DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN
ESCUELA DE INGENIERÍA
PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE

IIC2343 - Arquitectura de Computadores

Programabilidad

1. Motivación

Una máquina capaz de realizar operaciones, almacenar datos e interactuar con el usuario, todavía no puede ser llamada «computador». La característica adicional que dicha máquina debe tener es la capacidad de ser programable. La programabilidad de un máquina permitirá que a partir de operaciones básicas se puedan escribir «programas» avanzados, y ejecutarlos de manera automática.

2. Acumulación de operaciones

El diagrama de la figura 1 muestra una calculadora simple de 4 bits que permite realizar una de las operaciones de la ALU (definidas en la tabla 1) según las interacciones del usuario.

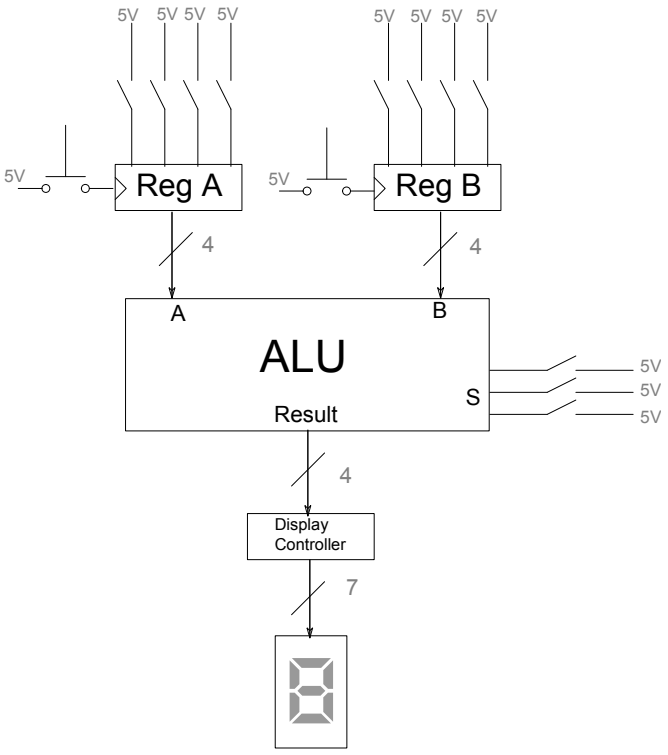

Figura 1: Calculadora de 4 bits.

select	s2	s1	s0	operación
0	0	0	0	Suma
1	0	0	1	Resta
2	0	1	0	And
3	0	1	1	Or
4	1	0	0	Not A
5	1	0	1	Xor
6	1	1	0	Shift Left A
7	1	1	1	Shift Right A

Tabla 1: Operaciones de la ALU.

El usuario de esta máquina debe:

- Ingresar los datos mediante interruptores
- Seleccionar la operación de la ALU mediante interruptores
- Almacenar los valores ingresados en los interruptores en los registros, mediante botones

Para realizar la operación $6 - 4$, el usuario deberá:

- Ingresar el número 6 en los interruptores del registro A y el número 4 en los interruptores del registro B
- Seleccionar la operación 001 de la ALU mediante los interruptores
- Presionar los botones de control de los registros

Una vez realizado el proceso, obtendrá el resultado (2) en el display. Si quisiera realizar una operación que ocupe como operando el resultado obtenido (por ejemplo $6-4+2$) el usuario debe colocar el resultado obtenido en los interruptores de manera manual. Para eliminar al usuario del loop de ingreso de datos, necesitamos acumular los valores resultantes en los registros. Esto se logra conectando la salida de la ALU a las entradas de carga de los registros, convirtiéndolos en **registros acumuladores** como se observa en la figura 2. Dado que no desplegaremos el resultado en el display (que está limitado a números de 4 bits), extendemos nuestro computador a 8 bits, aumentando el tamaño de los registros y la ALU.

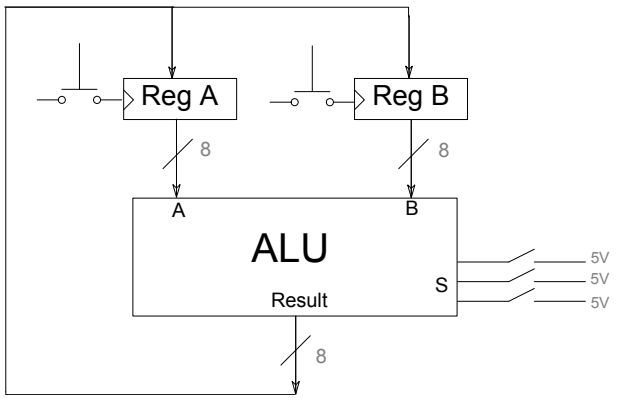

Figura 2: Las modificaciones realizadas permiten aumentar la cantidad de operaciones posibles.

Se observa que para lograrlo, se eliminaron los interruptores de carga de datos y el display. Veremos más adelante que este es un trade-off necesario, ya que eventualmente podremos conectar interruptores y display a la máquina y mantener la capacidad de acumulación. Una capacidad importante que se pierde es la de de cargar inicialmente los valores de los registros, lo que será resuelto más adelante. Por ahora, la acumulación del resultado es suficientemente relevante como para eliminar esas capacidades.

Dada la capacidad de acumulación agregada, es posible ahora realizar operaciones del tipo $A = A + B$ o $B = A - B$. El problema es que, como ambos registros están acumulando, cada vez que realicemos una operación **ambos registros quedan con el resultado**. Para solucionarlo, agregamos la capacidad de controlar la carga de los registros mediante una señal de control **Load** en ambos, como se ve en la figura 3.

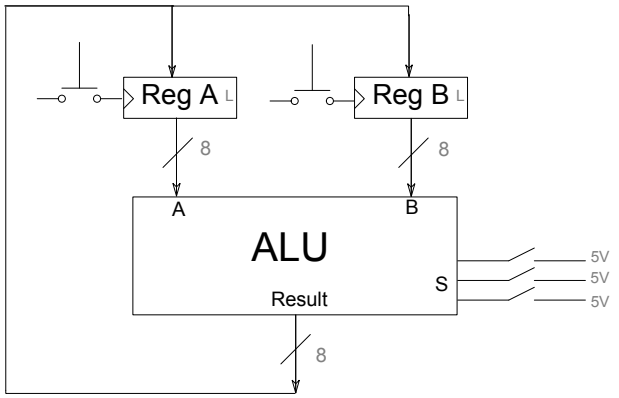

Figura 3: Se agregan señales de carga a los registros.

