Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2343 Arquitectura de Computadores

Representación de números

1. Motivación

Todo tipo de información puede ser representada mediante números. Una palabra, por ejemplo, puede ser representada mediante números reemplazando cada letra con un número, ocupando algún código que asocie para cada letra un número específico (como el código ASCII). Un dibujo puede representarse como un conjunto de posiciones (números) y el valor de color del dibujo en esa posición (también números). En definitiva, podemos reducir el problema de representar información a representar números, y por esto es importante entender que representaciones numéricas existentes y como se utilizan.

2. Representaciones numéricas

La representación numérica mas simple corresponde a usar un símbolo por cada incremento en uno de un número. Por ejemplo, si ocupamos el símbolo «*» para representar el número cuatro, dibujamos cuatro símbolos: «****». El problema de esta representación es que no escala: para representar el número un millón, necesitamos dibujar un millón de veces el símbolo «*» lo que sería un costo tanto en espacio (por ejemplo papel si estuviésemos escribiendo el número en un cuaderno) y de tiempo (escribir un millón de veces el símbolo tomará al menos un millón de segundos, aproximadamente 11 días sin parar).

Para solucionar los problemas que presentaba esta representación simple, se inventaron representaciones más avanzadas, que permitían de alguna forma representar números grandes con pocos símbolos. Una de estas representaciones fueron los números romanos, que ocupaban símbolos específicos para acumulaciones de números: V para cinco, X para diez, L para cincuenta, etc. Otra representación ideada fueron los números indo-arábicos, que es la que ocupamos hoy en día, basada en diez símbolos: 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9. Esta representación se caracteriza por pertenecer a un grupo de representaciones denominadas «representaciones posicionales».

2.1. Representaciones posicionales

Las representaciones posicionales, como la indo-arábica, se basan en dos elementos para determinar el valor de un número: la posición de los símbolos en la secuencia de números y la cantidad de símbolos posible, lo que se denomina la base. En el caso de los números indo-árabes la base es diez (i.e. hay diez símbolos posibles), y por esto esta representación numérica también se denomina «representación decimal».

Para entender como afecta la posición en el valor numerico, es necesario realizar un ejemplo. Como estamos tan acostumbrados a asociar un número a su representación en el sistema decimal, para este ejemplo, vamos a reemplazar los símbolos asociados al número uno y dos por «?» y «&» de manera de poder abstraernos del número y fijarnos en la representación.

Supongamos se tiene la secuencia de numérica «??&». La fórmula para poder interpretar este número en el sistema decimal es la siguiente:

$$? \times diez^{dos} + ? \times diez^{uno} + \& \times diez^{cero} \tag{1}$$

Al resolver esta ecuación obtendremos el valor de la secuencia numérica, pero debemos primero elegir una representación en la cual queremos hacer el cálculo. Si utilizamos la representación decimal, y reemplazamos los símbolos «?» y «&» por sus valores 1 y 2 obtenemos:

$$1 \times 10^2 + 1 \times 10^1 + 2 \times 10^0 = 112 \tag{2}$$

A partir de esto podemos obtener la regla general para obtener el valor de una secuencia de símbolos en representación decimal:

$$\sum_{k=0}^{n-1} s_k \times 10^k \tag{3}$$

donde s_k corresponde al símbolo en la posición k, n corresponde al número de símbolos en la secuencia y k toma valores desde 0 hasta $n-1$, es decir las posiciones comienzan desde 0 comenzando con el símbolo de más a la derecha.

De la fórmula se puede inferir que podemos crear representaciones posicionales con otras bases, y bastaría reemplazar el número diez de la fórmula adecuada. De esta manera, la ecuación general para obtener el valor numerico de una determinada representación posicional es:

$$\sum_{k=0}^{n-1} s_k \times b^k \tag{4}$$

donde b es la base de la representación elegida, la cual podría ser cualquier valor numérico. De hecho, el uso de la base diez que consideramos habitual se debe solamente a que los seres humanos tenemos diez dedos, y por tanto atribuimos a ese número una cierta característica especial. Si tuviéramos ocho dedos, seguramente ocuparíamos una sistema posicional con base ocho. En definitiva, no existe un sistema que de por si sea mejor que otro, todo depende de las circunstancias y de su uso.

