

Arquitectura de Computadoras

Requisitos

Conocer sistemas de numeración

- ▶ Sistema posicional
- ▶ Bases (2,10,16)
- ▶ Números signados (complementos)
- ▶ Aritmética de números signados (C/B,N,V,Z)
- ▶ Punto fijo y flotante
- ▶ Códigos

Conocer álgebra de boole

- ▶ Funciones booleanas
- ▶ Términos máximos y mínimos
- ▶ Simplificaciones
- ▶ Circuitos combinatorios
- ▶ Sumadores, codificadores, multiplexores

Conocer circuitos secuenciales

- ▶ Flip Flops (RS, D, T)
- ▶ Registro Latch
- ▶ Registro Shift
- ▶ Contadores
- ▶ Máquinas de estado finito.

Unidad 0 - Computadoras

Autores: Gho, Hnatiuk, Ferreyra

Versión 1.0.1

0.0 Computadoras

¿Qué es una computadora?

0.0.0 Bloques Funcionales

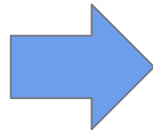
¿Qué es una computadora?



Programa:

- While(1) Escuchar audio hasta detectar palabra clave;
- Enviar audio a AWS y recibir transcripción (texto)
- Identificar en el texto palabras clave (acción, elemento, lugar)
- Ejecutar acción sobre elemento en lugar
- Saltar a Programa

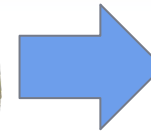
**Alexa: Encendé la luz
del comedor**



Micrófono



Amazon AWS

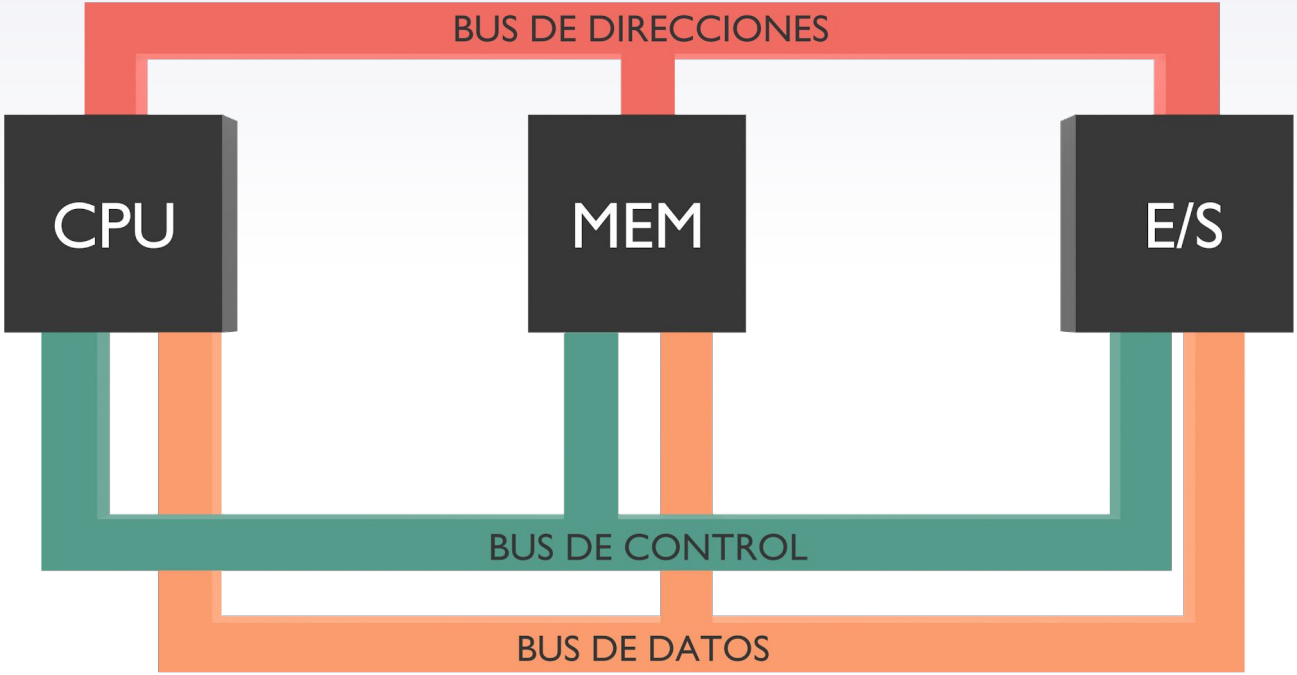


Relay



**Luz del
comedor**

Bloques Funcionales



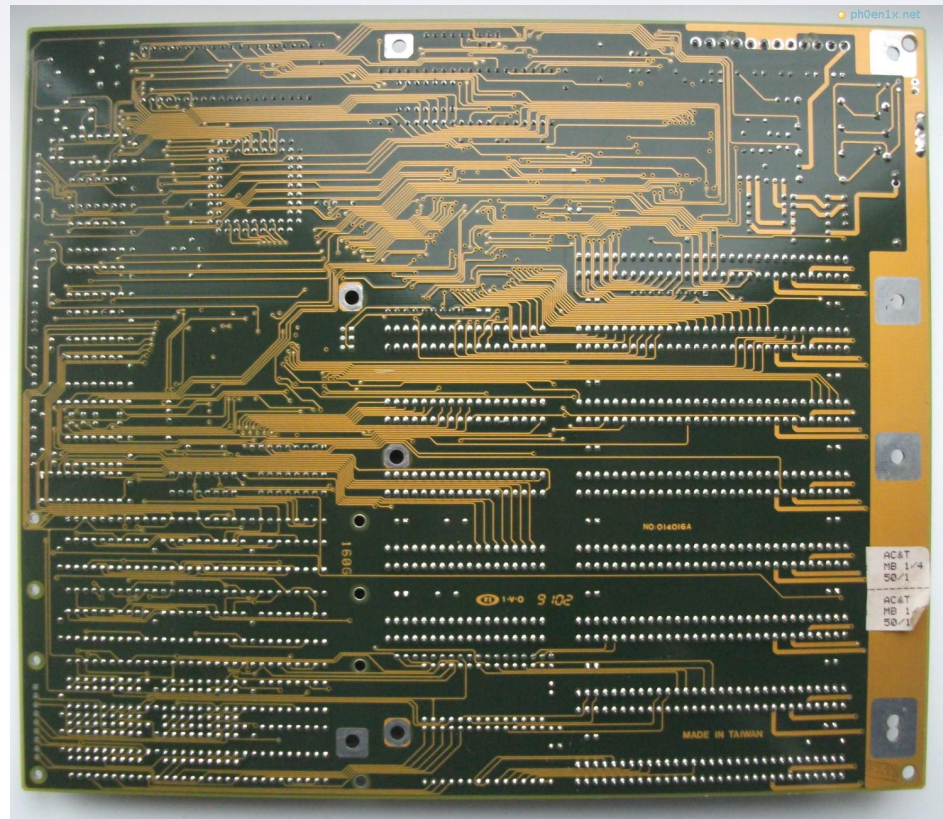
0.0.1 Buses

Conjunto de conductores que son interpretados en conjunto. Ej: El bus de direcciones es un conjunto de conductores que en un instante de tiempo representan una dirección. Cada uno de esos conductores tiene un peso en el valor de la dirección representada.

Este conjunto posee al menos dos terminales, y el sentido puede ser:

- Unidireccional: Uno de los terminales fuerza un valor en el bus mientras que el otro "lee" ese valor.
- Bidireccional: Puede cambiar quien es el terminal que fuerza el valor.

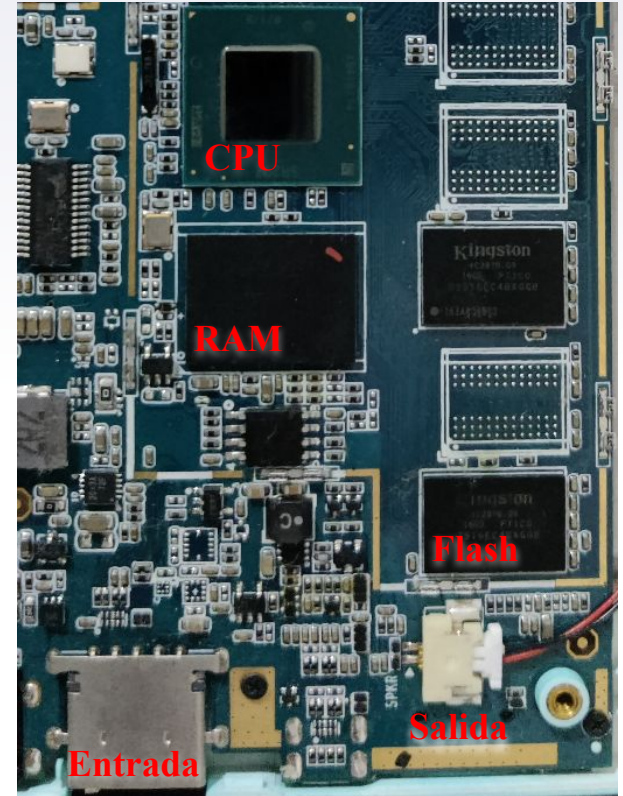
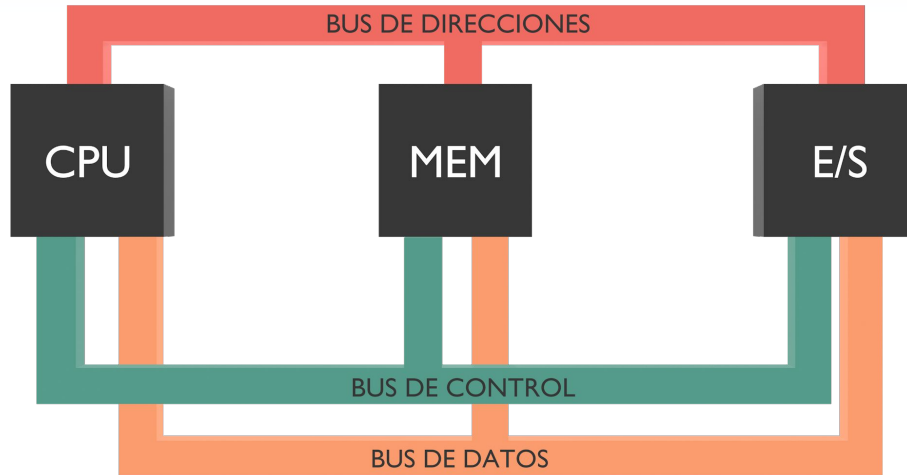
En ningún caso puede haber dos terminales al mismo tiempo forzando valores.



▶ 0.0.2 Unidad Central de Procesos (CPU)

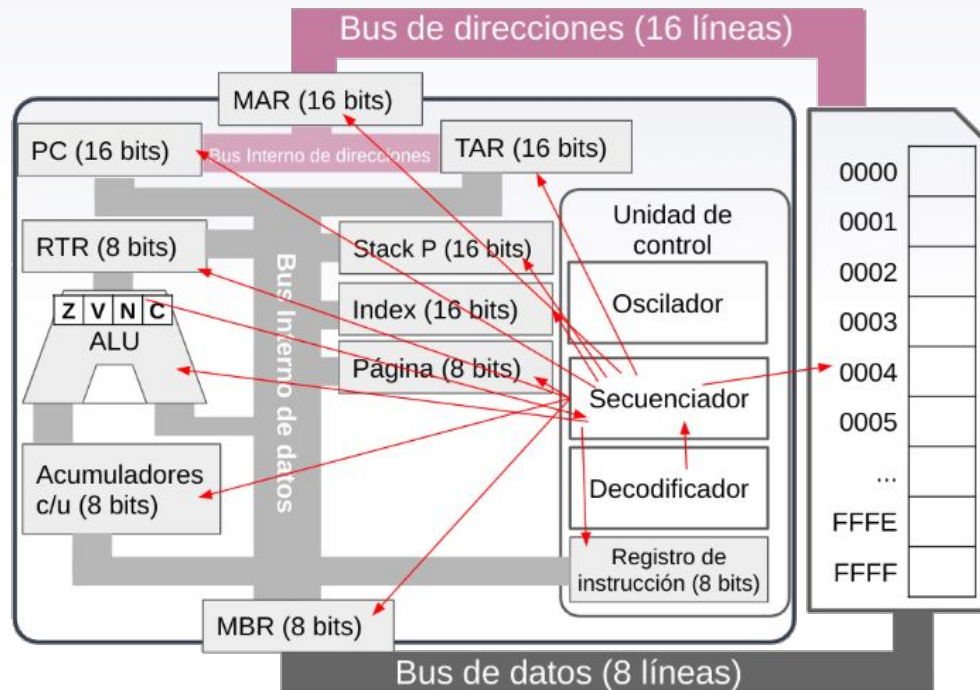
Máquina de estados programable

Una **CPU** tiene como finalidad ejecutar **instrucciones**. Las instrucciones se encuentran almacenadas en alguna unidad de **memoria** y son accedidas por la CPU mediante los **buses**. Las interfaces de **entrada/salida** son controladas por la CPU que interactúa con ellas siguiendo un **programa** (conjunto de instrucciones). Se utilizan **registros** para almacenar valores auxiliares o temporales durante la ejecución.



CPU Microprogramada

La unidad de control decodifica la instrucción y procede a ejecutar una secuencia de control sobre los distintos componentes de la CPU (registros, ALU, buses, etc).



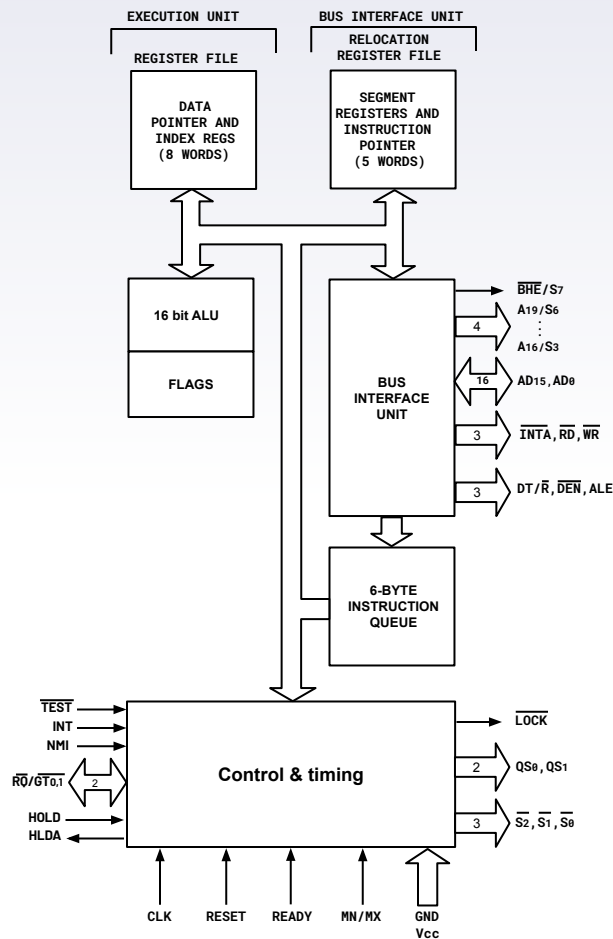


Diagrama de bloques de CPU de 8086

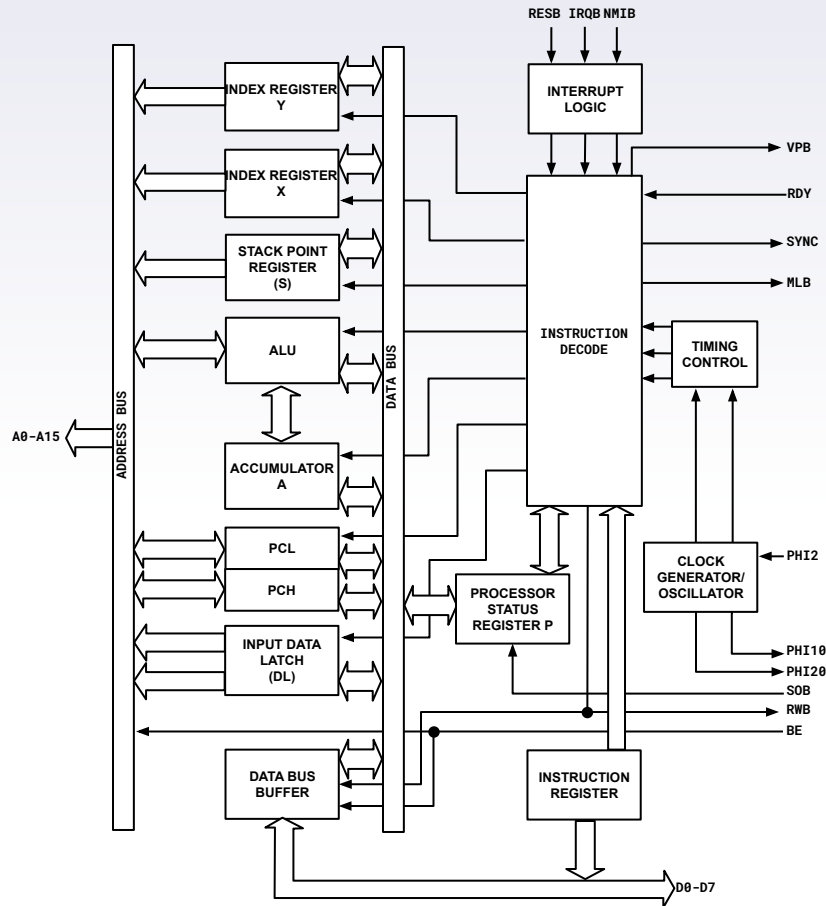


Diagrama de bloques simplificado de la arquitectura interna de W65C02S

Registros

ALU

Control

Timing

Buses internos

Buses externos

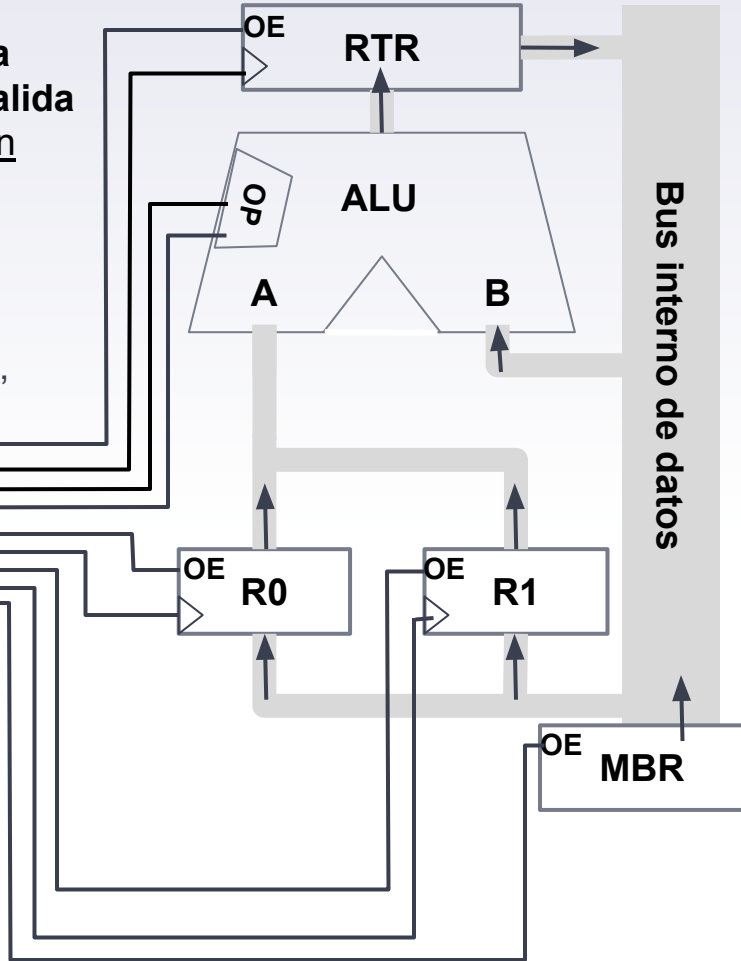
- ALU OP selecciona:
- 00 → **ALU= A + B**
 - 01 → **ALU= A - B**
 - 10 → **ALU= A + 0**
 - 11 → **ALU= A + 1**

La ALU necesita medio ciclo de clock para operar.

El MBR ya se encuentra cargado (no hace falta manejar su clock, solo su OE). Las flechas indican el flujo de datos.

En todos los registros **OE=1** habilita la salida del registro y **OE=0** pone la salida en Z (desconecta del bus). Cambios en OE no son instantáneos.
 Los clocks son activos en flanco ascendente para todos los registros.

Ciclo	OERTR	CLKRTR	ALUOP1	ALUOP0	OER0	CLKR0	OER1	CLKR1	OEMBR
000									



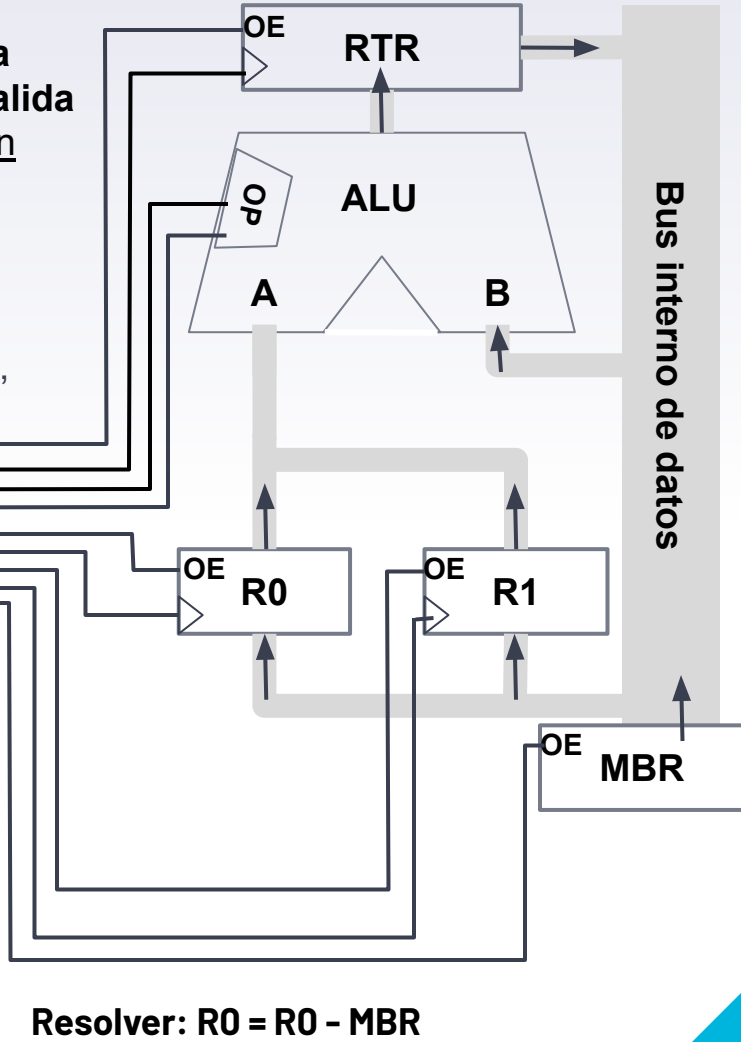
- ALU OP selecciona:
- 00 → **ALU= A + B**
 - 01 → **ALU= A - B**
 - 10 → **ALU= A + 0**
 - 11 → **ALU= A + 1**

La ALU necesita medio ciclo de clock para operar.

El MBR ya se encuentra cargado (no hace falta manejar su clock, solo su OE). Las flechas indican el flujo de datos.

En todos los registros **OE=1** habilita la salida del registro y **OE=0** pone la salida en Z (desconecta del bus). Cambios en OE no son instantáneos.
 Los clocks son activos en flanco ascendente para todos los registros.

Ciclo	OERTR	CLKRTR	ALUOP1	ALUOP0	OER0	CLKR0	OER1	CLKR1	OEMBR
000									



Resolver: **R0 = R0 - MBR**

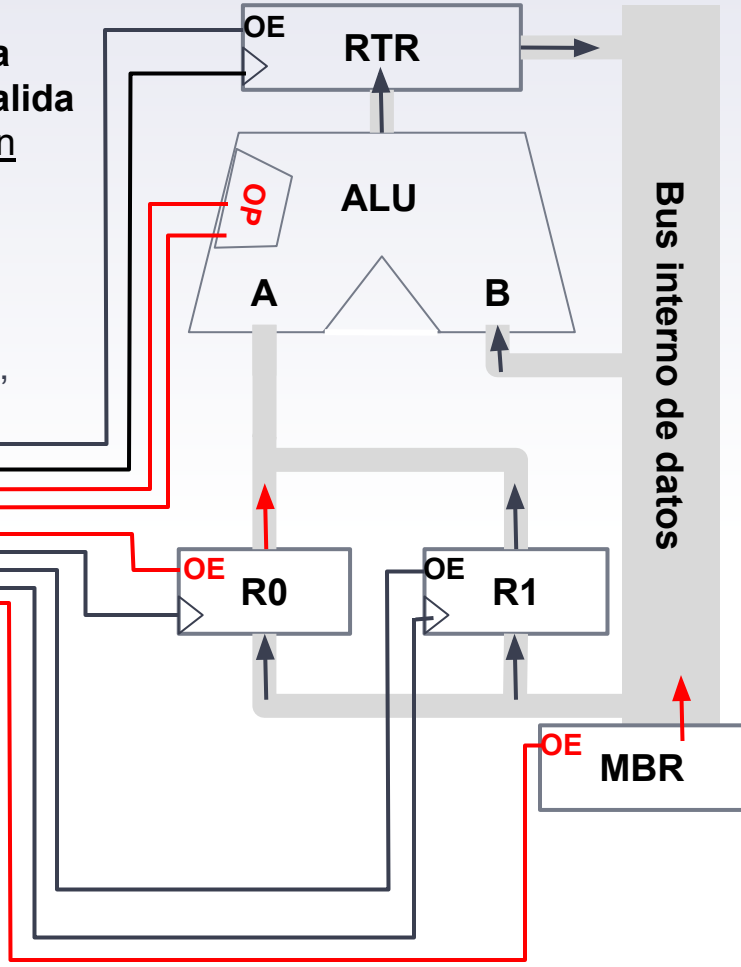
- ALU OP selecciona:
- 00 → $ALU = A + B$
 - 01 → $ALU = A - B$
 - 10 → $ALU = A + 0$
 - 11 → $ALU = A + 1$

La ALU necesita medio ciclo de clock para operar.

El MBR ya se encuentra cargado (no hace falta manejar su clock, solo su OE). Las flechas indican el flujo de datos.

En todos los registros **OE=1** habilita la salida del registro y **OE=0** pone la salida en Z (desconecta del bus). Cambios en OE no son instantáneos.
Los clocks son activos en flanco ascendente para todos los registros.

Ciclo	OERTR	CLKRTR	ALUOP1	ALUOP0	OER0	CLKR0	OER1	CLKR1	OEMBR
000	0	0	0	1	1	0	0	0	1



Resolver: $R0 = R0 - MBR$

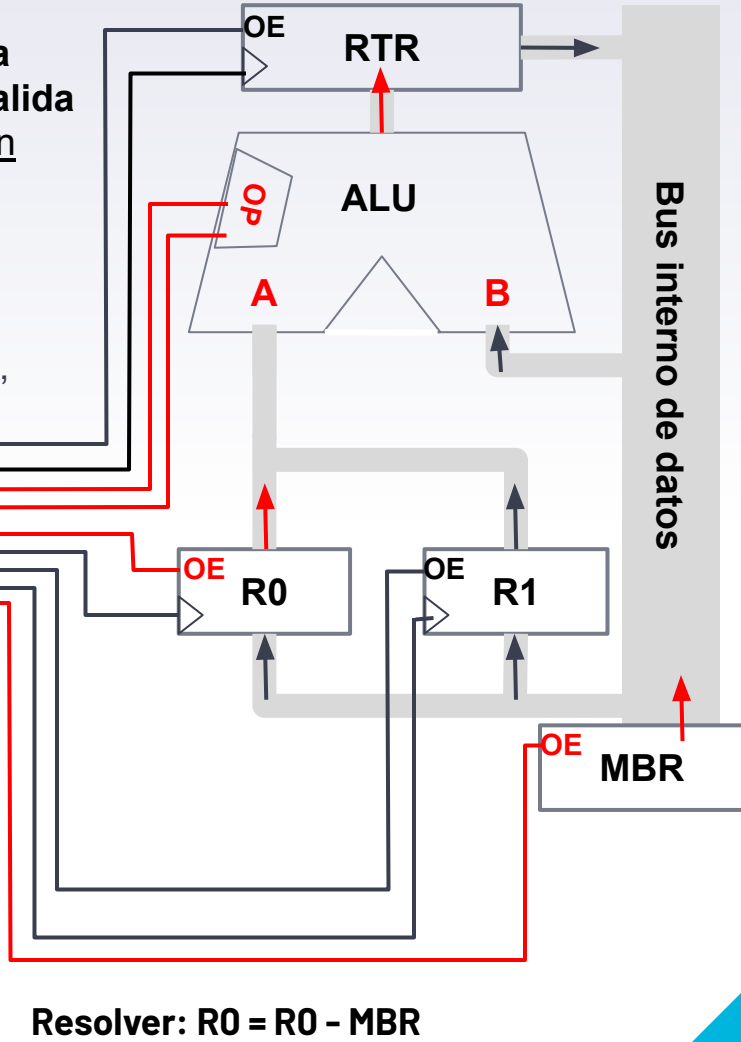
- ALU OP selecciona:
- 00 → **ALU= A + B**
 - 01 → **ALU= A - B**
 - 10 → **ALU= A + 0**
 - 11 → **ALU= A + 1**

La ALU necesita medio ciclo de clock para operar.

El MBR ya se encuentra cargado (no hace falta manejar su clock, solo su OE). Las flechas indican el flujo de datos.

En todos los registros **OE=1** habilita la salida del registro y **OE=0** pone la salida en Z (desconecta del bus). Cambios en OE no son instantáneos.
 Los clocks son activos en flanco ascendente para todos los registros.