De esta forma, para realizar la operación $A = A + B$ debemos seleccionar la operación suma en la ALU ($select = 000$), indicar que queremos cargar el registro A ($loadA = 1$) y que no queremos cargar el registro B ($loadB = 1$). Combinando estos tres indicadores de control podemos obtener las distintas operaciones posibles por la máquina, las que se observan en la tabla 2.

la	lb	s2	s1	s0	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 and B
0	1	0	1	0	B=A and B
1	0	0	1	1	A=A or B
0	1	0	1	1	B=A or B
1	0	1	0	0	A=not A
0	1	1	0	0	B=not A
1	0	1	0	1	A=A xor B
0	1	1	0	1	B=A xor B
1	0	1	1	0	A=shift left A
0	1	1	1	0	B=shift left A
1	0	1	1	1	A=shift right A
0	1	1	1	1	B=shift right A

Tabla 2: Señales de control y su operación asociada.

3. Instrucciones

Las secuencias de señales de control descritas en la tabla 2 se conocen como las **instrucciones** de la máquina. A partir de un conjunto de instrucciones, podemos construir un **programa**. Por ejemplo, supongamos que inicialmente $A = 0$ y $B = 1$ podemos hacer un programa que genere la secuencia de los primeros 8 números de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13

la	lb	s2	s1	s0	operación	A	B
0	0	-	-	-	-	**0**	1
1	0	0	0	0	A=A+B	**1**	1
0	1	0	0	0	B=A+B	1	**2**
1	0	0	0	0	A=A+B	**3**	2
0	1	0	0	0	B=A+B	3	**5**
1	0	0	0	0	A=A+B	**8**	5
0	1	0	0	0	B=A+B	8	**13**

Con la máquina que llevamos construida necesitamos ir ingresando cada instrucción una por una ocupando los interruptores para obtener el resultado de cada operación e ir ejecutando el programa. Para automatizar este proceso, debemos dar un salto conceptual importante: **almacenar las señales de control como si fueran datos**. Una vez almacenadas las instrucciones, podremos automatizar su ejecución secuencial y por tanto, automatizar el funcionamiento de la máquina y convertirla en una máquina programable y autónoma.

3.1. Almacenamiento de instrucciones

Para almacenar las instrucciones necesitamos un componente que permita contener una serie de valores independientes y que permita acceder a ellos. Una **memoria** cumple con esto: podemos almacenar cada

instrucción como una **palabra** en memoria, y acceder a estas mediante **direcciones** de memoria. Luego, la salida de la memoria puede ser conectada a las señales de control que necesitamos, como se observa en la figura 4. Esta memoria se conoce como **memoria de instrucciones**, y se utilizará una memoria de tipo ROM (read only), dado que, por ahora, nos basta con carga las instrucciones del programa una vez.

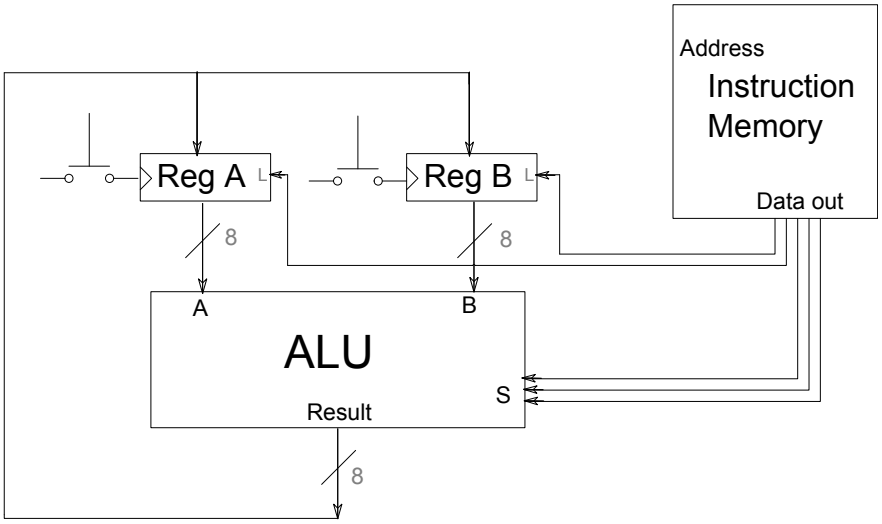

Figura 4: Se agrega una memoria ROM para almacenar instrucciones.

Para almacenar las instrucciones antes vistas, necesitamos palabras de 5 bits. La cantidad de palabras de la memoria limitará la cantidad de instrucciones que podemos hacer. En principio, la limitaremos a 16 palabras, lo que es suficiente para el programa antes visto. Dadas estas características, si almacenamos nuestro programa en la memoria ROM, los contenidos de esta serían:

dirección	instrucción
0000	10000
0001	01000
0010	10000
0011	01000
0100	10000
0101	01000

3.2. Direccionamiento de instrucciones

La memoria de almacenamiento para las instrucciones nos permite almacenar las instrucciones de manera ordenada. Sin embargo, todavía no tenemos una máquina completamente automática: de alguna forma tenemos que indicarle a la memoria que instrucción ejecutar. Para solucionar esto podemos aprovechar el hecho de que el avance de las instrucciones es secuencial: primero se comienza con la instrucción 0, ubicada en la dirección 0000, luego la instrucción 1 en la dirección 0001, etc. Podemos agregar entonces un **contador** que vaya aumentando de a 1 y con su valor direccionando la instrucción que corresponde a ejecutar, lo que se observa en la figura 5. Este contador se denomina **program counter** ya que es usado para indicar en que parte del programa estamos ubicados.

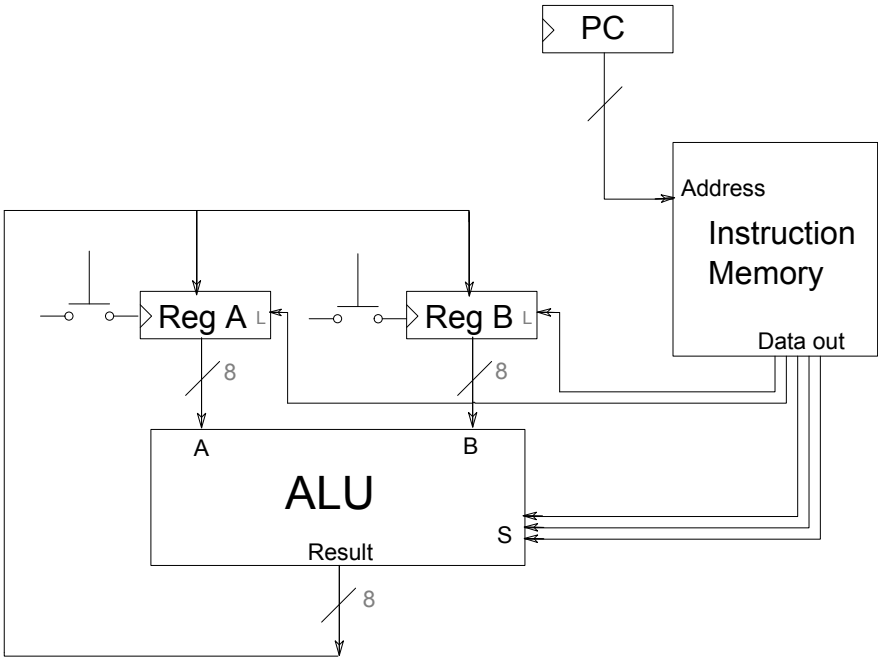

Figura 5: Se agrega un contador para direccionar secuencialmente la memoria de instrucciones: program counter.

