



**DCC**  
DEPARTAMENTO DE CIENCIA  
DE LA COMPUTACIÓN

IIC2343

# Arquitectura de Computadores

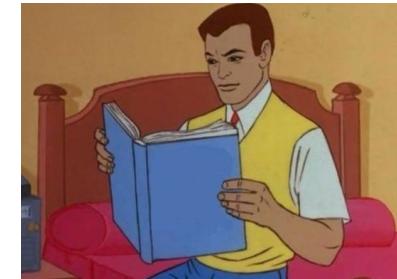
Clase 4 - Programabilidad

Profesor: Germán Leandro Contreras Sagredo

## Bibliografía

- **Apuntes históricos.** Hans Löbel, Alejandro Echeverría
  - 4 - Programabilidad
- **D. Patterson, Computer Organization and Design RISC-V Edition: The Hardware Software Interface. Morgan Kaufmann, 2020.**
  - Capítulos 2.1-2.2-2.3. Página 62, 101 en PDF.
  - Capítulo 2.5. Página 81, 120 en PDF.
  - Capítulos 4.1-4.2-4.3-4.4. Página 236, 320 en PDF.\*

\* Estos se basan en la implementación de una arquitectura que implementa RISC-V. Tomar solo como referencia.



## Objetivos de la clase

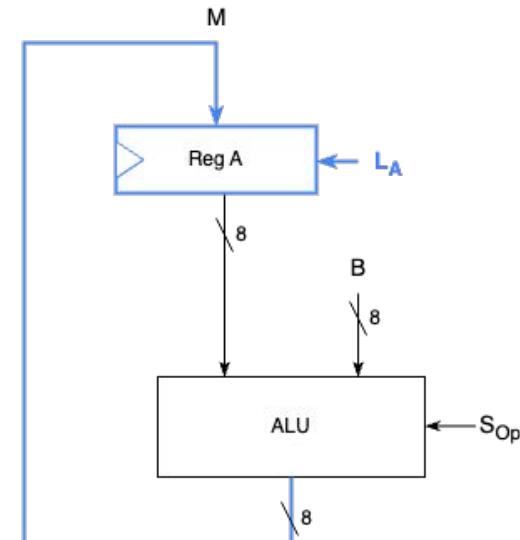
- Construir una máquina cuya combinación de señales permita que sea programable.
- Conocer nuevos métodos y componentes que simplifiquen la programabilidad de la máquina.
- Realizar ejercicios que consoliden los conocimientos anteriores.

## Hasta ahora...

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

Por otro lado, establecemos la cantidad de bits por bus de datos. Contamos con dos buses de entrada de 8 bits ( $A$ ,  $B$ ), señal de carga  $L_A$  de un bit y un bus de selección de operación de la ALU  $S_{Op}$  de 3 bits.

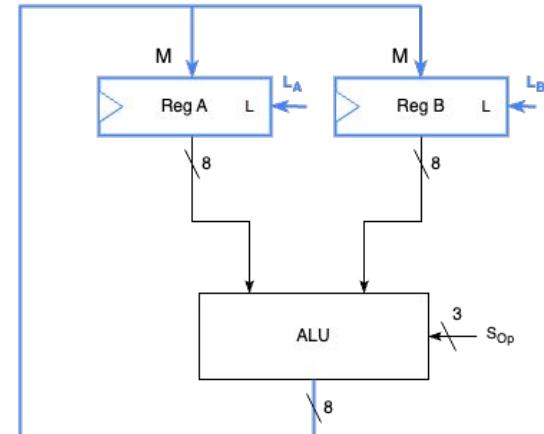
¿Podemos extender este esquema para  $B$ ?



## ALU extendida

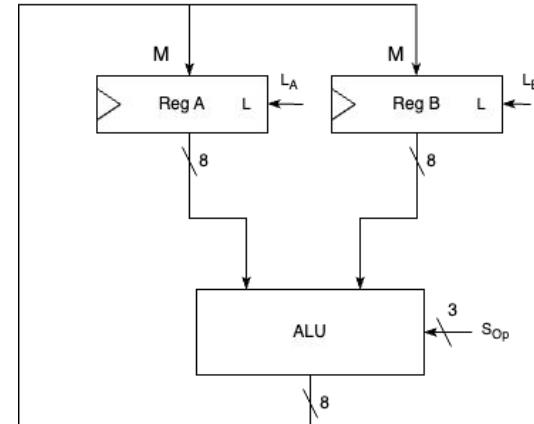
Sí. Incluimos un nuevo registro y, con él, una nueva señal de carga  $L_B$  de 1 bit.

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



## ALU extendida - Operaciones

$L_A$	$L_B$	$S_2$	$S_1$	$S_0$	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	$A = \text{SHL } A \text{ (SHL = Shift Left)}$
0	1	1	1	0	$B = \text{SHL } A$
1	0	1	1	1	$A = \text{SHR } A \text{ (SHR = Shift Right)}$
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

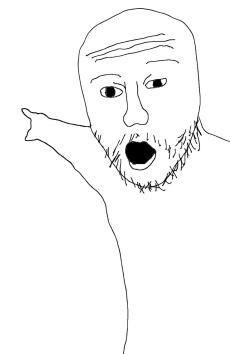
$L_A$	$L_B$	$S_2$	$S_1$	$S_0$	A	B	Operación
1	0	0	0	0	0	1	-
1	0	0	0	0	?	?	?
0	1	0	0	0	?	?	?
1	0	0	0	0	?	?	?
0	1	0	0	0	?	?	?
1	0	0	0	0	?	?	?
0	1	0	0	0	?	?	?

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

$L_A$	$L_B$	$S_2$	$S_1$	$S_0$	A	B	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.**

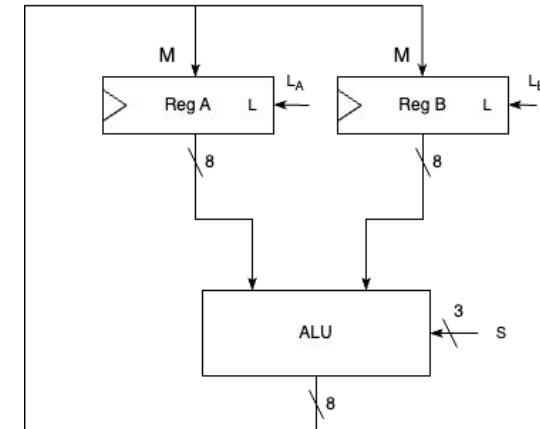


Con nuestra máquina **actual**, esta es la única forma de obtener la secuencia.

## 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?

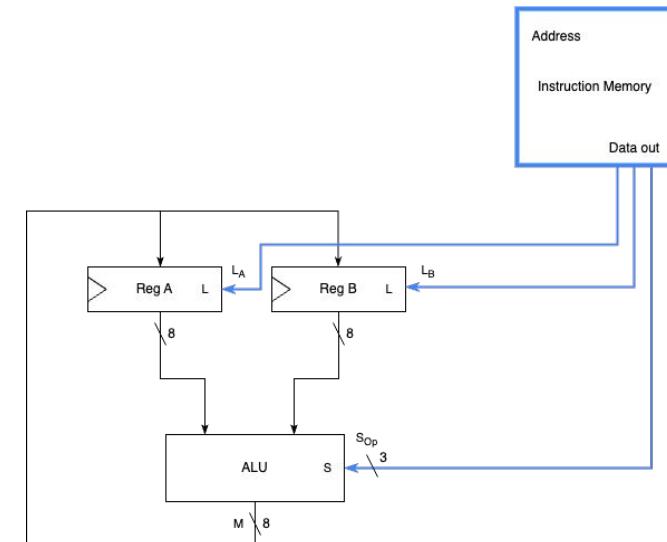


## Almacenamiento de instrucciones - ROM

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.

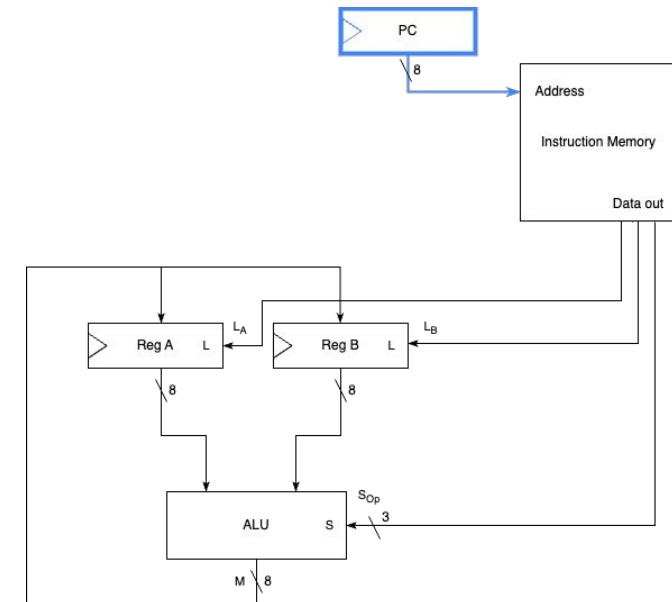
¿Cómo ejecutamos la secuencia en orden?



## Almacenamiento de instrucciones - *Program Counter*

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

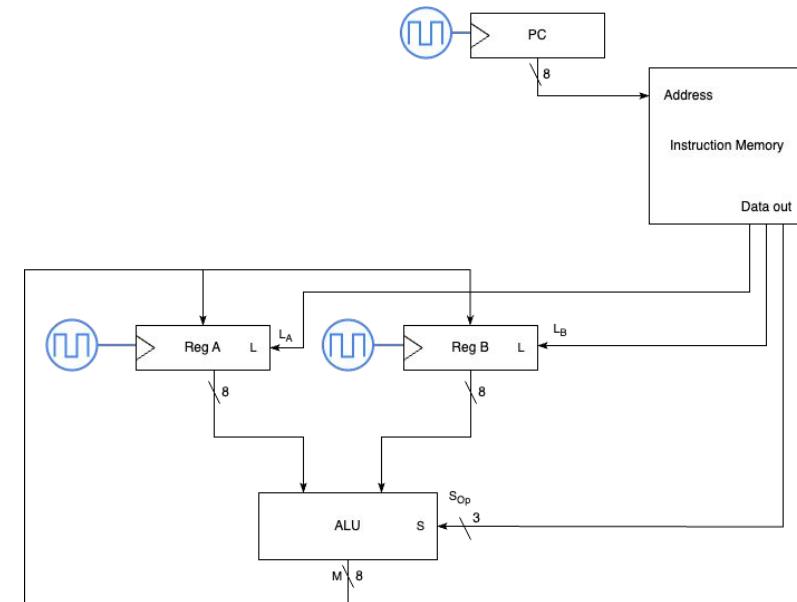
Ahora, ¿cómo nos aseguramos de que los registros y el programa estén en sincronía?



## Almacenamiento de instrucciones - *Clock*

Cada registro, contador y unidad de almacenamiento que requiera de un flanco de subida estará conectado a **una única señal *clock***. Así, el flanco de subida del *clock* permitirá el almacenamiento de datos en estos componentes **simultáneamente**.

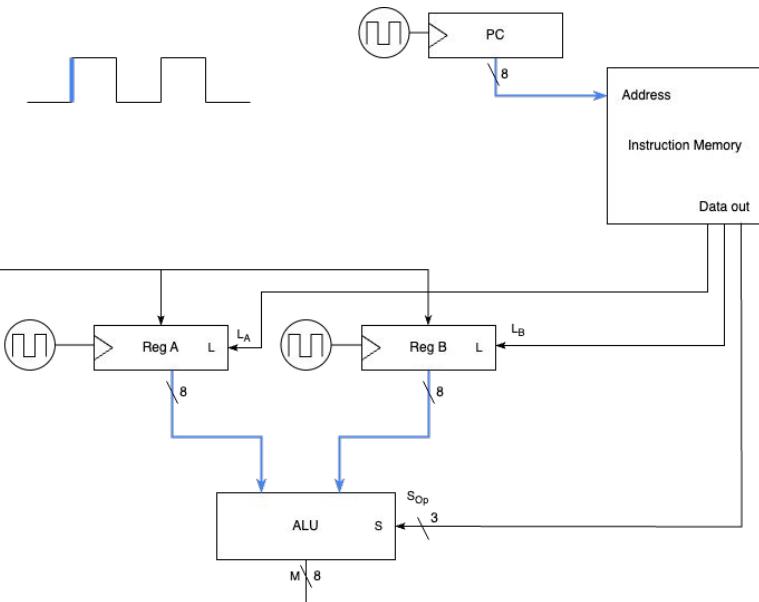
A continuación, veremos un ejemplo del funcionamiento completo.



