitos integrados
- software: protocolos de comunicación

Básicamente está conformado por:

- Bus →
- Líneas de datos
  - Líneas de direcciones
  - Líneas control

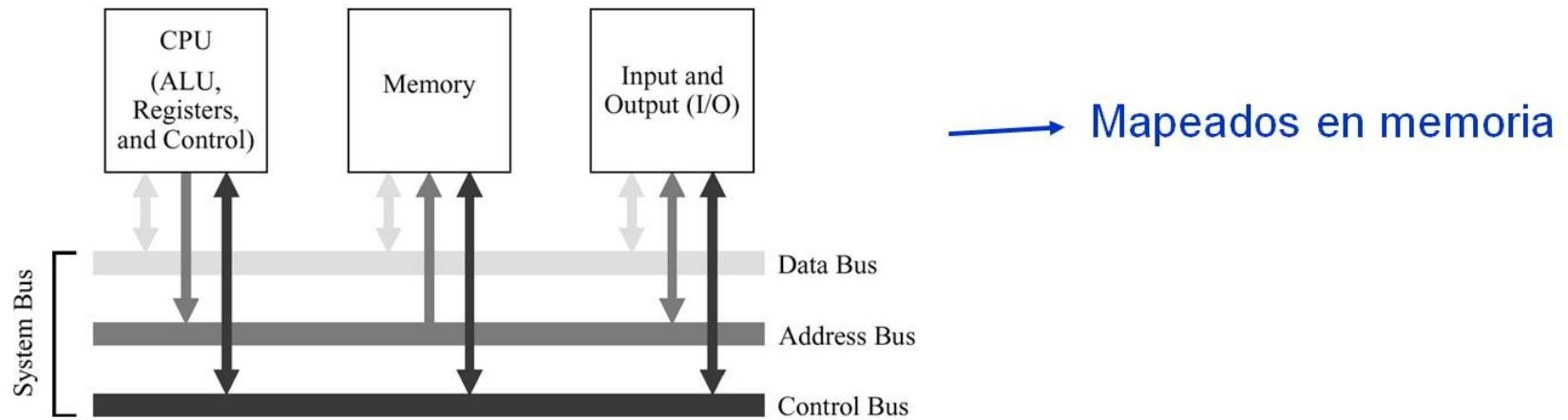
# Modelo bus de sistema

- ❖ La “arquitectura von Neumann” caracteriza a las computadoras actuales, pero se le ha incorporado un refinamiento: el **bus de sistema**
- ❖ Su propósito es reducir el número de interconexiones entre el CPU y sus subsistemas



\* Todos los dispositivos están cableados en paralelo al mismo bus ¿Cómo se comunican?

# Comunicación con los dispositivos de E/S



Cómo es el esquema de buses en un sistema NO-mapeado en memoria

## Breve historia de la arquitectura de computadoras

- Arquitectura Harvard
- Arquitectura Von Neumann
- Bus de sistema

### Microprocesadores:

- ¿Cuándo aparecieron los microprocesadores?
- ¿Dónde está el microprocesador de las anteriores arquitecturas?

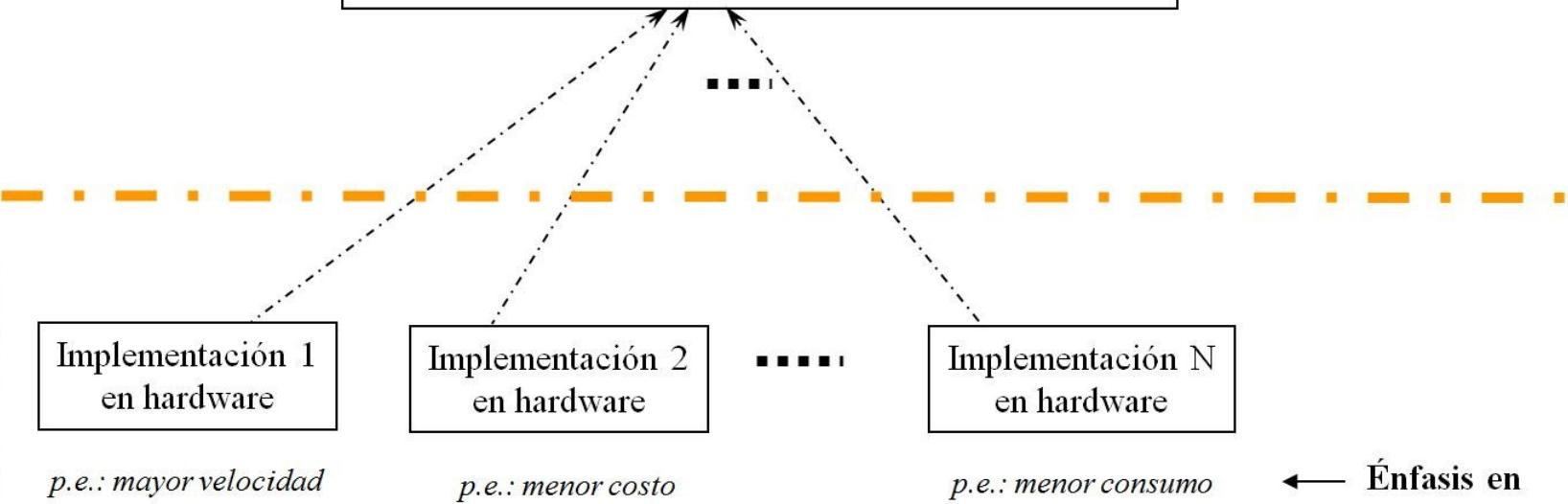
**“Arquitectura del CPU”**

- o “Arquitectura del microprocesador”**
- o “Microarquitectura”**

# MICROARQUITECTURA

## ISA

- Set de instrucciones
- Hardware visible al programador



*¿Cuál son las funciones específicas del CPU?*

*Ejecutar un programa significa:*

- 1) Buscar en memoria la próxima instrucción a ser ejecutada
- 2) Decodificar el código de operación de esa instrucción
- 3) Ejecutar la instrucción
- 4) Volver a (1)



**Ciclo de búsqueda-ejecución**

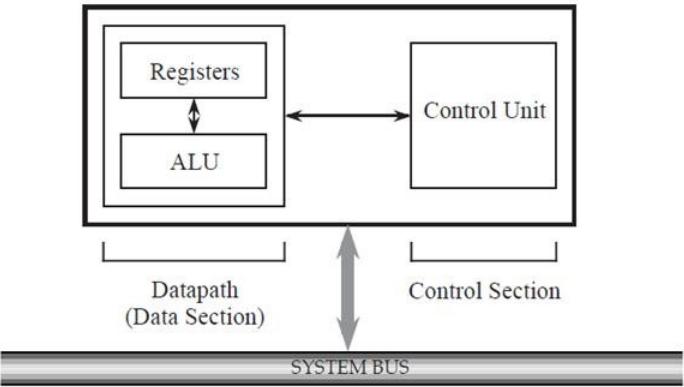
DEFINICIÓN ALTERNATIVA (libro de Murdocca-Heuring)

1. Buscar en memoria la siguiente instrucción a ejecutar.
2. Decodificar el código de operación.
3. Si los hubiera, leer los operandos desde memoria o desde registros.
4. Ejecutar la instrucción y almacenar los resultados.
5. Volver al paso 1

← ¿Se aplica a ARC?

# Microarquitectura =

- ✓ Unidad de control
- ✓ Unidad aritmético lógica
- ✓ Registros visibles al programador
- ✓ Cualquier otro registro adicional necesario para el funcionamiento de la unidad de control



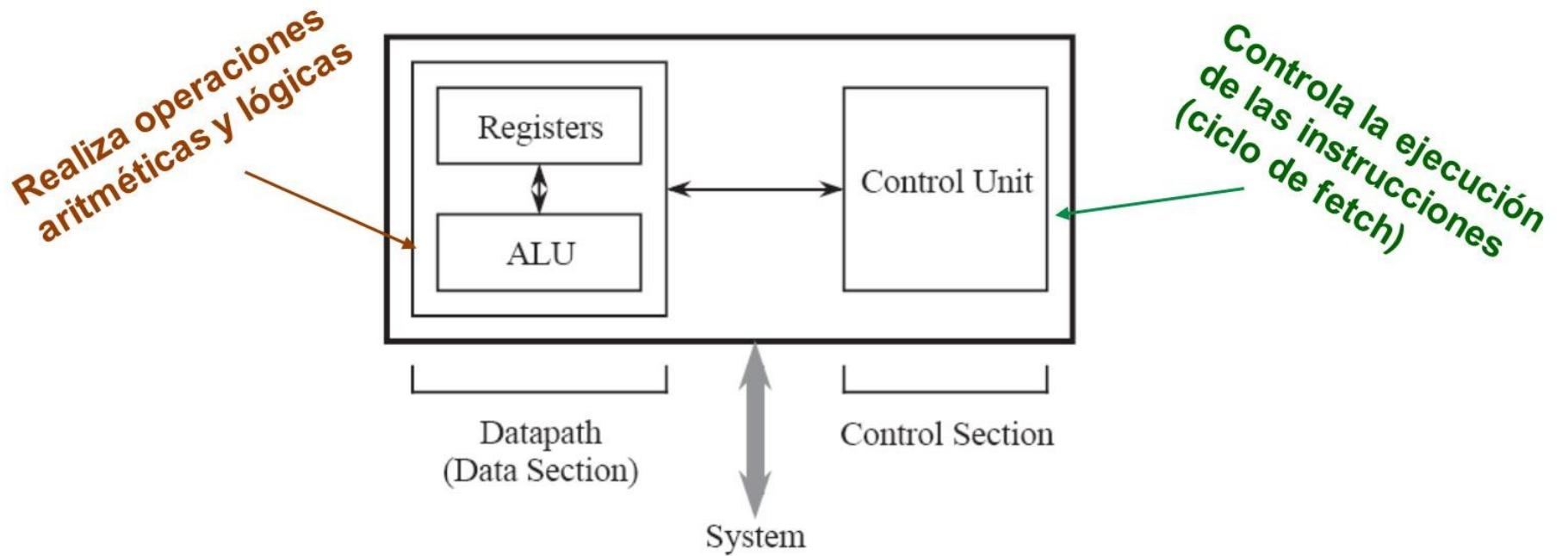
Objetivo: *implementar el ciclo de fetch*

# Microarquitectura =

Está conformada por:

❖ **Sección de control**

❖ **Sección de datos o “camino de datos” o “datapath”**



# Organización del Trayecto de Datos

- ¿Cuáles son los elementos que conforman el T. de D.?
  
- ¿Qué objetivo cumple la organización de estos elementos?

## Conexionado interno

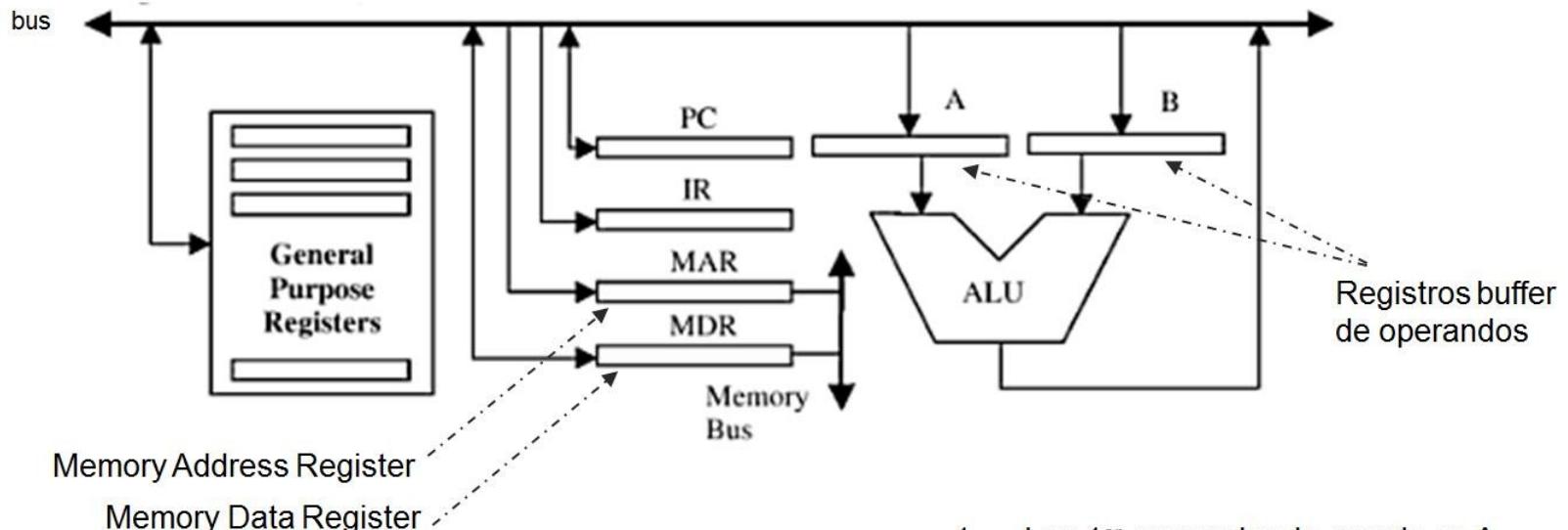
### Estructuras alternativas

- Organizada en 1 bus
- Organizada en 2 buses
- Organizada en 3 buses

## Conexionado externo

-- *bus de sistema*

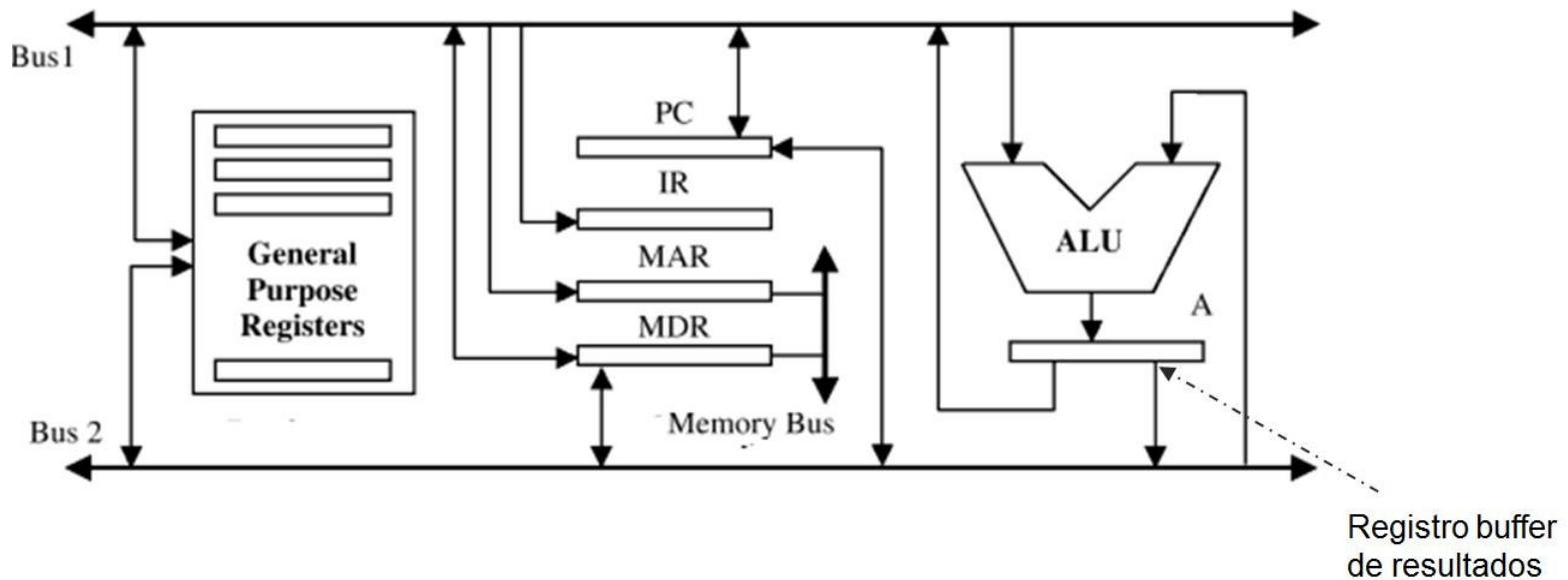
# Trayecto de Datos Organizado en 1 bus



1. Lee 1<sup>er</sup> operando y lo guarda en A
2. Lee 2<sup>do</sup> operando y lo guarda en B
3. Hace la operación (ALU)
4. Lee resultado y lo guarda en un registro