Ciclo	OERTR	CLKRTR	ALUOP1	ALUOP0	OER0	CLKR0	OER1	CLKR1	OEMBR
000	0	0	0	1	1	0	0	0	1



Resolver: **R0 = R0 - MBR**

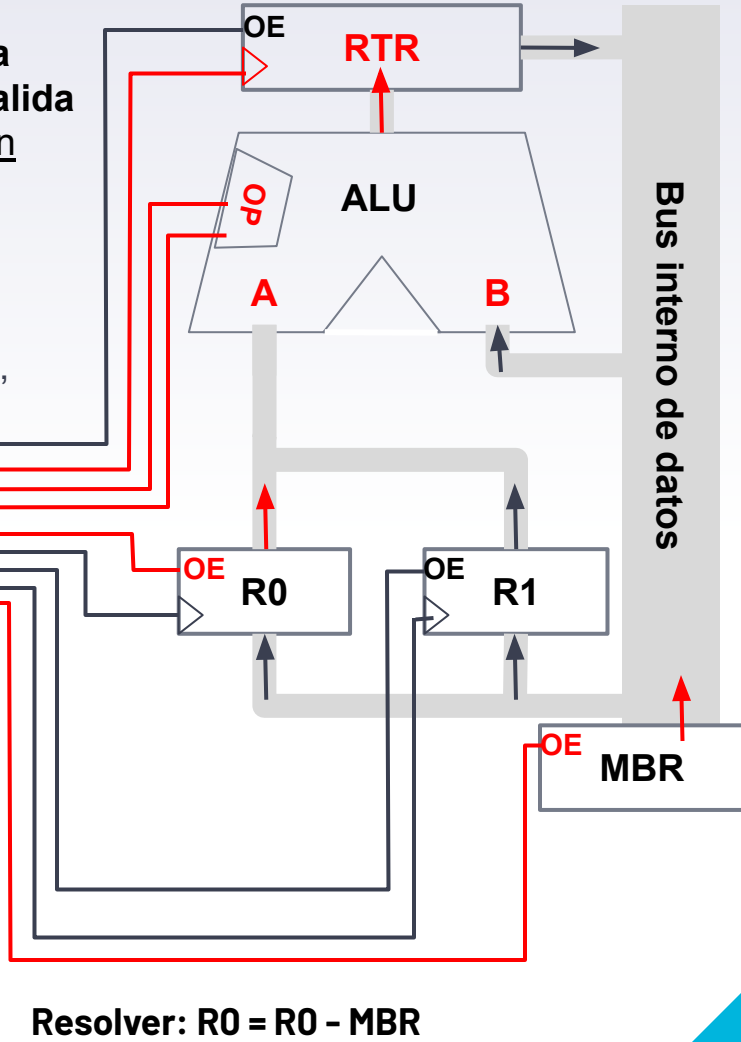
- ALU OP selecciona:
- 00 → **ALU= A + B**
 - 01 → **ALU= A - B**
 - 10 → **ALU= A + 0**
 - 11 → **ALU= A + 1**

La ALU necesita medio ciclo de clock para operar.

El MBR ya se encuentra cargado (no hace falta manejar su clock, solo su OE). Las flechas indican el flujo de datos.

En todos los registros **OE=1** habilita la salida del registro y **OE=0** pone la salida en Z (desconecta del bus). Cambios en OE no son instantáneos.
 Los clocks son activos en flanco ascendente para todos los registros.

Ciclo	OERTR	CLKRTR	ALUOP1	ALUOP0	OER0	CLKR0	OER1	CLKR1	OEMBR
000	0	0	0	1	1	0	0	0	1
001	0	1	0	1	1	0	0	0	1



Resolver: **R0 = R0 - MBR**

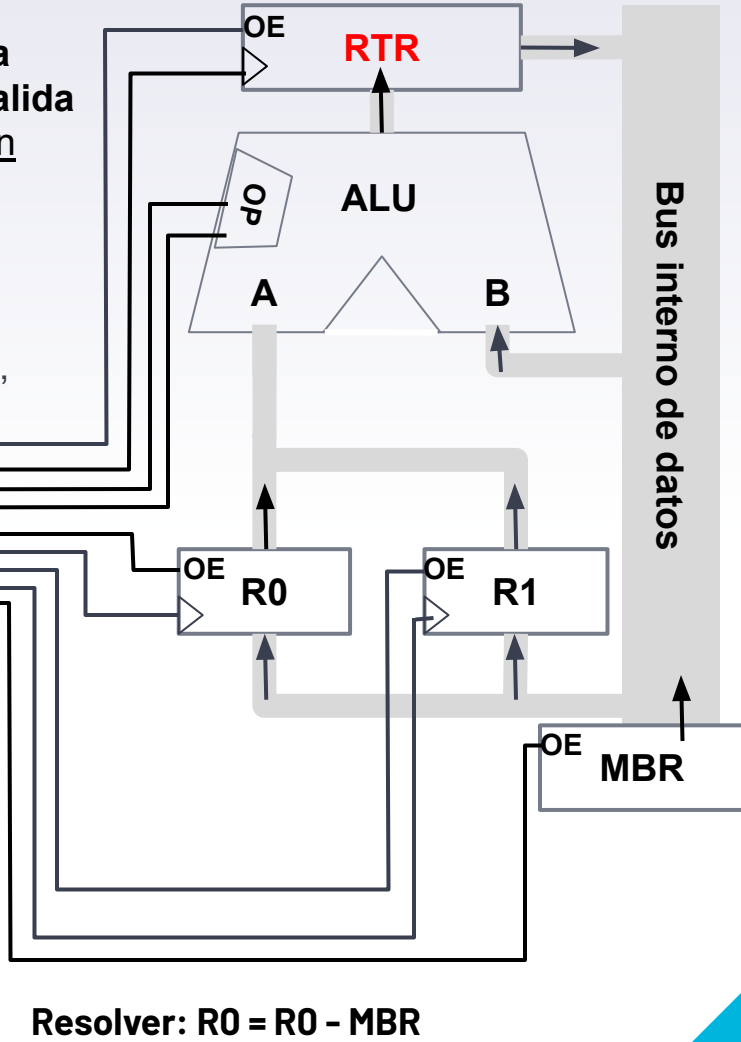
- ALU OP selecciona:
- 00 → **ALU= A + B**
 - 01 → **ALU= A - B**
 - 10 → **ALU= A + 0**
 - 11 → **ALU= A + 1**

La ALU necesita medio ciclo de clock para operar.

El MBR ya se encuentra cargado (no hace falta manejar su clock, solo su OE). Las flechas indican el flujo de datos.

En todos los registros **OE=1** habilita la salida del registro y **OE=0** pone la salida en Z (desconecta del bus). Cambios en OE no son instantáneos.
 Los clocks son activos en flanco ascendente para todos los registros.

Ciclo	OERTR	CLKRTR	ALUOP1	ALUOP0	OER0	CLKR0	OER1	CLKR1	OEMBR
000	0	0	0	1	1	0	0	0	1
001	0	1	0	1	1	0	0	0	1
010	0	0	0	0	0	0	0	0	0



Resolver: R0 = R0 - MBR

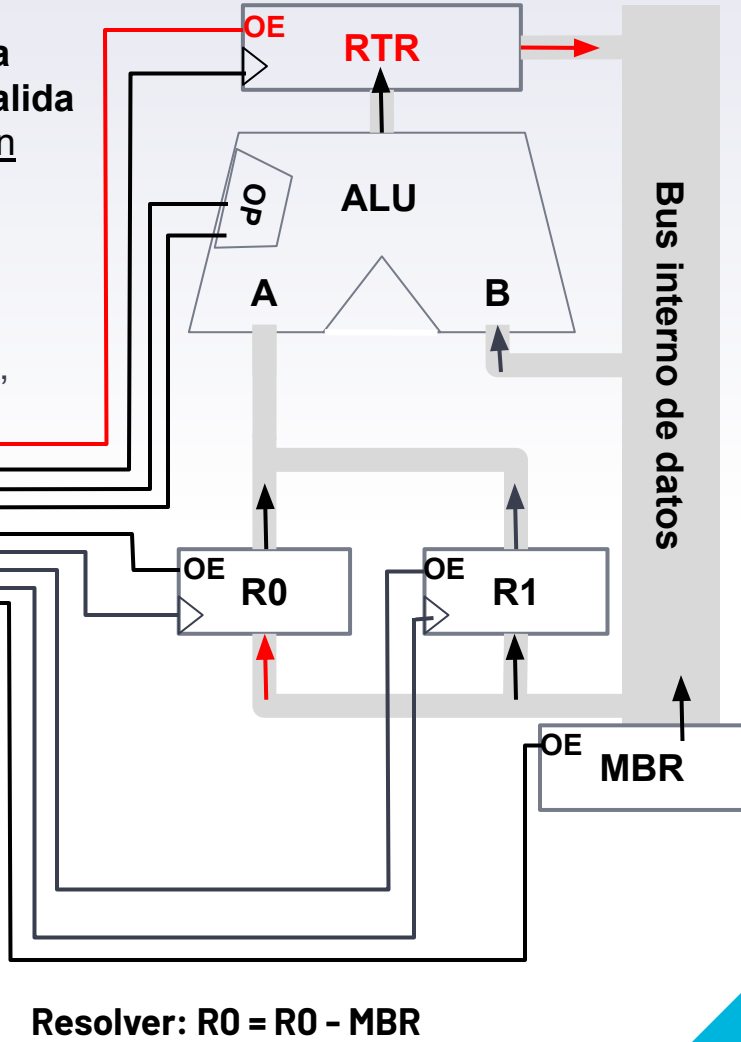
- ALU OP selecciona:
- 00 → **ALU= A + B**
 - 01 → **ALU= A - B**
 - 10 → **ALU= A + 0**
 - 11 → **ALU= A + 1**

La ALU necesita medio ciclo de clock para operar.

El MBR ya se encuentra cargado (no hace falta manejar su clock, solo su OE). Las flechas indican el flujo de datos.

En todos los registros **OE=1** habilita la salida del registro y **OE=0** pone la salida en Z (desconecta del bus). Cambios en OE no son instantáneos.
Los clocks son activos en flanco ascendente para todos los registros.

Ciclo	OERTR	CLKRTR	ALUOP1	ALUOP0	OER0	CLKR0	OER1	CLKR1	OEMBR
000	0	0	0	1	1	0	0	0	1
001	0	1	0	1	1	0	0	0	1
010	0	0	0	0	0	0	0	0	0
011	1	0	0	0	0	0	0	0	0



Resolver: **R0 = R0 - MBR**

ALU OP selecciona:

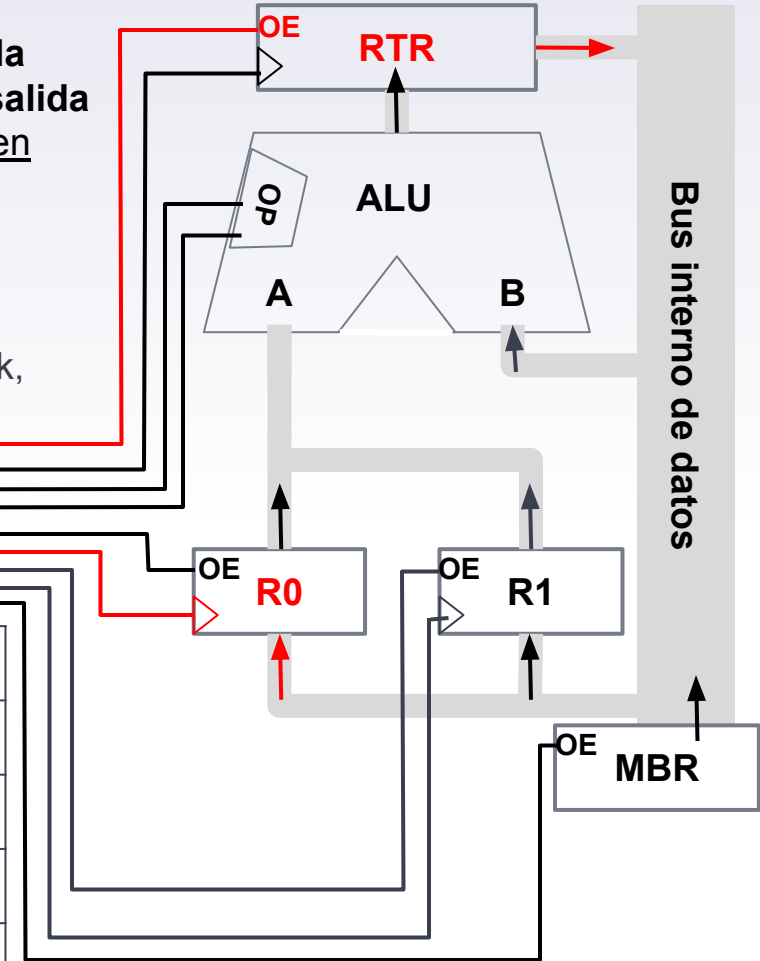
- 00 → ALU= A + B
- 01 → ALU= A - B
- 10 → ALU= A + 0
- 11 → ALU= A + 1

La ALU necesita medio ciclo de clock para operar.

El MBR ya se encuentra cargado (no hace falta manejar su clock, solo su OE). **Las flechas indican el flujo de datos.**

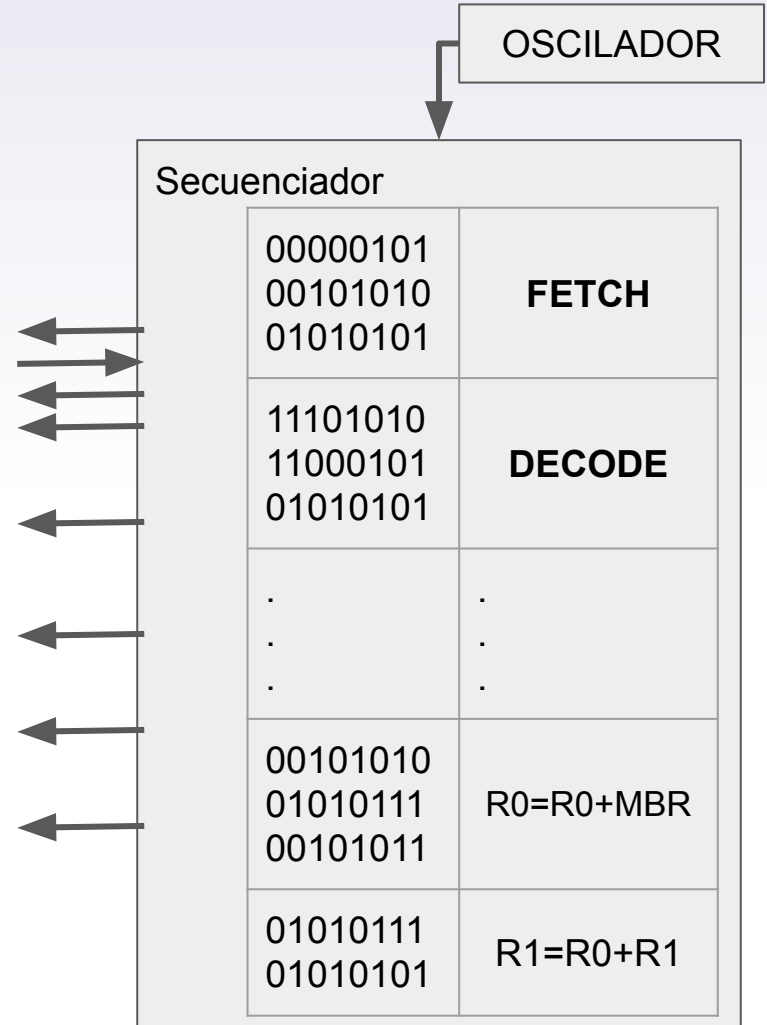
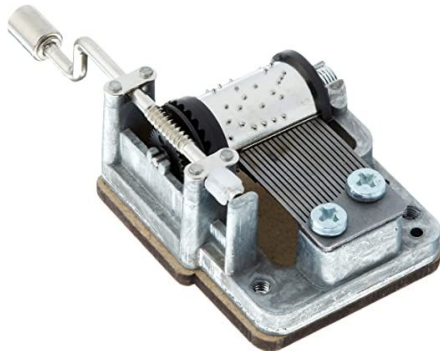
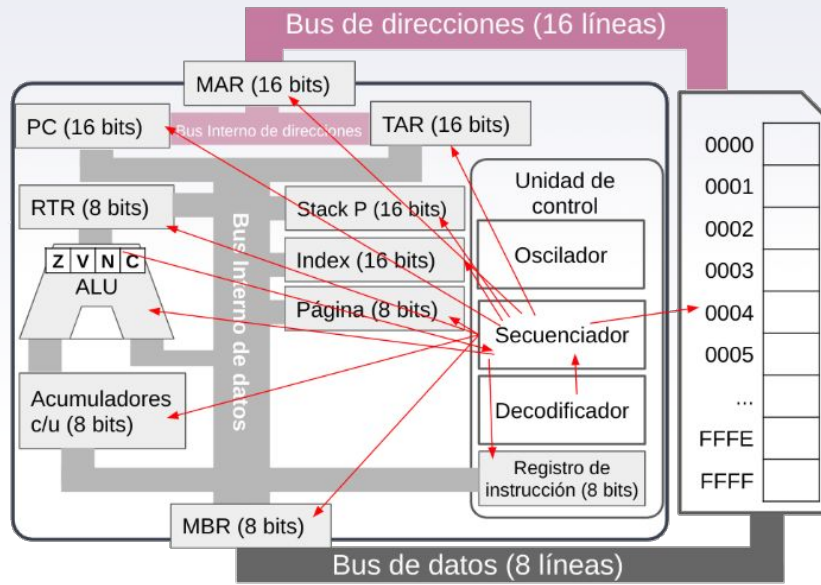
En todos los registros **OE=1** habilita la salida del registro y **OE=0** pone la salida en **Z** (desconecta del bus). Cambios en OE no son instantáneos.

Los clocks son activos en flanco ascendente para todos los registros.



Resolver: $R0 = R0 - MBR$

Ciclo	OERTR	CLKRTR	ALUOP1	ALUOP0	OER0	CLKR0	OER1	CLKR1	OEMBR
000	0	0	0	1	1	0	0	0	1
001	0	1	0	1	1	0	0	0	1
010	0	0	0	0	0	0	0	0	0
011	1	0	0	0	0	0	0	0	0
100	1	0	0	0	0	1	0	0	0

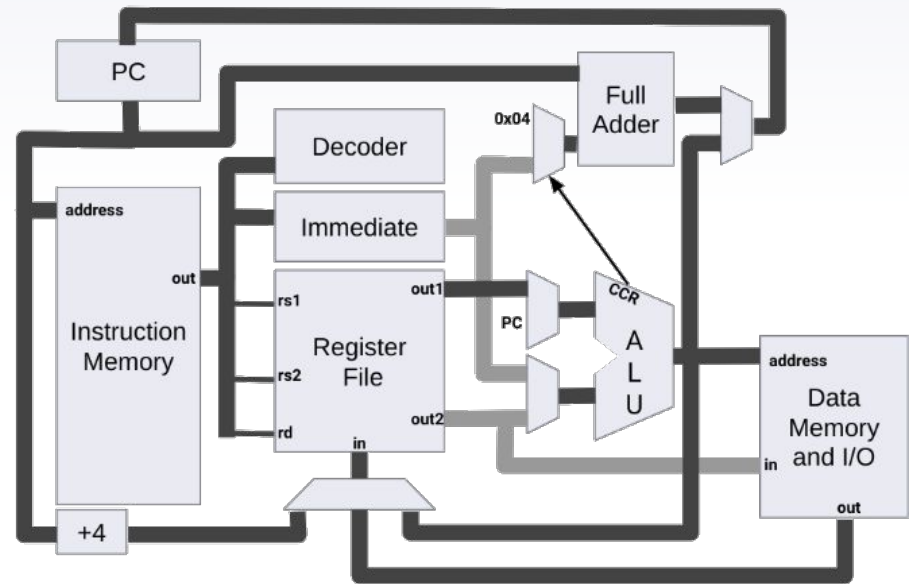


CPU Cableada

La instrucción se decodifica y se arma el "datapath", o sea, como fluyen los datos dentro de la CPU.

Los buses internos son siempre unidireccionales.

Esta estructura permite resolver las etapas en serie, siendo beneficioso para la implementación de pipelining.



RI=

Inm	Signo	Rd		Rs1		Rs2	
Ri7	Ri6	Ri5	Ri4	Ri3	Ri2	Ri1	Ri0

El bit **Inm** define que sale por Mux2

- **Inm=0** → Sale A (ALU)
- **Inm=1** → Sale un número inmediato de 5 bits compuesto por:
Signo Ri3 Ri2 Ri1 Ri0

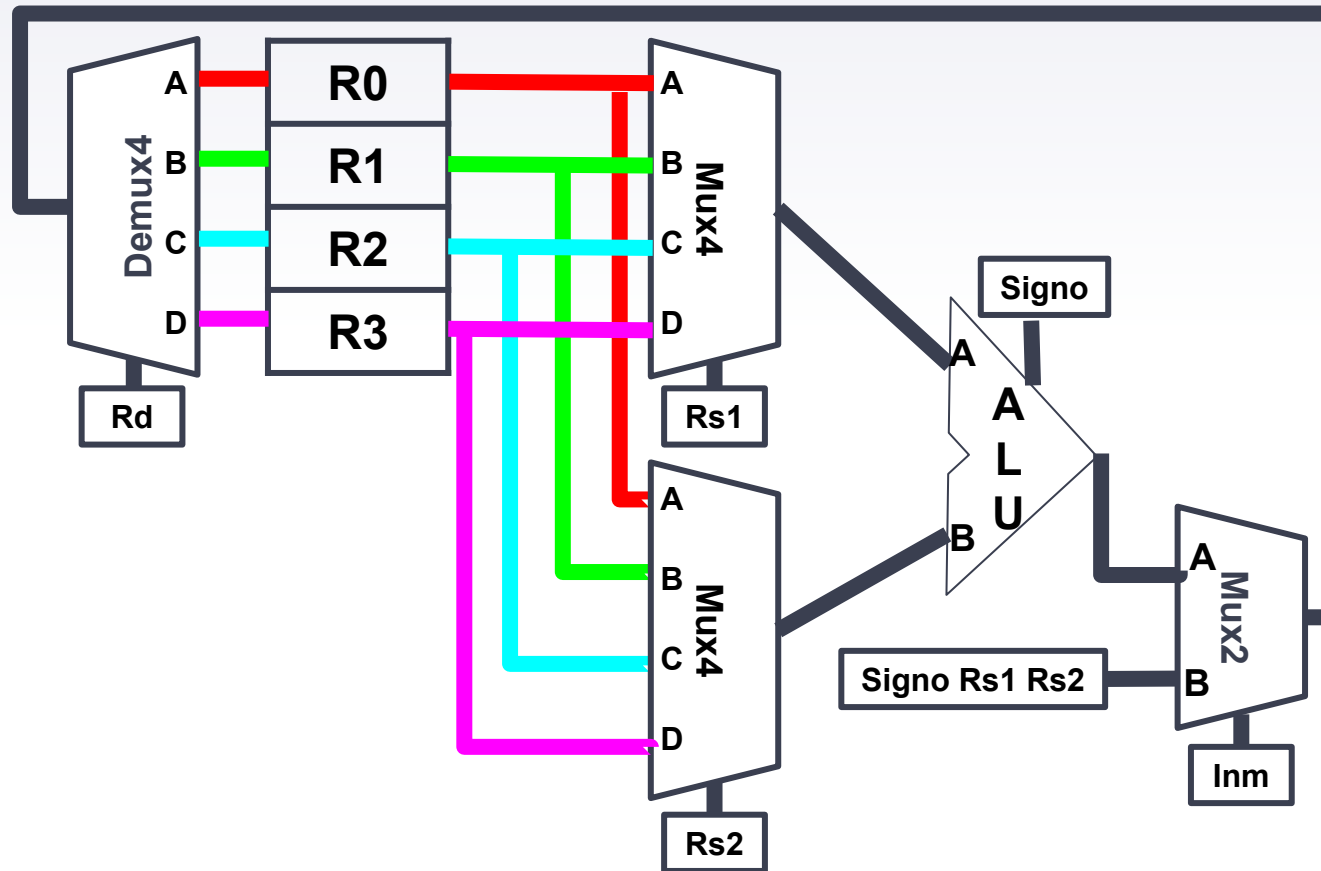
El bit **Signo** define la operación de la ALU

- **Signo=1** → $ALU = A - B$
- **Signo=0** → $ALU = A + B$

Los **Mux4** utilizan 2 bits de selección y eligen a su salida

- **00** = A
- **01** = B
- **10** = C
- **11** = D

El **Demux4** copia la entrada en A,B,C o D respectivamente según el valor de Rd.



RI=

Inm	Signo	Rd		Rs1		Rs2	
Ri7	Ri6	Ri5	Ri4	Ri3	Ri2	Ri1	Ri0

El bit **Inm** define que sale por Mux2

- **Inm=0** → Sale A (ALU)
- **Inm=1** → Sale un número inmediato de 5 bits compuesto por:
Signo Ri3 Ri2 Ri1 Ri0

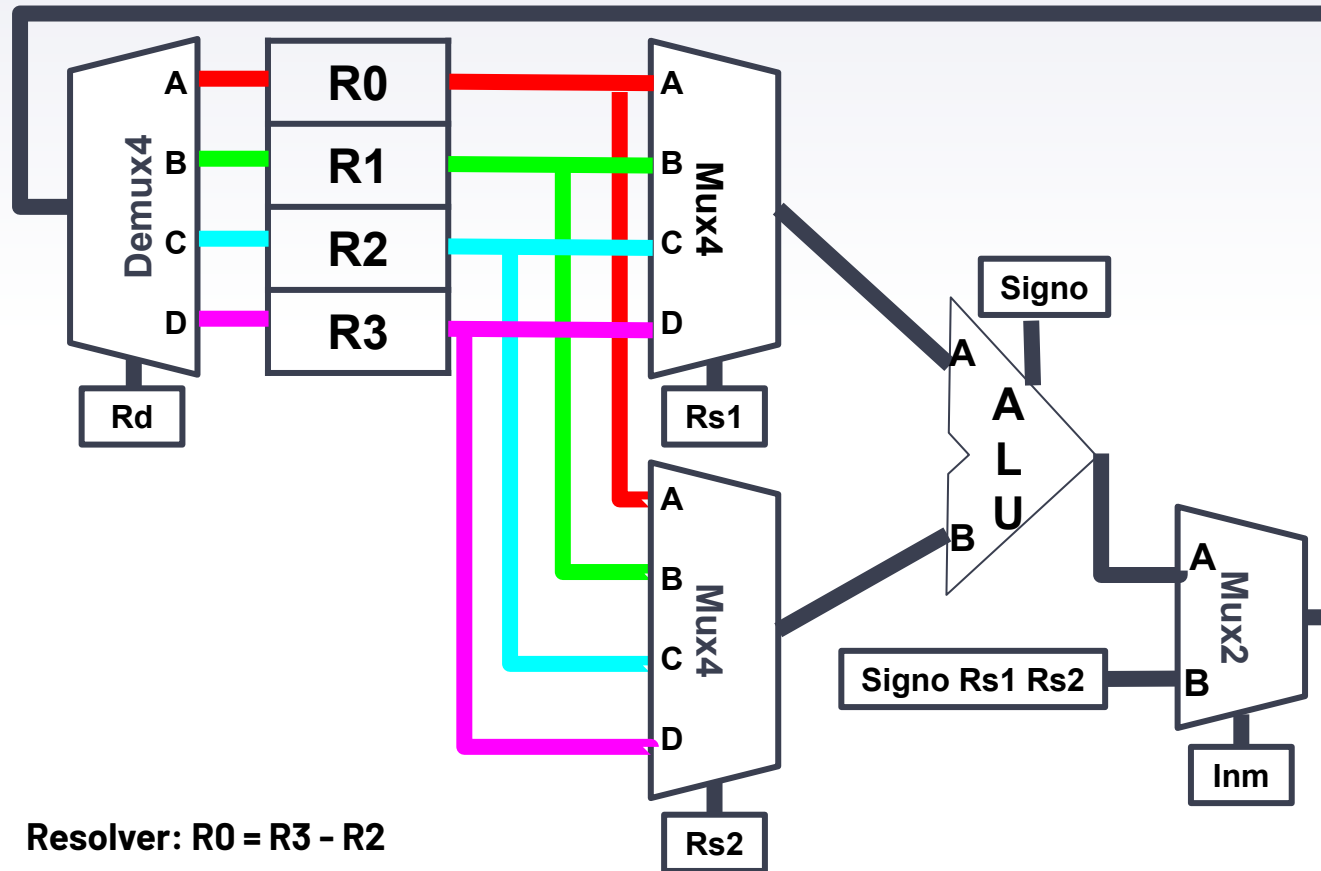
El bit **Signo** define la operación de la ALU

- **Signo=1** → $ALU = A - B$
- **Signo=0** → $ALU = A + B$

Los **Mux4** utilizan 2 bits de selección y eligen a su salida

- **00** = A
- **01** = B
- **10** = C
- **11** = D

El **Demux4** copia la entrada en A,B,C o D respectivamente según el valor de Rd.



RI=

Inm	Signo	Rd		Rs1		Rs2	
Ri7	Ri6	0	0	1	1	1	0

El bit **Inm** define que sale por Mux2

- **Inm=0** → Sale A (ALU)
- **Inm=1** → Sale un número inmediato de 5 bits compuesto por:
Signo Ri3 Ri2 Ri1 Ri0

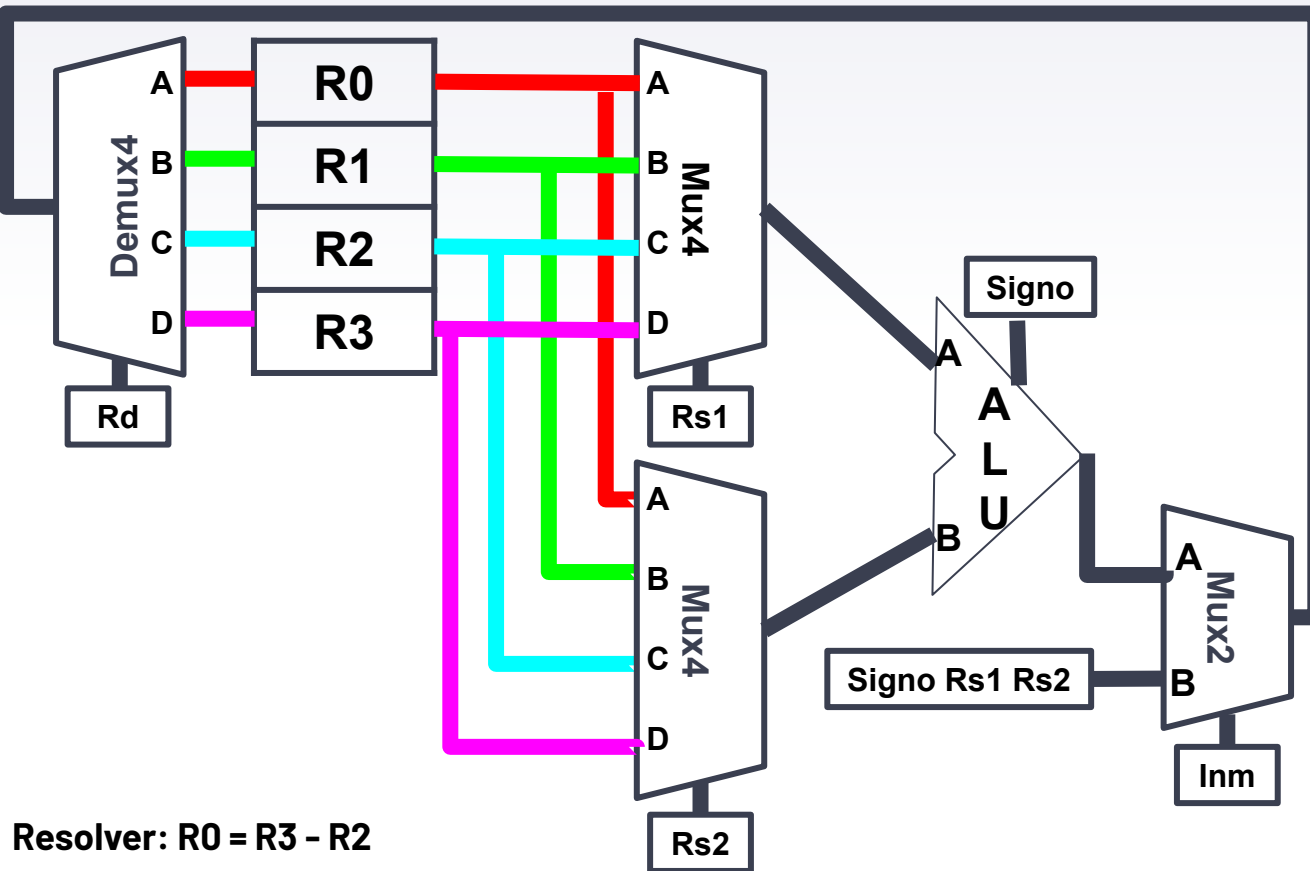
El bit **Signo** define la operación de la ALU

- **Signo=1** → $ALU = A - B$
- **Signo=0** → $ALU = A + B$

Los **Mux4** utilizan 2 bits de selección y eligen a su salida

- **00** = A
- **01** = B
- **10** = C
- **11** = D

El **Demux4** copia la entrada en A,B,C o D respectivamente según el valor de Rd.



