



**DCC**  
DEPARTAMENTO DE CIENCIA  
DE LA COMPUTACIÓN

**IIC2343**

# **Arquitectura de Computadores**

**Clase 4 - Programabilidad**

**Profesor: Germán Leandro Contreras Sagredo**

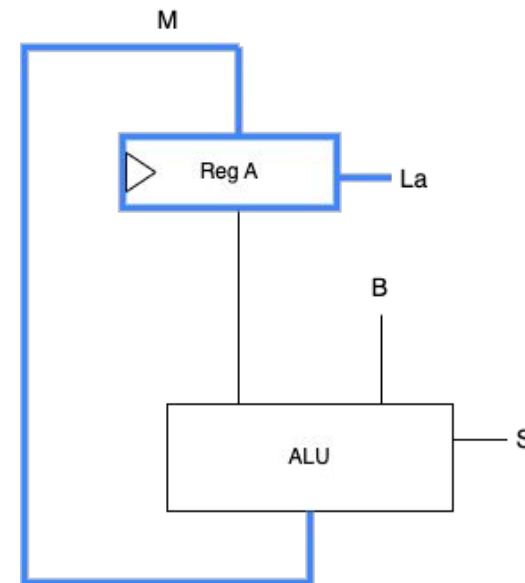
## Objetivos de la clase

- Construir una máquina cuya combinación de señales permita que sea programable.
- Conocer nuevos métodos y componentes que simplifiquen la programabilidad de la máquina.
- Realizar ejercicios que consoliden los conocimientos anteriores.

## Hasta ahora...

Ahora tenemos un esquema en el que sí tenemos una calculadora, cuyo resultado puede o no ser almacenado en el registro *A*.

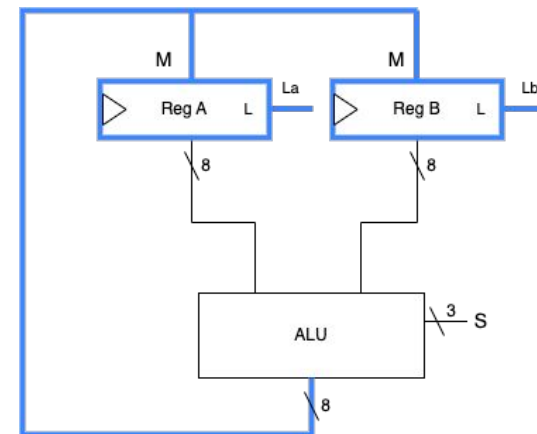
¿Podemos extenderlo a *B*?



## ALU extendida

Incluimos un nuevo registro y, con él, una nueva señal de carga  $Lb$ .

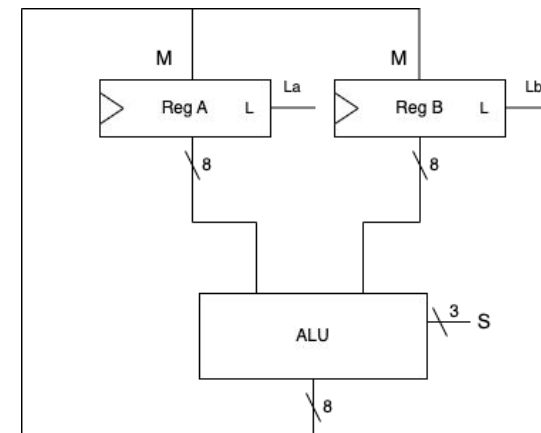
Por otro lado, establecemos la cantidad de bits por bus de datos. Contamos con dos buses de entrada de 8 bits ( $A$ ,  $B$ ), señales de carga  $La$  y  $Lb$  de un bit y un bus de selección de operación de la ALU  $S$  de 3 bits.



A continuación veremos las operaciones que podemos realizar con este esquema.

## ALU extendida - Operaciones

<i>La</i>	<i>Lb</i>	<i>S[2]</i>	<i>S[1]</i>	<i>S[0]</i>	Operación
1	0	0	0	0	$A = A + B$
0	1	0	0	0	$B = A + B$
1	0	0	0	1	$A = A - B$
0	1	0	0	1	$B = A - B$
1	0	0	1	0	$A = A \text{ AND } B$
0	1	0	1	0	$B = A \text{ AND } B$
1	0	0	1	1	$A = A \text{ OR } B$
0	1	0	1	1	$B = A \text{ OR } B$
1	0	1	0	0	$A = \text{NOT } A$
0	1	1	0	0	$B = \text{NOT } A$
1	0	1	0	1	$A = A \text{ XOR } B$
0	1	1	0	1	$B = A \text{ XOR } B$
1	0	1	1	0	$A = \text{SHL } A$ (SHL = Shift Left)
0	1	1	1	0	$B = \text{SHL } A$
1	0	1	1	1	$A = \text{SHR } A$ (SHL = Shift Right)
0	1	1	1	1	$B = \text{SHR } A$



Cada “palabra” de control (combinación de señales de control) representa una operación o una **instrucción**. Una secuencia de instrucciones es lo que conocemos como **programa**.

## Instrucciones y programas - Ejemplo

<i>La</i>	<i>Lb</i>	<i>S[2]</i>	<i>S[1]</i>	<i>S[0]</i>	<i>A</i>	<i>B</i>	Operación
1	0	0	0	0	0	1	-
1	0	0	0	0	?	?	?
0	1	0	0	0	?	?	?
1	0	0	0	0	?	?	?
0	1	0	0	0	?	?	?
1	0	0	0	0	?	?	?
0	1	0	0	0	?	?	?

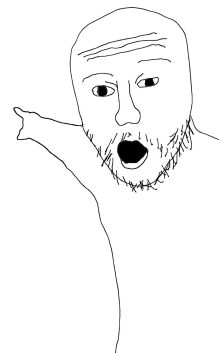
Asumamos que tenemos la siguiente secuencia de instrucciones y estados iniciales  $A = 0$  y  $B = 1$ . ¿Qué es lo que hace este programa?

## Instrucciones y programas - Ejemplo

<i>La</i>	<i>Lb</i>	<i>S[2]</i>	<i>S[1]</i>	<i>S[0]</i>	<i>A</i>	<i>B</i>	Operación
1	0	0	0	0	0	1	-
1	0	0	0	0	1	1	$A = A + B$
0	1	0	0	0	1	2	$B = A + B$
1	0	0	0	0	3	2	$A = A + B$
0	1	0	0	0	3	5	$B = A + B$
1	0	0	0	0	8	5	$A = A + B$
0	1	0	0	0	8	13	$B = A + B$

El programa anterior genera **los primeros 8 números de la secuencia de Fibonacci**.

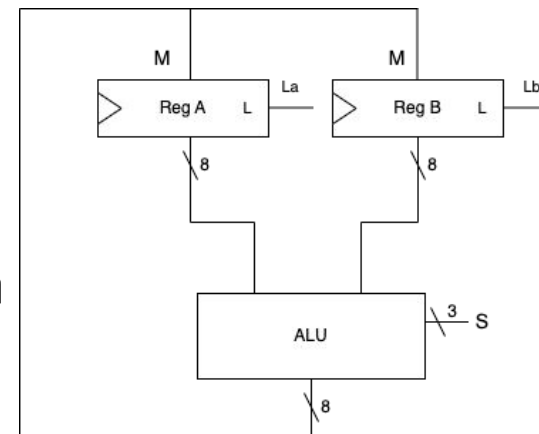
Con nuestra máquina **actual**, esta es la única forma de obtener la secuencia.



## Almacenamiento de instrucciones

Para que nuestra máquina sea “programable”, debe poder ejecutar una secuencia de instrucciones.

- ¿Cómo almacenamos las instrucciones de un programa?
- ¿Cómo ejecutamos cada instrucción del programa en orden?

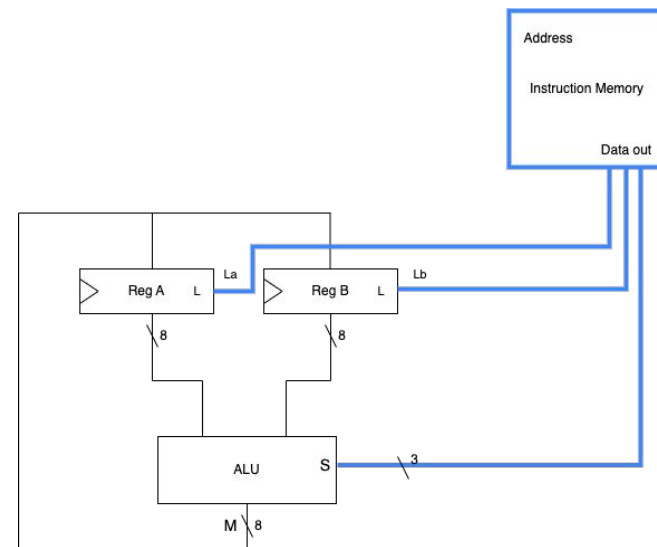




## Almacenamiento de instrucciones - ROM

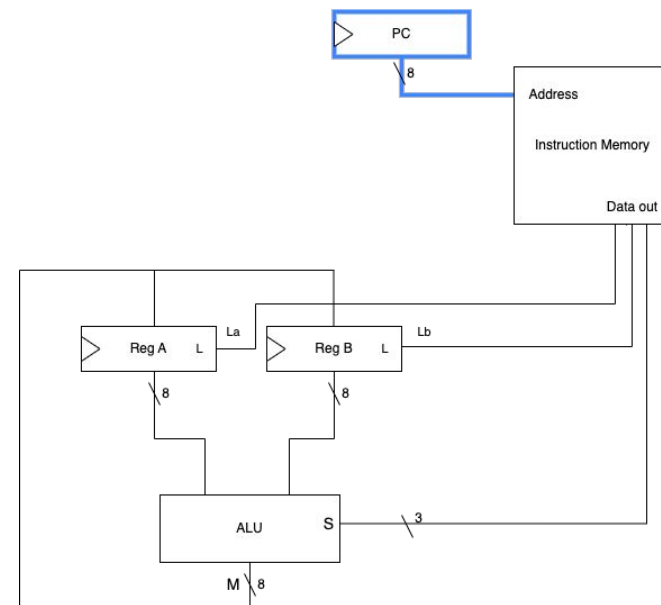
Como no nos interesa modificar ninguna instrucción de la secuencia mientras ejecutamos el programa, podemos almacenar todas las instrucciones en una memoria ROM (*Instruction Memory*). Cada instrucción (equivalente a una palabra de memoria) se almacena en una dirección de memoria distinta.

