El procesador

continuación

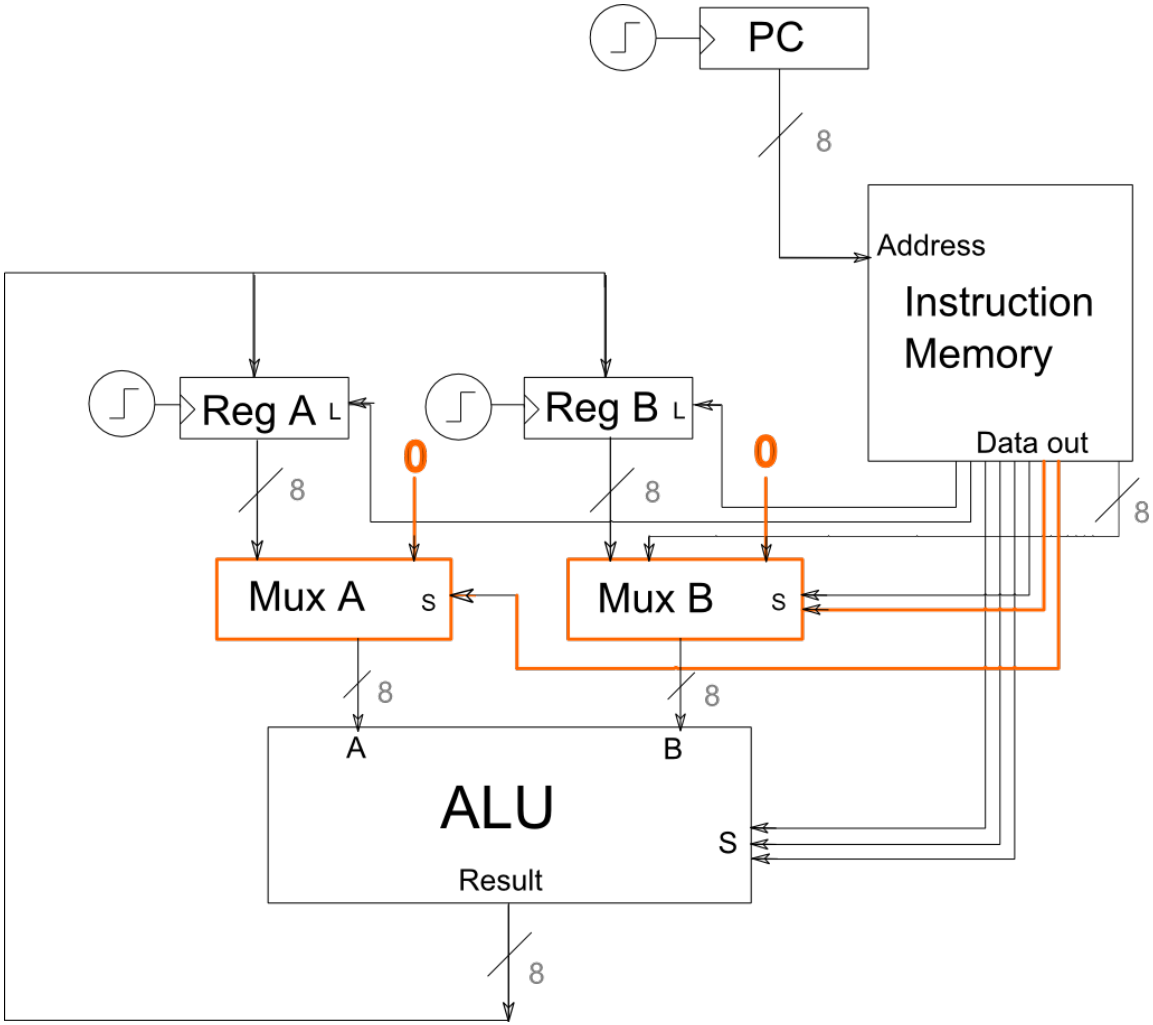
Arquitectura de Computadores – IIC2343

Yadran Eterovic S. (yadran@uc.cl)

2025-2

Por ahora, cada instrucción de nuestro computador básico tiene 16 bits:

- 8 bits que corresponden a las señales de control
- … y otros 8 bits para el literal



Instrucciones de 16 bits de largo implica que la Instruction Memory debe estar formada por palabras de 16 bits de largo

→ cada palabra de memoria se compone de 16 flip-flops

La	Lb	Sa0	Sb0	Sb1	Sop2	Sop1	Sop0	Operación
1	0	1	0	0	0	0	0	A=B
0	1	0	1	1	0	0	0	B=A
1	0	0	0	1	0	0	0	A=Lit
0	1	0	0	1	0	0	0	B=Lit
1	0	0	0	0	0	0	0	A=A+B
0	1	0	0	0	0	0	0	B=A+B
1	0	0	0	1	0	0	0	A=A+Lit
1	0	0	0	0	0	0	1	A=A-B
0	1	0	0	0	0	0	1	B=A-B
1	0	0	0	1	0	0	1	A=A-Lit
1	0	0	0	0	0	1	0	A=A and B
0	1	0	0	0	0	1	0	B=A and B
1	0	0	0	1	0	1	0	A=A and Lit
1	0	0	0	0	0	1	1	A=A or B
0	1	0	0	0	0	1	1	B=A or B
1	0	0	0	1	0	1	1	A=A or Lit
1	0	0	0	0	1	0	0	A=notA
0	1	0	0	0	1	0	0	B=notA
1	0	0	0	1	1	0	0	A=notLit
1	0	0	0	0	1	0	1	A=A xor B
0	1	0	0	0	1	0	1	B=A xor B
1	0	0	0	1	1	0	1	A=A xor Lit
1	0	0	0	0	1	1	0	A=shift left A
0	1	0	0	0	1	1	0	B=shift left A
1	0	0	0	1	1	1	0	A=shift left Lit
1	0	0	0	0	1	1	1	A=shift right A
0	1	0	0	0	1	1	1	B=shift right A
1	0	0	0	1	1	1	1	A=shift right Lit

Hay 8 señales de control, permitiendo $2^8 = 256$ instrucciones posibles

... pero sólo tenemos 28 instrucciones distintas

¿Cómo podemos ahorrar flip-flops en la memoria de instrucciones?

... o, ¿cómo podemos hacer que el largo de las instrucciones sea independiente del número de señales de control que hay que manejar?

La	Lb	Sa0	Sb0	Sb1	Sop2	Sop1	Sop0	Operación
1	0	1	0	0	0	0	0	A=B
0	1	0	1	1	0	0	0	B=A
1	0	0	0	1	0	0	0	A=Lit
0	1	0	0	1	0	0	0	B=Lit
1	0	0	0	0	0	0	0	A=A+B
0	1	0	0	0	0	0	0	B=A+B
1	0	0	0	1	0	0	0	A=A+Lit
1	0	0	0	0	0	0	1	A=A-B
0	1	0	0	0	0	0	1	B=A-B
1	0	0	0	1	0	0	1	A=A-Lit
1	0	0	0	0	0	1	0	A=A and B
0	1	0	0	0	0	1	0	B=A and B
1	0	0	0	1	0	1	0	A=A and Lit
1	0	0	0	0	0	1	1	A=A or B
0	1	0	0	0	0	1	1	B=A or B
1	0	0	0	1	0	1	1	A=A or Lit
1	0	0	0	0	1	0	0	A=notA
0	1	0	0	0	1	0	0	B=notA
1	0	0	0	1	1	0	0	A=notLit
1	0	0	0	0	1	0	1	A=A xor B
0	1	0	0	0	1	0	1	B=A xor B
1	0	0	0	1	1	0	1	A=A xor Lit
1	0	0	0	0	1	1	0	A=shift left A
0	1	0	0	0	1	1	0	B=shift left A
1	0	0	0	1	1	1	0	A=shift left Lit
1	0	0	0	0	1	1	1	A=shift right A
0	1	0	0	0	1	1	1	B=shift right A
1	0	0	0	1	1	1	1	A=shift right Lit

Codificamos las instrucciones usando ***opcodes***

Cada **opcode** está asociado de manera única a una instrucción distinta:

- numeramos (en binario) las instrucciones correlativamente

  ... y usamos estos números como identificadores de las instrucciones

- por ahora, vamos a usar opcodes de 6 bits, desde 000000 hasta 011011

  ... de modo que las instrucciones van a tener 14 bits: opcode + literal

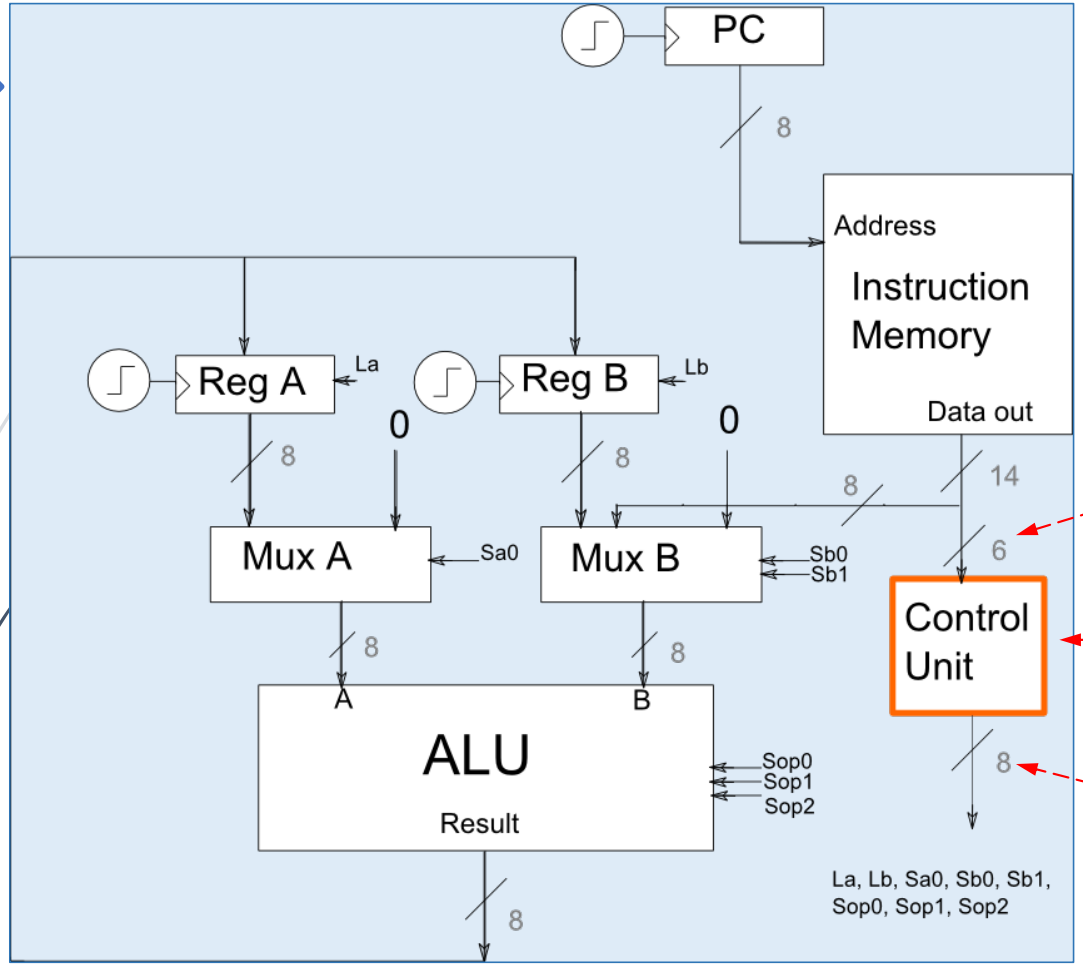
Opcode	La	Lb	Sa0	Sb0	Sb1	Sop2	Sop1	Sop0	Operación
000000	1	0	1	0	0	0	0	0	A=B
000001	0	1	0	1	1	0	0	0	B=A
000010	1	0	0	0	1	0	0	0	A=Lit
000011	0	1	0	0	1	0	0	0	B=Lit
000100	1	0	0	0	0	0	0	0	A=A+B
000101	0	1	0	0	0	0	0	0	B=A+B
000110	1	0	0	0	1	0	0	0	A=A+Lit
000111	1	0	0	0	0	0	0	1	A=A-B
001000	0	1	0	0	0	0	0	1	B=A-B
001001	1	0	0	0	1	0	0	1	A=A-Lit
001010	1	0	0	0	0	0	1	0	A=A and B
001011	0	1	0	0	0	0	1	0	B=A and B
001100	1	0	0	0	1	0	1	0	A=A and Lit
001101	1	0	0	0	0	0	1	1	A=A or B
001110	0	1	0	0	0	0	1	1	B=A or B
001111	1	0	0	0	1	0	1	1	A=A or Lit
010000	1	0	0	0	0	1	0	0	A=notA
010001	0	1	0	0	0	1	0	0	B=notA
010010	1	0	0	0	1	1	0	0	A=notLit
010011	1	0	0	0	0	1	0	1	A=A xor B
010100	0	1	0	0	0	1	0	1	B=A xor B
010101	1	0	0	0	1	1	0	1	A=A xor Lit
010110	1	0	0	0	0	1	1	0	A=shift left A
010111	0	1	0	0	0	1	1	0	B=shift left A
011000	1	0	0	0	1	1	1	0	A=shift left Lit
011001	1	0	0	0	0	1	1	1	A=shift right A
011010	0	1	0	0	0	1	1	1	B=shift right A
011011	1	0	0	0	1	1	1	1	A=shift right Lit

Opcode	La	Lb	Sa0	Sb0	Sb1	Sop2	Sop1	Sop0	Operación
000000	1	0	1	0	0	0	0	0	A=B
000001	0	1	0	1	1	0	0	0	B=A
000010	1	0	0	0	1	0	0	0	A=Lit
000011	0	1	0	0	1	0	0	0	B=Lit
000100	1	0	0	0	0	0	0	0	A=A+B
000101	0	1	0	0	0	0	0	0	B=A+B
000110	1	0	0	0	1	0	0	0	A=A+Lit
000111	1	0	0	0	0	0	0	1	A=A-B
001000	0	1	0	0	0	0	0	1	B=A-B
001001	1	0	0	0	1	0	0	1	A=A-Lit
001010	1	0	0	0	0	0	1	0	A=A and B
001011	0	1	0	0	0	0	1	0	B=A and B
001100	1	0	0	0	1	0	1	0	A=A and Lit
001101	1	0	0	0	0	0	1	1	A=A or B
001110	0	1	0	0	0	0	1	1	B=A or B
001111	1	0	0	0	1	0	1	1	A=A or Lit
010000	1	0	0	0	0	1	0	0	A=notA
010001	0	1	0	0	0	1	0	0	B=notA
010010	1	0	0	0	1	1	0	0	A=notLit
010011	1	0	0	0	0	1	0	1	A=A xor B
010100	0	1	0	0	0	1	0	1	B=A xor B
010101	1	0	0	0	1	1	0	1	A=A xor Lit
010110	1	0	0	0	0	1	1	0	A=shift left A
010111	0	1	0	0	0	1	1	0	B=shift left A
011000	1	0	0	0	1	1	1	0	A=shift left Lit
011001	1	0	0	0	0	1	1	1	A=shift right A
011010	0	1	0	0	0	1	1	1	B=shift right A
011011	1	0	0	0	1	1	1	1	A=shift right Lit

P.ej., para ejecutar A=A+6, ahora escribimos la instrucción de 14 bits

0 0 0 1 1 0 0 0 0 0 0 1 1 0 :

- los 6 bits de la izquierda corresponden al opcode
- los 8 bits de la derecha corresponden al literal 6 (en binario)

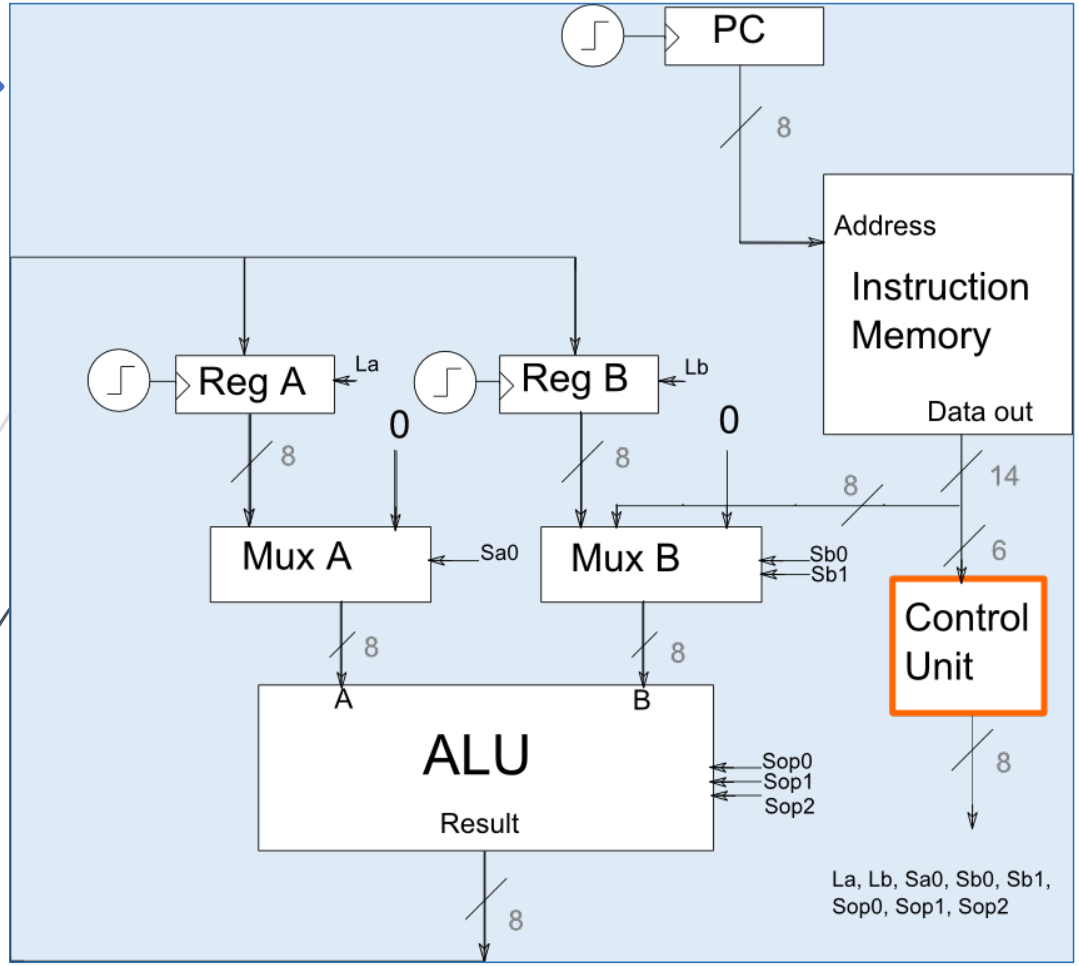


Las 6 líneas de entrada de la *Control Unit* corresponden a los 6 bits de los opcodes

Una **Control Unit** —un circuito digital— traduce los opcodes a las señales de control

Las 8 líneas de salida de la *Control Unit* corresponden a las 8 señales de control

... similarmente al diagrama de la diap. 2



La **Control Unit** es un circuito digital que recibe un input de 6 bits (un *opcode*)

... y produce un output de 8 bits (las señales de control)

El circuito propiamente dicho debe deducirse a partir de la tabla de verdad que se muestra en la diaps. 5 y 6

(a medida que vayamos agregando más funcionalidad a nuestro computador básico, los *opcodes* van a tener más bits y va a haber más señales de control)

(Independiente del uso de *opcodes* ...)

El lenguaje de máquina es difícil de usar para programar el computador

... y de leer al hacer *debugging* de un programa

P.ej., sería más fácil poder escribir

A = A – *literal*

—similarmente a lo que haríamos en un lenguaje de programación moderno—

... que tener que recordar **001001**

El **lenguaje *assembly*** nos ayuda

⟹ versión simbólica del lenguaje de máquina:

- cada instrucción tiene un **nombre** y dos **operandos**
- p.ej., la instrucción en la diap. anterior es una resta (**SUB**), cuyos operandos son el (contenido del) registro *A* (**A**) y un literal (**Lit**), y cuyo resultado se almacena en el mismo registro *A*

  … y por lo tanto se escribe simbólicamente como **`SUB A,Lit`**

Cada instrucción del lenguaje de máquina tiene una única instrucción correspondiente en el lenguaje assembly:

- única combinación de nombre de operación y especificación de dos operandos

El **lenguaje *assembly*** de nuestro computador básico se muestra en las dos primeras columnas de la próx. diap.; p.ej.:

```
MOV A,B
ADD A,Lit
```