2.2. Representaciones binaria, octal y hexadecimal

Existen tres representaciones habitualmente usadas en el contexto de la computación: binaria, octal y hexadecimal, siendo la primera la de mayor importancia. Tal como sus nombres lo señalan la representación binaria tiene base dos, la octal base ocho y la hexadecimal base dieciséis. Vamos a comenzar analizando la base octal, para luego pasar a las otras dos.

Dado que un número octal tiene base ocho, necesitamos ocho símbolos distintos para su representación. Por comodidad utilizaremos los primeros ocho números indo-arábicos: 0, 1, 2, 3, 4, 5, 6 y 7. Es importante destacar que podríamos ocupar cualquier otro grupo de ocho símbolos, solo ocuparemos estos por conveniencia, dado que estamos acostumbrados a trabajar matemáticamente con éstos.

Tomemos como ejemplo de número octal el número 112. El primer elemento importante a destacar es que esta secuencia de números no representa el número «ciento doce» que habitualmente asociaríamos a esos símbolos si estuviese en representación decimal. Es decir: la secuencia de símbolos es la misma, pero al representación distinta. Para evitar confusiones, para todas las bases no decimales se usa especificar junto al símbolo la base. En nuestro caso entonces, el número sería $(112)_8$, es decir la secuencia 112 en representación octal.

Si utilizamos la ecuación (15) podemos obtener el valor numérico de esta secuencia octal. Es importante resaltar que al ocupar esta ecuación debemos decidir en que representación queremos el resultado. Lo habitual es que éste quede en representación decimal, pero podría quedar también en otra si lo quisiéramos. Más

adelante veremos ejemplos de este tipo, pero por ahora haremos el cálculo de manera de obtener la el valor en representación decimal:

$$1 \times 8^2 + 1 \times 8^1 + 2 \times 8^0 = 74 \tag{5}$$

Es decir, la secuencia 112 en octal, representa el número decimal 74.

Revisemos ahora la representación binaria, la cual tiene como base el número dos, y por tanto requiere dos símbolos. Ocuparemos, nuevamente por comodidad, los dos primeros números indo-arábicos 0, 1. Tomemos como ejemplo el número $(1011)_2$. Nuevamente es importante recordar que esta secuencia no tienen ninguna relación con el número «mil once» en representación decimal, solamente tienen la misma secuencia de símbolos. Podemos nuevamente aplicar la ecuación (15) para obtener el valor numerico en representación decimal:

$$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 1^0 = 11 \tag{6}$$

La ventaja de esta representación es que nos basta con dos símbolos para representar todos los posibles números. La desventaja es que necesitamos más posiciones para representar números. Esta desventaja la hace poco práctica para ser usada por humanos, pero la ventaja de tener sólo dos estados la hace ideal para realizar cálculos automáticos, por ejemplo en un computador.

Podemos también definir una representación cuya base sea mayor que diez. Una representación importante que cumple con esto es la representación hexadecimal, la cual tiene como base el número dieciséis. Más adelante veremos que la utilidad de esta representación está en su relación con la representación binaria, y lo fácil que es convertir de una a otra. Para esta representación necesitamos dieciséis símbolos, por lo que a diferencia de las otras representaciones estudiadas no nos basta con los números indo-arábicos, necesitamos seis símbolos más. Para no tener que inventar nuevos símbolos específicos de esta representación, habitualmente se ocupan los diez símbolos indo-arábicos para representar los diez primeros dígitos y las primeras seis letras del abecedario (A=diez, B=once, C=doce, D=trece, E=catorce, F=quince) para representar los seis restantes.