¿Cómo ejecutamos la secuencia en orden?



## Almacenamiento de instrucciones - PC

Para la entrada de direccionamiento, ingresamos la salida de un **contador** llamado *Program Counter* (PC) que siempre tendrá su señal de incremento **activa**. De esta manera, nos aseguramos de que se recorra cada instrucción de la memoria por cada iteración (o flanco de subida).

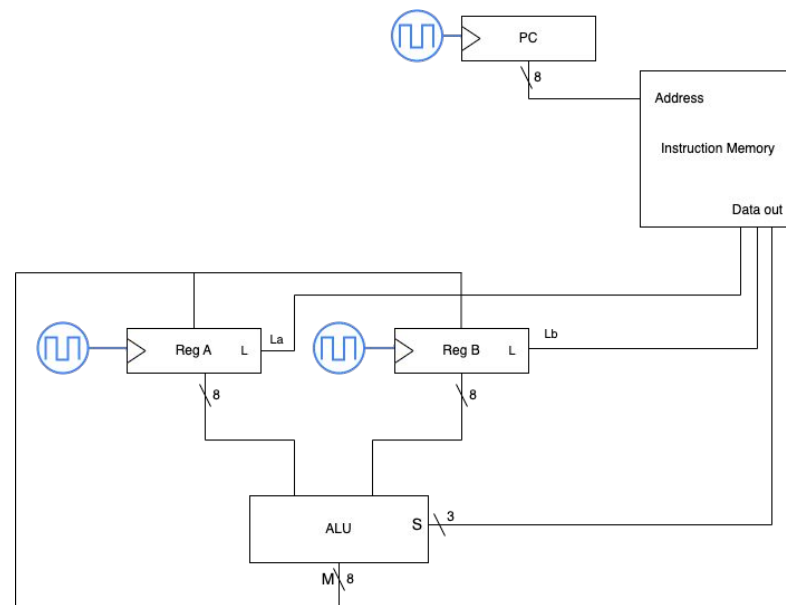


¿Cómo nos aseguramos de que los registros y el programa estén en sincronía?

## Almacenamiento de instrucciones - *Clock*

Cada registro, contador o unidad de almacenamiento que requiera de un flanco de subida estará conectado a **una única señal *clock***. De esta forma, el flanco de subida de esta señal permitirá el almacenamiento de datos en estas componentes de forma **simultánea**.

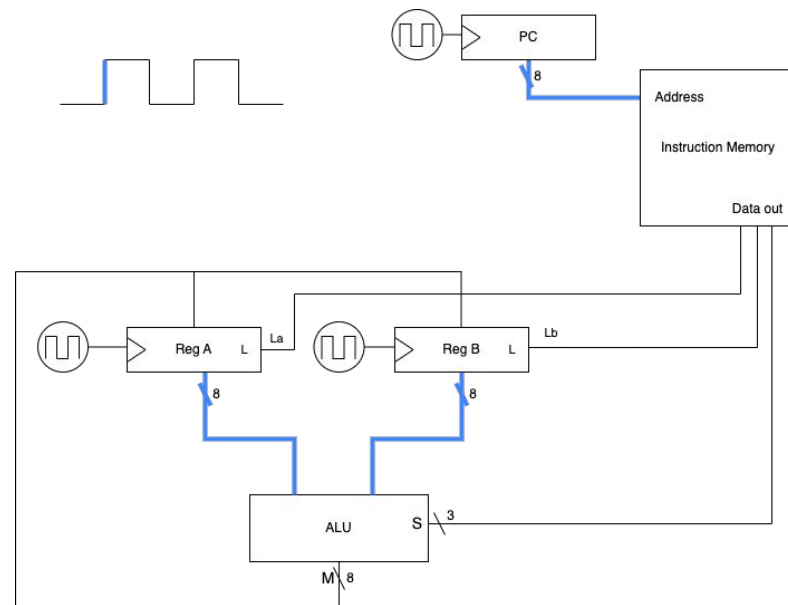
A continuación, veremos un ejemplo del funcionamiento completo.



## Ejecución de la máquina programable - Paso 1

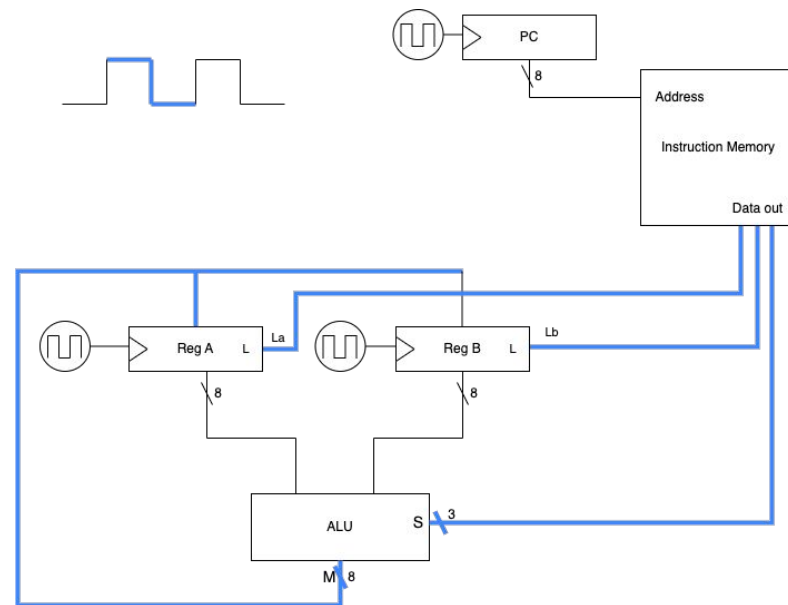
En el primer flanco de subida, se transmiten los valores de los registros a la ALU y a la memoria de instrucciones.

Asumimos, en primera instancia, que los registros no actualizan su valor.



## Ejecución de la máquina programable - Paso 2

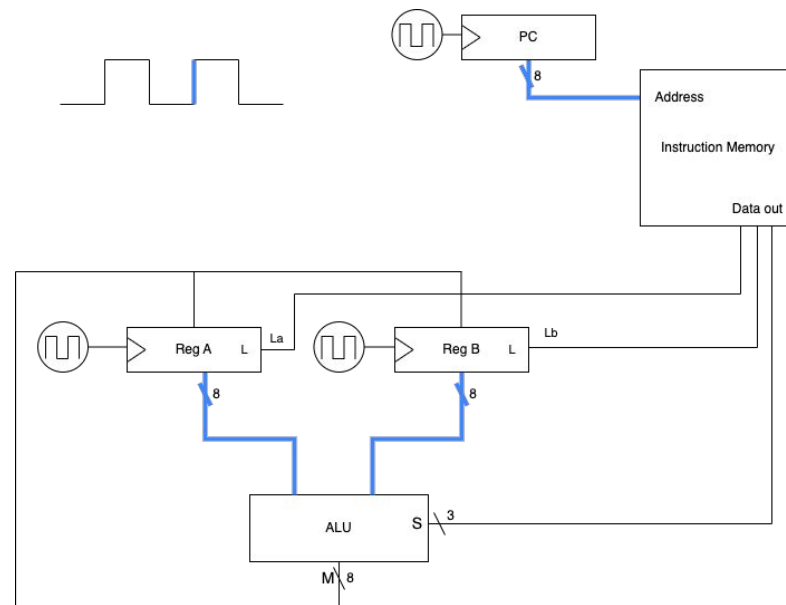
En el resto de la iteración, tanto las señales de control de la memoria de instrucciones como el resultado de la ALU se transmiten al resto de los componentes. Cuando el circuito se estabiliza, el resultado de la operación de la primera instrucción estará esperando a ser almacenado en los registros *A* o *B*, según corresponda.



## Ejecución de la máquina programable - Paso 3

En el siguiente flanco de subida, el registro *A* y/o el registro *B* actualizarán su valor con el resultado de la ALU.

Por otra parte, el registro PC incrementará en una unidad su valor y, de esta forma, transmitirá las señales de control de la siguiente instrucción.



## Ejecución de la máquina programable - Un último comentario

Como todos los componentes de almacenamiento se conectan a un único *clock* del sistema, es importante que su periodo **sea igual o mayor al tiempo de ejecución de la instrucción de máximo retraso de propagación de las instrucciones disponibles.**

Si no fuera así, alguno(s) de los registros no alcanzaría(n) a actualizar su valor y el resultado del programa no sería el esperado. La acción de aumentar la frecuencia del *clock* del sistema por **sobre** lo recomendado es lo que se conoce como **overclocking** (ejecución más rápida de nuestros programas, pero sin confiabilidad sobre el resultado final).

## Computador básico

Si bien ya podemos ejecutar programas simples:

- No podemos inicializar los registros con valores iniciales. Hasta ahora asumimos que parten con un valor.
- No podemos almacenar más de dos datos simultáneamente.
- Estamos **muy limitados** respecto a los programas que podemos ejecutar.

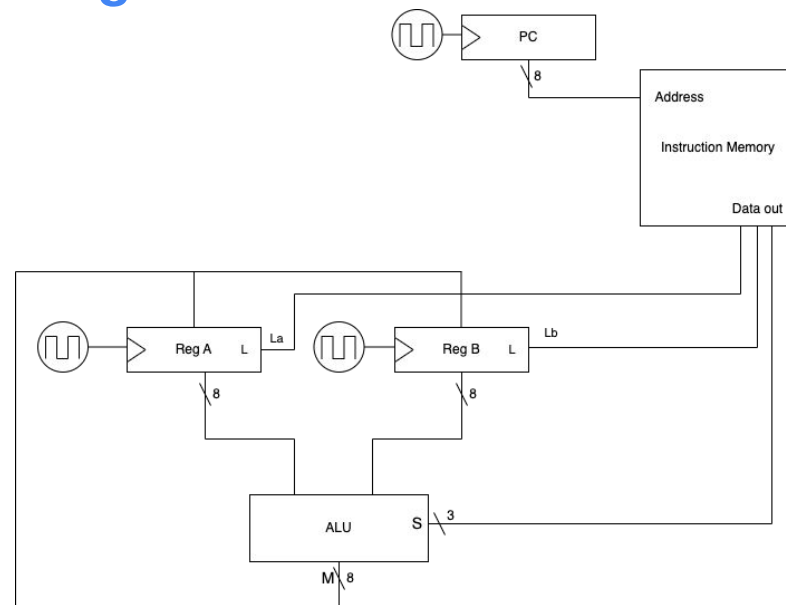




## Computador básico - Inicialización de registros

Lo primero que queremos es poder inicializar los valores de nuestros registros.

