

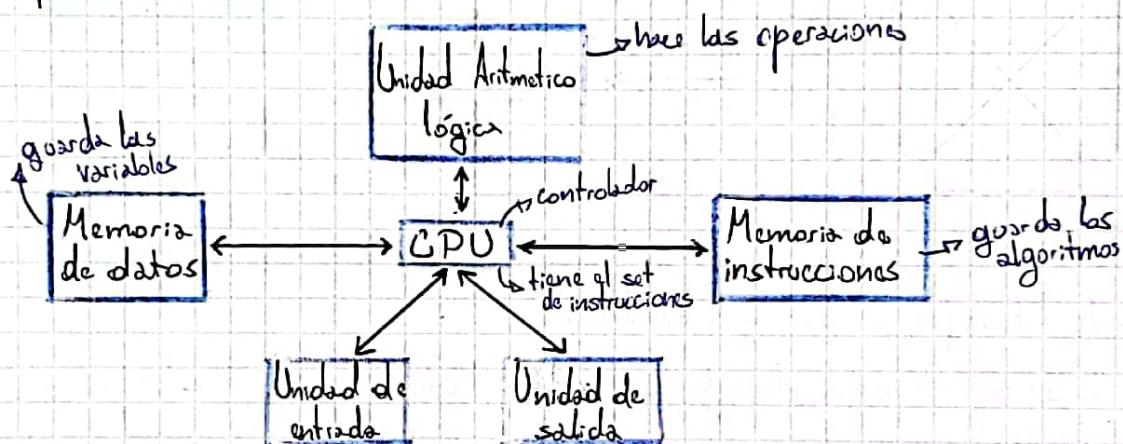
# MICROARQUITECTURA

## Arquitectura de una computadora

¿Qué hace una computadora? Ejecutar operaciones simples en una secuencia ordenada.

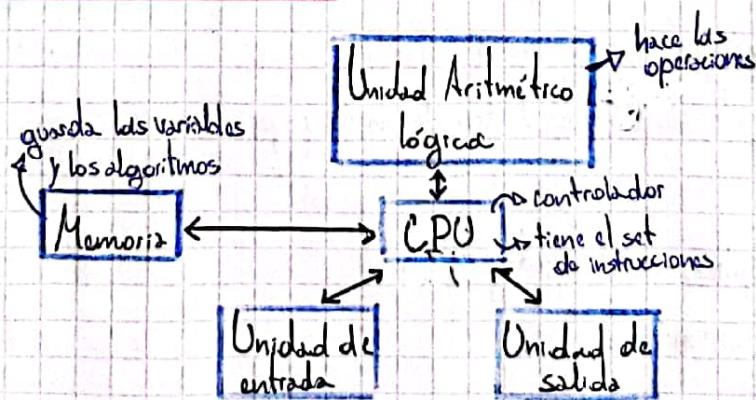
Modelos de arquitectura → Arquitectura Harvard (1940)  
→ Arquitectura Von Neumann (1945)

### Arquitectura Harvard



- Arquitectura de las primeras computadoras.
- Se usa actualmente en sistemas simples, como los electrodomésticos. Por ej., lavadoras (memoria de instrucciones fija y siempre ejecuta el mismo programa).

### Arquitectura Von Neumann



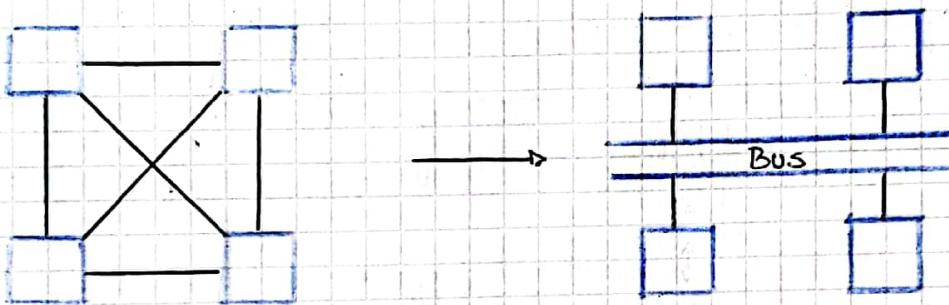
- Unifica el almacenamiento físico de datos y programas en un solo lugar.
- Trata a los programas parecido a las variables
- Puedo leer, escribir y modificar los programas.
- Se usa actualmente para compiladores y sistemas operativos

## Diferencias Arquitectura Harvard y Arquitectura Von Neumann

- Con Von Neumann se pueden modificar programas con la misma felicidad que modificar datos
  - ⇒ Fácil reprogramación, versatilidad característica de las computadoras actuales
- La Arquitectura Harvard tiene pequeño volumen de datos y de programas
  - ⇒ ideal para dispositivos con función única (no reprogramables).

## Conexión con estructuras tipo BUS

- Reduce el nº de rutas necesarias para la comunicación entre los distintos componentes, realizando las comunicaciones a través de un solo canal de datos



Línea → de datos: por donde viajan los datos

→ de direcciones: por donde viaja la dirección a la cual se quiere acceder

→ de control: qué operación se quiere hacer sobre el destino (leer, escribir)

