

# 66.70 Estructura del Computador

## **Microarquitectura**

```

    push ebp                                ;function prologue
    mov ebp, esp
    add esp, -12

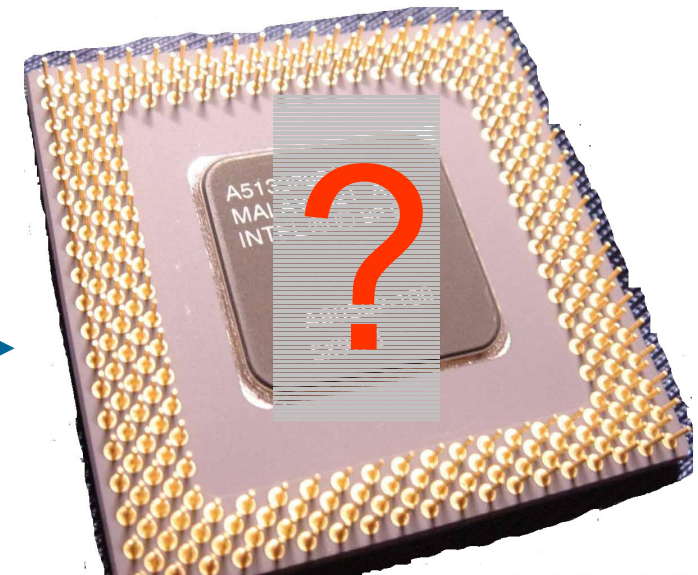
    cmp DWORD PTR [ebp + 8], 2             ;if (x<2)
    jl  $IF_TRUE0                          ;check if x<2
$IF_FALSE0:
    mov eax, 0                             ;fibonacci(x-1)+fibonacci(x-2);
    mov eax, DWORD PTR [ebp + 8]          ;calculate x-1
    sub eax, 1
    push eax                               ;push it on as an argument
    call _fib__fibonacci                  ;make the call to fib(x-1)
    mov DWORD PTR [ebp-8], eax             ;save the returned value in [ebp-8]

    mov eax, DWORD PTR [ebp-8]            ;calculate x-2
    sub eax, 2
    push eax                               ;push it on as an argument
    call _fib__fibonacci                  ;make the call to fib(x-2)
    mov DWORD PTR [ebp-12], eax            ;save the returned value in [ebp-12]
    mov eax, DWORD PTR [ebp-12]
    add eax, DWORD PTR [ebp-8]             ;move fib(x-1)+fib(x-2) into eax
    mov DWORD PTR [ebp-4], eax             ;move the value into a temporary for returning
$IF_END0:
    mov eax, DWORD PTR [ebp-4]             ;move the temporary with the return
                                           ;value into the return register

$fibonacci_epi:
    mov esp, ebp;function epilogue
    pop ebp
    ret
$IF_TRUE0:

```

ISA



ISA

- *Set de instrucciones*
- *Hardware visible al **programador***

MICROARQUITECTURA

Implementación 1  
en hardware

*p.e.: mayor velocidad*

Implementación 2  
en hardware

*p.e.: menor costo*

...

Implementación N  
en hardware

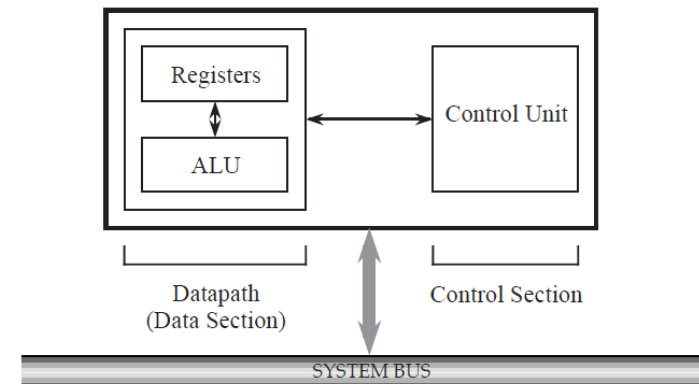
*p.e.: menor consumo*

← **Énfasis en**

# 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: *ciclo de fetch***



# Abordajes al diseño de la microarquitectura

Podemos diseñar la unidad de control según:

→ *Control microprogramado*

→ *Lógica cableada*

El trayecto de datos es esencialmente idéntico  
en ambos casos

# Implementación del Trayecto de Datos

## Estructuras para realizar el movimiento de datos

Movimiento de datos  
externo al CPU



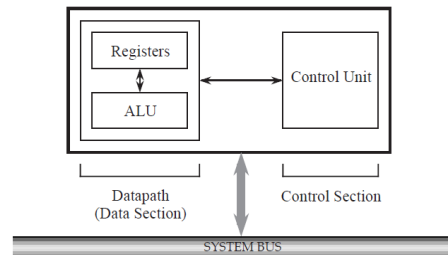
BUS DE SISTEMA

Movimiento de datos  
dentro del CPU

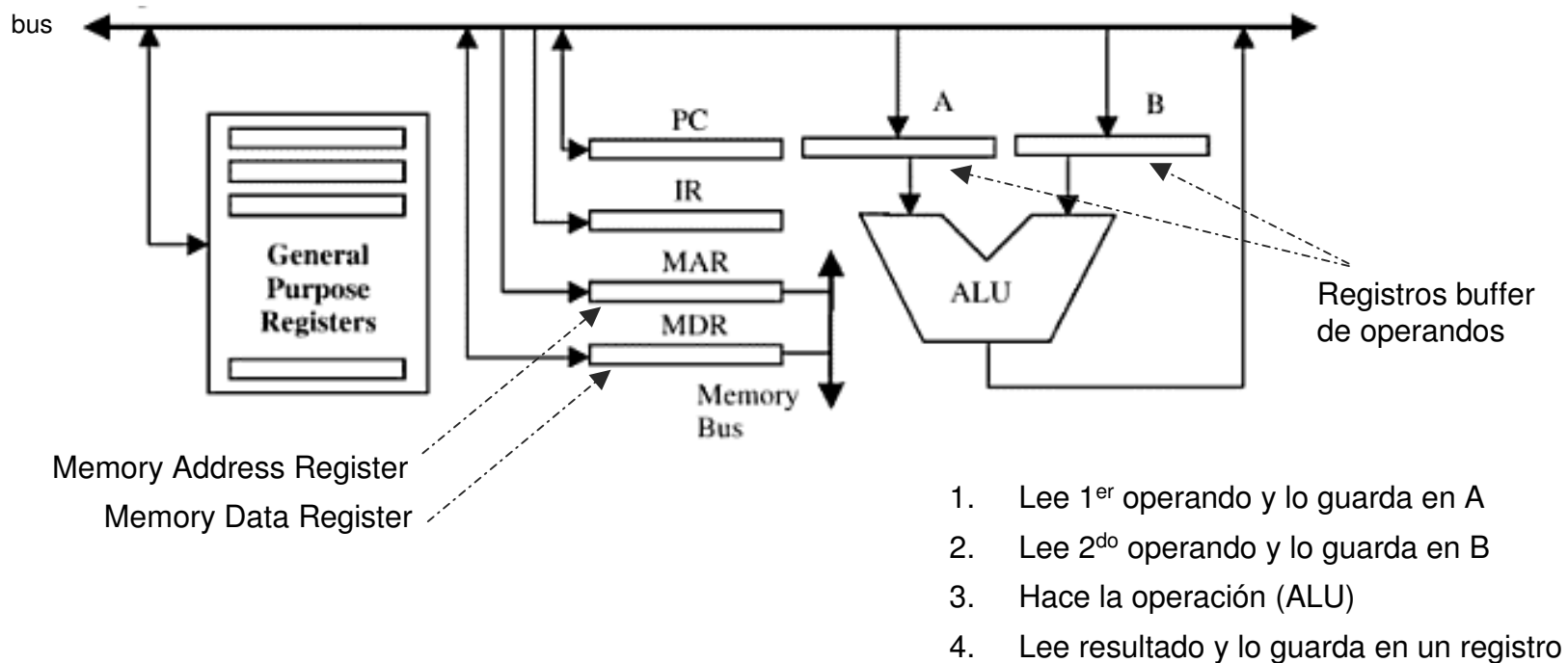


BUSES INTERNOS al CPU

- *Organizado en 1 bus*
- *Organizado en 2 buses*
- *Organizado en 3 buses*



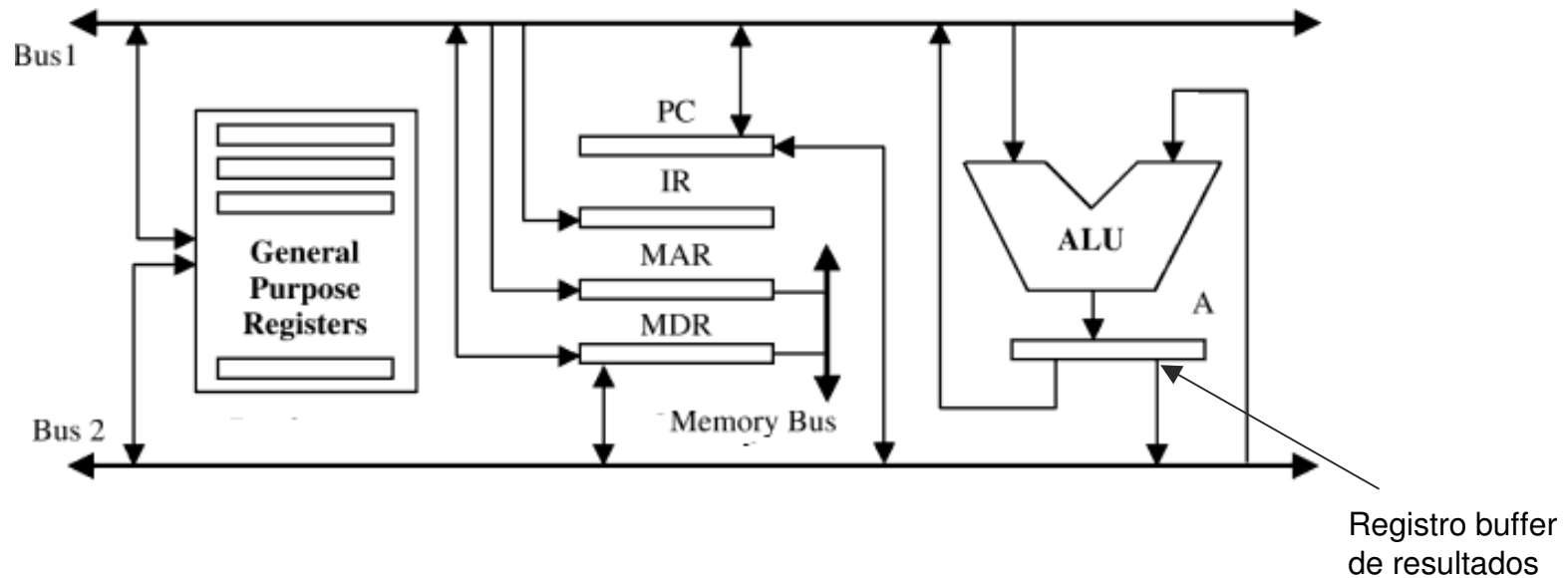
# Trayecto de Datos Organizado en 1 bus



– Simple y de bajo costo

– Limitado flujo de datos por ciclo de reloj → Baja performance

# Trayecto de Datos Organizado en 2 buses



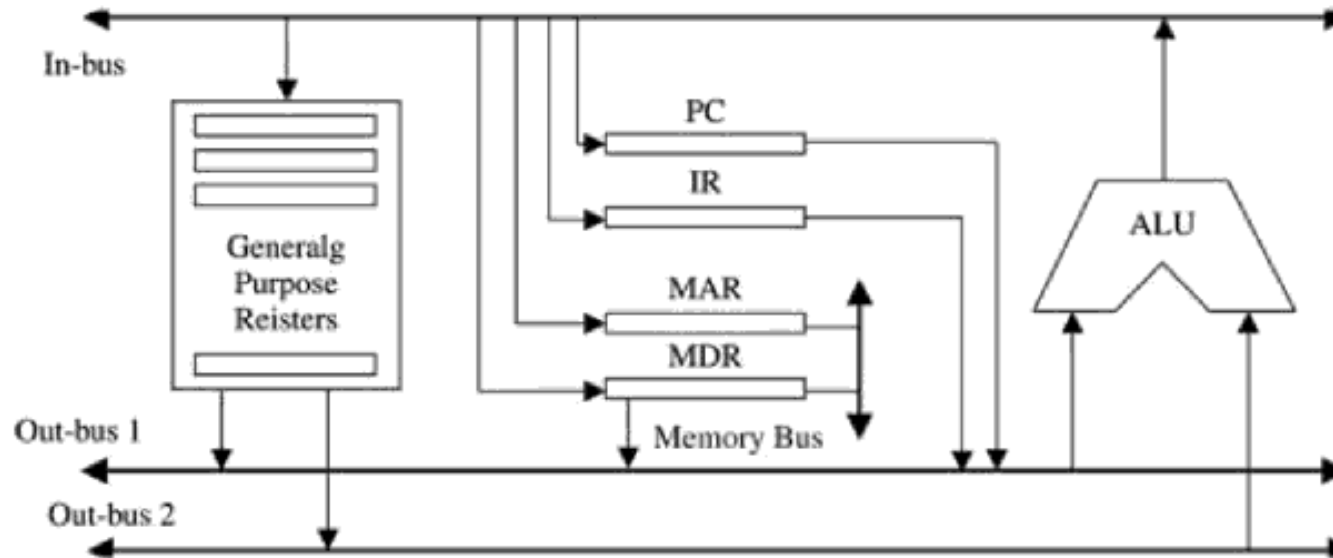
- *Obtiene los 2 operandos en el mismo ciclo de reloj*
- *Buffer para el resultado (bus1 y bus2 ocupados con operandos)*
- *Más rápido que organización en 1 bus*