*¿Cuál es la secuencia para acceder a un dato en memoria?*

# Trayecto de Datos Organizado en 2 buses

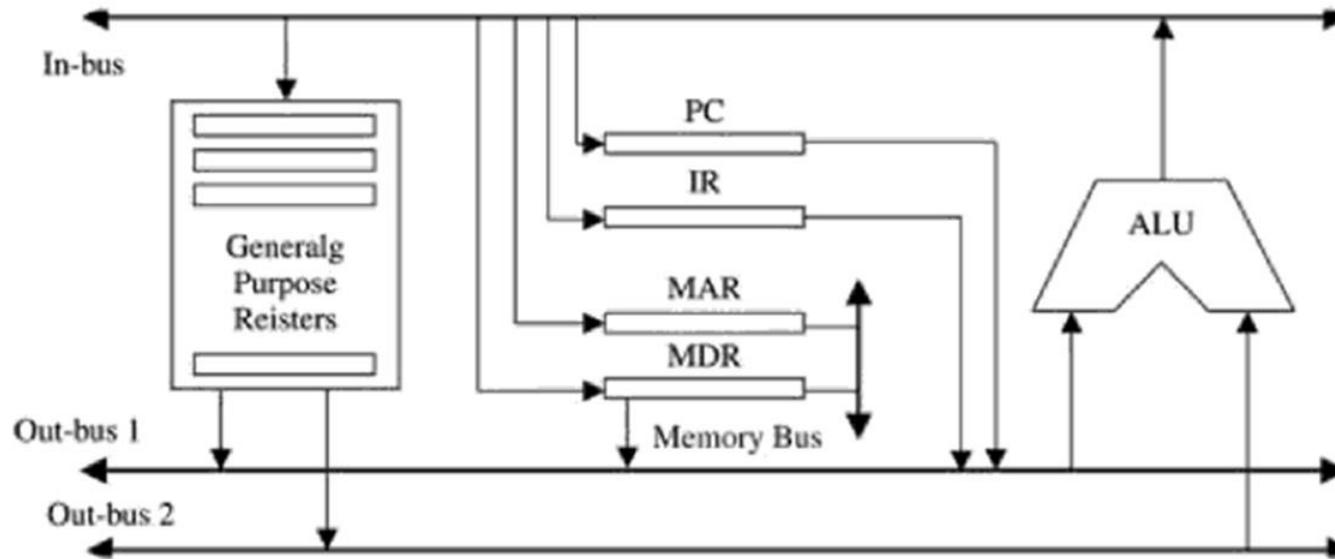


– Comparar con 1 bus la eficiencia para transferir datos por cada ciclo de reloj

# Trayecto de Datos Organizado en 3 buses

*Out-bus:* Dato sale de registro

*In-bus:* Dato entra a registro



- Comparar con 1 y 2 buses la eficiencia para transferir datos por cada ciclo de reloj

Esta implementación

Además de su organización en 3 buses  
Incluye la idea del bus unidireccional

# Unidad Aritmético-Lógica (ALU)

## Función

- Mover datos y hacer operaciones aritméticas/lógicas

Cuales? :

- Operaciones básicas de ALU para obtener un set de instrucciones dado  
*(no necesariamente coinciden con las del ISA)*

## Implementación

- Con 32 módulos de 1 bit puede obtenerse una ALU de 32 bits

## Performance

- Velocidad
- Área ocupada en el integrado
- Disipación de potencia

Ej.: Carry-look-ahead:  
- Alta velocidad  
- Ocupa mucho espacio

# Implementación de una ALU

## Funciones que debe realizar la ALU:

- Operaciones **aritméticas y lógicas**
- Desplazamientos a derecha e izquierda

## Soluciones alternativas:

- Para las operaciones aritmético-lógicas
  - Sol. 1: Circuitos sumadores, restadores, etc. que vimos anteriormente
  - Sol. 2: Usar una “tabla de consulta”
- Para los desplazamientos
  - Sol. 1: Registros de desplazamiento
  - Sol. 2: Desplazador rápido o “Barrel-shifter”

Como sería? (4bits)

¿Cuál es el problema de hacer un desplazamiento de n bits usando registros de desplazamiento?

# Tabla de consulta o “look-up table”

## Realiza :

- operaciones aritméticas
- operaciones lógicas

## Entradas:

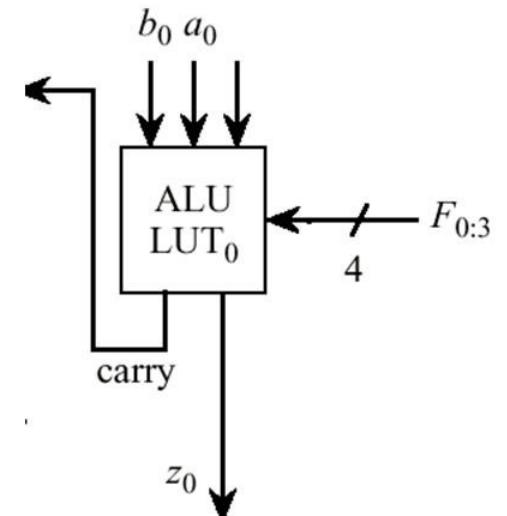
- 2 operandos 32 bits
- 4 bits para seleccionar la operación

## Salidas:

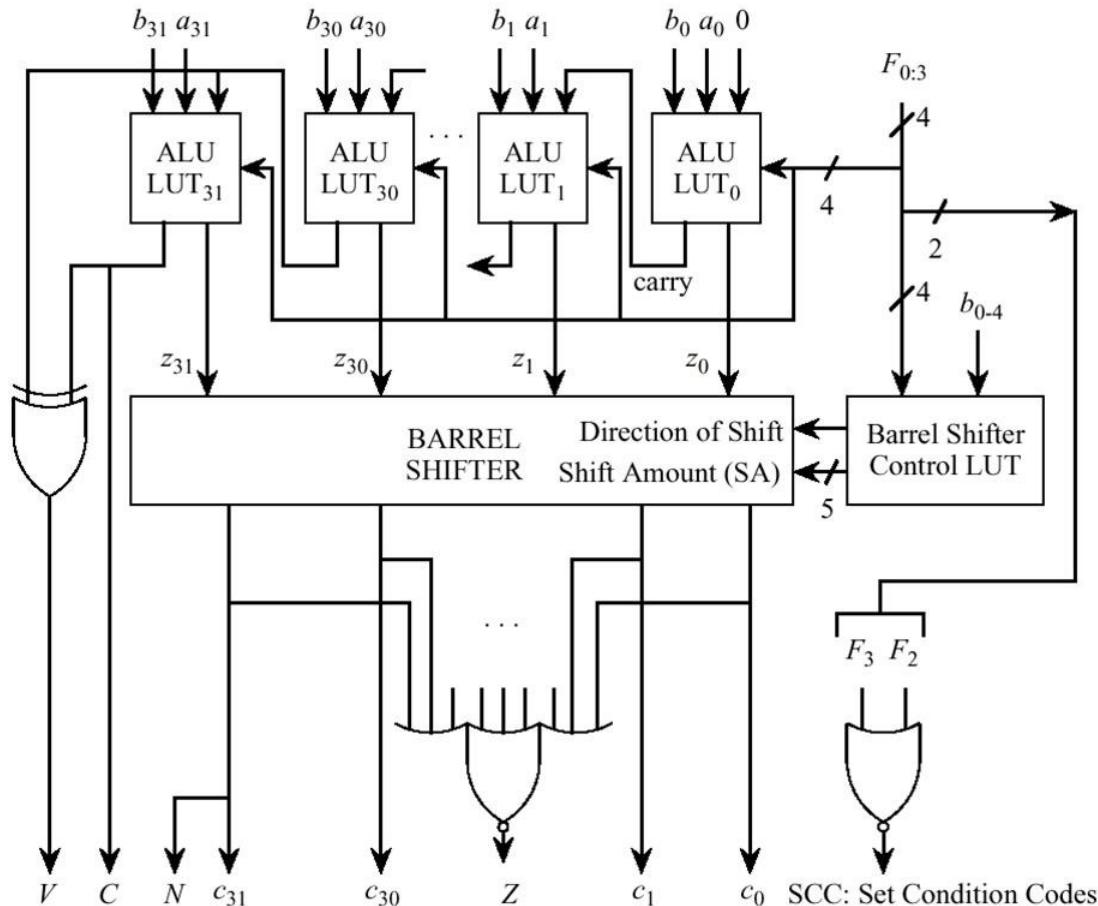
- Resultado de 32 bits
- Códigos de condición: n, z, v, c
- Si/No Códigos de condición: 1 bit

✓ Con 32 LUT de 1 bit pueden implementarse operaciones en 32 bits

# Tabla de verdad (parcial) de una LUT de 1 bit



# ALU implementada con “look-up table” y “barrel-shifter”



$F_3 \ F_2 \ F_1 \ F_0$	Operation
0 0 0 0	ANDCC (A, B)
0 0 0 1	ORCC (A, B)
0 0 1 0	NORCC (A, B)
0 0 1 1	ADDCC (A, B)
0 1 0 0	SRL (A, B)
0 1 0 1	AND (A, B)
0 1 1 0	OR (A, B)
0 1 1 1	NOR (A, B)
1 0 0 0	ADD (A, B)
1 0 0 1	LSHIFT2 (A)
1 0 1 0	LSHIFT10 (A)
1 0 1 1	SIMM13 (A)
1 1 0 0	SEXT13 (A)
1 1 0 1	INC (A)
1 1 1 0	INCPC (A)
1 1 1 1	RSHIFT5 (A)

# *Desplazador rápido o “barrel-shifter”*

## **Realiza desplazamientos:**

- a derecha o a izquierda
- de 0 hasta 31 bits en un solo ciclo de reloj

## **Entradas:**

- operando: 32 bits
- cantidad de bits a desplazar: 5 bits
- dirección: 1 bit

## **Salida:**

- Resultado: 32 bits

# Desplazador rápido (Barrel Shifter)

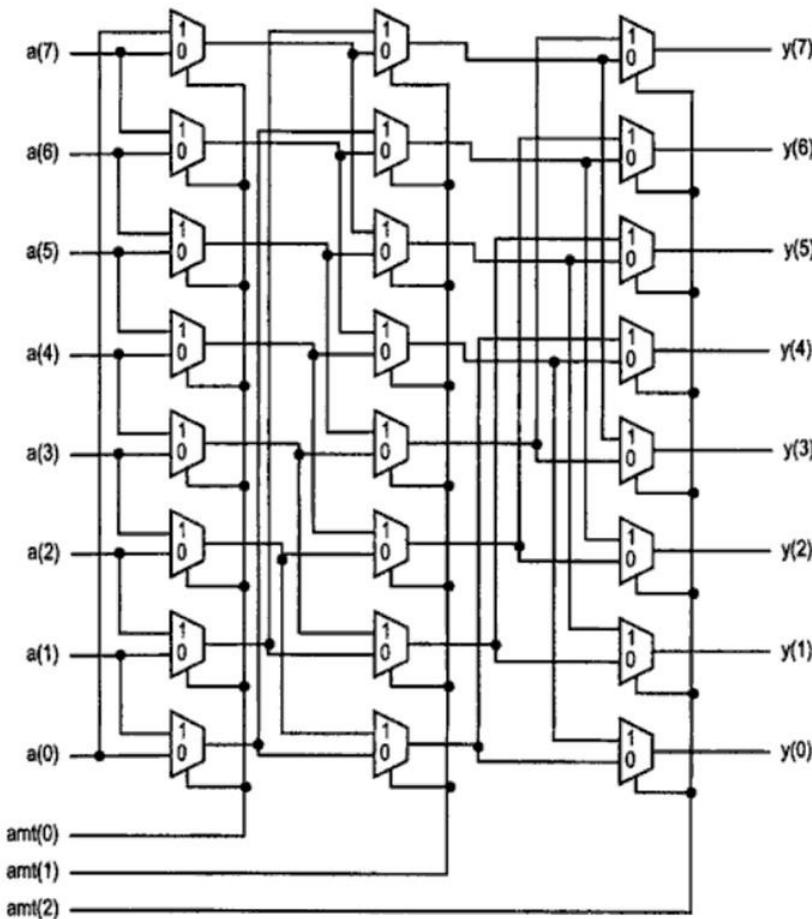
## Funciones

- Shift lógico de n bits der/izq
- Shift aritmético de n bits der/izq
- Shift circular (rotación) der/izq