Por ejemplo, la secuencia $(A1F)_{16}$ puede ser interpretada como un número hexadecimal. Para obtener el valor numérico en representación decimal, nuevamente ocupamos la ecuación (15):

$$A \times 16^2 + 1 \times 16^1 + F \times 16^0 = ? \tag{7}$$

En este caso tenemos un problema al aplicar directamente la ecuación: en la expresión aparecen símbolos (A y F) que no son válidos en la representación decimal. Para solucionar esto, podemos convertir estos símbolos directamente a representación decimal: A=10 y F=15, y con estos valores reescribir la expresión para obtener el resultado:

$$10 \times 16^2 + 1 \times 16^1 + 15 \times 16^0 = 2560 + 16 + 15 = 2591 \tag{8}$$

Este ejemplo demuestra una regla importante: si ocupamos la ecuación (15) para convertir un número de una representación A a otra representación B, en el caso de que la base de A sea mayor que B (por ejemplo dieciséis ¿diez) necesariamente necesitamos un paso previo antes de realizar el cálculo: convertir todos los símbolos no válidos de la representación A a la representación B (en el ejemplo $A = 10$ y $F = 15$).

A modo de ejemplo de la regla anterior, veamos el caso de convertir un número decimal a representación binaria, por ejemplo el número 123. Si aplicamos directamente la ecuación (15) tenemos:

$$1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 = 1 \times 100 + 2 \times 10 + 3 \times 1 \tag{9}$$

El primer problema que se nos presenta es el explicado en la regla anterior: los símbolos 2 y 3 no existen en la representación binaria, por lo cual debemos en primer lugar reemplazarlos por su representación: $2 = (10)_2$ y $3 = (11)_2$:

$$(1)_2 \times 100 + (10)_2 \times 10 + (11)_2 \times 1 \tag{10}$$

Ahora tenemos un segundo problema: los números (100 y 10) están representados en decimal, por tanto debemos también reemplazarlos por su valor en esta representación: $100 = (1100100)_2$ y $10 = (1010)_2$.

$$(1)_2 \times (1100100)_2 + (10)_2 \times (1010)_2 + (11)_2 \times (1)_2 = (1100100)_2 + (10100)_2 + (11)_2 = (1111011)_2 \tag{11}$$

Este ejemplo nos muestra una clara desventaja de ocupar la ecuación (15) para transformar desde representación decimal a una de menor base: necesitamos saber a priori la representación de la base y todos las potencias de la base (en este caso 100 y 10) en la representación no decimal para realizar el cálculo. Más adelante estudiaremos otros mecanismos que permiten realizar esta conversión sin necesidad de este conocimiento.

Nota: En general, cuando todos los números que se estén usando sean de la misma representación, se puede obviar la notación $(num)_{base}$. Para el caso de número binarios y hexadecimales, existen otras notaciones habituales usadas cuando se quiere diferenciar la representación:

- Un número binario $(num)_2$ se suele escribir también como $numb$, por ejemplo $(1011)_2$ se puedes escribir como $1011b$
- Un número hexadecimal $(num)_{16}$ se suele escribir también como $numh$, por ejemplo $(A1)_{16}$ se puedes escribir como $A1h$
- Un número hexadecimal $(num)_{16}$ se suele escribir también como $0xnum$, por ejemplo $(A1)_{16}$ se puedes escribir como $0xA1$

2.3. Aritmética en distintas representaciones

La gran ventaja de las representaciones posicionales es que los procedimientos aritméticos como suma y multiplicación son equivalentes para toda representación. Estudiaremos primero la aritmética de la representación decimal, a la cual estamos habituados, y a partir de ésta generalizaremos las reglas de la suma que son válidas para cualquiera de estas representaciones.

Como ejemplo realizaremos paso a paso la suma entre los números 112 y 93. El algoritmo tradicional para realizar la suma es el siguiente:

1. Escribir uno de los números debajo del otro, alineados por la derecha:

$$\begin{array}{r} 132 \\ 93 \\ \hline \end{array}$$

2. Sumar los dos dígitos de más a la derecha. Si la suma es menor que diez (5 en este caso), continuar con el siguiente dígito hacia la izquierda:

$$\begin{array}{r} 132 \\ 93 \\ \hline 5 \end{array}$$

3. Sumar los siguientes dos dígitos. Si la suma es menor que diez, continuar con el siguiente dígito hacia la izquierda. Si la suma es es mayor o igual que diez (12 en este caso), restarle 10 a la suma y colocar como resultado la resta (2 en este caso). Convertir los 10 en 1 y agregarlo como sumando a los siguiente dígitos (este valor se denomina **acarreo** o **carry** en inglés) :

$$\begin{array}{r} 1 \\ 132 \\ 93 \\ \hline 25 \end{array}$$

4. Sumar los siguientes dos dígitos. En caso de haber acarreo sumarlo también. Revisar los mismo casos que antes :

$$\begin{array}{r} 1 \\ 132 \\ 93 \\ \hline 225 \end{array}$$

El algoritmo general para la suma en representación decimal de dos números $num1$ y $num2$ sería:

1. Comenzar desde la derecha en la posición 0 de ambos números.
2. Mientras no lleguemos al fin de ambos números por la izquierda ni queden acarreos sin sumar:
   a) Sumar los dígitos de la posición actual más algún posible acarreo previo.
   b) Si la suma es menor que 10, colocar en la posición actual del resultado el valor de la suma.
   c) Si la suma es mayor igual que 10, restarle 10 a la suma, colocar lo que resulta de la resta en la posición actual del resultado y agregar un acarreo para la siguiente posición.

En el algoritmo anterior usamos solamente dos veces el número 10: al comparar que la suma de los dígitos sea mayor que 10 y luego en caso de que se cumpla esto, al restarle 10 al valor obtenido. En base a esto podemos generalizar este algoritmo para sumar cualquier par de números en representación posicional de base b reemplazando el número 10 por la variable b:

1. Comenzar desde la derecha en la posición 0 de ambos números.
2. Mientras no lleguemos al fin de ambos números por la izquierda ni queden acarreos sin sumar:
   a) Sumar los dígitos de la posición actual más algún posible acarreo previo.
   b) Si la suma es menor que b, colocar en la posición actual del resultado el valor de la suma.
   c) Si la suma es mayor igual que b, restarle b a la suma, colocar lo que resulta de la resta en la posición actual del resultado y agregar un acarreo para la siguiente posición.

Usemos este algoritmo con un ejemplo de dos números binarios $(110)_2$ y $(11)_2$, en el cual la base $b = 2$:

1. Comenzar desde la derecha en la posición 0 de ambos números:

$$\begin{array}{r} 110 \\ 11 \\ \hline \end{array}$$

2. Mientras no lleguemos al fin de ambos números por la izquierda ni queden acarreos sin sumar

   a) Sumar los dígitos de la posición actual más algún posible acarreo previo. Si la suma **(en este caso 1)** es menor que 2 , colocar en la posición actual del resultado el valor de la suma.

$$\begin{array}{r} 110 \\ 11 \\ \hline \mathbf{1} \end{array}$$

   b) Sumar los dígitos de la posición actual más algún posible acarreo previo. Si la suma **(en este caso 2)** es mayor igual que 2 , restarle 2 a la suma, colocar lo que resulta de la resta en la posición actual del resultado **(en este caso 0)** y agregar un acarreo para la siguiente posición.

$$\begin{array}{r} \mathbf{1} \\ 110 \\ 11 \\ \hline \mathbf{0}1 \end{array}$$

   c) Sumar los dígitos de la posición actual más algún posible acarreo previo. Si la suma **(en este caso 2)** es mayor igual que 2 , restarle 2 a la suma, colocar lo que resulta de la resta en la posición actual del resultado **(en este caso 0)** y agregar un acarreo para la siguiente posición.

$$\begin{array}{r} \mathbf{1}1 \\ 110 \\ 11 \\ \hline \mathbf{0}01 \end{array}$$

   d) Sumar los dígitos de la posición actual más algún posible acarreo previo. Si la suma **(en este caso 1)** es menor que 2 , colocar en la posición actual del resultado el valor de la suma.

$$\begin{array}{r} 11 \\ 110 \\ 11 \\ \hline \mathbf{1}001 \end{array}$$

Ejercicio: Multiplique los números binarios $(101)_2$ y $(10)_2$. Para hacerlo, primero analice el algoritmo de multiplicación de números decimales y luego aplíquelo a la multiplicación de números binarios.