# Trayecto de Datos Organizado en 3 buses

*Out-bus:* Dato sale de registro

*In-bus:* Dato entra a registro



- *Mueve 2 operandos y resultado en el mismo ciclo de reloj*
- *Buena performance*
- *Mayor complejidad de hardware*

## Esta implementación

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

# La Unidad Aritmético-Lógica

## Cuestiones relacionadas a su implementación

### **Performance**

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

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

### **Funcionalidad**

- *Debe: mover datos y hacer operaciones aritméticas/lógicas*
- *Operaciones básicas de ALU para obtener el set de instrucciones dado  
(no necesariamente coinciden con las del ISA)*

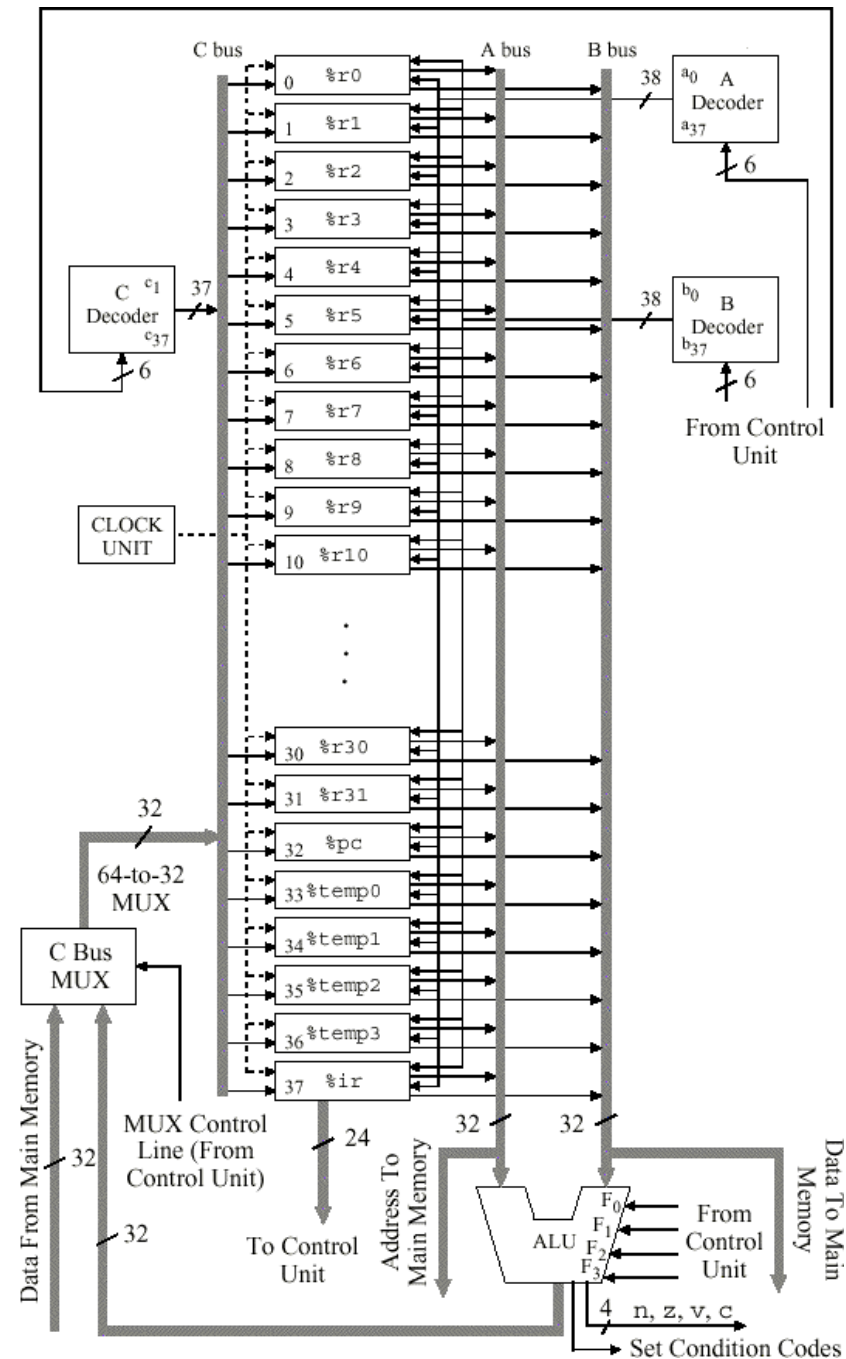
### **Implementación**

- *Con 32 módulos de 1 bit puede obtenerse una ALU de 32 bits*
- *Puede basarse en el uso de una “Look-up table”*
- *Definir estructura de buses*

# Una microarquitectura ARC

## *Trayecto de Datos*

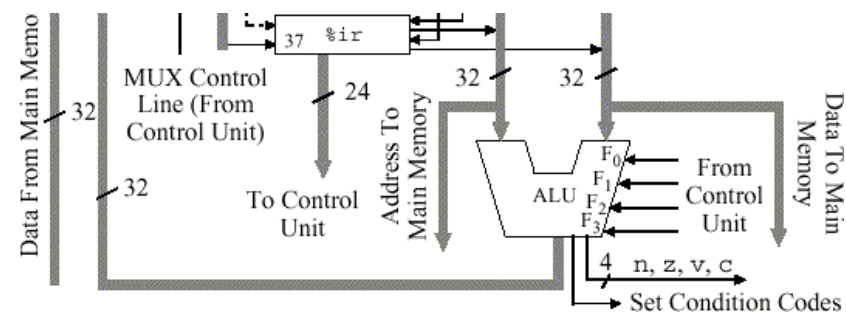
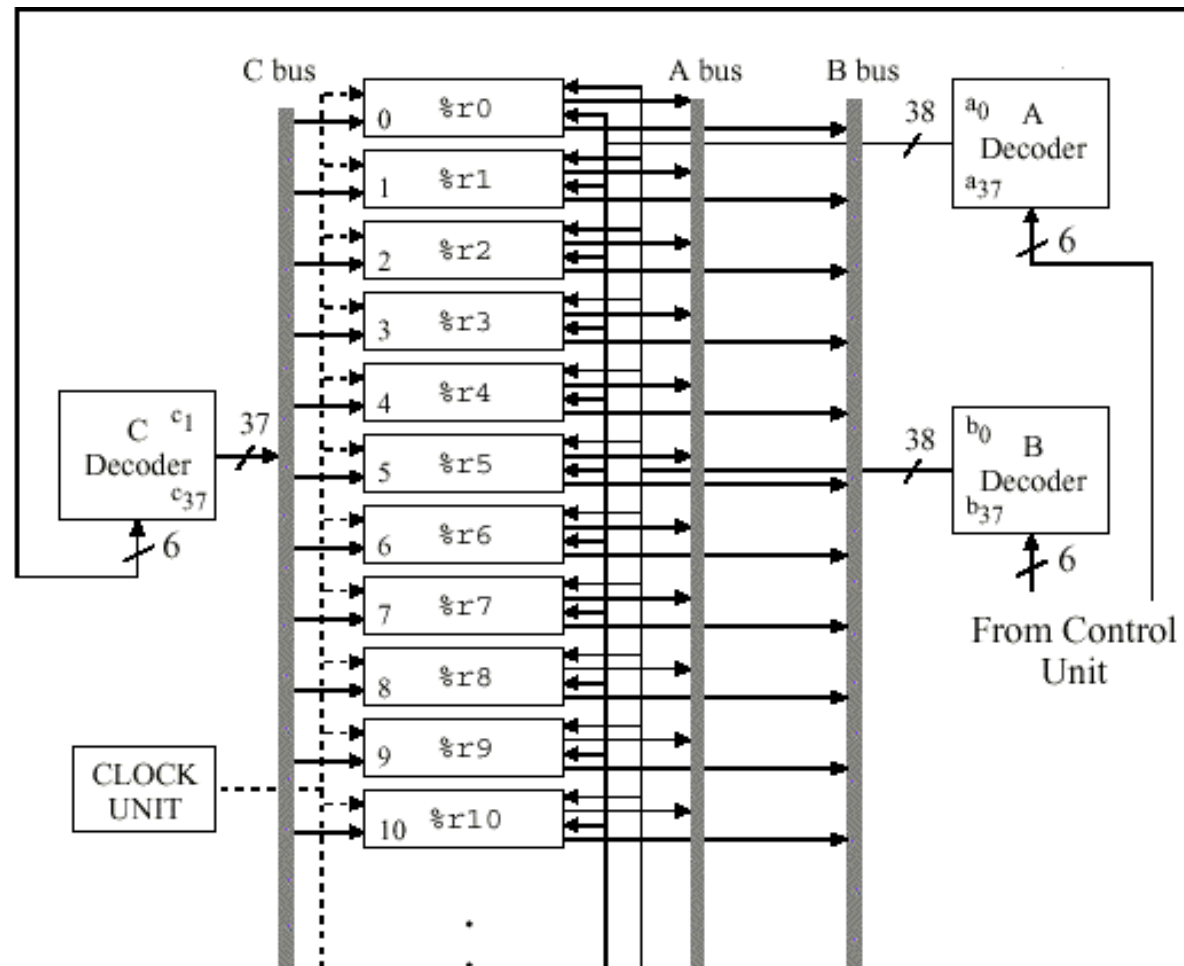
- ✓ Buses
- ✓ ALU
- ✓ Registros
- ✓ Decodificadores
- ✓ Multiplexores



# Una microarquitectura ARC

## Trayecto de Datos

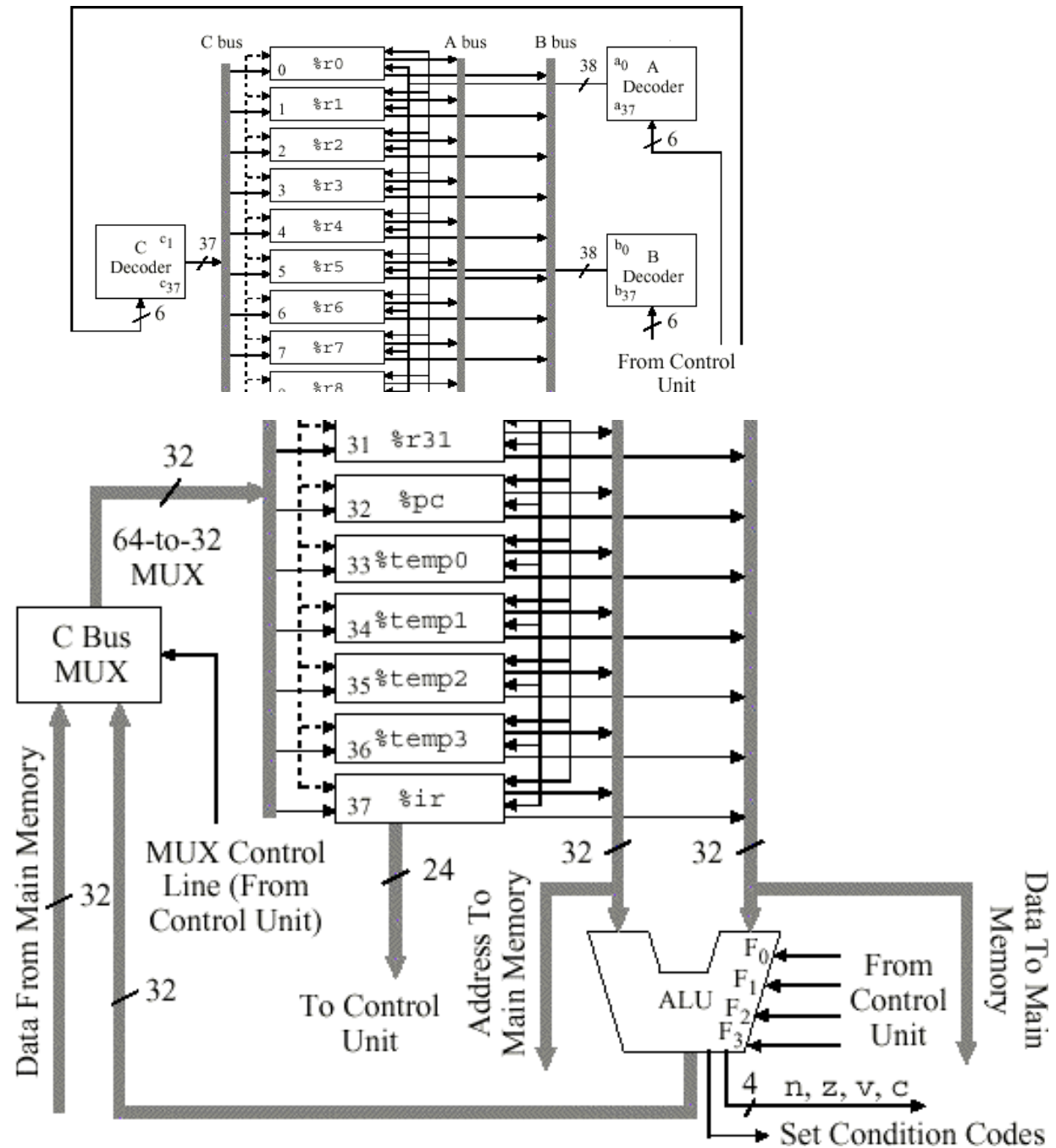
- ✓ Buses
- ✓ ALU
- ✓ Registros
- ✓ Decodificadores
- ✓ Multiplexores