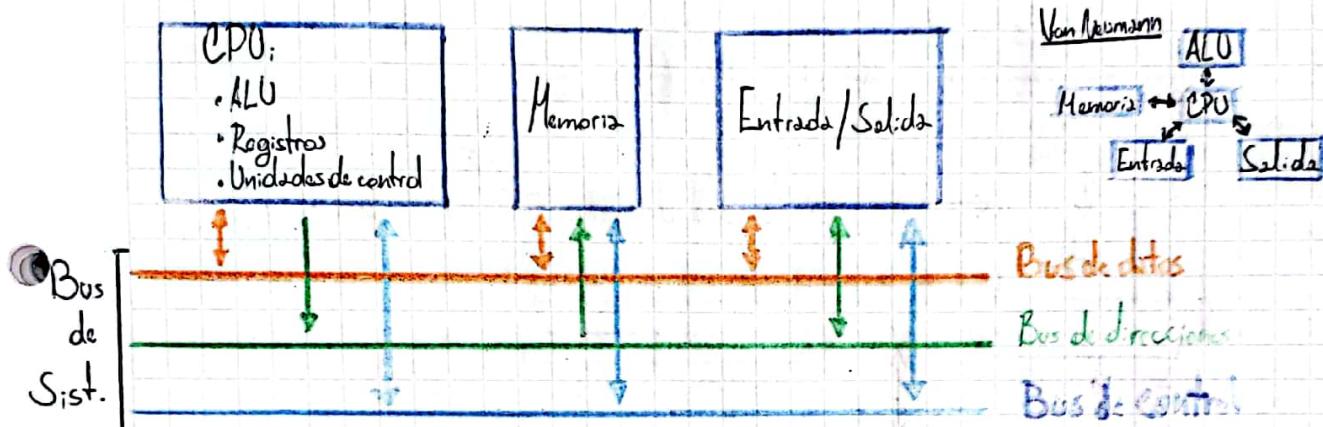
## Bus

- Sist. de comunicación para la transferencia de datos entre componentes electrónicos.
- Permite conectar → componentes de una computadora
  - Varias Computadoras entre si
  - Componentes electrónicos dentro de un microprocesador
- Componentes:
  - Hardware: cables, fibra óptica, circuitos integrados,
  - Software: protocolos de comunicación.

NOTA

## Modelo Bus de Sistema

- Arquitectura Von Neumann modificada: con un bus de sistema.
- Propósito: reducir el número de interconexiones entre la CPU y sus subsistemas.



Cada bus: conjunto de muchos cables. En cada cable va un bit.

## Arquitectura del CPU - Arquitectura del microprocesador - Microarquitectura

Microprocesador: dispositivo donde hoy en día se encuentra la CPU y la ALU.

↳ mejora tecnológica que permite ocupar menos espacio.

Cada implementación de un microprocesador hace énfasis en algo diferente. Por ej:

- mayor velocidad
- menor costo
- menor consumo

## Funciones del CPU - Ciclo de Fetch → Ejecutar un programa

- I. Buscar en memoria la próx. instrucción a ser ejecutada.
- II. Decodificar el ecd. de operación de esa instrucción.
- III. Ejecutar la instrucción.
- IV. Volver a I.

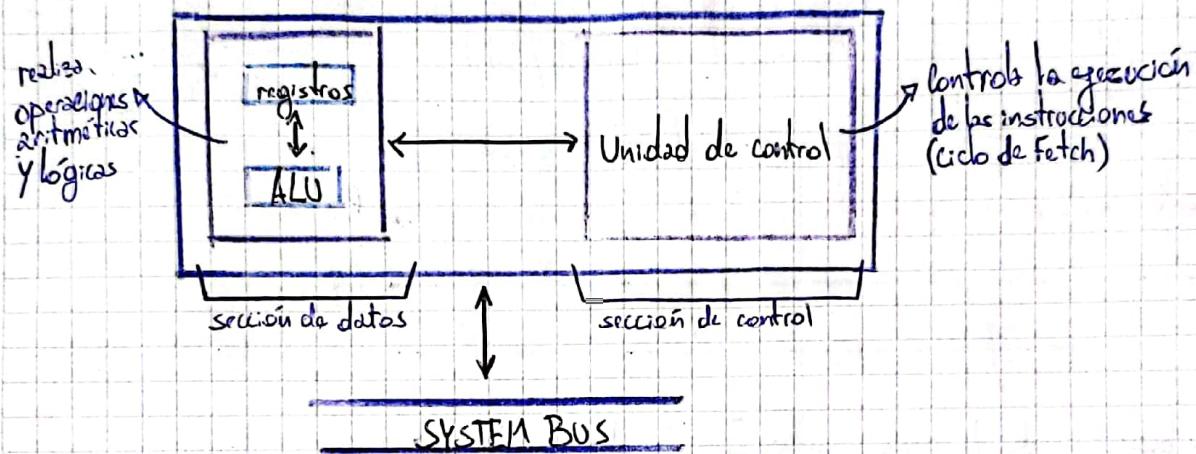
\* En Múndas agrega algo como "operar en memoria" → En ARC no es válido ya que solo se puede operar con registros

NOTA

## Microarquitectura

- Contiene
  - Unidad de control
  - ALU
  - registros accesibles por el programador
  - registros adicionales necesarios para el funcionamiento de la unidad de control

- Conformada por:
  - sección de control
  - sección de datos
  - camino de datos



## Organización Trajeto de Datos

### Conexión

#### \* Interno:

- En 1 bus
- En 2 buses
- En 3 buses

#### \* Externo:

- Bus de sistema

### Organizado en 1 bus

- Lee 1<sup>er</sup> operando y lo guarda en A
  - Lee 2<sup>do</sup> operando y lo guarda en B
  - Hace la operación (ALU)
  - Lee resultado y lo guarda en un registro
- tengo que mandar una entrada a la vez

### Organizado en 2 buses

- Lee 1<sup>er</sup> y 2<sup>do</sup> operando
- Ejecuto y guardo en registro

### Organizado en 3 buses

- tengo un bus para cada operando y uno para la salida

→ tengo un registro auxiliar para guardar el resultado

NOTA

## Unidad Aritmético-Lógica (ALU)

→ circuito digital combinacional

- Mueve datos (desplazamientos a derecha e izquierda)

- Hace operaciones aritméticas/lógicas

- Implementación: 32 módulos de 1 bit (ALU de 32 bits/4 Bytes) → puedo implementar operaciones en 32 bits

↳ trataría de 2 1-bit.

- para las operaciones aritméticas/lógicas

  - Circuitos sumadores, restadores.