El valor del program counter define la dirección de la memoria de instrucciones, y por tanto la operación a realizar. La siguiente tabla resume la relación entre el valor del program counter y la operación que se está realizando:

program counter	instrucción	operación
0000	10000	A=A+B
0001	01000	B=A+B
0010	10000	A=A+B
0011	01000	B=A+B
0100	10000	A=A+B
0101	01000	B=A+B

4. Automatización y sincronización

El último paso para automatizar completamente la máquina es automatizar la señales de control de los registros y la señal de incremento del program counter. Para realizar esto se ocupan los componentes denominados **clocks**.

4.1. Clocks

Los clocks son osciladores que generan alternadamente pulso de voltaje alto (equivalentes a un 1 lógico) y pulsos de voltaje bajo (equivalentes a un 0 lógico) a una **frecuencia constante**. El tipo de clock más usado corresponde a uno basado en cristales de cuarzo. Estos cristales tienen la propiedad de que al moverse, generan electricidad (los materiales con está propiedad se denominan piezoeléctricos). El movimiento que se les induce a los cristales es tal, que estos entran en resonancia, y por tanto vibran a

una frecuencia constante. Es esta vibración la que se convierte en el pulso eléctrico. Como todo sistema resonante, el cristal de cuarzo pierde energía en el tiempo, y comienza a dejar de vibrar. Para compensar esto, el cristal está continuamente alimentado con corriente eléctrica, dado que el cuarzo también tiene la propiedad de que al recibir corriente eléctrica vibrará.

Para mantener sincronizadas las operaciones se utilizará un **único clock** que se conecta a todos los componentes que lo requieren. En el caso de la máquina que estamos construyendo, se le agrega la señal de clock a los registros y al program counter, como se ve en la figura 6.

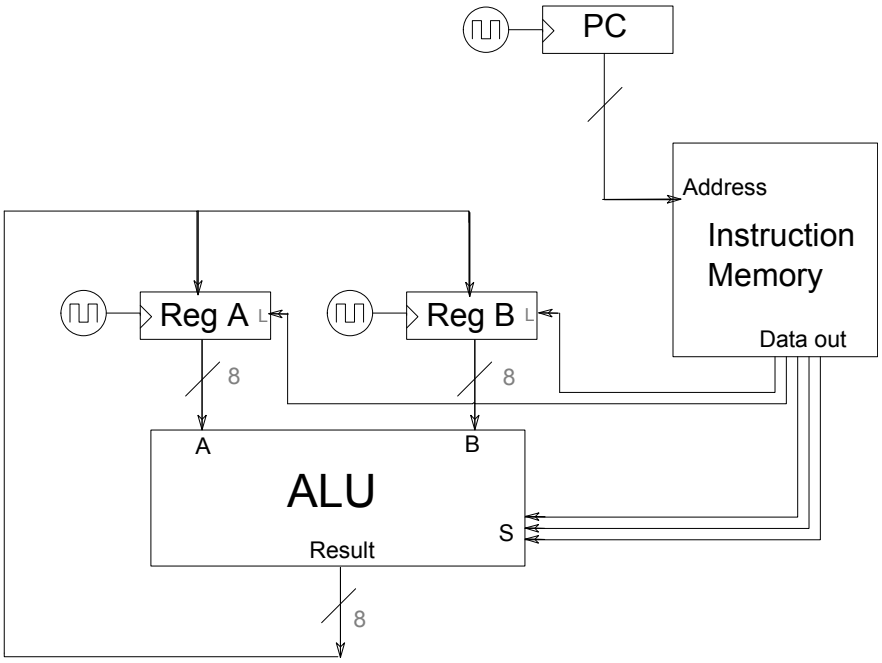

Figura 6: Se agrega un clock para sincronizar y automatizar el funcionamiento del computador.

4.2. Velocidad del clock

La velocidad del clock idealmente nos gustaría que fuera la más rápida posible: mientras más rápido es el clock, más operaciones se pueden hacer por segundo, y por tanto más rápido es el computador. Se podría pensar que la única limitación entonces será que tan rápido pueden oscilar los cristales de cuarzo, y que por tanto esa es la limitante tecnológica para tener computadores más rápidos. En la práctica esto no es así. La limitación real para la velocidad está en lo que se conoce como **retraso de propagación** que corresponde a cuanto se demora una señal eléctrica en completar un circuito. Este retraso se debe a dos factores: el retraso al pasar por una compuerta binaria o **gate delay** y el retraso por moverse a través de los cables o **wire delay**.

Veamos un ejemplo con el circuito del full-adder, que se observa junto al half-adder en la figura 7. Si cada compuerta tiene un retraso de t_{gate} segundos, podemos observar que entre las entradas y las salidas hay un máximo de 3 compuertas seguidas, y por tanto el gate delay del full-adder es $3 \times t_{gate}$. Si consideramos además un wire delay de t_{wire} segundos, tenemos que el retraso de propagación total del componente es $3 \times t_{gate} + t_{wire}$ y por tanto con este circuito podemos hacer 1 operación cada $3 \times t_{gate} + t_{wire}$.

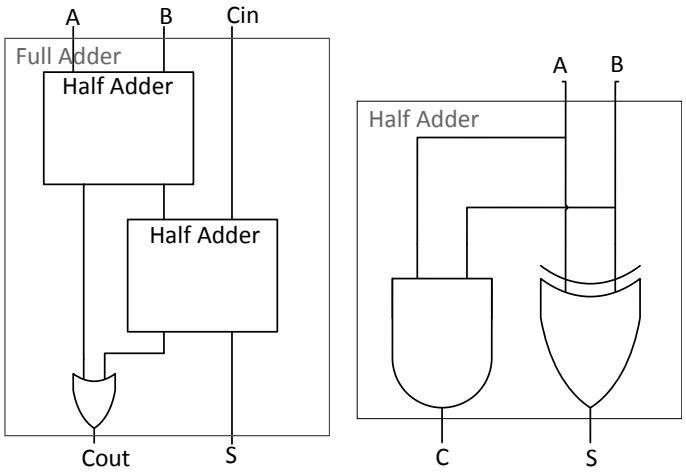

Figura 7: Circuitos de un full-adder y half-adder.