# Una microarquitectura ARC

## Trayecto de Datos

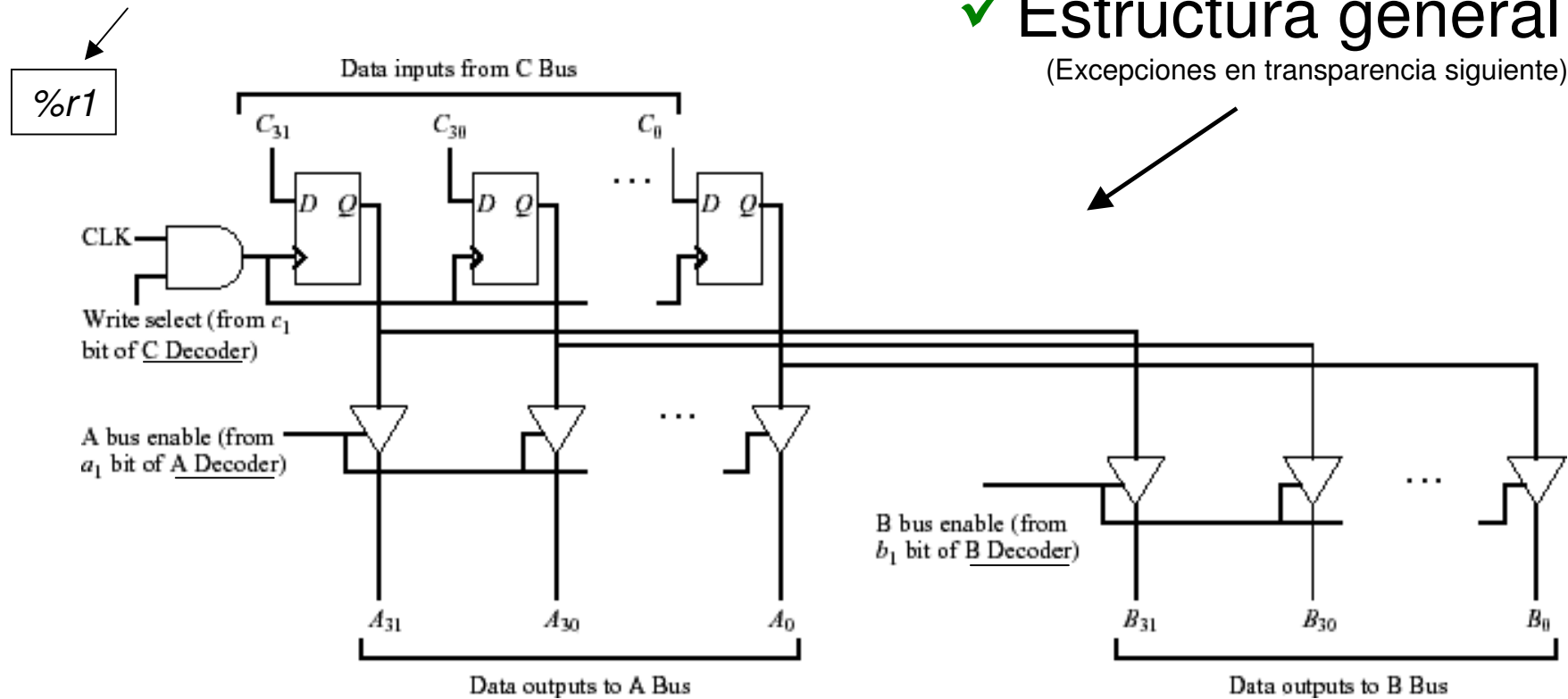
- ✓ Buses
- ✓ ALU
- ✓ Registros
- ✓ Decodificadores
- ✓ Multiplexores



# ARC

## Los registros

¿Porqué digo que es %r1 ?



- Flip-Flops D flanco descendente
- Escritura desde bus C habilitada por el correspondiente bit del decodificador C
- Lectura a buses A y B (a través de buffers Tri-state)

# ARC

## 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

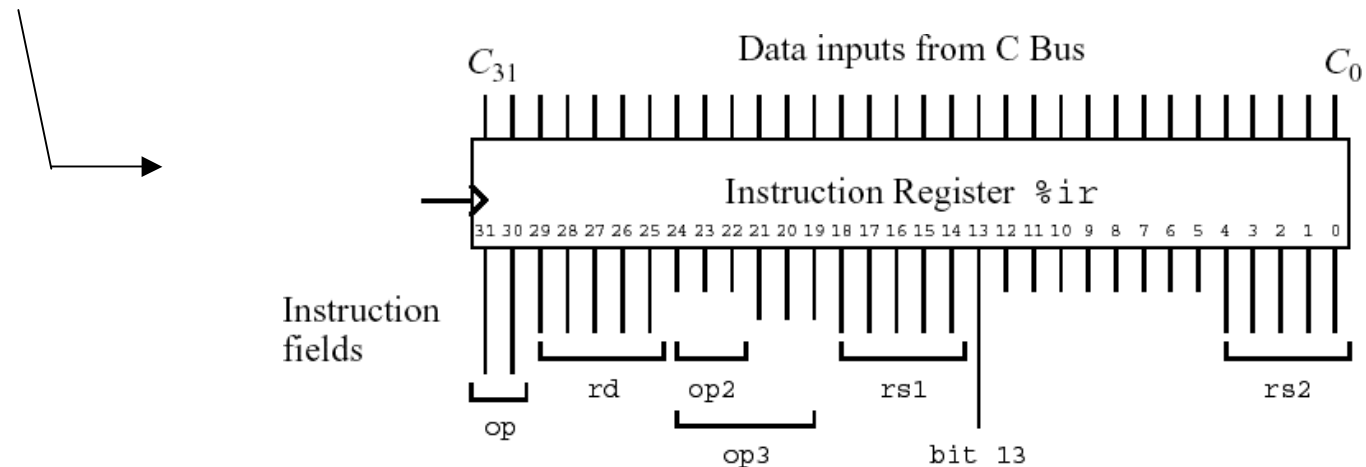
### ✓ Contador de programa **PC**

- Sólo almacena números múltiplos de 4

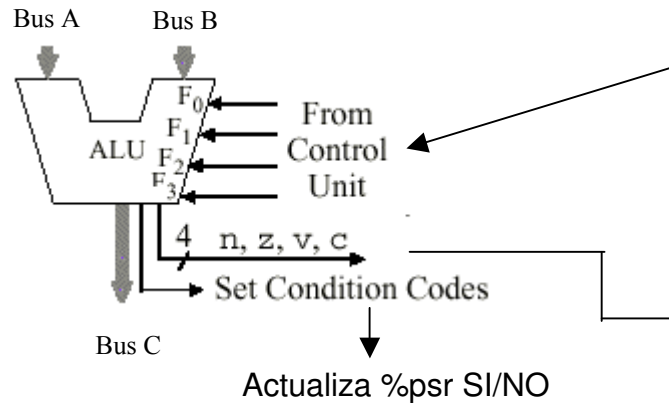
=> los 2 LSB cableados a cero

### ✓ Registro de instrucciones **IR**

- Tiene salidas específicas para campos del código de máquina



# La Unidad Aritmético-Lógica



## Operaciones a implementar en 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

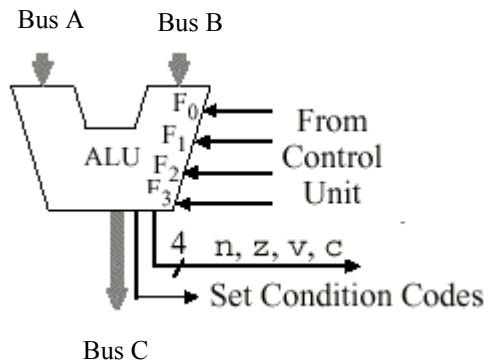
RSHIFT5: Shift right 5 pos. y ext. signo



# ARC

## ALU: *Funcionalidad*

# 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), %templ INC %templ ADDCC (%rtempl, %r1), %r1
Shift Left 1 bit	ADD (%r2, %r2), %r2


# Implementando una ALU

## Funciones que debe realizar:

- Operaciones aritméticas y lógicas
- Desplazamientos a derecha e izquierda

## Soluciones alternativas:

- Para las operaciones aritmético-lógicas
  - Circuitos sumadores, restadores, etc. que vimos anteriormente
  - Usar una “look-up table”
- Para los desplazamientos
  - Registros de desplazamiento
  - Desplazador rápido o “Barrel-shifter”



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

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

## LUT

- operaciones **aritmético-lógicas**

- 2 entradas de 32 bits (operandos)
- entrada de control: 4 bits (elige operación)
- salida de 32 bits (resultado)
- Salida CC: n, z, v, c (código de condición)
- Salida SCC (set CC): 1 bit (setea %psr: si/no )
- Implementable con 32 módulos de 1 bit

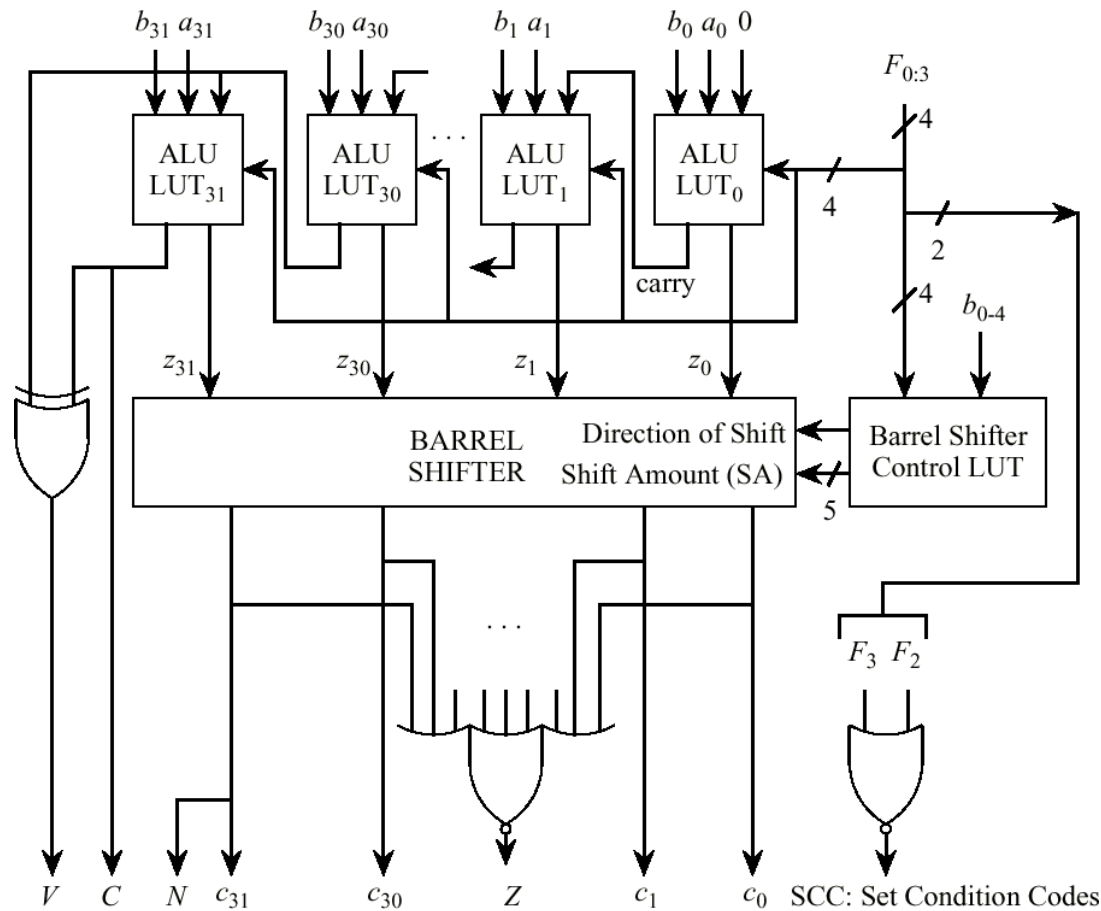
## Desplazador rápido (“Barrel Shifter”)

- **desplazamientos der/izq. de 0 a 31 bits en un solo ciclo de reloj**
  - entrada de operando: 32 bits
  - entrada n° bits despl.: 5 bits
  - entrada de dirección: 1 bit
  - salida 32 bits

ARC

ALU: *Implementación*

# ALU implementada con “look-up table” (LUT) 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)

SCC: Set Condition Codes

ARC

ALU: *Implementación*

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

		$F_3$	$F_2$	$F_1$	$F_0$	$Carry$ $In$	$a_i$	$b_i$	$z_i$	$Carry$ $Out$
ANDCC		0	0	0	0	0	0	0	0	0
		0	0	0	0	0	0	1	0	0
		0	0	0	0	0	1	0	0	0
		0	0	0	0	0	1	1	1	0
		0	0	0	0	1	0	0	0	0
		0	0	0	0	1	0	1	0	0
		0	0	0	0	1	1	0	0	0
		0	0	0	0	1	1	1	1	0
ORCC		0	0	0	1	0	0	0	0	0
		0	0	0	1	0	0	1	1	0
		0	0	0	1	0	1	0	1	0
		0	0	0	1	0	1	1	1	0
		0	0	0	1	1	0	0	0	0
		0	0	0	1	1	0	1	1	0
		0	0	0	1	1	1	1	1	0
						.				.
						.				.
						.				.