RI=

Inm	Signo	Rd		Rs1		Rs2	
0	1	0	0	1	1	1	0

El bit **Inm** define que sale por Mux2

- **Inm=0** → Sale A (ALU)
- **Inm=1** → Sale un número inmediato de 5 bits compuesto por:
Signo Ri3 Ri2 Ri1 Ri0

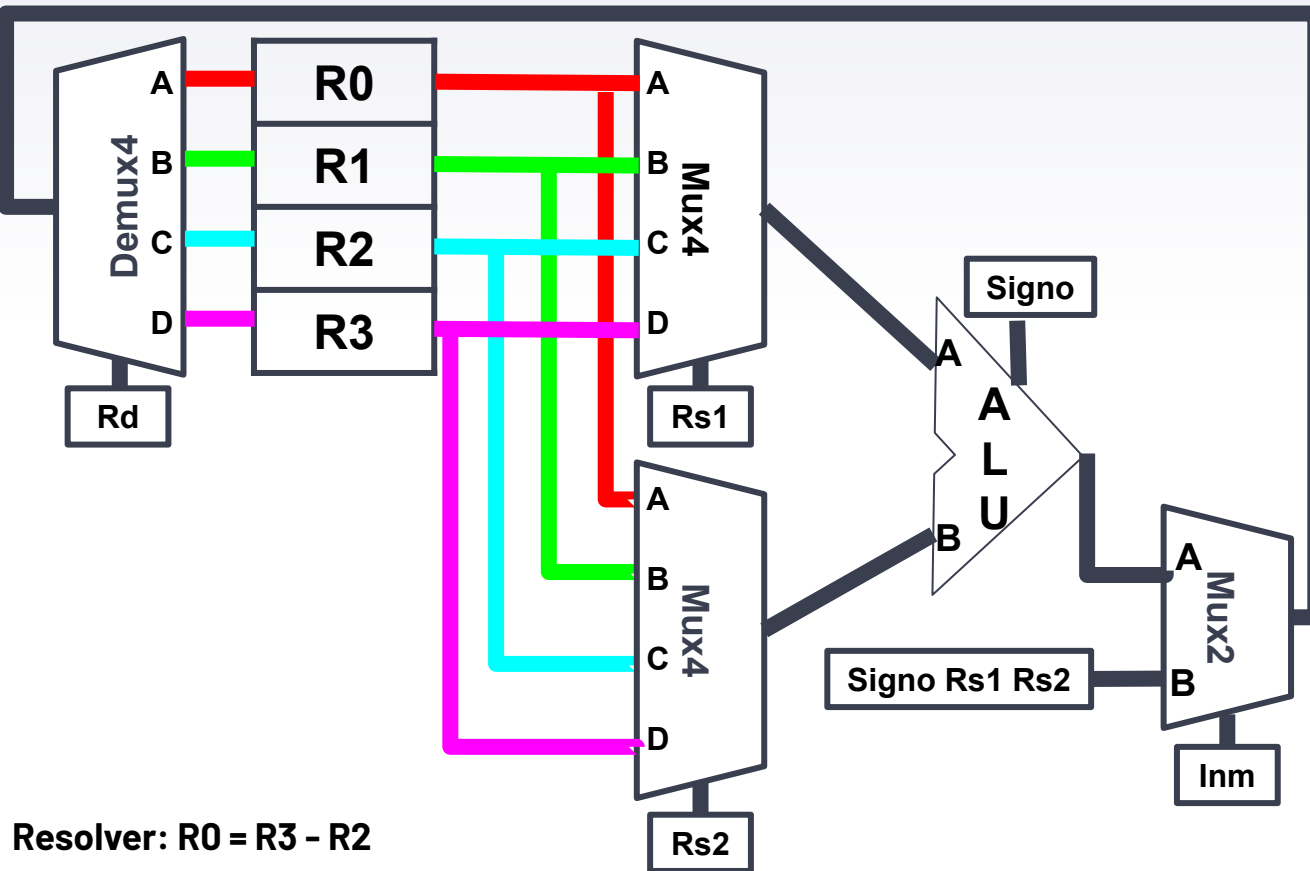
El bit **Signo** define la operación de la ALU

- **Signo=1** → $ALU = A - B$
- **Signo=0** → $ALU = A + B$

Los **Mux4** utilizan 2 bits de selección y eligen a su salida

- **00** = A
- **01** = B
- **10** = C
- **11** = D

El **Demux4** copia la entrada en A,B,C o D respectivamente según el valor de Rd.



Resolver: $R0 = R3 - R2$

RI=

Inm	Signo	Rd		Rs1		Rs2	
0	1	0	0	1	1	1	0

El bit **Inm** define que sale por Mux2

- **Inm=0** → Sale A (ALU)
- **Inm=1** → Sale un número inmediato de 5 bits compuesto por:
Signo Ri3 Ri2 Ri1 Ri0

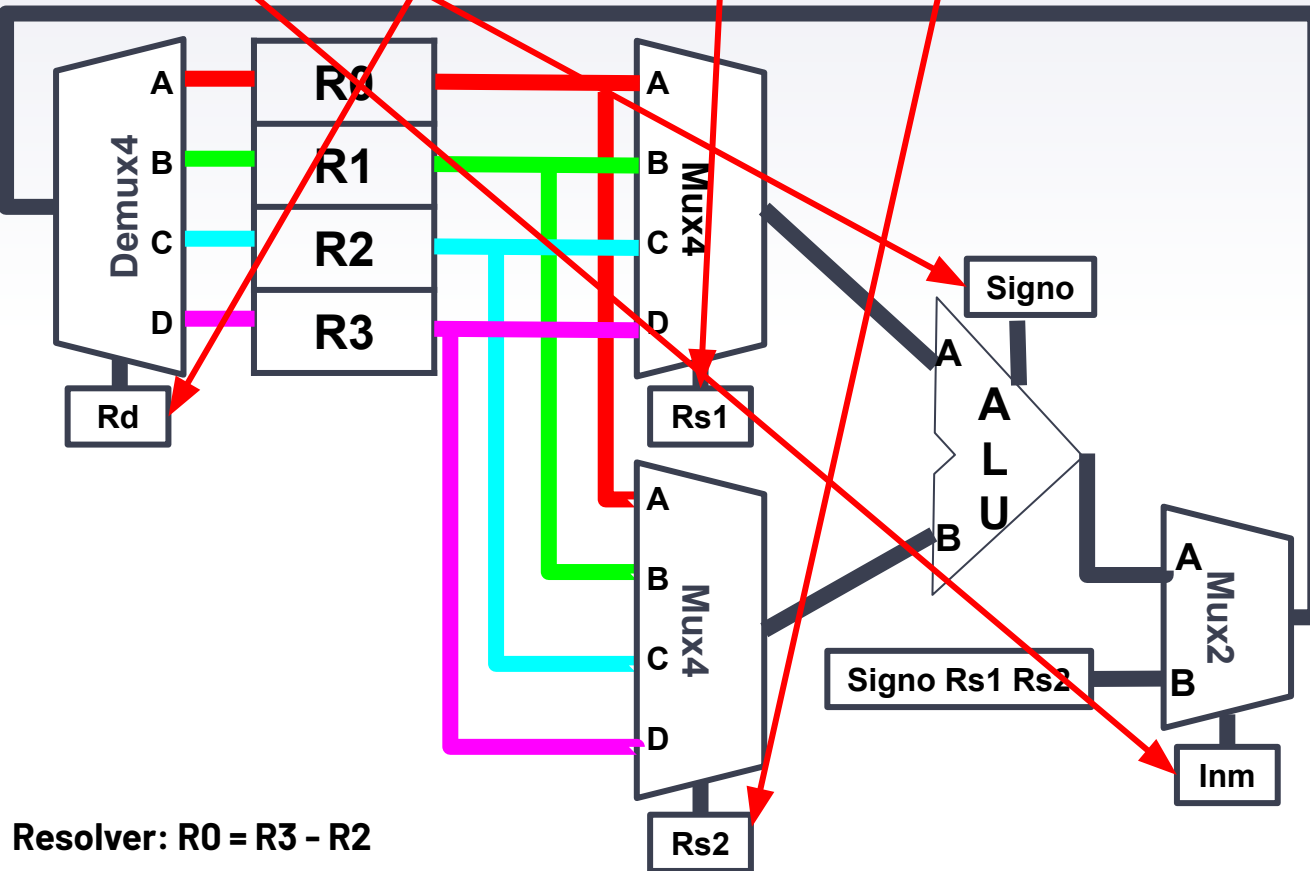
El bit **Signo** define la operación de la ALU

- **Signo=1** → $ALU = A - B$
- **Signo=0** → $ALU = A + B$

Los **Mux4** utilizan 2 bits de selección y eligen a su salida

- **00** = A
- **01** = B
- **10** = C
- **11** = D

El **Demux4** copia la entrada en A,B,C o D respectivamente según el valor de Rd.



Resolver: $R0 = R3 - R2$

RI=

Inm	Signo	Rd		Rs1		Rs2	
0	1	0	0	1	1	1	0

El bit **Inm** define que sale por Mux2

- **Inm=0** → Sale A (ALU)
- **Inm=1** → Sale un número inmediato de 5 bits compuesto por:
Signo Ri3 Ri2 Ri1 Ri0

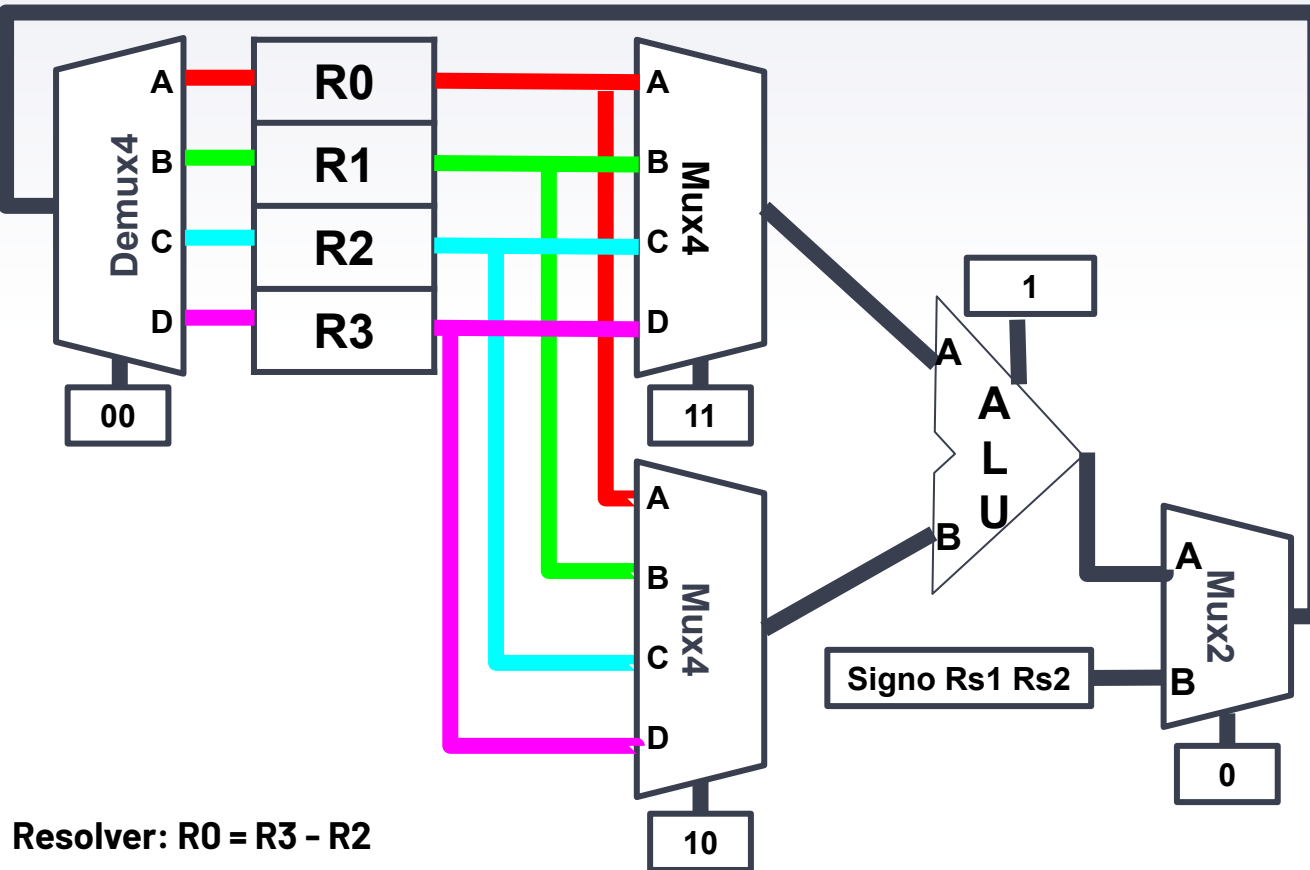
El bit **Signo** define la operación de la ALU

- **Signo=1** → $ALU = A - B$
- **Signo=0** → $ALU = A + B$

Los **Mux4** utilizan 2 bits de selección y eligen a su salida

- **00** = A
- **01** = B
- **10** = C
- **11** = D

El **Demux4** copia la entrada en A,B,C o D respectivamente según el valor de Rd.



Resolver: **R0 = R3 - R2**

RI=

Inm	Signo	Rd		Rs1		Rs2	
0	1	0	0	1	1	1	0

El bit **Inm** define que sale por Mux2

- **Inm=0** → Sale A (ALU)
- **Inm=1** → Sale un número inmediato de 5 bits compuesto por:
Signo Ri3 Ri2 Ri1 Ri0

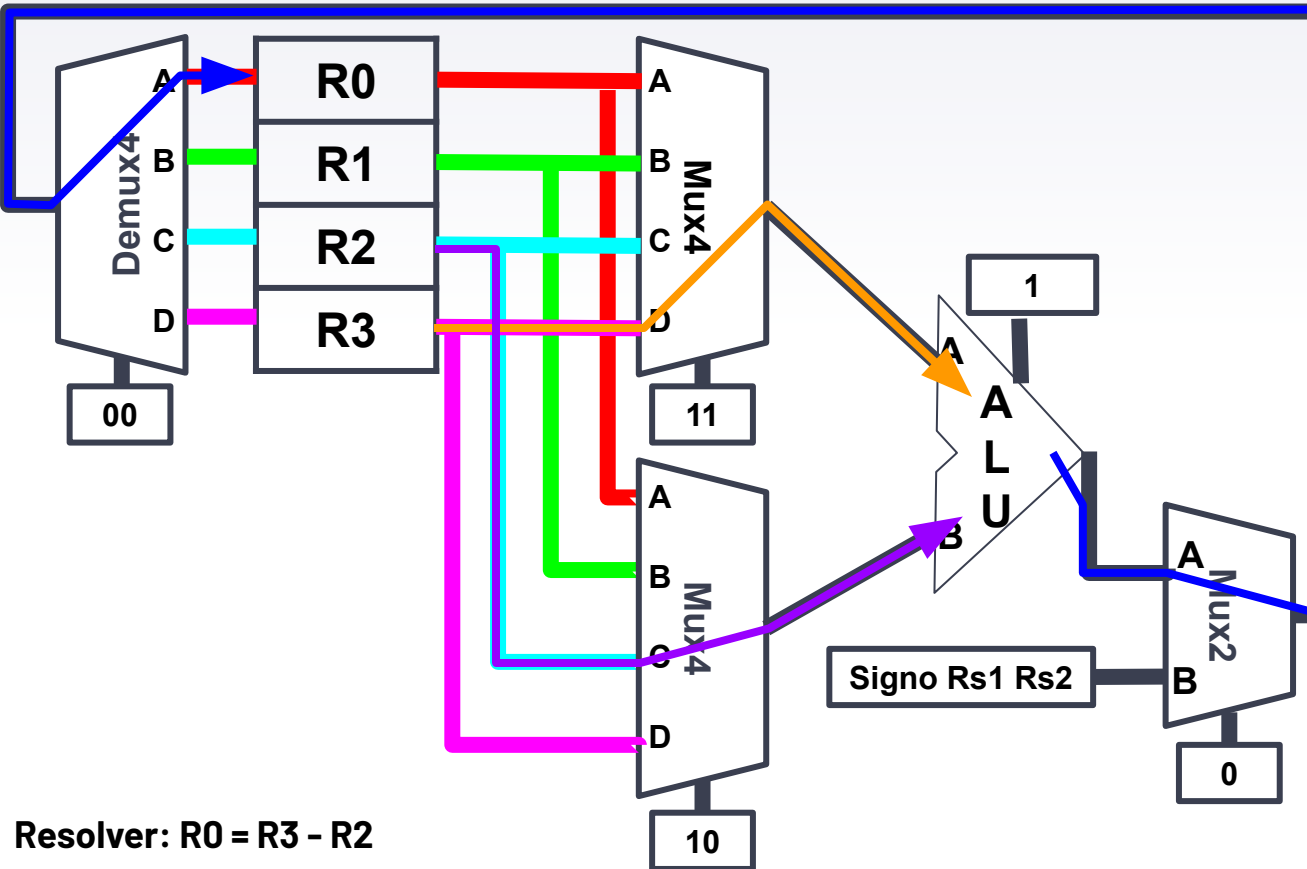
El bit **Signo** define la operación de la ALU

- **Signo=1** → $ALU = A - B$
- **Signo=0** → $ALU = A + B$

Los **Mux4** utilizan 2 bits de selección y eligen a su salida

- **00** = A
- **01** = B
- **10** = C
- **11** = D

El **Demux4** copia la entrada en A,B,C o D respectivamente según el valor de Rd.



RI=

Inm	Signo	Rd		Rs1		Rs2	
Ri7	Ri6	Ri5	Ri4	Ri3	Ri2	Ri1	Ri0

El bit **Inm** define que sale por Mux2

- **Inm=0** → Sale A (ALU)
- **Inm=1** → Sale un número inmediato de 5 bits compuesto por:
Signo Ri3 Ri2 Ri1 Ri0

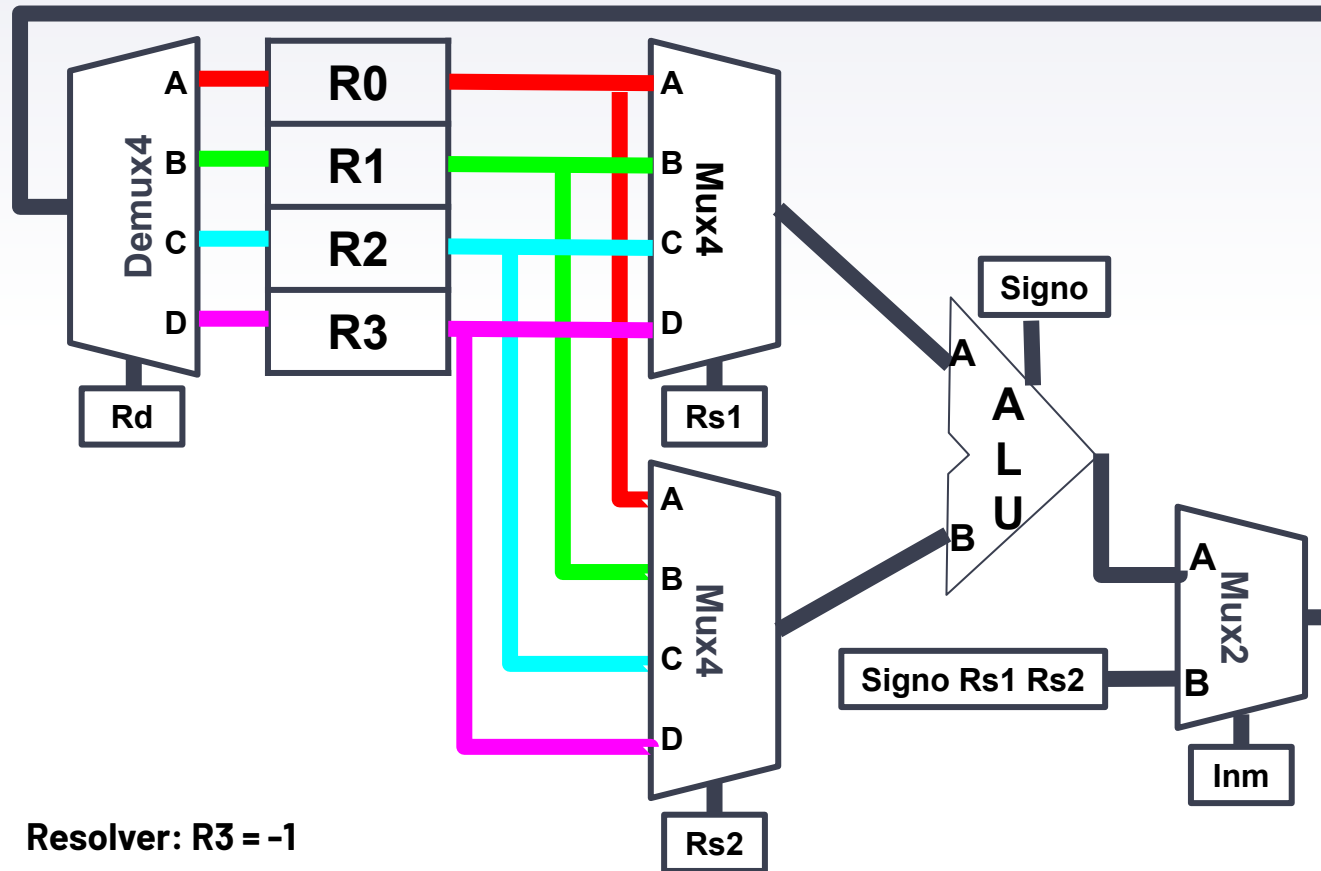
El bit **Signo** define la operación de la ALU

- **Signo=1** → $ALU = A - B$
- **Signo=0** → $ALU = A + B$

Los **Mux4** utilizan 2 bits de selección y eligen a su salida

- **00** = A
- **01** = B
- **10** = C
- **11** = D

El **Demux4** copia la entrada en A,B,C o D respectivamente según el valor de Rd.



Resolver: R3 = -1

RI=

Inm	Signo	Rd		Rs1		Rs2	
Ri7	Ri6	Ri5	Ri4	Ri3	Ri2	Ri1	Ri0

El bit **Inm** define que sale por Mux2

- **Inm=0** → Sale A (ALU)
- **Inm=1** → Sale un número inmediato de 5 bits compuesto por:
Signo Ri3 Ri2 Ri1 Ri0

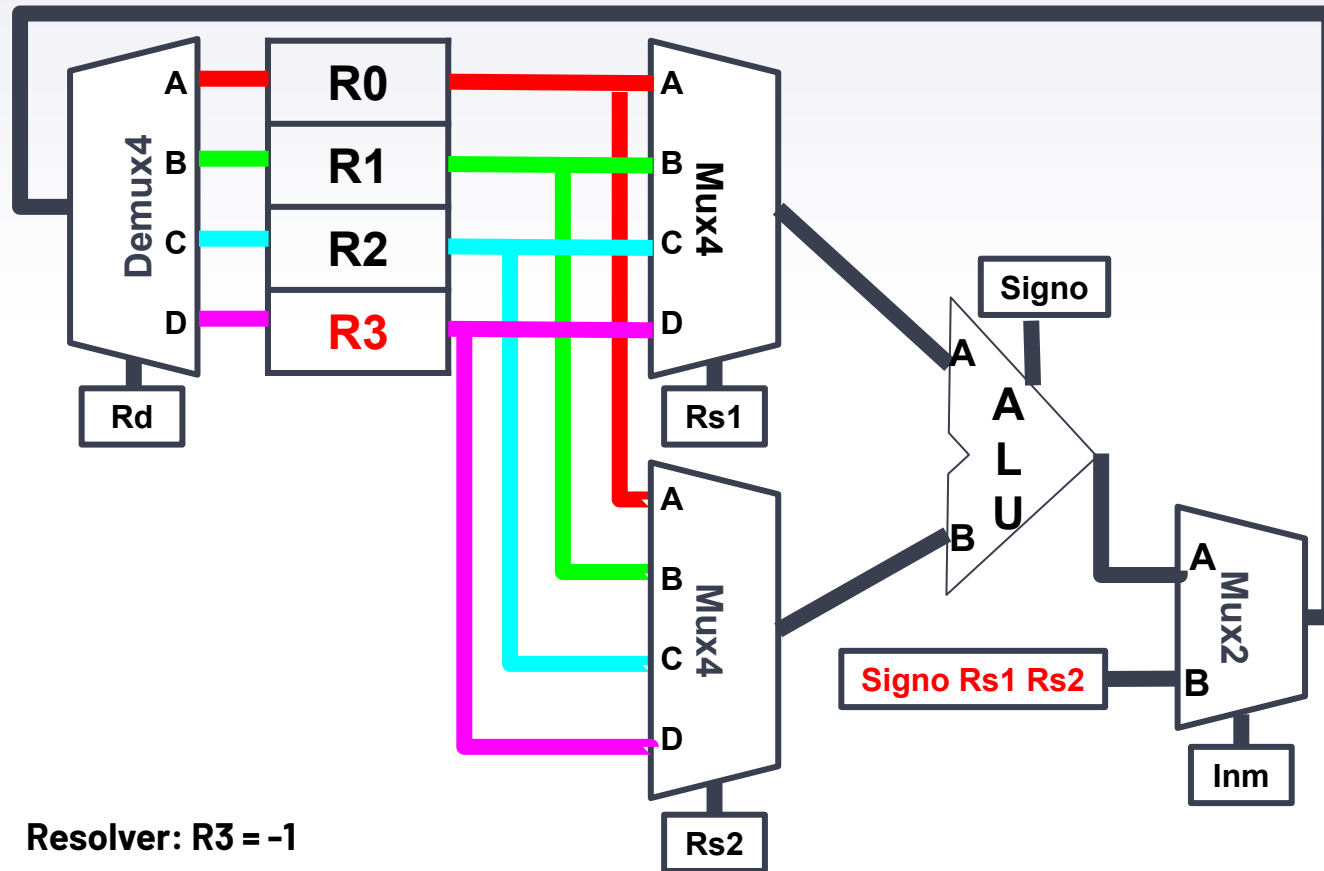
El bit **Signo** define la operación de la ALU

- **Signo=1** → $ALU = A - B$
- **Signo=0** → $ALU = A + B$

Los **Mux4** utilizan 2 bits de selección y eligen a su salida

- **00** = A
- **01** = B
- **10** = C
- **11** = D

El **Demux4** copia la entrada en A,B,C o D respectivamente según el valor de Rd.



RI=

Inm	Signo	Rd		Rs1		Rs2	
Ri7	Ri6	Ri5	Ri4	Ri3	Ri2	Ri1	Ri0

El bit **Inm** define que sale por Mux2

- **Inm=0** → Sale A (ALU)
- **Inm=1** → Sale un número inmediato de 5 bits compuesto por:
Signo Ri3 Ri2 Ri1 Ri0

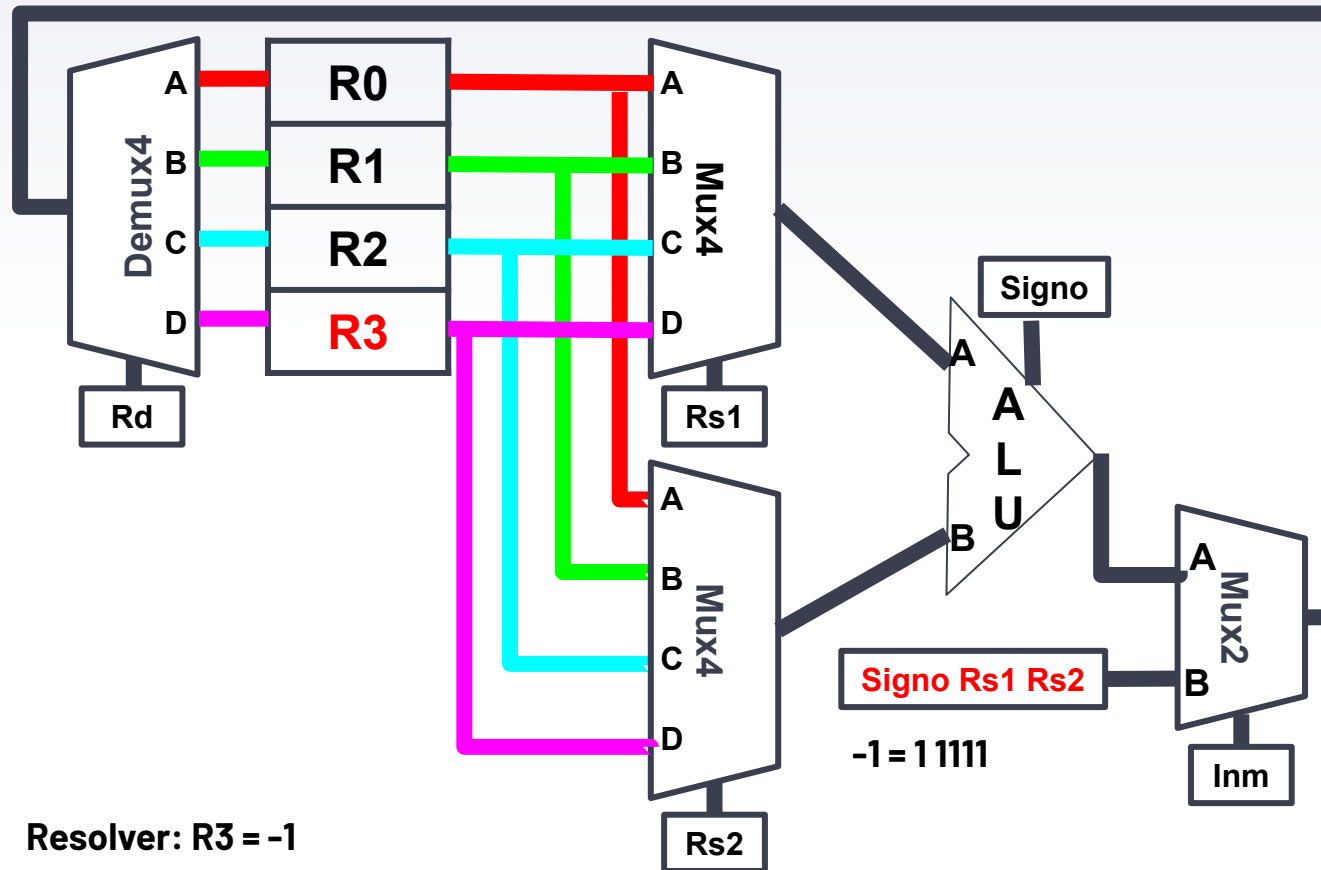
El bit **Signo** define la operación de la ALU

- **Signo=1** → $ALU = A - B$
- **Signo=0** → $ALU = A + B$

Los **Mux4** utilizan 2 bits de selección y eligen a su salida

- **00** = A
- **01** = B
- **10** = C
- **11** = D

El **Demux4** copia la entrada en A,B,C o D respectivamente según el valor de Rd.