## Principio de funcionamiento

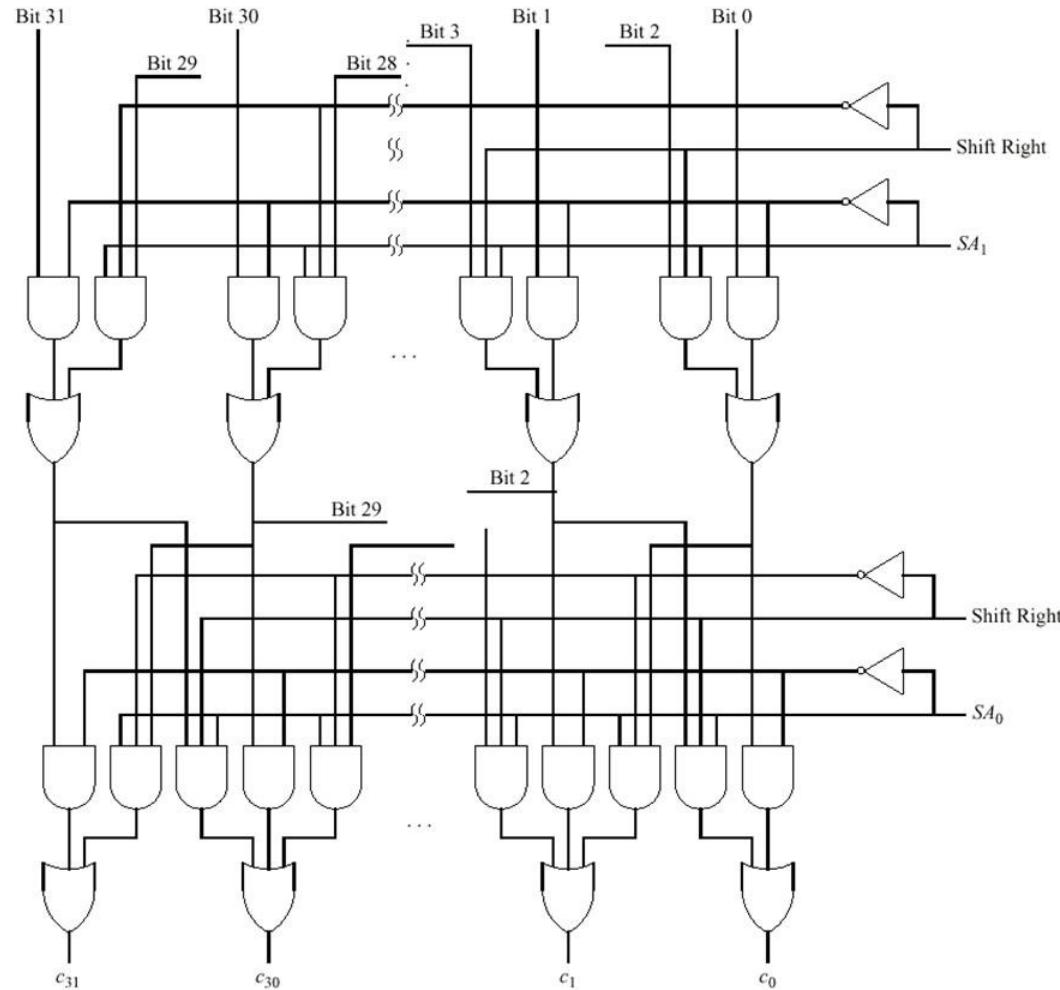
- Organización por niveles
- Cada nivel desplaza  $2^n$  al anterior
- Bits de control definen rango del shift

*Una implementación posible*



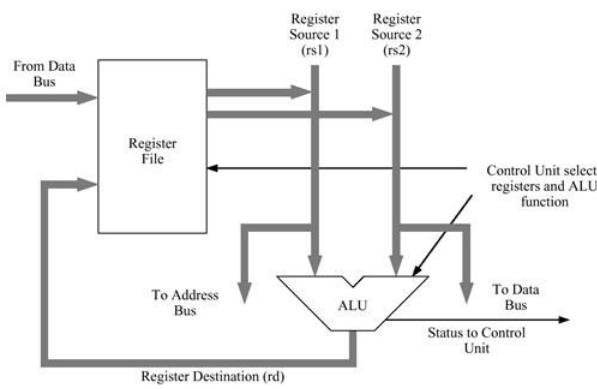
*Shift circular a derecha*

# Desplazador rápido (Barrel Shifter)

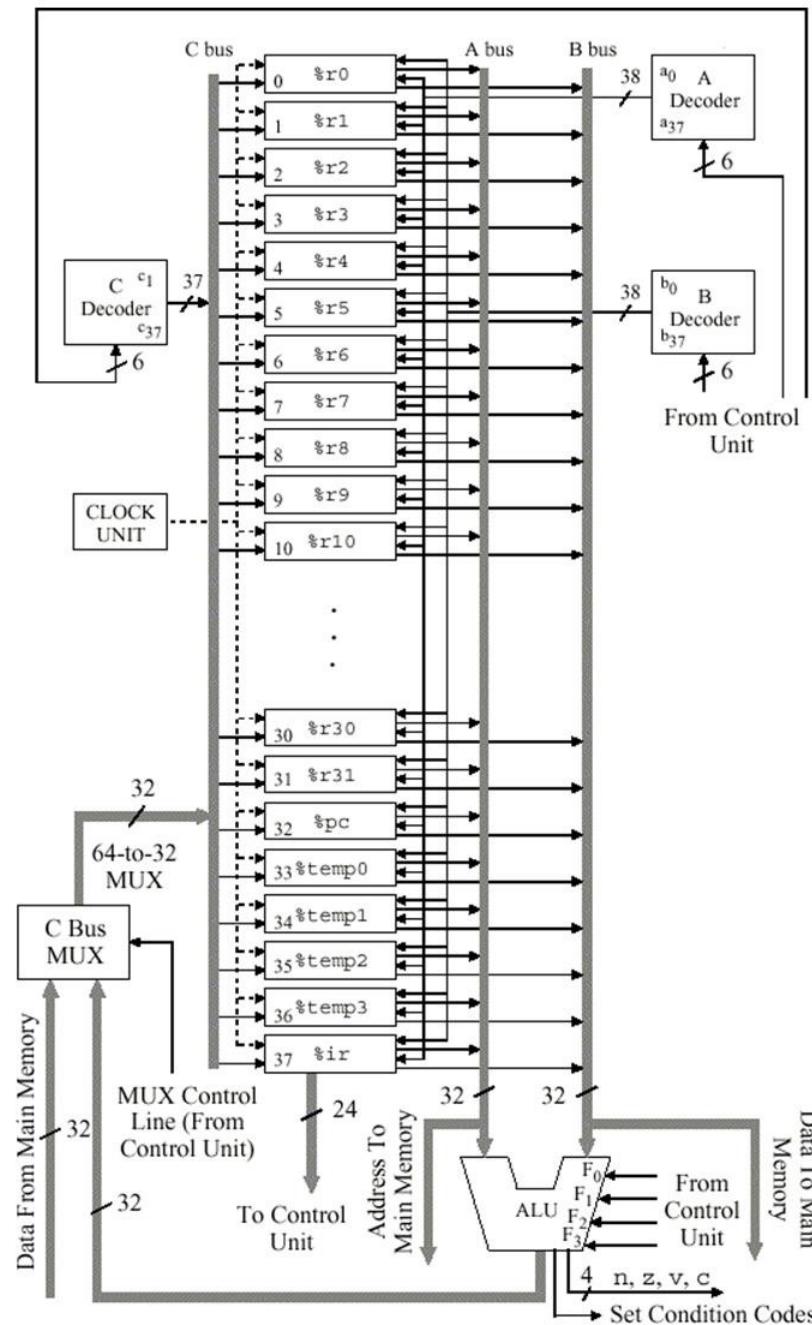


*Implementación presentada en el libro de Murdocca-Heuring*

# Un trayecto de datos para la microarquitectura ARC

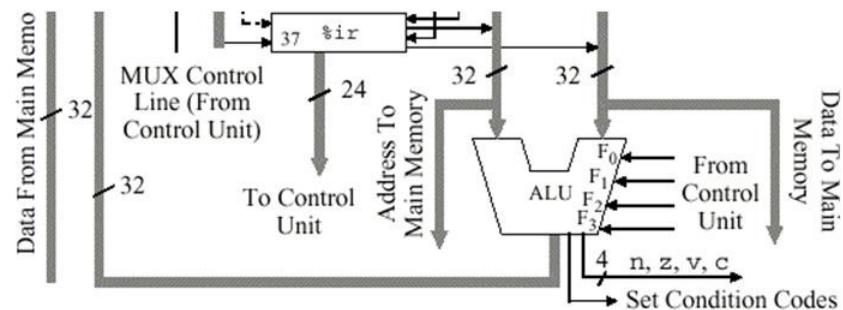
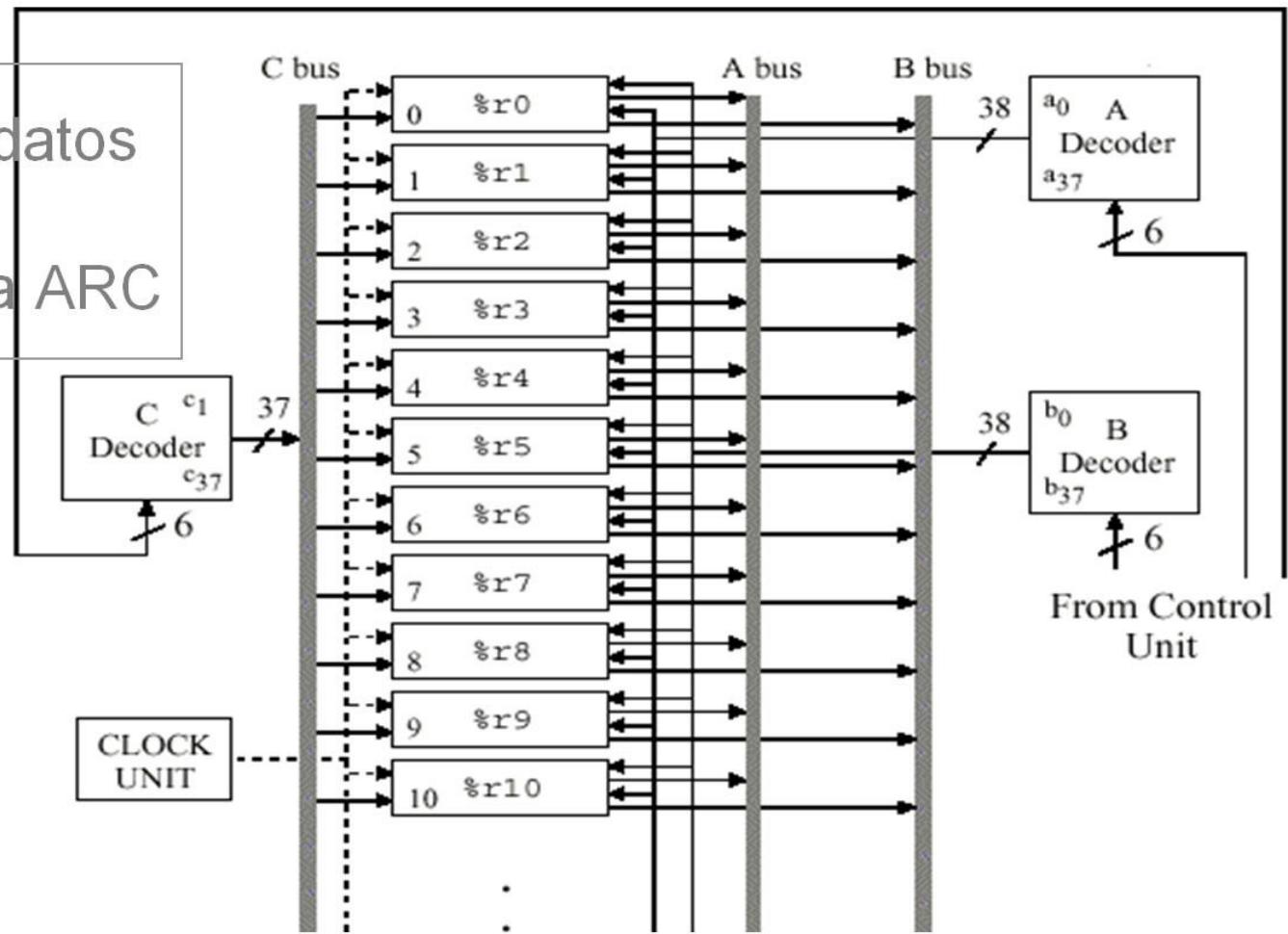