Para ello, ahora cada instrucción de la memoria de instrucciones incluirá un **literal**: valor numérico que podemos transmitir al resto de nuestras componentes.

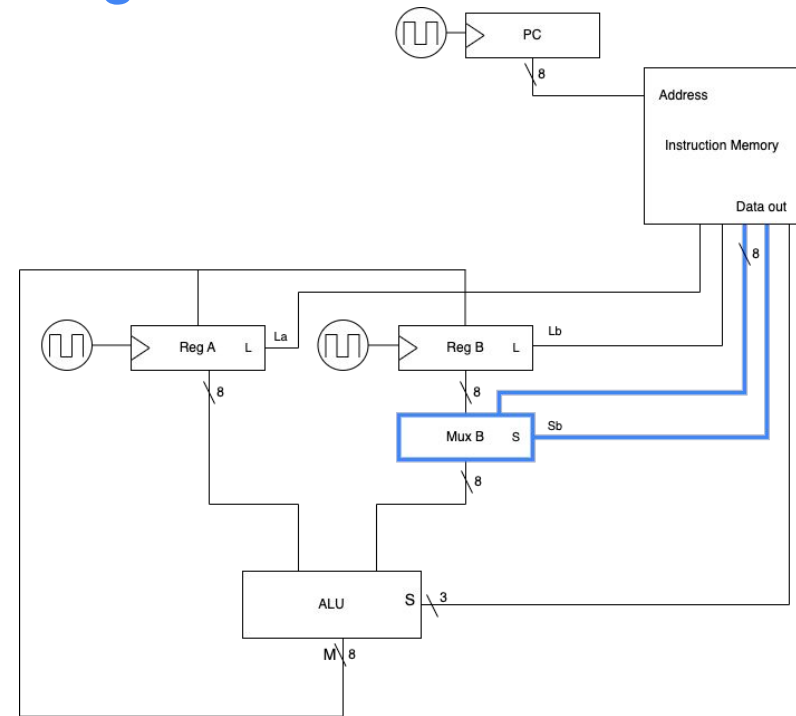


## Computador básico - Inicialización de registros

Con el nuevo componente Mux  $B$  y la señal de control  $Sb$ , podemos seleccionar el valor del registro  $B$  o el literal asociado a la instrucción para ingresar a la ALU.

Esto habilita nuevas operaciones para  $A$ , por ejemplo:  **$A + 5$**   
( $La = 1$ ;  $Sb = 1$ ;  $S = 000$ ).

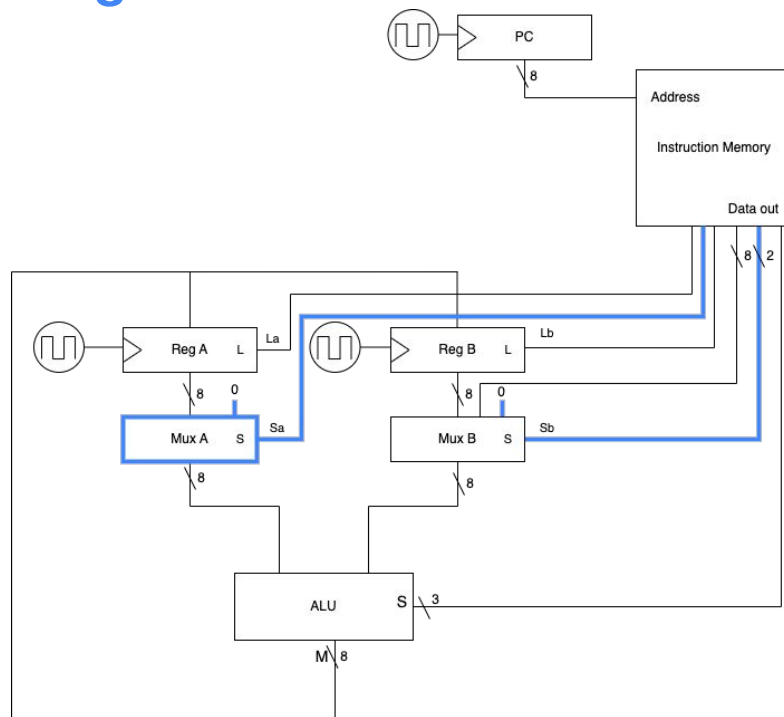
Ahora, ¿cómo inicializamos valores?



## Computador básico - Inicialización de registros

Para inicializar valores, necesitamos la capacidad de propagar el valor 0 para  $A$  o para  $B$  en la ALU. De aquí surge la necesidad de incluir el componente Mux  $A$  que recibe el valor de  $A$  y un 0, realizando la selección con una nueva señal de control  $Sa$ .

Al Mux  $B$  le añadimos una entrada adicional 0, lo que implica la necesidad de expandir  $Sb$  a 2 bits.



## Computador básico - Inicialización de registros

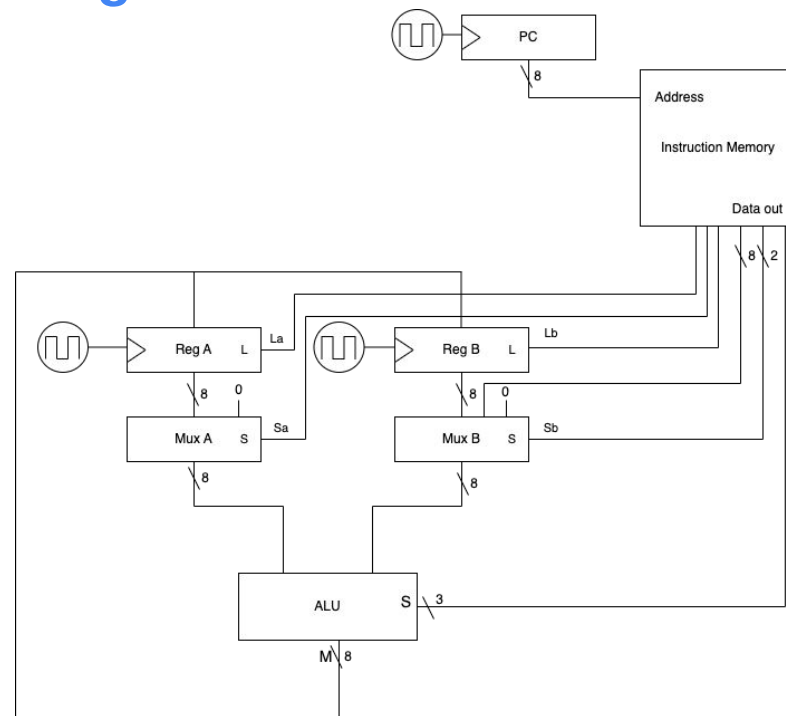
Entonces, para inicializar valores en  $A$  y en  $B$  realizamos las siguientes operaciones:

$$A = 0 + Lit$$

( $L_a = 1$ ;  $S_a = 1$ ;  $S_b = 01$ ,  $S = 000$ )

$$B = 0 + Lit$$

( $L_b = 1$ ;  $S_a = 1$ ;  $S_b = 01$ ,  $S = 000$ )



## Computador básico - Inicialización de registros

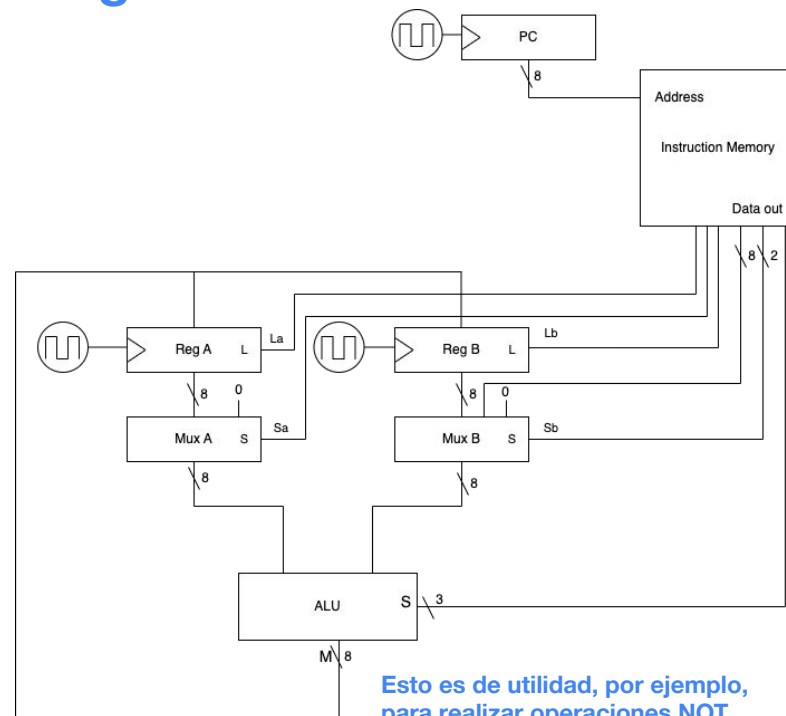
Con este nuevo esquema, también podemos copiar los valores de un registro a otro.

$$A = B + 0$$

$$(La = 1; Sa = 1; Sb = 00, S = 000)$$

$$B = A + 0$$

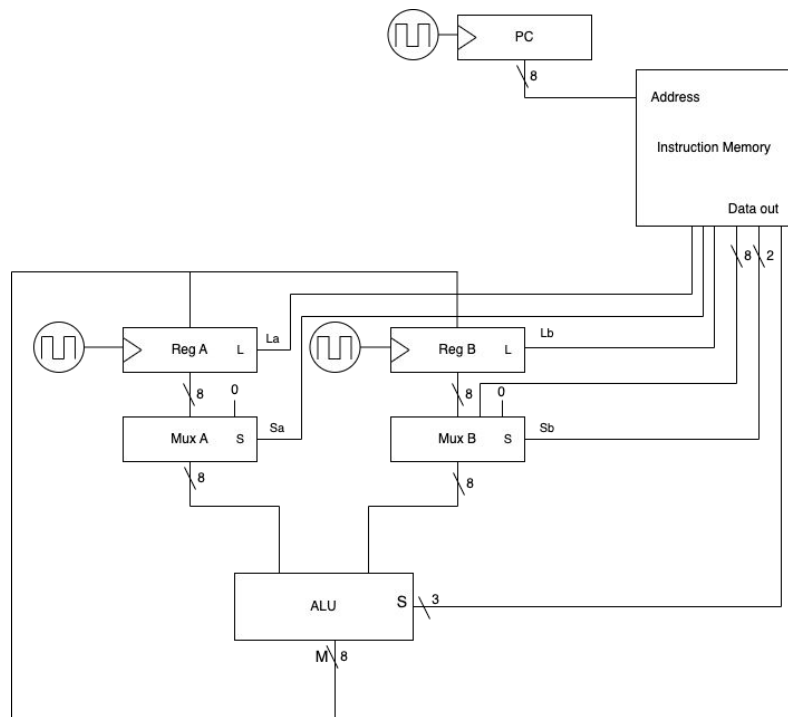
$$(Lb = 1; Sa = 0; Sb = 11, S = 000)$$



Esto es de utilidad, por ejemplo, para realizar operaciones NOT, SHL y SHR sobre *B* dado que la ALU solo puede realizarlas sobre su entrada *A*.