# Desplazador rápido (Barrel Shifter)

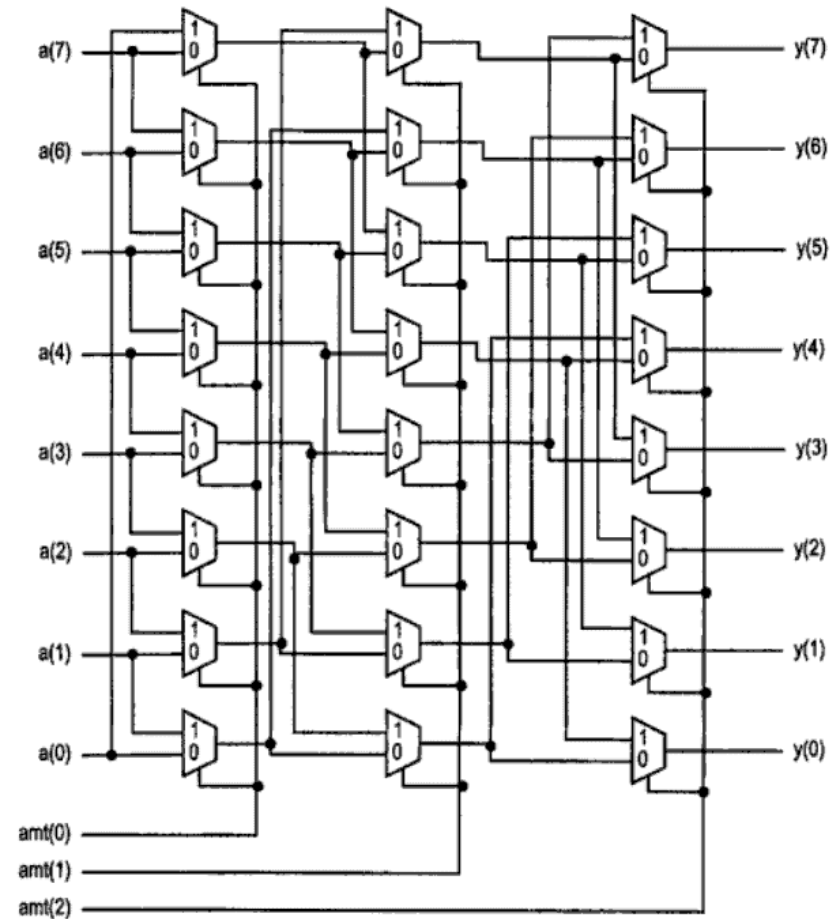
## Funciones

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

## Principio de funcionamiento

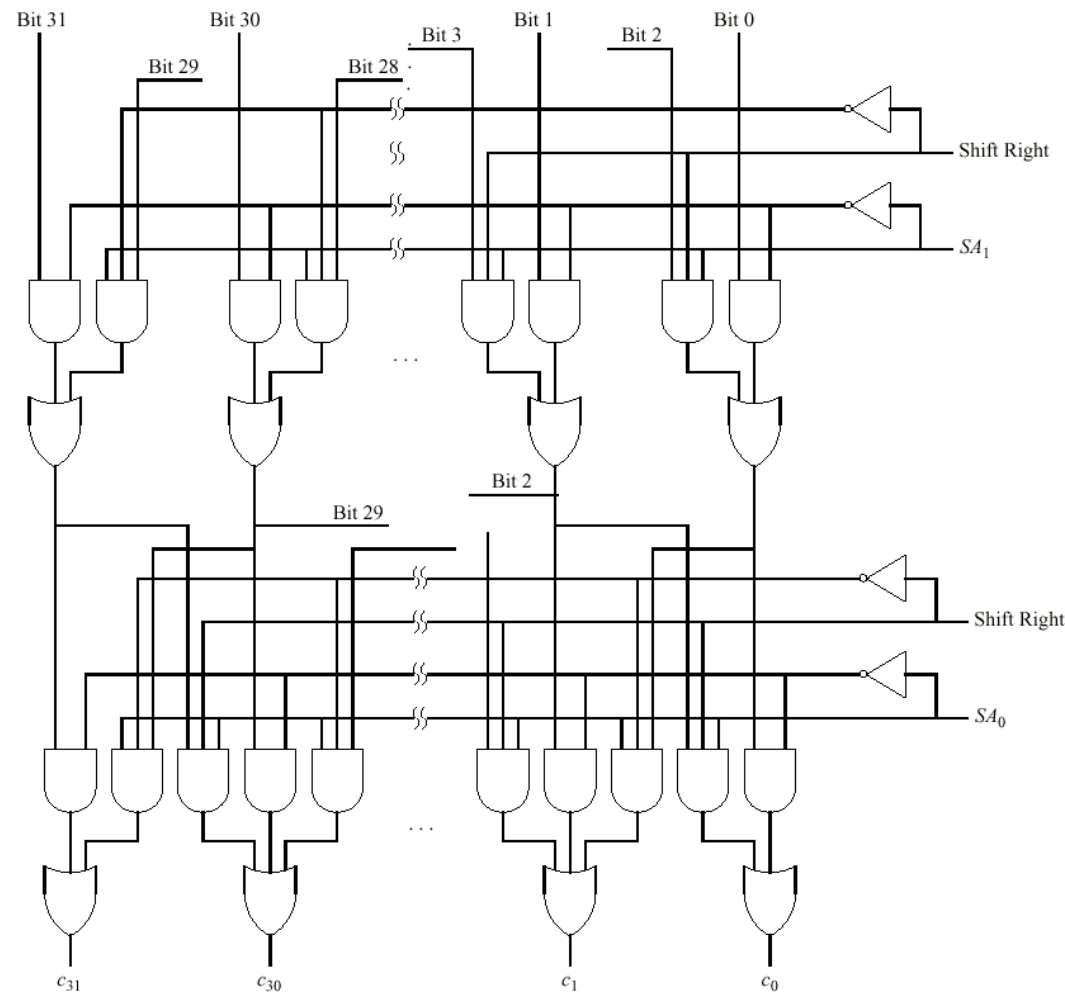
- Organización por niveles
- Cada nivel desplaza  $2^n$  al anterior
- Bits de control definen rango del shift

### *Una implementación posible*



*Shift circular a derecha*

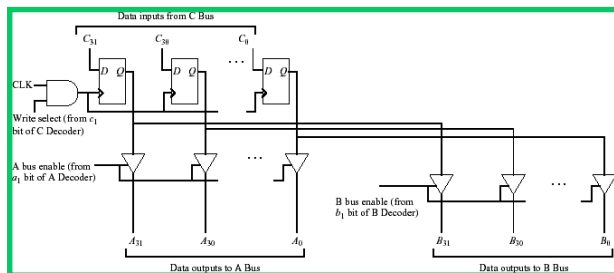
# Desplazador rápido (Barrel Shifter)



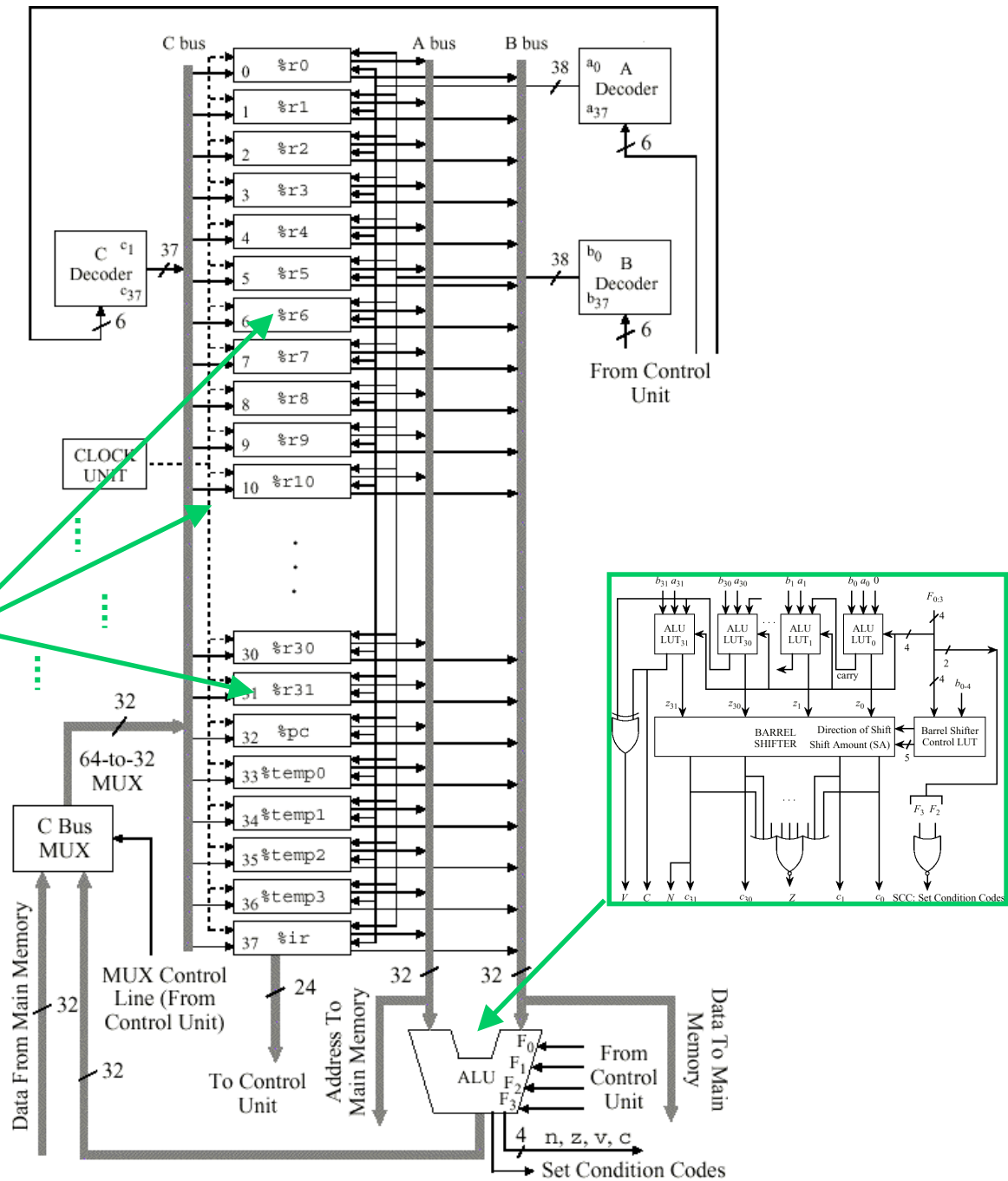
*Implementación presentada en el libro de Murdocca-Heuring*