- ✓ Buses
- ✓ ALU
- ✓ Registros
- ✓ Decodificadores
- ✓ Multiplexores



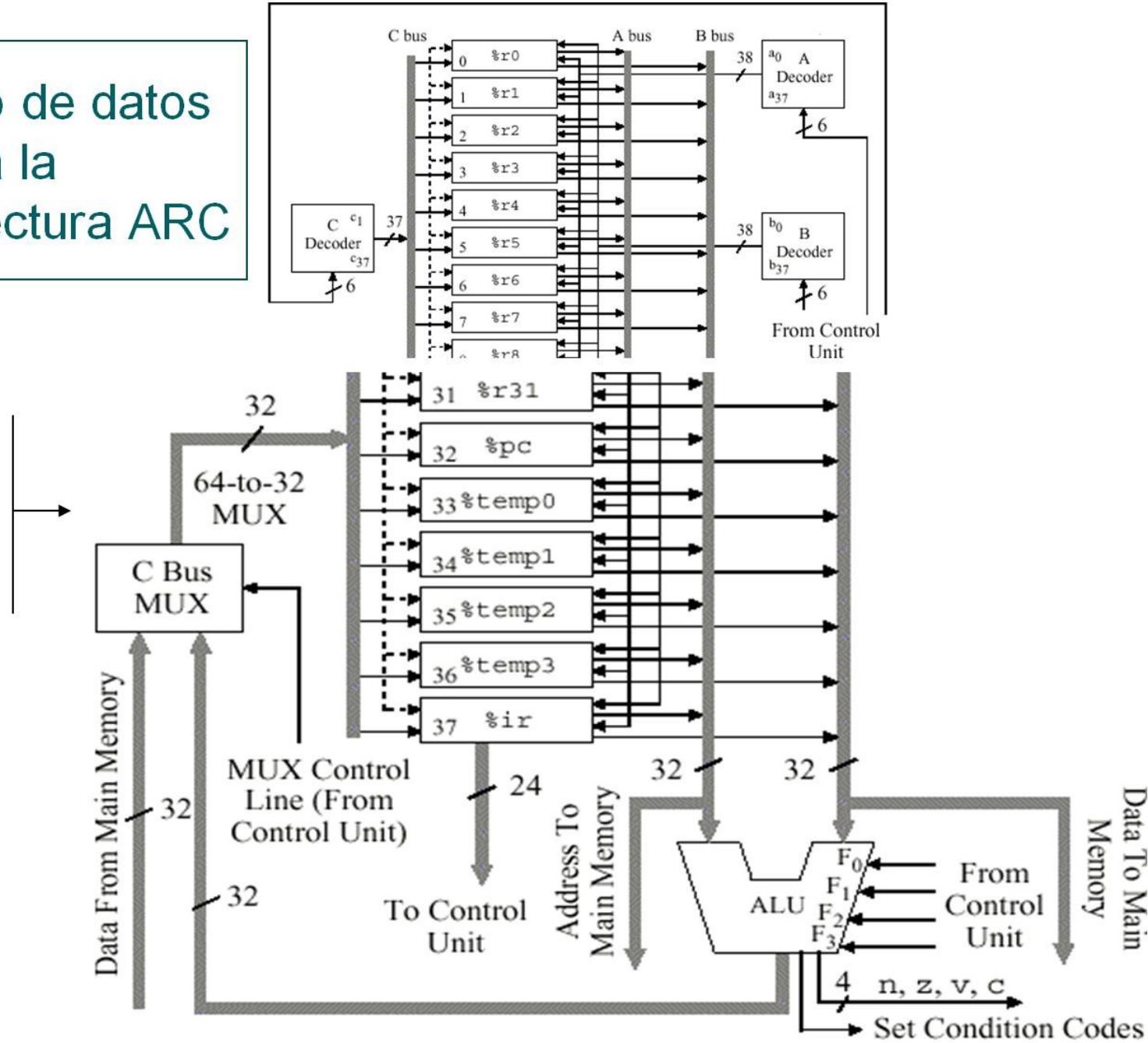
# Un trayecto de datos para la microarquitectura ARC

- ✓ Buses
- ✓ ALU
- ✓ Registros
- ✓ Decodificadores
- ✓ Multiplexores



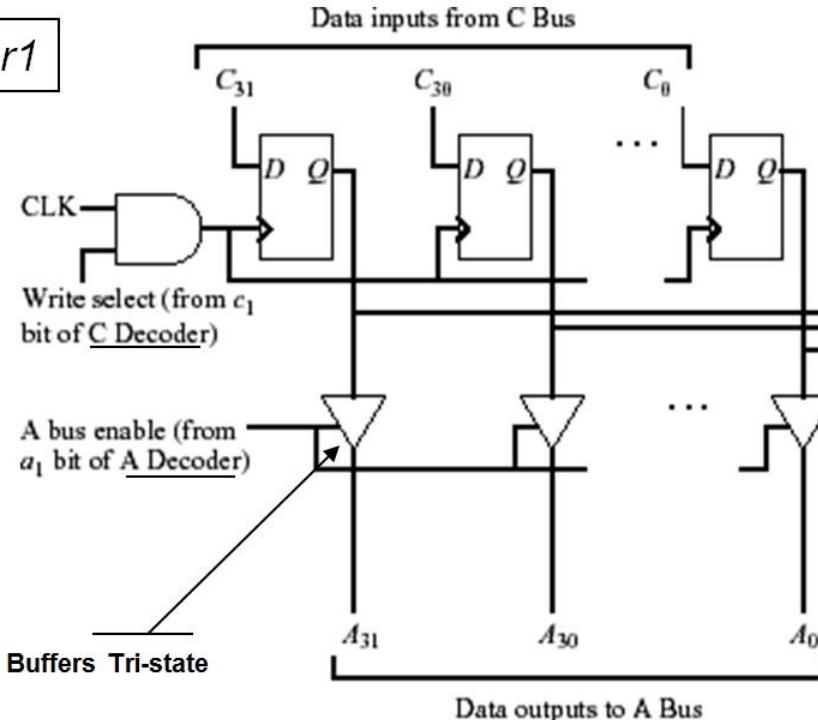
# Un trayecto de datos para la microarquitectura ARC

- ✓ Buses
- ✓ ALU
- ✓ Registros
- ✓ Decodificadores
- ✓ Multiplexores

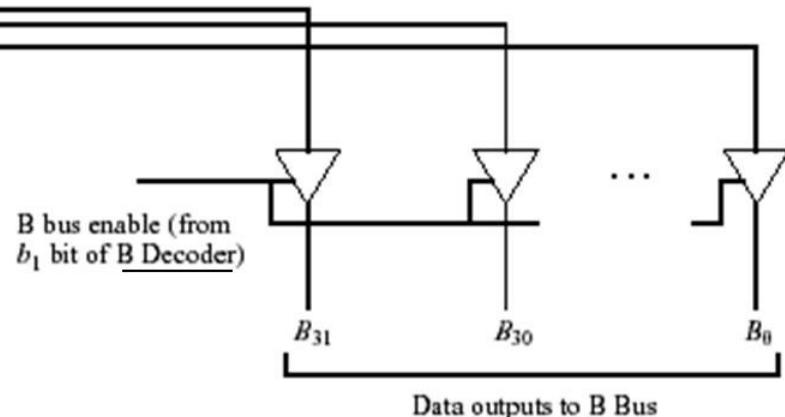


# Estructura interna de los registros

$\%r1$



← **Estructura general**  
 (excepciones en transparencia siguiente)



- Los Flip-Flops son tipo de D por flanco descendente

¿Porqué digo que es  $\%r1$ ?

- ¿Cómo se activa la lectura?
- ¿Cómo se activa la escritura?

# Registros que **no** responden a la estructura general

## ✓ Registro **%r0**

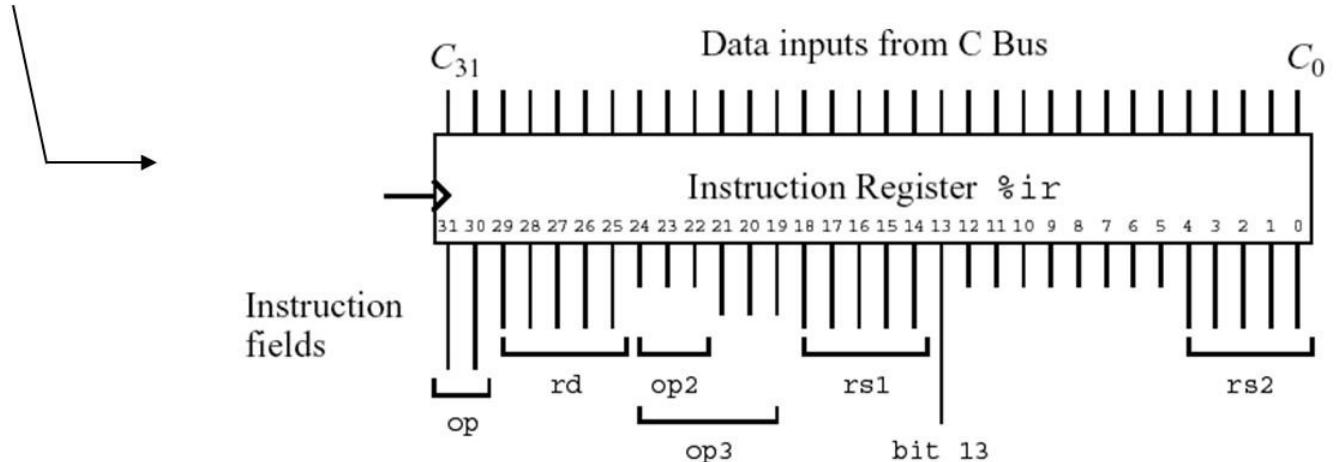
- No necesita flip-flops
- No tiene entrada de datos desde bus C
- No tiene entrada desde decodificador del bus C

## ✓ Contador de programa **PC**

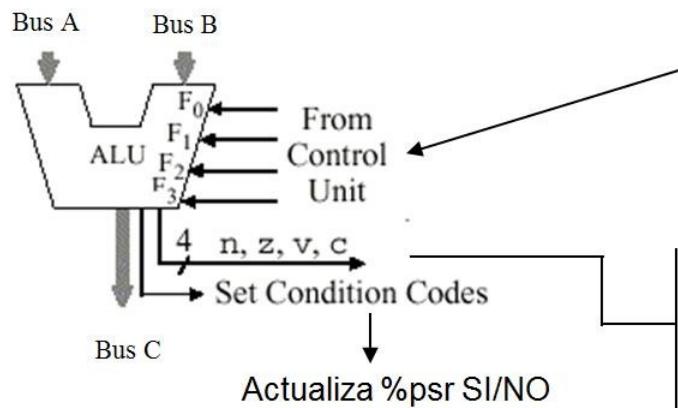
- Sólo almacena números múltiplos de 4  
=> los 2 LSB cableados a cero

## ✓ Registro de instrucciones **IR**

- Tiene salidas específicas por campos del código de máquina



# La Unidad Aritmético-Lógica

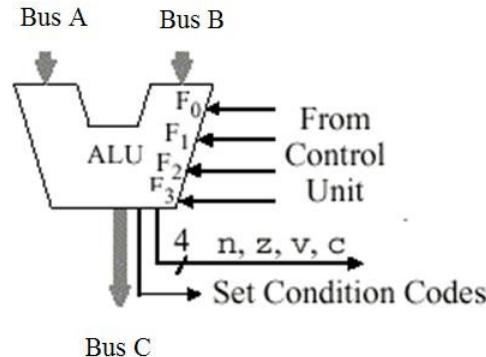


Operaciones implementadas por la ALU

$F_3$	$F_2$	$F_1$	$F_0$	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	ADDCC (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

- SRL: Shift right B pos.(0-31) el valor en A
- LSHIFT2: (Bus A) shift left 2 pos.
- LSHIFT10: (Bus A) shift left 10 pos.
- SIMM13: LSB(13,Bus A) MSB = 0's
- SEXT13: Simm13 en Bus A con Ext. Signo
- INC: (BusA) + 1
- INCPC: (BusA) + 4
- RSHIFT5: Shift right 5 pos. y ext. signo

# Desde instrucciones básicas de ALU al set de instrucciones del procesador



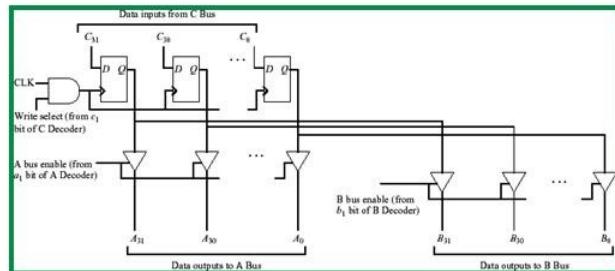
$F_3 \ F_2 \ F_1 \ F_0$	Operation	Changes Condition Codes
0 0 0 0	ANDCC (A, B)	yes
0 0 0 1	ORCC (A, B)	yes
0 0 1 0	NORCC (A, B)	yes
0 0 1 1	ADDCC (A, B)	yes
0 1 0 0	SRL (A, B)	no
0 1 0 1	AND (A, B)	no
0 1 1 0	OR (A, B)	no
0 1 1 1	NOR (A, B)	no
1 0 0 0	ADD (A, B)	no
1 0 0 1	LSHIFT2 (A)	no
1 0 1 0	LSHIFT10 (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

(lenguaje ilustrativo)

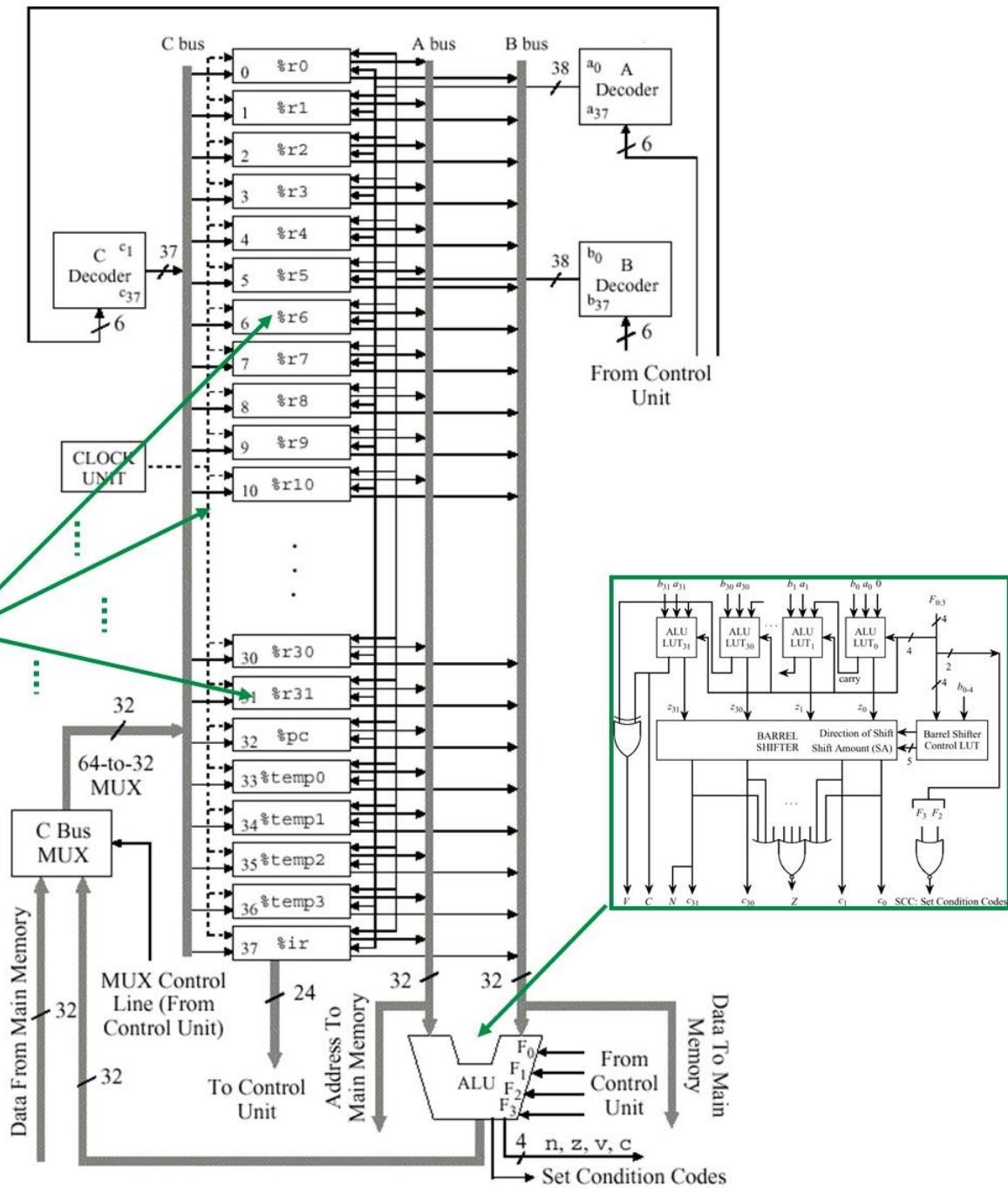
Complemento a 2	NOR (%r2, %r2), %r1 INC %r1
Resta	NOR (%r2, %r2), %temp1 INC %temp1 ADDCC (%rtemp1, %r1), %r1
Shift Left 1 bit	ADD (%r2, %r2), %r2