Resolver: **R3 = -1**

RI=

Inm	Signo	Rd		Rs1		Rs2	
Ri7	1	Ri5	Ri4	1	1	1	1

El bit **Inm** define que sale por Mux2

- **Inm=0** → Sale A (ALU)
- **Inm=1** → Sale un número inmediato de 5 bits compuesto por:
Signo Ri3 Ri2 Ri1 Ri0

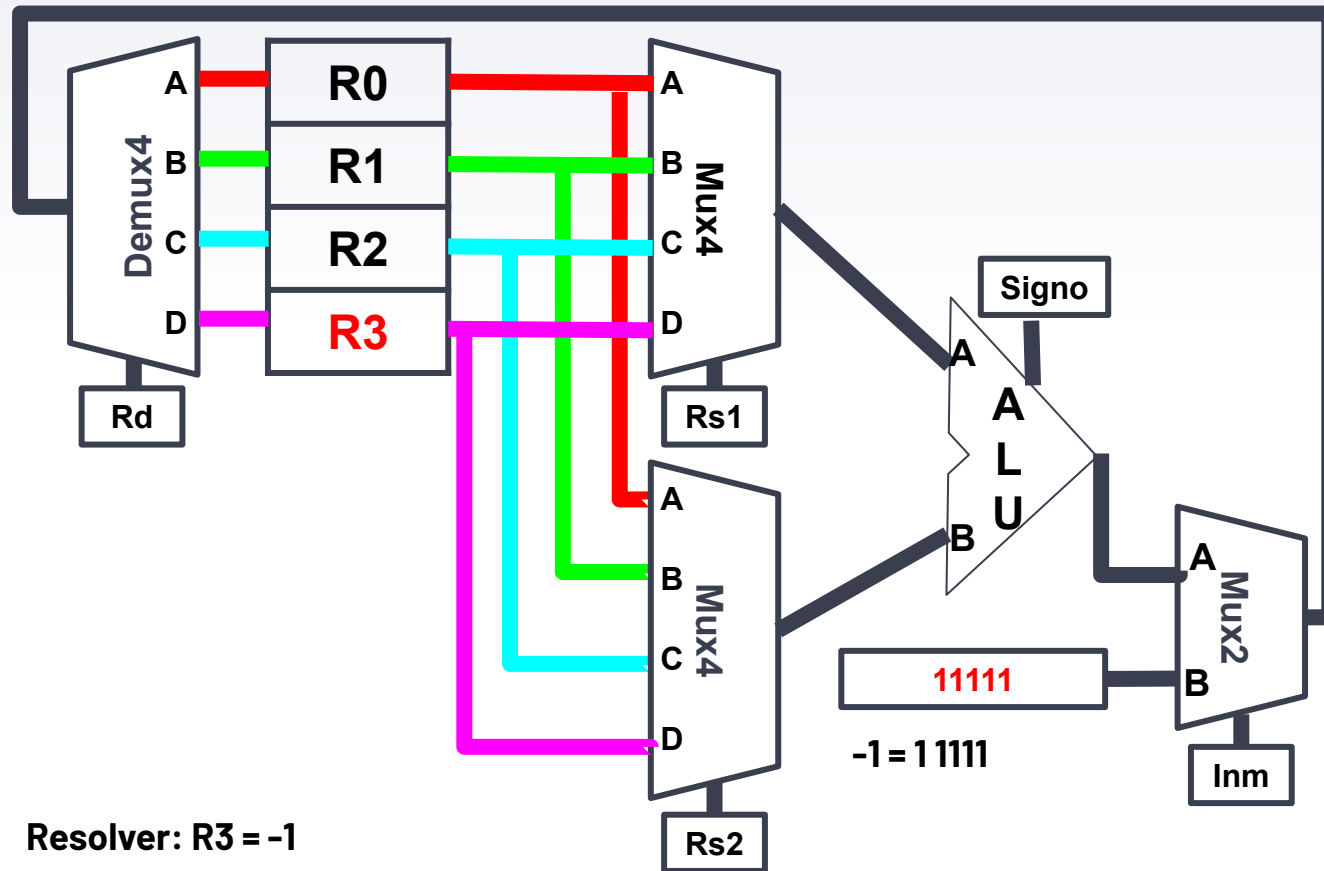
El bit **Signo** define la operación de la ALU

- **Signo=1** → $ALU = A - B$
- **Signo=0** → $ALU = A + B$

Los **Mux4** utilizan 2 bits de selección y eligen a su salida

- **00** = A
- **01** = B
- **10** = C
- **11** = D

El **Demux4** copia la entrada en A,B,C o D respectivamente según el valor de Rd.



Resolver: R3 = -1

RI=

Inm	Signo	Rd			Rs1		Rs2	
Ri7	1	1	1	1	1	1	1	1

El bit **Inm** define que sale por Mux2

- **Inm=0** → Sale A (ALU)
- **Inm=1** → Sale un número inmediato de 5 bits compuesto por:
Signo Ri3 Ri2 Ri1 Ri0

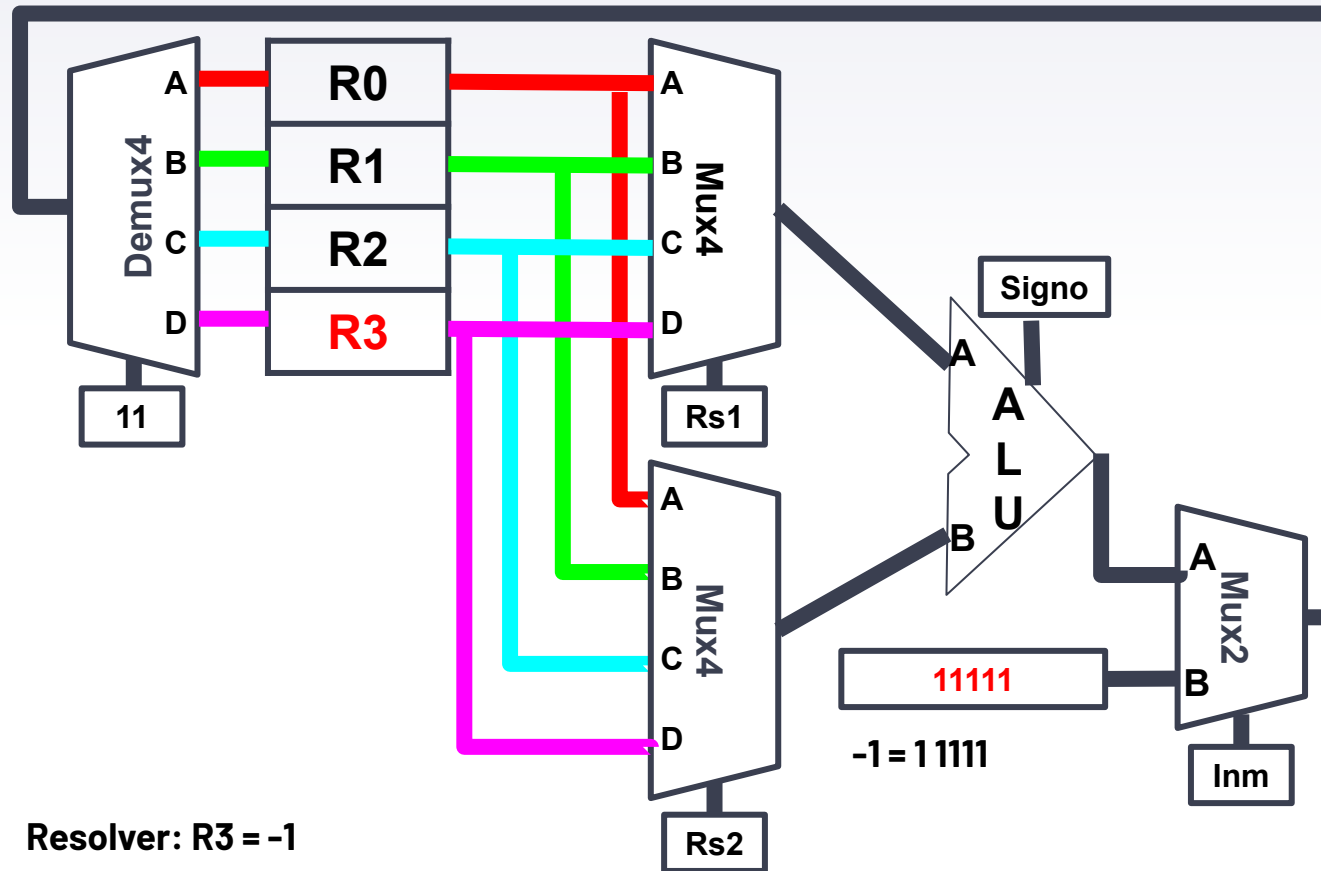
El bit **Signo** define la operación de la ALU

- **Signo=1** → $ALU = A - B$
- **Signo=0** → $ALU = A + B$

Los **Mux4** utilizan 2 bits de selección y eligen a su salida

- **00** = A
- **01** = B
- **10** = C
- **11** = D

El **Demux4** copia la entrada en A,B,C o D respectivamente según el valor de Rd.



RI=

Inm	Signo	Rd		Rs1		Rs2	
1	1	1	1	1	1	1	1

El bit **Inm** define que sale por Mux2

- **Inm=0** → Sale A (ALU)
- **Inm=1** → Sale un número inmediato de 5 bits compuesto por:
Signo Ri3 Ri2 Ri1 Ri0

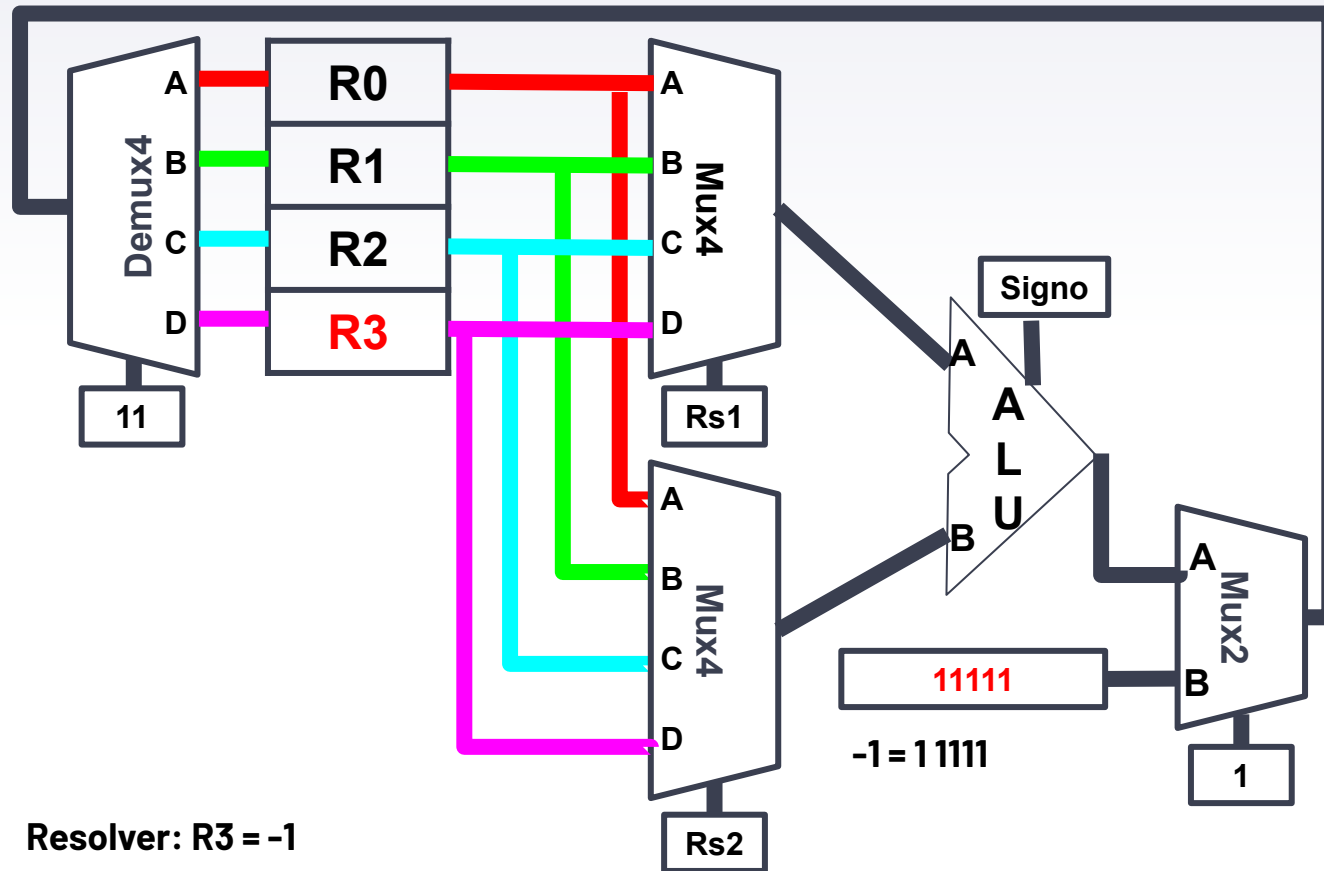
El bit **Signo** define la operación de la ALU

- **Signo=1** → $ALU = A - B$
- **Signo=0** → $ALU = A + B$

Los **Mux4** utilizan 2 bits de selección y eligen a su salida

- **00** = A
- **01** = B
- **10** = C
- **11** = D

El **Demux4** copia la entrada en A,B,C o D respectivamente según el valor de Rd.



Resolver: R3 = -1

RI=

Inm	Signo	Rd		Rs1		Rs2	
1	1	1	1	1	1	1	1

El bit **Inm** define que sale por Mux2

- **Inm=0** → Sale A (ALU)
- **Inm=1** → Sale un número inmediato de 5 bits compuesto por:
Signo Ri3 Ri2 Ri1 Ri0

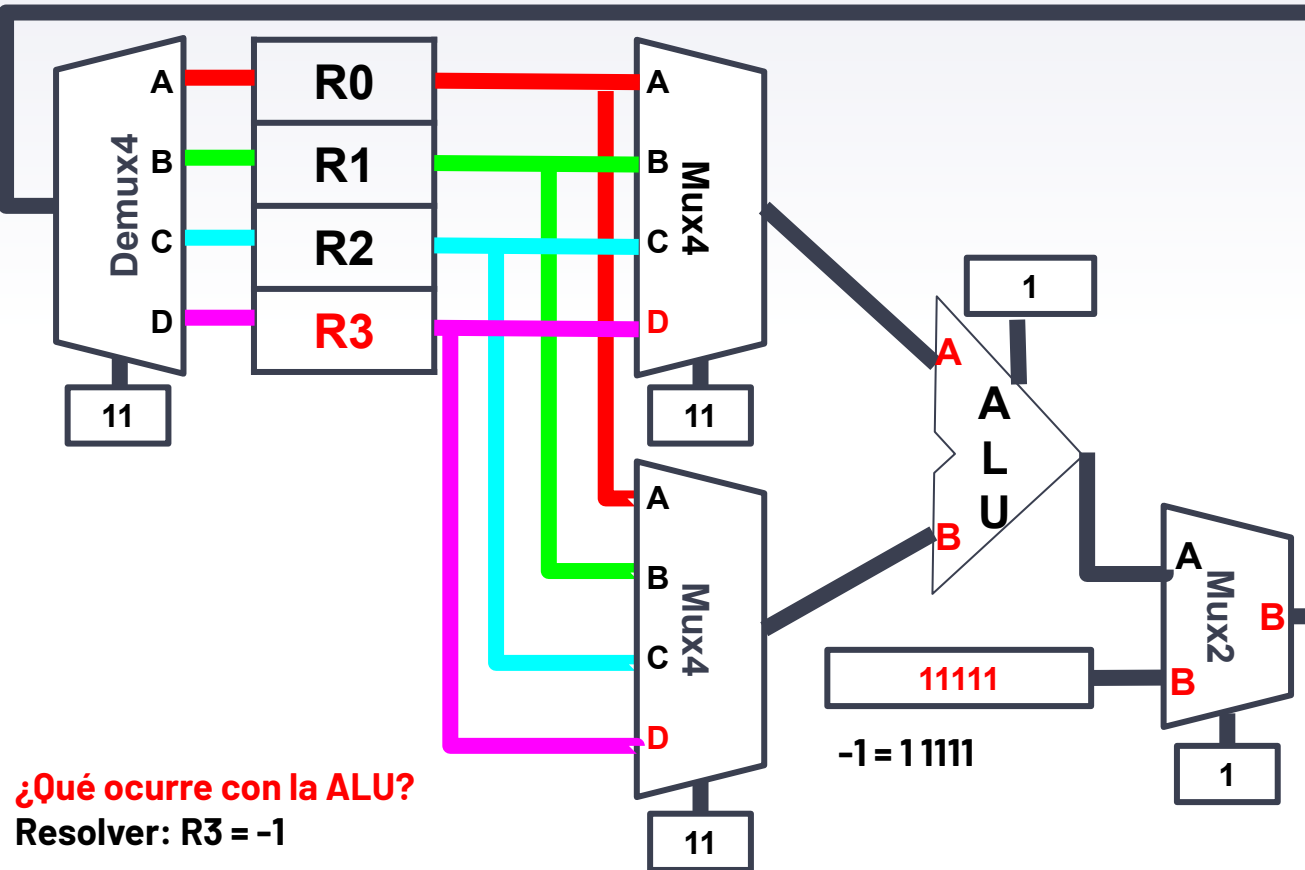
El bit **Signo** define la operación de la ALU

- **Signo=1** → $ALU = A - B$
- **Signo=0** → $ALU = A + B$

Los **Mux4** utilizan 2 bits de selección y eligen a su salida

- **00** = A
- **01** = B
- **10** = C
- **11** = D

El **Demux4** copia la entrada en A,B,C o D respectivamente según el valor de Rd.



¿Qué ocurre con la ALU?

Resolver: $R3 = -1$

0.0.3 Formato de instrucciones

- ¿Qué hacer?
- ¿Con qué datos?
- ¿Cómo llego a los datos?
- ¿Dónde almaceno el resultado?
- ¿Qué tengo que hacer a continuación?

OPCODE	Operando 1	Operando 2	Resultado	Dir. próxima instrucción
Sumar	Registro 3	Registro 5	Registro 0	0x12345678

Agrupando las instrucciones de forma consecutiva en la memoria, se añade un registro contador de programa que se incrementa luego de resolver una instrucción para saber la dirección de la siguiente. Con esto evitamos almacenar en cada instrucción la dirección de la siguiente.

- ¿Qué hacer?
- ¿Con qué datos?
- ¿Cómo llego a los datos?
- ¿Dónde almaceno el resultado?
- ~~¿Qué tengo que hacer a continuación?~~

OPCODE	Operando 1	Operando 2	Resultado
Sumar	Registro 3	Registro 5	Registro 0

12345678	inst1
12345679	inst2
1234567A	inst3
1234567B	inst4
1234567C	op1
1234567D	op2
...	
FFFFFFFFE	inst5
FFFFFFFFF	op3

Program Counter (PC)
0x12345679

Podemos reutilizar la referencia a uno de los operandos para almacenar el resultado. De esta forma evitamos tener una referencia extra para el resultado ya que el mismo siempre se almacena en la dirección del segundo operando.

- ¿Qué hacer?
- ¿Con qué datos?
- ¿Cómo llego a los datos?
- ~~¿Dónde almaceno el resultado?~~ $OP2 = OP1 + OP2$
- ~~¿Qué tengo que hacer a continuación?~~

OPCODE	Operando 1	Operando 2
Sumar	Registro 3	Registro 5

12345678	inst1
12345679	inst2
1234567A	inst3
1234567B	inst4
1234567C	op1
1234567D	op2
...	
FFFFFFFFE	inst5
FFFFFFFFF	op3

Program Counter (PC)
0x12345679

La referencia a los operandos depende del modo de direccionamiento utilizado. De alguna forma la CPU tiene que poder llegar a los datos (operandos)...

- ¿Qué hacer?
- ¿Con qué datos?
- ¿Cómo llego a los datos? Modos de direccionamiento
- ~~¿Dónde almaceno el resultado?~~ $OP2 = OP1 + OP2$
- ~~¿Qué tengo que hacer a continuación?~~

OPCODE	Operando 1	Operando 2
--------	------------	------------

Sumar

Registro 3

Registro 5

12345678	inst1
12345679	inst2
1234567A	inst3
1234567B	inst4
1234567C	op1
1234567D	op2
...	
FFFFFFFFE	inst5
FFFFFFFFF	op3

Program Counter (PC)

0x12345679

La referencia a los operandos depende del modo de direccionamiento utilizado. De alguna forma la CPU tiene que poder llegar a los datos (operandos)...

Datos...			¿Dónde están?	¿Cómo se indica dónde está?	¿Cómo se da la dirección del dato?		
					Dirección efectiva	→ Modo Absoluto	Absoluto directo
				Dirección del dato	Dirección de referencia + desplazamiento	→ Modo Relativo	Inmediato, indexado, relativo, paginado, segmentado, base y despl.
			En memoria	Dirección de la dirección del dato	En la instrucción	→ Modo indirecto	Absoluto indirecto
					En un registro	→ Modo registro indirecto	
	Con datos						
Instrucción			En registro	→ Modo registro			
	Sin datos			→ Modo implícito			

Modos teóricos
El concepto de cada uno puede implementarse individualmente o combinado con otro según el fabricante

Tipo R (registro)

En la arquitectura RISC-V, tenemos un formato de instrucción llamado R, donde cada instrucción posee 3 referencias a registro (**modo registro**). Una referencia indica el registro a utilizarse para almacenar el resultado (**rd**), y luego dos referencias a los operandos (**rs1** y **rs2**). Luego el código de operación indica que operación se realiza entre ambos operandos.

OPCODE	rs1	rs2	rd	
Sumar	R3	R5	R0	$R0 = R3 + R5$
Restar	R3	R5	R0	$R0 = R3 - R5$
Shift Left	R3	R5	R0	$R0 = R3 \ll R5$

En RISC-V: `add x0, x3, x5`

Tipo I (Inmediato/Constante)

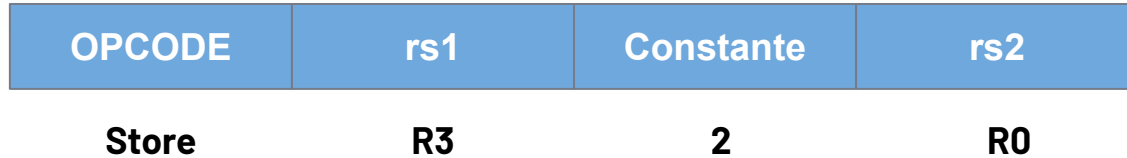
En la arquitectura RISC-V, tenemos un formato de instrucción llamado I, donde tenemos una referencia al registro donde se almacena el resultado (**rd**), una referencia al registro que contiene un operando (**rs1**) pero el segundo operando se codifica como constante en la instrucción. Su referencia es relativa al PC con offset 0 ya que se encuentra como constante en la instrucción misma.

OPCODE	rs1	Constante	rd	
Sumar	R3	2	R0	$R0 = R3 + 2$
Restar	R3	5	R0	$R0 = R3 - 5$
Shift Left	R3	4	R0	$R0 = R3 \ll 4$
Sumar	R3	-2	R0	$R0 = R3 + (-2)$

En RISC-V: `addi x0, x3, 2`

Tipo S (Base más desplazamiento)

En la arquitectura RISC-V se puede acceder a memoria construyendo la dirección tomando como base el contenido de un registro más una constante. En la operación Store, se toma el contenido de **rs2** (modo registro) y se lo almacena en la dirección construida sumando el contenido de **rs1** (modo registro indirecto) y la constante (modo inmediato).



Esta instrucción almacena el contenido de R0 en la dirección de memoria apuntada por el contenido de R3 más 2.

Tipo S (Base más desplazamiento)

En la arquitectura RISC-V se puede acceder a memoria construyendo la dirección tomando como base el contenido de un registro más una constante. En la operación Store, se toma el contenido de **rs2** (modo registro) y se lo almacena en la dirección construida sumando el contenido de **rs1** (modo registro indirecto) y la constante (modo inmediato).

OPCODE	rs1	Constante	rs2
--------	-----	-----------	-----

Store

R3

2

R0

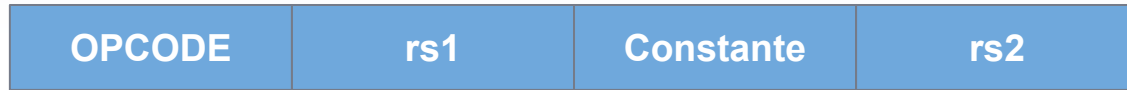
Esta instrucción almacena el contenido de R0 en la dirección de memoria apuntada por el contenido de R3 más 2.

Ejemplo: Suponiendo que R3 posee 0x12345678, y la constante es 2, entonces la dirección de destino es $(0x12345678 + 0x2 = 0x1234567A)$. Si el contenido de R0 fuese 0x80516502, entonces luego de ejecutar la instrucción el valor 0x80516502 se va a almacenar en la dirección de memoria 0x1234567A.

Si el valor de la constante hubiese sido -1, entonces la dirección de destino es 0x12345677.

Tipo S (Base más desplazamiento)

En la arquitectura RISC-V se puede acceder a memoria construyendo la dirección tomando como base el contenido de un registro más una constante. En la operación Store, se toma el contenido de **rs2** (modo registro) y se lo almacena en la dirección construida sumando el contenido de **rs1** (modo registro indirecto) y la constante (modo inmediato).



Store

R3

2

R0

Esta instrucción almacena el contenido de R0 en la dirección de memoria apuntada por el contenido de R3 más 2.

En RISC-V: `sw x0, 2(x3)`

Tipo J (Salto y retorno)

En la arquitectura RISC-V se puede sumar una constante al valor del PC. De esta forma cambiamos el flujo del programa (salto). El valor del PC antes de la suma se almacena en **rd** en caso que en el futuro se necesite volver.

OPCODE	rd	Constante
--------	----	-----------

En RISC-V: jal x1, 31

JAL

R1

31

Ejemplo: Si el PC se encuentra en 0x100 al momento de ejecutar JAL R1,0x1F , entonces se suma $0x100+0x1F$ haciendo que la próxima instrucción se lea de 0x11F.

El registro R1 va a almacenar la dirección de la instrucción siguiente a JAL.

En el caso de querer retornar el flujo del programa se puede utilizar la instrucción JALR, la cual utiliza rd y la constante, pero incluye también rs1, siendo $rs1+constante$ el valor que toma el PC. Por ende ejecutar:

JALR r1,r1,0 retorna a la instrucción siguiente a JAL.

En RISC-V: jalr x1, x1 , 0

A lo largo de la historia han existido muchas formas distintas de indicar en una instrucción como acceder a los operandos. La idea de los distintos modos de direccionamiento es hacer que la ejecución de instrucciones sea más eficiente.

OPCODE	Operando 1	Operando 2	Resultado	Dir. próxima instrucción
Sumar	Dir. 0x1234	Dir. 0x1235	Dir. 0x1236	0x12345678

Vemos que si tenemos 4 referencias a memoria de 32 bits, utilizamos el PC para evitar almacenar la última referencia.

OPCODE	Operando 1	Operando 2	Resultado	PC
Sumar	Dir. 0x1234	Dir. 0x1235	Dir. 0x1236	0x12345678

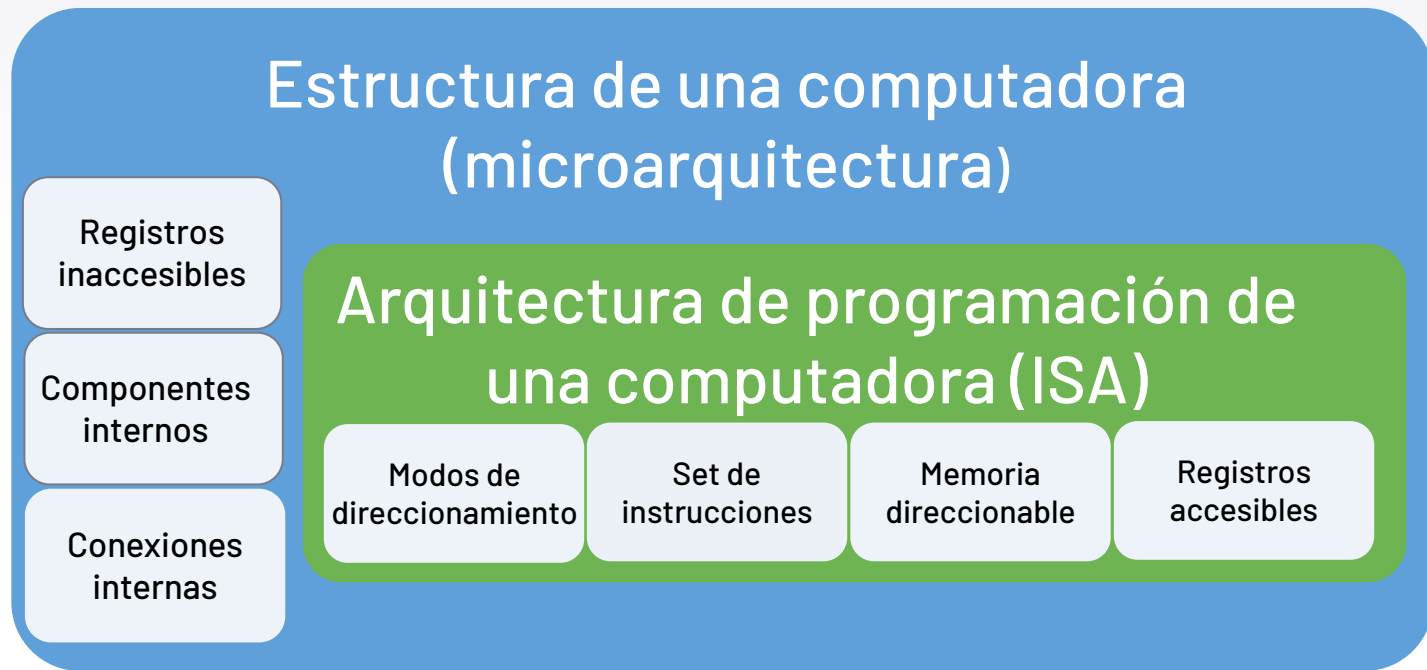
Utilizando un registro base, solo almacenamos los offsets en cada referencia.