## 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.

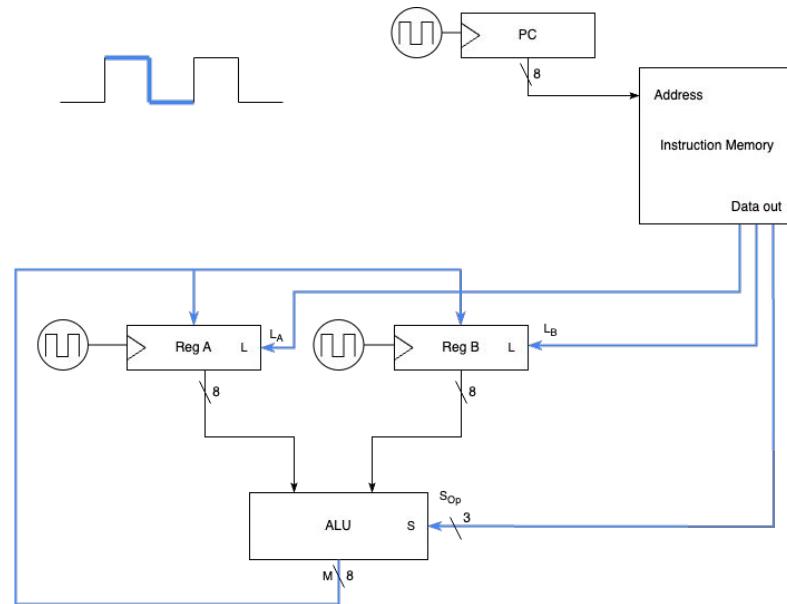
Asumimos, en primera instancia, que los registros no actualizan su valor.



## 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.

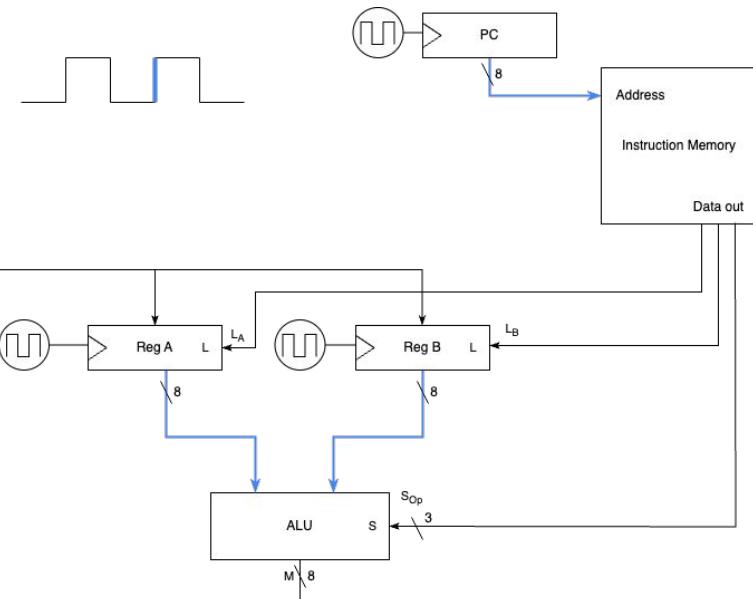
Cuando el circuito se estabiliza, el resultado de la operación de la primera instrucción estará esperando a ser almacenado en los registros *A* o *B*, según corresponda.



## Ejecución de la máquina programable - Paso 3

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

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



## Ejecución de la máquina programable - Un último comentario

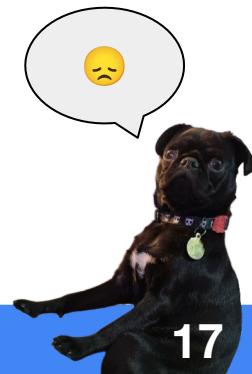
Como todos los componentes de almacenamiento se conectan a un único *clock* del sistema, es importante que su periodo sea **igual o mayor al tiempo de ejecución de la instrucción de máximo retraso de propagación de las instrucciones ejecutables.**

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 se conoce como **overclocking**: ejecución más rápida del computador, pero propensa a errores.

## Computador básico

Si bien ya podemos ejecutar programas simples:

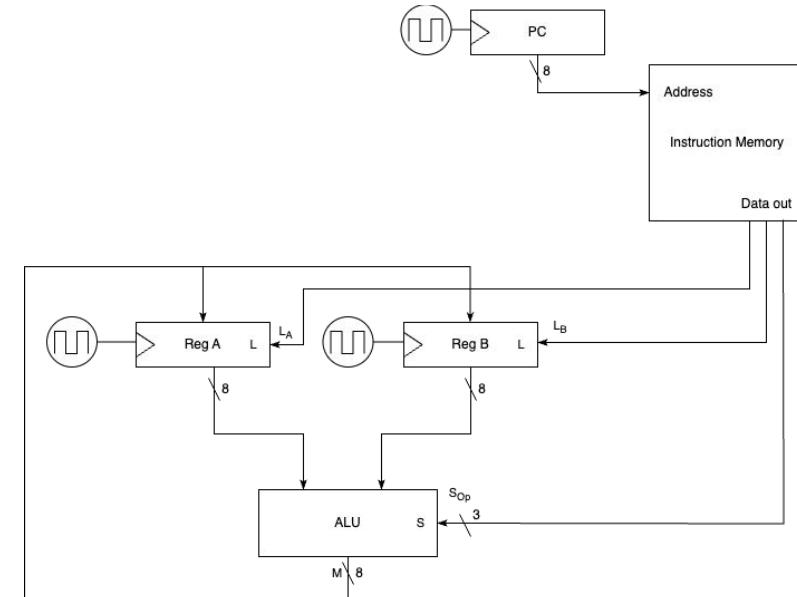
- No podemos inicializar los registros con valores iniciales. Hasta ahora asumimos que parten con un valor.
- No podemos almacenar más de dos datos simultáneamente.
- Estamos **muy limitados** respecto a los programas que podemos ejecutar.



## 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 nuestros componentes.

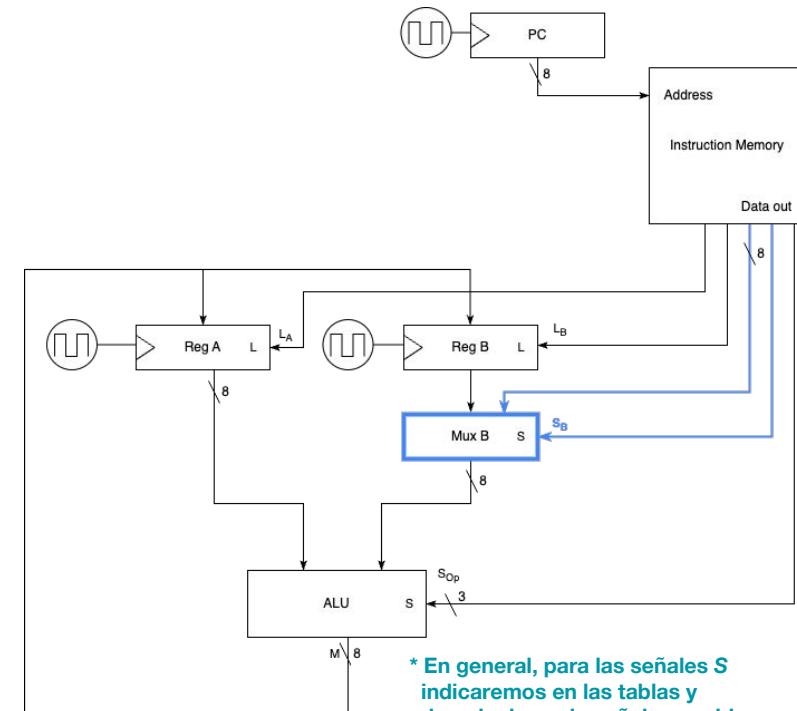


## Computador básico - Inicialización de registros

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

Esto habilita nuevas operaciones para  $A$ , por ejemplo:  **$A + 5$**   
( $L_A = 1$ ;  $S_B = \text{LIT}$ ;  $S_{Op} = \text{ADD}$ ).

Ahora, ¿cómo inicializamos valores?

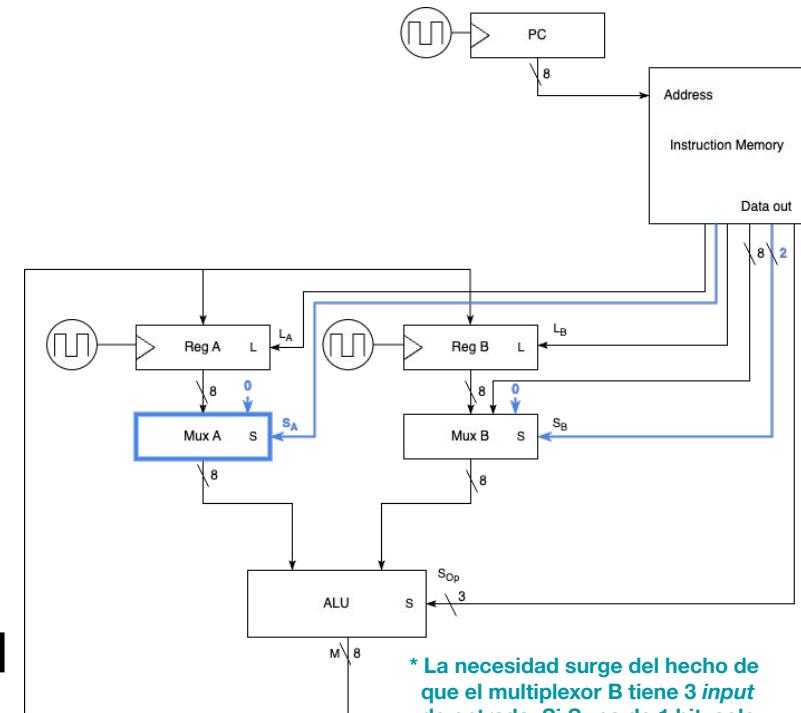


\* En general, para las señales  $S$  indicaremos en las tablas y descripciones la señal escogida, ya que los valores de  $S$  a los que se asocia cada entrada son arbitrarios.

## 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  $S_A$ .

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



\* La necesidad surge del hecho de que el multiplexor  $B$  tiene 3 input de entrada. Si  $S_B$  es de 1 bit, solo se pueden tener 2 entradas. Con 2 bits, se pueden tener hasta 4.