Lenguaje *assembly* del computador básico

Instrucción	Operandos	Opcode	La	Lb	Sa0	Sb0	Sb1	Sop2	Sop1	Sop0	Operación
MOV	A,B	000000	1	0	1	0	0	0	0	0	A=B
	B,A	000001	0	1	0	1	1	0	0	0	B=A
	A,Lit	000010	1	0	0	0	1	0	0	0	A=Lit
	B,Lit	000011	0	1	0	0	1	0	0	0	B=Lit
ADD	A,B	000100	1	0	0	0	0	0	0	0	A=A+B
	B,A	000101	0	1	0	0	0	0	0	0	B=A+B
	A,Lit	000110	1	0	0	0	1	0	0	0	A=A+Lit
SUB	A,B	000111	1	0	0	0	0	0	0	1	A=A-B
	B,A	001000	0	1	0	0	0	0	0	1	B=A-B
	A,Lit	001001	1	0	0	0	1	0	0	1	A=A-Lit
AND	A,B	001010	1	0	0	0	0	0	1	0	A=A and B
	B,A	001011	0	1	0	0	0	0	1	0	B=A and B
	A,Lit	001100	1	0	0	0	1	0	1	0	A=A and Lit
OR	A,B	001101	1	0	0	0	0	0	1	1	A=A or B
	B,A	001110	0	1	0	0	0	0	1	1	B=A or B
	A,Lit	001111	1	0	0	0	1	0	1	1	A=A or Lit
NOT	A,A	010000	1	0	0	0	0	1	0	0	A=notA
	B,A	010001	0	1	0	0	0	1	0	0	B=notA
	A,Lit	010010	1	0	0	0	1	1	0	0	A=notLit
XOR	A,A	010011	1	0	0	0	0	1	0	1	A=A xor B
	B,A	010100	0	1	0	0	0	1	0	1	B=A xor B
	A,Lit	010101	1	0	0	0	1	1	0	1	A=A xor Lit
SHL	A,A	010110	1	0	0	0	0	1	1	0	A=shift left A
	B,A	010111	0	1	0	0	0	1	1	0	B=shift left A
	A,Lit	011000	1	0	0	0	1	1	1	0	A=shift left Lit
SHR	A,A	011001	1	0	0	0	0	1	1	1	A=shift right A
	B,A	011010	0	1	0	0	0	1	1	1	B=shift right A
	A,Lit	011011	1	0	0	0	1	1	1	1	A=shift right Lit

Este lenguaje (de máquina y su versión *assembly*) y el computador básico correspondiente sólo nos permiten escribir y ejecutar programas muy simples:

- p.ej., estrictamente secuenciales y sin variables

Este lenguaje (de máquina y su versión *assembly*) y el computador básico correspondiente sólo nos permiten escribir y ejecutar programas muy simples:

- p.ej., estrictamente secuenciales y sin variables

Ahora vamos a agregar nuevas capacidades a nuestro computador básico (la *Control Unit* se va a volver más compleja, pero no la vamos a describir aquí):

- nuevas componentes y señales de control
- nuevas instrucciones

Este lenguaje (de máquina y su versión *assembly*) y el computador básico correspondiente sólo nos permiten escribir y ejecutar programas muy simples:

- p.ej., estrictamente secuenciales y sin variables

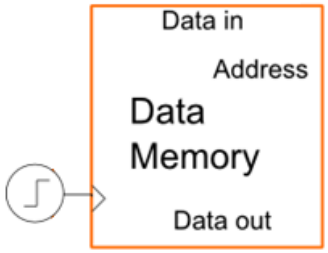
Ahora vamos a agregar nuevas capacidades a nuestro computador básico (la *Control Unit* se va a volver más compleja, pero no la vamos a describir aquí):

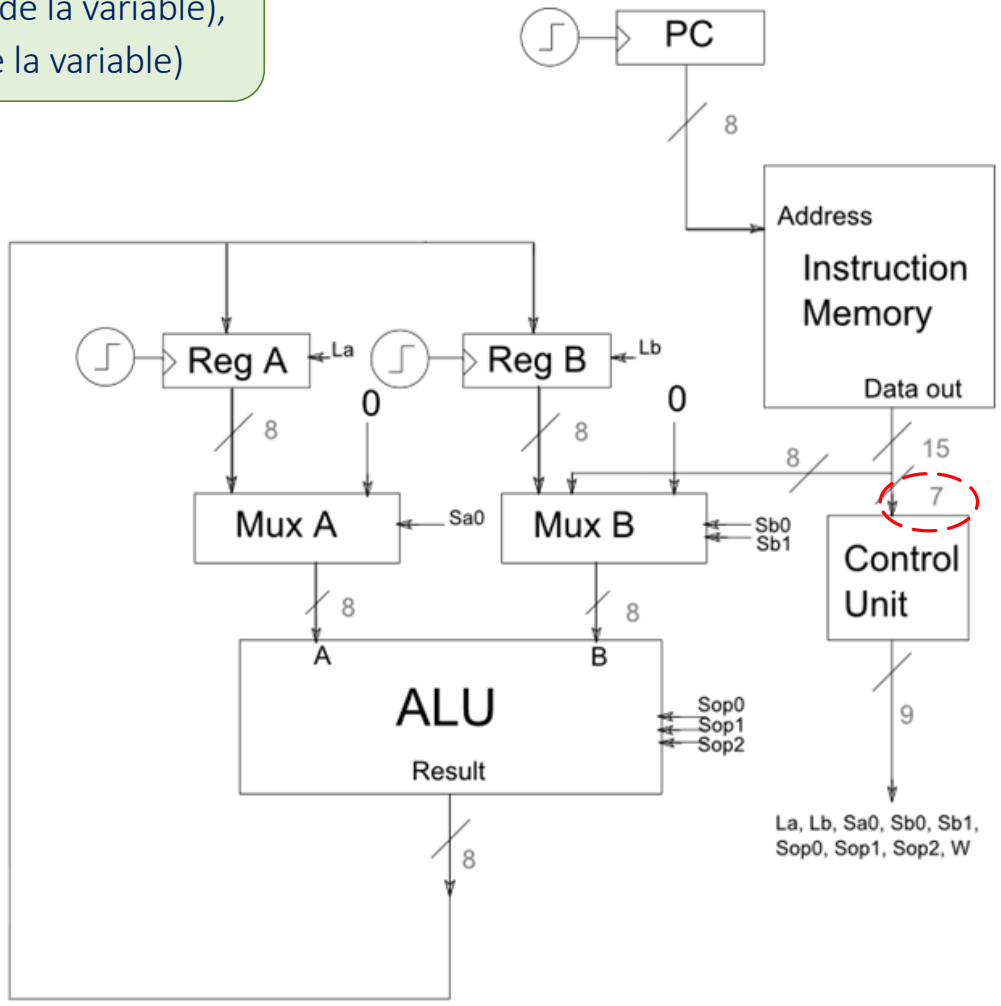
- nuevas componentes y señales de control
- nuevas instrucciones

… fundamentales para que pueda ejecutar programas más “reales”, como los que escribimos en lenguajes de programación modernos:

- lectura y escritura de datos —las variables— en una memoria de datos (*Data Memory*), adicional a la memoria de instrucciones
- ejecución de instrucciones en orden distinto al estrictamente secuencial —sentencias de tipo **`if`** y **`while`**— mediante ”saltos” condicionales e incondicionales
- llamado a y retorno desde funciones (subrutinas, métodos) con parámetros

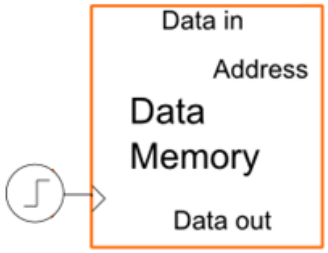
Para tener **variables** en un programa, necesitamos una memoria que podamos leer (usar el valor de la variable), pero también **escribir** (cambiar el valor de la variable)