OPCODE	Operando 1	Operando 2	Resultado	BASE	PC
Sumar	0	1	2	0x00001234	0x12345678

0.1 Arquitecturas

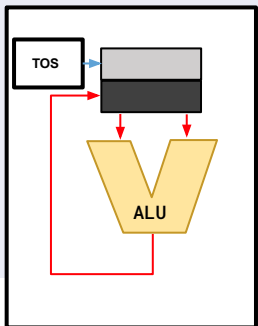
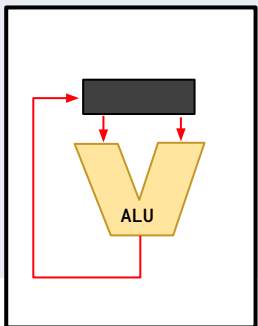
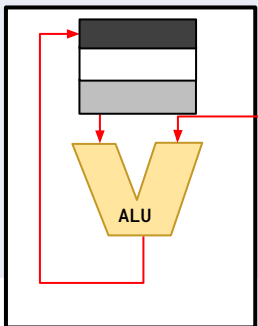
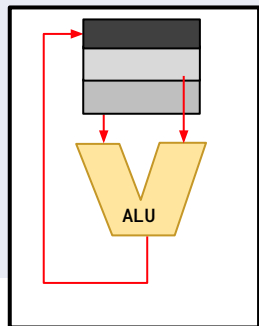
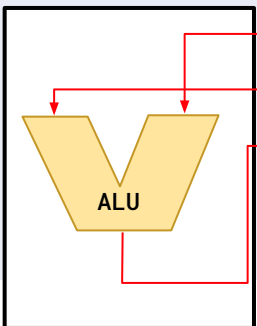
Mil Intentos y un invento

0.1.0 Estructura / Arquitectura

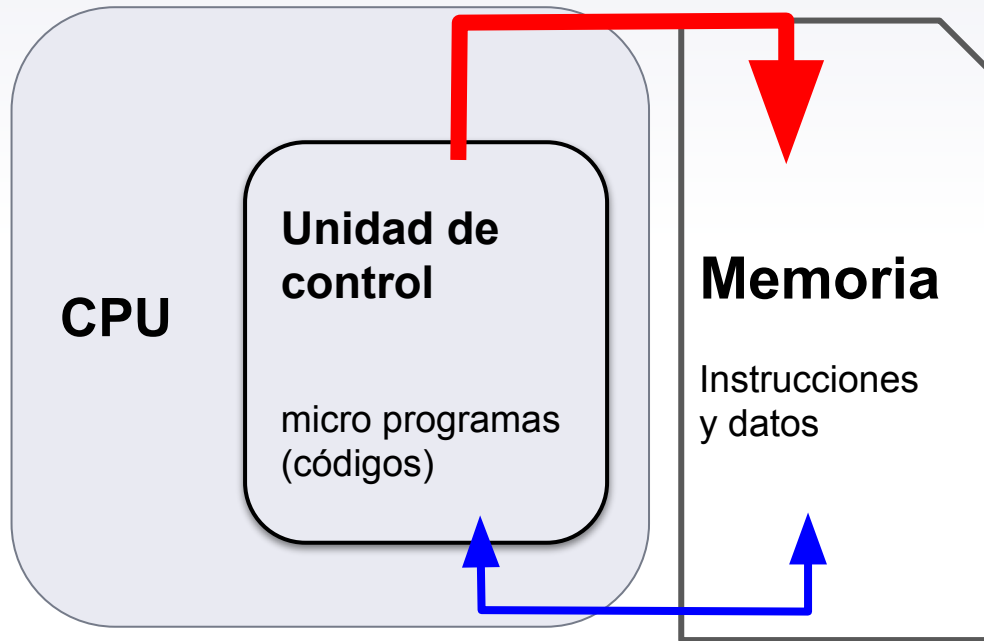


A la ISA podríamos agregar: Semántica de ejecución y manejo de E/S.

0.1.1 Tipos de arquitectura

Stack	Acumulador	Registro-memoria	Registro-registro	Memoria-memoria
push A push B add pop C	load A add B store C	load R1,A add R1,B store C,R1	load R1,A load R2,B add R3,R1,R2 store C,R3	add C,A,B
				

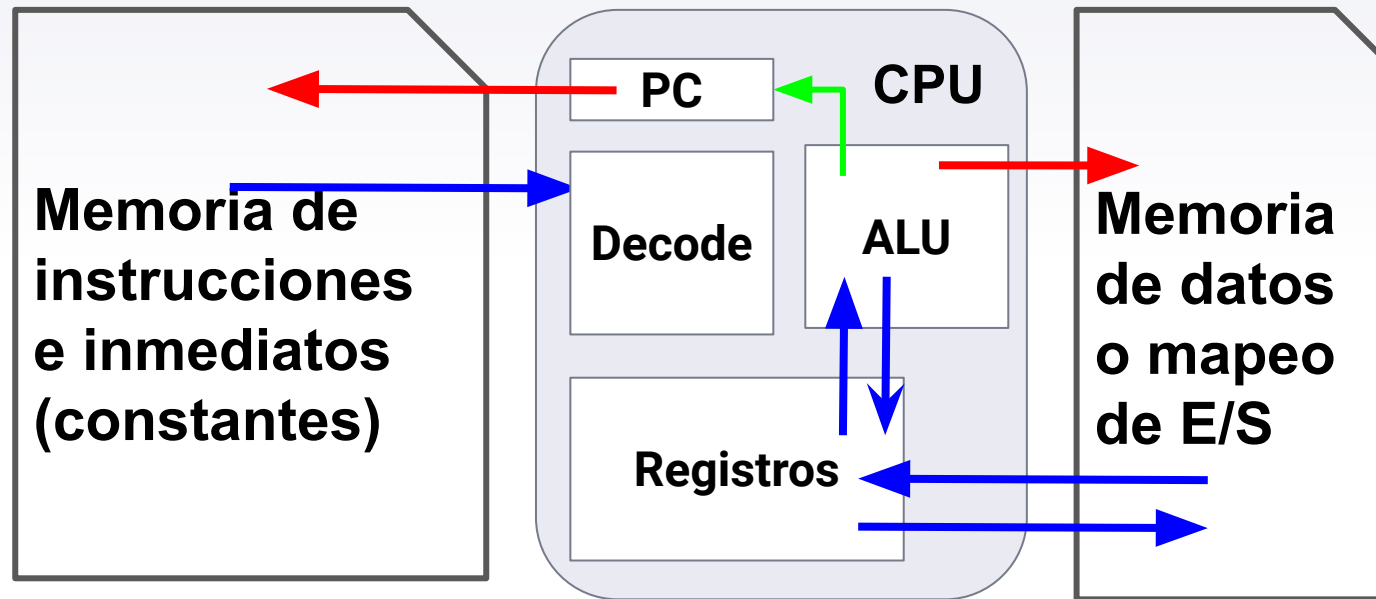
0.1.2 Tipo Von Neumann



Un **único** “camino” (path) para datos e instrucciones. A veces se leen instrucciones, a veces se acceden a datos. Sin embargo esto crea un cuello de botella. Dado que el bus de datos es bidireccional, deben existir tiempos de espera para los cambios de dirección, y mientras el mismo se utiliza por ejemplo para leer una instrucción no puede utilizarse para almacenar un resultado.

Los microprogramas simplifican la implementación de las instrucciones permitiendo resolver operaciones complejas con una instrucción a medida. Aunque esto requiere que el compilador implemente todas las instrucciones.

0.1.2 Tipo Harvard



Las instrucciones y los datos (variables) se acceden por caminos distintos. No existen buses bidireccionales por ende los datos fluyen en un único sentido. Esta arquitectura permite por ejemplo leer una instrucción apuntada por el PC mientras que en el mismo instante la ALU apunta a la memoria de datos para escribir un valor proveniente de un registro.

0.1.3 Espacios de direccionamiento



La memoria accesible por la CPU requiere una forma de distinguir entre distintas “palabras” de memoria. Es decir, distintas posiciones en la memoria que podemos utilizar para almacenar instrucciones o datos. La cantidad de líneas en el bus de direcciones representan este espacio. Todas las combinaciones posibles de unos y ceros que pueden representarse en el bus de direcciones equivalen a la cantidad de “direcciones” de memoria disponibles.

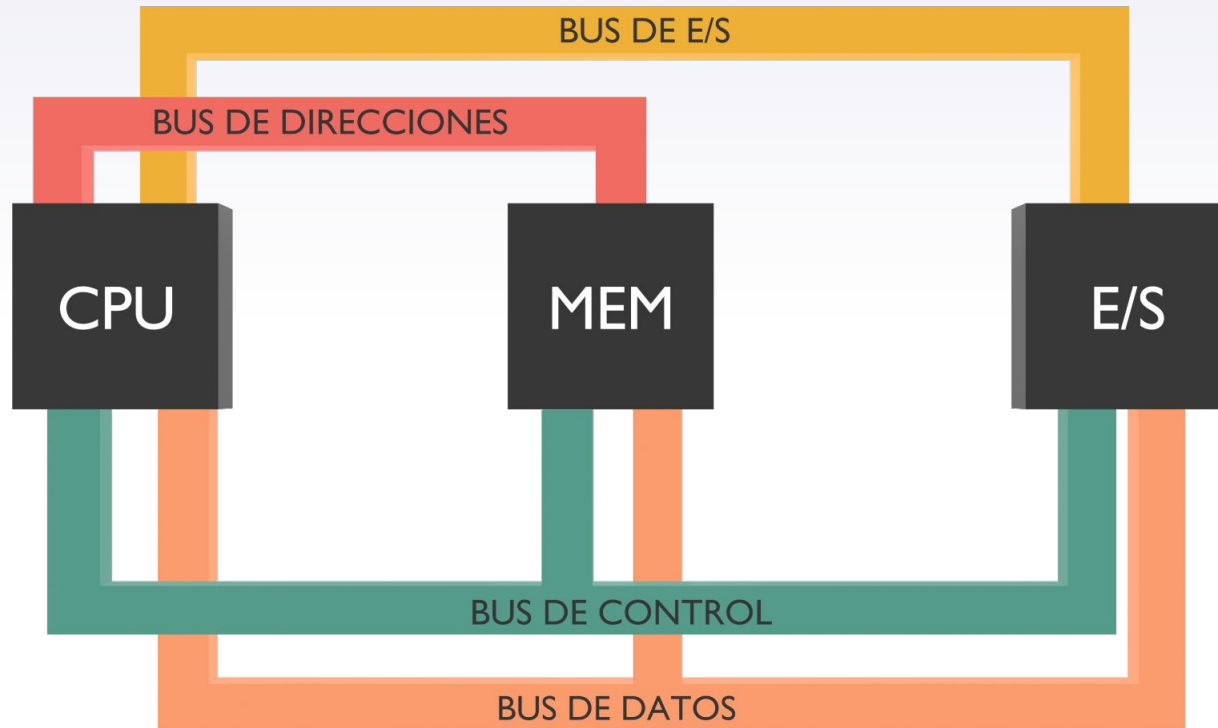
Esto no quiere decir que todas esas direcciones posean efectivamente memoria disponible.

Arquitecturas de 4 buses

Dado que desde la CPU necesitamos acceder a **memoria** para almacenar datos (variables) pero también tenemos que acceder a dispositivos de **entrada/salida** para interactuar con el mundo exterior, la arquitectura de 4 buses **separa** en un grupo las “direcciones” de memoria y las “direcciones” de entrada/salida.

Para cada grupo el programador utiliza instrucciones específicas. Por ejemplo la dirección 0x1234 puede referirse a la dirección de memoria 0x1234 o al registro de entrada/salida 0x1234 (que posee por ejemplo un LED). Para acceder a memoria el programador debe utilizar una instrucción específica de memoria con la dirección 0x1234. Para acceder a E/S el programador debe utilizar una instrucción específica de E/S con la dirección 0x1234.

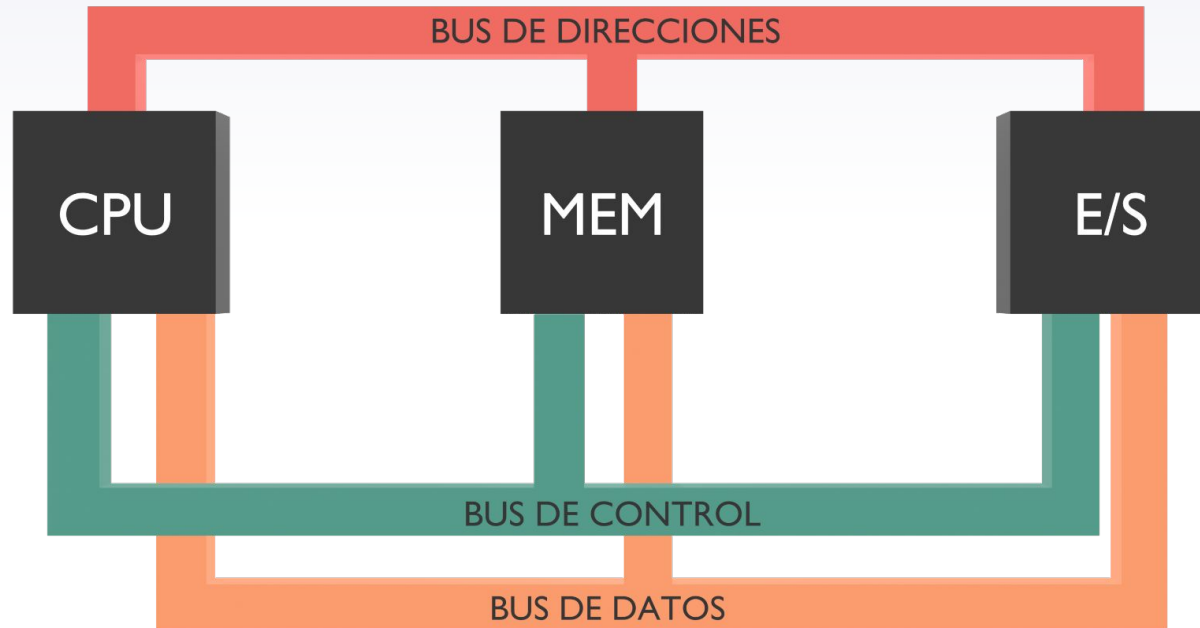
Arquitecturas de 4 buses



Arquitecturas de 3 buses

Muchos diseñadores de computadora eligen **compartir** el espacio de direccionamiento entre memoria y entrada/salida. De esta manera, no necesitan instrucciones específicas, ya que el espacio es el mismo. Como contraparte, la cantidad de memoria que puede ser implementada **no puede cubrir el espacio total** de direcciones ya que debe reservar algunas posiciones para dispositivos de entrada/salida.

Arquitecturas de 3 buses



▶ 0.1.4 CISC vs RISC

Complex Instruction Set Computer

Las instrucciones suelen resolver operaciones más complejas. Ej: en x86 la instrucción *CMPS* toma dos referencias a memoria (punteros a char). La instrucción lee el contenido de cada referencia y lo compara. Si son iguales incrementa los punteros y repite. Cuando encuentra que son distintos o que ambos contenidos son 0, entonces termina. Básicamente realiza un *strcmp()* en una única instrucción.

CISC requiere que los compiladores estén al tanto de todo el set e implementen las mismas. Con arquitecturas microprogramadas es relativamente sencillo agregar instrucciones complejas.



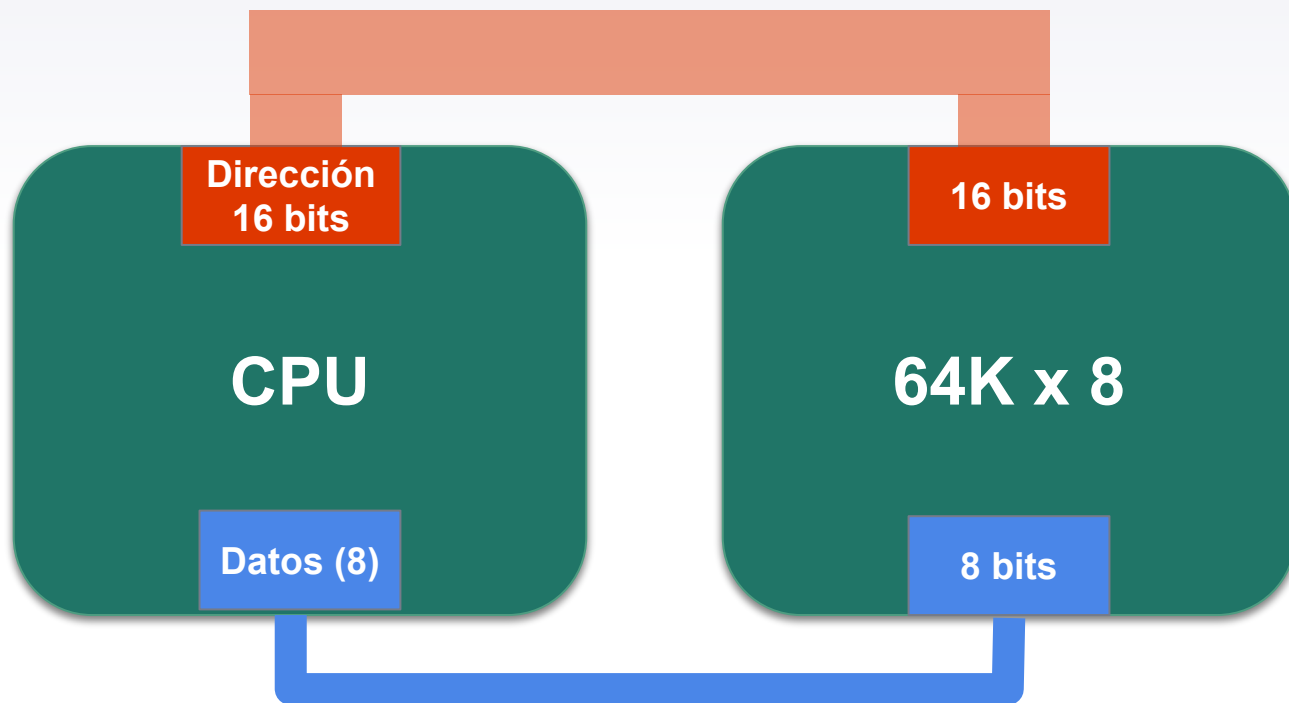
▶ 0.1.4 CISC vs RISC

Reduced Instruction Set Computer

En los 80's la segmentación de arquitecturas era muy grande. Los compiladores generalmente ignoraban instrucciones muy específicas. Teniendo esto en cuenta se comienza a diseñar set (conjuntos) de instrucciones con solo las operaciones **más elementales**. Se busca lograr que existan pocas instrucciones pero que la resolución de las mismas sea muy **eficiente**. Si bien resolver un problema complejo ahora va a utilizar más instrucciones las mismas son lo suficientemente eficientes como para que no exista penalidades de tiempo.



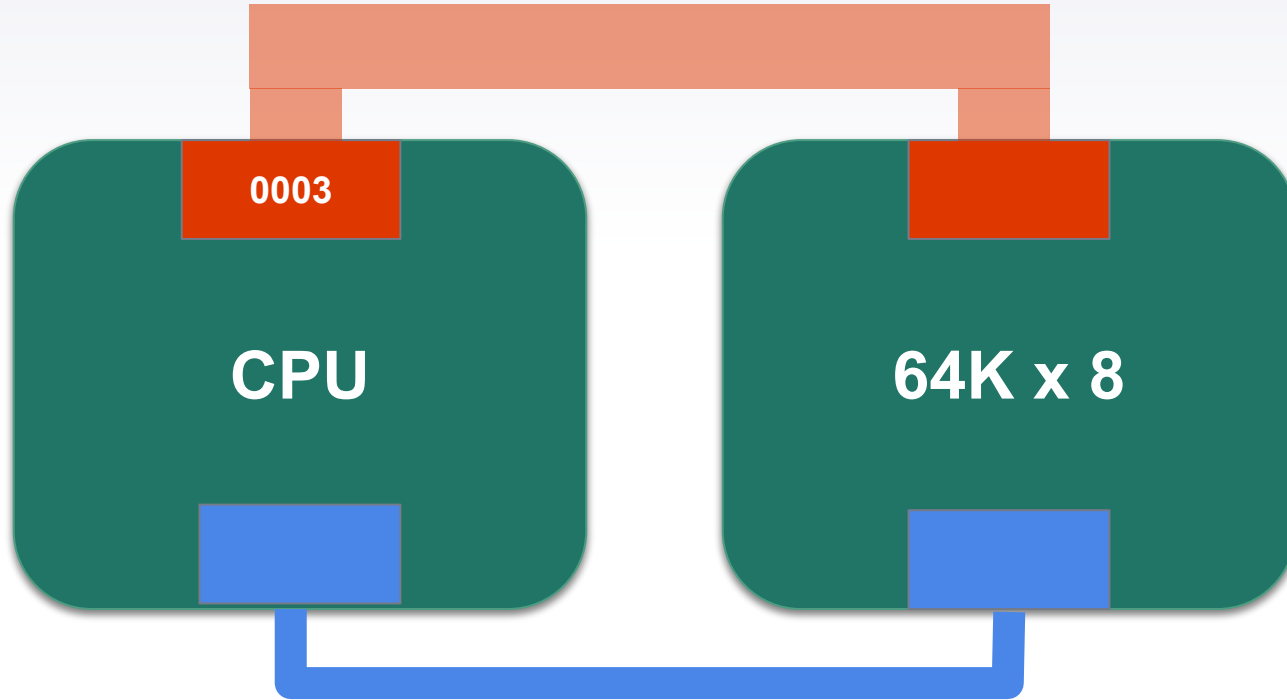
0.1.5 Byte alignment



Cada palabra de memoria
almacena 8 bits.
Cada dirección de memoria
apunta a 8 bits.

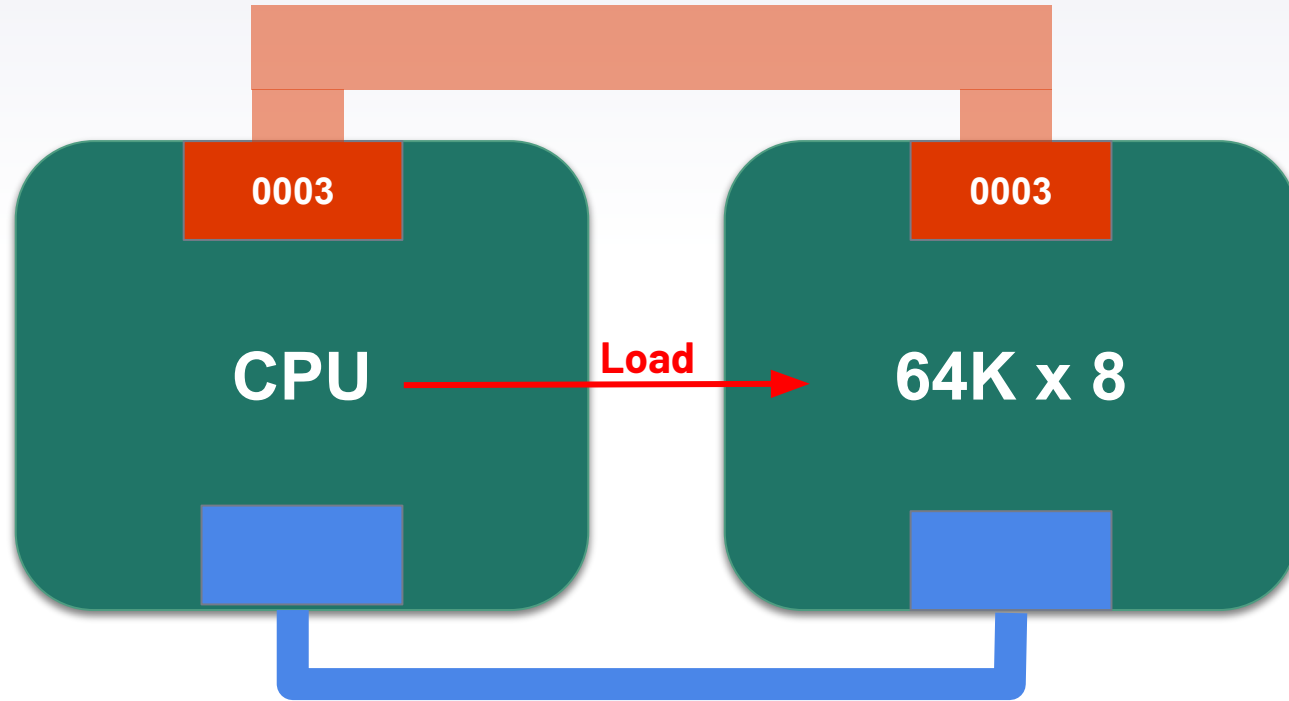
0000	
0001	
0002	
0003	
0004	
0005	
...	
FFFE	
FFFF	

**Cada palabra de memoria
almacena 8 bits.
Cada dirección de memoria
apunta a 8 bits.**



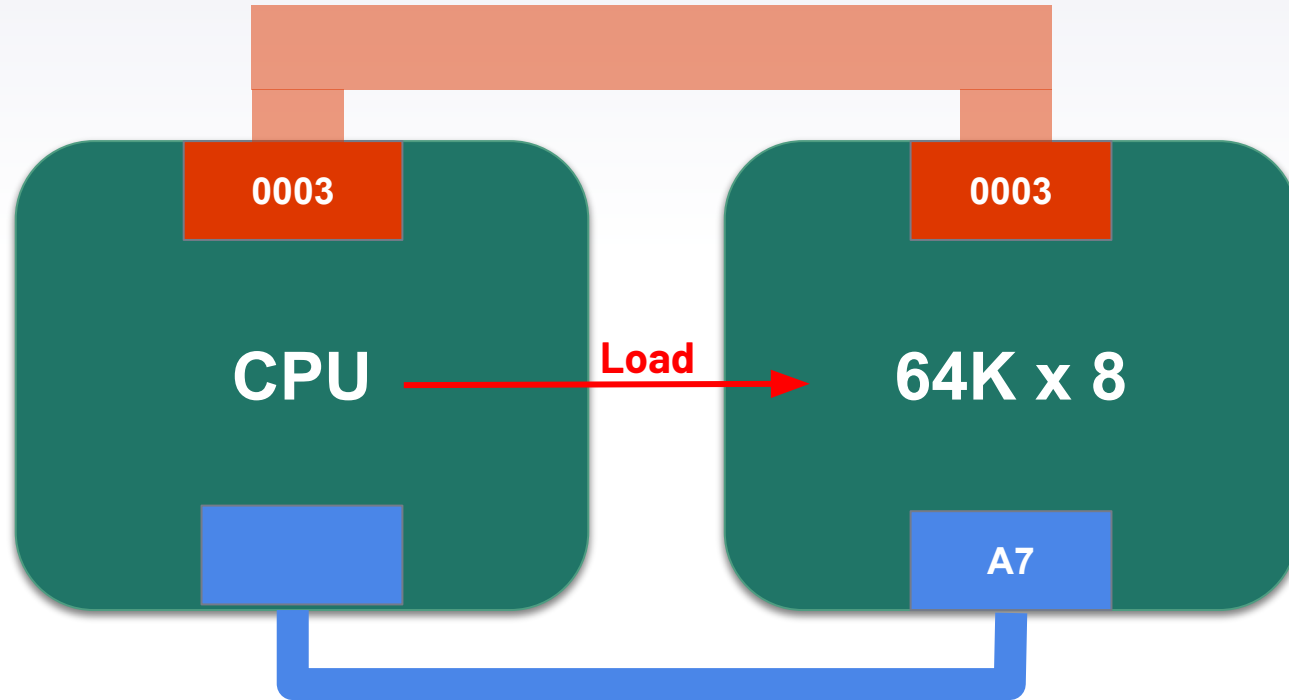
0000	
0001	
0002	
0003	A7
0004	
0005	
...	
FFFE	
FFFF	

**Cada palabra de memoria
almacena 8 bits.
Cada dirección de memoria
apunta a 8 bits.**



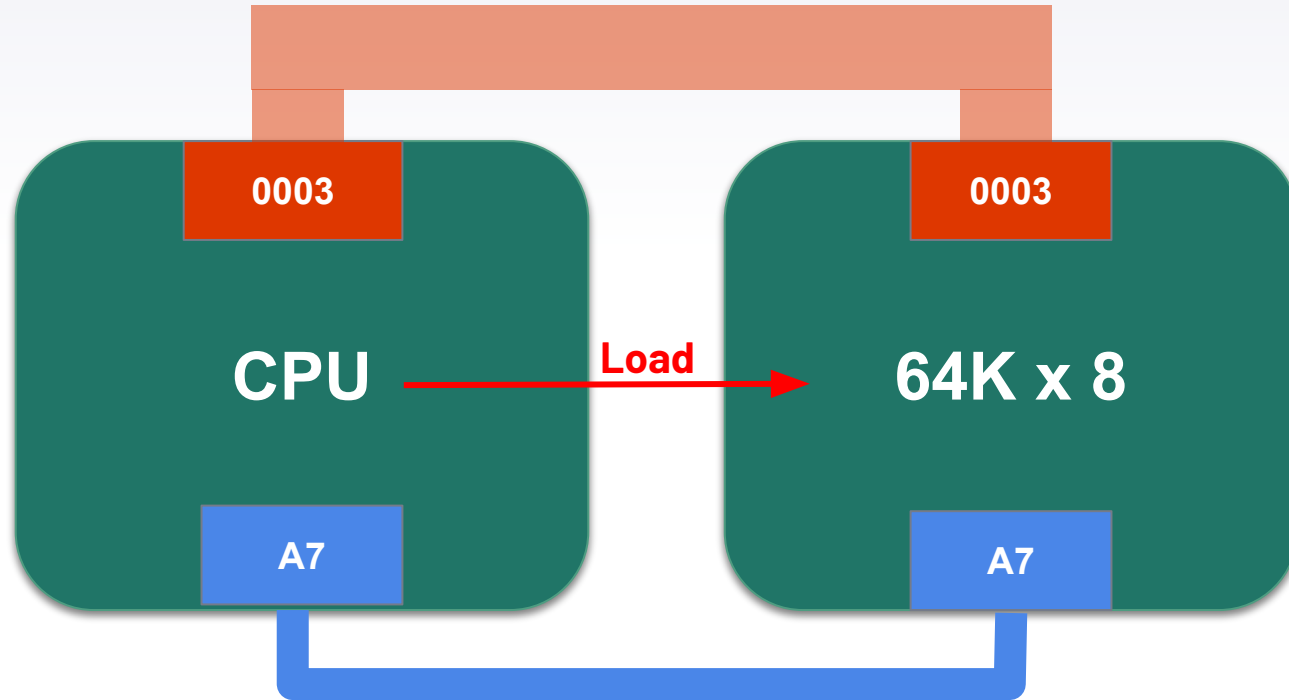
0000	
0001	
0002	
0003	A7
0004	
0005	
...	
FFFE	
FFFF	