4.3. Funcionamiento del computador con clock

El procesamiento y ejecución de una instrucción para el computador básico visto hasta ahora consta de tres etapas. Primero, el valor del program counter se debe obtener para direccionar la memoria de instrucciones, y a la vez incrementar su valor para dejarlo listo para el procesamiento de la próxima instrucción. Segundo, la instrucción debe ser obtenida desde la memoria, decodificada en la unidad de control, y las señales de control enviadas a los distintos componentes para indicarles que hacer. Tercero, los registros deben ser habilitados para cargar sus nuevos valores, si corresponde.

Un enfoque simple para automatizar el computador es conectar el clock a los distintos circuitos secuenciales (registros y contadores). Un clock se puede pensar como una secuencia de 0s y 1s, valores que se van alternando a una frecuencia fija. Estos valores pueden ser ocupados para alimentar la señal de control de un circuito como un flip-flop, el componente base de los registros. De esta manera en estos registros, mientras la señal del clock este en 1, se estará permitiendo que pase el valor de entrada y se almacenado, y cuando la señal baja a cero, ya no pasan valores.

En nuestro computador básico, si ocupamos este tipo de registros aparecen dos problemas. Primero, el program counter va a incrementarse más de lo que corresponde mientras el clock esté en 1. Lo que necesitamos es que se incremente una sola vez. Segundo, los registros van recibir lo que tienen en su entrada mientras el clock esté en 1, pero como están retroalimentados, van a estar modificando sus valores de manera continua durante el estado alto del clock.

Para entender como solucionar este problema, es necesario agregar el concepto de **circuito activado por flancos**. En este tipo de circuitos, a diferencia de los registros simples explicados anteriormente, sólo se considera la señal de control en los momentos en que hay un cambio entre 0 y 1 (flanco de subida) o entre 1 y 0 (flanco de bajada). De esta manera, estos circuitos aseguran que para un ciclo de clock, la señal de control se activará sólo una vez, evitando los problemas anteriores.

Entonces, para lograr que las tres etapas del ciclo de la instrucción se ejecuten en un sólo clock, necesitamos ocupar registros y program counter activados por flancos. Como queremos que el program counter se actualice primero, esté será activado por flanco de subida (figura 8). Luego en el estado clock=1, el program counter no se actualiza, los registros tampoco, lo que da tiempo para procesar la instrucción y llevar las señales de control correspondientes para ejecutar las operaciones (figura 9). Finalmente, en el siguiente flanco de subida, los registros se habilitan para almacenar los resultados de las operaciones si corresponden (figura 10) y el program counter se actualiza al siguiente valor, reiniciando el proceso.

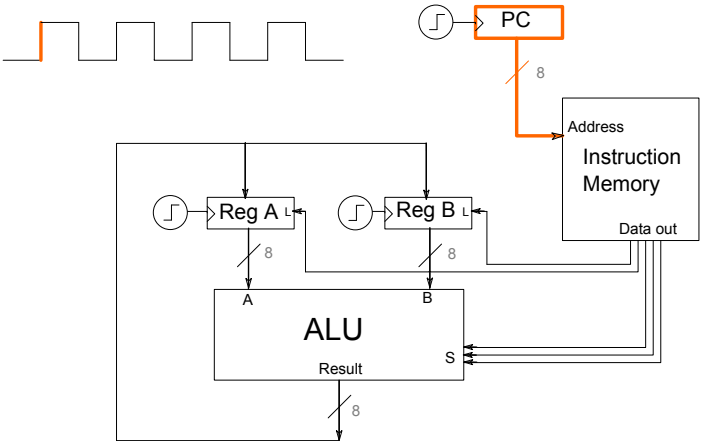

Figura 8: Program counter activado por flanco de subida.

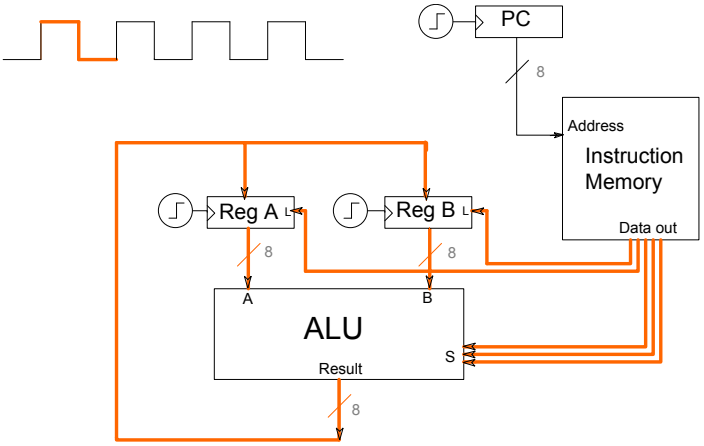

Figura 9: Cuando el clock subió y luego cuando baja, el program counter y los registros no se actualizan, dejando tiempo para procesar la instrucción y ejecutarla.

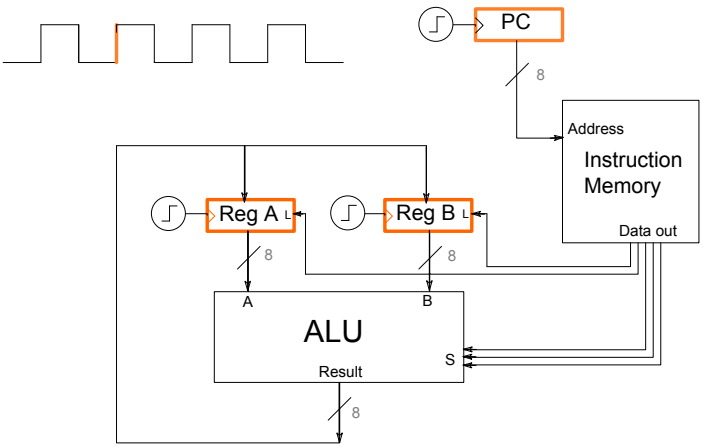

Figura 10: Los registros son activados en flanco de subida, guardando los resultados si correspondían.

5. Extendiendo el computador básico

5.1. Operaciones con literales

Un primer elemento que se debe agregar a la máquina programable es la capacidad de operar con **literales**. Un literal se refiere a un valor numérico que se define explícitamente. Por ejemplo la instrucción $A = A + 5$ involucra la suma del registro A con un literal, en este caso 5. La instrucción $A = A + B$ en cambio no tiene literales en sus operandos.