## Computador básico - Inicialización de registros

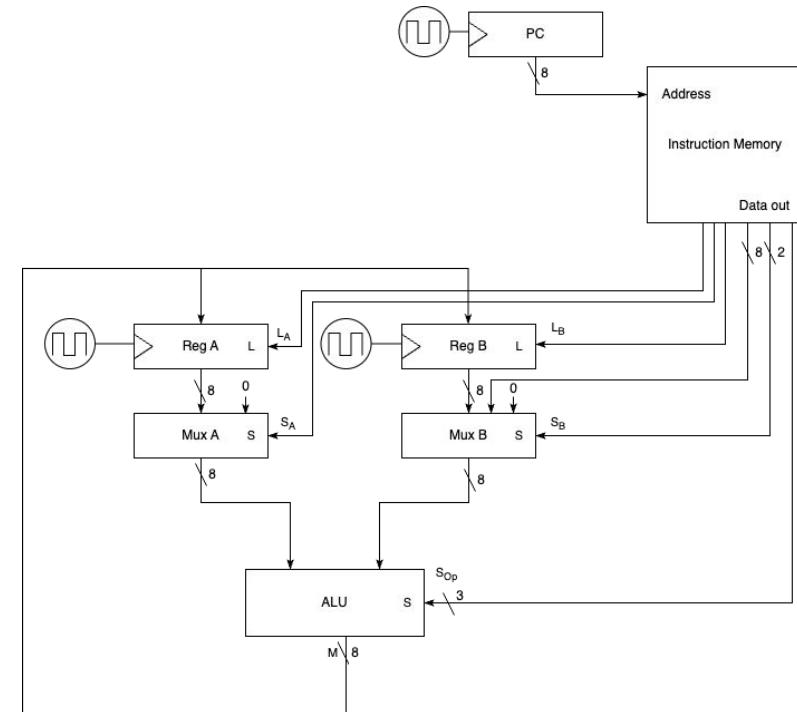
Entonces, para inicializar valores en  $A$  y en  $B$  realizamos las siguientes operaciones:

$$A = 0 + \text{LIT}$$

( $L_A = 1; S_A = 0; S_B = \text{LIT}, S_{Op} = \text{ADD}$ )

$$B = 0 + \text{LIT}$$

( $L_B = 1; S_A = 0; S_B = \text{LIT}, S_{Op} = \text{ADD}$ )



## Computador básico - Inicialización de registros

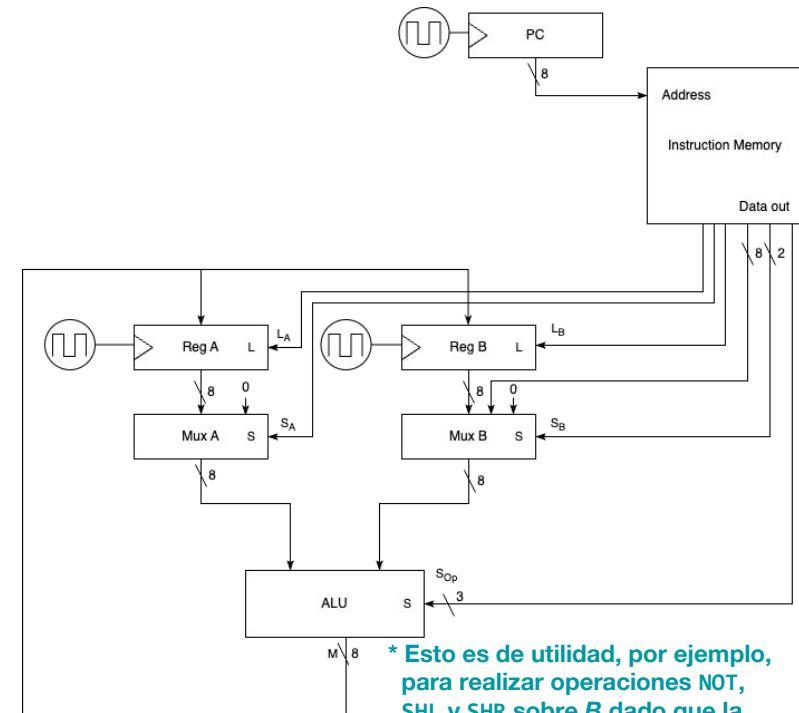
Con este nuevo esquema, también podemos copiar los valores de un registro a otro.

$$A = B + 0$$

( $L_A = 1$ ;  $S_A = 0$ ;  $S_B = B$ ,  $S_{Op} = \text{ADD}$ )

$$B = A + 0$$

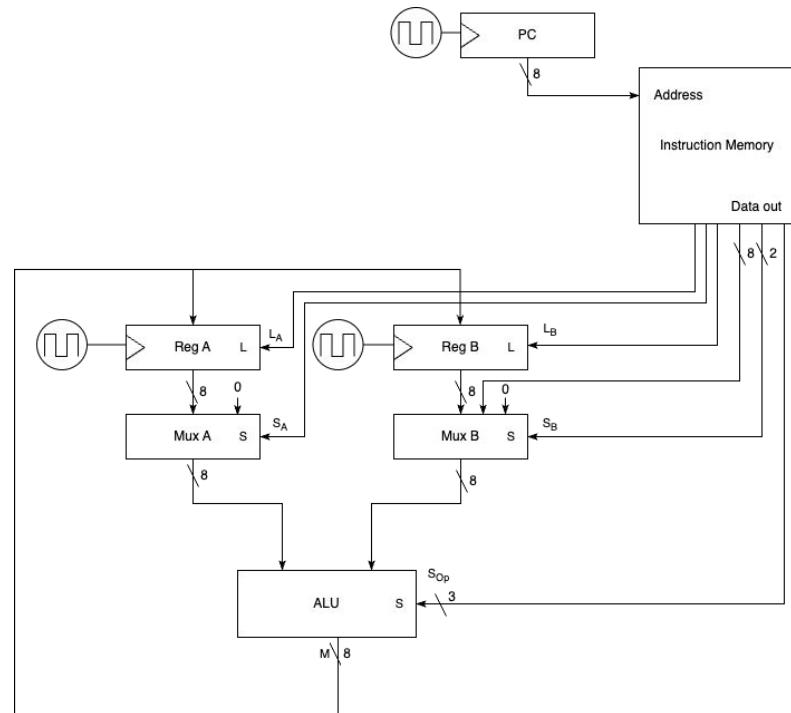
( $L_B = 1$ ;  $S_A = A$ ;  $S_B = 0$ ,  $S_{Op} = \text{ADD}$ )



\* 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

$L_A$	$L_B$	$S_A$	$S_{B[1]}$	$S_{B[0]}$	$S_2$	$S_1$	$S_0$	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	$A = \text{Lit}$
0	1	1	0	1	0	0	0	$B = \text{Lit}$
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	$A = A + \text{Lit}$
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	$A = A - \text{Lit}$
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

$L_A$	$L_B$	$S_A$	$S_{B[1]}$	$S_{B[0]}$	$S_2$	$S_1$	$S_0$	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	$A = \text{Lit}$
0	1	1	0	1	0	0	0	$B = \text{Lit}$
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	$A = A + \text{Lit}$
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	$A = A - \text{Lit}$
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	$L_A$	$L_B$	$S_A$	$S_{B[1]}$	$S_{B[0]}$	$S_2$	$S_1$	$S_0$	Operación
0000000	1	0	1	0	0	0	0	0	$A = B$
0000001	0	1	0	1	0	0	0	0	$B = A$
0000010	1	0	1	0	1	0	0	0	$A = \text{Lit}$
0000011	0	1	1	0	1	0	0	0	$B = \text{Lit}$
0000100	1	0	0	0	0	0	0	0	$A = A + B$
0000101	0	1	0	0	0	0	0	0	$B = A + B$
0000110	1	0	0	0	1	0	0	0	$A = A + \text{Lit}$
0000111	1	0	0	0	0	0	0	1	$A = A - B$
0001000	0	1	0	0	0	0	0	1	$B = A - B$
0001001	1	0	0	0	1	0	0	1	$A = A - \text{Lit}$
0001010	1	0	0	0	0	0	1	0	$A = A \text{ AND } B$
0001011	0	1	0	0	0	0	1	0	$B = A \text{ AND } B$
0001100	1	0	0	0	1	0	1	0	$A = A \text{ AND Lit}$
0001101	1	0	0	0	0	0	1	1	$A = A \text{ OR } B$
0001110	0	1	0	0	0	0	1	1	$B = A \text{ OR } B$
0001111	1	0	0	0	1	0	1	1	$A = A \text{ OR Lit}$
0010000	1	0	0	0	0	1	0	0	$A = \text{NOT } A$
0010001	0	1	0	0	0	1	0	0	$B = \text{NOT } A$
0010010	1	0	0	0	0	1	0	1	$A = A \text{ XOR } B$
0010011	0	1	0	0	0	1	0	1	$B = A \text{ XOR } B$
0010100	1	0	0	0	1	1	0	1	$A = A \text{ XOR Lit}$
0010101	1	0	0	0	0	1	1	0	$A = \text{SHL } A$
0010110	0	1	0	0	0	1	1	0	$B = \text{SHL } A$
0010111	1	0	0	0	0	1	1	1	$A = \text{SHR } A$
0011000	0	1	0	0	0	1	1	1	$B = \text{SHR } A$

Usamos *opcodes* (códigos de operación) que, con menos bits, definen la combinación de señales de control que ejecuta una instrucción. Esto optimiza el espacio utilizando en la memoria de instrucciones.

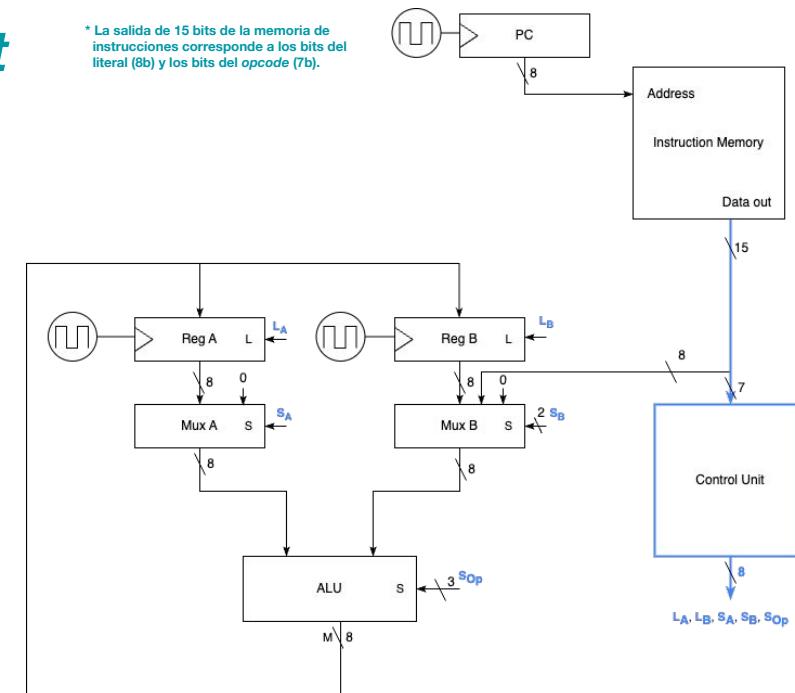
Ahora, falta un componente que decodifique *opcodes*.

## 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 ejecutan la instrucción de ingreso.

Este componente puede ser una ROM o un decodificador. Para efectos prácticos, la máquina funcionará de ambas formas.

\* La salida de 15 bits de la memoria de instrucciones corresponde a los bits del literal (8b) y los bits del *opcode* (7b).



\* Se asume que las señales de control ahora vienen de la **Control Unit**.