  - tabla de consulta o "look up table" (LUT)

- para los desplazamientos

  - registros de desplazamiento

  - desplazador rápido o "barrel-shifters"

## Tabla de consulta o "look up table" (LUT)

- Realiza:

  - operaciones aritméticas

  - operaciones lógicas

- Entradas:

  - 2 operandos de 32 bits = 4 Bytes

  - 4 bits para seleccionar la operación

- Salidas:

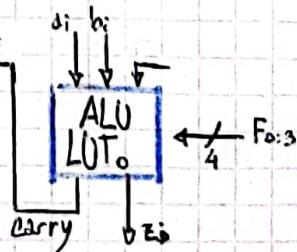
  - resultado de 32 bits = 4 Bytes

  - flags: n, z, v, c

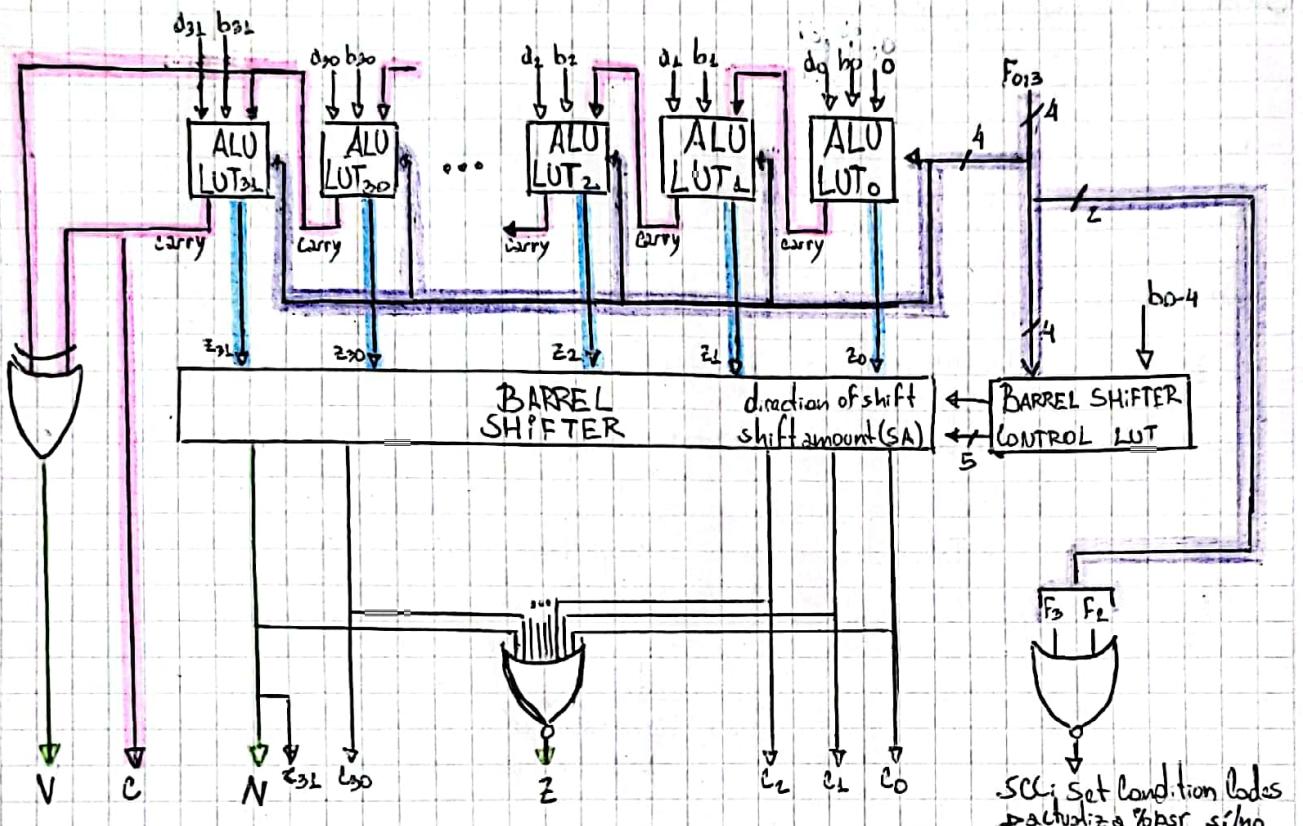
  - condición (si/no); 1 bit

Tabla de verdad parcial de  
una LUT de 1 bit

Entradas							
F <sub>3</sub>	F <sub>2</sub>	F <sub>1</sub>	F <sub>0</sub>	Carry in	a <sub>i</sub>	b <sub>i</sub>	z <sub>i</sub>
0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0
0	0	0	0	1	0	0	0
0	0	0	0	1	0	1	1
0	0	0	1	0	0	0	0
0	0	0	1	0	0	1	1
0	0	0	1	1	0	0	0
0	0	0	1	1	0	1	1
0	0	1	0	0	0	0	0
0	0	1	0	0	0	1	1
0	0	1	0	1	0	0	0
0	0	1	0	1	0	1	1
0	0	1	1	0	0	0	0
0	0	1	1	0	0	1	1
0	0	1	1	1	0	0	0
0	0	1	1	1	0	1	1
0	1	0	0	0	0	0	0
0	1	0	0	0	0	1	1
0	1	0	0	1	0	0	0
0	1	0	0	1	0	1	1
0	1	0	1	0	0	0	0
0	1	0	1	0	0	1	1
0	1	0	1	1	0	0	0
0	1	0	1	1	0	1	1
0	1	1	0	0	0	0	0
0	1	1	0	0	0	1	1
0	1	1	0	1	0	0	0
0	1	1	0	1	0	1	1
0	1	1	1	0	0	0	0
0	1	1	1	0	0	1	1
0	1	1	1	1	0	0	0
0	1	1	1	1	0	1	1
1	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1
1	0	0	0	1	0	0	0
1	0	0	0	1	0	1	1
1	0	0	1	0	0	0	0
1	0	0	1	0	0	1	1
1	0	0	1	1	0	0	0
1	0	0	1	1	0	1	1
1	0	1	0	0	0	0	0
1	0	1	0	0	0	1	1
1	0	1	0	1	0	0	0
1	0	1	0	1	0	1	1
1	0	1	1	0	0	0	0
1	0	1	1	0	0	1	1
1	0	1	1	1	0	0	0
1	0	1	1	1	0	1	1
1	1	0	0	0	0	0	0
1	1	0	0	0	0	1	1
1	1	0	0	1	0	0	0
1	1	0	0	1	0	1	1
1	1	0	1	0	0	0	0
1	1	0	1	0	0	1	1
1	1	0	1	1	0	0	0
1	1	0	1	1	0	1	1
1	1	1	0	0	0	0	0
1	1	1	0	0	0	1	1
1	1	1	0	1	0	0	0
1	1	1	0	1	0	1	1
1	1	1	1	0	0	0	0
1	1	1	1	0	0	1	1
1	1	1	1	1	0	0	0
1	1	1	1	1	0	1	1