Dado que el valor del literal que se incluirá en una instrucción, por ejemplo $A = A + Literal$ es variable y debe ser entregado de forma explícita, se debe incluir el valor como **parte de la instrucción**. De esta manera las instrucciones quedarán compuestas ahora por dos partes: las señales de control que indican la operación y los **parámetros** asociados a esta operación. Para incluir los parámetros en la instrucción debemos extender el tamaño de las palabras de la memoria de instrucciones, idealmente agregando n bits, donde n es el tamaño de los registros de operación.

Además de extender la memoria de instrucciones, para realizar una operación del tipo $A = A + Literal$ es necesario permitir seleccionar el segundo operando de la operación, para lo cual se agrega un multiplexor, como se observa en la figura 11, y por tanto una nueva señales de control. Para permitir una mayor capacidad en las operaciones y en los valores de los literales, los registros y la ALU ahora serán de 8 bits, y por tanto la memoria ROM será extendida para contener 8 bits más correspondiente al literal, y 1 bit más correspondiente a la señal de control de selección del multiplexor.

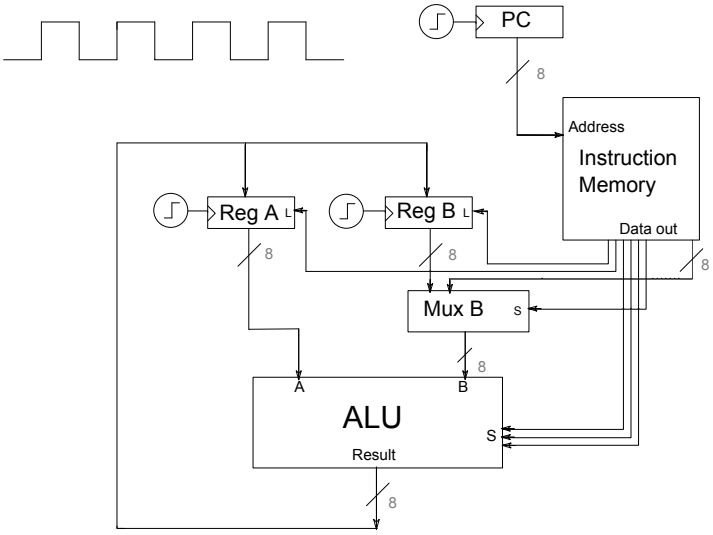

Figura 11: Computador con operaciones con literales.

Con estas modificaciones podemos realizar todas las operaciones de la ALU tanto entre los registros A y B como entre el registro A y el literal que venga de parámetro. Sin embargo, una capacidad importante que falta es poder cargar literales directamente en los registros. Para lograr esto aprovecharemos el «truco» aritmético de sumarle cero a un valor para no modificarlo. De esta forma cargar el registros A corresponderá a la operación $A = 0 + Lit$ y cargar el registro B corresponderá a $B = 0 + Lit$. Adicionalmente podemos ocupar este mismo truco para hacer transferencia entre registros: $B = A + 0$ y $A = 0 + B$.

Para permitir realizar estas sumas con 0 debemos agregar un nuevo multiplexor, esta vez al registro A, que permita elegir entre el resultado del registro y el valor 0. Adicionalmente, al multiplexor del registro B le agregamos una entrada que permite también elegir el valor 0, como se ve en la figura 12.

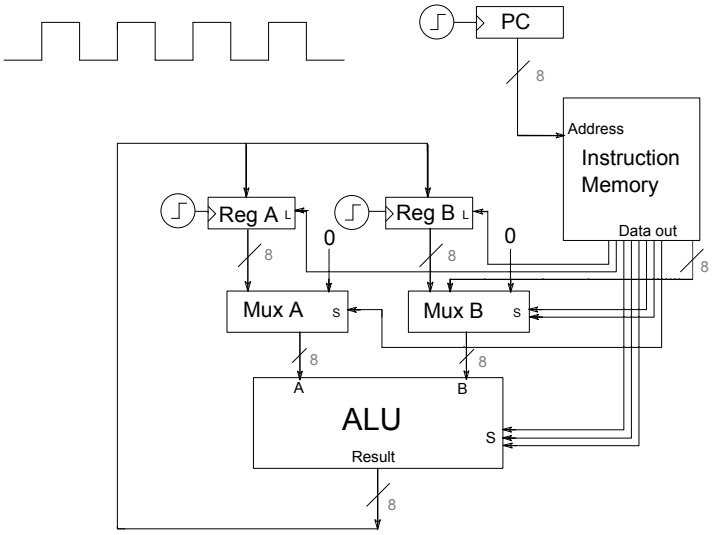

Figura 12: Computador con carga de valores de registros mediante literales.

6. Unidad de control

Al agregar soporte para operaciones con literales, carga de literales en registros y carga de valores entre registros, hemos incrementado la señales de control a 8, y por tanto incluyendo el literal también de 8 bits, las palabras de la memoria de instrucción quedan de 16 bits. Considerando que aún quedan muchas operaciones por incorporar al computador, se observa que el tamaño de las palabras de la memoria, seguirá creciendo. Sin embargo, podemos observar que aunque se tienen 8 bits de control, no hay $2^8 = 256$ operaciones distintas en el computador, como se observa en la tabla 3. Por ejemplo todas las combinaciones de $LoadA = 0$ y $LoadB = 0$ no se ocupan.

La	Lb	Sa0	Sb0	Sb1	Sop2	Sop1	Sop0	Operación
1	0	1	0	0	0	0	0	A=B
0	1	0	1	1	0	0	0	B=A
1	0	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 and B
0	1	0	0	0	0	1	0	B=A and B
1	0	0	0	1	0	1	0	A=A and Lit
1	0	0	0	0	0	1	1	A=A or B
0	1	0	0	0	0	1	1	B=A or B
1	0	0	0	1	0	1	1	A=A or Lit
1	0	0	0	0	1	0	0	A=not A
0	1	0	0	0	1	0	0	B=not A
1	0	0	0	0	1	0	1	A=A xor B
0	1	0	0	0	1	0	1	B=A xor B
1	0	0	0	1	1	0	1	A=A xor Lit
1	0	0	0	0	1	1	0	A=shift left A
0	1	0	0	0	1	1	0	B=shift left A
1	0	0	0	0	1	1	1	A=shift right A
0	1	0	0	0	1	1	1	B=shift right A

Tabla 3: Señales de control y su operación asociada.