# Computador básico - Assembly

Opcode	L <sub>A</sub>	L <sub>B</sub>	S <sub>A</sub>	S <sub>B[1]</sub>	S <sub>B[0]</sub>	S <sub>2</sub>	S <sub>1</sub>	S <sub>0</sub>	Operación
0000000	1	0	1	0	0	0	0	0	A = B
0000001	0	1	0	1	0	0	0	0	B = A
0000010	1	0	1	0	1	0	0	0	A = Lit
0000011	0	1	1	0	1	0	0	0	B = Lit
0000100	1	0	0	0	0	0	0	0	A = A + B
0000101	0	1	0	0	0	0	0	0	B = A + B
0000110	1	0	0	0	1	0	0	0	A = A + Lit
0000111	1	0	0	0	0	0	0	1	A = A - B
0001000	0	1	0	0	0	0	0	1	B = A - B
0001001	1	0	0	0	1	0	0	1	A = A - Lit
0001010	1	0	0	0	0	0	1	0	A = A AND B
0001011	0	1	0	0	0	0	1	0	B = A AND B
0001100	1	0	0	0	1	0	1	0	A = A AND Lit
0001101	1	0	0	0	0	0	1	1	A = A OR B
0001110	0	1	0	0	0	0	1	1	B = A OR B
0001111	1	0	0	0	1	0	1	1	A = A OR Lit
0010000	1	0	0	0	0	1	0	0	A = NOT A
0010001	0	1	0	0	0	1	0	0	B = NOT A
0010010	1	0	0	0	0	1	0	1	A = A XOR B
0010011	0	1	0	0	0	1	0	1	B = A XOR B
0010100	1	0	0	0	1	1	0	1	A = A XOR Lit
0010101	1	0	0	0	0	1	1	0	A = SHL A
0010110	0	1	0	0	0	1	1	0	B = SHL A
0010111	1	0	0	0	0	1	1	1	A = SHR A
0011000	0	1	0	0	0	1	1	1	B = SHR A

Este esquema sigue teniendo un grado de dificultad al momento de programar, ya que sigue dependiendo de la representación binaria.

Para evitarlo, haremos uso del lenguaje de programación de bajo nivel **Assembly**.

# Computador básico - Assembly

Instrucción	Operandos	Opcode	$L_A$	$L_B$	$S_A$	$S_{B[1]}$	$S_{B[0]}$	$S_2$	$S_1$	$S_0$	Operación
MOV	A,B	0000000	1	0	1	0	0	0	0	0	$A = B$
	B,A	0000001	0	1	0	1	0	0	0	0	$B = A$
	A,Lit	0000010	1	0	1	0	1	0	0	0	$A = \text{Lit}$
	B,Lit	0000011	0	1	1	0	1	0	0	0	$B = \text{Lit}$
ADD	A,B	0000100	1	0	0	0	0	0	0	0	$A = A + B$
	B,A	0000101	0	1	0	0	0	0	0	0	$B = A + B$
	A,Lit	0000110	1	0	0	0	1	0	0	0	$A = A + \text{Lit}$
SUB	A,B	0000111	1	0	0	0	0	0	0	1	$A = A - B$
	B,A	0001000	0	1	0	0	0	0	0	1	$B = A - B$
	A,Lit	0001001	1	0	0	0	1	0	0	1	$A = A - \text{Lit}$
AND	A,B	0001010	1	0	0	0	0	0	1	0	$A = A \text{ AND } B$
	B,A	0001011	0	1	0	0	0	0	1	0	$B = A \text{ AND } B$
	A,Lit	0001100	1	0	0	0	1	0	1	0	$A = A \text{ AND } \text{Lit}$
OR	A,B	0001101	1	0	0	0	0	0	1	1	$A = A \text{ OR } B$
	B,A	0001110	0	1	0	0	0	0	1	1	$B = A \text{ OR } B$
	A,Lit	0001111	1	0	0	0	1	0	1	1	$A = A \text{ OR } \text{Lit}$
NOT	A,A	0010000	1	0	0	0	0	1	0	0	$A = \text{NOT } A$
	B,A	0010001	0	1	0	0	0	1	0	0	$B = \text{NOT } A$
XOR	A,B	0010010	1	0	0	0	0	1	0	1	$A = A \text{ XOR } B$
	B,A	0010011	0	1	0	0	0	1	0	1	$B = A \text{ XOR } B$
	A,Lit	0010100	1	0	0	0	1	1	0	1	$A = A \text{ XOR } \text{Lit}$
SHL	A,A	0010101	1	0	0	0	0	1	1	0	$A = \text{SHL } A$
	B,A	0010110	0	1	0	0	0	1	1	0	$B = \text{SHL } A$
SHR	A,A	0010111	1	0	0	0	0	1	1	1	$A = \text{SHR } A$
	B,A	0011000	0	1	0	0	0	1	1	1	$B = \text{SHR } A$

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 de un **Assembler**, el que se encarga de reemplazar cada instrucción por el código correspondiente.

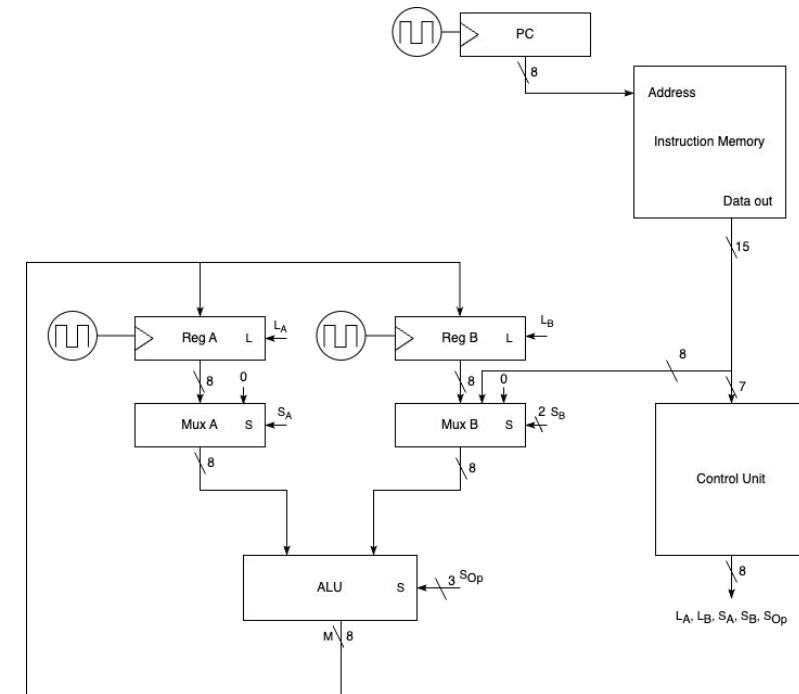
## Computador básico - Assembly

**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.

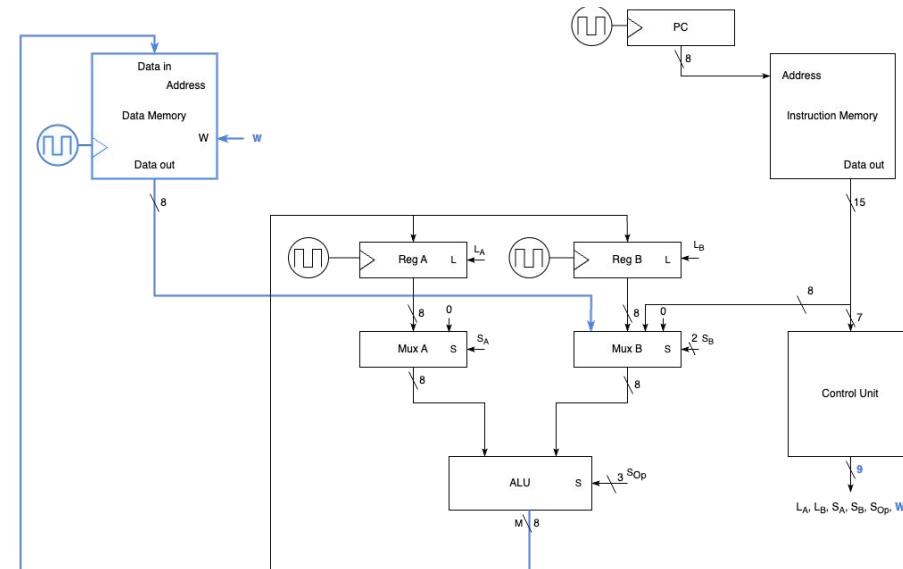
¿Qué componente podríamos agregar?



## 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, se sincroniza con el *clock* del sistema. Por otra parte, usa el *slot* disponible del Mux B para poder utilizar su valor en la ALU.

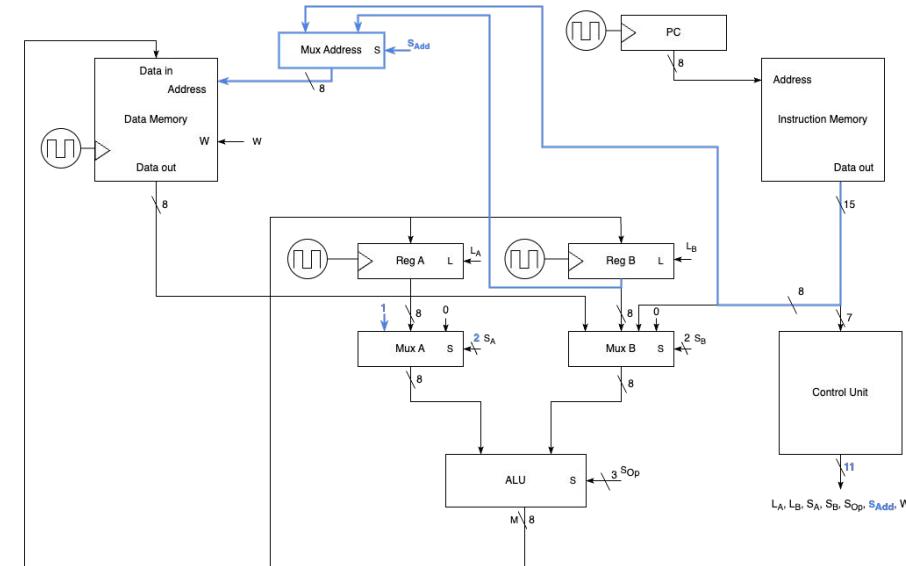


## Computador básico - Direccionamiento

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.

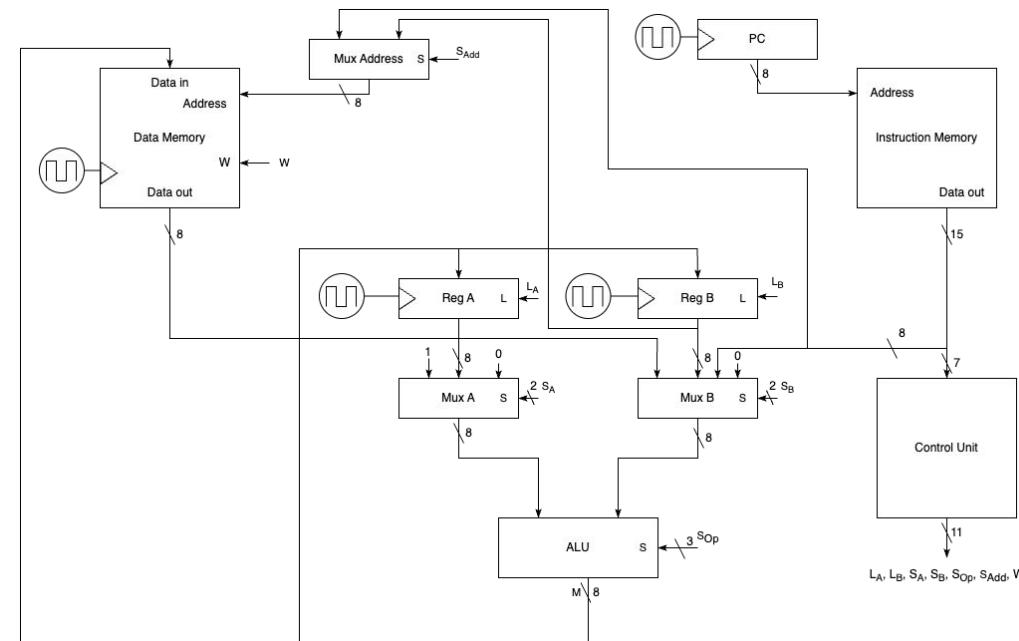
Se selecciona uno de los dos con un nuevo Mux Address.