ALU implementada con  
"lookup table" y "barrel-shifter"



SCL: Set Condition Codes  
 actualiza %psr si/no

$V \rightarrow$  XOR de los últimos dos carries

$C \rightarrow$  Último carry

$N \rightarrow$  dígito más significativo ( $C_31$ ) del resultado

$Z \rightarrow$  NOR todos los bits del resultado

$F_3$	$F_2$	$F_1$	$F_0$	Operation	Parameters	Change condition codes
0	0	0	0	ANDCC(A,B)	A,B	yes
0	0	0	1	ORCC(A,B)	A,B	yes
0	0	1	0	ORNCC(A,B)	A,B	yes
0	0	1	1	ADDCC(A,B)	A,B	yes
0	1	0	0	SRL(A,B)	A,B	no
0	1	0	1	AND(A,B)	A,B	no
0	1	1	0	OR(A,B)	A,B	no
0	1	1	1	ORN(A,B)	A,B	no
1	0	0	0	ADD(A,B)	A,B	no
1	0	0	1	LSHIFT2(A)	A	no
1	0	1	0	LSHIFT10(A)	A	no
1	0	1	1	SIMM13(A)	A	no
1	1	0	0	SEXT13(A)	A	no
1	1	0	1	INC(A)	A	no
1	1	1	0	INCP(A)	A	no
1	1	1	1	RSHIFT5(A)	A	no

## Desplazador rápido o "barrel-shifter"

### • Realiza desplazamientos

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

### • Entradas

- operando: 32 bits
- cantidad de bits a desplazar: 5 bits (de 0 a 31 bits)
- dirección: 1 bit

### • Salida: resultado: 32 bits.

• Implementación: multiplexores donde cada uno tiene una entrada de control 0 o 1.

↳ si entrada control = 0  $\Rightarrow$  entra la "entrada directa".

si entrada control = 1  $\Rightarrow$  entra la entrada corrida  $i$  posiciones con  $i=1, 2, 4$ .

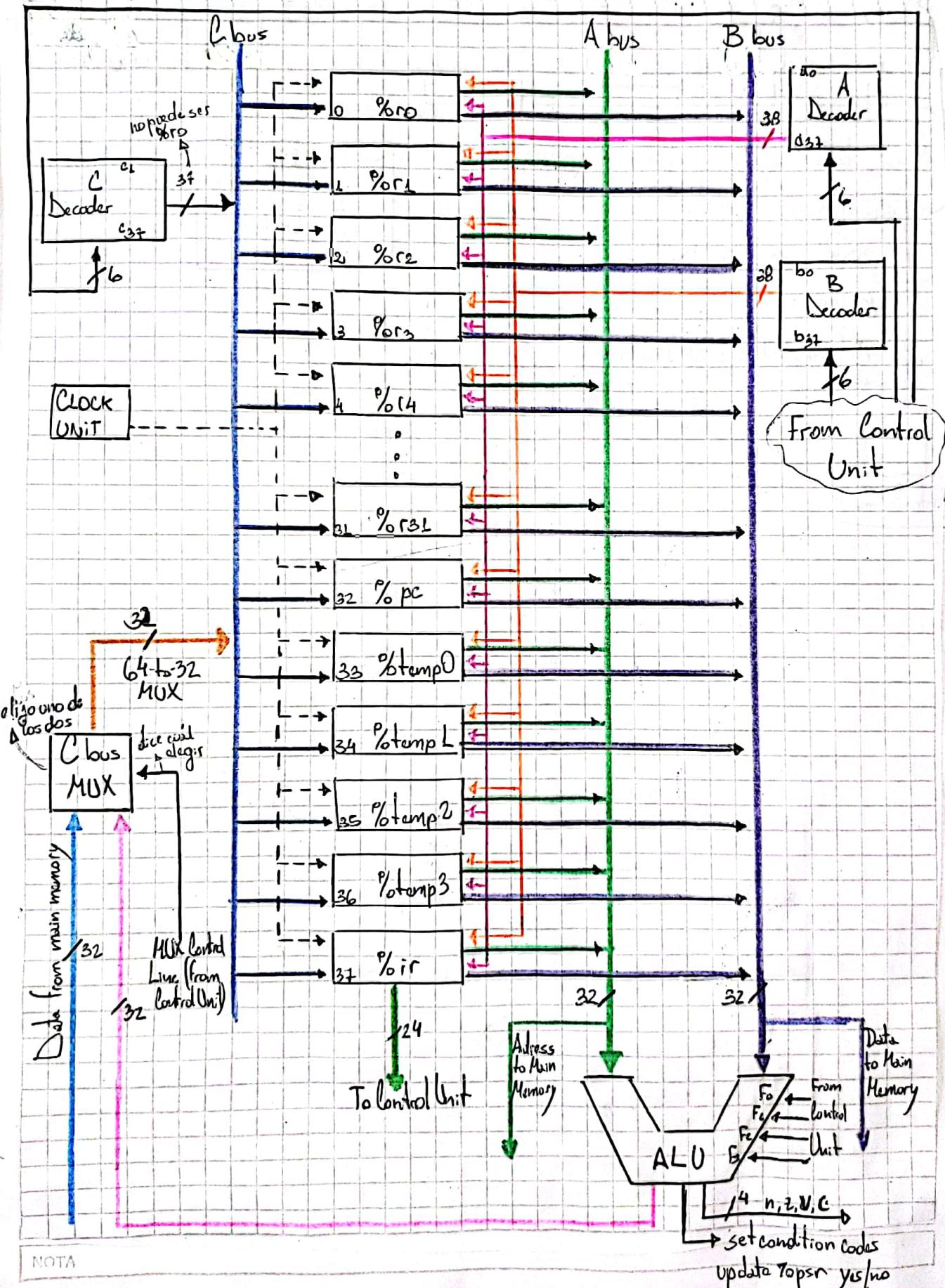
↳ 1 columna izquierda

2 columna del medio

4 columnas derecha

A bus: direcciones/lectura  
 B decoder: n° reg.  
 B bus: datos/lectura  
 C bus: datos/escritura