Para aprovechar esta situación se puede agregar una **unidad de control**. La unidad de control por ahora será simplemente una segunda memoria ROM, que tendrá palabras de 8 bits, las cuales contendrán las señales de control, pero sus direcciones serán de sólo 6 bits. De esta forma, la memoria de instrucción no almacenará directamente las señales de control, sino un código de operación u **opcode** el cual se utilizará para direccionar la segunda memoria ROM y está se encargará de indicar directamente las señales de control, lo que se observa en la tabla 4.

Opcode	La	Lb	Sa0	Sb0	Sb1	Sop2	Sop1	Sop0	Operación
000000	1	0	1	0	0	0	0	0	A=B
000001	0	1	0	1	1	0	0	0	B=A
000010	1	0	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 and B
001011	0	1	0	0	0	0	1	0	B=A and B
001100	1	0	0	0	1	0	1	0	A=A and Lit
001101	1	0	0	0	0	0	1	1	A=A or B
001110	0	1	0	0	0	0	1	1	B=A or B
001111	1	0	0	0	1	0	1	1	A=A or Lit
010000	1	0	0	0	0	1	0	0	A=not A
010001	0	1	0	0	0	1	0	0	B=not A
010010	1	0	0	0	0	1	0	1	A=A xor B
010011	0	1	0	0	0	1	0	1	B=A xor B
010100	1	0	0	0	1	1	0	1	A=A xor Lit
010101	1	0	0	0	0	1	1	0	A=shift left A
010110	0	1	0	0	0	1	1	0	B=shift left A
010111	1	0	0	0	0	1	1	1	A=shift right A
011000	0	1	0	0	0	1	1	1	B=shift right A

Tabla 4: Opcode y señales de control.

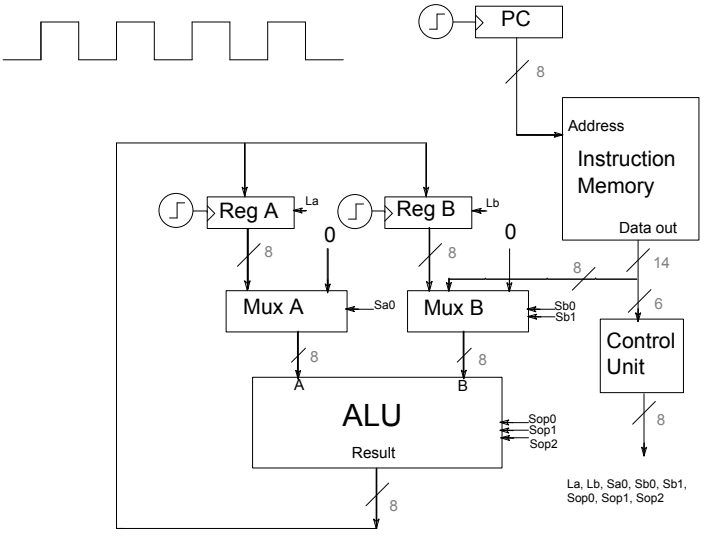

Figura 13: Computador con unidad de control.

7. Assembly y set de instrucciones

El uso de los opcodes descritos en la tabla 4 permite abstraerse de las señales de control del computador y trabajar con identificadores que sean independientes de la implementación física del computador. Este primer nivel de abstracción facilita la construcción de las instrucciones de un programa, pero el hecho de seguir trabajando con representación binaria todavía agrega un grado dificultad para un programador humano. Para facilitar el trabajo de la persona que programa un computador, se define un lenguaje de programación denominado **assembly** el cual permite asignarle nombre a las instrucciones y poder escribir programas ocupando estos nombres en vez de las representaciones binarias.

La forma más simple de escribir un assembly para el computador antes visto es asignar un nombre distinto para cada opcode, como se observa en la tabla 5. Con este assembly, cada opcode equivale a una palabra única que indica de manera abreviada que función realiza. Por ejemplo la instrucción $MOVAB$ indica que se está moviendo el valor del registro B en el registro A. La instrucción $MOVAL\ Lit$ indica que se está moviendo el valor del literal Lit en el registro A.

Instrucción	Opcode	La	Lb	Sa0	Sb0	Sb1	Sop2	Sop1	Sop0	Operación
MOVAB	000000	1	0	1	0	0	0	0	0	A=B
MOVBA	000001	0	1	0	1	1	0	0	0	B=A
MOVAL	000010	1	0	1	0	1	0	0	0	A=Lit
MOVBL	000011	0	1	1	0	1	0	0	0	B=Lit
ADDA	000100	1	0	0	0	0	0	0	0	A=A+B
ADDB	000101	0	1	0	0	0	0	0	0	B=A+B
ADDL	000110	1	0	0	0	1	0	0	0	A=A+Lit
SUBA	000111	1	0	0	0	0	0	0	1	A=A-B
SUBB	001000	0	1	0	0	0	0	0	1	B=A-B
SUBL	001001	1	0	0	0	1	0	0	1	A=A-Lit
ANDA	001010	1	0	0	0	0	0	1	0	A=A and B
ANDB	001011	0	1	0	0	0	0	1	0	B=A and B
ANDL	001100	1	0	0	0	1	0	1	0	A=A and Lit
ORA	001101	1	0	0	0	0	0	1	1	A=A or B
ORB	001110	0	1	0	0	0	0	1	1	B=A or B
ORL	001111	1	0	0	0	1	0	1	1	A=A or Lit
NOTA	010000	1	0	0	0	0	1	0	0	A=not A
NOTB	010001	0	1	0	0	0	1	0	0	B=not A
XORA	010010	1	0	0	0	0	1	0	1	A=A xor B
XORB	010011	0	1	0	0	0	1	0	1	B=A xor B
XORL	010100	1	0	0	0	1	1	0	1	A=A xor Lit
SHLA	010101	1	0	0	0	0	1	1	0	A=shift left A
SHLB	010110	0	1	0	0	0	1	1	0	B=shift left A
SHRA	010111	1	0	0	0	0	1	1	1	A=shift right A
SHRB	011000	0	1	0	0	0	1	1	1	B=shift right A

Tabla 5: Instrucción simple, opcode y señales de control.

La conversión entre las palabras del assembly y los opcodes la realiza el programa denominado **assembler**. Este programa se encargará de convertir entonces un programa escrito en lenguaje entendible por un ser humano en el lenguaje de la máquina. Un **compilador** será el programa encargado de convertir

un lenguaje de alto nivel en el assembly de la máquina, permitiendo un nivel aún más alto de abstracción. La diferencia principal entre estos dos programas es que el assembler realiza una conversión simple, prácticamente haciendo sólo reemplazos entre palabras y códigos; el compilador requiere mayor complejidad dado que los lenguajes de alto nivel son más complejos.