# Una microarquitectura ARC

## ***Trayecto de Datos***



- ✓ Buses
  - ✓ ALU
  - ✓ Registros
  - ✓ Decodificadores
  - ✓ Multiplexores



# Diseño de una unidad de control

Podemos diseñar una unidad de control por:

→ *Lógica microprogramada*

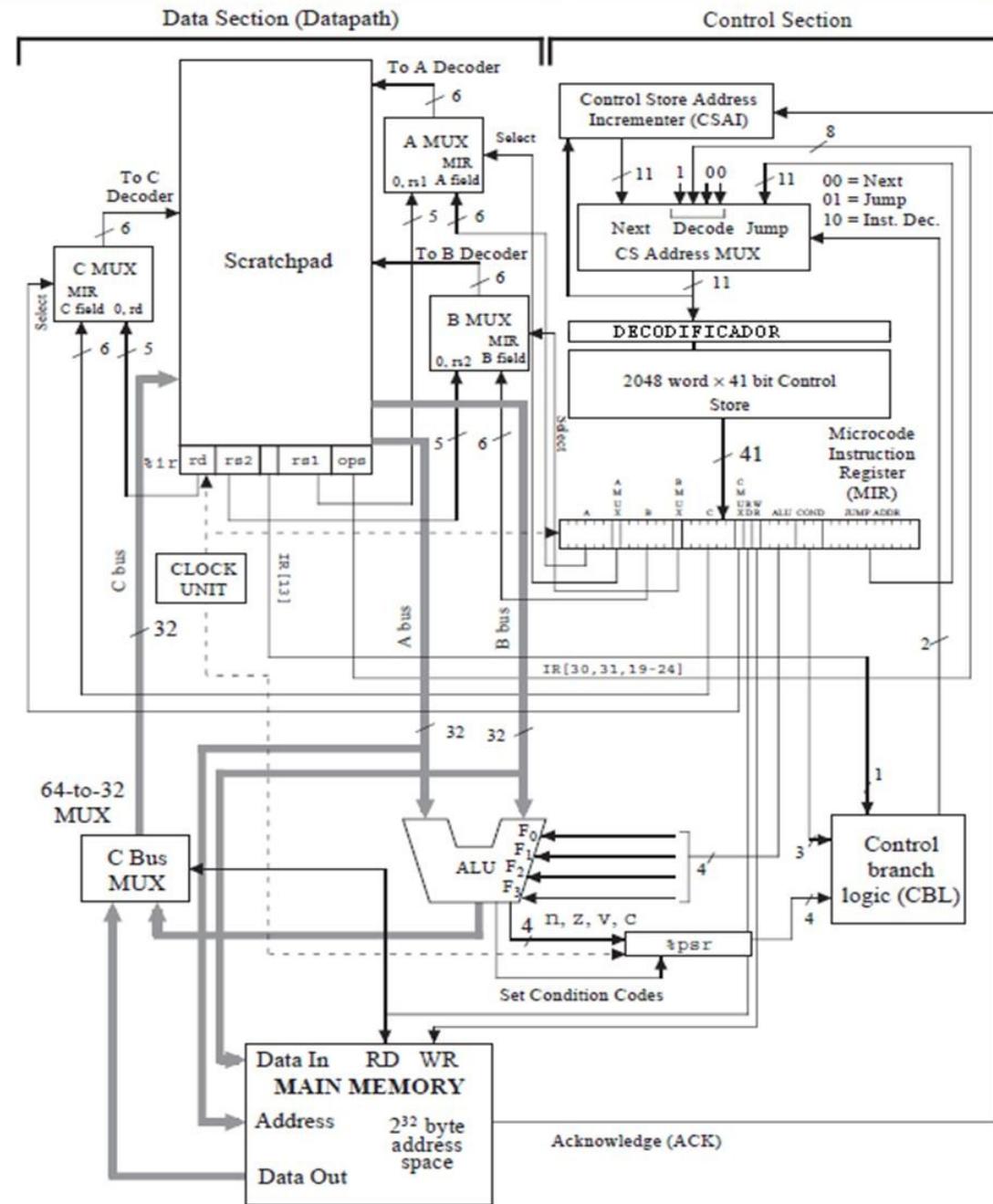
→ *Lógica cableada*

El trayecto de datos es esencialmente idéntico  
en ambos casos

# Microarquitectura

## Tr. de Datos + Un. de Control

Microprogramada



## Control store (ROM)

- $2^{11}$  microinstrucciones
- cada microinstrucción ocupa 41 bits

## MIR

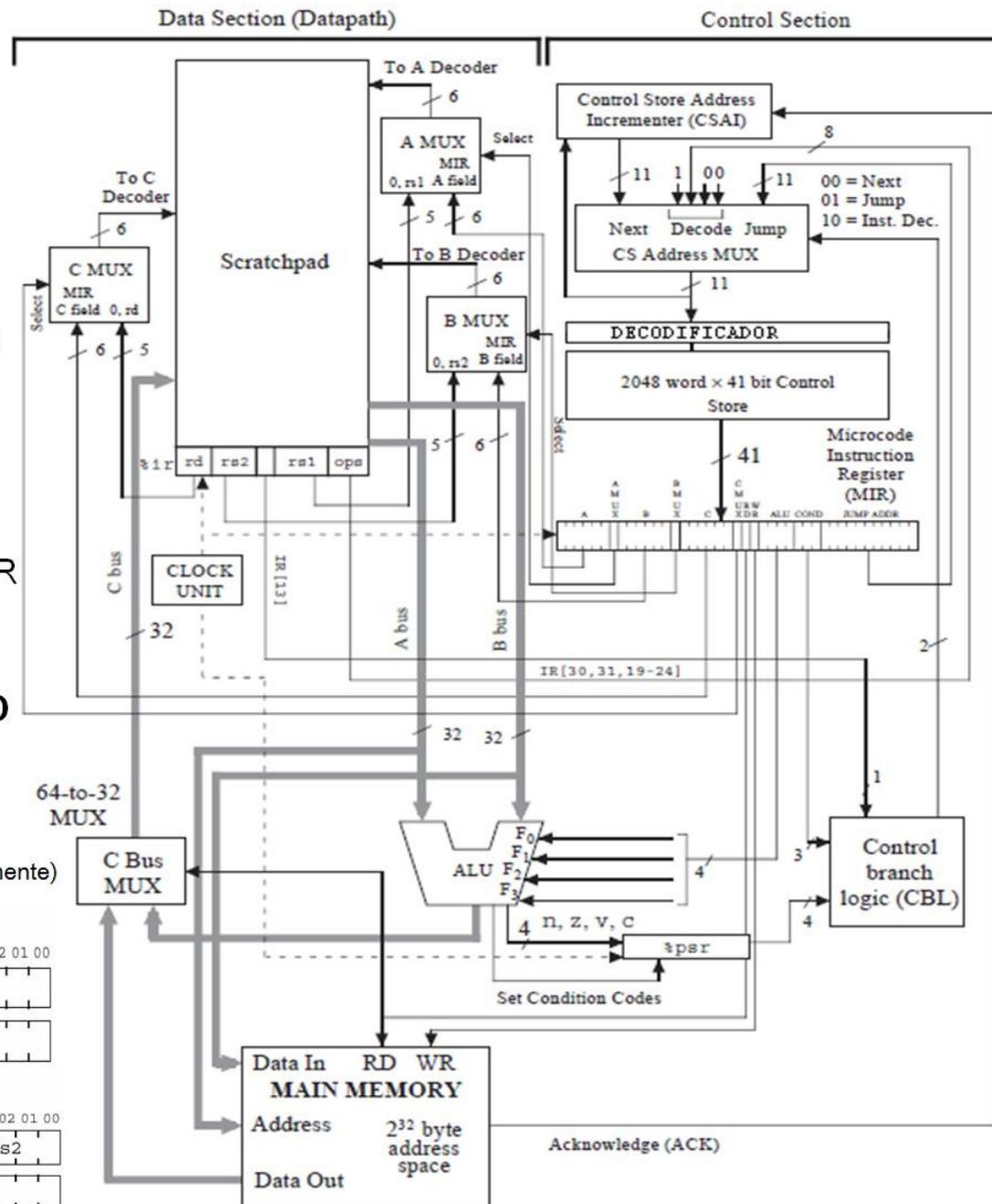
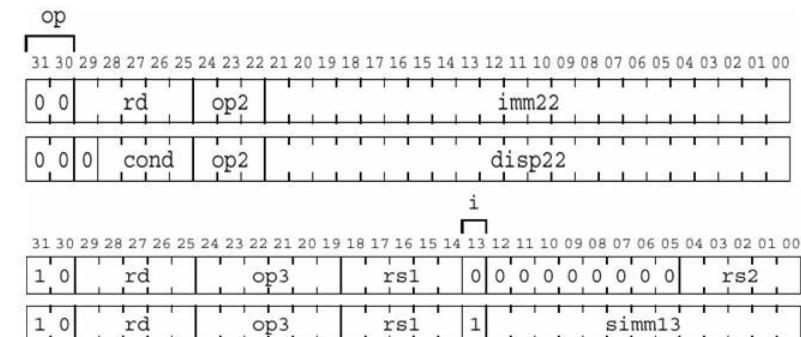
- Guarda microinstrucciones. así como el IR guarda instrucciones del set del procesador

## Ejecución del microcódigo

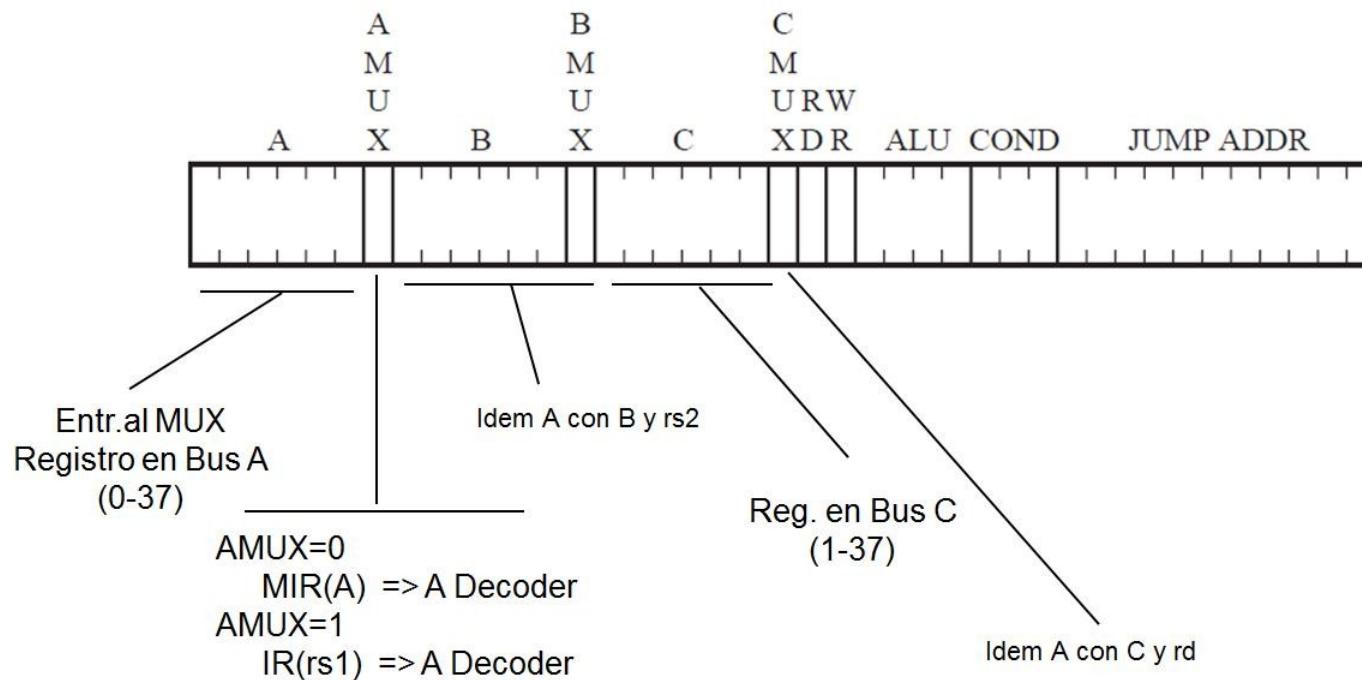
- CS-Addr-MUX apunta a próx. microinstr.
- Microinstrucción: se copia desde CS al MIR
- Control de flujo: CBL y CS-Addr-MUX