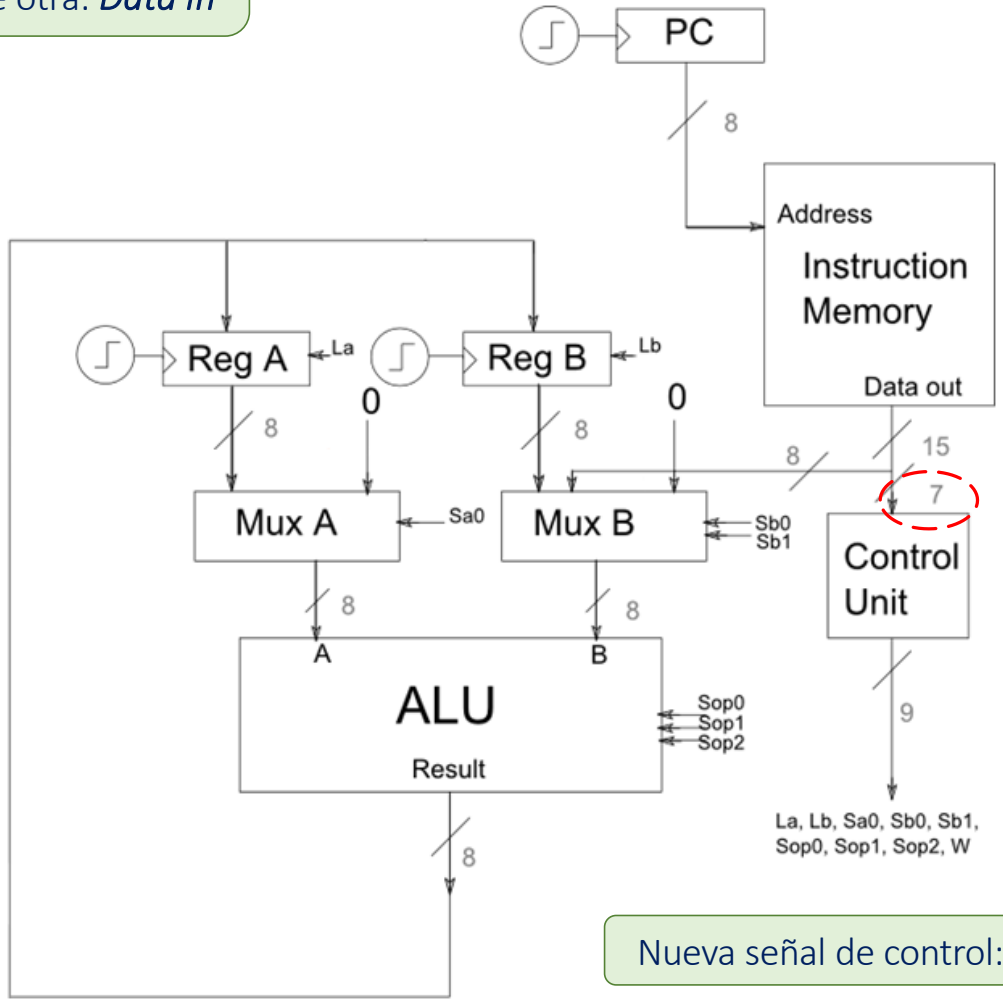




La ***Data Memory*** es similar a la *Instruction Memory*,

… pero además de la entrada *Address* tiene otra: ***Data in***





Nueva señal de control: ***W***

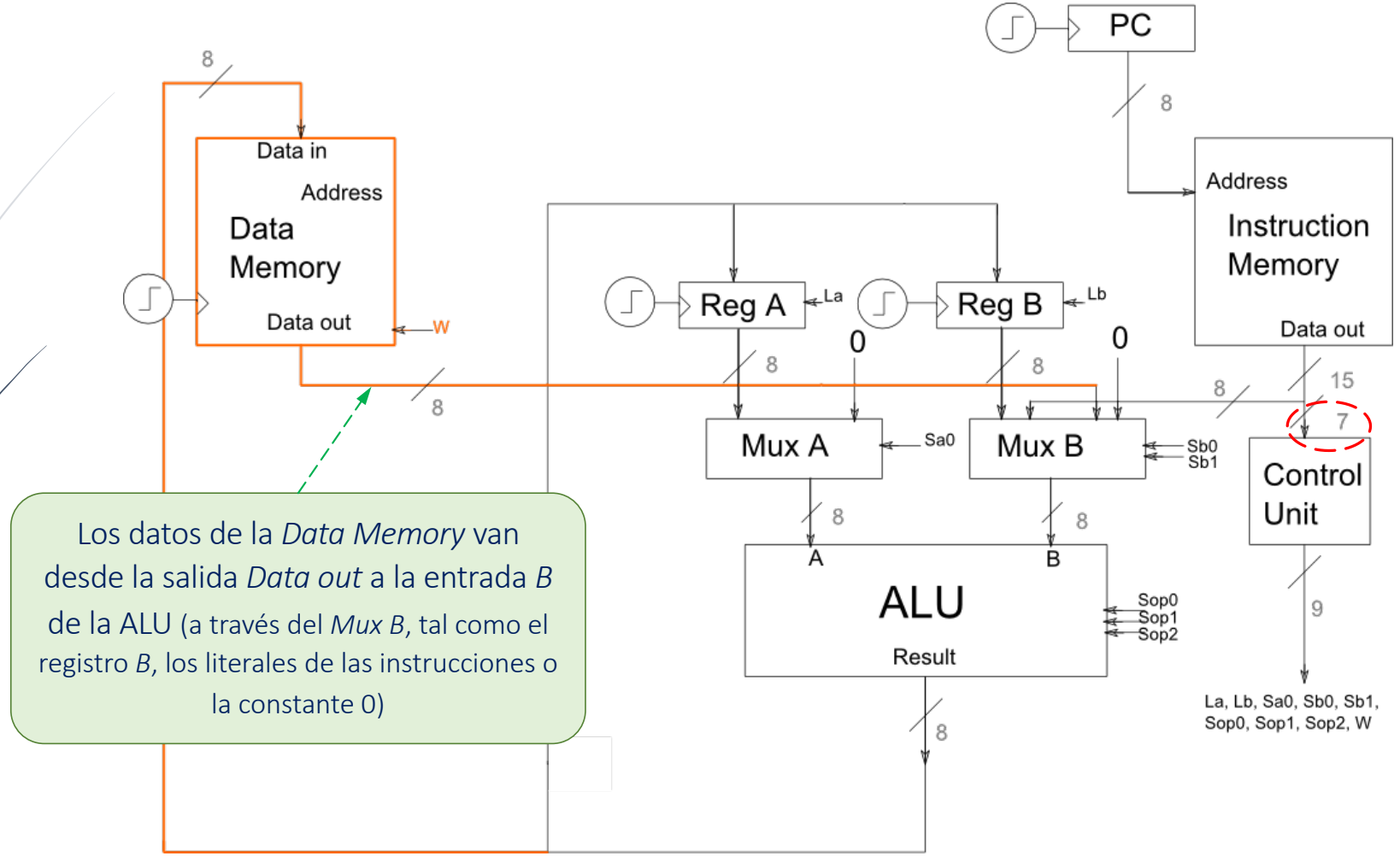
Las instrucciones del programa (el **CODE**) siguen estando en la *Instruction Memory*

… pero ahora además tenemos las variables del programa (la **DATA**) en la *Data Memory*

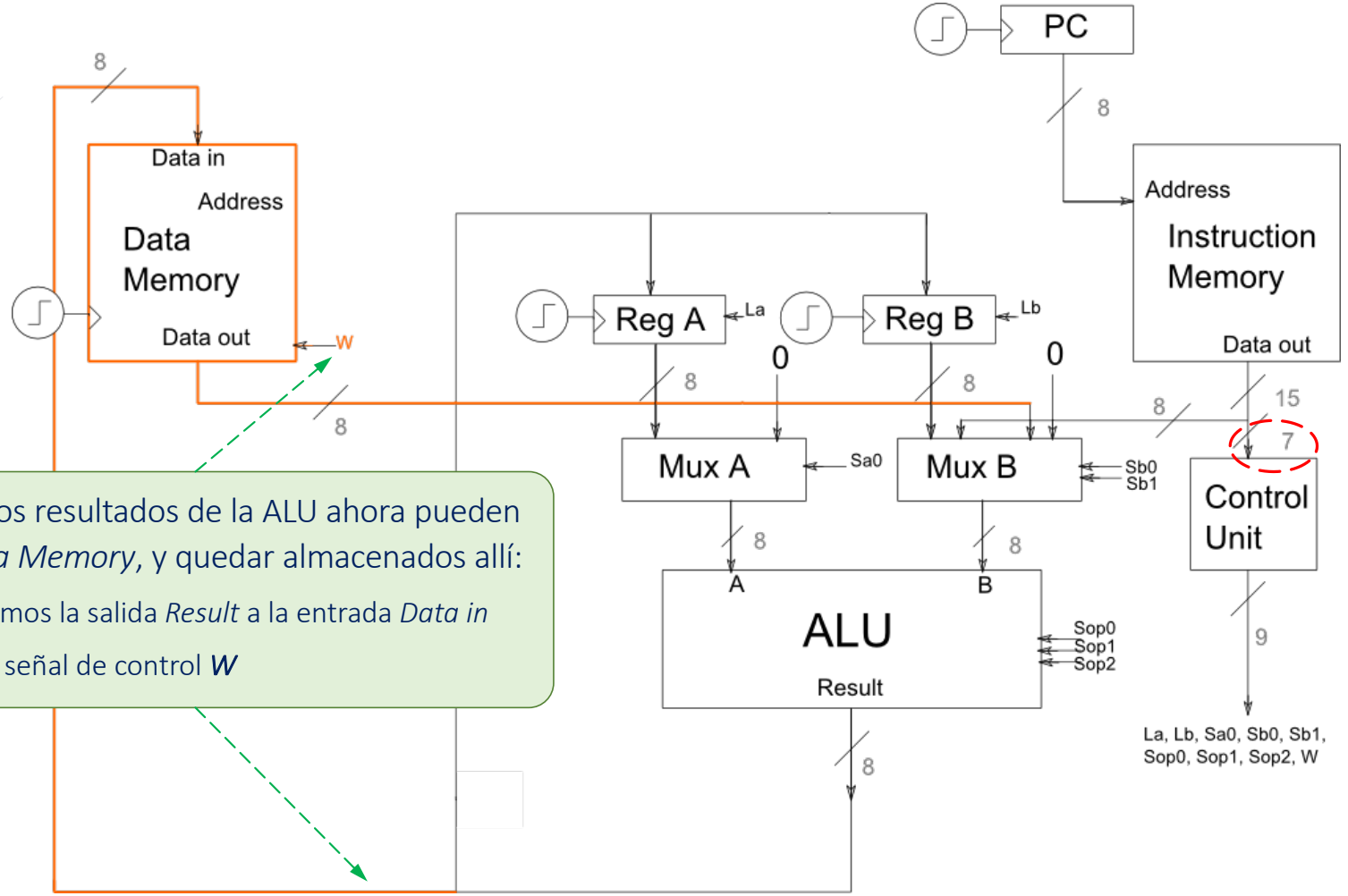
Cada variable (su valor, p.ej., ***Dato 1***) ocupa una palabra de memoria

… y la identificamos por un *label* (el "nombre" de la variable, p.ej., ***var1***), en vez de su dirección (**0x01**)

Dirección	Label	Instrucción/Dato
	DATA:	
0x00	var0	Dato 0
0x01	var1	Dato 1
0x02	var2	Dato 2
0x03		Dato 3
0x04		Dato 4
	CODE:	
0x00		Instrucción 0
0x01		Instrucción 1
0x02		Instrucción 2
0x03		Instrucción 3
0x04		Instrucción 4



Los datos de la *Data Memory* van desde la salida *Data out* a la entrada *B* de la ALU (a través del *Mux B*, tal como el registro *B*, los literales de las instrucciones o la constante 0)



Además, los resultados de la ALU ahora pueden ir a la *Data Memory*, y quedar almacenados allí:

- conectamos la salida *Result* a la entrada *Data in*
- notar la señal de control ***W***

Nuestro lenguaje *assembly* va a tener ahora **instrucciones adicionales** (*Instrucción* + *Operandos*, en la primera columna)

… similares a las que conocemos, pero que **operan sobre celdas de memoria de la *Data Memory***

- en la columna "*Operación*" el arreglo **`Mem`** representa la *Data Memory*

Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
MOV	A,(Dir)	A=Mem[Dir]		MOV A,(var1)
	B,(Dir)	B=Mem[Dir]		MOV B,(var2)
	(Dir),A	Mem[Dir]=A		MOV (var1),A
	(Dir),B	Mem[Dir]=B		MOV (var2),B
	A,(B)	A=Mem[B]		-
	B,(B)	B=Mem[B]		-
	(B),A	Mem[B]=A		-
ADD	A,(Dir)	A=A+Mem[Dir]		ADD A,(var1)
	A,(B)	A=A+Mem[B]		-
	(Dir)	Mem[Dir]=A+B		ADD (var1)
SUB	A,(Dir)	A=A-Mem[Dir]		SUB A,var1
	A,(B)	A=A-Mem[B]		-
	(Dir)	Mem[Dir]=A-B		SUB (var1)
AND	A,(Dir)	A=A and Mem[Dir]		AND A,(var1)
	A,(B)	A=A and Mem[B]		-
	(Dir)	Mem[Dir]=A and B		-
OR	A,(Dir)	A=A or Mem[Dir]		OR A,(var1)
	A,(B)	A=A or Mem[B]		-
	(Dir)	Mem[Dir]=A or B		OR (var1)
NOT	A,(Dir)	A=notMem[Dir]		NOT A,(var1)
	A,(B)	A=notMem[B]		-
	(Dir)	Mem[Dir]=not A		NOT (var1)
XOR	A,(Dir)	A=A xor Mem[Dir]		XOR A,(var1)
	A,(B)	A=A xor Mem[B]		-
	(Dir)	Mem[Dir]=A xor B		XOR (var1)
SHL	A,(Dir)	A=shift left Mem[Dir]		SHL A,(var1)
	A,(B)	A=shiflt left Mem[B]		-
	(Dir)	Mem[Dir]=shift left A		SHL (var1)
SHR	A,(Dir)	A=shift right Mem[Dir]		SHR A,(var1)
	A,(B)	A=shiflt right Mem[B]		-
	(Dir)	Mem[Dir]=shift right A		SHR(var1)
INC	B	B=B+1		-