Podemos reescribir las instrucciones del assembly antes visto de una forma que sea más legible, separando la instrucción de los operandos, como se observa en la tabla 6. De esta forma la instrucción MOV representará diversos opcode, dependiendo de sus operandos. Por ejemplo $MOV\,A, B$ indica la operación $A = B$ con opcode 000000, y la operación $MOV\,A, Lit$ indica la operación $A = Lit$ con opcode 000010.

Instrucción	Operandos	Opcode	La	Lb	Sa0	Sb0	Sb1	Sop2	Sop1	Sop0	Operación
MOV	A,B	000000	1	0	1	0	0	0	0	0	A=B
	B,A	000001	0	1	0	1	1	0	0	0	B=A
	A,Lit	000010	1	0	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=shift left A
	B,A	010110	0	1	0	0	0	1	1	0	B=shift left A
SHR	A,A	010111	1	0	0	0	0	1	1	1	A=shift right A
	B,A	011000	0	1	0	0	0	1	1	1	B=shift right A

Tabla 6: Instrucción con operandos, opcode y señales de control.

8. Memoria de datos

Los registros de próposito general de un computador representan la forma de almacenamiento más simple y con acceso más directo a la unidad de ejecución. Sin embargo, aunque se puede extender el número de registros, es imposible escalar lo suficiente para permitir manejar cantidades considerables de información en un programa. Debido a esto es necesario agregar un componente especial al computador, que permita almacenar y agregar datos que puedan ser **variables** durante el transcurso del programa. El componente que se utiliza es la **memoria de datos**, la cual corresponde a una memoria RAM de lectura

y escritura.

Al igual que la memoria de instrucciones, la memoria de datos se compone de una secuencia de palabras las cuales pueden ser accedidas mediante direcciones espécificas asociadas a cada una. La diferencia está en que la memoria de datos debe permitir modificar estas palabras, a diferencia de la de instrucciones, donde la información de las instrucciones no cambia durante el transcurso del programa. Para esto, la memoria cuenta con una entrada de datos y una señal de control que indica si la memoria está en modo escirtura o lectura.

Para integrar la memoria de datos al computador básico se requieren tres conexiones de datos: una conexión con la **entrada de datos**, otra con la **salida de datos** y otra con la **dirección de los datos**. Adicionalmente se requiere agregar una nueva señal de control **W** a la unidad de control, que cuando tome el valor 1 indique que la memoria está en modo escritura (write), y cuando está en 0, en modo lectura. La entrada de datos, al igual que para el caso de los registros será obtenida de la salida de la ALU; la salida de datos se conectará al multiplexor B, para poder ser ocupada como operando en las operaciones de la ALU.

La figura 14 muestra el diagrama con la memoria de datos agregada y con estas conexiones realizadas. Se observa que la memoria de datos tiene palabras de **8 bits** con lo cual se pueden ocupar directamente para operar con los registros y la ALU.

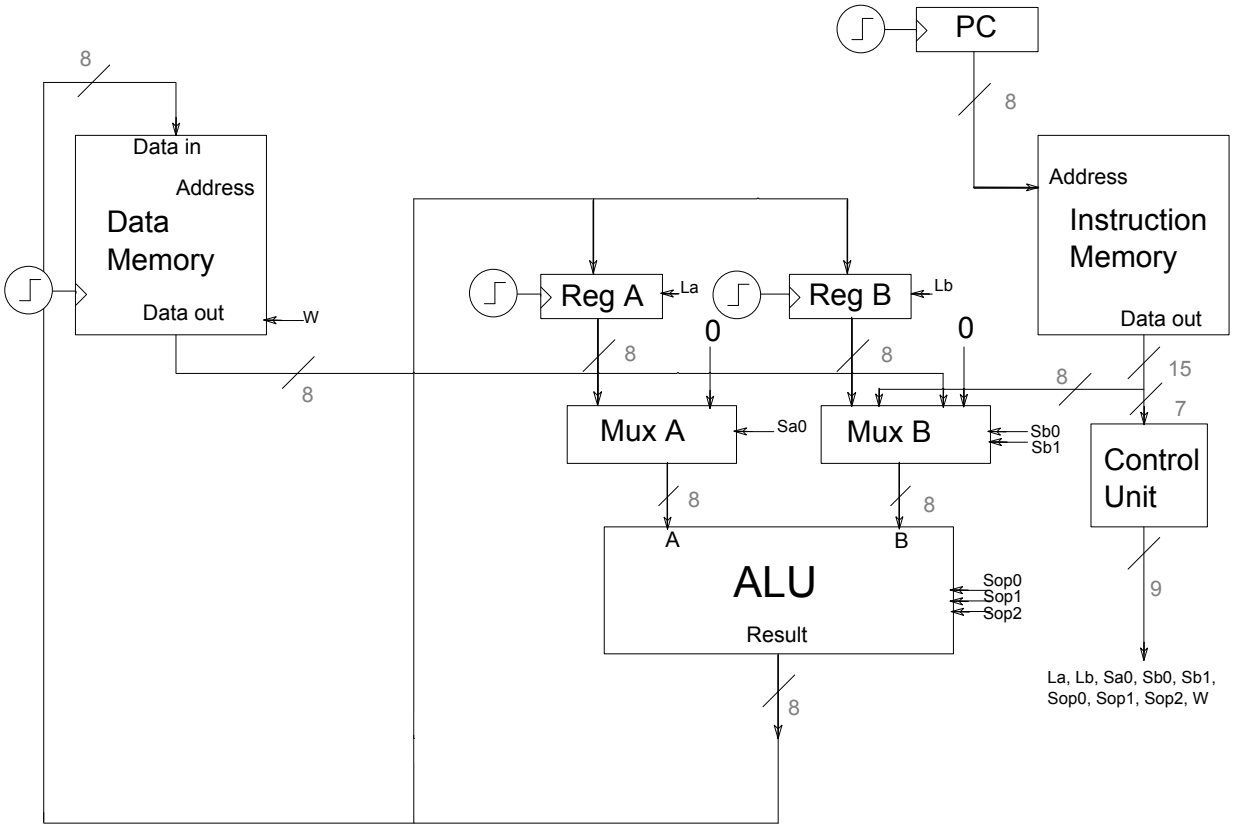

Figura 14: Memoria de datos agregada al computador básico

La conexión que falta es la dirección para la memoria. Existen diversas formas en que se puede realizar el direccionamiento de la memoria, las que se conocen como los **modos de direccionamiento** de un computador.

Al computador básico, le agregaremos dos modos de direccionamiento a la memoria. Un primer modo

será el **direccionamiento directo** en el cual la dirección de memoria a leer o escribir viene como parámetro de la instrucción (literal). Un segundo modo será el direccionamiento indirecto por registro, en el cual la dirección de memoria a leer o escribir será obtenida desde un registro, en este caso ocupando el registro B como registro de dirección. La figura 15 muestra el diagrama de las conexiones necesarias para realizar ambos modos de direccionamiento.