### Traecto de datos



- decodificadores: eligen el registro que tiene habilitada la comunicación con el bus en ese momento
- buses:
  - Bus A: tiene una conexión que copia todo lo que tiene A y lo manda al bus de direcciones de la memoria
  - Bus B: tiene una conexión que copia todo lo que tiene B y lo manda al bus de datos de la memoria
  - C Bus MUX elige si la entrada de datos al registro por el bus C viene de la memoria o de una operación de la ALU
- Clock: recién con la llegada de un flanco, se actualiza el registro que determina el C decodificador.

### Estructura de los registros

- Utilizan el tristate cuando el decodificador no las selecciona, para así no leer ni escribir a ningún bus
- Registros especiales
  - %ro: no guarda ni tiene entrada  $\Rightarrow$  no necesita Flip-Flops
  - %pc: solo almacena múltiplos de 4  $\Rightarrow$  los 2 LSB cableados a 0.
  - %ir: tiene salidas específicas por campos del cod. de máquina obtenido desde memoria

### Diseño de Unidad de Control

- Se puede diseñar por:

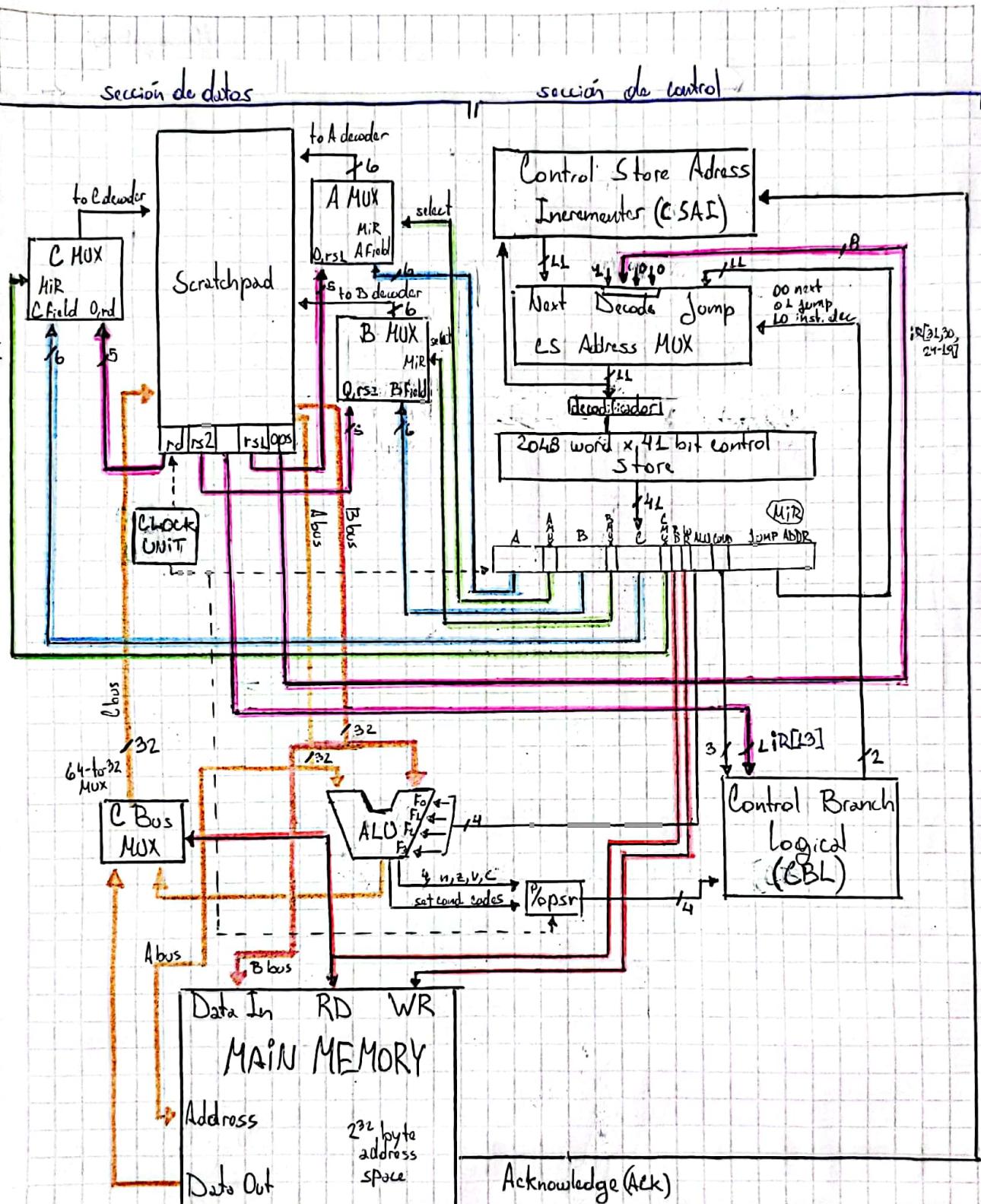
• Lógica microprogramada: circuito lógico sencillo que permite realizar la ejecución mediante instrucciones muy elementales llamadas microinstrucciones.  
• grabado en ROM

• Lógica cableada: lógica de las transiciones de estados como en los contadores sincrónicos

- El trayecto de datos es el mismo independientemente de la implementación que se utilice.