# Una microarquitectura ARC

## Trayecto de Datos



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



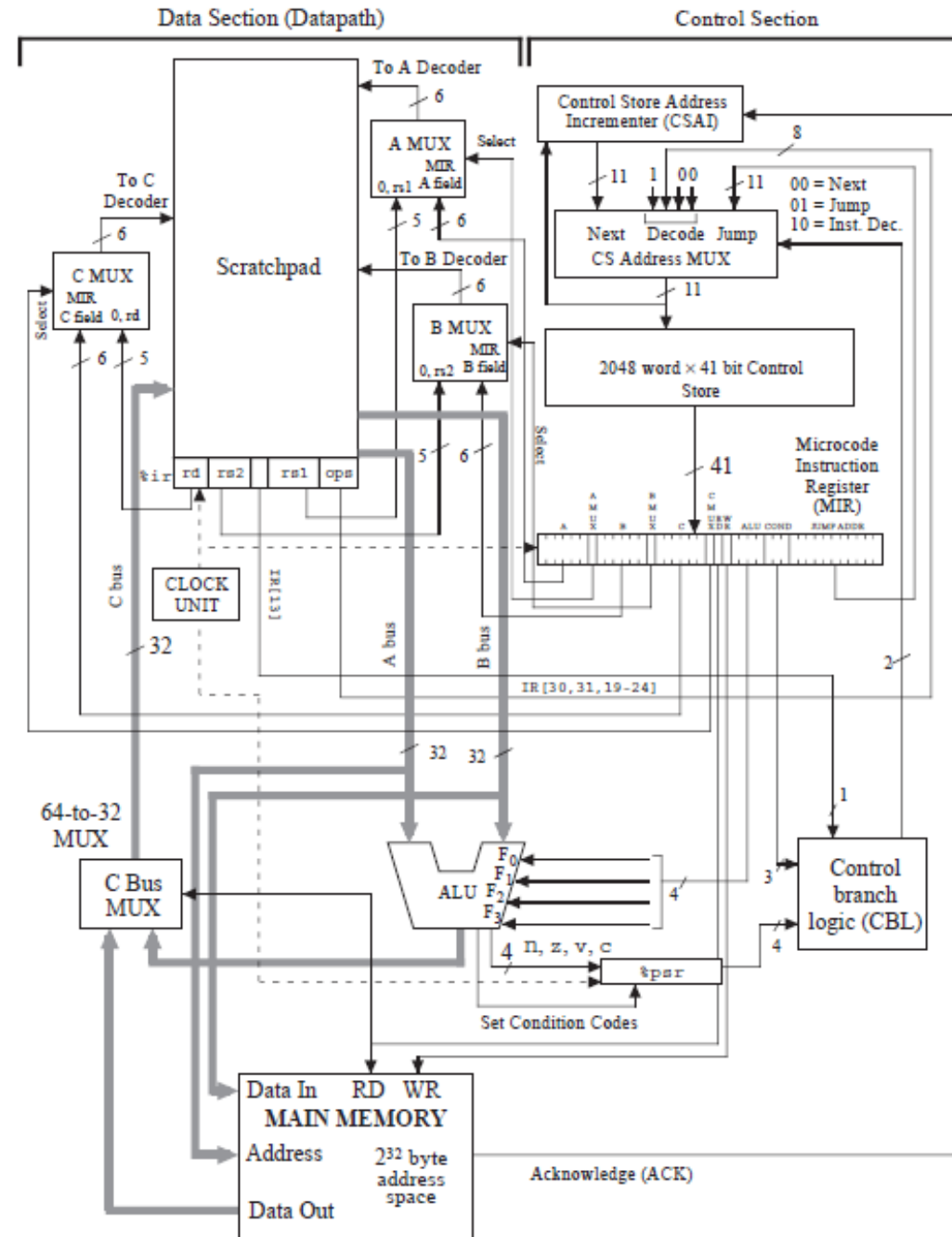


# ARC

## Microarquitectura

# Tr. de Datos + Un. de Control

## Microprogramada



## Control store (ROM)

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

## MIR

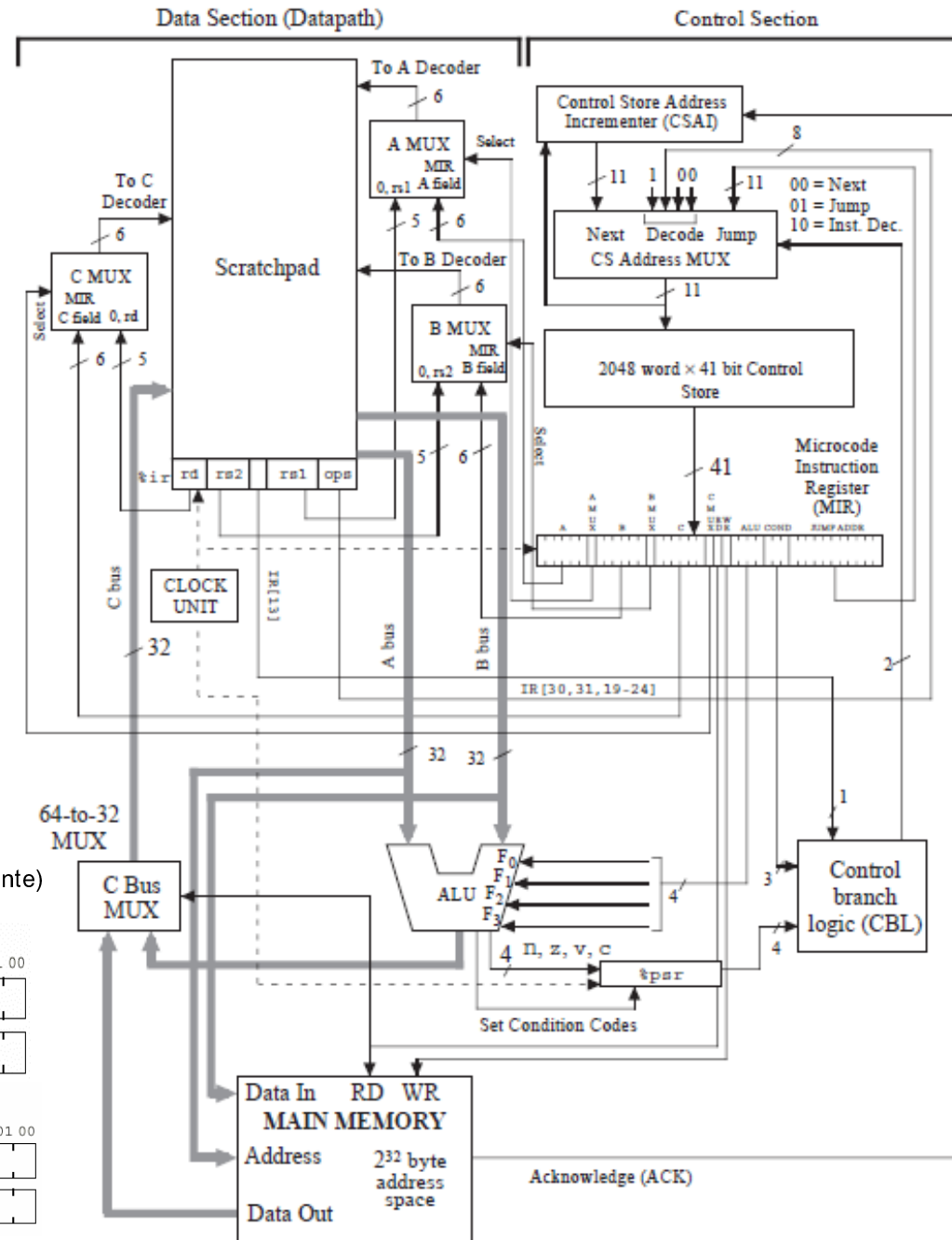
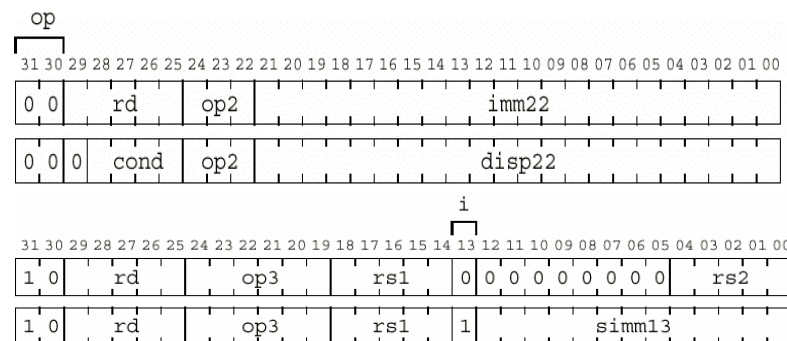
- Guarda microinstrucciones. así como el IR guarda instrucciones del set del procesador

## Ejecución del microcódigo

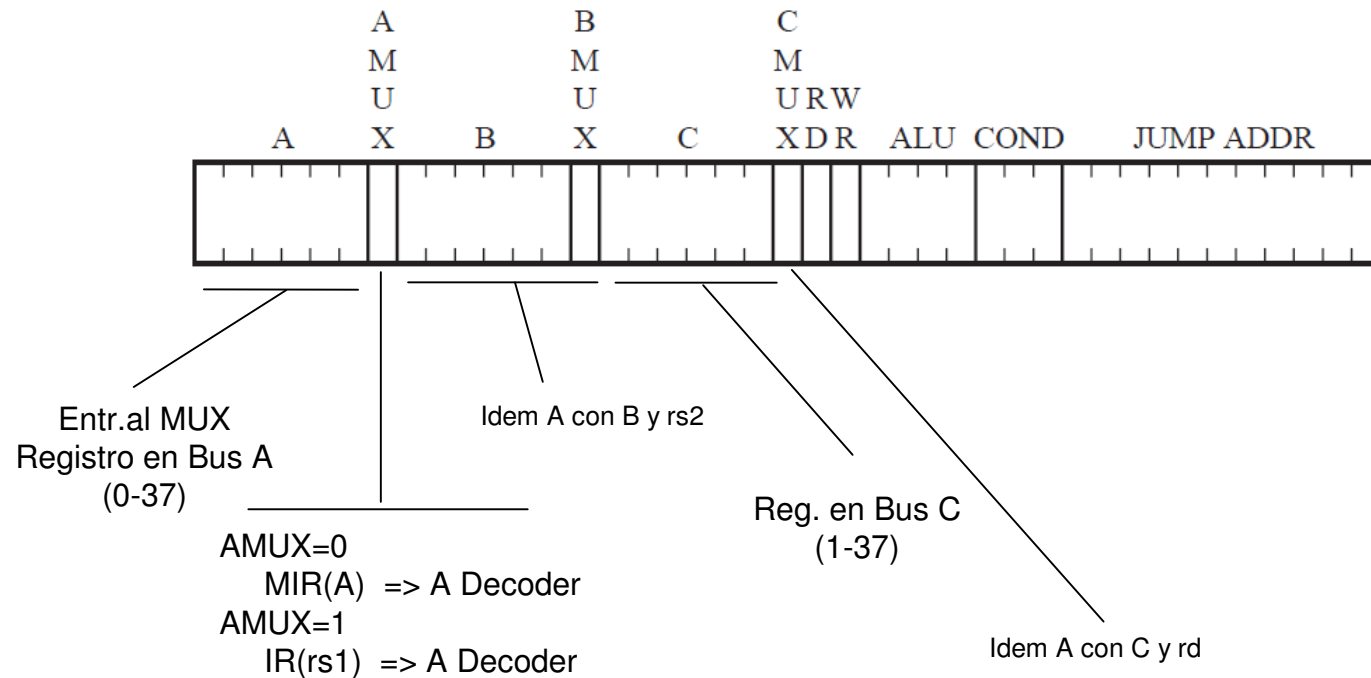
- CS-Addr-MUX apunta a próx. microinstr.
- MicroInstrucción: se copia desde CS al MIR
- Control de flujo: CBL y CS-Addr-MUX

## IR con salidas por cada campo

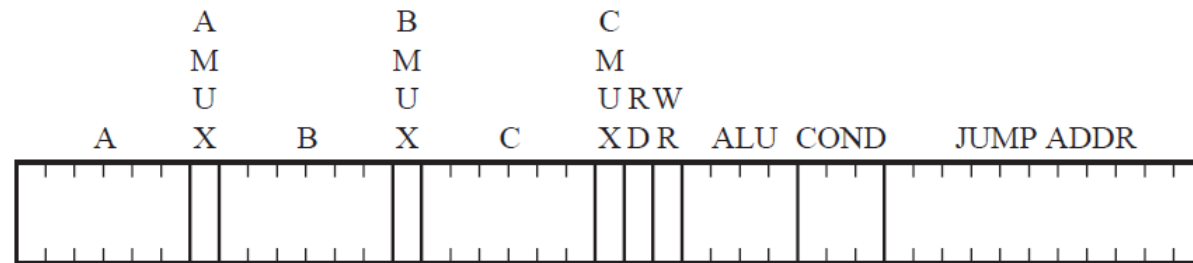
- Registros origen y destino: *rd*, *rs1*, *rs2*
- Si es direccionamiento inmediato: *Bit[13]*
- Códigos de operación: *op*, *op2* y *op3*  
(bits 31-30, 24-22, 24-19 respectivamente)