## IR con salidas por cada campo

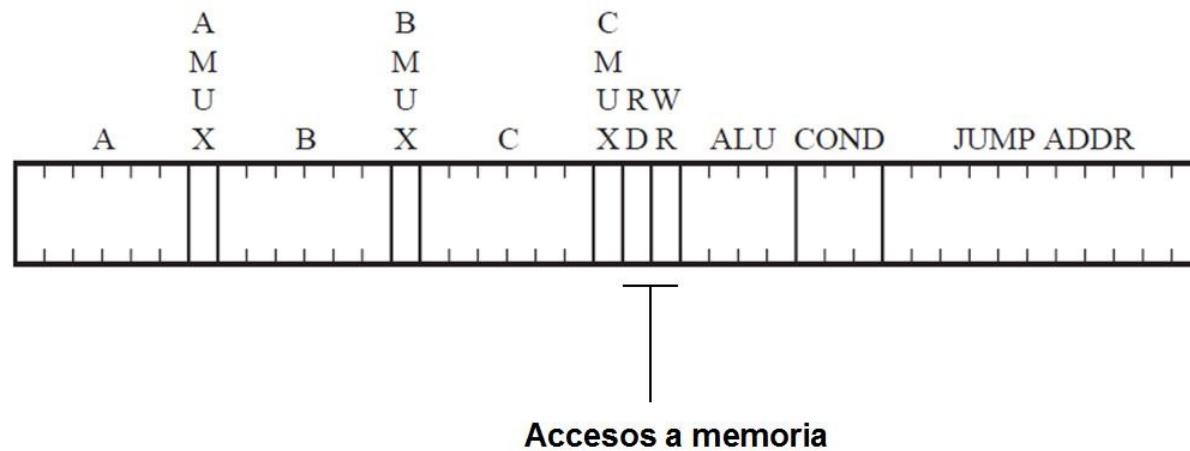
- Registros origen y destino: *rd*, *rs1*, *rs2*
- Si es direccionamiento inmediato: *Bit[13]*
- Códigos de operación: *op*, *op2* y *op3*  
(bits 31-30, 24-22, 24-19 respectivamente)



# Formato de las microinstrucciones



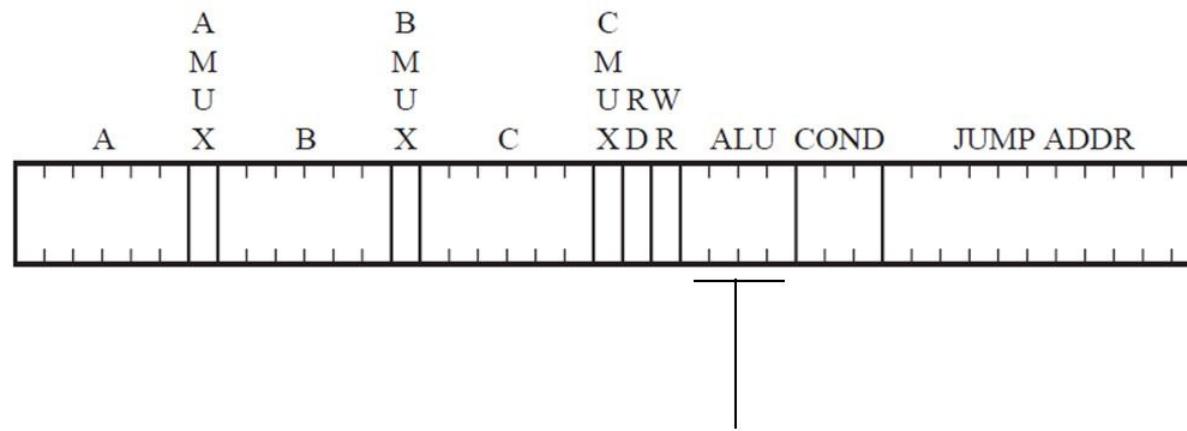
# Formato de las microinstrucciones



RD	WR	Acción
0	0	No accede a memoria
0	1	Escribe en memoria
1	0	Lee de memoria
1	1	-- (estado prohibido)

- Dirección de memoria en bus A
- Datos en bus B ó C

# Formato de las microinstrucciones



- En accesos a memoria la ALU no es necesaria
- **No** existe un código para “ALU apagada”

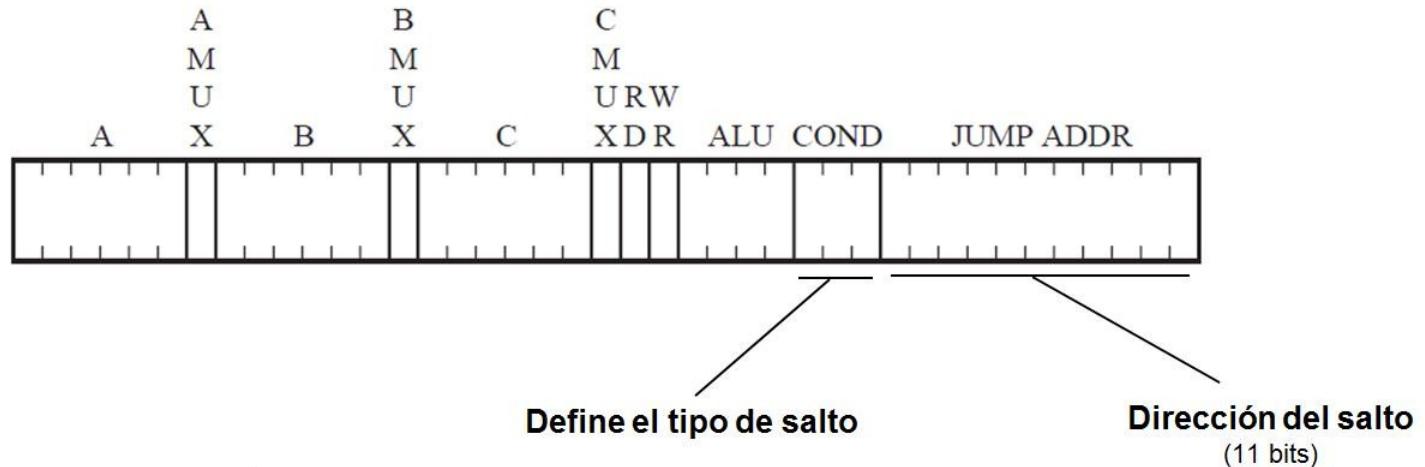
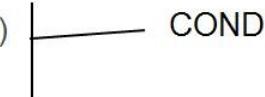


Usar cualquier operación  
que no altere códigos de condición

# Formato de las microinstrucciones

La Dirección del salto puede ser:

- Próxima microinstr en CS (el CSAI incrementa addr actual)
- Microinstr. apuntada por campo JUMP ADDR del MIR



**COND**

$C_2$	$C_1$	$C_0$	Operation
0	0	0	Use NEXT ADDR
0	0	1	Use JUMP ADDR if $n = 1$
0	1	0	Use JUMP ADDR if $z = 1$
0	1	1	Use JUMP ADDR if $v = 1$
1	0	0	Use JUMP ADDR if $c = 1$
1	0	1	Use JUMP ADDR if $IR[13] = 1$
1	1	0	Use JUMP ADDR
1	1	1	DECODE

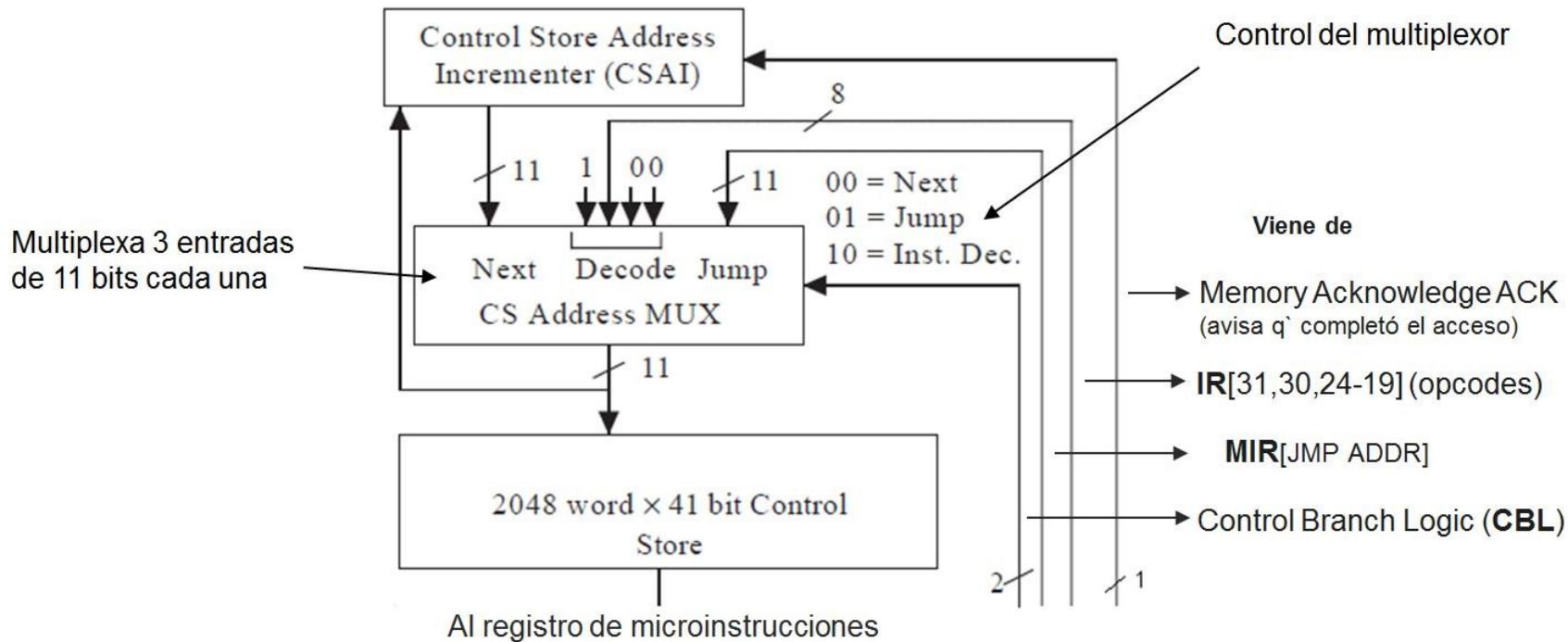
Avanza a la siguiente dirección del CS

Salta si el flag está en 1

Salta si el direccionamiento es "inmediato"

Salto incondicional

# Elegir la próxima microinstrucción a ejecutar



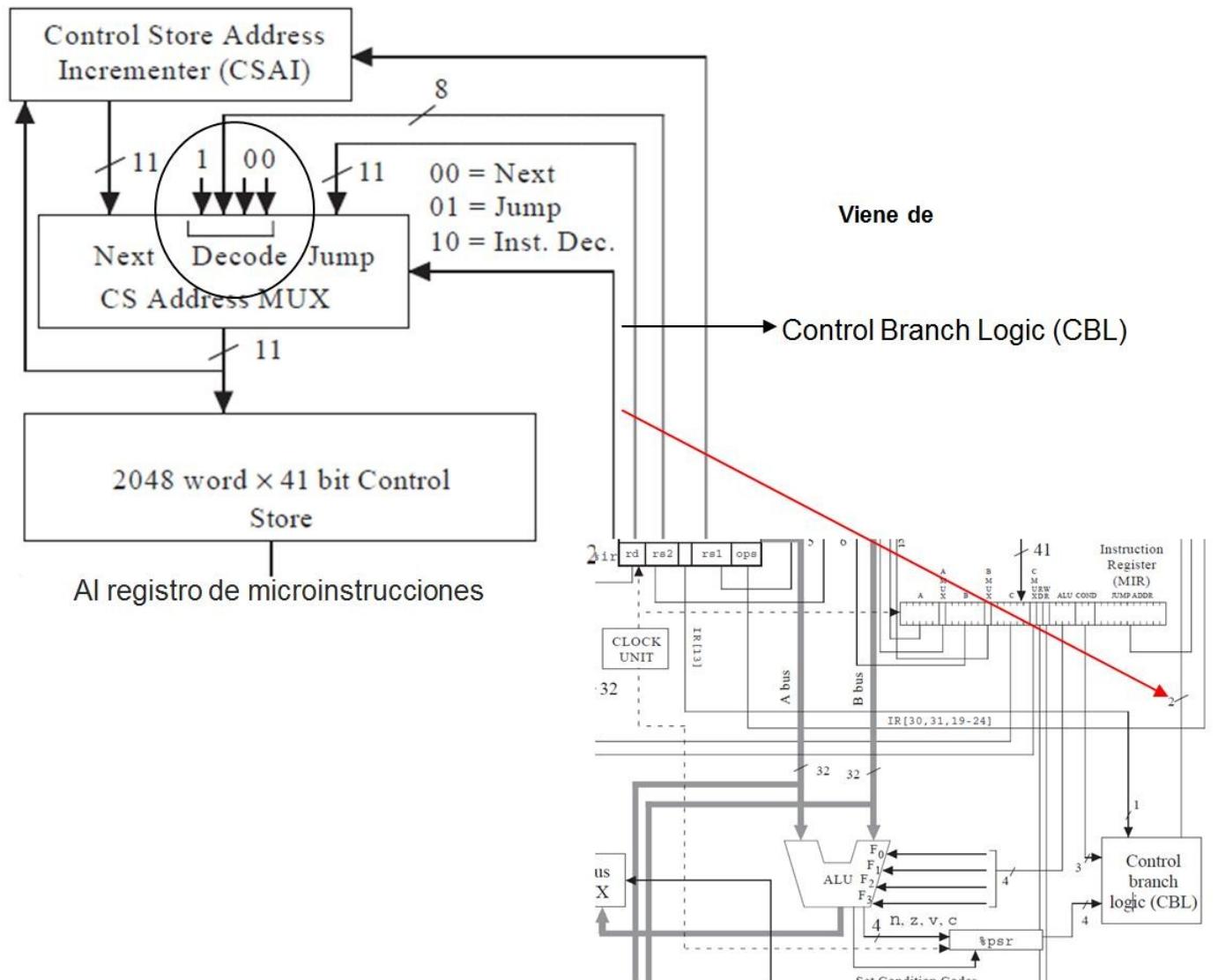
$C_2$	$C_1$	$C_0$	Operation
0	0	0	Use NEXT ADDR
0	0	1	Use JUMP ADDR if $n = 1$
0	1	0	Use JUMP ADDR if $z = 1$
0	1	1	Use JUMP ADDR if $v = 1$
1	0	0	Use JUMP ADDR if $c = 1$
1	0	1	Use JUMP ADDR if $IR[13] = 1$
1	1	0	Use JUMP ADDR
1	1	1	DECODE

CBL presenta 00 y CSaddrMUX toma dirección de CSAI q' suma 1 a la CS Address actual