### Memoria microprogramada

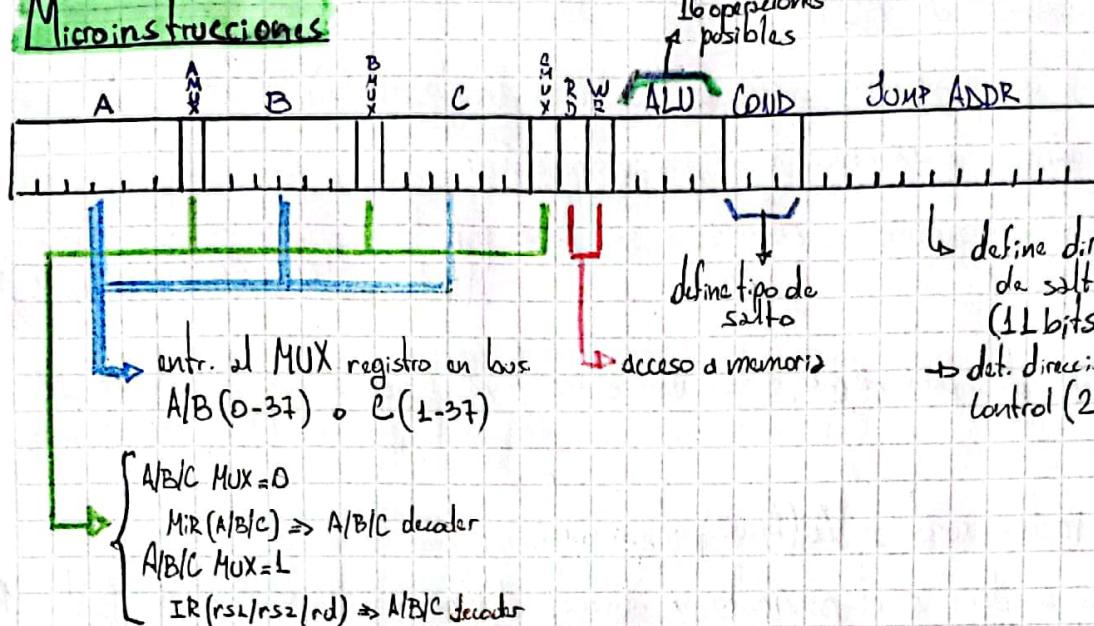
- El algoritmo que permite hacer el ciclo de fio está grabado en la ROM de 2048 palabras de 41 bits cada una.
- El decodificador elige cuál de las 2048 palabras se copia al MiR (microcode instruction register) de 41 bits



- AMUX/BMUX/CMUX eligen entrada en los multiplexores (registro de instrucción o registro de microinstrucciones)
- CSAI suma 1 a la dirección actual.
- ACK avisa que completó el acceso

NOTA

## Microinstrucciones



## RD - WR

RD	WR	Acción
0	0	no accede a memoria
0	1	escribe en memoria
L	0	lee de memoria
L	L	— (estado prohibido)

## COND

C <sub>2</sub>	C <sub>1</sub>	C <sub>0</sub>	Operación	
0	0	0	Use NEXT ADDR	→ avanza sig. dir.
0	0	1	Use JUMP ADDR if n=L	→
0	1	0	Use JUMP ADDR if z=L	→ salta dependiendo del flag
0	1	1	Use JUMP ADDR if v=L	→
1	0	0	Use JUMP ADDR if c=L	→
L	0	L	Use JUMP ADDR if IR[L3]=1	→ salta si dirección
L	L	0	Use JUMP ADDR	miembro es inmediato
L	L	L	DECODE	→ salto incondicional

## ALU

F <sub>2</sub>	F <sub>1</sub>	F <sub>0</sub>	Operation	Parameters
0	0	0	ANDCC	A B
0	0	L	ORCC	A B
0	0	0	ORNCC	A B
0	0	L	ADDCL	A B
0	L	0	SRL	A B
0	L	L	AND	A B
0	L	0	DR	A B
0	L	L	ORN	A B
L	0	0	ADD	A B
L	0	0	LSHIFT2	A
L	0	L	LSHIFT10	A
L	0	L	SHML3	A
L	L	0	SEXTL3	A
L	L	0	INC	A
L	L	L	INCPL	A
L	L	L	RSHIFT5	A

## REGISTROS

Número	Registro
00	%ro
01	%rl
02	%r2
03	%rs
04	%r4
05	%r5
...	
30	%r30
31	%r31
32	%pe
33	%temp0
34	%temp1
35	%temp2
36	%temp3
37	%ir

NOTA

## PASOS PARA ENTENDER LA MICROINSTRUCCIÓN

### 1. Miro el contenido de ALU

→ Eso me va a decir qué instrucción es. (la busco en la tabla).

- Me fijo cuántos parámetros tiene esta instrucción:

- Si: tiene un parámetro, éste va a ser A ⇒ para los siguientes pasos sólo miro el valor de A y AMUX (ignoro B y BMUX)

- Si: tiene dos parámetros, éstos van a ser A y B ⇒ para los siguientes pasos miro A, AMUX, B y BMUX.

### 2. Miro el multiplexor de A: AMUX

→ Eso me va a decir si el parámetro lo obtengo del registro de instrucciones (iR) o del registro de microinstrucciones (MiR)

- Si:  $AMUX = 1 \Rightarrow$  el valor está en el iR.

el parámetro es  $R[r_{s1}] \Rightarrow$  ignoro valor de A.

- Si:  $AMUX = 0 \Rightarrow$  el valor está en el MiR

- Miro el valor de A: ese va a ser el número del registro donde está el dato.

ojo: si el número de registro es mayor a  $3L$ , éste tiene un "número especial" (mirar tabla)

el parámetro es  $R[nombre\ del\ registro]$

↳ ejemplo:  $R[pc]$  o  $R[temp0]$ .

### 3. Si la instrucción tiene dos parámetros, miro el multiplexor de B: BMUX

→ Mismo que con AMUX

- Si:  $BMUX = 0 \Rightarrow$  el parámetro es  $R[r_{s2}] \Rightarrow$  ignoro valor de B.