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)	$A = \text{Mem}[\text{Dir}]$	MOV A,(var0)
	B,(Dir)	$B = \text{Mem}[\text{Dir}]$	MOV B,(var1)
	(Dir),A	$\text{Mem}[\text{Dir}] = A$	MOV (var0),A
	(Dir),B	$\text{Mem}[\text{Dir}] = B$	MOV (var1),B
	A,(B)	$A = \text{Mem}[B]$	-
	B,(B)	$B = \text{Mem}[B]$	-
ADD	(B),A	$\text{Mem}[B] = A$	-
	A,(Dir)	$A = A + \text{Mem}[\text{Dir}]$	ADD A,(var0)
	A,(B)	$A = A + \text{Mem}[B]$	-
	(Dir)	$\text{Mem}[\text{Dir}] = A + B$	ADD (var0)
SUB	A,(Dir)	$A = A - \text{Mem}[\text{Dir}]$	SUB A,(var0)
	A,(B)	$A = A - \text{Mem}[B]$	-
	(Dir)	$\text{Mem}[\text{Dir}] = A - B$	SUB (var0)
AND	A,(Dir)	$A = A \text{ AND } \text{Mem}[\text{Dir}]$	AND A,(var0)
	A,(B)	$A = A \text{ AND } \text{Mem}[B]$	-
	(Dir)	$\text{Mem}[\text{Dir}] = A \text{ AND } B$	AND (var0)
OR	A,(Dir)	$A = A \text{ OR } \text{Mem}[\text{Dir}]$	OR A,(var0)
	A,(B)	$A = A \text{ OR } \text{Mem}[B]$	-
	(Dir)	$\text{Mem}[\text{Dir}] = A \text{ OR } B$	OR (var0)
NOT	(Dir)	$\text{Mem}[\text{Dir}] = \text{NOT } A$	NOT (var0)
XOR	A,(Dir)	$A = A \text{ XOR } \text{Mem}[\text{Dir}]$	XOR A,(var0)
	A,(B)	$A = A \text{ XOR } \text{Mem}[B]$	-
	(Dir)	$\text{Mem}[\text{Dir}] = A \text{ XOR } B$	XOR (var0)
SHL	(Dir)	$\text{Mem}[\text{Dir}] = \text{SHL } A$	SHL (var0)
SHR	(Dir)	$\text{Mem}[\text{Dir}] = \text{SHR } A$	SHR (var0)
INC	B	$B = B + 1$	-
	(B)	$\text{Mem}[B] = \text{Mem}[B] + 1$	-
	(Dir)	$\text{Mem}[\text{Dir}] = \text{Mem}[\text{Dir}] + 1$	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, contamos con dos segmentos: **DATA** (definición de variables y arreglos) y **CODE** (definición de instrucciones).

El *Assembler* mapea las variables de DATA a direcciones de memoria a través de operaciones MOV, como veremos ahora.

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 - Transformación del Assembler

## ¿Cómo se traduce un programa a lenguaje de máquina?

```
DATA:  
arr 1  
    2  
len 2  
CODE:  
MOV A,(arr)  
MOV B,(len)
```

```
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  
MOV A,(0) ;dirección arr  
MOV B,(2) ;dirección len
```

Opcode Literal

Opcode	Literal
0000010	000000001
0000110	000000000
0000010	000000010
0000110	000000001
0000010	000000010
0000110	000000010
0000100	000000000
0000101	000000010

El Assembler asocia la dirección 0 a arr y la dirección 2 a len, reemplazando los *labels* por sus literales asociados. Luego, agrega al principio del programa las instrucciones que escriben los datos de las variables en la RAM.

Con las instrucciones de escritura añadidas, las traduce todas al *opcode* asociado, agregando a estas el literal que les corresponde (por ejemplo, para MOV A,2 el literal es 2; para MOV (1),A el literal es 1; etc.).

## 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 recorremos todo el arreglo y duplicamos el valor de cada uno de sus elementos.

### DATA:

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

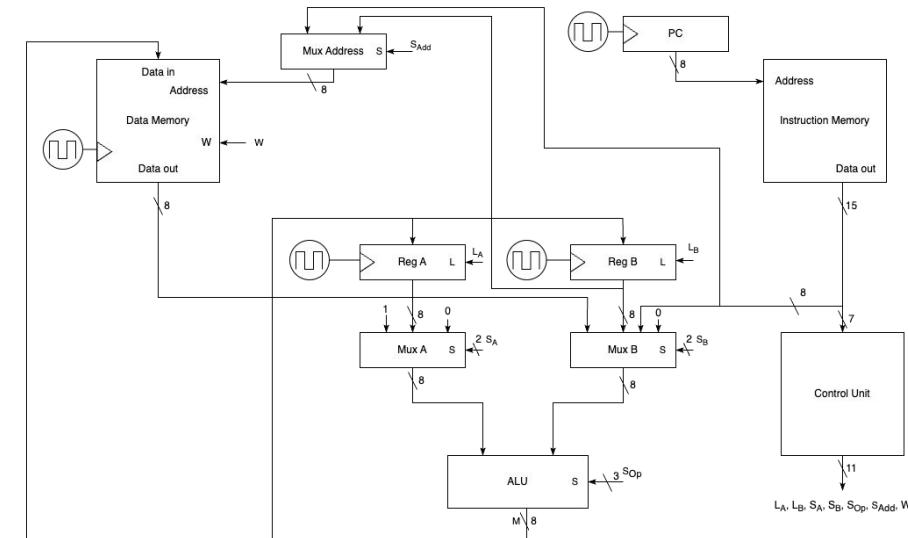
### CODE:

```
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
```

## Computador básico - *Register File*

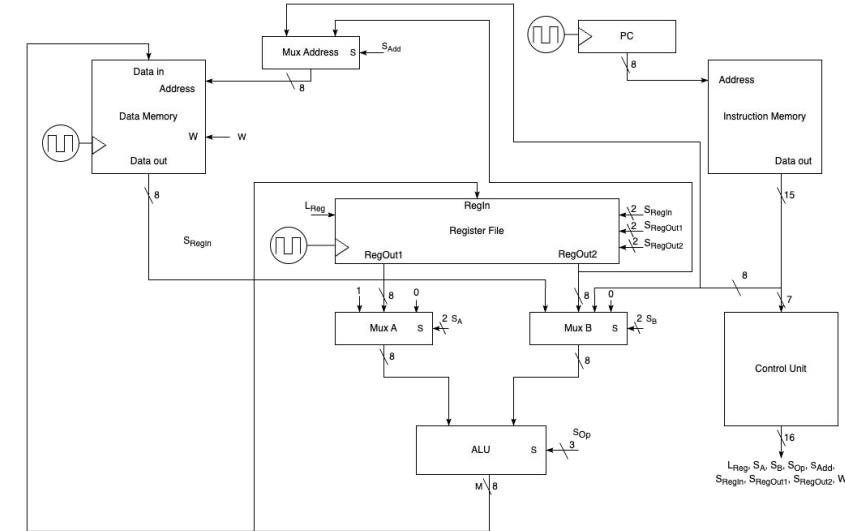
Si bien nuestro computador actual cuenta con solo dos registros, en la práctica se pueden tener varios.

Para asegurar la escalabilidad independiente de la cantidad de registros, se puede hacer uso de un ***Register File***, que corresponde a un **banco de registros**.



## Computador básico - *Register File*

- RegIn: Entrada de datos. Si la señal  $L_{Reg}$  se habilita, el dato se guarda en el registro seleccionado por la señal  $S_{RegIn}$ .
- RegOut<sub>1</sub>: Primera salida de datos. Corresponde al valor del registro seleccionado por la señal  $S_{RegOut1}$ .
- RegOut<sub>2</sub>: Segunda salida de datos. Corresponde al valor del registro seleccionado por la señal  $S_{RegOut2}$ .



La cantidad de bits de  $S_{RegIn}$ ,  $S_{RegOut1}$  y  $S_{RegOut2}$  depende de la cantidad de registros dentro del **Register File**. En este caso, se consideran 4 registros.

## Errores comunes

A continuación, veremos los errores más comunes que cometan en las interrogaciones de forma que puedan evitarlos.

**PEBCAC**

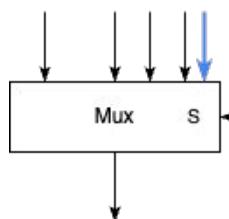


**Problem Exists Between  
Chair and Computer**

## Errores comunes

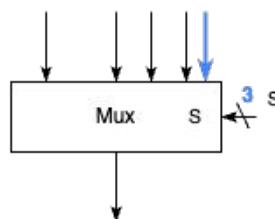
### Adición de señales al multiplexor

- Al añadir una nueva señal de entrada a un multiplexor, se debe **cuidar la cantidad de bits de la señal de selección S asociada.**
- **Ejemplo:** Adición de una nueva señal a un multiplexor que ya poseía 4 entradas.



**INCORRECTO**

Con 2 bits de selección, solo se pueden escoger  $2^2 = 4$  entradas distintas.



**CORRECTO**

Con 3 bits de selección, solo se pueden escoger  $2^3 = 8$  entradas distintas. No hay problema con que "sobren" entradas.

## Errores comunes

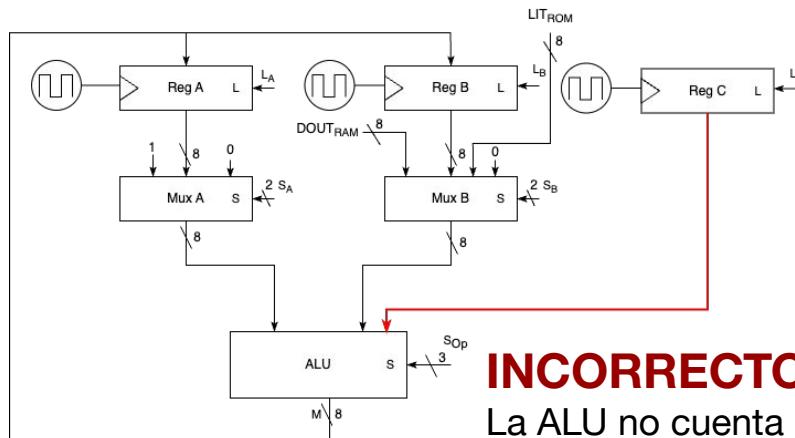
### Conexiones directas a la ALU

- En contexto de evaluación, lo más común es solicitar modificaciones sobre la arquitectura del computador básico.
- Una de ellas consiste en incluir nuevos valores para operar dentro de la ALU. No obstante, **a la ALU no se le pueden agregar nuevas conexiones directamente**. Lo correcto es hacer uso de los multiplexores ya existentes.