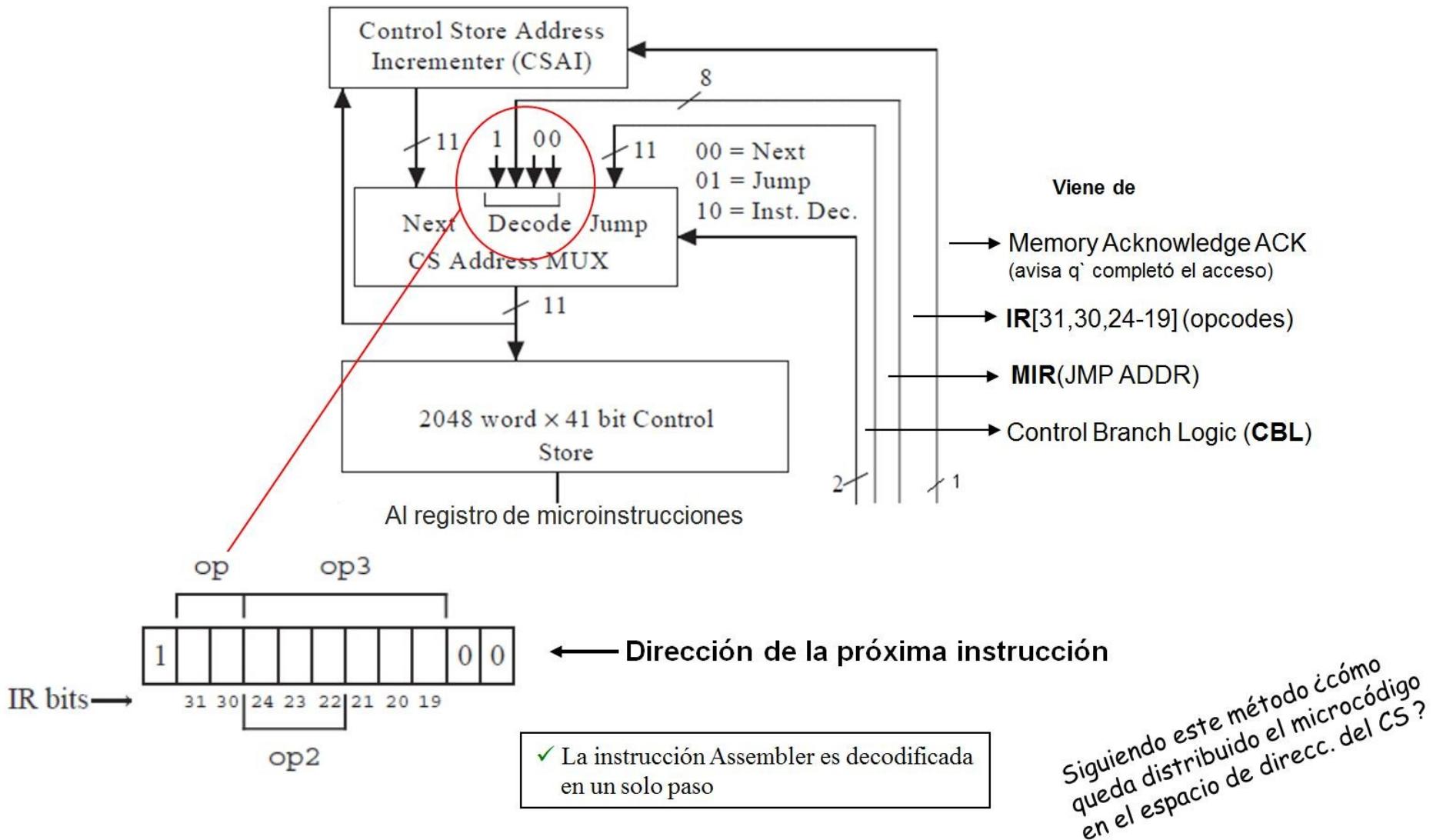
CBL presenta 01 y CSaddrMUX toma dirección de su entrada jump

⇒ "Decodifica" instrucción Assembler => la dirección del salto es la del inicio de la correspondiente rutina del microprograma

# Decodificar una instrucción de Assembly



# Decodificar una instrucción de Assembly



# Assembler vs. microprograma

- Programa en lenguaje de bajo nivel
  - Se implementa en Assembler del procesador (p.e., ARC)
  - Visible al programador
  - Implementa programas de propósito general
- Microprograma
  - Código que implementa cada instrucción del Assembler ARC
  - Invisible al programador
  - El microcódigo en binario está grabado en ROM (*firmware*)
  - Debe definirse un lenguaje ad-hoc
  - Un mismo set de instrucciones admite ser implementado con muchas versiones de firmware.

# Un microcódigo para la arquitectura ARC

## Sintaxis propuesta

- (*Dir en el Control Store*): (*sentencia*); / (*Comentario*)
- En cada posición del CS hay una o más instrucciones de microcódigo

En 1 ciclo de reloj se ejecutan todas las microinstrucciones de una posición del CS

---

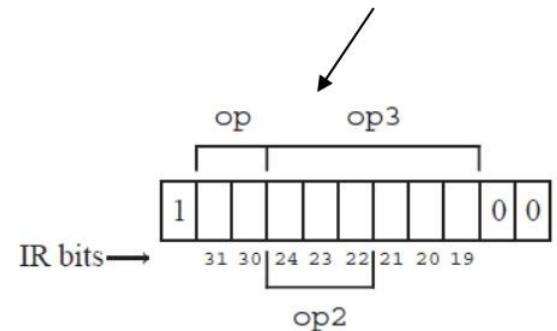
### / Decodificando el assembler ARC

0: R[ir] = AND(R[pc],R[pc]); READ;

/ Lee una instrucción desde memoria principal

1: DECODE;

/ Salto a microrutina que decodifica instr. ARC según *opcode*



# Microprograma en binario

(Se almacena en Control Store y en el MIR)

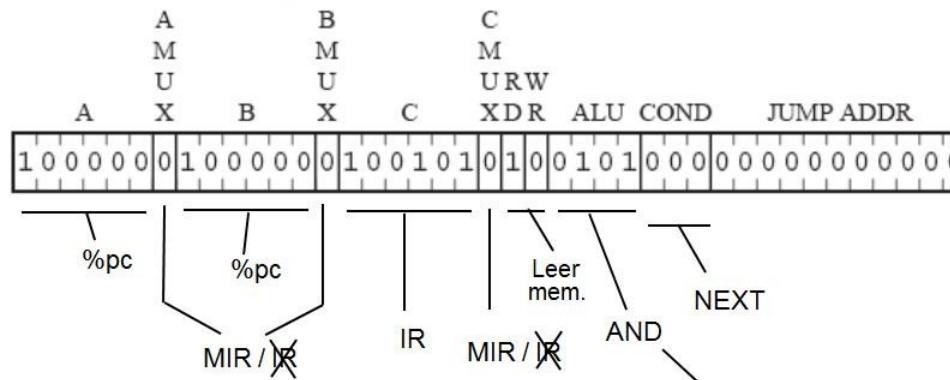
/ Decodificando el Assembler ARC

0: R[ir] = AND(R[pc],R[pc]); READ;

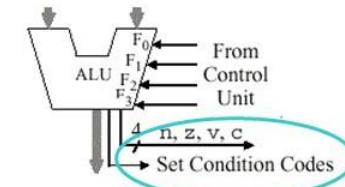
/ Lee una instrucción desde memoria principal a IR

1: DECODE;

/ Salto a microrutina que decodifica instr. ARC según *opcode*



$F_3\ F_2\ F_1\ F_0$	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	ADDC (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	SMML13 (A)	no
1 1 0 0	SEXT13 (A)	no
1 1 0 1	INC (A)	no
1 1 1 0	INCP (A)	no
1 1 1 1	RSHIFT5 (A)	no



# Microprograma en binario

(Se almacena en Control Store y en el MIR)

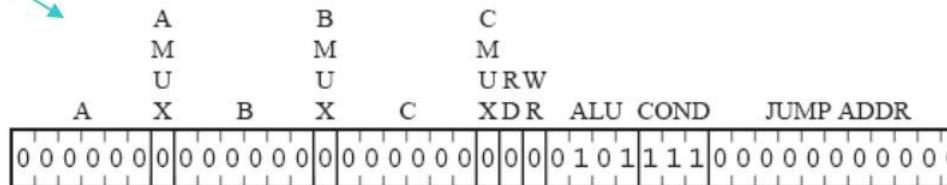
/ Decodificando el Assembler ARC

0: R[ir] = AND(R[pc],R[pc]); READ;

/ Lee una instrucción desde memoria principal a IR

1: DECODE;

/ Salto a microrutina que decodifica instr. ARC según *opcode*

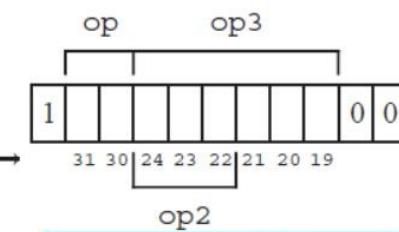


No altera registros

No altera registros

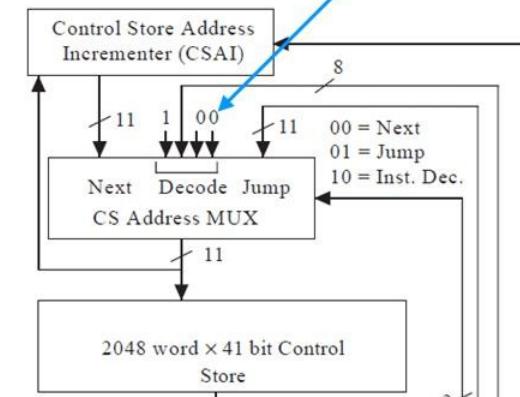
DECODE

IR bits →



Algo que no altere flags

$C_2\ C_1\ C_0$	Operation
0 0 0	Use NEXT ADDR
0 0 1	Use JUMP ADDR if $n = 1$
0 1 0	Use JUMP ADDR if $z = 1$
0 1 1	Use JUMP ADDR if $v = 1$
1 0 0	Use JUMP ADDR if $c = 1$
1 0 1	Use JUMP ADDR if $IR[13] = 1$
1 1 0	Use JUMP ADDR
1 1 1	DECODE

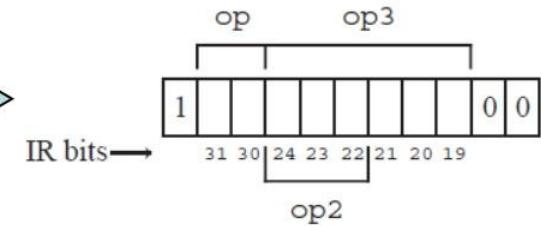
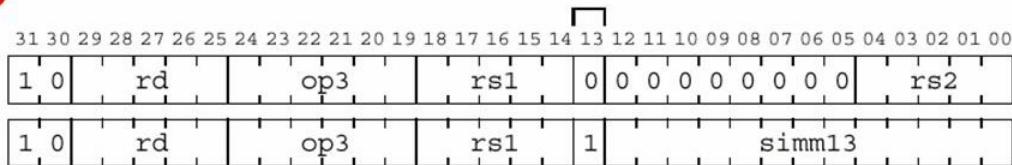


# De la instrucción Assembler a su microcódigo

Addcc

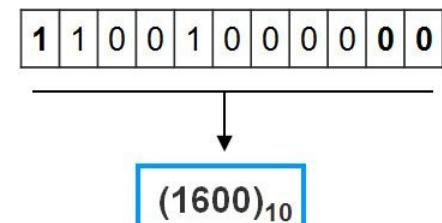
1600:	If R[IR[13]] then GOTO 1602;	/ Si el segundo operando está en modo inmediato salta
1601:	R[rd] = ADDCC(R[rs1],R[rs2]);	/ Realiza ADDCC sobre los registros dados
	GOTO 2047;	
1602:	R[temp0] = SEXT13(R[ir]);	/ Obtiene el campo simm13 y extiende su signo
1603:	R[rd] = ADDCC(R[rs1],R[temp0]);	/ Realiza ADDCC con registro y el simm13
	GOTO 2047;	
2047:	R[pc] = INCPC(R[pc]); GOTO 0;	/ Incrementa %pc y recomienza

1 Determinar la dirección de inicio de la rutina en microcódigo:



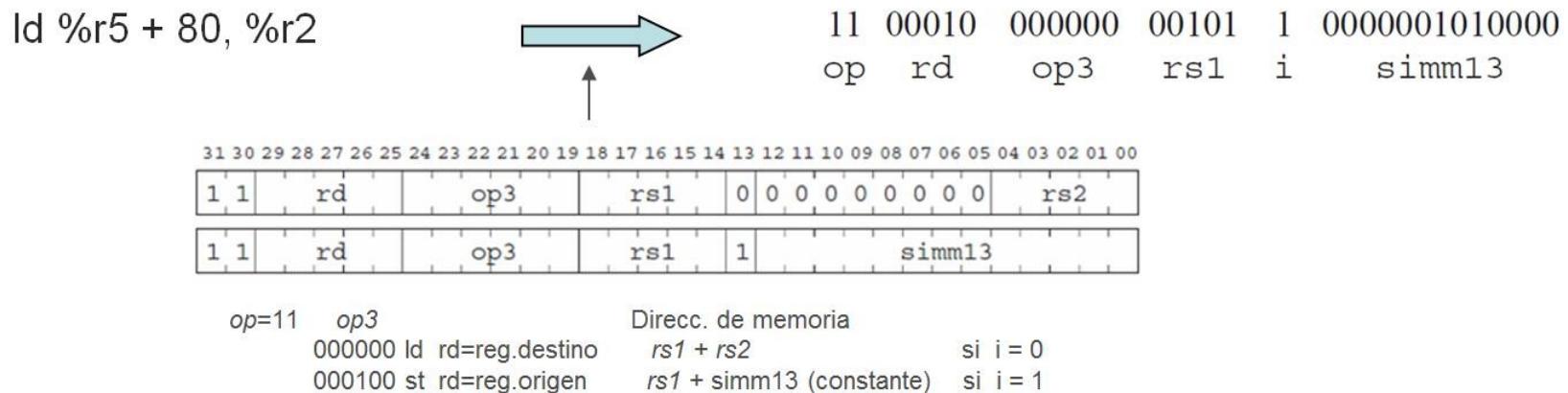
## Formato de instrucciones aritméticas/lógicas

op=10	op3		
010000	addcc	1er. registro origen = rs1	
010001	andcc	2do reg. origen = rs2	si i=0
010010	orcc	2do reg. origen = constante <i>simm13</i>	si i=1
010110	orncc	Reg. destino = rd	
100110	srl		
111000	jmpl		