… por lo que nos permiten manejar **variables** en los programas:

- son 32 instrucciones ***adicionales*** a las 28 que ya teníamos
- tienen *opcodes* ***distintos*** a los *opcodes* de esas 28 instrucciones (que sólo hacen referencia a los registros *A* y *B*)
- para acomodar estas nuevas instrucciones, y otras que ya vamos a ver, los *opcodes* tienen ahora 7 bits (desde la diap. 15)

Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
MOV	A,(Dir)	A=Mem[Dir]		MOV A,(var1)
	B,(Dir)	B=Mem[Dir]		MOV B,(var2)
	(Dir),A	Mem[Dir]=A		MOV (var1),A
	(Dir),B	Mem[Dir]=B		MOV (var2),B
	A,(B)	A=Mem[B]		-
	B,(B)	B=Mem[B]		-
	(B),A	Mem[B]=A		-
ADD	A,(Dir)	A=A+Mem[Dir]		ADD A,(var1)
	A,(B)	A=A+Mem[B]		-
	(Dir)	Mem[Dir]=A+B		ADD (var1)
SUB	A,(Dir)	A=A-Mem[Dir]		SUB A,var1
	A,(B)	A=A-Mem[B]		-
	(Dir)	Mem[Dir]=A-B		SUB (var1)
AND	A,(Dir)	A=A and Mem[Dir]		AND A,(var1)
	A,(B)	A=A and Mem[B]		-
	(Dir)	Mem[Dir]=A and B		-
OR	A,(Dir)	A=A or Mem[Dir]		OR A,(var1)
	A,(B)	A=A or Mem[B]		-
	(Dir)	Mem[Dir]=A or B		OR (var1)
NOT	A,(Dir)	A=notMem[Dir]		NOT A,(var1)
	A,(B)	A=notMem[B]		-
	(Dir)	Mem[Dir]=not A		NOT (var1)
XOR	A,(Dir)	A=A xor Mem[Dir]		XOR A,(var1)
	A,(B)	A=A xor Mem[B]		-
	(Dir)	Mem[Dir]=A xor B		XOR (var1)
SHL	A,(Dir)	A=shift left Mem[Dir]		SHL A,(var1)
	A,(B)	A=shiflt left Mem[B]		-
	(Dir)	Mem[Dir]=shift left A		SHL (var1)
SHR	A,(Dir)	A=shift right Mem[Dir]		SHR A,(var1)
	A,(B)	A=shiflt right Mem[B]		-
	(Dir)	Mem[Dir]=shift right A		SHR(var1)
INC	B	B=B+1		-

Algunas instrucciones hacen referencia explícitamente a la variable —*direccionamiento directo*:

- la variable se especifica como **(** *variable* **)** —la instrucción de máquina usa los 8 bits del literal para la *variable*
- p.ej., **MOV A,(var1)** significa guardar en el registro *A* el contenido de la dirección de memoria (es decir, el valor) de la variable **var1**

En cambio, en ***direccionamiento indirecto***, la variable es especificada a través del registro *B*

… ⟹ en el registro *B* está almacenada la *dirección de memoria* de la variable que nos interesa:

- el operando se especifica como **`(B)`**
- p.ej., **`MOV A,(B)`** significa guardar en el registro *A* el contenido de la celda de memoria (de la *Data Memory*) cuya dirección está almacenada en el registro *B*

Algunas instrucciones hacen referencia explícitamente a la variable —***direccionamiento directo***:

- la variable se especifica como **(** *variable* **)** —la instrucción de máquina usa los 8 bits del literal para la *variable**
- p.ej., **`MOV A,(var1)`** significa guardar en el registro *A* el contenido de la dirección de memoria (es decir, el valor) de la variable **`var1`**

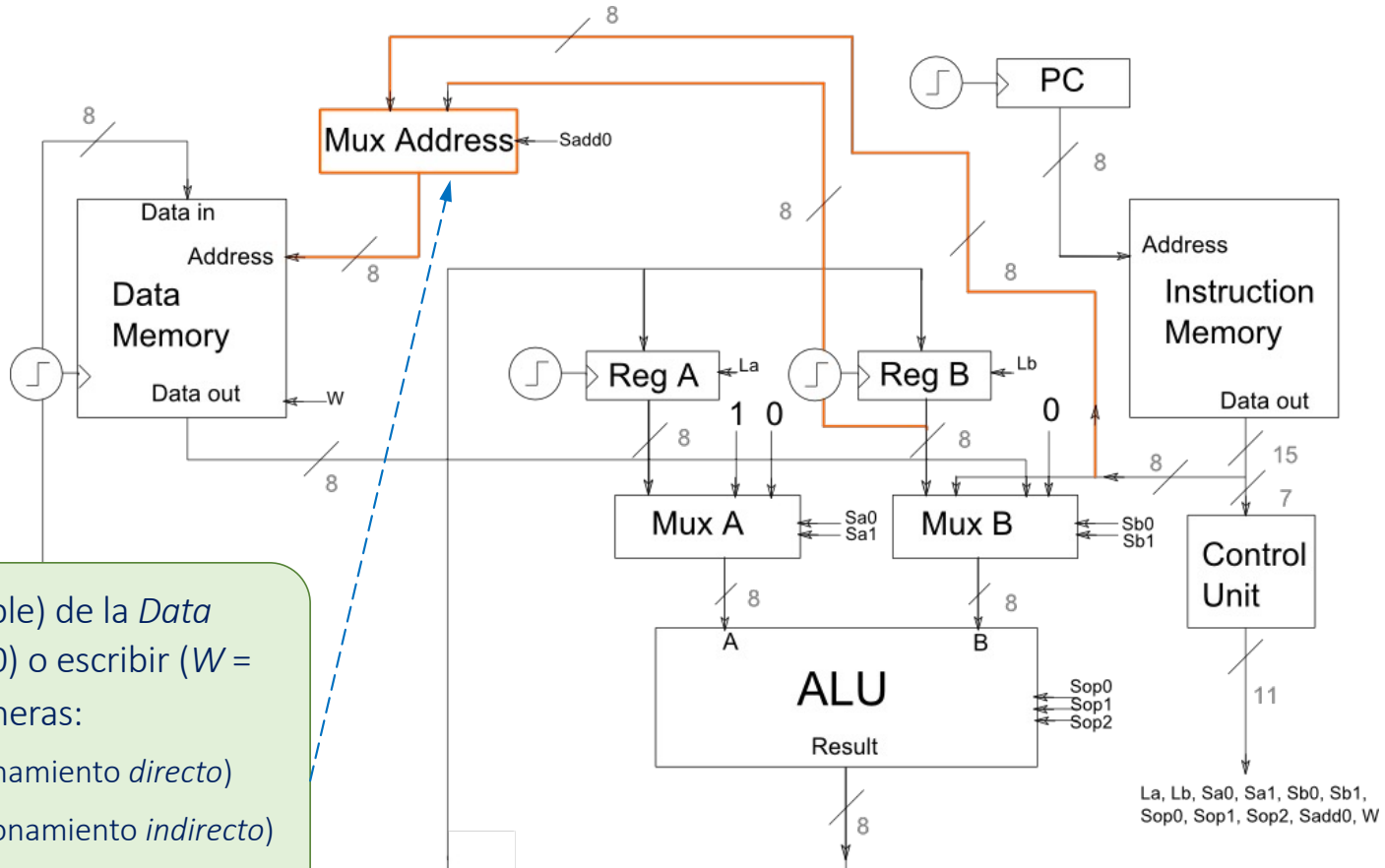
*Así, a partir de ahora, el literal tiene tres interpretaciones posibles:

- un número entero entre –128 y +127, en las instrucciones aritméticas
- simplemente un patrón de 8 bits, en las instrucciones lógicas y *shifts*
- un número entero sin signo entre 0 y 255, correspondiente a una dirección de memoria (de la *Data Memory*), en las instrucciones que hacen referencia a una variable del programa

La dirección de la palabra (la variable) de la *Data Memory* que queremos leer (*W* = 0) o escribir (*W* = 1) se puede especificar de dos maneras:

- el literal de la instrucción (direccionamiento *directo*)
- el contenido del registro *B* (direccionamiento *indirecto*)

⟹ colocamos un multiplexor, *Mux Addess*, antes de la entrada *Address* de la *Data Memory*



En las instrucciones con direccionamiento directo uno de los operandos es el nombre de la variable, entre **(** y **)** —ver columna "*Ejemplo de uso*"

… pero la instrucción correspondiente en lenguaje de máquina usa la dirección de la variable en la *Data Memory*:

- el arreglo **Mem** —ver columna "*Operación*"— representa el arreglo de celdas de memoria de la *Data Memory*

P.ej., la próx. diap. muestra dos versiones de un programa que suma los valores de dos variables, ***a*** y ***b***

Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
MOV	A,(Dir)	A=Mem[Dir]		MOV A,(var1)
	B,(Dir)	B=Mem[Dir]		MOV B,(var2)
	(Dir),A	Mem[Dir]=A		MOV (var1),A
	(Dir),B	Mem[Dir]=B		MOV (var2),B
	A,(B)	A=Mem[B]		-
	B,(B)	B=Mem[B]		-
	(B),A	Mem[B]=A		-
ADD	A,(Dir)	A=A+Mem[Dir]		ADD A,(var1)
	A,(B)	A=A+Mem[B]		-
	(Dir)	Mem[Dir]=A+B		ADD (var1)
SUB	A,(Dir)	A=A-Mem[Dir]		SUB A,var1
	A,(B)	A=A-Mem[B]		-
	(Dir)	Mem[Dir]=A-B		SUB (var1)
AND	A,(Dir)	A=A and Mem[Dir]		AND A,(var1)
	A,(B)	A=A and Mem[B]		-
	(Dir)	Mem[Dir]=A and B		-
OR	A,(Dir)	A=A or Mem[Dir]		OR A,(var1)
	A,(B)	A=A or Mem[B]		-
	(Dir)	Mem[Dir]=A or B		OR (var1)
NOT	A,(Dir)	A=notMem[Dir]		NOT A,(var1)
	A,(B)	A=notMem[B]		-
	(Dir)	Mem[Dir]=not A		NOT (var1)
XOR	A,(Dir)	A=A xor Mem[Dir]		XOR A,(var1)
	A,(B)	A=A xor Mem[B]		-
	(Dir)	Mem[Dir]=A xor B		XOR (var1)
SHL	A,(Dir)	A=shift left Mem[Dir]		SHL A,(var1)
	A,(B)	A=shiflt left Mem[B]		-
	(Dir)	Mem[Dir]=shift left A		SHL (var1)
SHR	A,(Dir)	A=shift right Mem[Dir]		SHR A,(var1)
	A,(B)	A=shiflt right Mem[B]		-
	(Dir)	Mem[Dir]=shift right A		SHR(var1)
INC	B	B=B+1		-