# Computador básico - Tabla de instrucciones actual

<i>La</i>	<i>Lb</i>	<i>Sa</i>	<i>Sb[1]</i>	<i>Sb[0]</i>	<i>S[2]</i>	<i>S[1]</i>	<i>S[0]</i>	Operación
1	0	1	0	0	0	0	0	$A = B$
0	1	0	1	1	0	0	0	$B = A$
1	0	1	0	1	0	0	0	$A = Lit$
0	1	1	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	$A = A \text{ AND } B$
0	1	0	0	0	0	1	0	$B = A \text{ AND } B$
1	0	0	0	1	0	1	0	$A = A \text{ AND } Lit$
1	0	0	0	0	0	1	1	$A = A \text{ OR } B$
0	1	0	0	0	0	1	1	$B = A \text{ OR } B$
1	0	0	0	1	0	1	1	$A = A \text{ OR } Lit$
1	0	0	0	0	1	0	0	$A = \text{NOT } A$
0	1	0	0	0	1	0	0	$B = \text{NOT } A$
1	0	0	0	0	1	0	1	$A = A \text{ XOR } B$
0	1	0	0	0	1	0	1	$B = A \text{ XOR } B$
1	0	0	0	1	1	0	1	$A = A \text{ XOR } Lit$
1	0	0	0	0	1	1	0	$A = \text{SHL } A$
0	1	0	0	0	1	1	0	$B = \text{SHL } A$
1	0	0	0	0	1	1	1	$A = \text{SHR } A$
0	1	0	0	0	1	1	1	$B = \text{SHR } A$



## Computador básico - Tabla de instrucciones actual

La	Lb	Sa	Sb[1]	Sb[0]	S[2]	S[1]	S[0]	Operación
1	0	1	0	0	0	0	0	$A = B$
0	1	0	1	1	0	0	0	$B = A$
1	0	1	0	1	0	0	0	$A = Lit$
0	1	1	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	$A = A \text{ AND } B$
0	1	0	0	0	0	1	0	$B = A \text{ AND } B$
1	0	0	0	1	0	1	0	$A = A \text{ AND } Lit$
1	0	0	0	0	0	1	1	$A = A \text{ OR } B$
0	1	0	0	0	0	1	1	$B = A \text{ OR } B$
1	0	0	0	1	0	1	1	$A = A \text{ OR } Lit$
1	0	0	0	0	1	0	0	$A = \text{NOT } A$
0	1	0	0	0	1	0	0	$B = \text{NOT } A$
1	0	0	0	0	1	0	1	$A = A \text{ XOR } B$
0	1	0	0	0	1	0	1	$B = A \text{ XOR } B$
1	0	0	0	1	1	0	1	$A = A \text{ XOR } Lit$
1	0	0	0	0	1	1	0	$A = \text{SHL } A$
0	1	0	0	0	1	1	0	$B = \text{SHL } A$
1	0	0	0	0	1	1	1	$A = \text{SHR } A$
0	1	0	0	0	1	1	1	$B = \text{SHR } A$

Si bien tenemos 8 señales de control distintas, de momento solo contamos con 26 instrucciones de interés (y no 256 como uno esperaría). ¿Qué podemos hacer para mejorar esto?

## Computador básico - Opcodes

Opcode	La	Lb	Sa	Sb[1]	Sb[0]	S[2]	S[1]	S[0]	Operación
000000	1	0	1	0	0	0	0	0	$A = B$
000001	0	1	0	1	0	0	0	0	$B = A$
000010	1	0	1	0	1	0	0	0	$A = Lit$
000011	0	1	1	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	$A = A \text{ AND } B$
001011	0	1	0	0	0	0	1	0	$B = A \text{ AND } B$
001100	1	0	0	0	1	0	1	0	$A = A \text{ AND } Lit$
001101	1	0	0	0	0	0	1	1	$A = A \text{ OR } B$
001110	0	1	0	0	0	0	1	1	$B = A \text{ OR } B$
001111	1	0	0	0	1	0	1	1	$A = A \text{ OR } Lit$
010000	1	0	0	0	0	1	0	0	$A = \text{NOT } A$
010001	0	1	0	0	0	1	0	0	$B = \text{NOT } A$
010010	1	0	0	0	0	1	0	1	$A = A \text{ XOR } B$
010011	0	1	0	0	0	1	0	1	$B = A \text{ XOR } B$
010100	1	0	0	0	1	1	0	1	$A = A \text{ XOR } Lit$
010101	1	0	0	0	0	1	1	0	$A = \text{SHL } A$
010110	0	1	0	0	0	1	1	0	$B = \text{SHL } A$
010111	1	0	0	0	0	1	1	1	$A = \text{SHR } A$
011000	0	1	0	0	0	1	1	1	$B = \text{SHR } A$

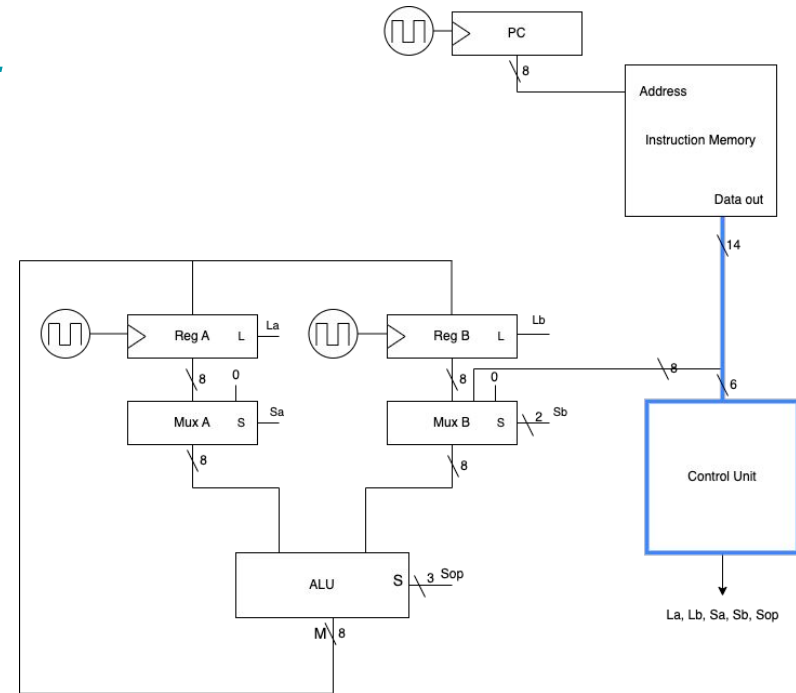
Hacemos uso de los *opcodes* (o códigos de operación) que, haciendo uso de una cantidad menor de bits, definen la combinación de señales de control que ejecutan la instrucción deseada. Nos falta, no obstante, un componente que pueda decodificar los *opcodes*.



## Computador básico - Unidad de control

Añadimos el componente **Control Unit** o unidad de control. A partir de un *opcode*, este decodifica la señal de entrada y transmite al resto del circuito las señales de control que derivan en la ejecución de la instrucción de ingreso.

Puede ser una ROM o decodificador. Para efectos prácticos, la máquina funcionará de ambas formas.



## Computador básico - Assembly

Opcode	La	Lb	Sa	Sb[1]	Sb[0]	S[2]	S[1]	S[0]	Operación
000000	1	0	1	0	0	0	0	0	$A = B$
000001	0	1	0	1	0	0	0	0	$B = A$
000010	1	0	1	0	1	0	0	0	$A = Lit$
000011	0	1	1	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	$A = A \text{ AND } B$
001011	0	1	0	0	0	0	1	0	$B = A \text{ AND } B$
001100	1	0	0	0	1	0	1	0	$A = A \text{ AND } Lit$
001101	1	0	0	0	0	0	1	1	$A = A \text{ OR } B$
001110	0	1	0	0	0	0	1	1	$B = A \text{ OR } B$
001111	1	0	0	0	1	0	1	1	$A = A \text{ OR } Lit$
010000	1	0	0	0	0	1	0	0	$A = \text{NOT } A$
010001	0	1	0	0	0	1	0	0	$B = \text{NOT } A$
010010	1	0	0	0	0	1	0	1	$A = A \text{ XOR } B$
010011	0	1	0	0	0	1	0	1	$B = A \text{ XOR } B$
010100	1	0	0	0	1	1	0	1	$A = A \text{ XOR } Lit$
010101	1	0	0	0	0	1	1	0	$A = \text{SHL } A$
010110	0	1	0	0	0	1	1	0	$B = \text{SHL } A$
010111	1	0	0	0	0	1	1	1	$A = \text{SHR } A$
011000	0	1	0	0	0	1	1	1	$B = \text{SHR } A$

Este esquema sigue teniendo un grado de dificultad al momento de programar, ya que sigue dependiendo de la representación binaria. Para evitarlo, hacemos uso del lenguaje de programación **Assembly**.