2.4. Algoritmos de conversión entre representaciones

Como se analizó previamente la ecuación (15) representa un método general de conversión entre bases. Esta ecuación funciona bien para transformar un número en representación no decimal a la decimal (en la cual podemos realizar operaciones aritméticas rápidamente), sin embargo, como también se vio, para el caso inverso no era tan útil, y por tanto es necesario explorar otros algoritmos. Adicionalmente, para el caso de los números binarios existen otros algoritmos o heurísticas que pueden resultar más simples.

En esta sección se revisarán primero algoritmos de conversión binario-decimal y decimal-binario, y luego algoritmos de conversión entre representaciones hexadecimal, octal y binaria.

2.4.1. Algoritmos de conversión binario-decimal

La ecuación (15) representa un método útil para convertir entre números binarios y decimales, sin embargo, requiere que seamos capaces de recordar rápidamente las potencias del número dos, lo cual puede ser fácil para potencias bajas, pero para potencias altas puede volverse complejo y por tanto ser necesario tener que calcular estas potencias previo a la conversión. A continuación se presentan dos algoritmos que permiten convertir un número binario a decimal sin conocer las potencias de dos.

Algoritmo de acarreo inverso

La idea de este algoritmo es comenzar desde la izquierda e ir invirtiendo el proceso de acarreo, es decir por cada vez que vemos un número 1 lo devolvemos a la derecha como un número 2. Veamos el algoritmo a través de un ejemplo: vamos a convertir el número $(101101)_2$ a representación decimal.

1. Tomamos el primer 1, **1**01101, lo convertimos en 2 y lo sumamos al dígito de la derecha: 0**2**1101
2. Repetimos el paso para el siguiente dígito. Esta vez tenemos un 2 lo que equivale a dos 1 y por tanto debemos sumar dos veces 2 = 4 al siguiente dígito de la derecha: 00**5**101
3. Repetimos el paso para el siguiente dígito. Esta vez tenemos un 5 lo que equivale a cinco 1 y por tanto debemos sumar cinco veces 2 = 10 al siguiente dígito de la derecha: 000**11**01
4. Repetimos y ahora sumamos 22 al siguiente dígito de la derecha: 0000**22**1
5. Repetimos y ahora sumamos 44 al siguiente dígito de la derecha: 00000**45**. Llegamos al fin del número, y por tanto tenemos que el resultado de la conversión es 45.

Algoritmo de multiplicar e incrementar

Este algoritmo es equivalente al anterior, pero presenta reglas más simples y no requiere pensar el proceso de acarreo inverso. Las reglas de este algoritmo son:

- Tomar el 1 de más a la izquierda del número binario, e ir avanzando de izquierda a derecha. Comenzar con el resultado en 1.
- Por cada número 0 que se encuentra, multiplicar el resultado actual por 2.
- Por cada número 1 que se encuentra, multiplicar el resultado actual por 2 y sumar 1.

Veamos el algoritmo a través de un ejemplo: vamos a convertir el número $(101101)_2$ a representación decimal.

1. Tomamos el 1 de más a la izquierda, **1**01101. Comenzamos con el resultado = 1.
2. Tomamos el 0 que sigue, 1**0**1101: multiplicamos por 2 el resultado: resultado = 2
3. Tomamos el 1 que sigue, 10**1**101: multiplicamos por 2 el resultado y sumamos 1: resultado = 5
4. Tomamos el 1 que sigue, 101**1**01: multiplicamos por 2 el resultado y sumamos 1: resultado = 11
5. Tomamos el 0 que sigue, 1011**0**1: multiplicamos por 2 el resultado: resultado = 22
6. Tomamos el 1 que sigue, 10110**1**: multiplicamos por 2 el resultado y sumamos 1: resultado = 45

2.4.2. Algoritmos de conversión decimal-binario

Como mencionamos anteriormente, la ecuación (15) no es un método práctico para realizar esta conversión. A continuación presentaremos dos algoritmos para realizar la conversión:

Algoritmo de acarreos sucesivos

La idea de este algoritmo es comenzar con todos los número acumulados a la derecha (se puede pensar como si se tuvieran fichas acumuladas como una torre) e ir acarreando grupos de a dos hacia la izquierda, convirtiéndolos en 1 a medida que se acarrean.