**Cada palabra de memoria
almacena 8 bits.
Cada dirección de memoria
apunta a 8 bits.**



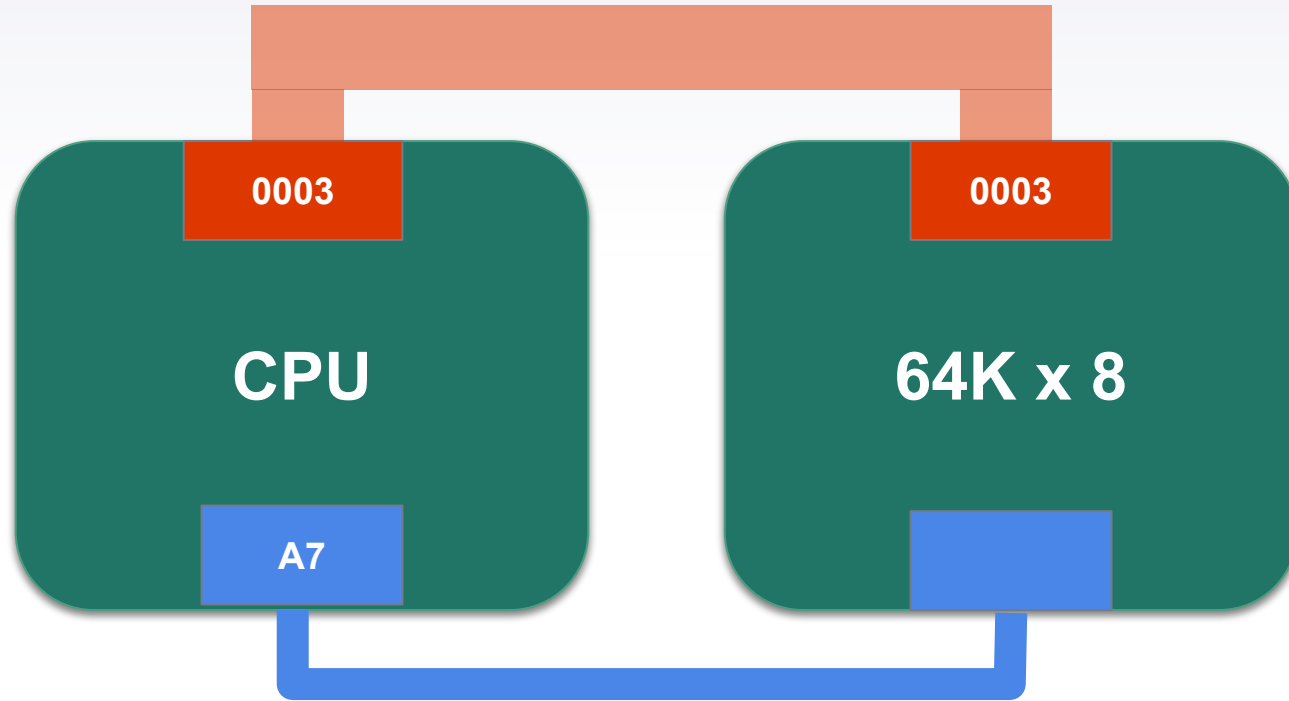
0000	
0001	
0002	
0003	A7
0004	
0005	
...	
FFFE	
FFFF	

**Cada palabra de memoria
almacena 8 bits.
Cada dirección de memoria
apunta a 8 bits.**



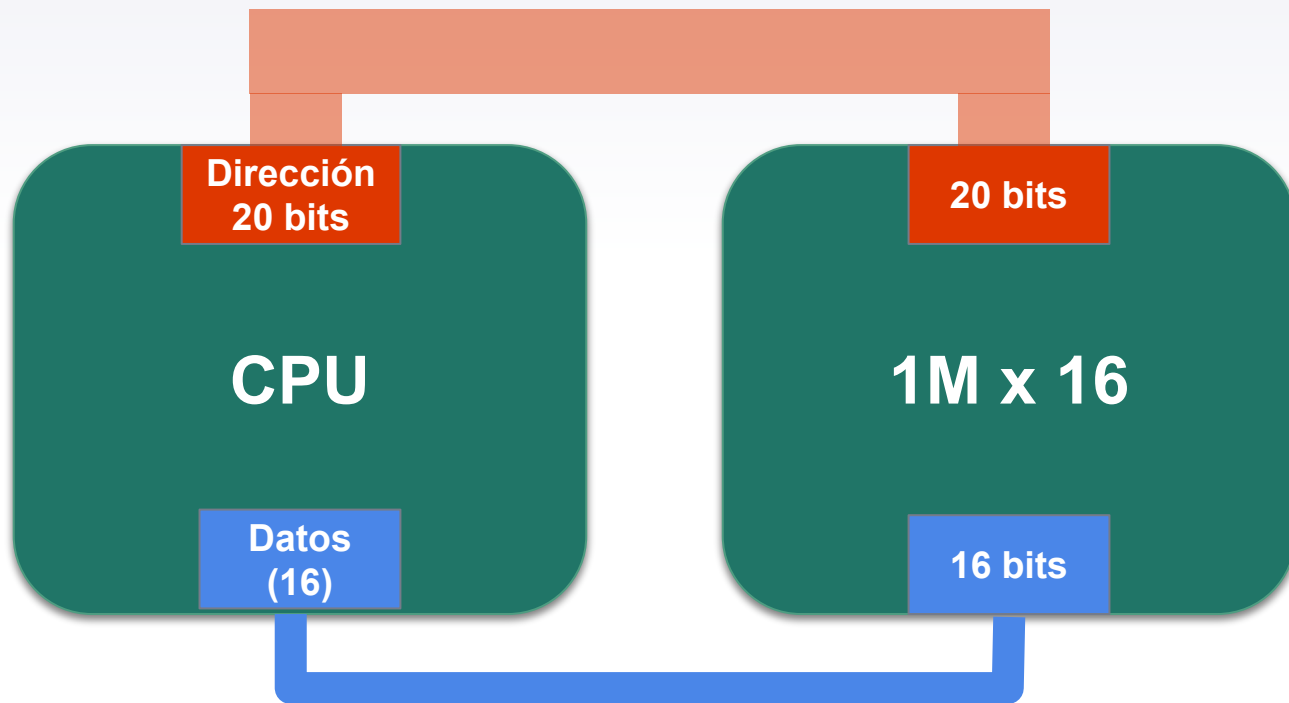
0000	
0001	
0002	
0003	A7
0004	
0005	
...	
FFFE	
FFFF	

**Cada palabra de memoria
almacena 8 bits.
Cada dirección de memoria
apunta a 8 bits.**



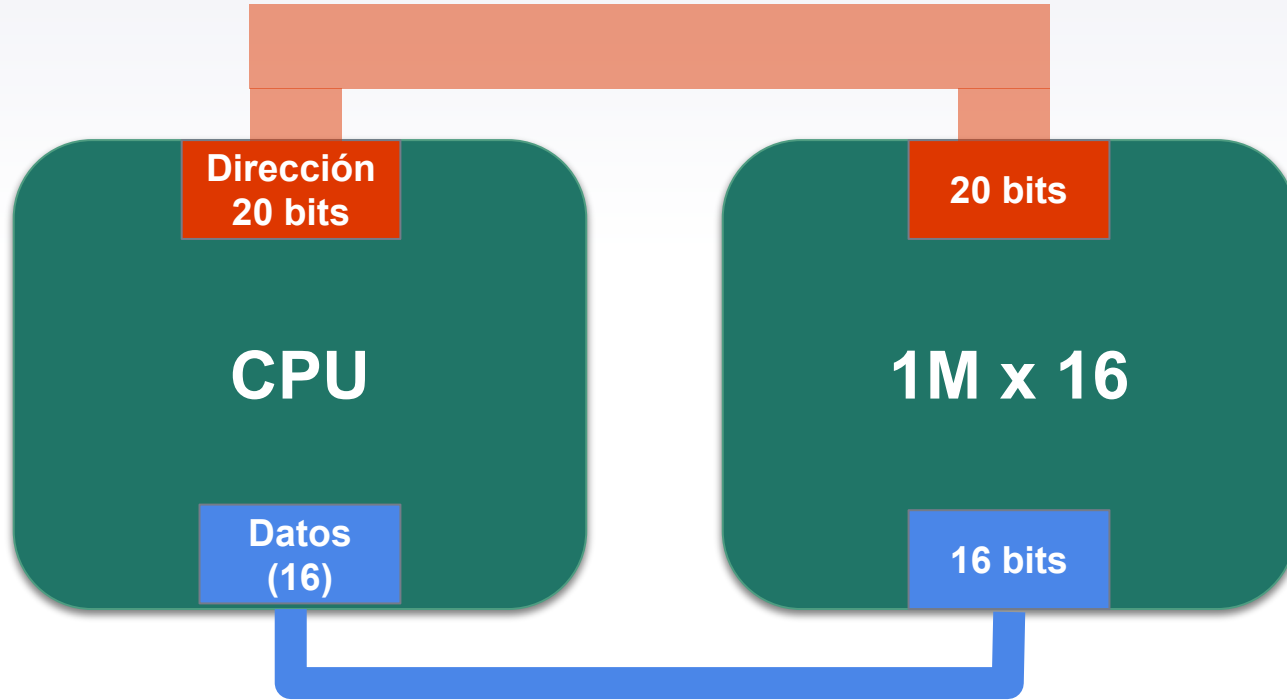
0000	
0001	
0002	
0003	A7
0004	
0005	
...	
FFFE	
FFFF	

Aparecen las computadoras de 16 bits de tamaño de palabra (con 20 bits de direcciones para acceder a 1M posiciones de memoria).



00000	
00001	
00002	
00003	
00004	
00005	
...	
FFFFE	
FFFFF	

Cualquiera pensaría que en cada posición de memoria se pueden almacenar 16 bits



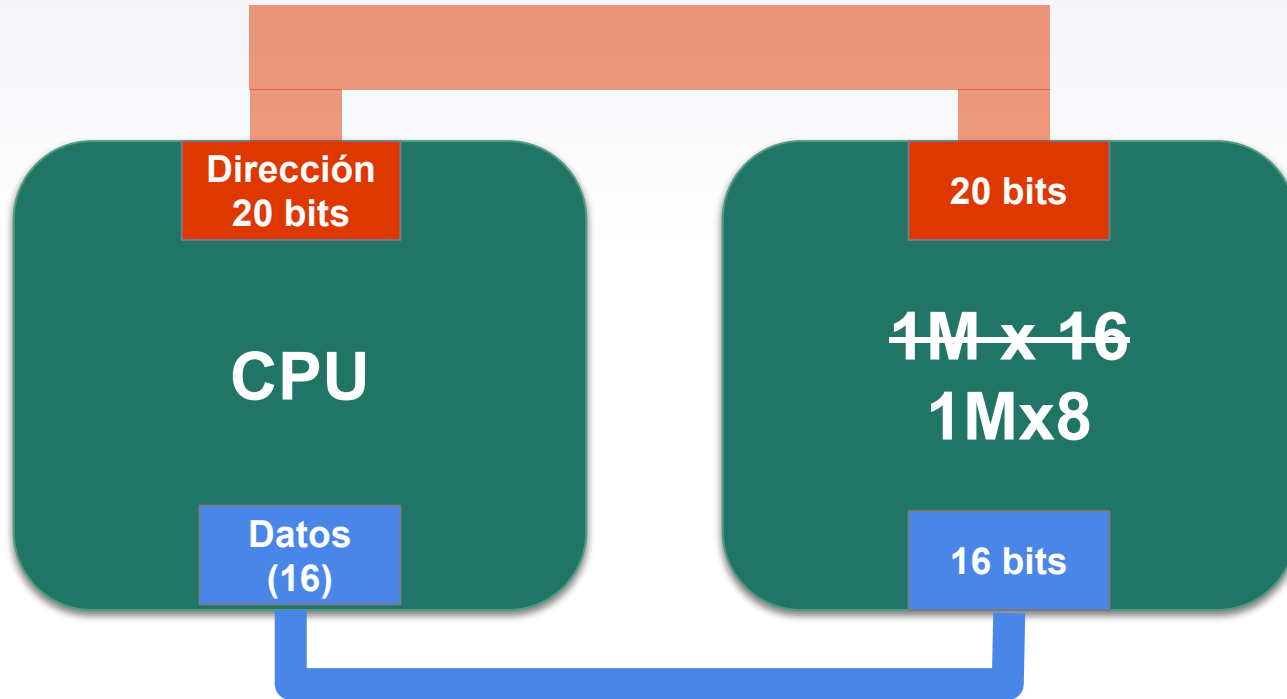
00000	
00001	
00002	650C
00003	
00004	
00005	
...	
FFFFE	
FFFFF	

Cualquiera pensaría que en cada posición de memoria se pueden almacenar 16 bits. **Pero no!.**

Lo que hicieron los fabricantes fue crear dos instrucciones:

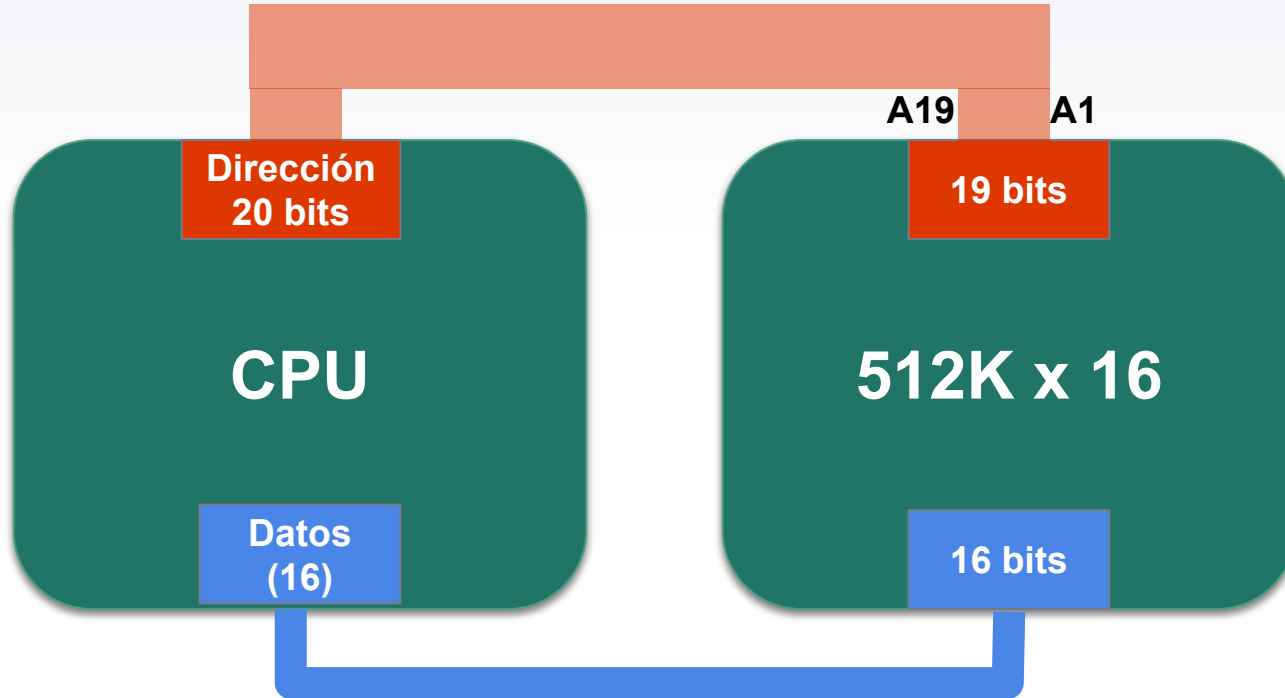
- Leer Palabra (lw = Load Word)
- Leer Byte (lb = Load Byte)

Y **cada dirección de memoria se refiere a un único byte.**



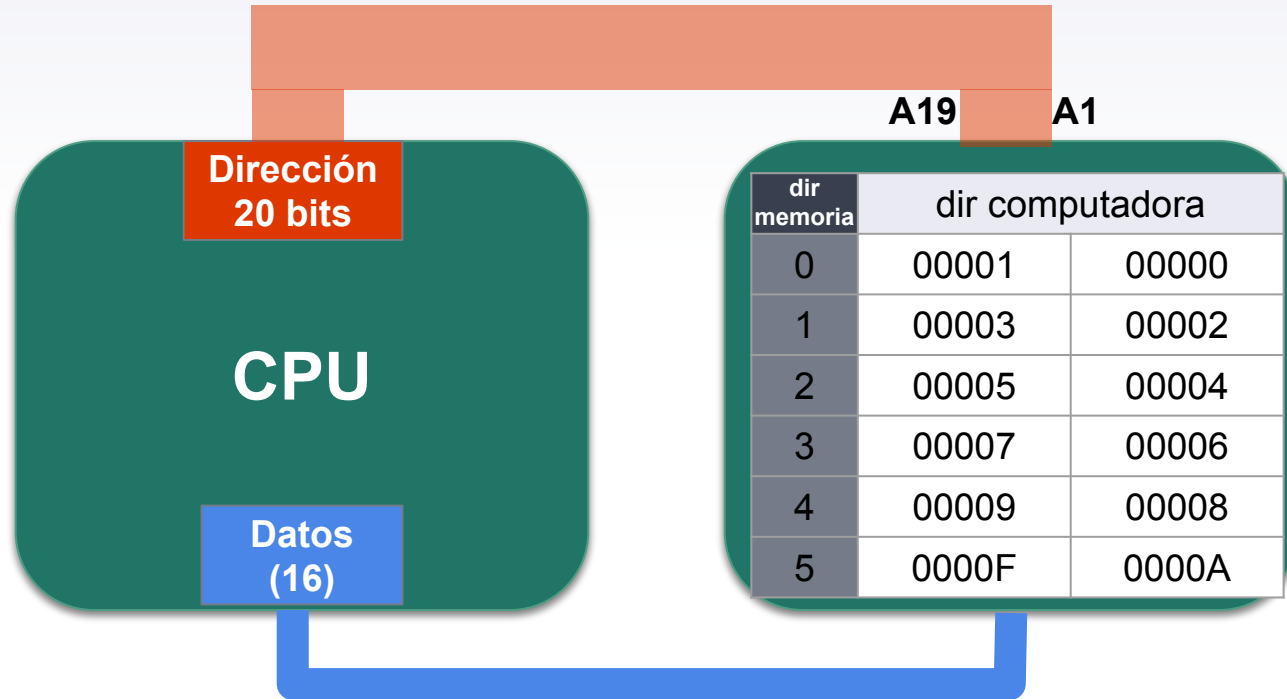
00000	
00001	
00002	6500
00003	
00004	
00005	
...	
FFFFE	
FFFFF	

Los fabricantes de memoria ponen en cada posición de memoria 16 bits. Pero la CPU direcciona "al byte", o sea, cuando escribe una dirección en el bus de direcciones está apuntando a un byte, NO a una palabra de 16 bits. Esos 20 bits apuntan a 1M x8. En capacidad es lo mismo que 512K x 16. Solo 19 de los 20 cables del bus de direcciones se conectan a la memoria.

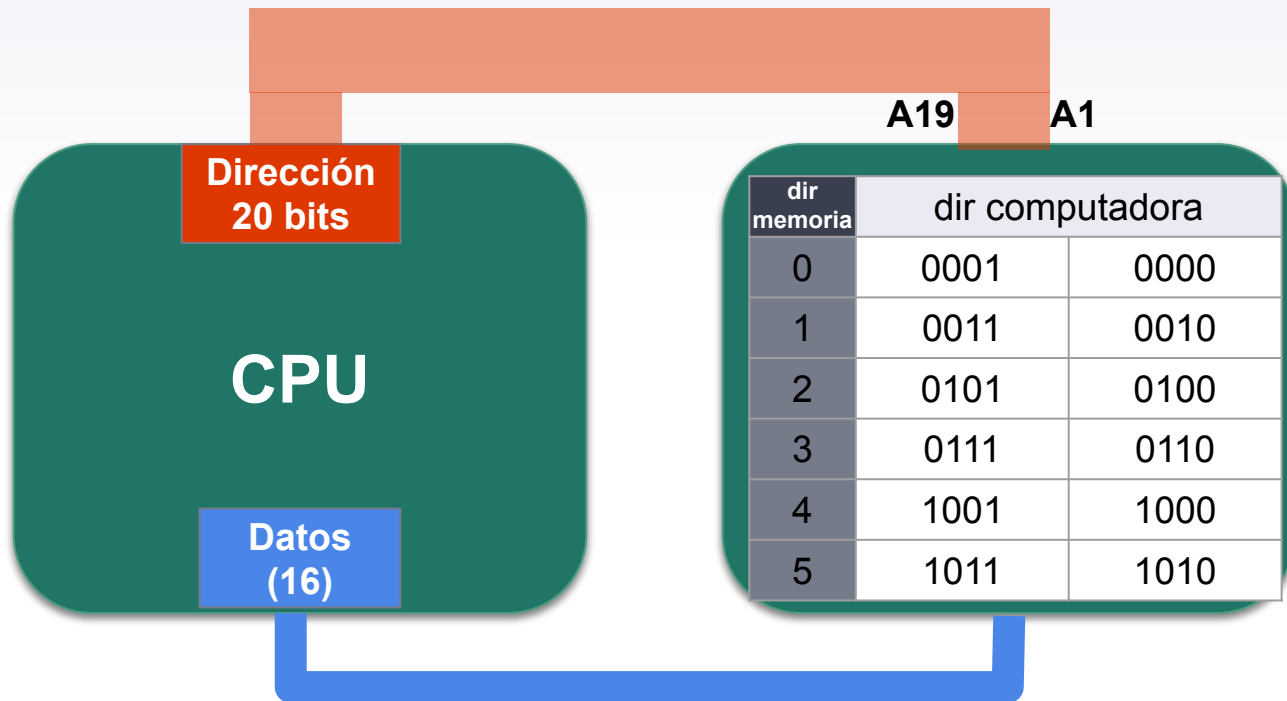


00000	
00001	
00002	6500
00003	
00004	
00005	
...	
7FFFE	
7FFFF	

Ahora cada dirección de memoria real contiene 16 bits, pero cada byte es apuntado por una dirección distinta de la CPU.



Ahora cada dirección de memoria real contiene 16 bits, pero cada byte es apuntado por una dirección distinta de la CPU. Escribiendo las direcciones en binario, vemos que el bit menos significativo indica si es el byte menos significativo o más significativo de una palabra de memoria.



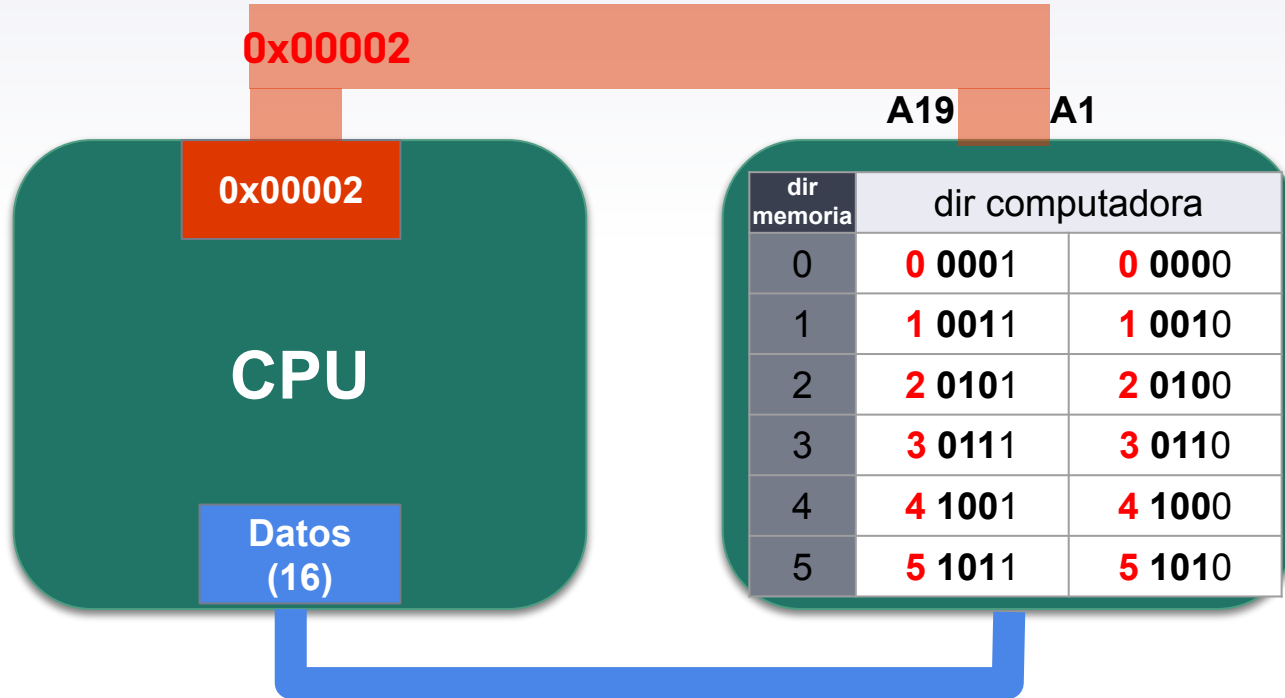
Ahora cada dirección de memoria real contiene 16 bits, pero cada byte es apuntado por una dirección distinta de la CPU. Escribiendo las direcciones en binario, vemos que el bit menos significativo indica si es el byte menos significativo o más significativo de una palabra de memoria.



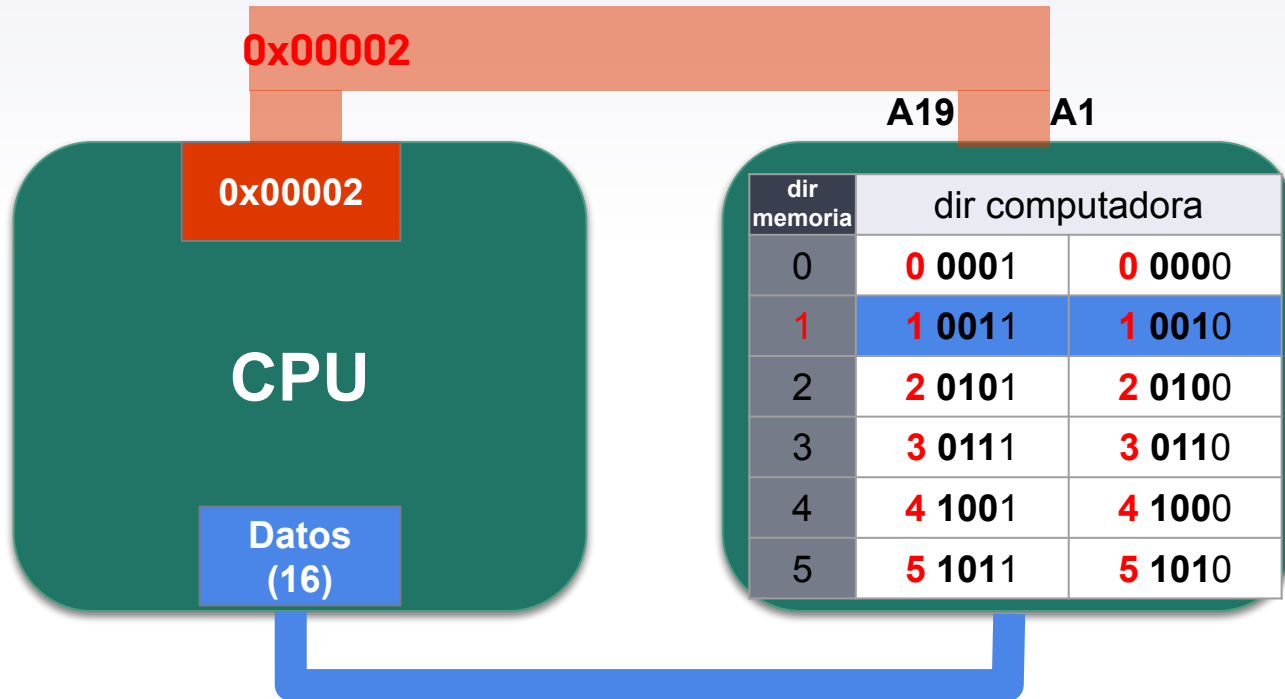
Ahora cada dirección de memoria real contiene 16 bits, pero cada byte es apuntado por una dirección distinta de la CPU. Escribiendo las direcciones en binario, vemos que el bit menos significativo indica si es el byte menos significativo o más significativo de una palabra de memoria.



Ejemplo: la CPU accede a la dirección 0x00002. Pone en el bus de direcciones 0x00002.



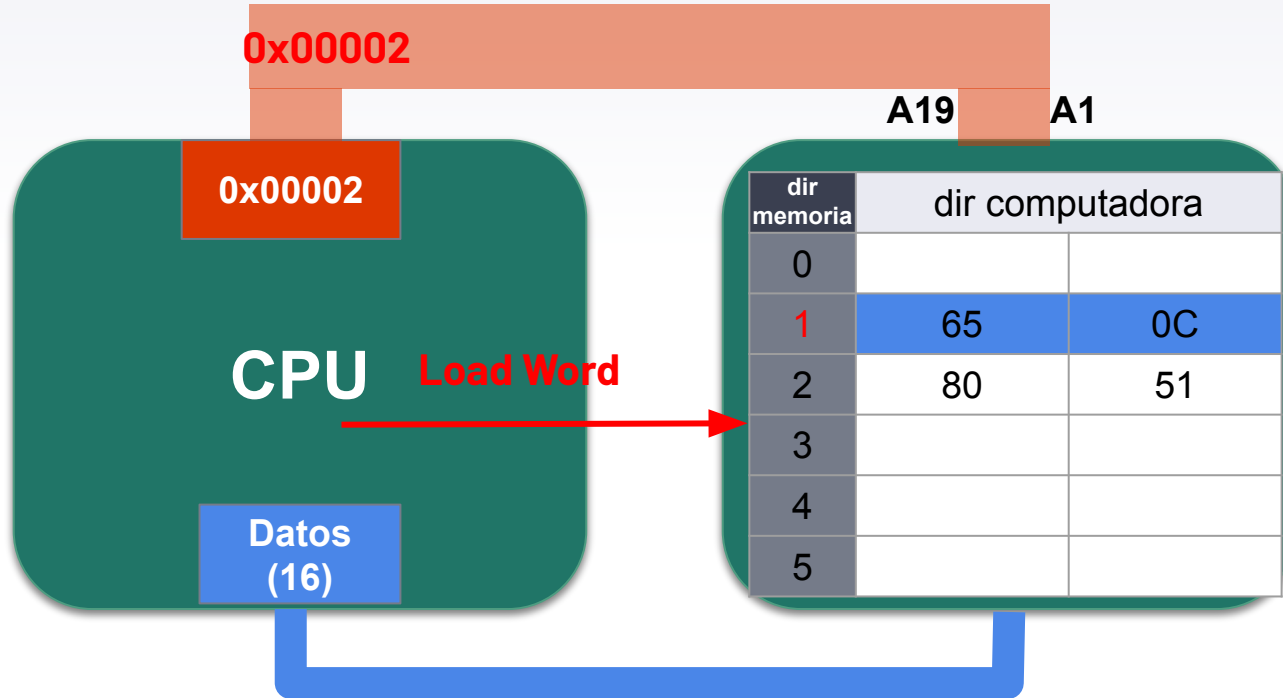
Ejemplo: la CPU accede a la dirección 0x00002. Pone en el bus de direcciones 0x00002. Esta dirección en binario equivale a 0000 0000 0000 0000 0010. Notemos que el bit menos significativo no está conectado a la memoria. Por ende la memoria ve solo el número: 0000 0000 0000 0000 001 --> Equivale a la dirección de memoria 1.



Ejemplo: la CPU accede a la dirección 0x00002. Pone en el bus de direcciones 0x00002. Esta dirección en binario equivale a 0000 0000 0000 0000 0010. Notemos que el bit menos significativo no está conectado a la memoria. Por ende la memoria ve solo el número:

0000 0000 0000 0000 001 --> Equivale a la dirección de memoria 1.