- Si:  $BMUX = 1 \Rightarrow$  exactamente igual que en el paso anterior

### 4. Miro el multiplexor de C: CMUX

→ Eso me va a decir si el destino lo obtengo del iR o del MiR

Misma lógica que con AMUX

NOTA

- Si CMUX = 1  $\Rightarrow$  el registro de destino es R[rd]  $\Rightarrow$  ignoro valor de C.
- Si CMUX = 0  $\Rightarrow$  exactamente igual que con AMUX.

## 5. Miro el contenido de RD-WR

- Si es READ escribo READ;
- Si es WRITE escribo WRITE

## 6. Miro el contenido de COND

$\rightarrow$  Eso me va a decir la condición de salto (mira la tabla)

- Si es NEXT ADDR  $\Rightarrow$  no escribo nada más  $\Rightarrow$  ignoro valor de JUMP ADDR.
- Si es JUMP ADDR ;f flag = 1  $\Rightarrow$  escribo IF FLAG THEN  
    Por ejemplo IF Z THEN
- Si es JUMP ADDR ;f IR[L3] = 1  $\Rightarrow$  escribo IF R[IR[L3]] THEN
- Si es JUMP ADDR  $\Rightarrow$  no escribo condición
- Si es DECODE  $\Rightarrow$  escribo DECODE;  $\Rightarrow$  ignoro valor de JUMP ADDR.

## 7. Si el valor de COND es JUMP ADDR, JUMP ADDR ;f Flag = 1 ó

JUMP ADDR ;f IR[L3] = 1, miro el valor del campo JUMP ADDR

$\rightarrow$  Eso me dice la dirección de salto:

Escribo GOTO dirección;

Por ejemplo GOTO B96;

$\rightarrow$  Si no se cumple la condición o no hay condición, actúa como NEXT (incrementa en 1 la dirección actual)

### Sintaxis

registro\_destino = instrucción (parámetro1, parámetro2);

READ/WRITE;

si corresponde

if condición then  
GOTO dirección;

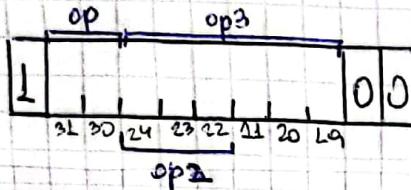
optional

NOTA

## Ejecución próxima microinstrucción a ejecutar

- CBL recibe los flags (%psr), la condición de salto (COND) y cómo es el direccionamiento (IR[23])  $\rightarrow$  su salida Funcion como entrada de control del multiplexor que elige cómo avanzar:

- NEXT: dirección actual (CS Address Actual) + 1 (incrementa el CSAI)
- JUMP: dirección del campo JUMP ADDR del M<sup>o</sup>R.
- DECODE: op - op2 - op3 del scratchpad (IR[31, 30, 24-19])



$\rightarrow$  dirección donde está la instrucción de ROM próxima a ser ejecutada  
va de 2024 a (2048 - 4 = 2044), de 4 en 4.

$\rightarrow$  sobre si mirar op2 o op3 según valor op.

## Microprograma

### Assembler vs Microprograma

- Programa en lenguaje de bajo nivel:

- Se implementa en Assembler del procesador (por ej. ARC).
- Invisible al programador.
- Implementa programas de propósito general

### Microprograma

- Código que implementa cada instrucción del assembler
- 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

Sintaxis usada en ARC para el firmware:

Dirección en el control store: sentencia; /Comentario

→ En cada posición del LS hay una o más instrucciones del microcódigo.

En un ciclo de reloj se ejecutan todas las microinstrucciones de una posición del LS

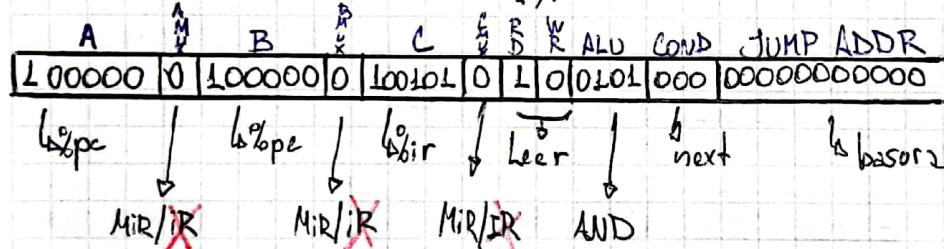
→ punto y coma solo las separa, pero todas se ejecutan al mismo tiempo.

### Decodificación de instrucciones - Microprograma

Primeras  
inst.

- 0:  $R[i_r] = AND(R[p_c], R[p_c])$ ; READ; /Lee una instrucción desde memoria principal
- 1. DECODE /Salto a microrutina que decodifica inst. ARC según opcode

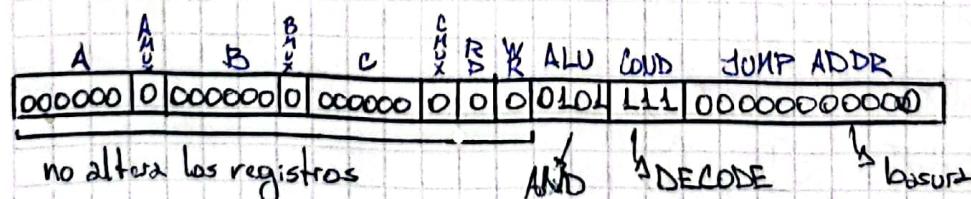
- 0:  $R[i_r] = AND(R[p_c], R[p_c])$ ; READ



En B podría poner cualquier cosa, y en la instrucción también (siempre y cuando no cambie los flags)

Sólo me importa poner el %pc en A, y el %ir en C, además de tener el read y el NEXT.