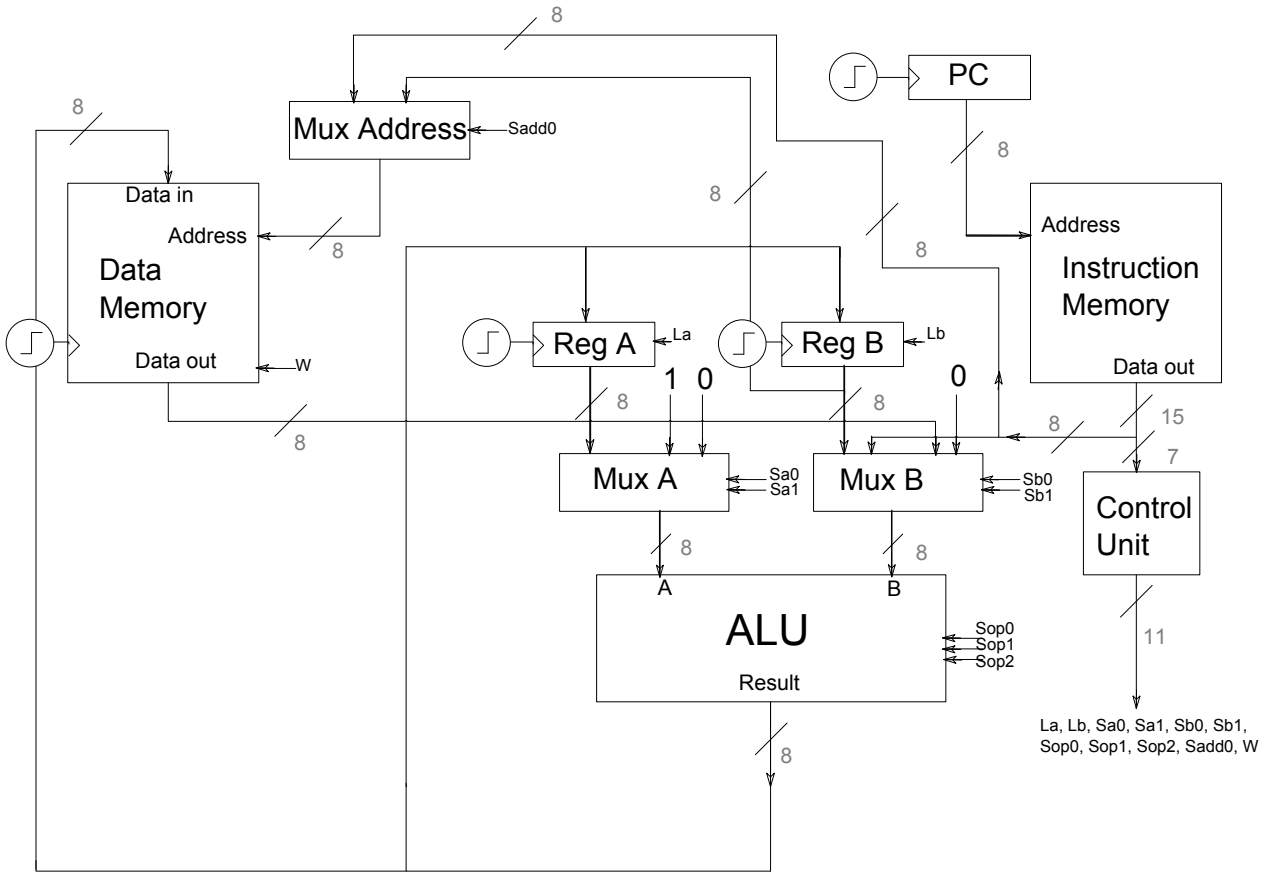

Figura 15: Modos de direccionamiento del computador básico

8.1. Manejo de memoria de datos y modos de direccionamiento en assembly

Para poder trabajar con datos de memoria en programas escritos en assembly, debemos definir primero como explicitar los valores que están almacenados en la memoria de datos. Una forma de hacer esto es dividir el programa assembly en dos segmentos, uno de **datos** y uno de **código**. En nuestro caso dividiremos ambos segmentos usando la plabra clave `DATA:` para indicar el comienzo de la sección de datos, y `CODE:` para indicar el comienzo de la sección de código. Adicionalmente, podemos ocupar label para referirnos a direcciones específicas de la memoria de datos. Estos labels se interpretarán como el **nombre de la variable** asociada al dato guardado en esa dirección de memoria.

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

El segundo elemento necesario agregar al assembly son las instrucciones específicas para utilizar el direccionamiento directo e indirecto por registro. Para esto, se utilizará la nomenclatura (*direccion*) para indicar el que se quiere acceder al dato indicado por la dirección. En el caso del direccionamiento directo, se utilizará «(label)» para acceder al dato en la dirección de memoria asociada a ese label; en el caso del direccionamiento indirecto por registro, se utilizará «(B)» para acceder al dato en la dirección de memoria asociada al valor del registro B.

A continuación se muestran todas las instrucciones de direccionamiento del assembly del computador básico. Se observa que se agregó la instrucción `INC B` para facilitar el uso del registro B como registro de direccionamiento, y permitir ir recorriendo secuencias de valores en memoria.

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

Tabla 7: Instrucciones de direccionamiento

9. Ejercicios

1. Escriba las instrucciones en lenguaje de la máquina de la figura 5 para ejecutar la operación $(x << 3) + (x << 1)$. ¿Que relación tiene x con el resultado de la operación? Asuma que puede cargar inicialmente los registros A y B con los valores que estime conveniente.

2. Investigue las distintas tecnologías usadas para generar componentes osciladores. ¿Que ventajas tienen los cristales de cuarzo que los hacen ideales para los computadores?

3. Escriba ocupando el assembly de la tabla 6 un programa que realice las siguientes operaciones:

   - Cargar los valores 5 y 6 en los registros A y B respectivamente
   - Sumar los valores de los registros A y B y guardarlos en A.
   - Intercambiar los valores de A y B
   - Restar los valores de A y B
   - Setear en 0 el valor de B

4. Implemente usando el assembly de la tabla 4 un algoritmo que multiplique el valor del registro A por 10 y lo almacenen en el registro B.

5. ¿Que operaciones que son soportadas por la máquina no tienen representación en el assembly descrito en la tabla 4?

10. Referencias e información adicional

- Hennessy, J.; Patterson, D.: Computer Organization and Design: The Hardware/Software Interface, 4 Ed., Morgan-Kaufmann, 2008. Chapter 4: The processor.