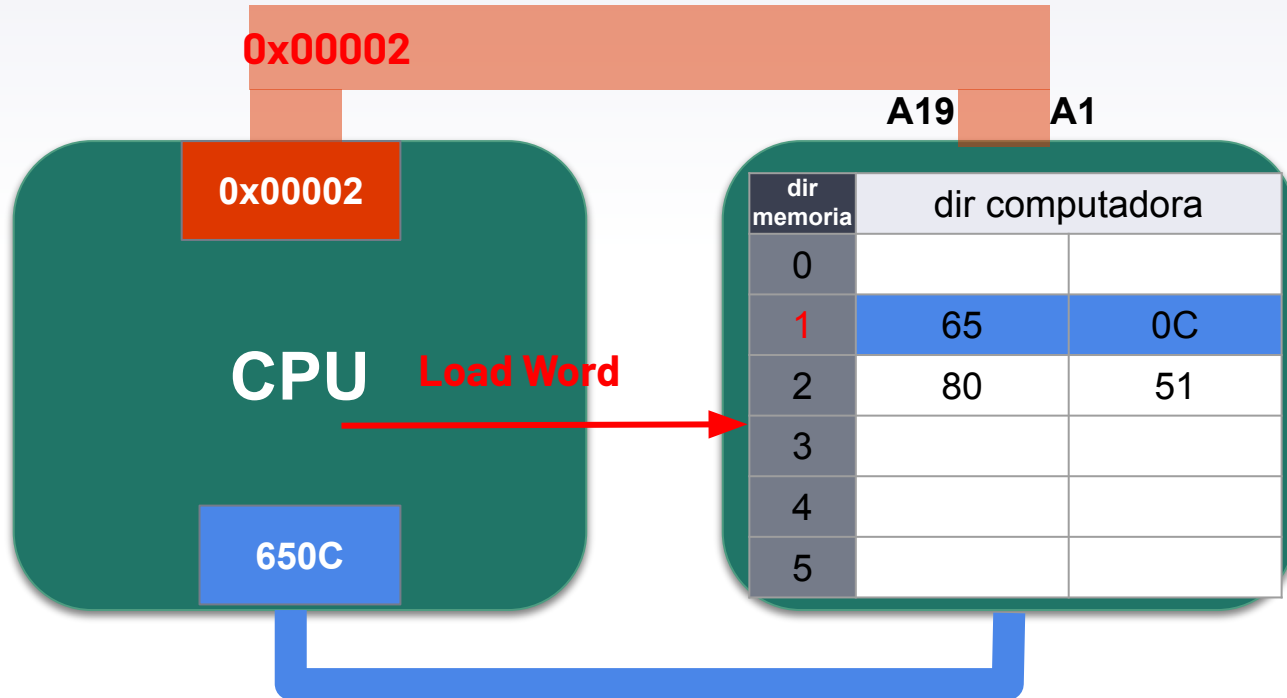
Al ejecutar Load word, el contenido de la dirección de memoria 1 se copia al bus de datos.



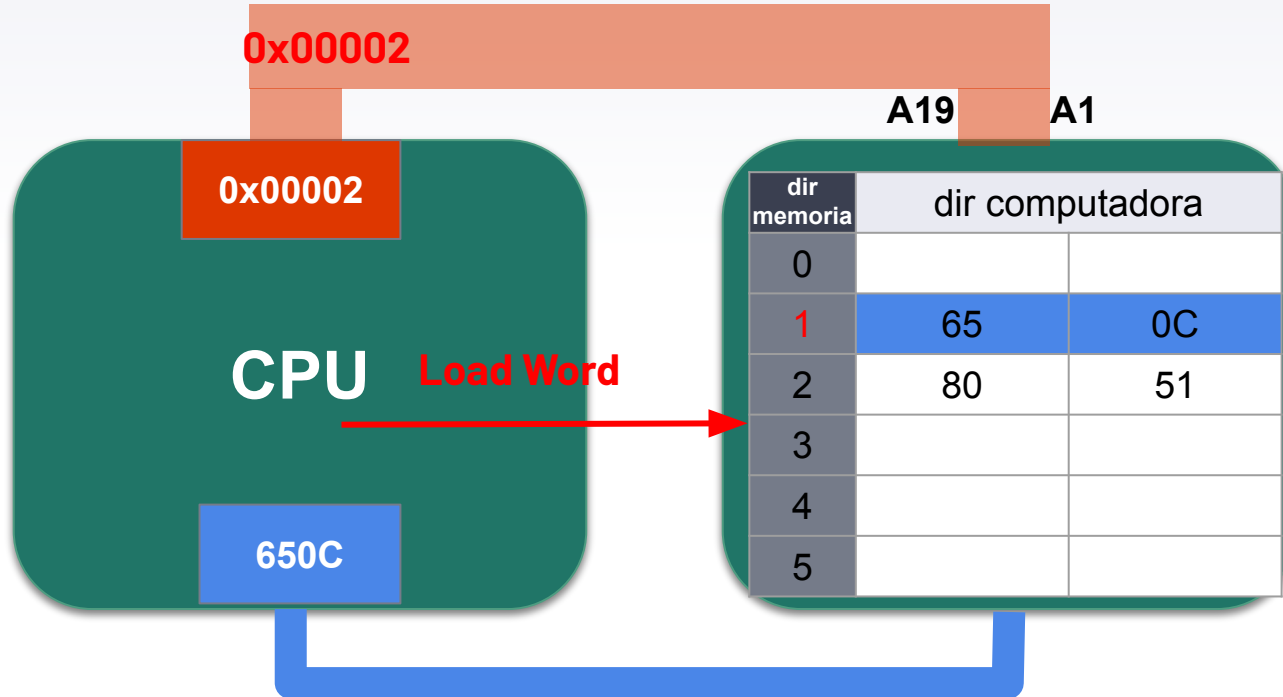
Ejemplo: la CPU accede a la dirección 0x00002. Pone en el bus de direcciones 0x00002. Esta dirección en binario equivale a 0000 0000 0000 0000 0010. Notemos que el bit menos significativo no está conectado a la memoria. Por ende la memoria ve solo el número:

0000 0000 0000 0000 001 --> Equivale a la dirección de memoria 1.

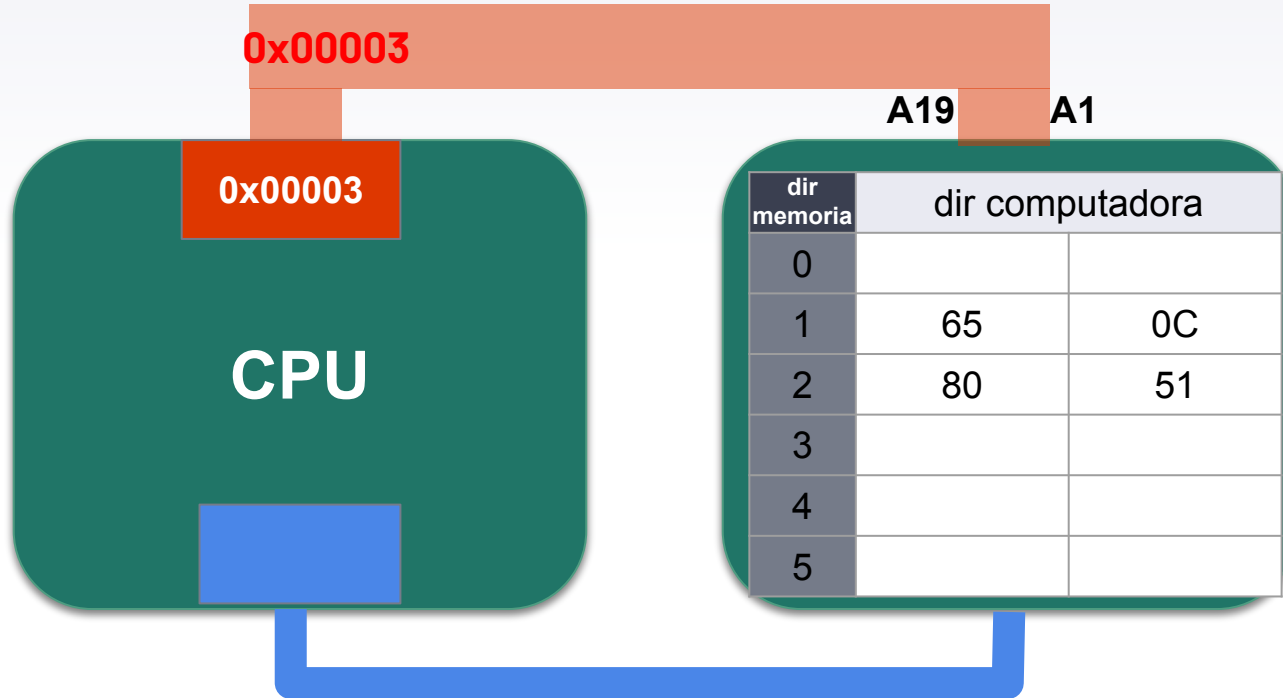
Al ejecutar Load word, el contenido de la dirección de memoria 1 se copia al bus de datos.



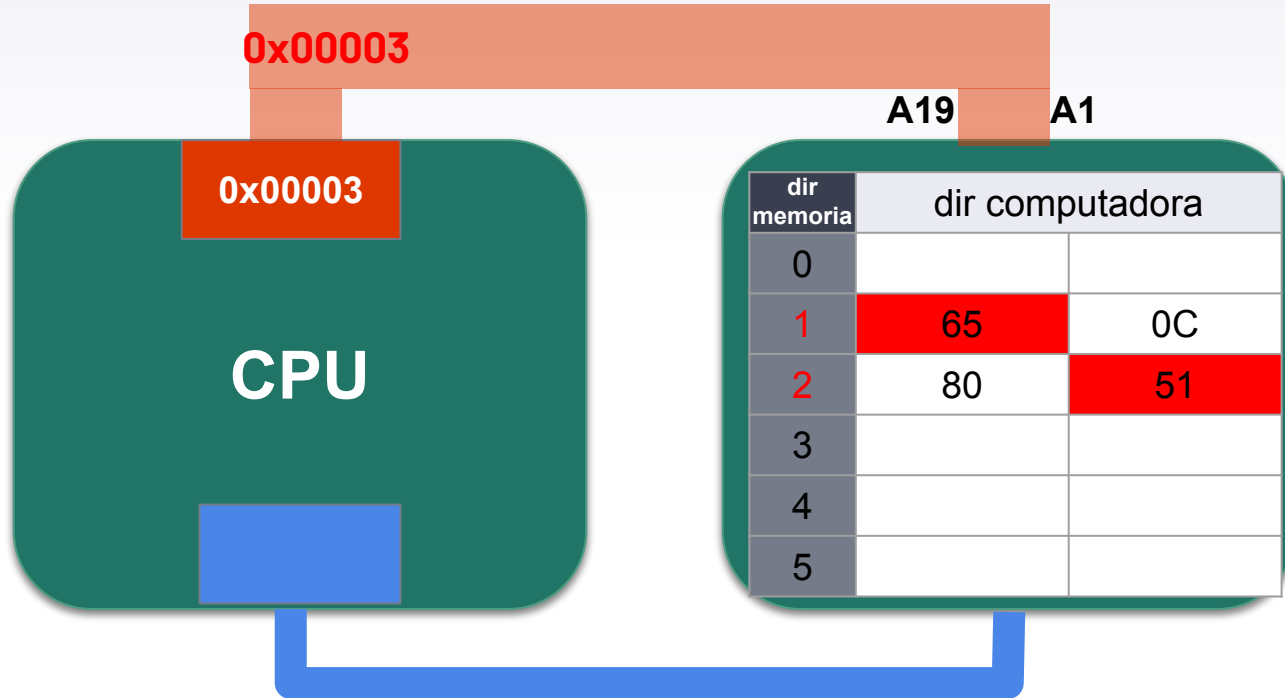
Notemos que si bien se leyó la posición de memoria 0x00002, esta está direccionada al byte, por ende como la CPU utiliza Load Word y necesita acceder a 16 bits, se lee también la posición de memoria 0x00003. Es por esto que el valor final en el bus de datos es 650C, ya que incluye la dirección 0x00003 y 0x00002. Ambas direcciones se pueden leer en una sola palabra de memoria (la 1).



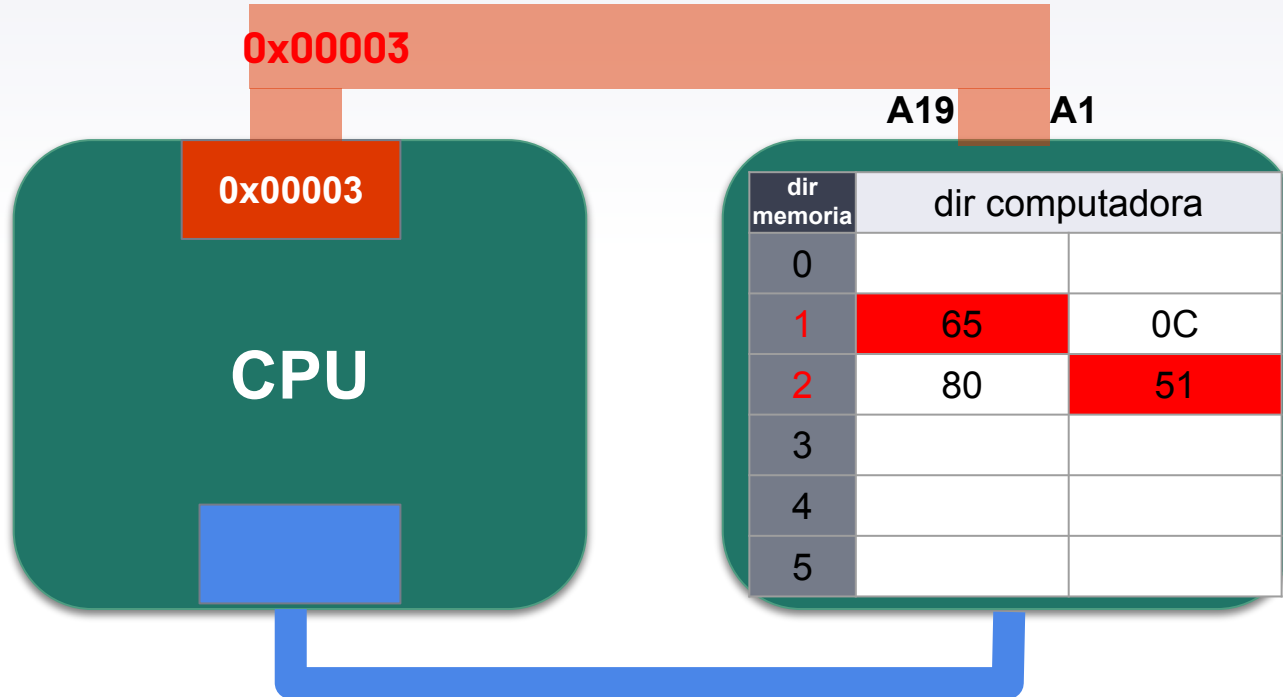
Ejemplo2: Ahora la CPU trata de acceder a la dirección 0x00003. Dado que 0x00003 apunta a un byte, y la CPU quiere leer 16 bits con Load Word... debe leer 0x00003 y 0x00004. Vemos que 0x00003 en binario es 0000 0000 0000 0000 0011, que despreciando su último bit es 0000 0000 0000 0000 001 (dir de memoria 1). Pero 0x00004 en binario es 0000 0000 0000 0000 0100, despreciando su último bit es 0000 0000 0000 0000 0010. Dir de memoria 2.



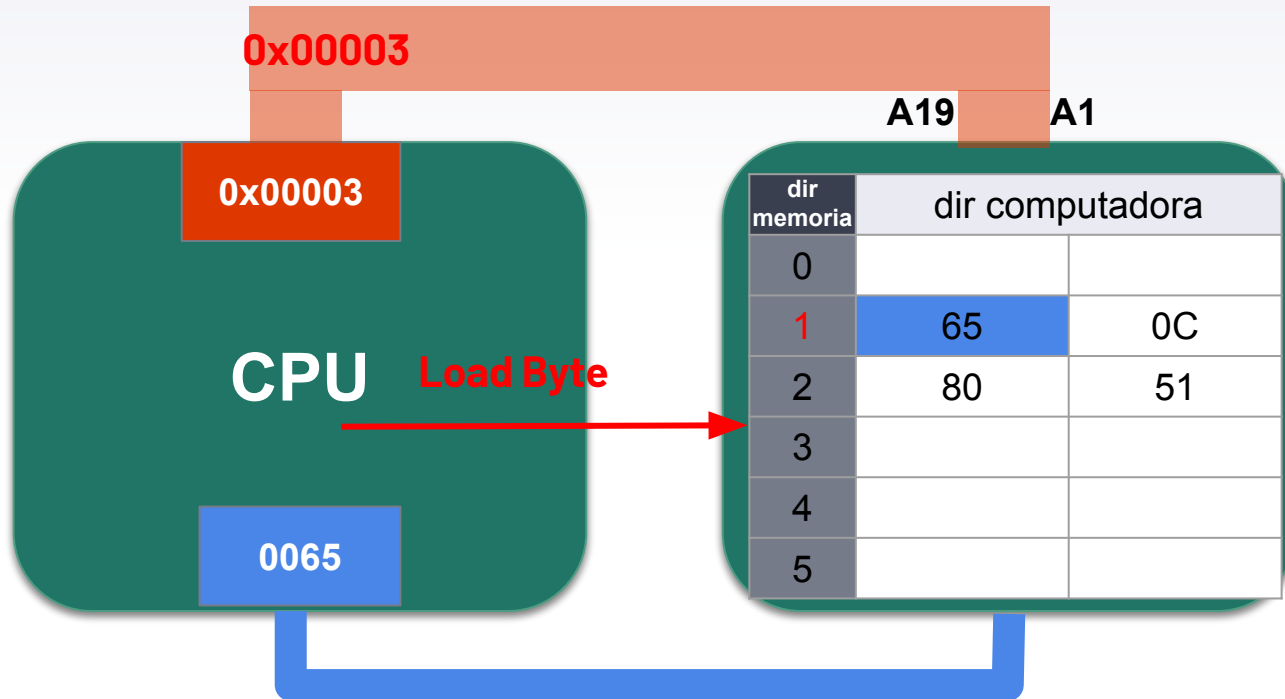
Ejemplo2: Ahora la CPU trata de acceder a la dirección 0x00003. Dado que 0x00003 apunta a un byte, y la CPU quiere leer 16 bits con Load Word... debe leer 0x00003 y 0x00004. Vemos que 0x00003 en binario es 0000 0000 0000 0000 0011, que despreciando su último bit es 0000 0000 0000 0000 001 (**dir de memoria 1**). Pero 0x00004 en binario es 0000 0000 0000 0000 0100, despreciando su último bit es 0000 0000 0000 0000 0010 (**dir de memoria 2**).



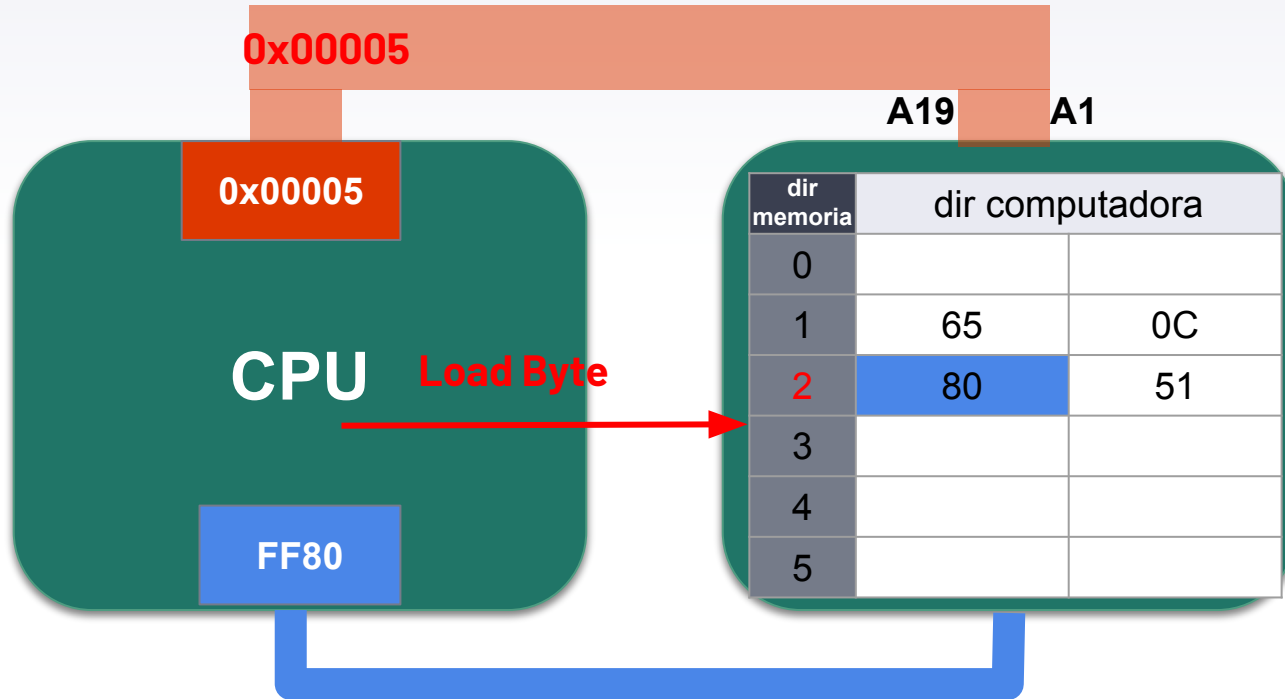
Las palabras de memoria sólo pueden ser leídas de a una a la vez. Es decir en una sola lectura de memoria no podemos leer la dir de memoria 1 y 2 en paralelo. Este tipo de acceso se conoce como acceso no alineado, y se debe a que la dirección no está alineada con el tamaño de palabra de memoria y la operación de lectura. Algunas arquitecturas no soportan este tipo de accesos. Otras arquitecturas imponen una penalidad de tiempo en el acceso.



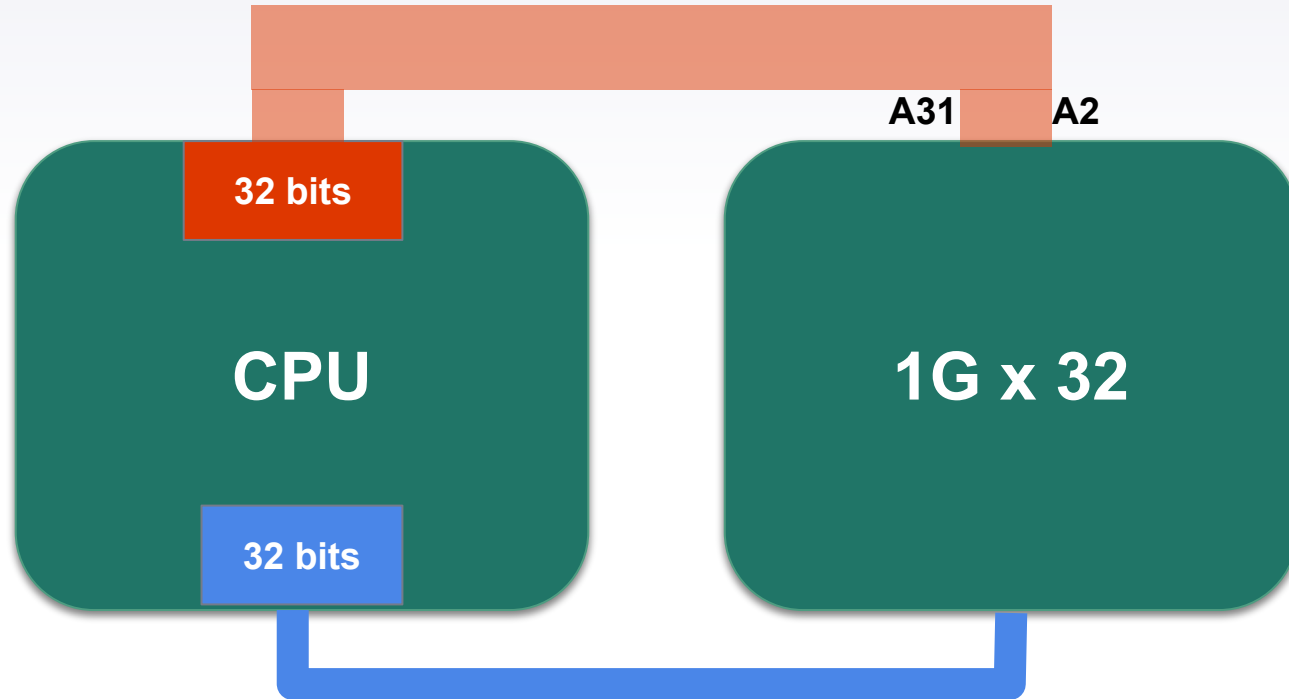
El tipo de operación es importante, ya que si hubiésemos hecho LoadByte, el byte no cruza el límite de una palabra de memoria, y es posible acceder al dato sin problemas. La CPU completa los 8 bits más significativos propagando el signo. En este caso 0x65 es positivo por ende propaga con ceros.



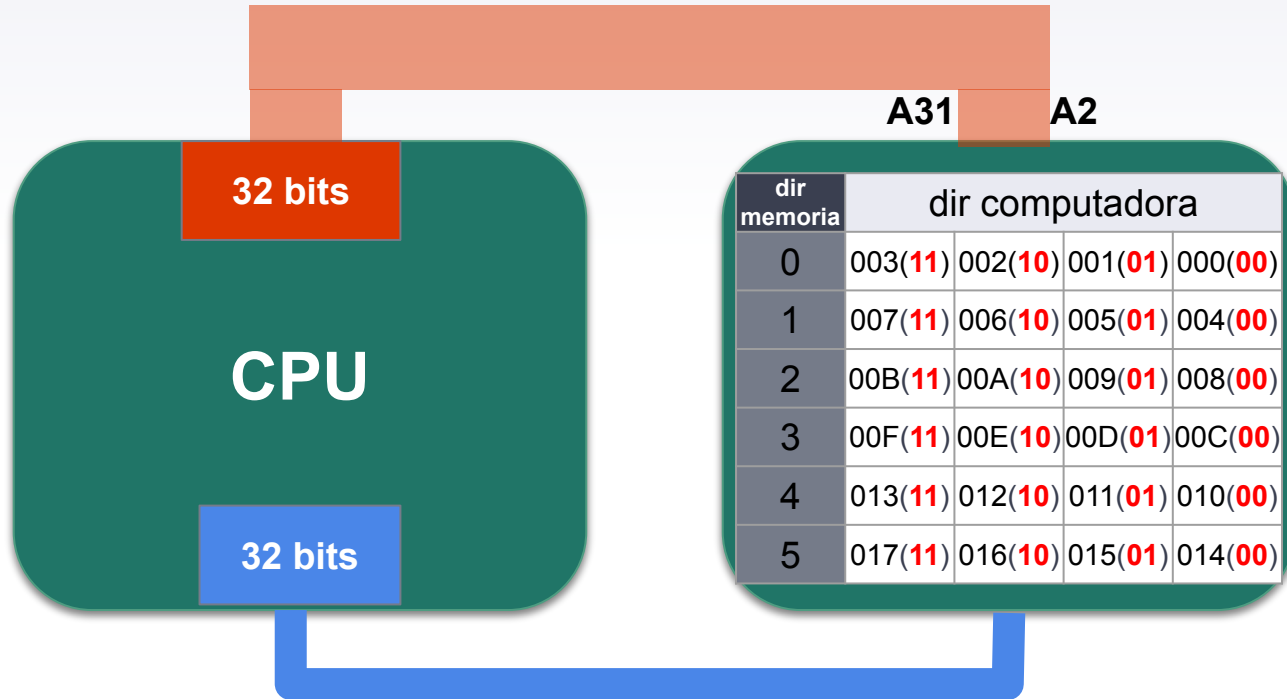
Si accedemos con Load Byte a 0x00005, en esa posición existe un número negativo (en 8 bits), por ende al leer el mismo y convertirse a 16 bits se propaga el bit de signo haciendo que el valor efectivo sea 0xFF80.



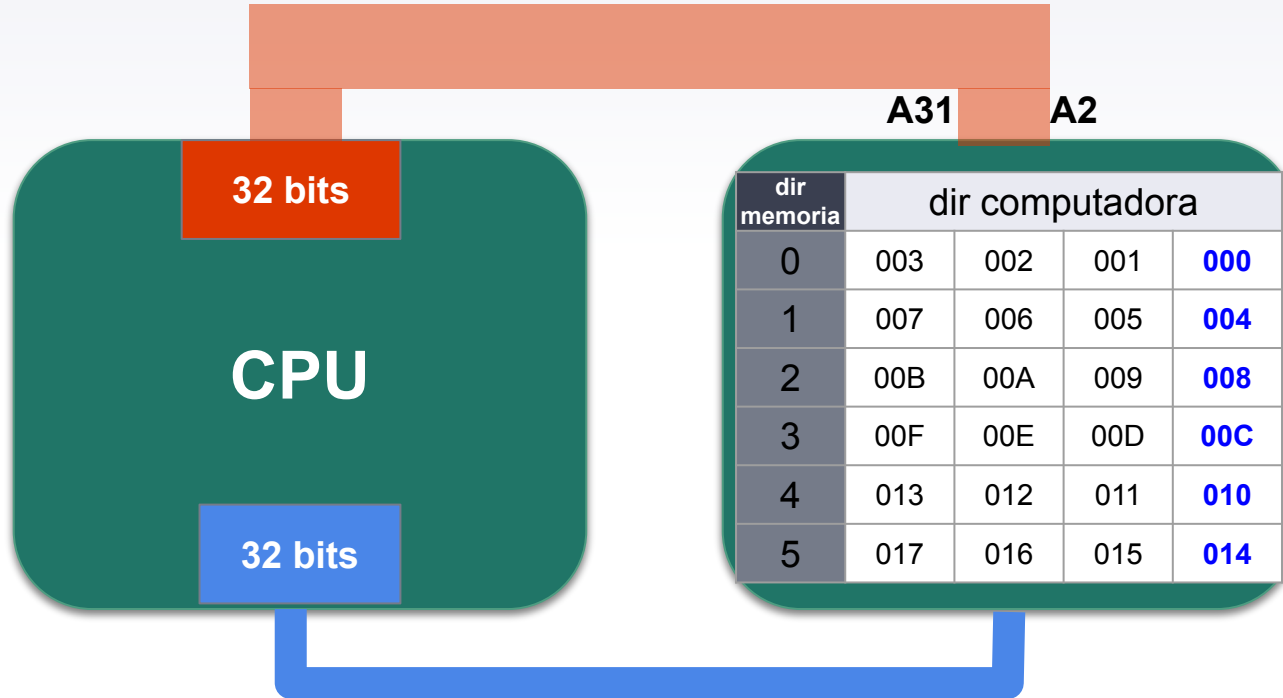
En el caso de RISC-V, el direccionamiento también es al byte. Las palabras de memoria son de 32 bits, y se direcciona con 32 bits. Es por esto que existen 1G palabras de memoria posibles (cada una de 32 bits) pero existen 4G direcciones (cada una apuntando a un byte).