## Computador básico - Assembly

Instrucción	Operandos	Opcod	La	Lb	Sa	Sb[1]	Sb[0]	S[2]	S[1]	S[0]	Operación
MOV	A,B	000000	1	0	1	0	0	0	0	0	A = B
	B,A	000001	0	1	0	1	0	0	0	0	B = A
	A,Lit	000010	1	0	1	0	1	0	0	0	A = Lit
	B,Lit	000011	0	1	1	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 = NOT A
	B,A	010001	0	1	0	0	0	1	0	0	B = NOT A
XOR	A,B	010010	1	0	0	0	0	1	0	1	A = A XOR B
	B,A	010011	0	1	0	0	0	1	0	1	B = A XOR B
	A,Lit	010100	1	0	0	0	1	1	0	1	A = A XOR Lit
SHL	A,A	010101	1	0	0	0	0	1	1	0	A = SHL A
	B,A	010110	0	1	0	0	0	1	1	0	B = SHL A
SHR	A,A	010111	1	0	0	0	0	1	1	1	A = SHR A
	B,A	011000	0	1	0	0	0	1	1	1	B = SHR A

Con Assembly programamos las instrucciones de la máquina. En la práctica, la traducción de estas instrucciones a *opcodes* (lenguaje de máquina) se realiza a través del **Assembler**, que se encarga de reemplazar cada instrucción por el código correspondiente.

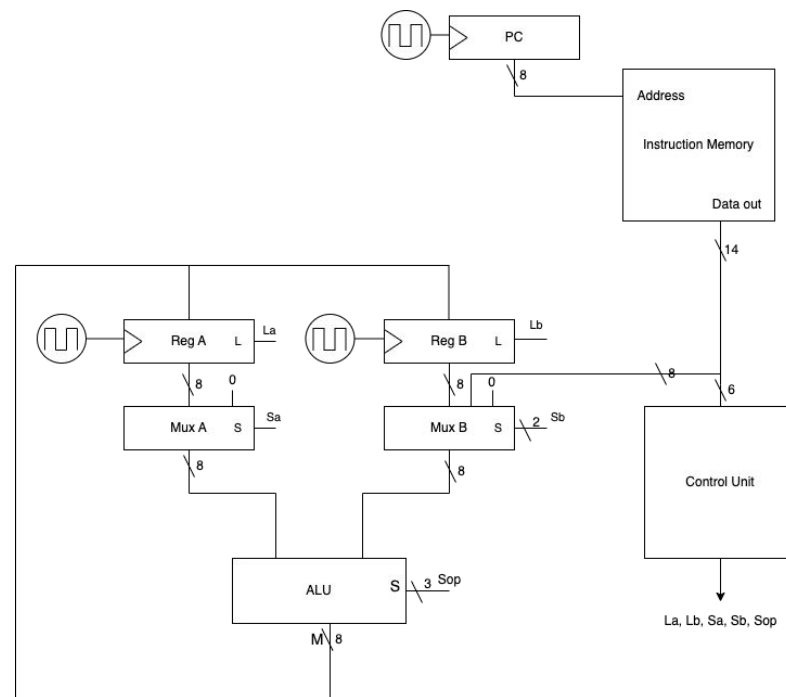
## Computador básico - Assembly

**Ejercicio:** Haciendo uso de Assembly, programe la secuencia de los primeros 8 números de Fibonacci. Al finalizar, almacene el resultado en el registro *A* y deje el registro *B* “limpio” (valor igual a cero).

## Computador básico - Memoria de datos

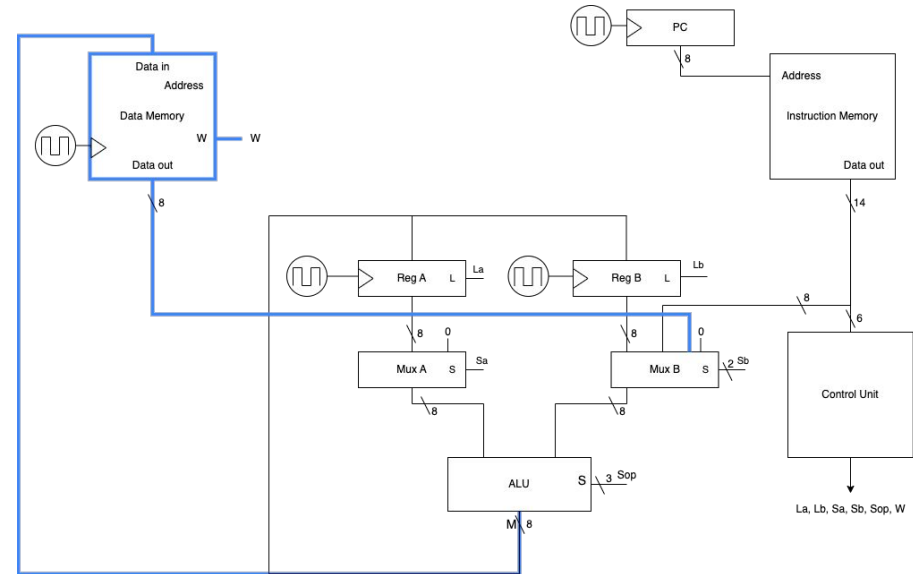
Logramos reducir el volumen de datos de la memoria de instrucciones, pero todavía estamos limitados respecto a la cantidad de datos que podemos almacenar durante la ejecución de nuestro programa.

¿Qué componente podríamos agregar?



## Computador básico - Memoria de datos

Añadimos una RAM para almacenar datos **durante la ejecución del programa**. Este componente es el *Data Memory* (o memoria de datos). Al usar registros de memoria en su interior, requiere sincronizarse con el *clock* del sistema. Por otra parte, usamos el *slot* disponible del Mux *B* para poder utilizar su valor en la ALU.

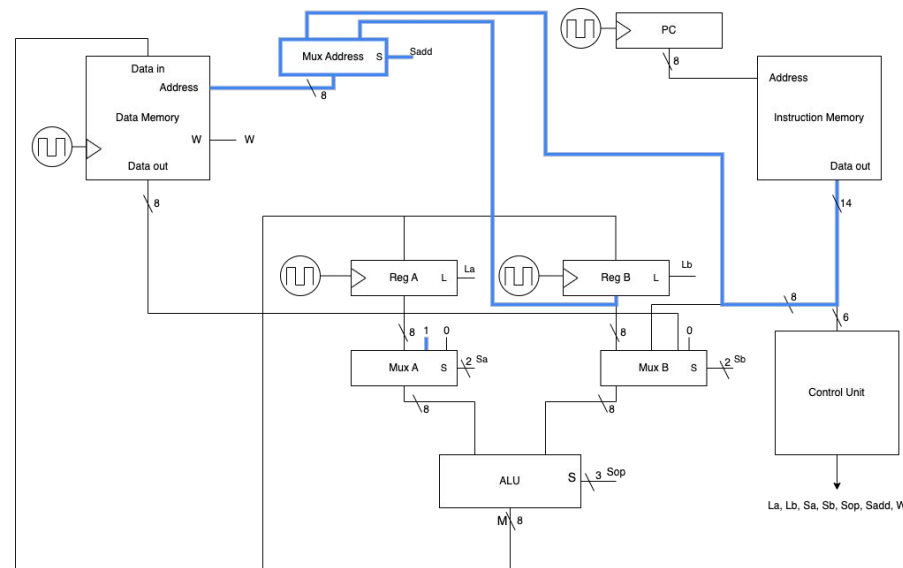


## Computador básico - Direccionamiento

Para el direccionamiento, tenemos dos métodos:

- **Directo:** Se indica la dirección de memoria con un literal.
- **Indirecto:** Se indica la dirección de memoria con el valor de un registro.

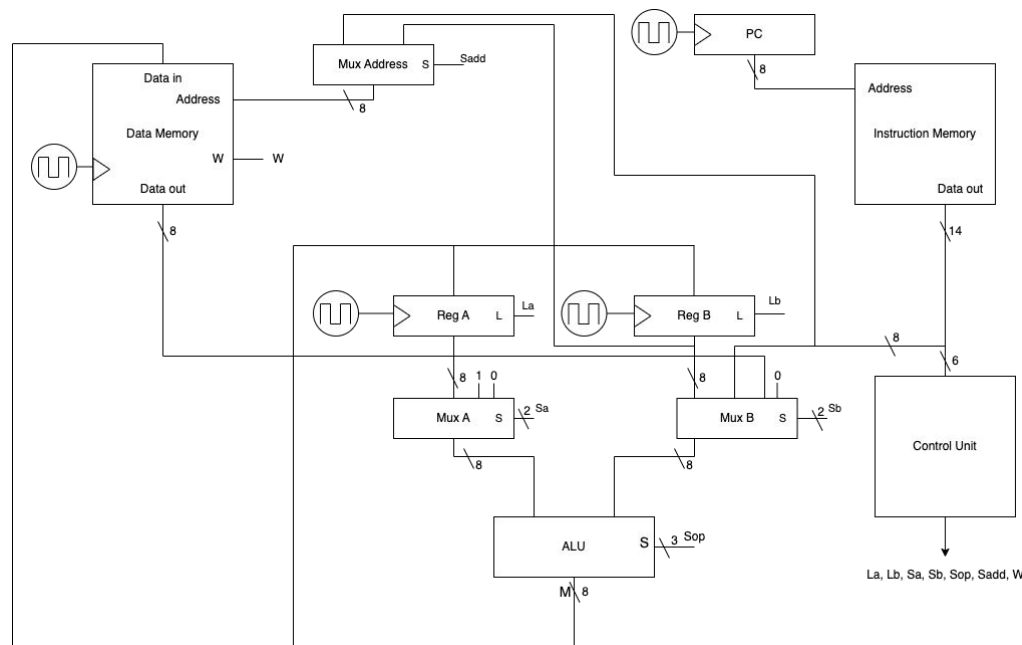
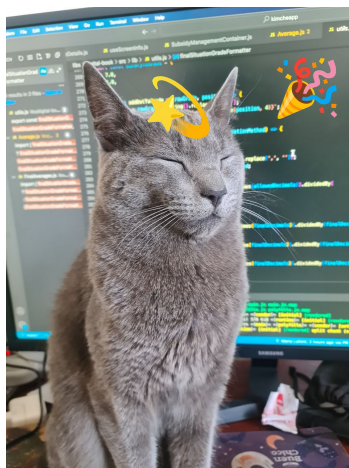
Se selecciona uno de los dos con un nuevo Mux *Address*.



Se añade, además, el valor “1” al Mux A (aumentando además en un bit su bus de selección). Esto lo utilizaremos para incluir un nuevo conjunto de instrucciones INC, pero la próxima clase veremos su verdadera utilidad en términos de control de flujo.