# Formato de las microinstrucciones



# Formato de las microinstrucciones

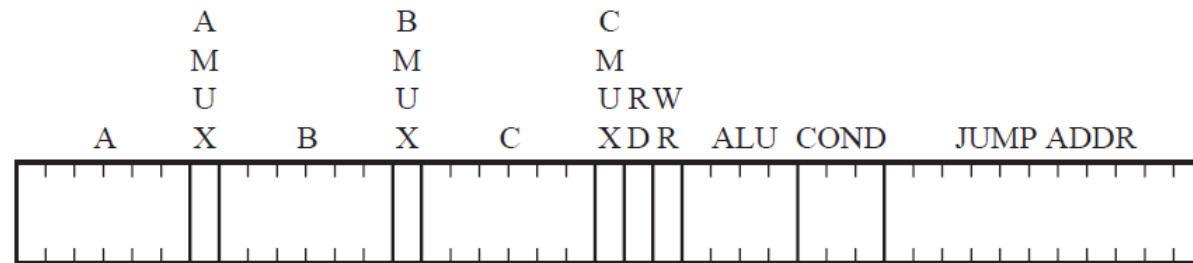


Accesos a memoria

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
- Dato en bus B

# Formato de las microinstrucciones



**Código de operación de ALU**  
16 operaciones posibles

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

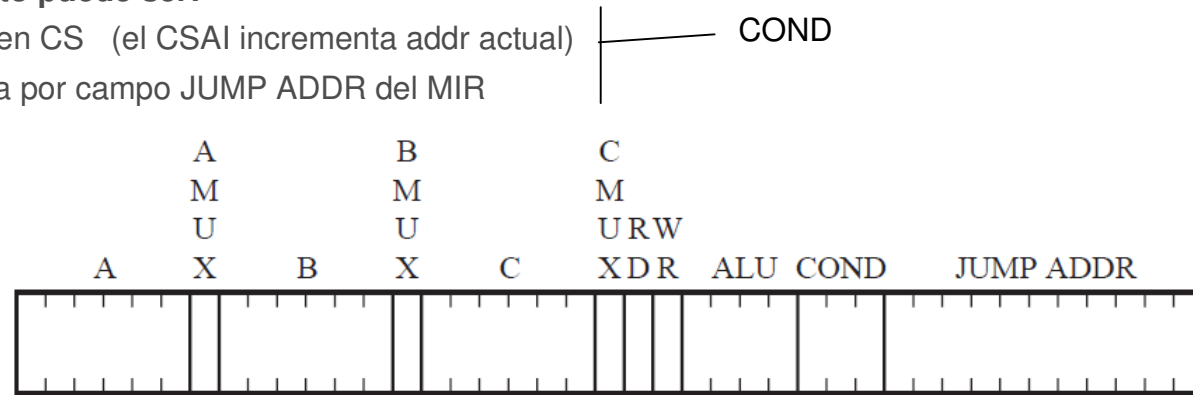


**Usar cualquier operación  
que no altere códigos de condición**

# 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



Define el salto condicional

Dirección del salto (11 bits)

COND			Operation
C <sub>2</sub>	C <sub>1</sub>	C <sub>0</sub>	
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

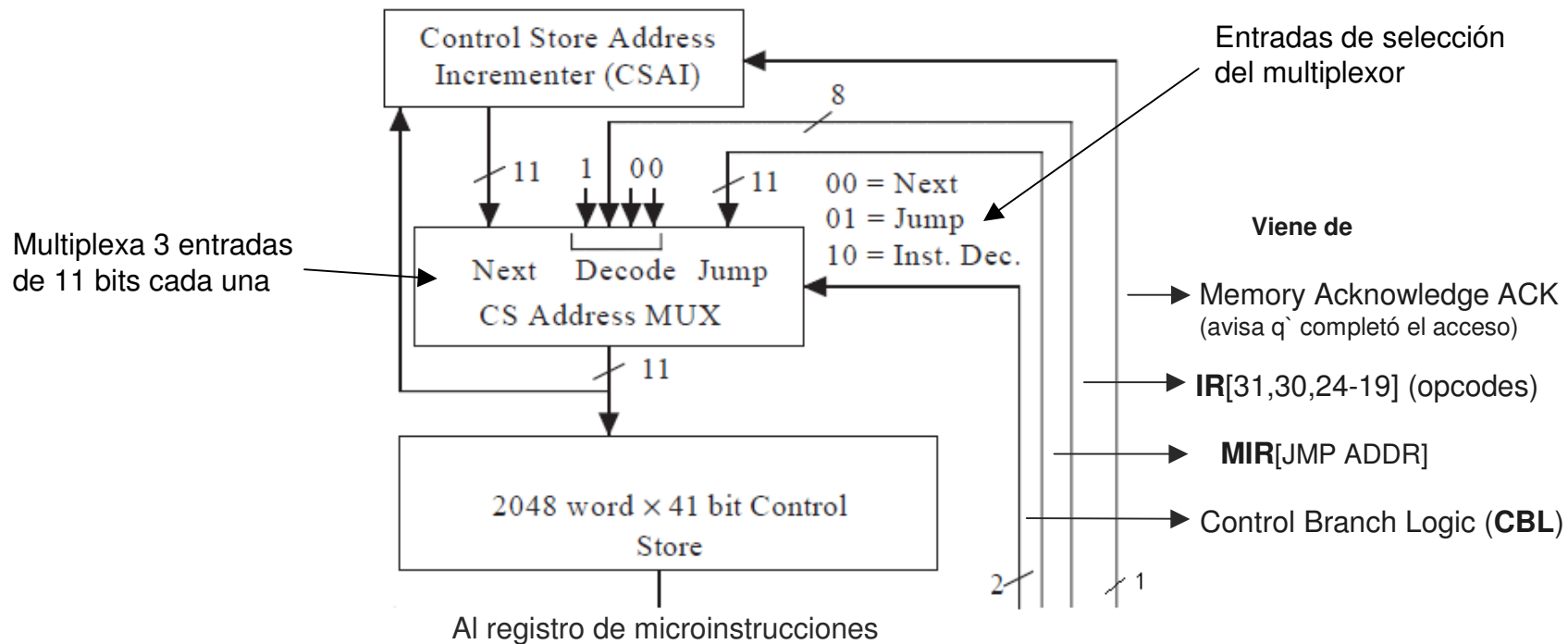
siguiente

Salta si el el flag está en 1

Salta si el direccionamiento es "inmediato"

salto incondicional

# Dirección de la próxima instrucción



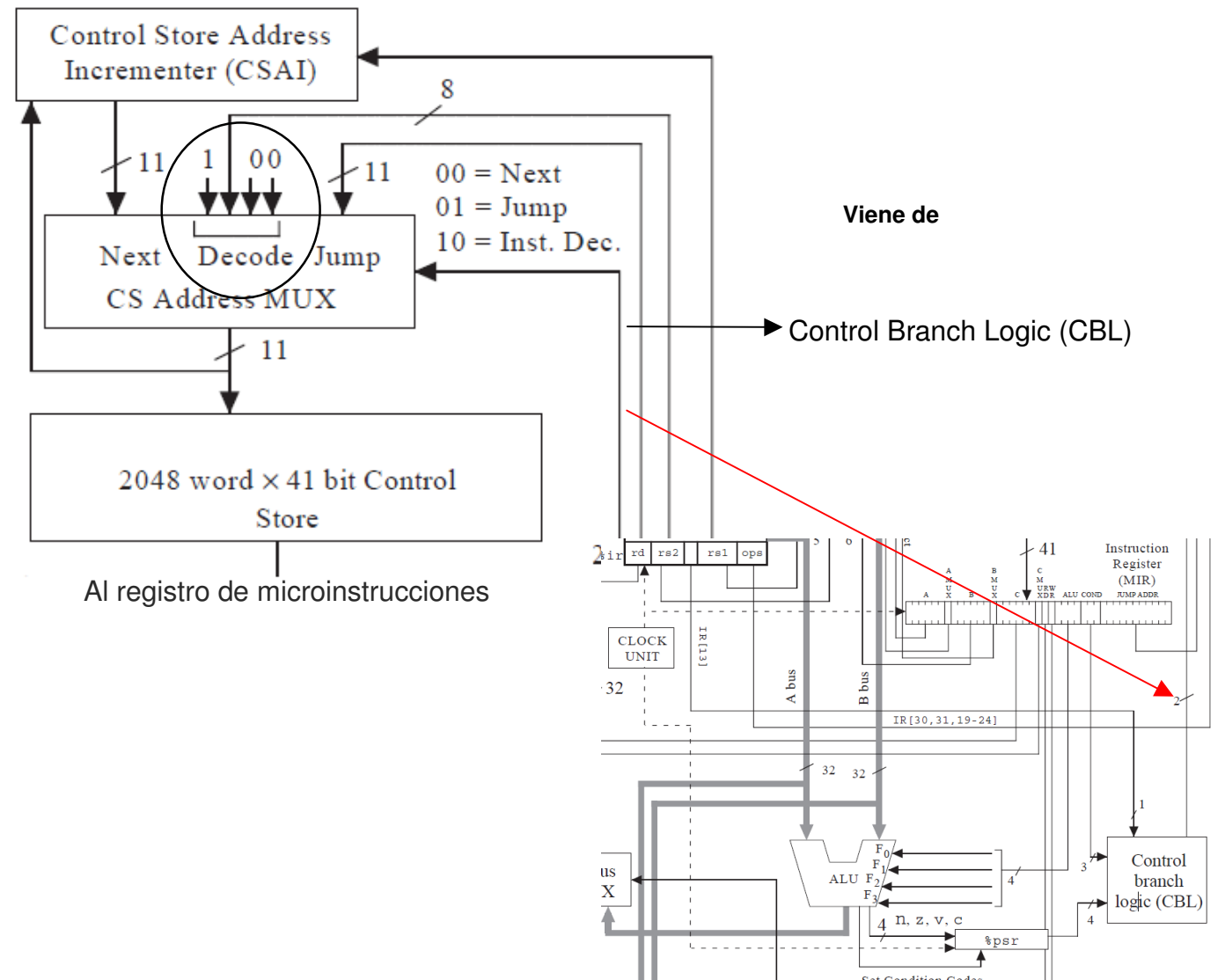
$C_2$	$C_1$	$C_0$	Operation
0	0	0	Use NEXT ADDR
0	0	1	Use JUMP ADDR if n = 1
0	1	0	Use JUMP ADDR if z = 1
0	1	1	Use JUMP ADDR if v = 1
1	0	0	Use JUMP ADDR if c = 1
1	0	1	Use JUMP ADDR if IR[13] = 1
1	1	0	Use JUMP ADDR
1	1	1	DECODE

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*

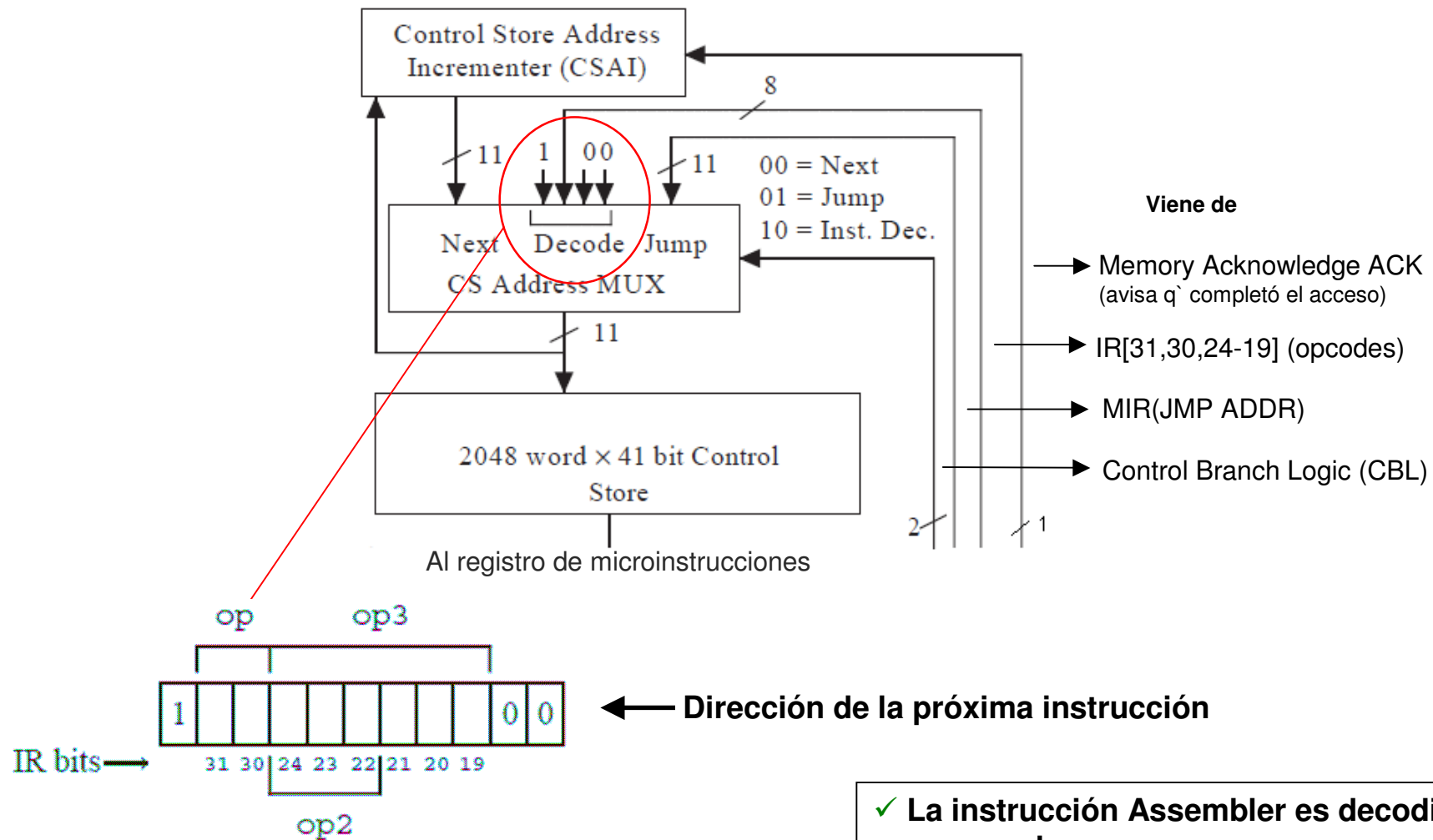
⇒ "Decodifica" instrucción Assembler ⇒ la dirección del salto es la del inicio de la correspondiente rutina del microprograma