## Errores comunes

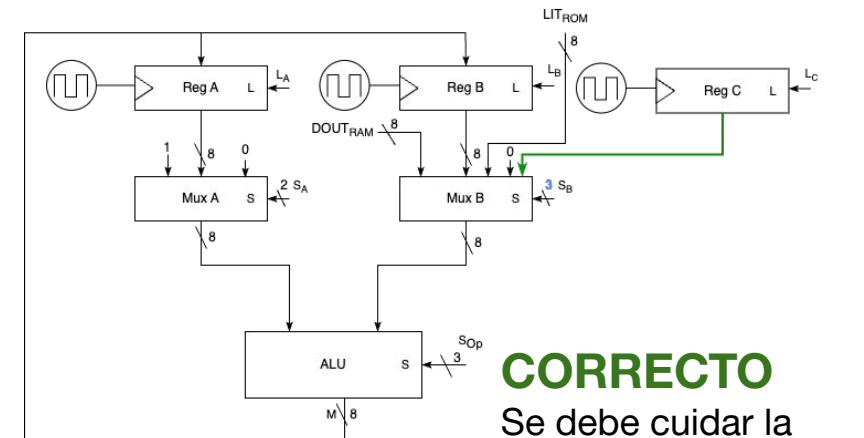
### Conexiones directas a la ALU

**Ejemplo:** Agregue un tercer registro C que pueda ser operado como operando *B* de la ALU.



**INCORRECTO**

La ALU no cuenta con una  
entrada C para operandos



**CORRECTO**

Se debe cuidar la  
cantidad de bits de  $S_B$

## Errores comunes

### Conexiones directas a los registros

- Similar al caso anterior, en ocasiones se les solicita dar la posibilidad de almacenar nuevos valores en los registros.
- Esto se puede hacer a partir de la ALU, o bien añadiendo un nuevo multiplexor cuya salida se conecte con la entrada de datos de un registro. Lo que **no se debe hacer es conectar lo solicitado directamente como entrada del registro.**

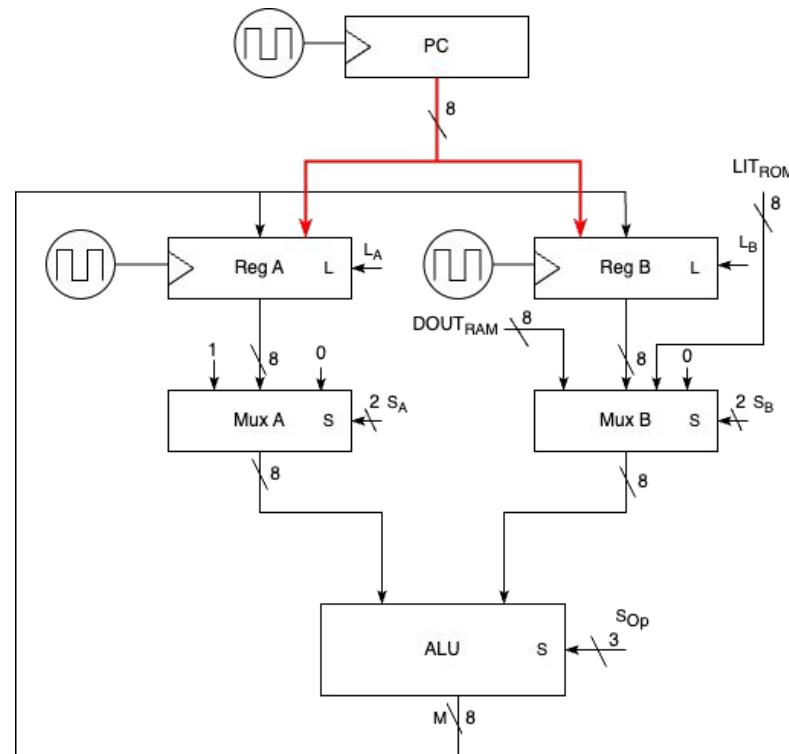
## Erros comunes

# Conexiones directas a los registros

**Ejemplo:** Modifique la arquitectura para almacenar en los registros A y B el valor del contador PC.

# INCORRECTO

Los registros solo pueden recibir un dato de entrada. ¿Qué valor estaríamos escogiendo en este escenario? No se podría determinar en una corrección y, en cualquier caso, no respetaría la estructura de un registro.



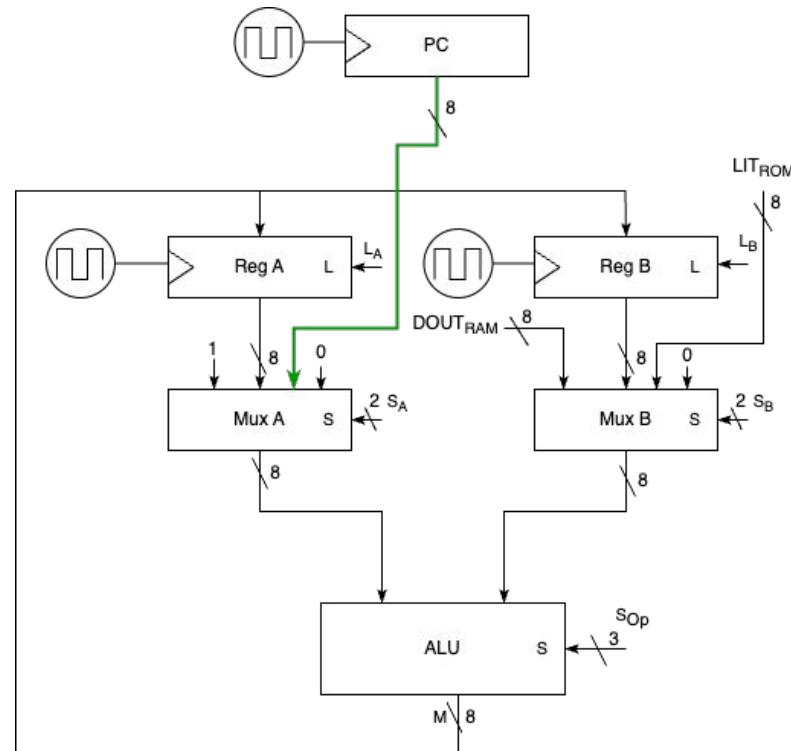
## Errores comunes

### Conexiones directas a los registros

**Ejemplo:** Modifique la arquitectura para almacenar en los registros A y B el valor del contador PC.

#### CORRECTO

**Alternativa 1:** Podemos seleccionarlo en uno de los multiplexores de la ALU. En este caso, si  $S_A = \text{PC}$ ,  $S_B = \text{ZERO}$ ,  $S_{OP} = \text{ADD}$  y  $L_A = 1$ , entonces cargaremos en el registro A el valor de PC.



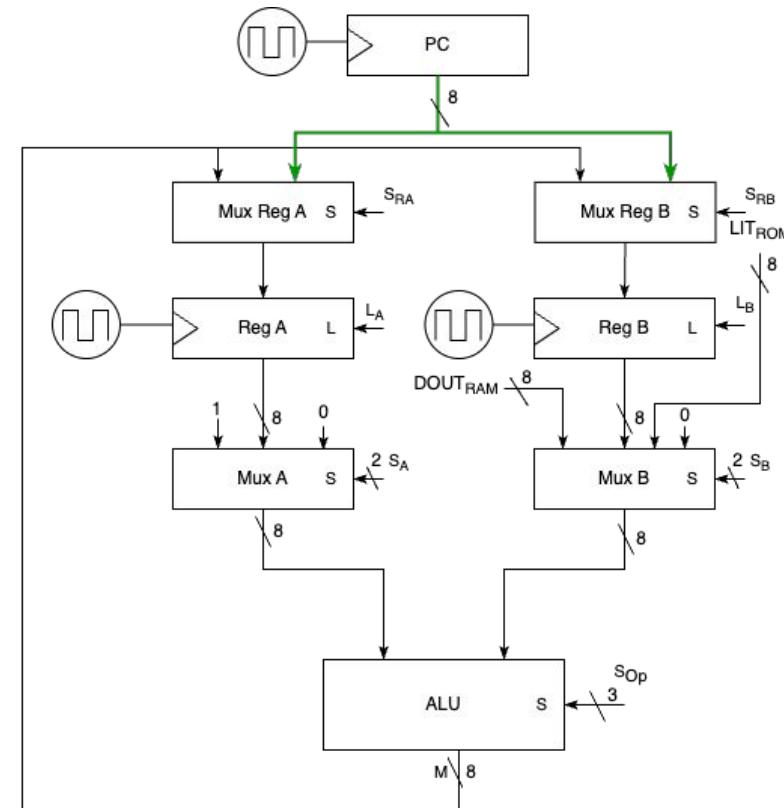
## Errores comunes

### Conexiones directas a los registros

**Ejemplo:** Modifique la arquitectura para almacenar en los registros A y B el valor del contador PC.

#### CORRECTO

**Alternativa 2:** Podemos agregar multiplexores para seleccionar el dato a guardar en cada registro. En este caso, si  $S_{RA} = PC$  y  $L_A = 1$ , entonces cargaremos en el registro A el valor de PC.



## Errores comunes

### Señales de selección en una combinación de señales

- Cuando se agrega una nueva instrucción al computador, es necesario incluir la combinación de señales que la ejecuta.
- Esta señal de selección debe llevar **el nombre de la señal de entrada** y **no su combinación de bits**.

$$S_A = 00$$

**INCORRECTO**

No tenemos certeza de la señal que están seleccionando.

$$S_A = A$$

**CORRECTO**

No queda duda alguna de la señal que seleccionan.

\* Cuando se selecciona un 0 o un 1 en un multiplexor, las diferenciamos de una combinación de bits llamándolas ZERO y ONE respectivamente.  
Ejemplo:  $S_A = \text{ZERO}$ ,  $S_B = \text{ONE}$

## Errores comunes

### Señales de selección en una combinación de señales

- Otro error relacionado es **darle un valor a una señal de selección cuando no incide en el resultado de una operación.**
- **Ejemplo:** Valor de la señal  $S_{Add}$  cuando no se realiza una operación de lectura o escritura en la memoria de datos.

$$S_{Add} = \text{LIT}$$

**INCORRECTO**

El valor de la dirección de memoria no cambia el resultado de la operación.

$$S_{Add} = -$$

**CORRECTO**

Se explicita que su valor no es relevante para la operación.

\* Siempre denotaremos con “-” el valor de una señal cuyo valor no es relevante para la ejecución de una operación.

## Errores comunes

### Señales de carga en una combinación de señales

- A diferencia de las señales de selección, a las de carga **siempre se les debe dar un valor**. Si no es relevante, **se les debe asignar explícitamente un cero**.
- **Ejemplo:** Ejecución de la instrucción **MOV A, (dir)**.

$$\begin{array}{l} L_B = - \\ W = - \end{array}$$

**INCORRECTO**

Si su valor no fuera relevante, se podrían habilitar ambos, pero el estado final sería incorrecto.

$$\begin{array}{l} L_B = 0 \\ W = 0 \end{array}$$

**CORRECTO**

Se explicita que su valor no es relevante para la operación y el estado final será correcto.