## Computador básico - Primera versión

Este corresponde a la primera versión del computador básico del curso.





## Computador básico - Instrucciones de direccionamiento

Instrucción	Operandos	Operación	Ejemplo
MOV	$A, (Dir)$	$A = Mem[Dir]$	MOV $A, (var0)$
	$B, (Dir)$	$B = Mem[Dir]$	MOV $B, (var1)$
	$(Dir), A$	$Mem[Dir] = A$	MOV $(var0), A$
	$(Dir), B$	$Mem[Dir] = B$	MOV $(var1), B$
	$A, (B)$	$A = Mem[B]$	-
	$B, (B)$	$B = Mem[B]$	-
ADD	$(B), A$	$Mem[B] = A$	-
	$A, (Dir)$	$A = A + Mem[Dir]$	ADD $A, (var0)$
	$A, (B)$	$A = A + Mem[B]$	-
	$(Dir)$	$Mem[Dir] = A + B$	ADD $(var0)$
SUB	$A, (Dir)$	$A = A - Mem[Dir]$	SUB $A, (var0)$
	$A, (B)$	$A = A - Mem[B]$	-
	$(Dir)$	$Mem[Dir] = A - B$	SUB $(var0)$
AND	$A, (Dir)$	$A = A \text{ AND } Mem[Dir]$	AND $A, (var0)$
	$A, (B)$	$A = A \text{ AND } Mem[B]$	-
	$(Dir)$	$Mem[Dir] = A \text{ AND } B$	AND $(var0)$
OR	$A, (Dir)$	$A = A \text{ OR } Mem[Dir]$	OR $A, (var0)$
	$A, (B)$	$A = A \text{ OR } Mem[B]$	-
	$(Dir)$	$Mem[Dir] = A \text{ OR } B$	OR $(var0)$
NOT	$(Dir)$	$Mem[Dir] = \text{NOT } A$	NOT $(var0)$
XOR	$A, (Dir)$	$A = A \text{ XOR } Mem[Dir]$	XOR $A, (var0)$
	$A, (B)$	$A = A \text{ XOR } Mem[B]$	-
	$(Dir)$	$Mem[Dir] = A \text{ XOR } B$	XOR $(var0)$
SHL	$(Dir)$	$Mem[Dir] = \text{SHL } A$	SHL $(var0)$
SHR	$(Dir)$	$Mem[Dir] = \text{SHR } A$	SHR $(var0)$
INC	$B$	$B = B + 1$	-
	$(B)$	$Mem[B] = Mem[B] + 1$	-
	$(Dir)$	$Mem[Dir] = Mem[Dir] + 1$	INC $(var2)$

El nuevo listado de instrucciones adjunto nos permite manejar variables en memoria. Utilizamos la nomenclatura (*Label*) para el direccionamiento directo y (*B*) para el direccionamiento indirecto (haciendo uso del registro *B*).

Las instrucciones INC serán de utilidad cuando podamos recorrer arreglos de forma **controlada**.

## Computador básico - Variables en Assembly

Nos falta algo: ¿cómo manejamos las variables a nivel de código?

En un programa Assembly, contaremos con dos segmentos: **DATA** (definición de variables y arreglos) y **CODE** (definición de instrucciones). El Assembler se encarga de *mapear* cada variable definida en el primer segmento del programa a una dirección de memoria (a través de operaciones MOV, como veremos a continuación).

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

## Computador básico - Variables en Assembly

Para el segmento DATA de la figura, el Assembler realiza dos acciones:

1. Asigna a cada variable una dirección de memoria según el orden en que se definen.  
 $x = 0x00 = 0$ ;  $y = 0x01 = 1$ .
2. Agrega, al principio del programa, las instrucciones que almacenan estos valores en memoria según las direcciones asignadas.

### DATA:

x 2

y 4

Instrucciones de almacenamiento

MOV A, 2

MOV (0), A dirección x

MOV A, 4

MOV (1), A dirección y

Esta es una forma de hacerlo, depende del Assembler la forma en la que se realizará la escritura en memoria.

## Computador básico - Variables en Assembly

En el caso de los **arreglos**, el *label* se asocia solo a su primer elemento, como en el ejemplo. No obstante, el Assembler asigna al resto de elementos las direcciones contiguas en memoria.

Para trabajar con arreglos en Assembly es necesario, además, definir una variable para su largo. La próxima clase veremos por qué.

**DATA:**

```
arr 1  
    2  
len 2
```

Instrucciones de almacenamiento

```
MOV A,1  
MOV (0),A dirección arr  
MOV A,2  
MOV (1),A dirección arr+1  
MOV A,2  
MOV (2),A dirección len
```

Esta es una forma de hacerlo, depende del Assembler la forma en la que se realizará la escritura en memoria.

## Computador básico - Direccionamiento en Assembly

A continuación, un ejemplo de código Assembly con direccionamiento **directo**.

Utilizamos tres variables en memoria. Posteriormente, almacenamos en var3 el resultado de la suma entre las variables var1 y var2.

### DATA:

var1 1

var2 3

var3 0

### CODE:

MOV A, (var1)

MOV B, (var2)

ADD A, B

MOV (var3), A

## Computador básico - Direccionamiento en Assembly

A continuación, un ejemplo de código Assembly con direccionamiento **indirecto**.

En este caso, a través del registro *B* y direccionamiento indirecto recorreremos todo el arreglo y duplicamos el valor de cada uno de sus elementos.

### DATA:

```
var 1
arr 1
    3
    5
```

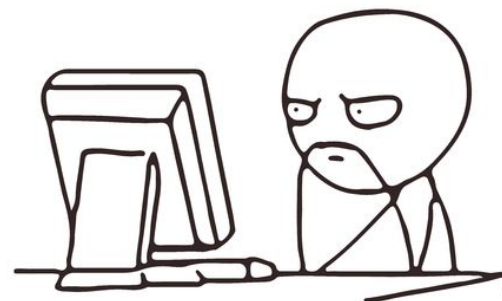
### CODE:

```
MOV B,arr ;Esta instrucción equivale a
MOV A,(B) ;MOV B,Lit - Lit = dirección
SHL A,A   ;de arr
MOV (B),A
INC B
MOV A,(B)
SHL A,A
MOV (B),A
INC B
MOV A,(B)
SHL A,A
MOV (B),A
```

## Ejercicios

Ahora, veremos algunos ejercicios.

Estos se basan en preguntas de tareas y pruebas de semestres anteriores, por lo que nos servirán de preparación para las evaluaciones.



## Ejercicios

Considere el siguiente programa.

Construya un programa en Assembly que obtenga el mismo resultado (considere que  $x$ ,  $y$  y  $z$  parten con sus valores almacenados en memoria).

```
x = 2
y = 4
z = 0 # Variable auxiliar
z = x
x = x + y
y = y - z
```



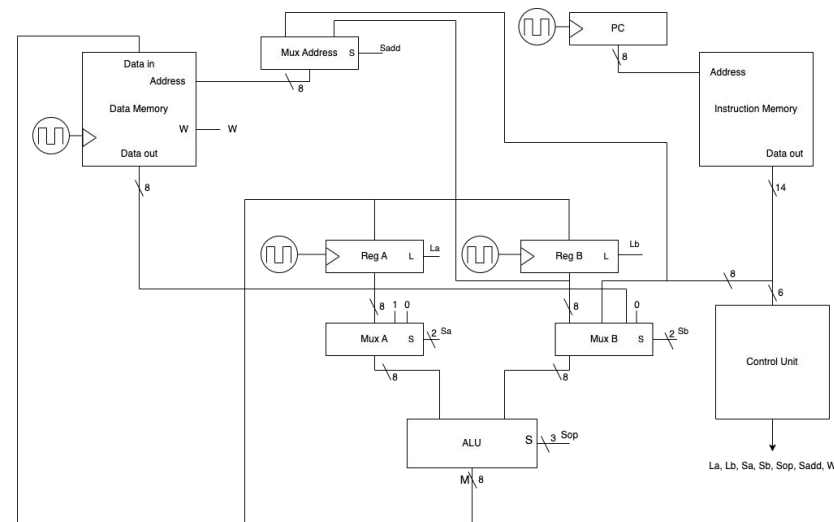
## Ejercicios

Considere el siguiente programa.

Ahora, programe en Assembly un código que obtenga el mismo resultado de  $x$  e  $y$ , pero sin hacer uso de la variable  $z$  en el segmento DATA.

```
x = 2
y = 4
z = 0 # Variable auxiliar
z = x
x = x + y
y = y - z
```

A partir de los programas anteriores, explique el flujo resultante de cada uno de ellos en el diagrama del computador básico que se muestra a continuación.



## Ejercicios

¿En qué casos es posible soportar la instrucción *ADD B,Lit* en el computador básico **sin modificar su *hardware* ni sobreescribir datos**? Para los casos negativos, indique qué modificaciones al *hardware* y/o Assembly se deberían hacer para soportarla.

## Ejercicios

Modifique el diagrama del computador básico de manera que soporte la ejecución de la instrucción **GOTO dir**, que fuerza que la siguiente instrucción en ejecutarse sea la ubicada en la dirección **dir**.

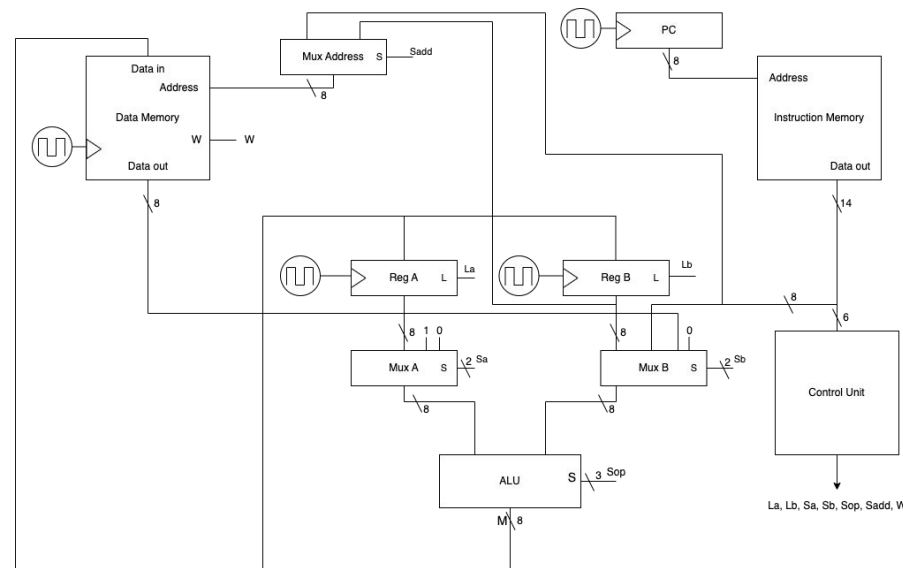
Realizaremos este ejercicio a continuación.