# Decodificar una instrucción





# Decodificar una instrucción



✓ La instrucción Assembler es decodificada en un solo paso

# Microprogramación

- Programa de propósito general
  - Se implementa en Assembler del procesador (p.e., ARC)
  - Visible al programador
  - Implementa programas de propósito general
- Microprogramación
  - 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

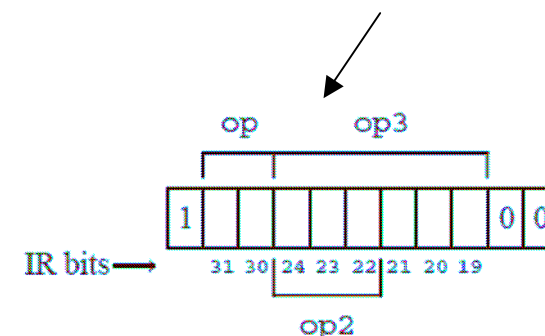
---

**/ Decodificando el assembler ARC**

0:  $R[ir] = \text{AND}(R[pc], R[pc]); \text{READ};$   
1:  $\text{DECODE};$

/ Lee una instrucción desde memoria principal

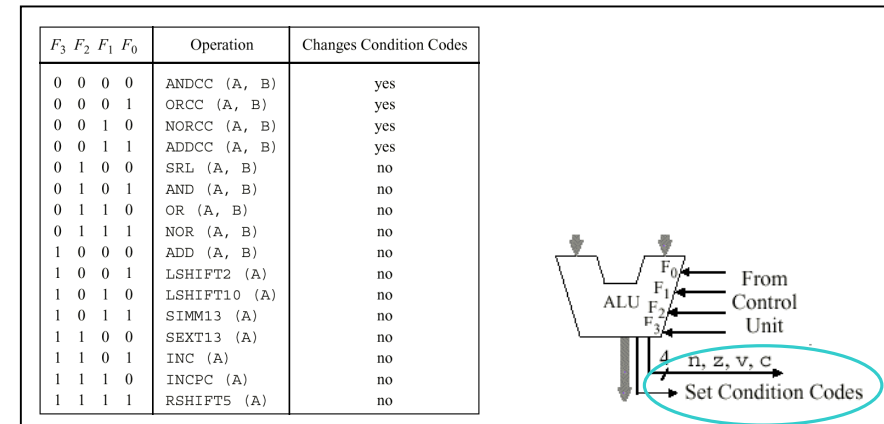
/ Salto a rutina de instrucción ARC según *opcode*



(Se almacena en control Store y en el MIR)

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

/ Lee una instrucción desde memoria principal

/ Salto a rutina de instrucción ARC según *opcode*

# Microprograma en binario

(Se almacena en control Store y en el MIR)

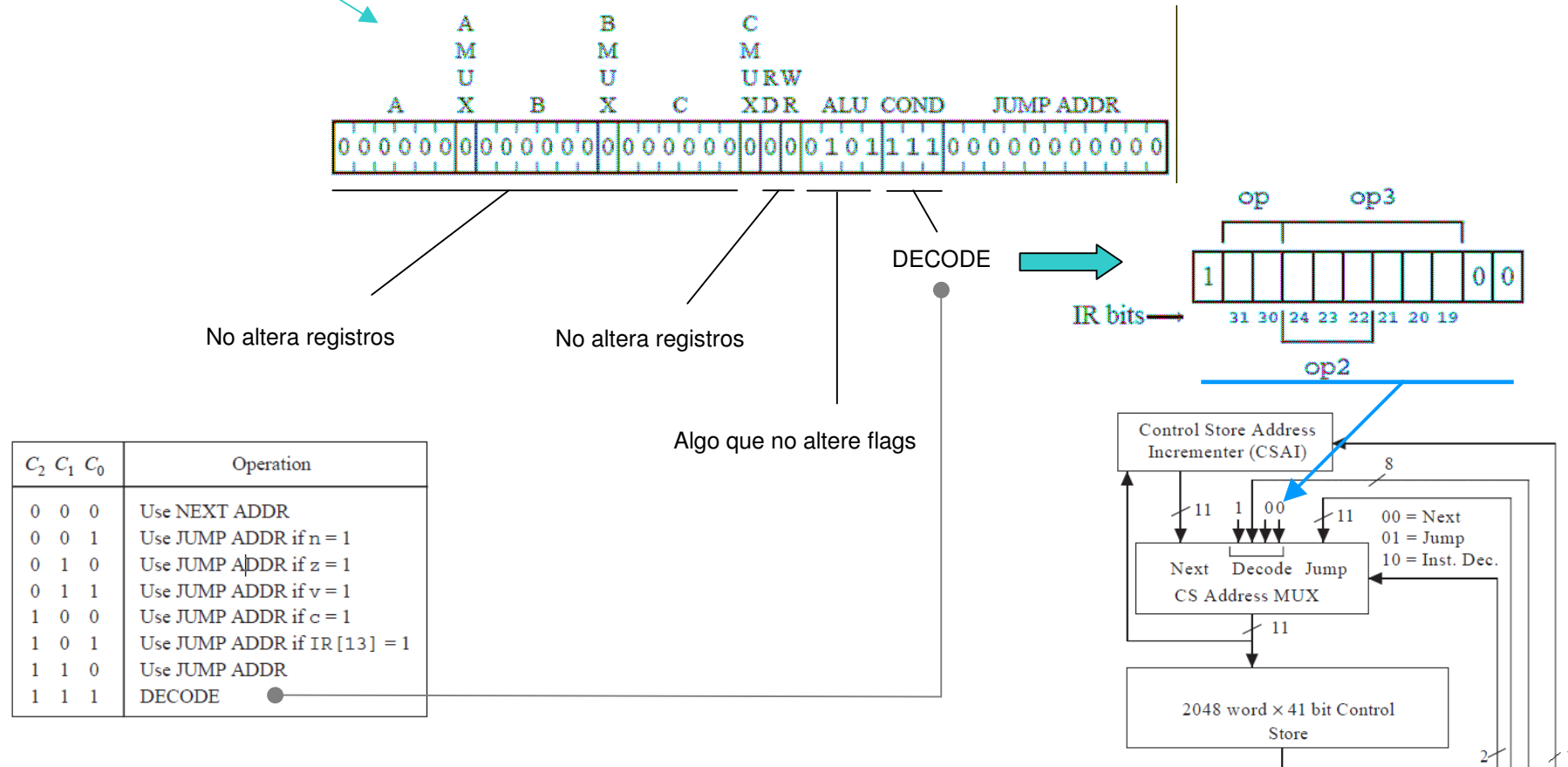
## / Decodificando el Assembler ARC

0:  $R[ir] = \text{AND}(R[pc], R[pc]); \text{READ};$

/ Lee una instrucción desde memoria principal

1: DECODE;

/ Salto a rutina de instrucción ARC según *opcode*



# De la instrucción Assembler a su microcódigo

## Addcc

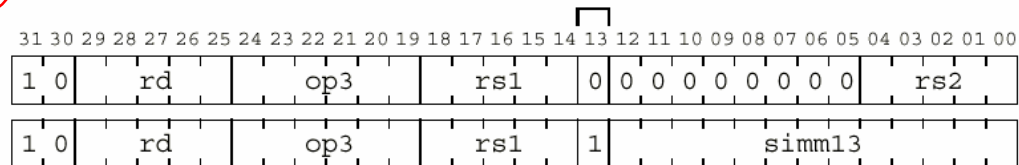
```

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
    
```

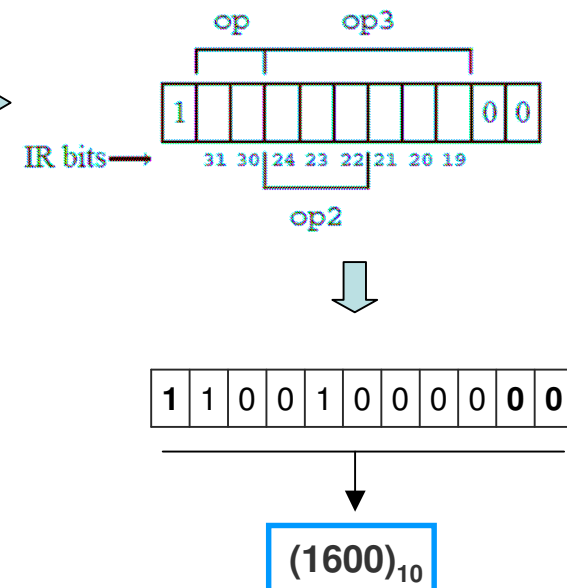
2  
¿binario?

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



### Formato Instrucc. aritméticas/lógicas

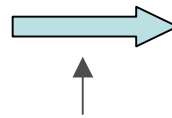
op=10	op3		
010000	addcc	1er. registro origen = rs1	
010001	andcc	2do reg. origen = rs2	si i=0
010010	orcc	2do reg. origen = constante <i>simm13</i>	si i=1
010110	orncc	Reg. destino = rd	
100110	srl		
111000	jmp1		



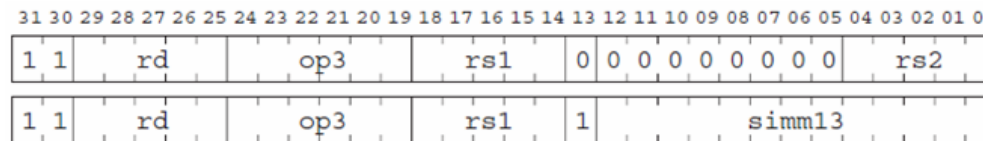
# De la instrucción Assembler a su microcódigo

## Lectura de memoria principal

ld %r5 + 80, %r2



11 00010 000000 00101 1 0000001010000  
op rd op3 rs1 i simm13



op=11 op3

000000 ld rd=reg.destino  
000100 st rd=reg.origen

Direcc. de memoria

rs1 + rs2 si i = 0  
rs1 + simm13 (constante) si i = 1

### / ld

```

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 direccion y salta
      GOTO 1793;

⋮

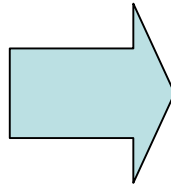
2047: R[pc] = INCPC(R[pc]); GOTO 0;     / Incrementa %pc y recomienza
    
```