- 1. DECODE



En la instrucción pongo cualquier cosa que no altere flags

## EJEMPLO

### Addcc

- l600:  $\text{if } R[\text{IR}[13]] \text{ then GOTO l602;}$  / si el segundo operando está en modo inmediato salta  
 l601:  $R[\text{rd}] = \text{ADDCC}(R[\text{rs}_1], R[\text{rs}_2]);$  / addcc sobre registros  
 $\text{GOTO 2047;}$  / termina
- l602:  $R[\text{temp}0] = \text{SEXT13}[R[\text{IR}]]$  / variable auxiliar/temporal  
 guardado en  $\uparrow$  campo  $\text{simmL3}$  (la cte) y extiende signo
- l603:  $R[\text{rd}] = \text{ADDCC}(R[\text{rs}_1], R[\text{temp}0]);$  / addcc sobre registro y cte.  
 $\text{GOTO 2047;}$  / termina
- ...
- 2047:  $R[\text{pc}] = \text{INCPc}(R[\text{pc}]);$  / incremento en 4 pc y ...  
 $\text{GOTO 0;}$  / vuelvo a 0 para detectar instrucción.

### Saltos incondicionales

0,0,0	cond	op2	d:sp22	→ IR
-------	------	-----	--------	------

El microprograma utiliza  $[1\text{op}, \text{op2}, 1, 1, 00]$  → pero siempre apunta a lo mismo,  
necesito saber COND

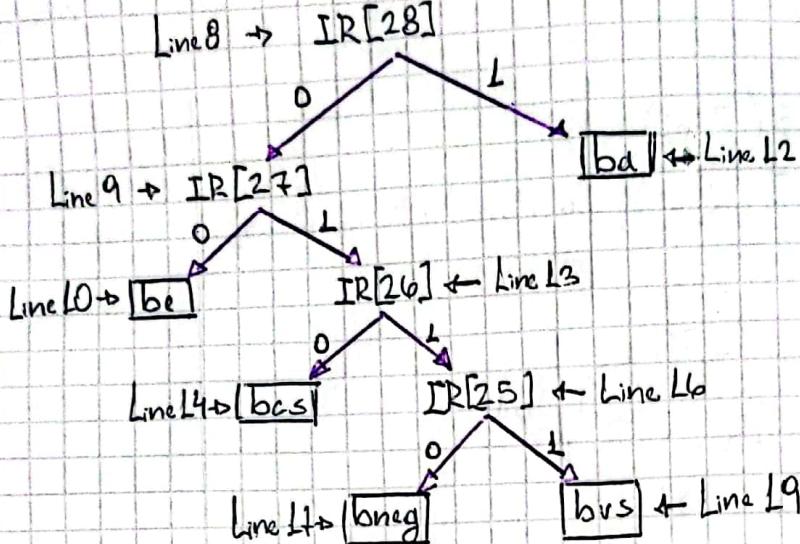
Además necesito saber dirección de destino d:sp22.

l088: GOTO 2; / Saber tipo de branch.

...

- 2 :  $R[\text{temp}0] = \text{LSHIFT10}(R[\text{r}1]);$   
 3 :  $R[\text{temp}0] = \text{RSHIFT5}(R[\text{temp}0]);$  → %temp0: d:sp22  
 4 :  $R[\text{temp}0] = \text{RSHIFT5}(R[\text{temp}0]);$

- 5:  $R[ir] = \text{RSHIFT5}(R[ir]);$
- 6:  $R[ir] = \text{RSHIFT5}(R[ir]); \rightarrow \text{Move COND to IR}[3]$
- 7:  $R[ir] = \text{RSHIFT5}(R[ir]);$
- 8: if  $R[IR][3]$  then GOTO L2;  
 $R[IR] = \text{ADD}(R[IR], R[IR]);$  | is bda?
- 9: if  $R[IR][2]$  then GOTO L3;  
 $R[IR] = \text{ADD}(R[IR], R[IR]);$  | is bne? (is not be)
- 10: if Z then GOTO L2; | execute be  
 $R[IR] = \text{ADD}(R[IR], R[IR]);$
- L1: GOTO 2047 | be not taken
- L2:  $R[pc] = \text{ADD}(R[pc], R[\text{temp}0]);$   
GOTO 0; | branch taken
- L3: if  $R[IR][3]$  then GOTO L6;  
 $R[IR] = \text{ADD}(R[IR], R[IR]);$  | is bcs?
- L4: if C then GOTO L2; | execute bcs
- L5: GOTO 2047; | bcs not taken
- L6: if  $R[IR][3]$  then GOTO L9; | is bvs?
- L7: if N then GOTO L2; | execute bneg
- L8: GOTO 2047; | bneg not taken
- L9: if V then GOTO L2; | execute bvs
- L10: GOTO 2047; | bus not taken
- ...
- 2047:  $R[pc] = \text{INCPCL}(R[pc]);$   
GOTO 0;



Cond	branch
0 0 0 1	be
0 1 0 1	bcs
0 1 1 0	bneg
0 1 1 1	bres
1 0 0 0	ba

## CALL

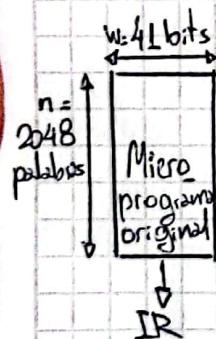
→ Con call no tengo op2 ni op3 → en  $2^6$  posiciones voy a tener problemas

Repite mismo microcódigo en 64 posiciones → mucho espacio ocupado pero muy ráce

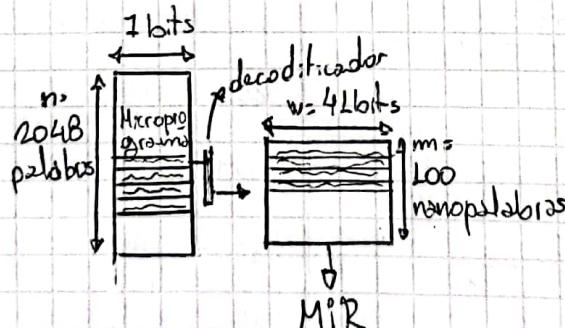
## Uso nanoprogramación

Tengo una tabla que tiene las microinstrucciones sin repetir, y en la memoria tengo punteros a la tabla → poco espacio ocupado pero lento

### Sin nanoprogramación



### Con nanoprogramación



NOTA