## Ejercicios - Desarrollo en clases

Lo que se busca es **cambiar** la instrucción a ejecutar según una dirección de memoria **dir**.

Esta corresponderá a un literal dentro de la instrucción **GOTO dir**, pero ¿dónde lo almacenamos?

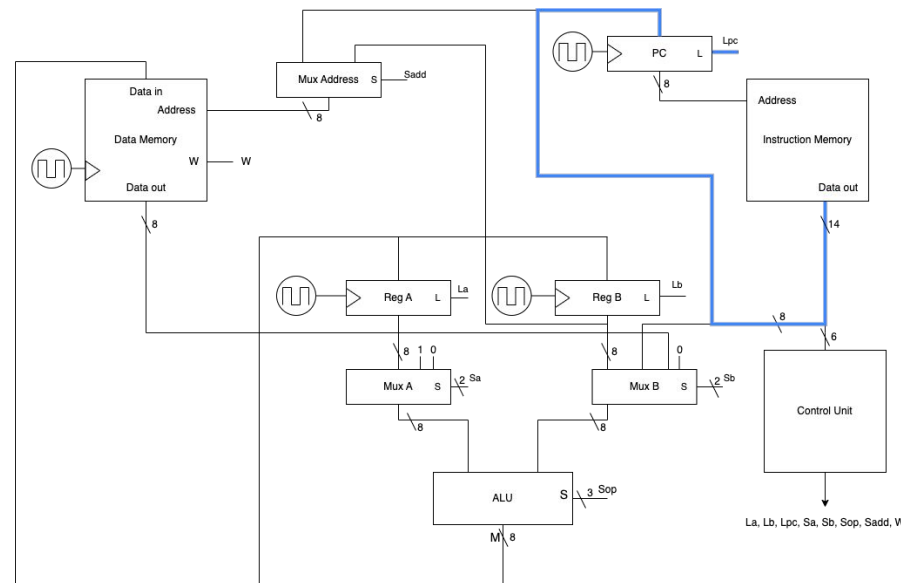
En el registro responsable de la ejecución de instrucciones: **PC**.



## Ejercicios - Desarrollo en clases

Se agrega la señal de control  $Lpc$  que habilita la escritura sobre el registro PC con el literal de la instrucción. Esto permitirá la ejecución forzada de la instrucción almacenada en la dirección *dir*.

Esto corresponde a un **salto incondicional** y será el primer paso para terminar nuestra máquina programable.



## Antes de terminar

¿Dudas?

¿Consultas?

¿Inquietudes?

¿Comentarios?





**DCC**

DEPARTAMENTO DE CIENCIA  
DE LA COMPUTACIÓN

**IIC2343**

# **Arquitectura de Computadores**

**Clase 4 - Programabilidad**

**Profesor: Germán Leandro Contreras Sagredo**

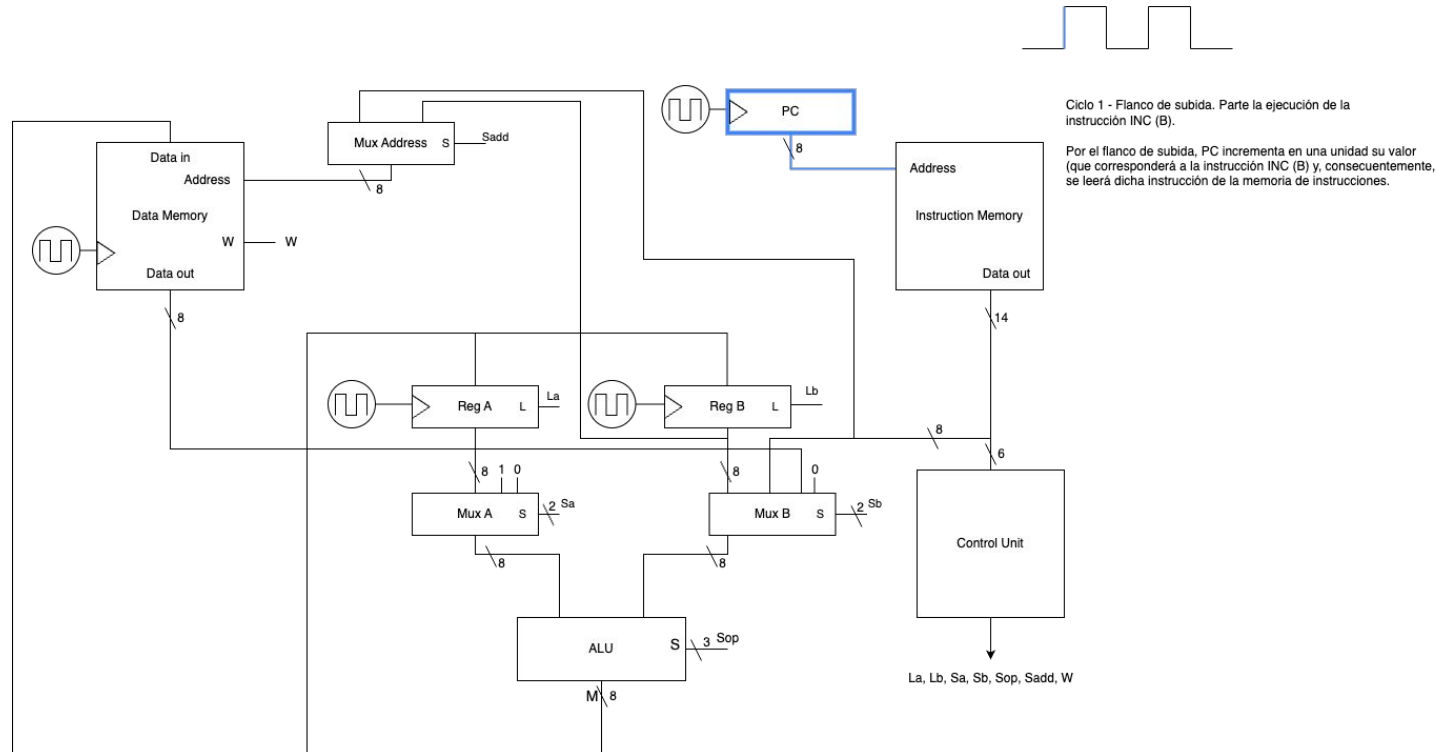


## Anexo - Ejecución de la instrucción INC (B)

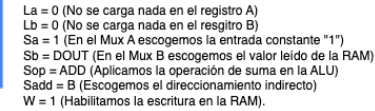
Puede parecer contra-intuitivo, pero existen instrucciones que pueden leer una dirección en memoria y luego actualizar su valor. Un ejemplo para esto es la instrucción INC (B), que incrementa en una unidad el valor almacenado en la dirección apuntada por *B*.

¿Cómo se logra esto? Lo veremos a continuación viendo el paso a paso de la ejecución en la microarquitectura de nuestro computador.

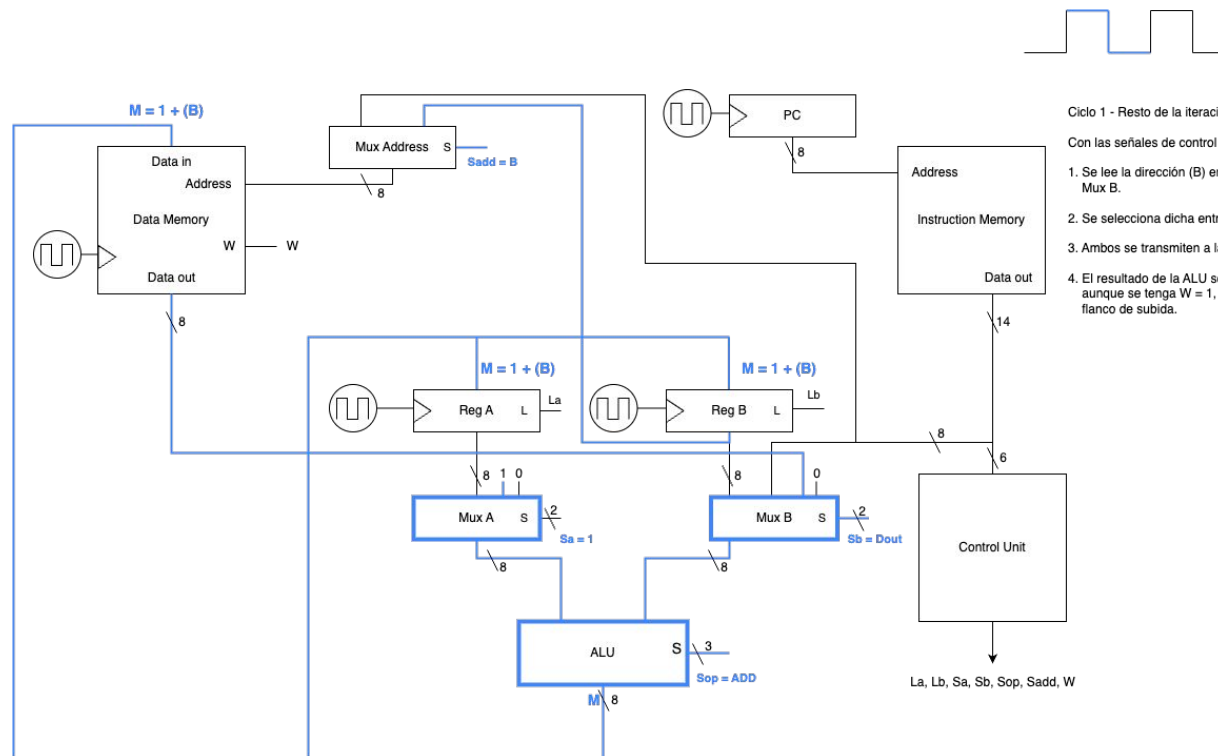
## Anexo - Ejecución de la instrucción INC (B)



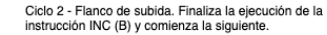
## A square wave signal is shown. The signal is high for the first two time intervals, then drops to a lower level for the next two intervals, and then returns to the high level for the final two intervals. The segment where the signal is low (the second and third intervals) is highlighted in blue.