Veamos el algoritmo a través de un ejemplo: vamos a convertir el número 45 a representación binaria.

1. Comenzamos con 45 «fichas» a la derecha.
2. Acarreamos todos los pares de fichas que hayan a la izquierda, en este caso 22 y por cada par, sumamos un 1 a la izquierda. En la primera posición nos sobra una ficha, lo que representa un número 1. En este momento llevamos el número 22 **1**
3. Acarreamos todos los pares de fichas que hayan en la próxima posición, en este caso 11 y por cada par, sumamos un 1 a la izquierda. Esta vez no nos sobra una ficha, lo que representa un número 0. En este momento llevamos el número 11 **0**1
4. Acarreamos todos los pares de fichas que hayan en la próxima posición, en este caso 5 y por cada par, sumamos un 1 a la izquierda. Esta vez sobra una ficha, lo que representa un número 1. En este momento llevamos el número 5 **1**01
5. Acarreamos todos los pares de fichas que hayan en la próxima posición, en este caso 2 y por cada par, sumamos un 1 a la izquierda. Esta vez sobra una ficha, lo que representa un número 1. En este momento llevamos el número 2 **1**101
6. Acarreamos todos los pares de fichas que hayan en la próxima posición, en este caso 1 y por cada par, sumamos un 1 a la izquierda. Esta vez no nos sobra una ficha, lo que representa un número 0. En este momento llevamos el número 1 **0**1101.
7. Cómo el número de más a la izquierda es un 1 no tenemos nada más que acarrear y el resultado es 101101

Algoritmo de divisiones sucesivas

Este el algoritmo es equivalente al anterior, pero permite mecanizar de mejor forma la conversión, sin tener que pensar en el acarreo. Consiste en los siguientes pasos:

- Dividir el número actual por 2: si la división es exacta (es decir, no hay resto), agregar un 0 a la izquierda del resultado. Actualizar el número actual como el resultado de la división.
- Dividir el número actual por 2: si la división no es exacta (es decir, hay resto), agregar un 1 a la izquierda del resultado. Actualizar el número actual como el resultado de la división.
- Detenerse si es que el número actual es 0.

Veamos el algoritmo a través de un ejemplo: vamos a convertir el número 45 a representación binaria.

a) Dividimos el número actual 45 en dos = 22, resto = 1. Como hay resto, agregamos un 1 a la izquierda del resultado=**1**.

b) Dividimos el número actual 22 en dos = 11, resto = 0. Como no hay resto, agregamos un 0 a la izquierda del resultado=**0**1.

c) Dividimos el número actual 11 en dos = 5, resto = 1. Como hay resto, agregamos un 1 a la izquierda del resultado=**1**01.

d) Dividimos el número actual 5 en dos = 2, resto = 1. Como hay resto, agregamos un 1 a la izquierda del resultado=**1**101.

e) Dividimos el número actual 2 en dos = 1, resto = 0. Como no hay resto, agregamos un 0 a la izquierda del resultado=**0**1101.

f) Dividimos el número actual 1 en dos = 0, resto = 1. Como hay resto, agregamos un 1 a la izquierda del resultado=**1**01101.

g) Nos detemos por que el resultado actual es 0.

2.4.3. Algoritmos de conversión entre representaciones hexadecimal, octal y binaria

La conversión entre representaciones binaria y octal y binaria y hexadecimal es mucho más simple que la conversión entre binarios y decimales. Esto se debe a la relación de sus bases: ocho y dieciséis son potencias de dos. De esta forma, para convertir un número binario a octal, basta ir agrupando de a **tres** ($8 = 2^{\mathbf{3}}$) dígitos binarios e ir reemplazando su valor por el número octal correspondiente. A modo de ejemplo, realicemos la conversión del número $(10110)_2$ a octal:

binario:	010 110
octal:	2 6

Se observa además que el método funciona en ambas direcciones: para convertir el número octal $(26)_8$ en binario basta tomar cada uno de sus dígitos, representarlo como número binario y luego ubicarlos sucesivamente: $(2)_8 = (010)_2$ y $(6)_8 = (110)_2$ por lo tanto $(26)_8 = (10110)_2$