# 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;
   / ld
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;

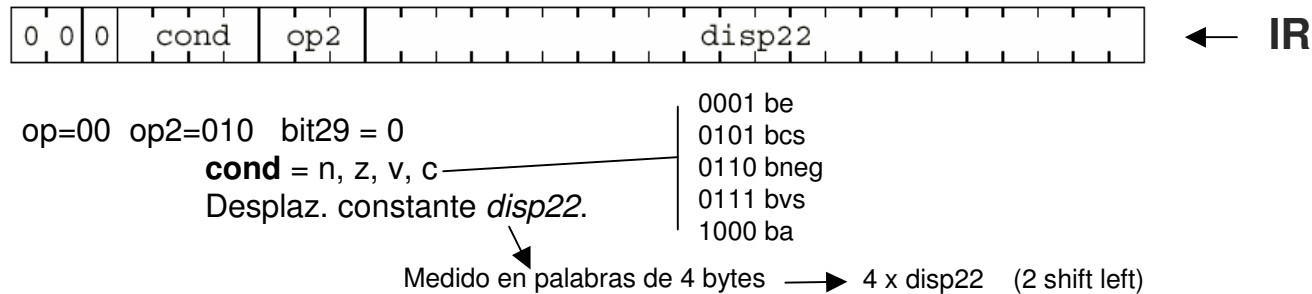
2047: R[pc] ← INCPC(R[pc]); GOTO 0;
```



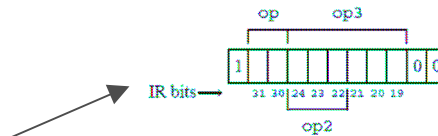
# De la instrucción Assembler a su microcódigo

## Salto condicional

Assembler ARC



Microprograma



La operación DECODE no toma en cuenta **cond** => **debo extraer cond**  
Para conocer la dirección de destino => debo extraer disp22

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

# De la instrucción Assembler a su microcódigo

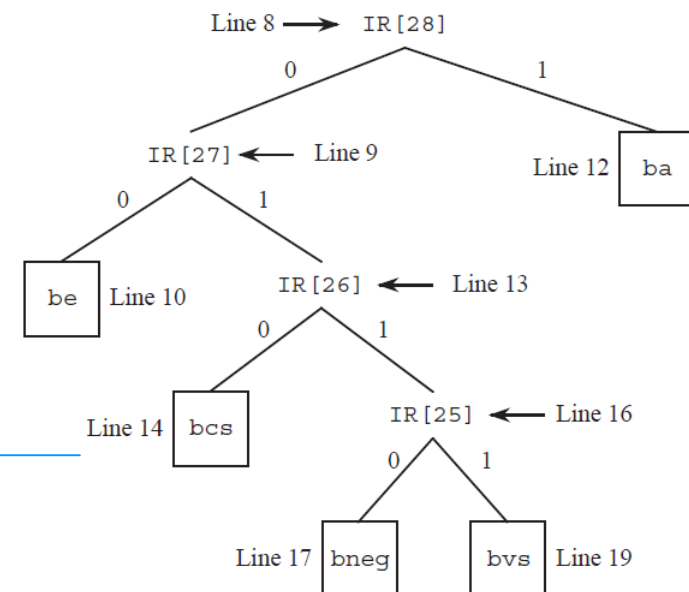
## Saltos condicionales

- 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
- 15 bits a der
- Desplaz a izq de a 1 bit por vez
- %temp0= disp22
- Lee COND bit a bit
- 2047: R[pc] = INCPC(R[pc]); GOTO 0; / Increment %pc and start over

# De la instrucción Assembler a su microcódigo

## Salto 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



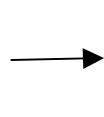
cond				branch
28	27	26	25	
0	0	0	1	be
0	1	0	1	bcs
0	1	1	0	bneg
0	1	1	1	bvs
1	0	0	0	ba

- 2047: R[pc] = INCPC(R[pc]); GOTO 0; / Increment %pc and start over

# De la instrucción Assembler a su microcódigo

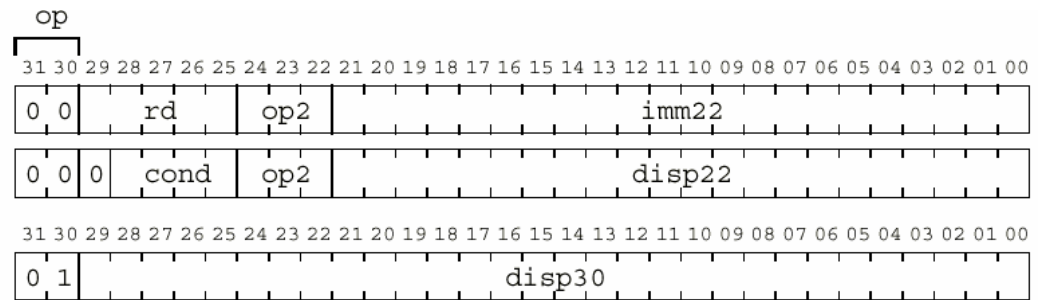
## Entradas duplicadas al Control Store

No usan OP3

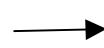


SETHI Format

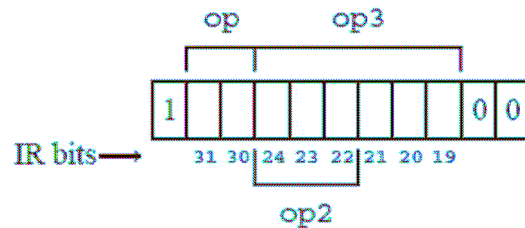
Branch Format



No usan OP2 ni OP3



CALL format



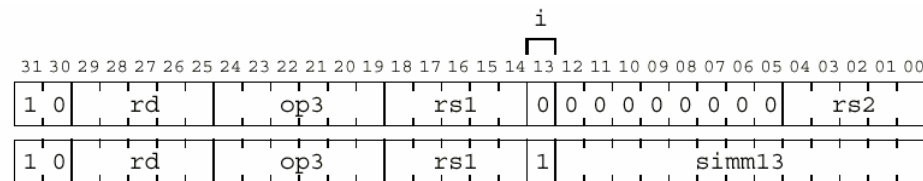
# De la instrucción Assembler a su microcódigo

Agregar instrucciones nuevas al ISA

- 1) Dirección de inicio en el CS
- 2) Microcódigo (“microassembler”)
- 3) Binario en el Control Store

## Ejemplo

- Instrucción *subcc*,
- Sigue el formato ARC aritmético con *op3=001100*



ARITMÉTICA op=10

op3  
 010000 addcc  
 010001 andcc  
 010010 orcc  
 010110 orncc  
 100110 srl  
 111000 jmp

1er. registro origen = rs1

2do reg. origen = rs2

2do reg. origen = constante *simm13*

Reg. destino = rd

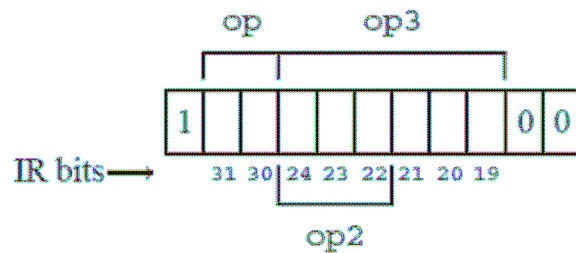
si i=0

si i=1

# De la instrucción Assembler a su microcódigo

Agregar instrucciones nuevas al ISA (*subcc*)

✓ Dirección de inicio en CS



*op3=001100*

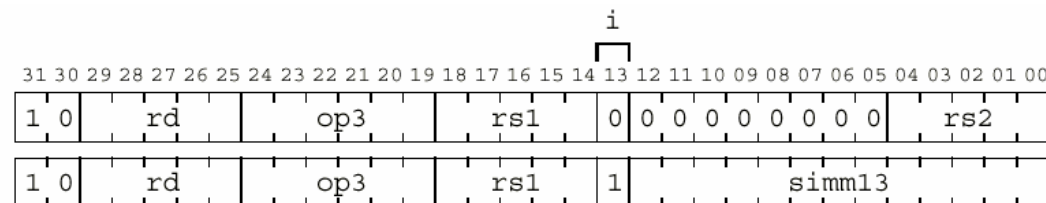
*Empieza en:*  
11000110000  
(1584)<sub>10</sub>

# De la instrucción Assembler a su microcódigo

Agregar instrucciones nuevas al ISA (*subcc*)

## ✓ Microcódigo

1584: R[temp0] = SEXT13(R[ir]); / Lee el substraendo en inmediato  
IF IR[13] THEN GOTO 1586; / Está el substraendo en modo inmediato?  
1585: R[temp0] = ADD(R[0], R[rs2]); / Lee el substraendo de registro rs2  
1586: R[temp0] = NOR(R[temp0], R[0]); / Calcula el complemento a 1  
1587: R[temp0] = INC(R[temp0]); GOTO 1603; / Calcula el complemento a 2  
  
1603: R[rd] = ADDCC(R[rs1], R[temp0]);  
GOTO 2047;



# De la instrucción Assembler a su microcódigo

Agregar instrucciones nuevas al ISA (*subcc*)

## ✓ Microcódigo

1584: R[temp0] = SEXT13(R[ir]);

IF IR[13] THEN GOTO 1586;

1585: R[temp0] = ADD(R[0], R[rs2]);

1586: R[temp0] = NOR(R[temp0], R[0]);

1587: R[temp0] = INC(R[temp0]); GOTO 1603;

1603: R[rd] = ADDCC(R[rs1], R[temp0]);

GOTO 2047;

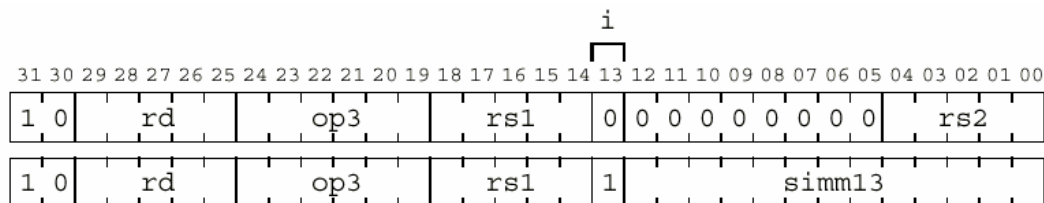
/ Lee el substraendo en inmediato

/ Está el substraendo en modo inmediato?

/ Lee el substraendo de registro rs2

/ Calcula el complemento a 1

/ Calcula el complemento a 2



También podría ser:

OR(R[0], R[rs2]);

OR(R[rs2], R[0])

SRL(R[rs2], R[0];

Expresables como::

R[temp0] <= R[rs2];



# De la instrucción Assembler a su microcódigo

Agregar instrucciones nuevas al ISA (*subcc*)

## ✓ Binario en el Control Store

1584: R[temp0] = SEXT13(R[ir]);  
 IF IR[13] THEN GOTO 1586;  
 1585: R[temp0] = ADD(R[0], R[rs2]);  
 1586: R[temp0] = NOR(R[temp0], R[0]);  
 1587: R[temp0] = INC(R[temp0]); GOTO 1603;  
  
 1603: R[rd] = ADDCC(R[rs1], R[temp0]);  
 GOTO 2047;

2

C <sub>2</sub>	C <sub>1</sub>	C <sub>0</sub>	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

3

F <sub>3</sub>	F <sub>2</sub>	F <sub>1</sub>	F <sub>0</sub>	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

1

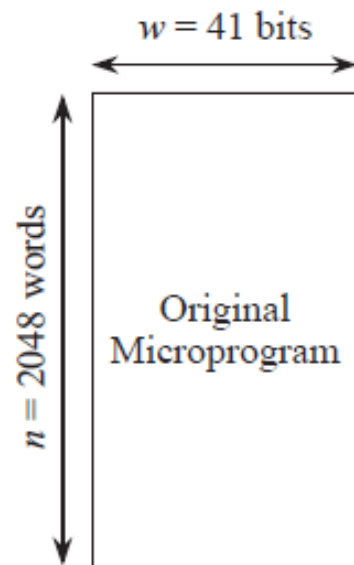
2

3

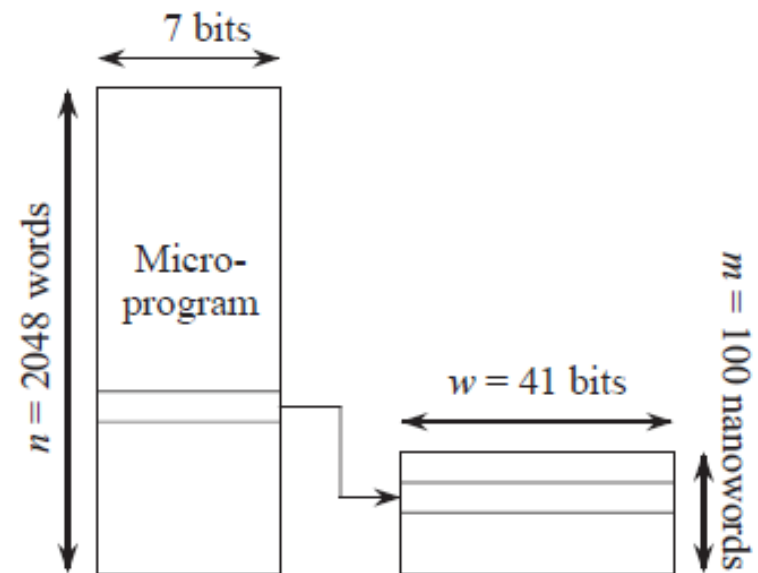
	A	M	X	B	M	X	C	M	URW	XDR	ALU	COND	JUMP	ADDR
1584	1	0	0	1	0	1	0	0	0	0	0	0	0	0
1585	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1586	1	0	0	0	0	1	0	0	0	0	0	0	0	0
1587	1	0	0	0	0	1	0	0	0	0	0	0	0	0

# Nanoprogramación

- Sin nanoprogramación -



- Con nanoprogramación -



ESPACIO OCUPADO  
EN EL CHIP

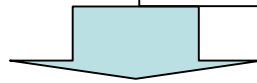
Vs. VELOCIDAD

# Condiciones de excepción

Un programa está ejecutándose según lo esperado pero...

- Una instrucción que no pertenece al ISA
- Hardware no instalado
- Operación de punto flotante: *overflow* o *underflow*
- *División por cero*

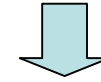
- Se excedió la  $^{\circ}T$  máxima de operación
- Fallos de alimentación
- Una tecla fue pulsada
- Un periférico lento terminó la acción pedida



Debe existir una procedimiento que permita al sistema sobrevivir a estas condiciones de excepción



"Trampas"



"Interrupciones"

# Trampas e interrupciones

## Trampas

- El microcódigo debería detectar la condición y saltar al *trap handler*
- Son sincrónicas

## Interrupciones

- El dispositivo solicita interrupción, el CPU contesta, el disp. se identifica y la CPU pasa el control al handler de interrupción correspondiente
- Son asincrónicas

## Vectorización

- Una tabla de rutinas de interrupción y trampas guarda sólo la primer instrucción de los handler; esta es un salto al procedimiento
- El procedimiento concreto puede ser cambiado a nivel usuario
- El cpu guarda en stack el PC y el %psr
- El handler es responsable de devolver todos los registros inalterados cuando termina

Address	Contents	Trap Handler
	:	
60	JUMP TO 2000	Illegal instruction
64	JUMP TO 3000	Overflow
68	JUMP TO 3600	Underflow
72	JUMP TO 5224	Zerodivide
76	JUMP TO 4180	Disk
80	JUMP TO 5364	Printer
84	JUMP TO 5908	TTY
88	JUMP TO 6048	Timer
	:	

---

# ***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***