## 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
```

**Ayudantía 2, 2018-2, Germán Contreras**

## Ejercicios

Considere el siguiente programa.

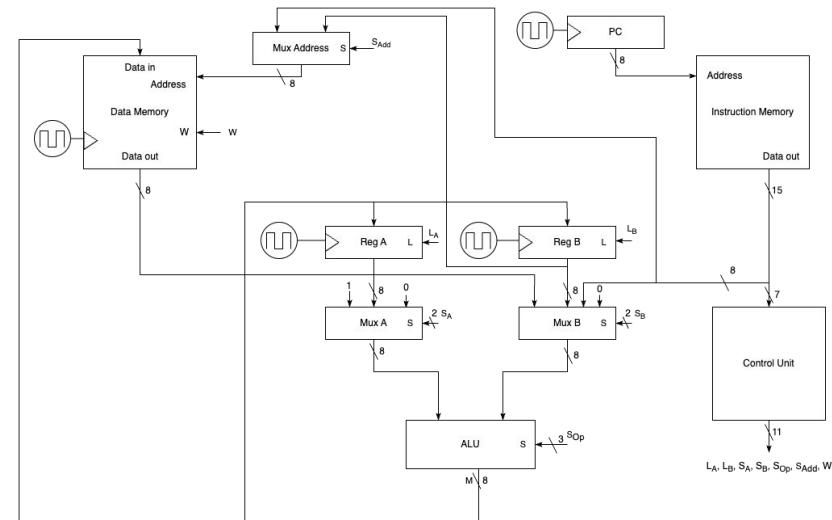
Ahora, programe en Assembly un código que obtenga el mismo resultado de  $x$  e  $y$ , pero sin hacer uso de la variable  $z$  en el segmento DATA.

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

**Ayudantía 2, 2018-2, Germán Contreras**

## Ejercicios

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.



*Ayudantía 2, 2018-2, Germán Contreras*

## 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.

*Interrogación 1, 2016-1*

## Ejercicios

Modifique el diagrama del computador básico sin soporte de saltos para habilitar las siguientes instrucciones en una sola iteración:

- Op1 B, (A);  $Op1 \in \{MOV, ADD, SUB, AND, OR, XOR\}$
- Op1 A, (A);  $Op1 \in \{MOV, ADD, SUB, AND, OR, XOR\}$
- INC A
- INC (A)

En resumen, se le pide dar soporte para **direcciónamiento indirecto a través del registro A**. No es necesario asociar un *opcode* a las instrucciones anteriores, pero sí debe incluir la combinación de señales que ejecuta cada una de ellas. Además, debe indicar si alguna de las combinaciones de señales de instrucciones existentes se ve modificada por su diseño.

## Tarea 2, 2023-1

## Ejercicios

Escriba un programa en Assembly del computador básico que calcule el inverso aditivo de un número entero de 8 bits.

*Interrogación 2, 2014-2*

## 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.**

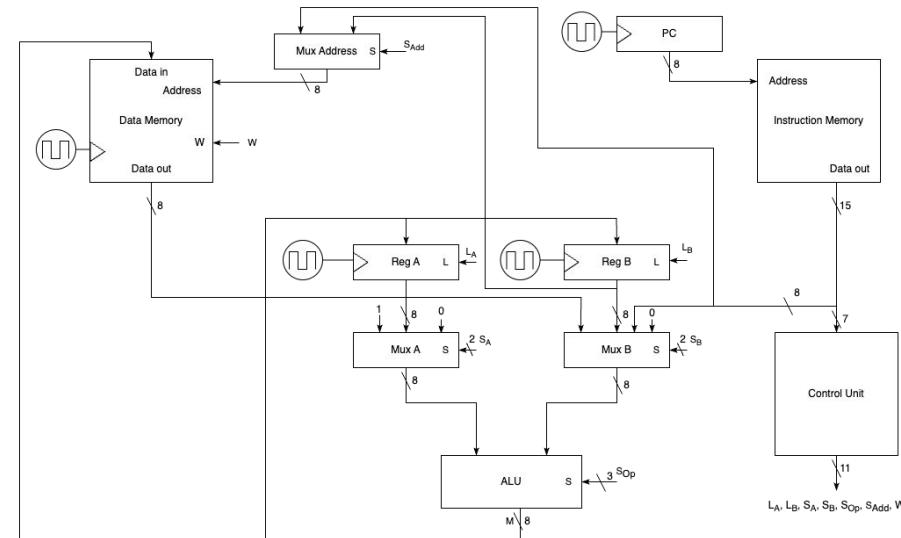
*Interrogación 1, 2015-2*

## 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?

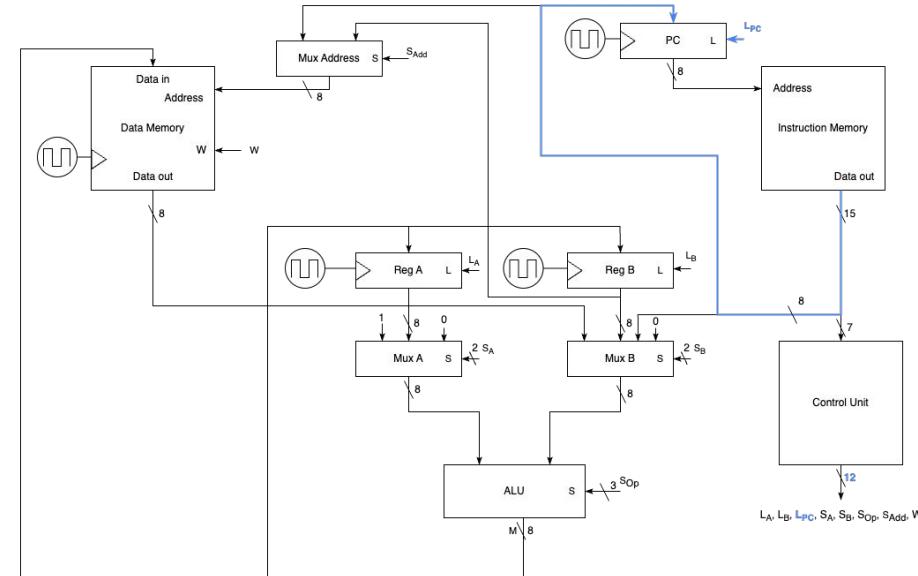
En el registro responsable de la ejecución de instrucciones: **PC**.



## Ejercicios - Desarrollo en clases

Se agrega la señal de control  $L_{PC}$  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.



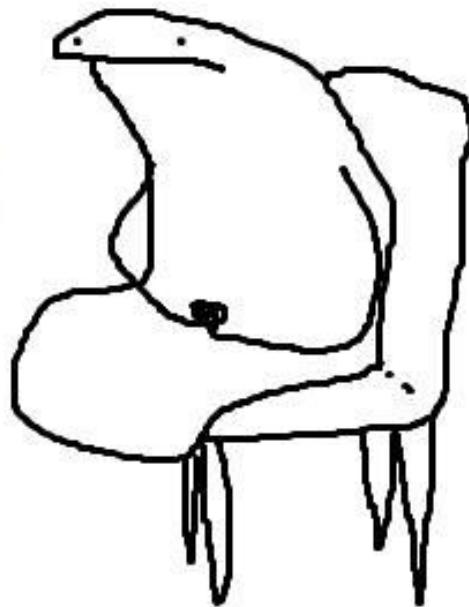
## Antes de terminar

¿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*.

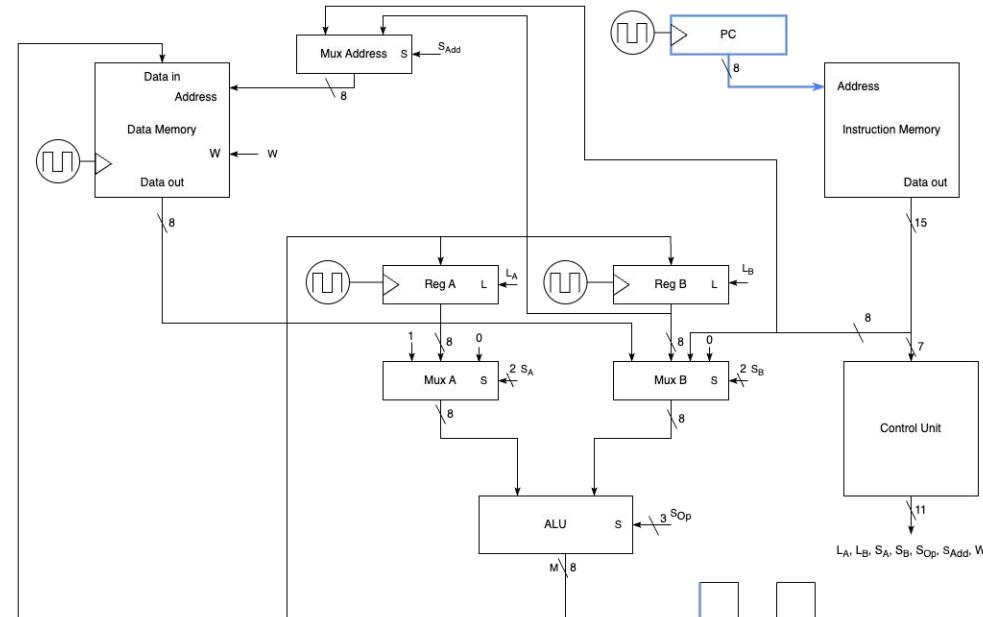
¿Cómo se logra esto? Lo veremos a continuación viendo el paso a paso de la ejecución en la microarquitectura de nuestro computador.

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

### Ciclo 1 - Flanco de subida

Parte la ejecución de la instrucción INC (B).

Por el flanco de subida, PC incrementa en una unidad su valor (que corresponderá a la instrucción INC (B) y, consecuentemente, se leerá dicha instrucción de la memoria de instrucciones).

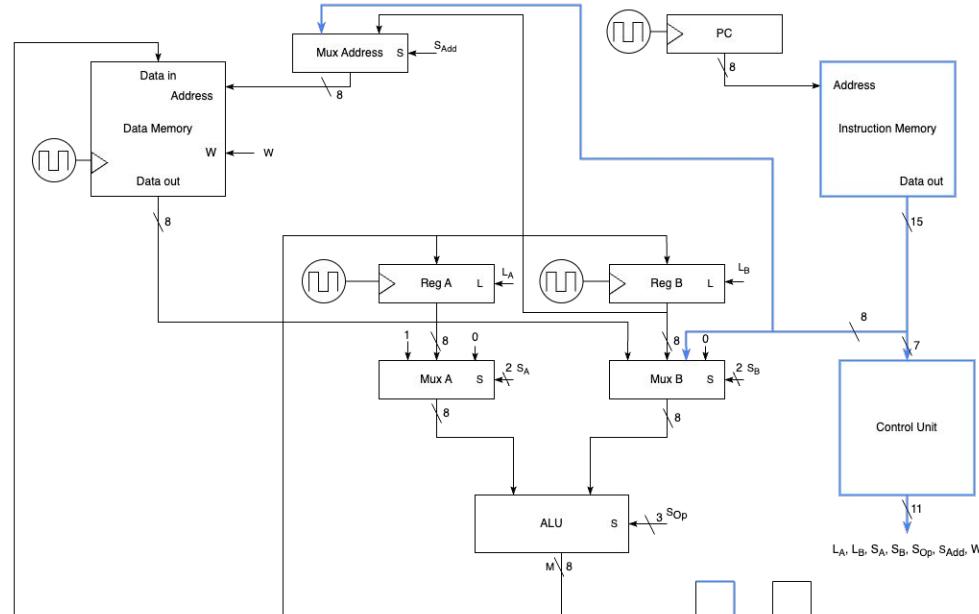


## Anexo - Ejecución de la instrucción INC (B)

### Ciclo 1 - Resto de la iteración

Se lee la instrucción en la memoria de instrucciones y se propaga tanto el literal de esta (que en este caso no se utiliza) como su *opcode*, el que es traducido por la *Control Unit* en las siguientes señales:

- $L_A = 0$  (no se actualiza el registro A)
- $L_B = 0$  (no se actualiza el registro B)
- $S_A = 1$  (señal constante “1”)
- $S_B = DOUT$  (salida de la RAM)
- $S_{Add} = B$  (direcccionamiento indirecto)
- $W = 1$  (habilitamos escritura en la RAM)

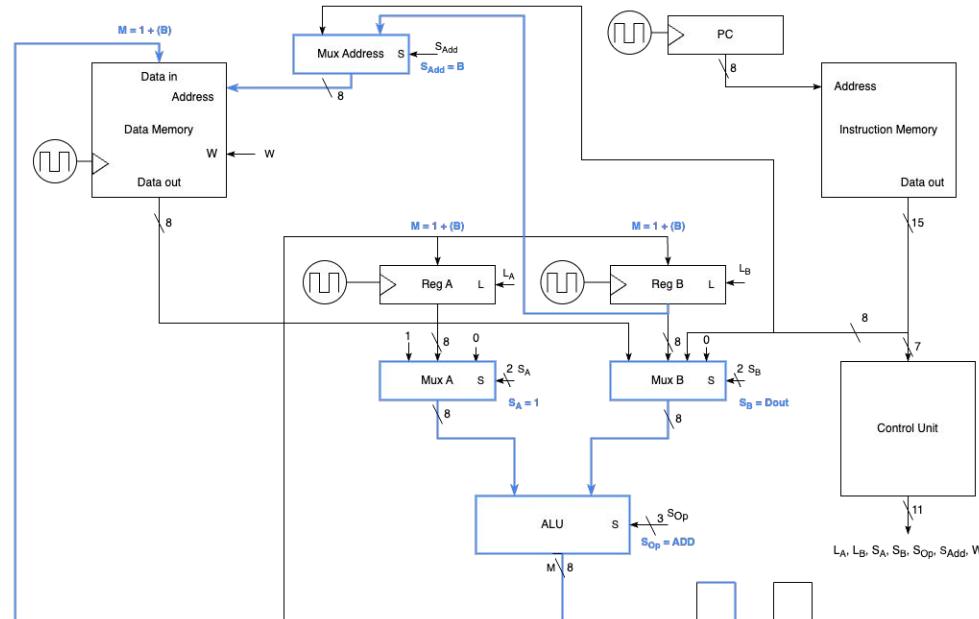


## Anexo - Ejecución de la instrucción INC (B)

### Ciclo 1 - Resto de la iteración

Con las señales de control transmitidas, ocurre lo siguiente:

1. Se lee la dirección (B) en la memoria de datos y se transmite al Mux B.
2. Se selecciona dicha entrada del Mux B y la entrada “1” del Mux A.
3. Ambos se transmiten a la ALU y se ejecuta la operación  $1 + \text{Mem}[B]$ .
4. El resultado se transmite a la entrada DataIn de la RAM. Aunque la señal W esté activa, la escritura no se realiza hasta un flanco de subida.

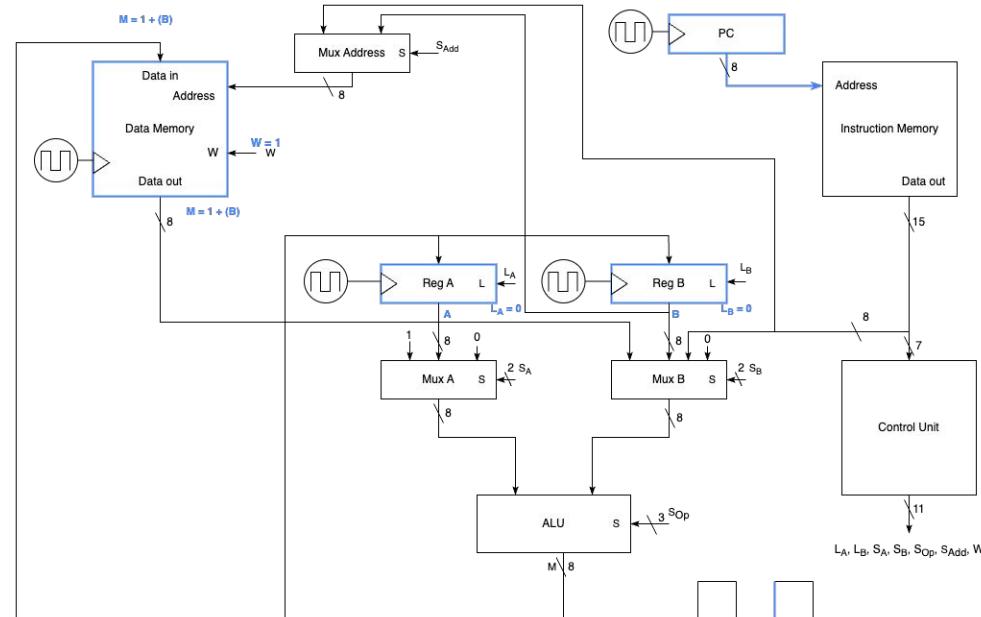


## Anexo - Ejecución de la instrucción INC (B)

### Ciclo 2 - Flanco de subida

Se activa la actualización de los registros A, B, PC y RAM. Como  $L_A = L_B = 0$ , los registros mantienen sus valores, pero la RAM actualiza su valor en la dirección B con  $1 + (B)$  (es decir, incrementa en una unidad su valor).

Como PC también actualiza su valor, se actualiza la dirección de la instrucción a ejecutar. 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

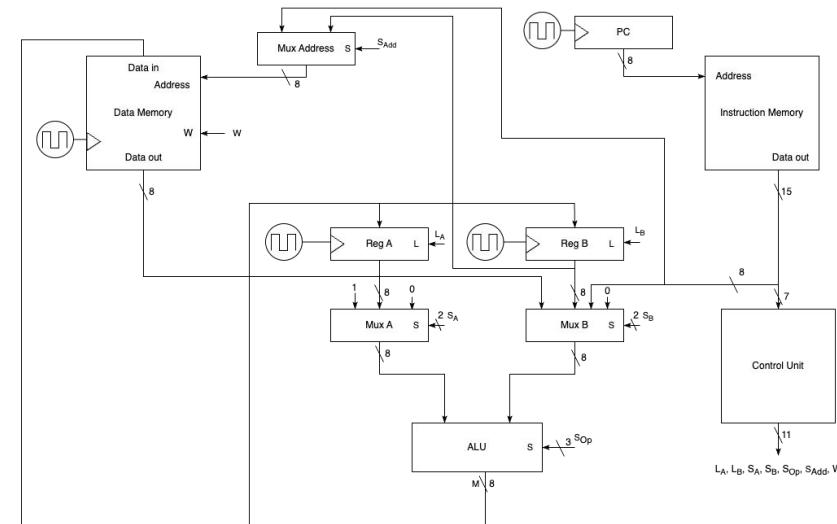
CODE:

```
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  $y$ ) 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.

## Ejercicios - Respuesta

Modifique el diagrama del computador básico sin soporte de saltos para habilitar las siguientes instrucciones en una sola iteración:

- Op1 B, (A); Op1  $\in \{\text{MOV, ADD, SUB, AND, OR, XOR}\}$
- Op1 A, (A); Op1  $\in \{\text{MOV, ADD, SUB, AND, OR, XOR}\}$
- INC A
- INC (A)

En resumen, se le pide dar soporte para **direcciónamiento indirecto a través del registro A**. No es necesario asociar un *opcode* a las instrucciones anteriores, pero sí debe incluir la combinación de señales que ejecuta cada una de ellas. Además, debe indicar si alguna de las combinaciones de señales de instrucciones existentes se ve modificada por su diseño.

**Respuesta en la siguiente diapositiva.**

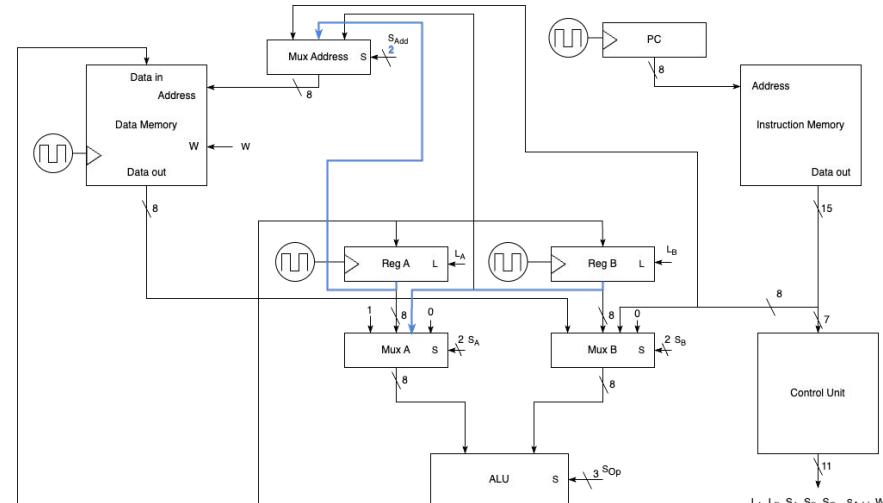
# Ejercicios - Respuesta

**Solución:** Las modificaciones mínimas que requiere el diagrama son las siguientes:

- Agregar una conexión entre el registro *A* y el *input* de direccionamiento de la memoria de datos. La forma más directa es conectando dicho registro con el componente **Mux Address**, lo que a su vez implica aumentar a 2 bits el bus de selección de este.
- Agregar una conexión entre el registro *B* y el componente **Mux A**. Esto tiene como fin poder hacer instrucciones del tipo **Op1 B, (A)** en el orden esperado (donde *B* es el primer operando).

La descripción anterior se traduce en el siguiente diagrama y tabla de instrucciones:

Instrucción	Operandos	Literal	La	Lb	Sa	Sb	Sop	Sadd	W
ADD	B,(A)	-	0	1	B	DOUT	ADD	A	0
	A,(A)	-	1	0	A	DOUT	ADD	A	0
SUB	B,(A)	-	0	1	B	DOUT	SUB	A	0
	A,(A)	-	1	0	A	DOUT	SUB	A	0
AND	B,(A)	-	0	1	B	DOUT	AND	A	0
	A,(A)	-	1	0	A	DOUT	AND	A	0
OR	B,(A)	-	0	1	B	DOUT	OR	A	0
	A,(A)	-	1	0	A	DOUT	OR	A	0
XOR	B,(A)	-	0	1	B	DOUT	XOR	A	0
	A,(A)	-	1	0	A	DOUT	XOR	A	0
INC	A	1	1	0	A	LIT	ADD	-	0
	(A)	-	0	0	ONE	DOUT	ADD	A	1



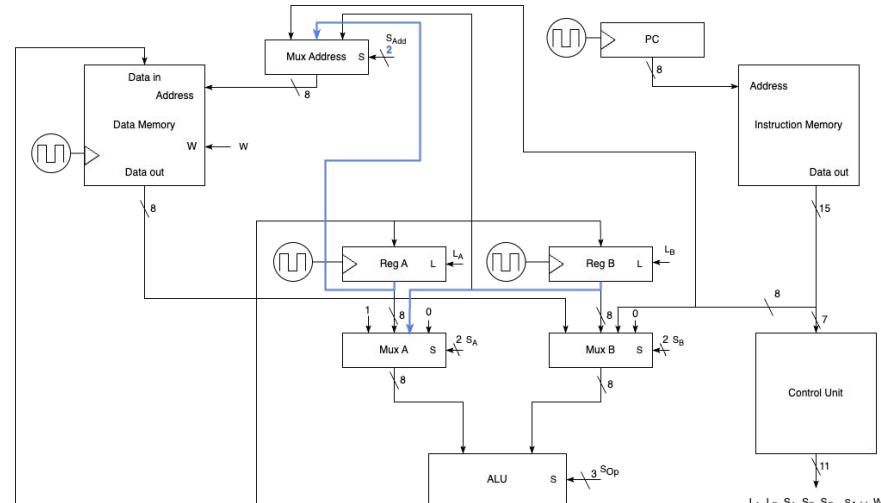
# Ejercicios - Respuesta

DOUT representa la salida de la memoria de datos; LIT representa el literal obtenido de la memoria de instrucciones; y ONE representa la selección de la señal constante 1 del componente Mux A.

En este caso, INC A se implementa con la operación ADD A,Lit, con Lit = 1. Esta transformación queda a cargo del *Assembler*. Se podría implementar agregando una señal constante 1 en el componente Mux B, pero se opta por implementar por lo ya existente.

Estos cambios podrían habilitar más instrucciones, pero solo se listan las solicitadas en el enunciado.

Instrucción	Operandos	Literal	La	Lb	Sa	Sb	Sop	Sadd	W
ADD	B,(A)	-	0	1	B	DOUT	ADD	A	0
	A,(A)	-	1	0	A	DOUT	ADD	A	0
SUB	B,(A)	-	0	1	B	DOUT	SUB	A	0
	A,(A)	-	1	0	A	DOUT	SUB	A	0
AND	B,(A)	-	0	1	B	DOUT	AND	A	0
	A,(A)	-	1	0	A	DOUT	AND	A	0
OR	B,(A)	-	0	1	B	DOUT	OR	A	0
	A,(A)	-	1	0	A	DOUT	OR	A	0
XOR	B,(A)	-	0	1	B	DOUT	XOR	A	0
	A,(A)	-	1	0	A	DOUT	XOR	A	0
INC	A	1	1	0	A	LIT	ADD	-	0
	(A)	-	0	0	ONE	DOUT	ADD	A	1



## Ejercicios - Respuesta

Escriba un programa en Assembly del computador básico que calcule el inverso aditivo de un número entero de 8 bits.

```
DATA:  
    n 5  
    n_inv 0  
CODE:  
    MOV A,(n)      ;A = n  
    NOT A,A        ;A = -n  
    ADD A,1        ;A = -n + 1  
    MOV (n_inv),A ;n_inv = A = -n + 1
```

\* El inverso aditivo de un número en base binaria corresponde al complemento de 2:  $C_2(x) = \bar{x} + 1$