De manera equivalente, la conversión binaria-hexadecimal consiste en ir agrupando de a **cuatro** ($16 = 2^{\mathbf{4}}$) dígitos binarios e ir reemplazando su valor por el número hexadecimal correspondiente. A modo de ejemplo, realicemos la conversión del número $(10110)_2$ a hexadecimal:

binario:	0001 0110
hexadecimal:	1 6

Al igual que con los octales, la conversión es bidireccional: para convertir el número hexadecimal $(16)_{16}$ en binario basta tomar cada uno de sus dígitos, representarlo como número binario y luego ubicarlos sucesivamente: $(1)_{16} = (0001)_2$ y $(6)_{16} = (0110)_2$ por lo tanto $(16)_{16} = (10110)_2$.

Debido a esta fácil conversión entre representaciones, es habitual que se ocupen tanto la representación octal como la hexadecimal para referirse a números binarios, ya que requieren menos símbolos.

2.5. Representación de números negativos

Hasta ahora hemos visto representaciones sólo de números enteros y positivos. La primera pregunta que surge es ¿cómo representamos números negativos? Nuestra experiencia nos dice que la respuesta es simple: agregamos un «-» a la izquierda del número, por ejemplo -12 sería un número decimal negativo; $(-1100)_2$ sería un número binario negativo.

El problema de esto es que necesitamos agregar un nuevo símbolo a nuestro sistema numérico para poder representar números negativos. En el caso de los números binarios, en vez de tener sólo dos símbolos, necesitaríamos tener uno adicional sólo para indicar que un número es negativo. Nos gustaría buscar un mecanismo que aproveche los símbolos que ya tenemos en nuestra representación, y así evitar tener que agregar un símbolo nuevo.

A continuación se revisarán distintos métodos para representar números negativos binarios ocupando sólo los símbolos 0 y 1.

Dígito de signo

Un primer método que se puede ocupar es agregar un dígito extra a la izquierda del número que en caso de ser 0 indica que el número es positivo, y en caso de ser 1, que el número es negativo. Por ejemplo si tenemos el número 10011 y queremos representar el número -10011, reemplazamos el «-» por un 1, obteniendo: **1**10011.

Importante: al ocupar este tipo de representaciones ocurre que ahora no es suficiente ver la secuencia de símbolos para saber su valor: debemos saber además que **tipo** de información se está guardando, por ejemplo en este caso: un número negativo, tal que el primer dígito indica el signo.

El problema de esta representación es que no nos sirve para realizar operaciones aritméticas. Para que una representación de un número negativo sirva debe cumplir la ecuación: $A + (negativo(A)) = 0$ es decir, debe ser un inverso aditivo válido.

En nuestro caso, si al número positivo **0**10011 le sumamos el número negativo **1**10011 obtenemos 1000110. Debemos entonces buscar otras representaciones para poder realizar operaciones aritméticas con números negativos.

Complemento a 1

Una posible mejora corresponde la representación denominada «complemento a 1». Esta consiste en agregar un dígito de signo, al igual, que en el caso anterior, pero además se reemplazan todos los 0s por 1s y los 1s por 0s, sin contar el dígito de signo. Siguiendo nuestro ejemplo con el número 10011, primero se le agrega un cero a la izquierda: **0**10011, lo que se interpreta como «agregar el bit de signo al número positivo». Luego se hace la inversión para todos los dígitos **1**01100, lo que nos deja un número «negativo» dado que su bit de signo es 1.

Ahora si realizamos la suma entre **0**10011 y **1**01100 obtenemos **1**11111. Si hacemos lo mismo con el número 101 por ejemplo, obtendremos que el número negativo sería **1**010 y si los sumamos, obtenemos **1**111.