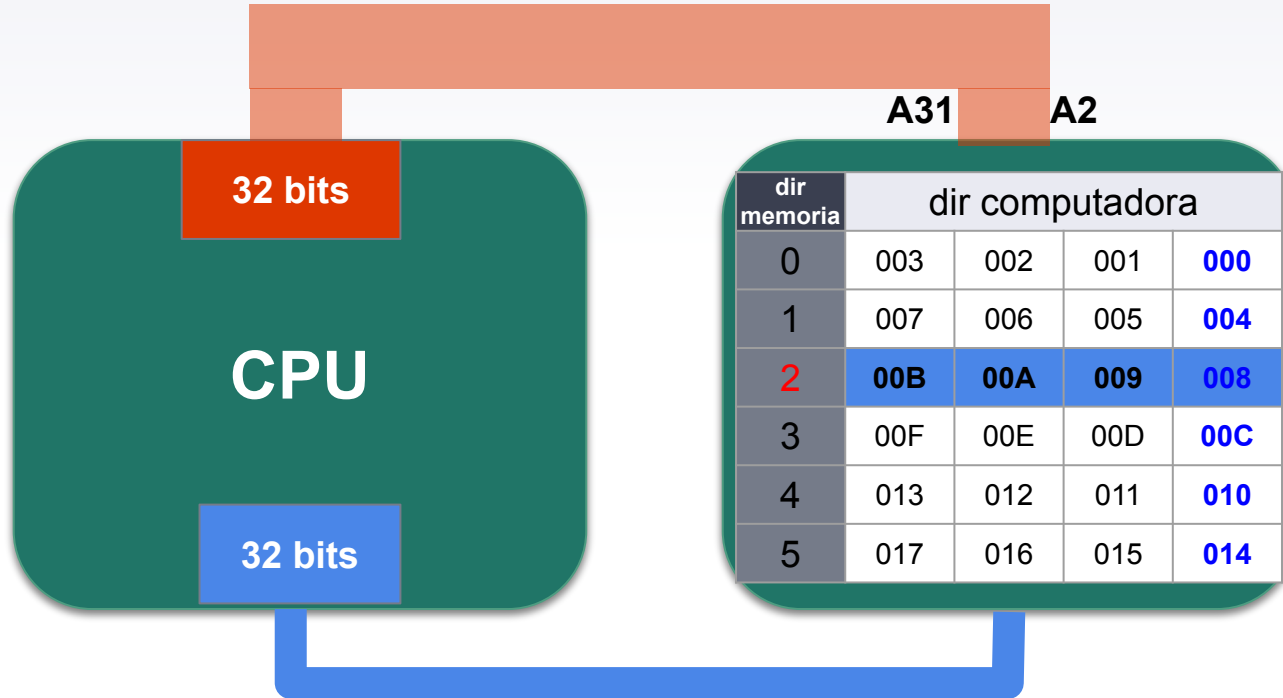
Los dos bits menos significativos de una dirección deben terminar en 00 para que LW (load word) se encuentre alineado.



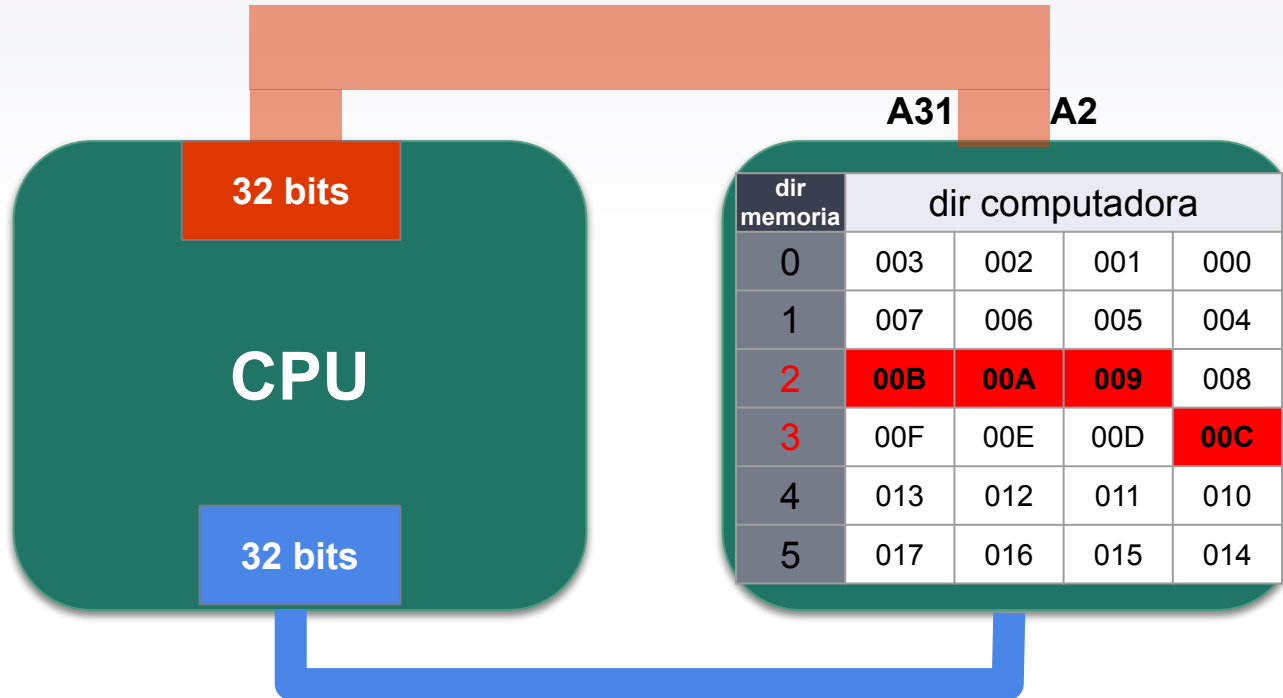
Los dos bits menos significativos de una dirección deben terminar en 00 para que LW (load word) se encuentre alineado. Por ende las direcciones que en hexadecimal terminen con 0, 4, 8 y C estarán siempre alineadas para LW y SW (Store Word).



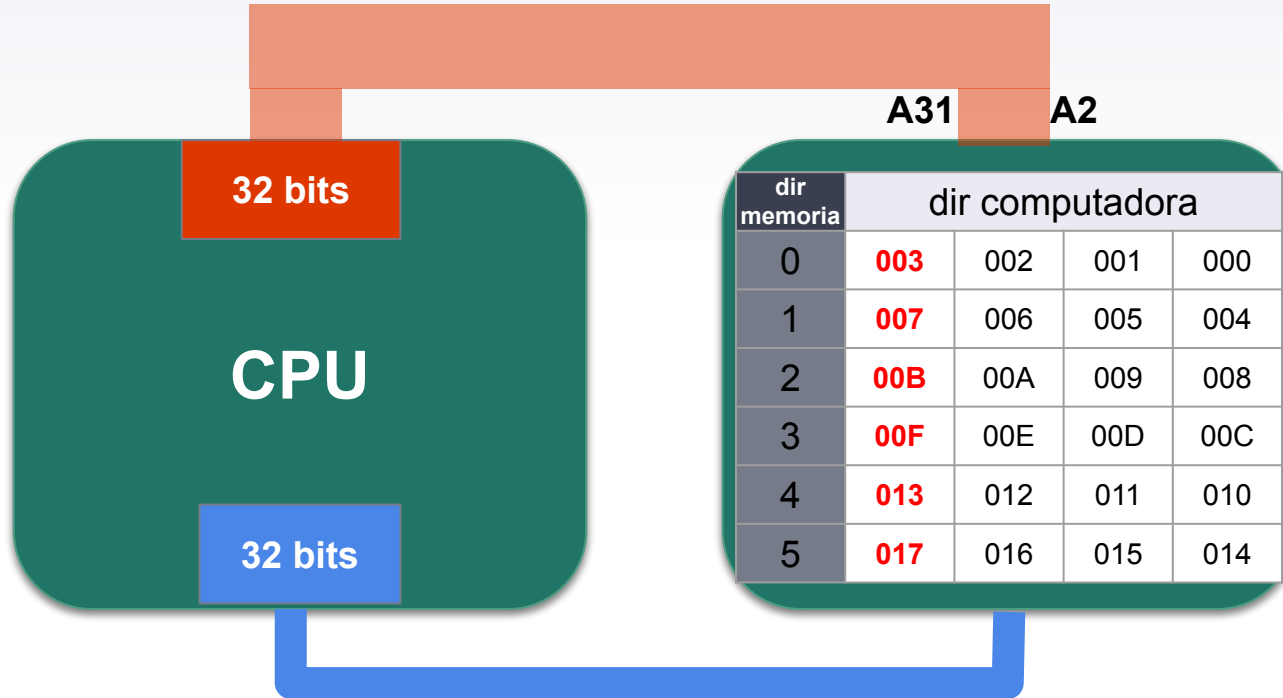
Ejemplo: Se accede a la dirección 0x00000008. Vemos que se leen los bytes de las direcciones 0x00000008, 0x00000009, 0x0000000A y 0x0000000B. Todos estos bytes se encuentran dentro de la palabra de memoria 2.



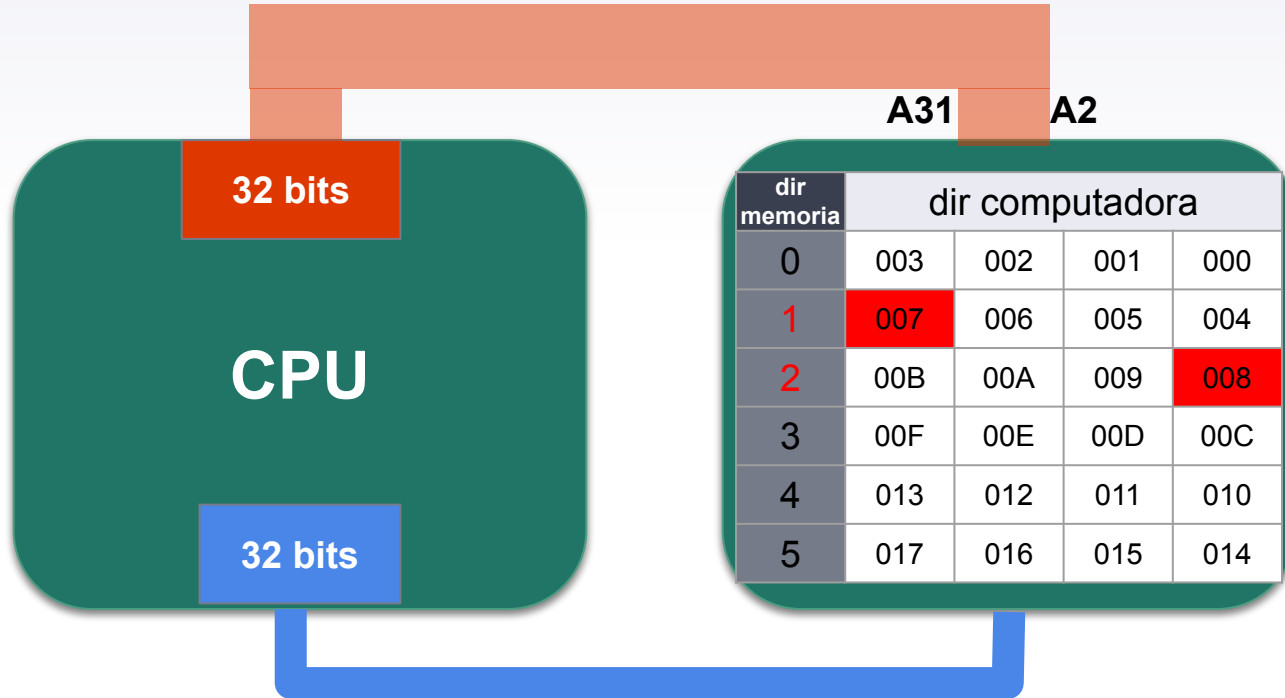
Sin embargo si se accede con LW a la dirección 0x000000009, esto involucra la 9,A,B y C. Vemos que 0x0000000C se encuentra en la siguiente palabra de memoria, por ende el acceso es NO alineado.



En RISC-V existe forma de acceder a un Half Word (16 bits) con LHW (Load Half Word). En este caso mientras los 16 bits estén contenidos dentro de una única palabra de memoria, el acceso es alineado. Pero vemos que si una dirección termina en 3,7,B o F, entonces LHW producirá un acceso no alineado.

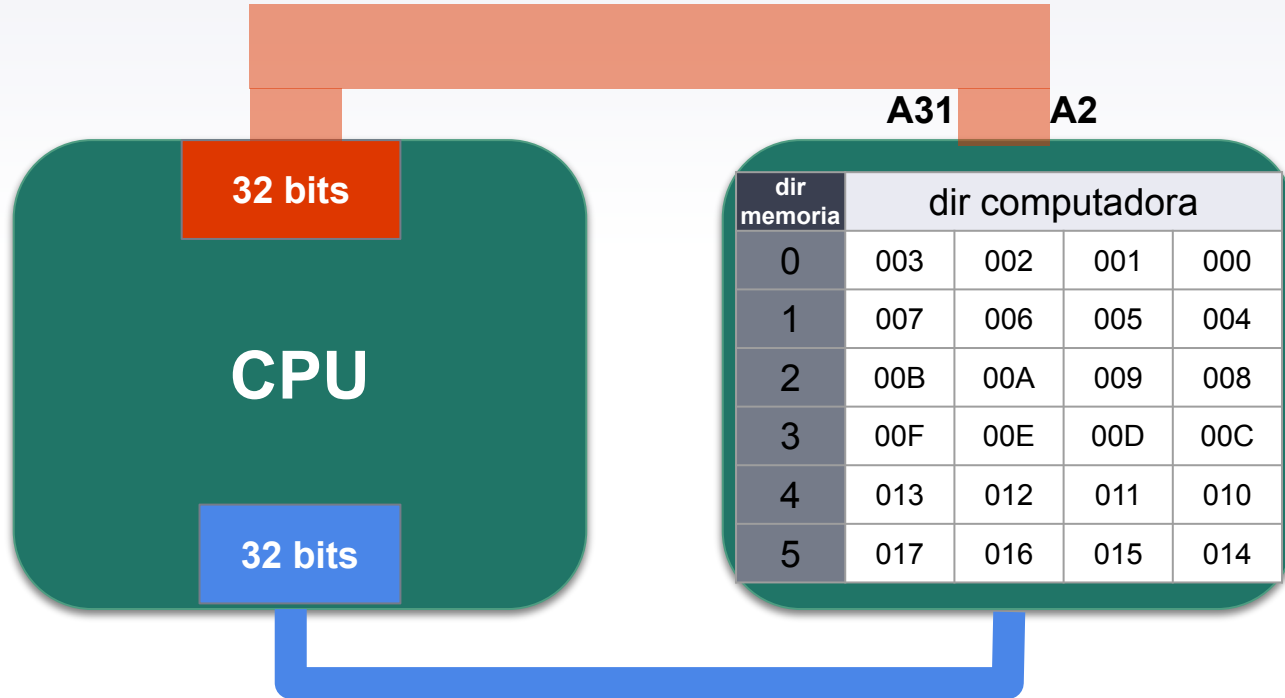


En RISC-V existe forma de acceder a un Half Word (16 bits) con LHW (Load Half Word). En este caso mientras los 16 bits estén contenidos dentro de una única palabra de memoria, el acceso es alineado. Pero vemos que si una dirección termina en 3,7,B o F, entonces LHW producirá un acceso no alineado. Ej: LHW 0x00000007, accede a 7 y 8 pero están en palabras de memoria distintas. Sin embargo, acceder a LHW 0x0000000A funciona sin problemas.



Imaginemos un programa en C que almacena datos en un vector:

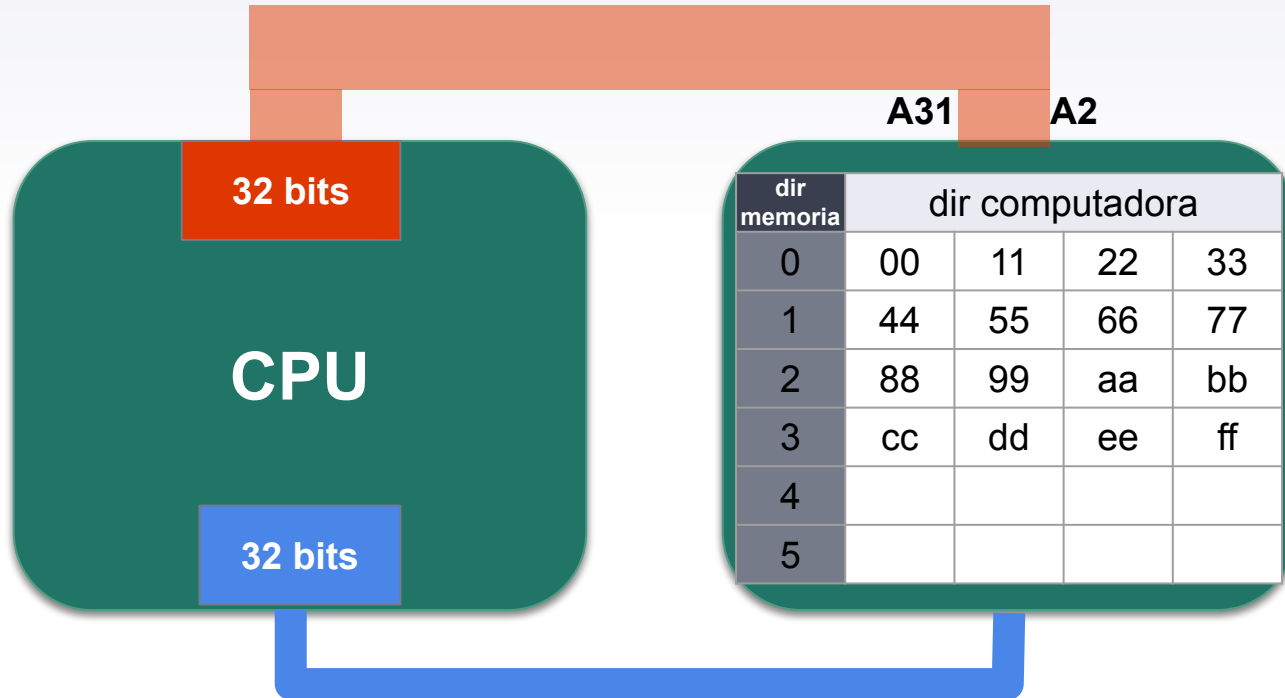
```
int vector[ ] = { 0x00112233 , 0x44556677, 0x8899aabb, 0xccddeeff}.
```



Imaginemos un programa en C que almacena datos en un vector:

int vector[] = { 0x00112233 , 0x44556677, 0x8899aabb, 0xccddeeff}. En memoria se almacena de forma consecutiva...

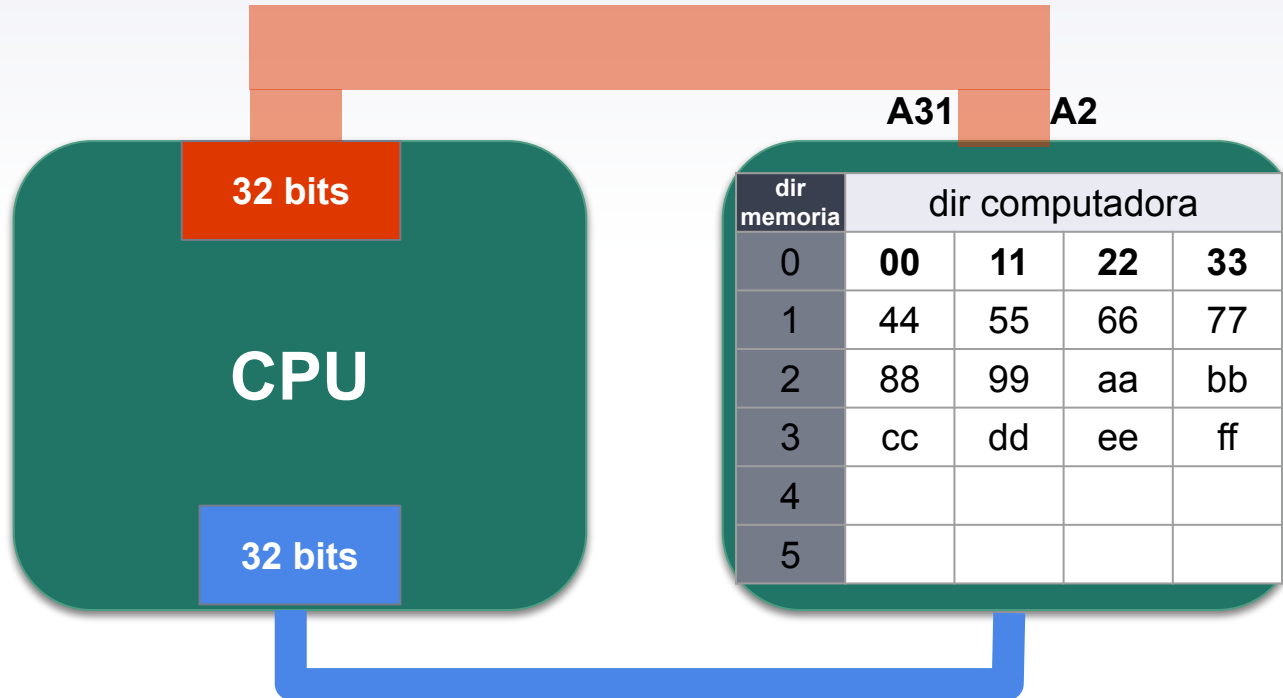
Si queremos acceder al elemento 0, usamos: **int a = vector[0];**



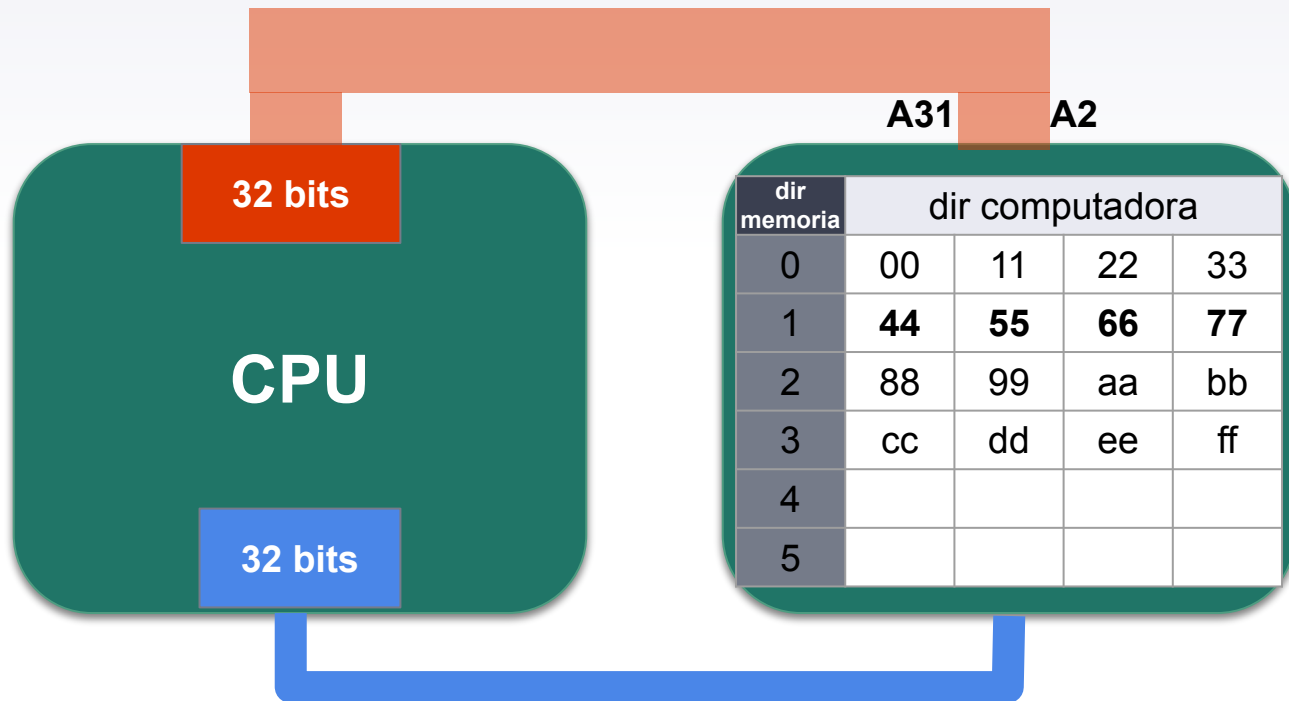
Imaginemos un programa en C que almacena datos en un vector:

int vector[] = { 0x00112233 , 0x44556677, 0x8899aabb, 0xccddeeff}. En memoria se almacena de forma consecutiva...

Si queremos acceder al elemento 0, usamos: **int a = vector[0];** Luego en A se copia el contenido de las posiciones de memoria 0x00000000, 0x00000001, 0x00000002 y 0x00000003.

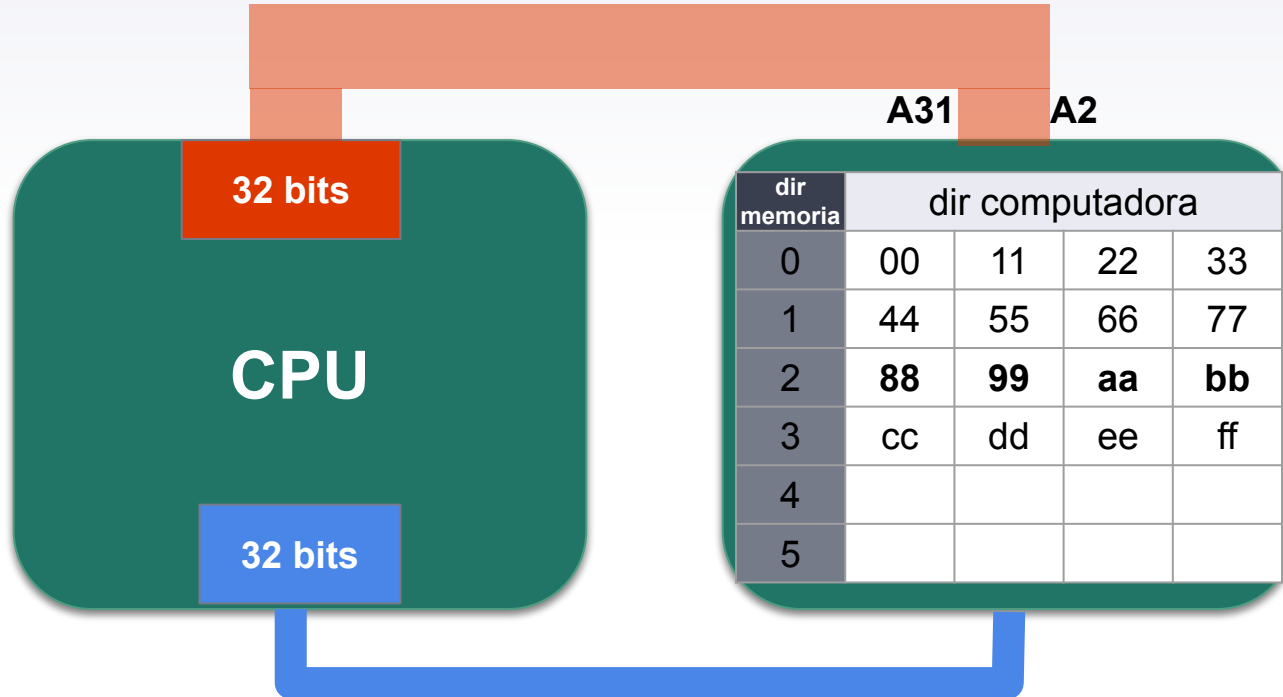


Pero si accedemos con **int a = vector[1]**, si bien la base del vector es la posición 0x00000000, el offset (índice) de 1, no se refiere a 0x00000001, sino a 0x00000004. Esto se debe al direccionamiento al byte. Ese 1 que se usa de índice equivale a 4 direcciones de memoria, por ende se traduce como 1×4 . La dirección del elemento del vector es entonces la base (0x00000000) más el índice (0x00000004).



Si accedemos al tercer elemento con **int a = vector[2]**, entonces nuevamente vamos a multiplicar 2×4 , obteniendo 8, por ende la dirección del tercer elemento del vector es 0x00000008.

Si recorremos secuencialmente la memoria en una arquitectura de 32 bits con direccionamiento al byte, entonces tenemos que incrementar en 4 la dirección cada vez que queramos pasar a la siguiente posición de memoria.



0.1.5 Byte Ordering

Con el direccionamiento al byte podemos almacenar una palabra, supongamos 32 bits, y acceder a la misma byte a byte. Imaginemos el número **0x20110626**. Este número está compuesto por 4 bytes [**0x20** , **0x11**, **0x06**, **0x26**]. Esos 4 bytes se pueden distribuir de varias formas dentro de los 32 bits que ocupan una palabra de memoria. Sin embargo las dos formas más comunes son

Big Endian

dir memoria	dir computadora			
0	0003	0002	0001	0000
dato	26	06	11	20

Little Endian

dir memoria	dir computadora			
0	0003	0002	0001	0000
dato	20	11	06	26

BMS

Si una CPU almacena el contenido de un registro (32 bits) en memoria, utiliza alguna de estas dos convenciones. Cuando recupera el número de memoria, sabe que convención usar por ende es transparente al programador. Sin embargo si accedemos a los datos byte a byte, necesitamos saber cual de las convenciones se utiliza para obtener el dato correspondiente.

Observe cómo se almacena un entero de 32 bits en little endian.

Memory window (Address: 061FF18, Bytes: 32):

0x61ff18:	26 06 11 20	33 33 00 43	0c ff
0x61ff28:	80 ff 61 00	33 12 40 00	01 00

main.c window:

```
7 int main()  
8 {  
9     int b = 0x20110626; // con 0x le damos valor  
10    float a = 128.2; // pero el .2 no era period  
11  
12    printf("Flotante a: %f\n", a);  
13    printf("VAR b: %x\n", b);  
14    printf("Direccion de b: %p\n", &b);  
15 }
```

Output window:

```
Flotante a: 128.199997  
VAR b: 20110626  
Direccion de b: 0061FF18
```

(e.g. 0x401000, or &variable, or \$\$eax)

0x61ff18:	26 06 11 20	33 33 00 43	0c ff
0x61ff28:	80 ff 61 00	33 12 40 00	01 00

▶ 0.1.6 MIPS y RISC-V

En el año 1985 John Hennessy (colega de David Patterson) crea junto a alumnos de posgrado de Stanford la arquitectura MIPS (Microprocessor without Interlock Pipeline Stages). Es un diseño *fabless*. La arquitectura es de tipo **Harvard** e implementa el flamante set de instrucciones reducido (**RISC**).

MIPS existió como empresa y fabricó procesadores utilizados en muchos sistemas como Playstation (R3000 1994), Nintendo 64 (NEC R4300 64 bits 1996) y Playstation 2 (R5900 64 bits 2000).

La arquitectura MIPS influenció otras arquitecturas como DEC Alpha (1992), pero sin duda fue la base para la Acorn Risc Machine (ARM) cuyo diseño fue tomado por Apple para su proyecto Newton y aun cuando Acorn Computers dejó de existir como tal, se formó Advanced RISC Machines (ARM).

En el año 2010 un grupo de alumnos de posgrado de Berkeley (U.C.) junto con David Patterson plantearon la arquitectura **RISC-V** como una arquitectura completamente libre (royalty free). Toma **lo mejor de MIPS/ARM** en una arquitectura nueva y **escalable** que asegura compatibilidad a todos los usuarios de esta ISA.

Hoy la RISC-V Foundation es financiada por muchas empresas con distintos niveles de membresía.