Versión 1:

- los sumandos ***a*** y ***b*** son cargados* en los registros *A* y *B*, desde la *Data Memory*
- luego se ejecuta la instrucción **ADD A,B**, que suma ambos registros y deja el resultado en *A*
- finalmente, el contenido de *A* es almacenado en la variable ***suma*** en la *Data Memory*

Versión 2:

- una vez que los sumandos están en los registros, se ejecuta la instrucción **ADD (suma)**, que suma ambos registros y almacena directamente el resultado en la variable ***suma***

* Cuando un valor se copia de la memoria de datos a un registro, se llama **cargar** (*load*) el registro; y cuando un valor se copia de un registro a la memoria, se llama **almacenar** (*store*) en la variable (dirección de memoria)

```
DATA:
a       17
b       54
suma    —no inicializado

CODE_1: —versión 1
        MOV A,(a)
        MOV B,(b)
        ADD A,B
        MOV (suma),A

CODE_2: —versión 2
        MOV A,(a)
        MOV B,(b)
        ADD (suma)
```

La “magia” de convertir los nombres de las variables (o *labels*) a las direcciones que esas variables ocupan en la *Data Memory* es responsabilidad de los programas llamados *assembler* y *loader*:

- p.ej., en la fig., las variables *var0*, *var1* y *var2* corresponden a las direcciones 0x00, 0x01 y 0x02

  … los valores de esas variables son *Dato 0*, *Dato 1* y *Dato 2*

Dirección	Label	Instrucción/Dato
	DATA:	
0x00	var0	Dato 0
0x01	var1	Dato 1
0x02	var2	Dato 2
0x03		Dato 3
0x04		Dato 4
	CODE:	
0x00		Instrucción 0
0x01		Instrucción 1
0x02		Instrucción 2
0x03		Instrucción 3
0x04		Instrucción 4

Aún no podemos escribir programas con instrucciones **`if`** o **`while`**:

- como sabemos (de Python, Java, C, etc.), en ambos casos, el orden de ejecución de las instrucciones no sigue el orden en que están escritas

  … sino que depende del resultado de la evaluación de una condición

Aún no podemos escribir programas con instrucciones **`if`** o **`while`**:

- como sabemos (de Python, Java, C, etc.), en ambos casos, el orden de ejecución de las instrucciones no sigue el orden en que están escritas … sino que depende del resultado de la evaluación de una condición

Necesitamos 1) instrucciones que comparen dos valores entre sí, y de alguna manera codifiquen el resultado de la comparación:

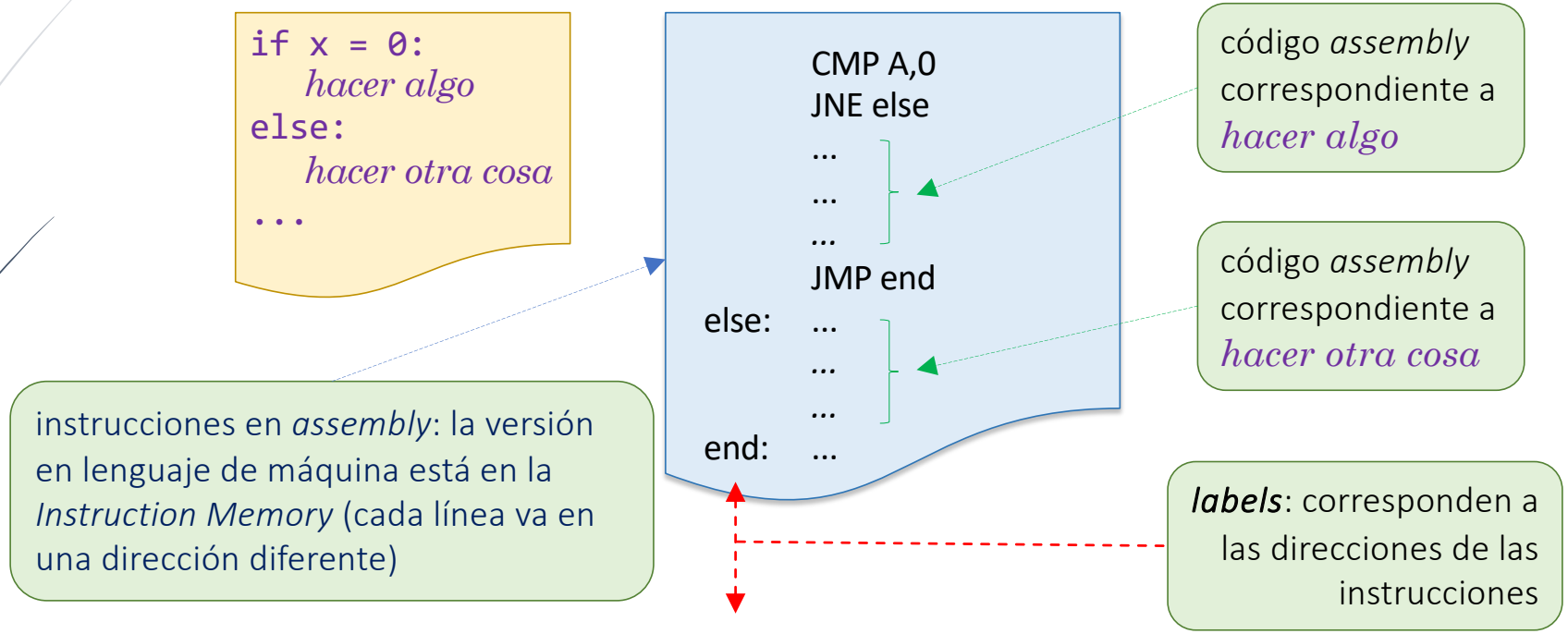
- p.ej., **`CMP A,B`** compara los contenidos de los registros *A* y *B*

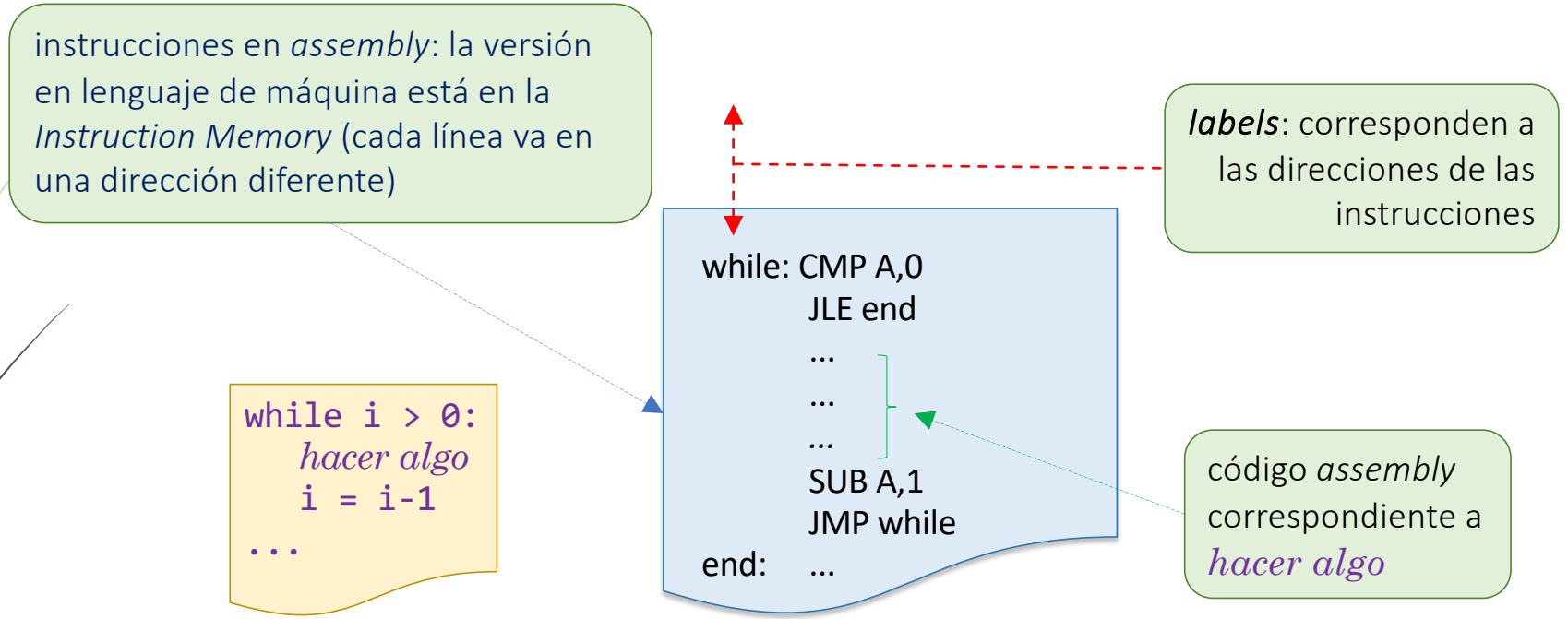
Necesitamos **1**) instrucciones que comparen dos valores entre sí, y de alguna manera codifiquen el resultado de la comparación:

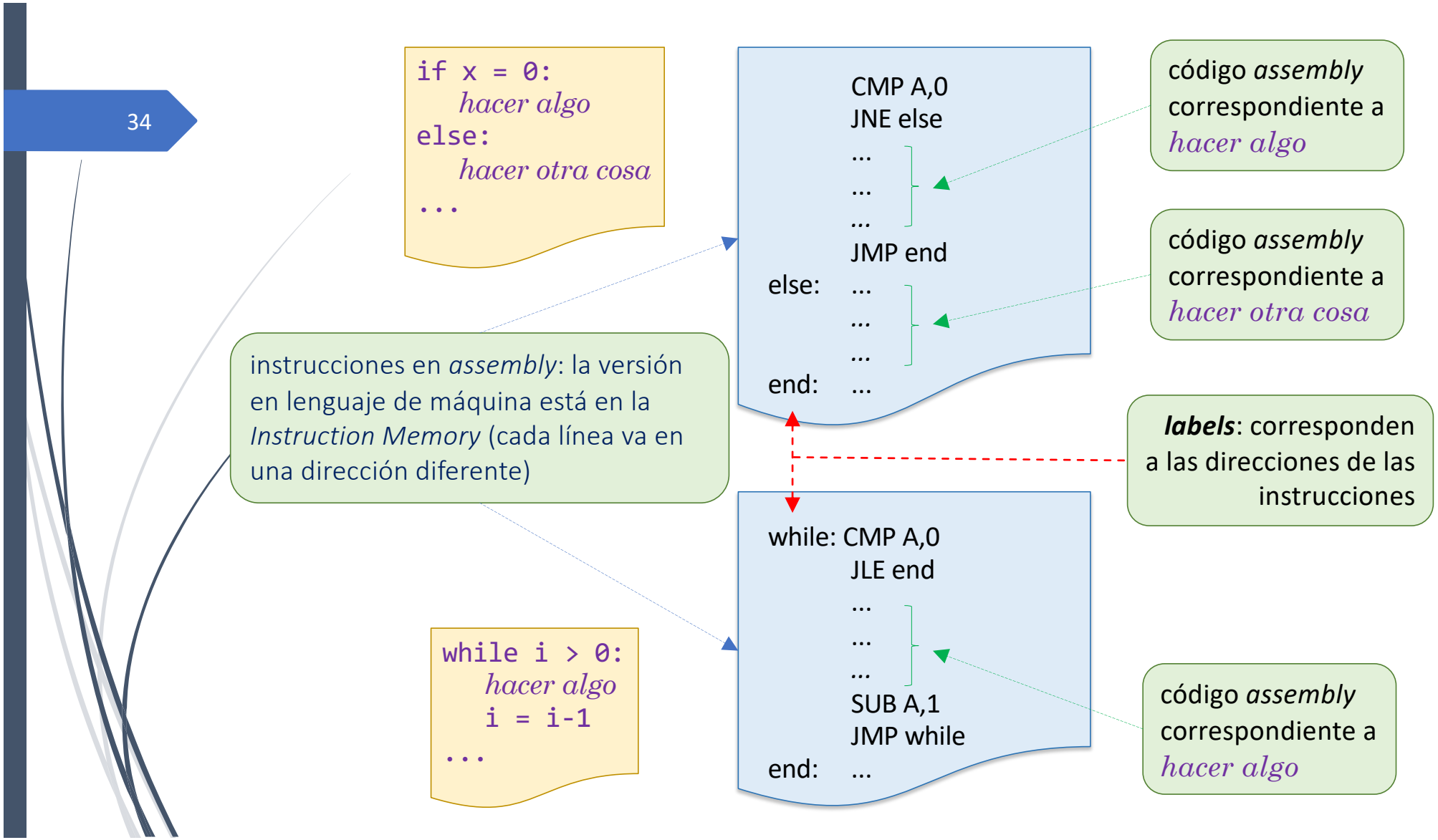
- p.ej., **`CMP A,B`** compara los contenidos de los registros *A* y *B*

… y **2**) instrucciones que especifiquen la dirección en la *Instruction Memory* de la próxima instrucción que debe ser ejecutada; p.ej.:

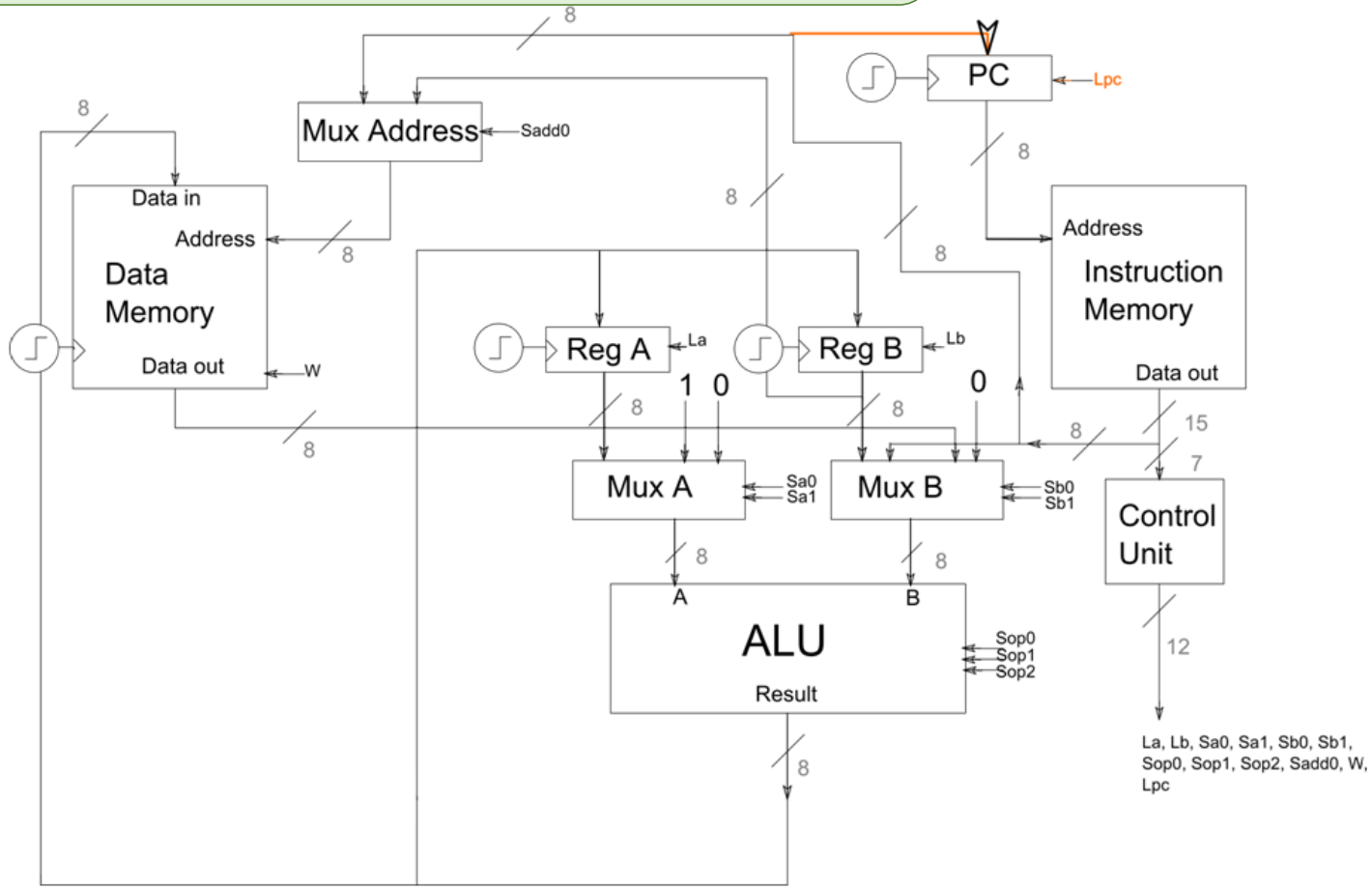
- **`JMP end`** "salta" directamente a ejecutar la instrucción identificada con el *label* (etiqueta) "**end**" —**saltos incondicionales**
- **`JLE end`** "salta" a ejecutar la instrucción etiquetada "**end**" sólo si el resultado de la comparación ejecutada justo antes fue "menor o igual" (*less than or equal to*) —**saltos condicionales**


if x = 0:
hacer algo
else:
hacer otra cosa
...
CMP A,0
JNE else
...
...
...
JMP end
else: ...
...
...
end: ...
código assembly correspondiente a hacer algo
código assembly correspondiente a hacer otra cosa
instrucciones en assembly: la versión en lenguaje de máquina está en la Instruction Memory (cada línea va en una dirección diferente)
labels: corresponden a las direcciones de las instrucciones


instrucciones en *assembly*: la versión en lenguaje de máquina está en la *Instruction Memory* (cada línea va en una dirección diferente)
labels: corresponden a las direcciones de las instrucciones
while i > 0:
hacer algo
i = i-1
...
while: CMP A,0
JLE end
...
...
...
SUB A,1
JMP while
end: ...
código *assembly* correspondiente a *hacer algo*