# Anexo - Ejecución de la instrucción INC (B)



A square wave signal is shown, consisting of a horizontal line at a low level, followed by a vertical line up to a high level, a horizontal line at the high level, a vertical line down to the low level, and another horizontal line at the low level. A blue vertical line is drawn at the rising edge of the second pulse.



Por el flanco de subida, se activa la actualización de los registros A, B, PC y de la RAM. Como  $L_a = L_b = 0$ , estos mantienen sus valores originales. No obstante, en este instante la **RAM actualiza su valor** en la dirección B (valor transmitido durante el ciclo) con  $1 + (B)$  (es decir, incrementa en una unidad el valor original).

Como PC también actualiza su valor, se actualiza la dirección de la instrucción a ejecutar y así comenzará su ejecución. No obstante, el valor actualizado en la RAM estará resguardado ya que las señales de control nuevas serán propagadas **después** del flanco de subida.

## Anexo - Resolución de ejercicios

### ¡Importante!

Estos ejercicios pueden tener más de un desarrollo correcto. Las respuestas a continuación no son más que soluciones que **no excluyen** otras alternativas igual de correctas.



## Ejercicios - Respuesta

Haciendo uso de Assembly, programe la secuencia de los primeros 8 números de Fibonacci. Al finalizar, almacene el resultado en el registro *A* y deje el registro *B* “limpio” (valor igual a cero).

```
MOV A,0 ;A = 0
MOV B,1 ;B = 1
ADD A,B ;A = A + B = 1
ADD B,A ;B = A + B = 2
ADD A,B ;A = A + B = 3
ADD B,A ;B = A + B = 5
ADD A,B ;A = A + B = 8
ADD B,A ;B = A + B = 13
ADD A,B ;A = B = 13
MOV B,0 ;B = 0
```

## Ejercicios - Respuesta

Considere el siguiente programa.

Construya un programa en Assembly que obtenga el mismo resultado (considere que  $x$ ,  $y$  y  $z$  parten con sus valores almacenados en memoria).

```
x = 2
y = 4
z = 0 # Variable auxiliar
z = x
x = x + y
y = y - z
```

Respuesta en la siguiente diapositiva.



## Ejercicios - Respuesta

DATA :

x 2

y 4

z 0

CODE:

MOV A, (x) ; Guardamos x en A

MOV (z), A ; Guardamos A en z, variable auxiliar

MOV B, (y) ; Guardamos y en B

ADD A, B ; Guardamos en A x+y

MOV (x), A ; Guardamos en x el resultado de la suma

MOV A, (y) ; Guardamos y en A

MOV B, (z) ; Guardamos z en B

SUB A, B ; Guardamos en A y-z

MOV (y), A ; Guardamos en y el resultado de la resta

## Ejercicios - Respuesta

Considere el siguiente programa.

Ahora, programe en Assembly un código que obtenga el mismo resultado de  $x$  e  $y$ , pero sin hacer uso de la variable  $z$  en el segmento DATA.

```
x = 2
y = 4
z = 0 # Variable auxiliar
z = x
x = x + y
y = y - z
```

Respuesta en la siguiente diapositiva.

## Ejercicios - Respuesta

DATA:

x 2

y 4

CODE:

MOV A,(x) ;Guardamos x en A

MOV B,(y) ;Guardamos en B el valor de y

ADD A,B ;Guardamos en A x+y

MOV B,(x) ;Guardamos x en B

MOV (x),A ;Guardamos en x el resultado de la suma

MOV A,(y) ;Guardamos y en A

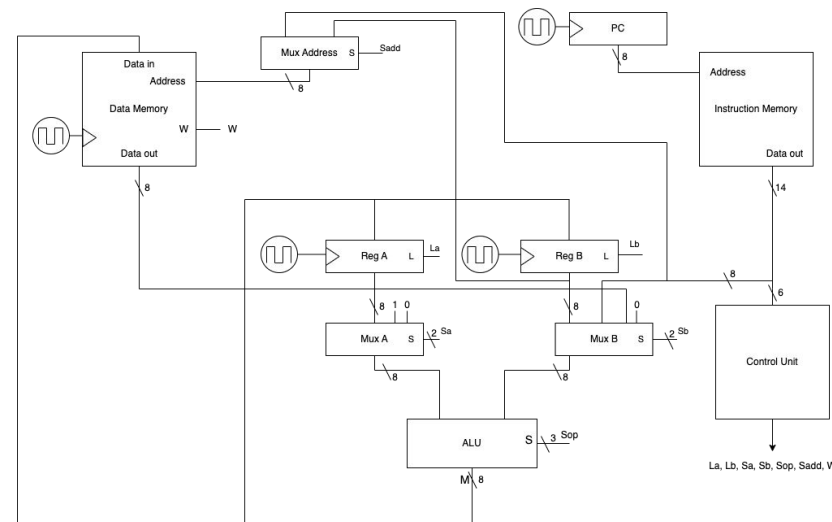
SUB A,B ;Guardamos en A y-z

MOV (y),A ;Guardamos en y el resultado de la resta

## Ejercicios - Respuesta

A partir de los programas anteriores, explique el flujo resultante de cada uno de ellos en el diagrama del computador básico que se muestra a continuación.

Respuesta en la siguiente diapositiva.



# Ejercicios - Respuesta

Antes de detallar el flujo, veremos cómo se almacena cada variable en la memoria de datos:

- 1) `MOV A,2`: Almacena en el registro A el valor de la variable  $x$ .
- 2) `MOV (0),A`: Almacena en la primera dirección de la memoria de datos el valor del registro A (correspondiente a  $x$ ).
- 3) `MOV A,4`: Almacena en el registro A el valor de la variable  $y$ .
- 4) `MOV (1),A`: Almacena en segunda dirección de la memoria de datos el valor del registro A (correspondiente a  $y$ ).
- 5) `MOV A,0`: Almacena en el registro A el valor de la variable  $z$ .
- 6) `MOV (2),A`: Almacena en la primera dirección de la memoria de datos el valor del registro A (correspondiente a  $z$ ).

Es decir, por cada variable declarada se ejecutan dos instrucciones que permiten almacenar en la memoria de datos sus valores. Por lo general, el *Assembler* parte en la primera dirección y sigue de forma sucesiva para el resto de las variables declaradas<sup>3</sup>.

Siguiendo con el código:

- 1) `MOV A,(x)`: El literal de la ROM (correspondiente a la dirección de  $x$ ) es escogido por el multiplexor **Address** para obtener su valor y se habilita la carga en el registro A para que el número sea almacenado en él.
- 2) `MOV (z),A`: El literal de la ROM (correspondiente a la dirección de  $z$ ) es escogido por el multiplexor **Address** y se habilita la señal  $W$  para poder escribir en la RAM. Luego, se configura la ALU de forma que el resultado sea el valor del registro A, escribiendo ese número en la dirección  $z$ .
- 3) `MOV B,(y)`: El literal de la ROM (correspondiente a la dirección de  $z$ ) es escogido por el multiplexor **Address** para obtener su valor y se habilita la carga en el registro B para que el número sea almacenado en él.

---

<sup>3</sup>Esto es muy útil al declarar arreglos.

## Ejercicios - Respuesta

- 4) **ADD A,B**: Se configuran los multiplexores de A y B para escoger los valores de sus registros. Luego, se configura la ALU para obtener la suma de ambos valores y, finalmente, se habilita la escritura en el registro A para que almacene el resultado.
- 5) **MOV (x),A**: El literal de la ROM (correspondiente a la dirección de  $x$ ) es escogido por el multiplexor **Address** y se habilita la señal  $W$  para poder escribir en la RAM. Luego, se configura la ALU de forma que el resultado sea el valor del registro A, escribiendo ese número en la dirección  $x$  (resultado de  $x + y$ ).
- 6) **MOV A,(y)**: El literal de la ROM (correspondiente a la dirección de  $y$ ) es escogido por el multiplexor **Address** para obtener su valor y se habilita la carga en el registro A para que el número sea almacenado en él.
- 7) **MOV B,(z)**: El literal de la ROM (correspondiente a la dirección de  $z$ ) es escogido por el multiplexor **Address** para obtener su valor y se habilita la carga en el registro B para que el número sea almacenado en él.
- 8) **SUB A,B**: Se configuran los multiplexores de A y B para escoger los valores de sus registros. Luego, se configura la ALU para obtener la suma de ambos valores y, finalmente, se habilita la escritura en el registro A para que almacene el resultado.
- 9) **MOV (y),A**: El literal de la ROM (correspondiente a la dirección de  $y$ ) es escogido por el multiplexor **Address** y se habilita la señal  $W$  para poder escribir en la RAM. Luego, se configura la ALU de forma que el resultado sea el valor del registro A, escribiendo ese número en la dirección  $y$  (resultado de  $y - z$ ).

Se obvia el flujo del segundo programa, dado que es una versión simplificada del primero sin mayores cambios en las instrucciones utilizadas.

## Ejercicios - Respuesta

¿En qué casos es posible soportar la instrucción *ADD B,Lit* en el computador básico **sin modificar su *hardware* ni sobreescribir datos**? Para los casos negativos, indique qué modificaciones al *hardware* y/o Assembly se deberían hacer para soportarla.

Respuesta en la siguiente diapositiva.

## Ejercicios - Respuesta

Si dentro del computador básico revisamos los componentes Mux A y Mux B, podemos ver que los únicos literales que podrían ser seleccionados para ser almacenados en A son 0 y 1. Por lo tanto, sin modificar el computador básico, podemos soportar `ADD B,0`<sup>4</sup> y `ADD B,1` (que existe y llamamos `INC B`). Si quisiéramos habilitar la instrucción `ADD B,Lit` para cualquier literal, sería necesario entonces tener una conexión entre el Mux A y la ROM (para así poder recibir el literal a utilizar, al igual como está incluido en B) y definir la combinación  $S_{a0}, S_{a1}$  para escogerlo y ajustar las señales de control para esta nueva instrucción a partir de un nuevo *opcode*.

---

<sup>4</sup>Esto claramente no tiene mucho sentido.