# De la instrucción Assembler a su microcódigo

## Lectura de memoria principal



/ |d

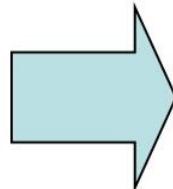
- |       |                                     |  |
|-------|-------------------------------------|--|
| 1792: | $R[temp0] = ADD(R[rs1], R[rs2]);$   | / Calcula dirección a leer con dos registros puntero |
|       | If $R[IR[13]]$ Then GOTO 1794;      | / Si es con (registro + constante) salta             |
| 1793: | $R[rd] = AND(R[temp0], R[temp0]);$  | / Coloca la dirección en el bus A                    |
|       | READ; GOTO 2047;                    | / Lee el dato al registro rd y termina               |
| 1794: | $R[temp0] = SEXT13(R[ir]);$         | / Obtiene el campo simm13 para la dirección a leer   |
| 1795: | $R[temp0] = ADD(R[rs1], R[temp0]);$ | / Calcula la dirección y salta                       |
|       | GOTO 1793;                          |  |
|       |                                     |  |
| 2047: | $R[pc] = INCPC(R[pc]);$ GOTO 0;     | / Incrementa %pc y recomienza                        |

# De la instrucción Assembler a su microcódigo

## Lectura de memoria principal

Assembler

ld %r5 + 80, %r2

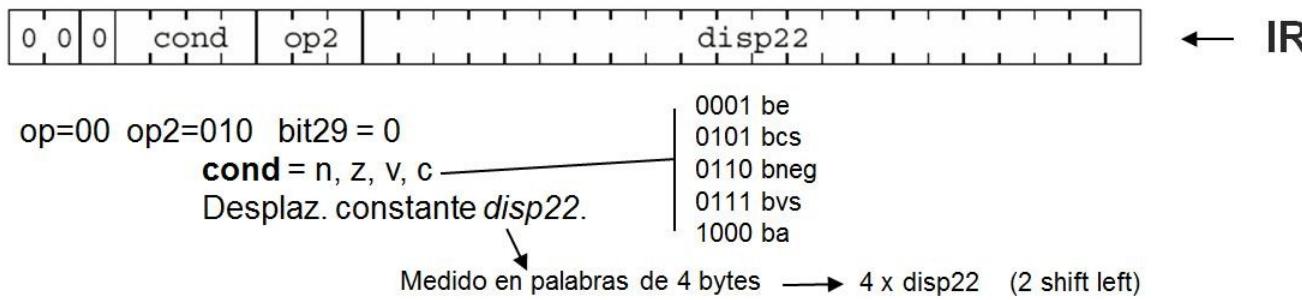


Ejecución en microcódigo

```
0: R[ir] ← AND(R[pc], R[pc]); READ;  
1: DECODE;  
   / 1d  
1792: R[temp0] ← ADD(R[rs1], R[rs2]);  
      IF R[IR[13]] THEN GOTO 1794;  
1793: R[rd] ← AND(R[temp0], R[temp0]);  
      READ; GOTO 2047;  
1794: R[temp0] ← SEXT13(R[ir]);  
1795: R[temp0] ← ADD(R[rs1], R[temp0]);  
      GOTO 1793;  
  
2047: R[pc] = INCPC(R[pc]); GOTO 0;
```

# De la instrucción Assembler a su microcódigo

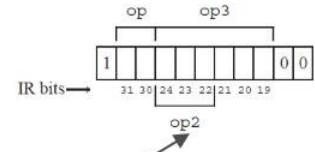
## Saltos condicionales



*n, z, v, c*

Todos los saltos condicionales apuntan al mismo microcódigo según  
 => **debo extraer cond**

Para conocer la dirección de destino => **debo extraer disp22**

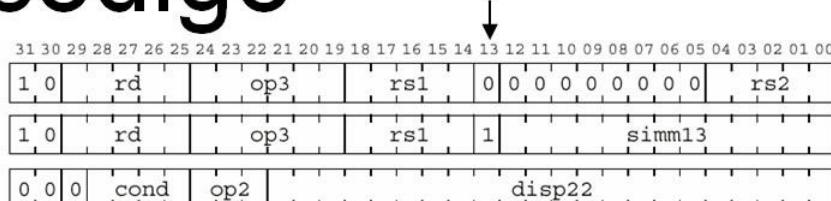


- Para extraer *cond* => leerlo bit a bit con desplazamientos y *bit[13]*
- Para extraer *disp22* => Desplazo a izq, 10 bits y luego desplazo a derecha

# De la instrucción Assembler a su microcódigo

## **Saltos condicionales**

- |                                     |   |
|-------------------------------------|---|
| 1088: GOTO 2;                       | / Decoding tree for branches              |
| 2: R[temp0] = LSHIFT10(R[ir]);      | / Sign extend the 22 LSB's of %temp0      |
| 3: R[temp0] = RSHIFT5(R[temp0]);    | / by shifting left 10 bits, then right 10 |
| 4: R[temp0] = RSHIFT5(R[temp0]);    | / bits. RSHIFT5 does sign extension.      |
| 5: R[ir] = RSHIFT5(R[ir]);          | / Move COND field to IR[13] by            |
| 6: R[ir] = RSHIFT5(R[ir]);          | / applying RSHIFT5 three times. (The      |
| 7: R[ir] = RSHIFT5(R[ir]);          | / sign extension is inconsequential.)     |
| 8: IF R[IR[13]] THEN GOTO 12;       | / Is it ba?                               |
| R[ir] = ADD(R[ir],R[ir]);           |   |
| 9: IF R[IR[13]] THEN GOTO 13;       | / Is it not be?                           |
| R[ir] = ADD(R[ir],R[ir]);           |   |
| 10: IF Z THEN GOTO 12;              | / Execute be                              |
| R[ir] = ADD(R[ir],R[ir]);           |   |
| 11: GOTO 2047;                      | / Branch for be not taken                 |
| 12: R[pc] = ADD(R[pc],R[temp0]);    | / Branch is taken                         |
| GOTO 0;                             |   |
| 13: IF R[IR[13]] THEN GOTO 16;      | / Is it bcs?                              |
| R[ir] = ADD(R[ir],R[ir]);           |   |
| 14: IF C THEN GOTO 12;              | / Execute bcs                             |
| 15: GOTO 2047;                      | / Branch for bcs not taken                |
| 16: IF R[IR[13]] THEN GOTO 19;      | / Is it bvs?                              |
| 17: IF N THEN GOTO 12;              | / Execute bneg                            |
| 18: GOTO 2047;                      | / Branch for bneg not taken               |
| 19: IF V THEN GOTO 12;              | / Execute bvs                             |
| 20: GOTO 2047;                      | / Branch for bvs not taken                |
| 2047: R[pc] = INCPC(R[pc]); GOTO 0; | / Increment %pc and start over            |



%temp0=disp22

15 bits  
a der

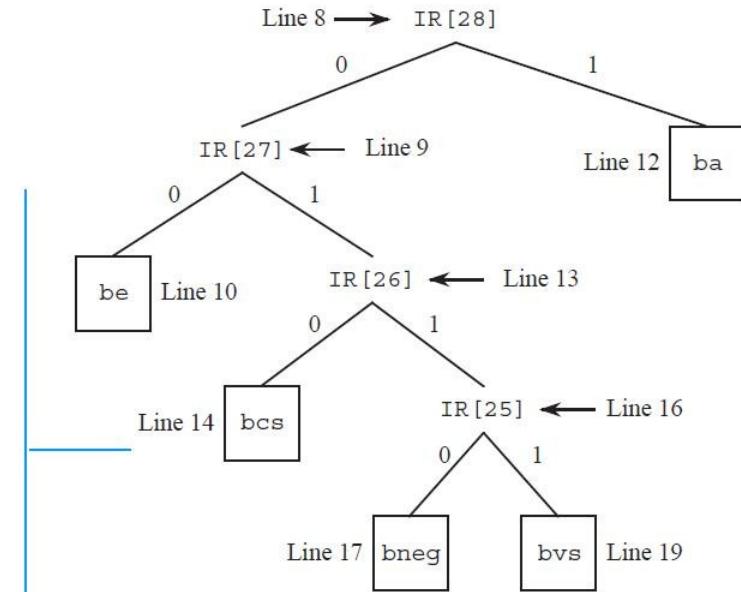
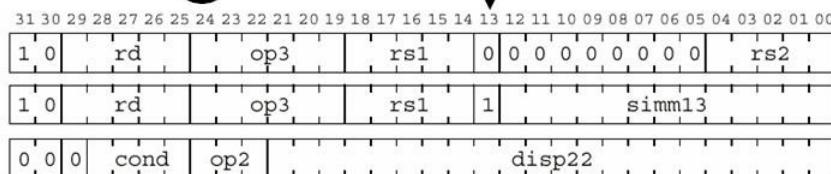
Lee COND bit a bit

**Desplaza izq de a 1 bit por vez**

# De la instrucción Assembler a su microcódigo

## Saltos condicionales

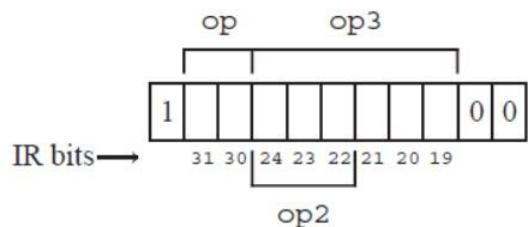
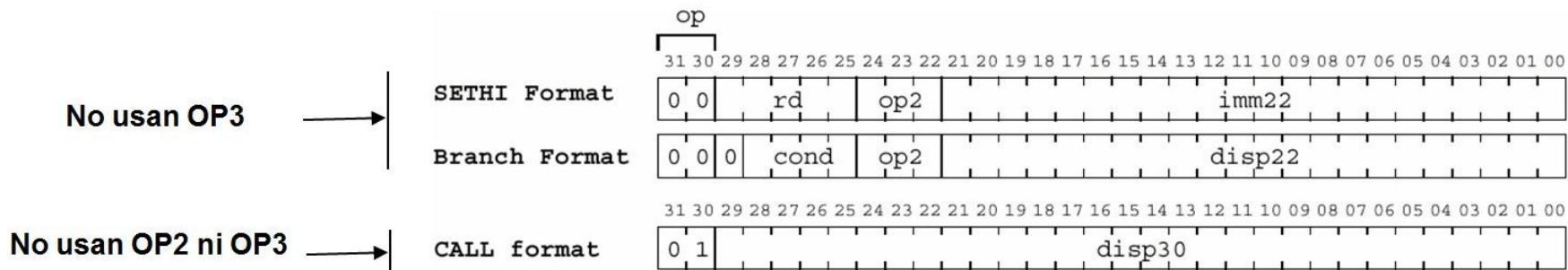
- 1088: GOTO 2; / Decoding tree for branches
- 2: R[temp0] = LSHIFT10(R[ir]); / Sign extend the 22 LSB's of %temp0
- 3: R[temp0] = RSHIFT5(R[temp0]); / by shifting left 10 bits, then right 10
- 4: R[temp0] = RSHIFT5(R[temp0]); / bits. RSHIFT5 does sign extension.
- 5: R[ir] = RSHIFT5(R[ir]); / Move COND field to IR[13] by
- 6: R[ir] = RSHIFT5(R[ir]); / applying RSHIFT5 three times. (The
- 7: R[ir] = RSHIFT5(R[ir]); / sign extension is inconsequential.)
- 8: IF R[IR[13]] THEN GOTO 12; / Is it ba?
- R[ir] = ADD(R[ir],R[ir]);
- 9: IF R[IR[13]] THEN GOTO 13; / Is it not be?
- R[ir] = ADD(R[ir],R[ir]);
- 10: IF Z THEN GOTO 12; / Execute be
- R[ir] = ADD(R[ir],R[ir]);
- 11: GOTO 2047; ← / Branch for be not taken
- 12: R[pc] = ADD(R[pc],R[temp0]); / Branch is taken
- GOTO 0; ←
- 13: IF R[IR[13]] THEN GOTO 16; / Is it bcs?
- R[ir] = ADD(R[ir],R[ir]);
- 14: IF C THEN GOTO 12; / Execute bcs
- 15: GOTO 2047; / Branch for bcs not taken
- 16: IF R[IR[13]] THEN GOTO 19; / Is it bvs?
- 17: IF N THEN GOTO 12; / Execute bneg
- 18: GOTO 2047; / Branch for bneg not taken
- 19: IF V THEN GOTO 12; / Execute bvs
- 20: GOTO 2047; / Branch for bvs not taken
- 2047: R[pc] = INCPC(R[pc]); GOTO 0; / Increment %pc and start over



cond	branch
28	be
27	bcs
26	bneg
25	bvs
1	ba

# De la instrucción Assembler a su microcódigo

## Entradas duplicadas al Control Store

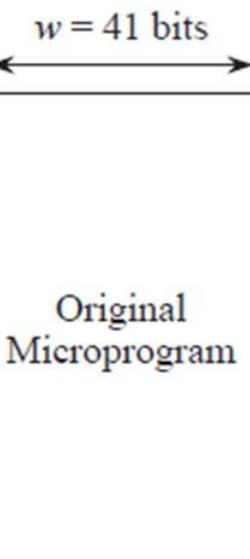


¿Cuál es el problema que generan las entradas duplicadas?  
¿Qué hacer para solucionarlo?

# Nanoprogramación

Sin nanoprogramación

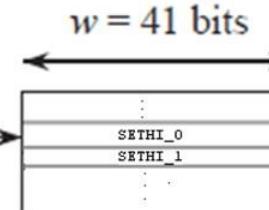
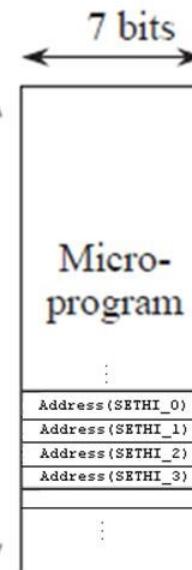
$n = 2048$  words



IR

Con nanoprogramación

$n = 2048$  words



MIR

ESPACIO OCUPADO  
EN EL CHIP

Vs. VELOCIDAD

# **CPU**

## **Diseño Cableado**

Cableado = Flip-Flops + Lógica combinacional

Pasos del microprograma => estados de  
una “máquina de estados finitos”

Diseñar Sección de Control = Definir transiciones entre estados y  
líneas de control

Diseñar Sección de Datos = Producir salidas para cada estado

# *Diseño del CPU*

CPU Microprogramada vs. CPU Cableada →

- Velocidad
- Facilidad para actualización
- Costo y complejidad