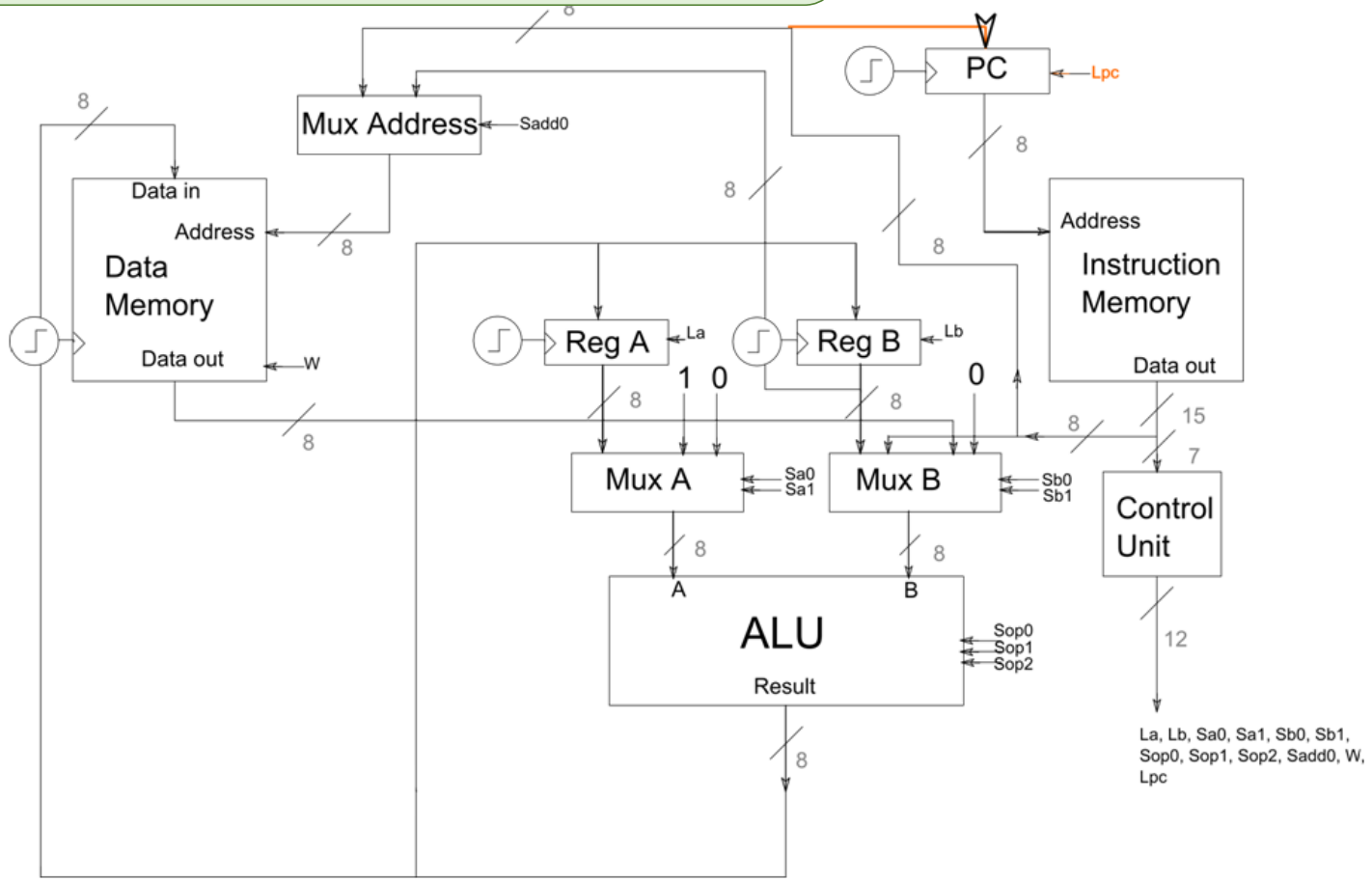

if x = 0:
hacer algo
else:
hacer otra cosa
...
CMP A,0
JNE else
...
...
...
JMP end
else: ...
...
...
end: ...
código assembly correspondiente a hacer algo
código assembly correspondiente a hacer otra cosa
instrucciones en assembly: la versión en lenguaje de máquina está en la Instruction Memory (cada línea va en una dirección diferente)
labels: corresponden a las direcciones de las instrucciones
while i > 0:
hacer algo
i = i-1
...
while: CMP A,0
JLE end
...
...
...
SUB A,1
JMP while
end: ...
código assembly correspondiente a hacer algo

Para permitir **saltos incondicionales** (**JMP**) hay que poder poner en el *PC* la dirección de la instrucción que queremos ejecutar a continuación: conectamos la salida *Data out* de la *Instruction Memory* al registro ***PC*** …

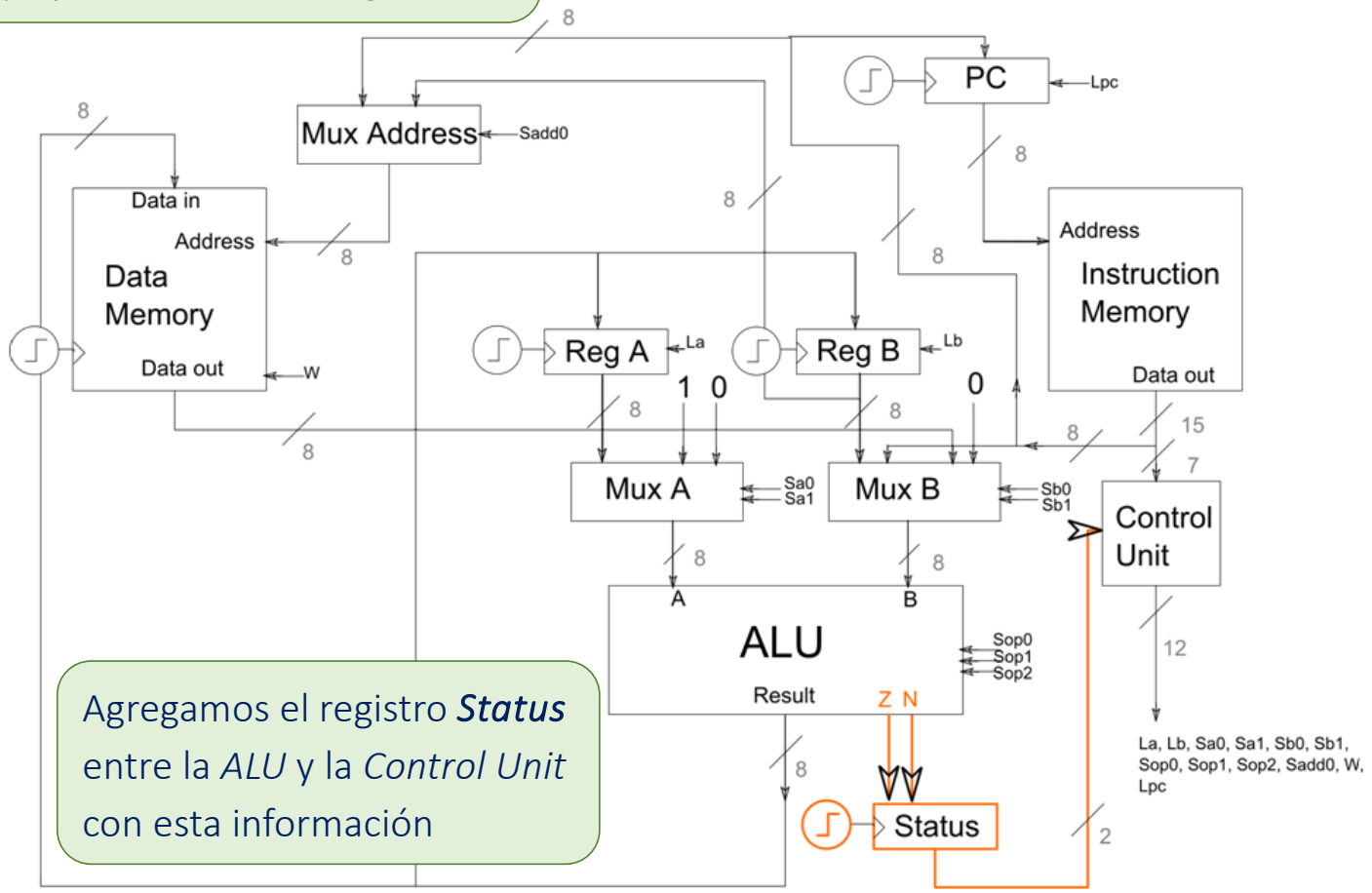


… y la nueva señal de control L_{PC} indica cómo es actualizado el ***PC***:

- ya sea con la dirección que viene desde la *Instruction Memory*
- … o simplemente sumándole 1



Para **saltos condicionales** (**`JNE`**, **`JLE`**), la *Control Unit* tiene que examinar el resultado de la *ALU* justo después de hecha la comparación; p.ej., si es 0 o si es negativo



Agregamos el registro ***Status*** entre la *ALU* y la *Control Unit* con esta información

Ocho nuevas instrucciones de nuestro lenguaje assembly (**que ya totaliza 69 instrucciones ⇒ opcodes de 7 bits**):

- **`CMP`** toma dos operandos y los compara por la vía de restarlos

  … el resultado queda codificado en los bits ***N*** (¿es negativo?) y ***Z*** (¿es cero?) del registro ***Status***

- las instrucciones **`Jxx`** tienen un solo operando: la dirección de una instrucción en la *Instruction Memory* (especificada por una etiqueta, o *label*)

  … esa dirección (**`Dir`**) es escrita en el registro ***PC*** si y sólo si se cumple la condición codificada en términos de los bits ***N*** y/o ***Z***

Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
CMP	A,B	A-B		
	A,Lit	A-Lit		CMP A,0
JEQ	Dir	PC = Dir	Z=1	JEQ label
JNE	Dir	PC = Dir	Z=0	JNE label
JGT	Dir	PC = Dir	N=0 y Z=0	JGT label
JLT	Dir	PC = Dir	N=1	JLT label
JGE	Dir	PC = Dir	N=0	JGE label
JLE	Dir	PC = Dir	Z=1 o N=1	JLE label

Con las 69 instrucciones vistas, nuestro assembly nos permite escribir programas como los siguientes:

… multiplicar y dividir números enteros (con y sin signo)

… procesar todos los elementos de un arreglo de números enteros:

- p.ej., aplicarle la misma operación a cada número

  … o sumar todos los números entre sí

Cada uno de estos programas se puede escribir de varias maneras:

- tal como ocurre al escribir programas en un lenguaje de programación de alto nivel

Multiplicación de dos números no negativos por la vía de sumar repetidamente el *multiplicando*, tantas veces como lo indica el *multiplicador*

Primero, se inicializa en 0 la variable **prod**, que va a almacenar el resultado de la multiplicación:

- no hay una instrucción para asignar directamente el literal 0 a una variable
- → primero el registro *A* es cargado con 0

  … y luego el contenido de *A* es almacenado en **prod**

Luego, el registro *B* es cargado con 0 —representa, a lo largo de todo el programa, el número de repeticiones realizadas

… y el registro *A*, con el valor del multiplicador —el número de repeticiones que hay que realizar:

- el registro *A* toma diferentes roles a lo largo del programa, a diferencia del *B*

```
DATA:
a     15          —multiplicando
b     7           —multiplicador
prod        —producto, no inicializado

CODE:             —multiplicación, sin signo
      MOV A,0
      MOV (prod),A  —inicializamos prod = 0
      MOV B,0       —B va a ser el contador de repeticiones
      MOV A,(b)     —A = número de repeticiones (multiplicador)
loop: CMP A,B           —comparamos
      JEQ end       —si son iguales, terminamos (saltamos a end)
      MOV A,(a)     —tomamos el multiplicando
      ADD A,(prod)  —se lo sumamos una vez más a prod
      MOV (prod),A  —y actualizamos el valor de prod
      INC B         —incrementamos el contador de repeticiones
      MOV A,(b)     —A = número de repeticiones (multiplicador)
      JMP loop      —repetimos el ciclo
end:
```

Ahora comienza el ciclo que se repite (label **loop**)

Primero, se compara el número de repeticiones que hay que realizar (registro *A*) con el número de repeticiones realizadas (registro *B*)

… si son iguales, entonces el programa termina (**JEQ end**)

… de lo contrario, se ejecuta una repetición —se suma una vez el multiplicando a la variable **prod**:

- se carga el multiplicando en el registro *A*, se le suma lo que ya está acumulado en la variable **prod**, y con el resultado se actualiza la variable **prod**
- se incrementa el contador de repeticiones —el registro *B*
- se carga el multiplicador en el registro *A*
- … y se inicia una nueva repetición

```
DATA:
a    15          —multiplicando
b    7           —multiplicador
prod         —producto, no inicializado

CODE:            —multiplicación, sin signo
     MOV A,0
     MOV (prod),A  —inicializamos prod = 0
     MOV B,0       —B va a ser el contador de repeticiones
     MOV A,(b)     —A = número de repeticiones (multiplicador)
loop:  CMP A,B          —comparamos
     JEQ end       —si son iguales, terminamos (saltamos a end)
     MOV A,(a)     —tomamos el multiplicando
     ADD A,(prod)  —se lo sumamos una vez más a prod
     MOV (prod),A  —y actualizamos el valor de prod
     INC B         —incrementamos el contador de repeticiones
     MOV A,(b)     —A = número de repeticiones (multiplicador)
     JMP loop      —repetimos el ciclo
end:
```