Podemos observar que aparece un patrón común al ocupar complemento a 1: la suma entre un número y su inverso aditivo resultará en una secuencia de unos. Esto nos indica que estamos por buen camino, y nos falta sólo un detalle para llegar a una representación correcta.

Complemento a 2

La representación que soluciona el problema del complemento a 1, se denomina «complemento a 2». Consiste en ejecutar los mismos pasos que en complemento a 1, y adicionalmente sumarle 1 al número. Veamos paso a paso con un ejemplo, el número 10011:

- El primer paso consiste en agregar el cero a la izquierda: **0**10011
- Luego se realizar el reemplazo de 0s y 1s: **1**01100
- Por último, le sumamos un 1 al número: **1**01101

Ahora probaremos si efectivamente esta representación nos entrega un inverso aditivo válido. Si sumamos **0**10011 con **1**01101 obtenemos: **1**000000, lo que podemos interpretar como «0», ya que el 1 de más a la izquierda lo podemos interpretar como el signo, y sabemos que $0 = -0$ por lo tanto se cumple que la representación en complemento a 2 si funciona como inverso aditivo.

De ahora en adelante, cuando hablemos de un número binario negativo, asumiremos que está representado en complemento a 2, a menos que se diga lo contrario.

3. Representación de números reales

Los números enteros representan sólo un porcentaje menor de todos los posibles números que se pueden representar, por lo que es necesario estudiar como representar números fraccionales y reales en un computador. Sin embargo, la representación de números fraccionales presenta una serie de complicaciones y limitaciones que debemos entender para poder ocuparlas correctamente y evitar errores.

3.1. Fracciones decimales y binarias

Un número entero en representación decimal puede ser representado en su forma posicional como la suma de sus dígitos ponderado por potencias de la base 10. Por ejemplo el número 112 puede representarse como:

$$1 \times 10^2 + 1 \times 10^1 + 2 \times 10^0 = 112 \tag{12}$$

De manera similar, un número en representación binaria puede ser también representado en su forma posicional como la suma de sus dígitos ponderado por potencias de la base, en este caso 2. Por ejemplo el número $(1100)_2$ puede representarse como:

$$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 8 + 4 + 0 + 0 = 12 \tag{13}$$

La representación posicional puede ser extendida para números fraccionarios, ocupando potencias negativas de la base al ponderar. Por ejemplo el número en representación decimal 112,234 puede representarse como:

$$1 \times 10^2 + 1 \times 10^1 + 2 \times 10^0 + 2 \times 10^{-1} + 3 \times 10^{-2} + 4 \times 10^{-3} = 100 + 10 + 2 + 0{,}2 + 0{,}03 + 0{,}004 = 112{,}234 \tag{14}$$

De manera equivalente, podemos extender la representación posicional para números fraccionarios en representación binaria. Por ejemplo el número $(1100{,}011)_2$ puede representarse como:

$$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = 8 + 4 + 0 + 0 + 0 + 0{,}25 + 0{,}125 = 12{,}375 \tag{15}$$

Al igual que en los números enteros, la ecuación (4) puede ser interpretada como un algoritmo de conversión binario-decimal. Nos gustaría encontrar también un algoritmo de conversión decimal-binario para números fraccionales, de manera de por ejemplo obtener la representación del número 0,1 en binario. Un algoritmo simple es el siguiente:

- Reescribir el número decimal en su forma fraccional: $0{,}1 = \frac{1}{10}$
- Transformar el numerador y denominador a binario: $\frac{1}{10} = \frac{(1)_2}{(1010)_2}$
- Realizar la división: $1 : 1010 = ?$

Para poder completar este algoritmo, debemos primero revisar como se realiza la división de números binarios.

División decimal y binaria

Tal como en el caso de la multiplicación y la suma, la división de números binarios ocupa exactamente el mismo procedimiento que su contraparte decimal. Revisaremos primero, entonces, un ejemplo de una división decimal: 60 : 25:

- El primero paso corresponde a ver cuantas veces cabe completamente el divisor (número de la derecha) en el dividendo (número de la izquierda). En este caso cabe 2 veces, ya que $2 \times 25 = 50$, por lo cual lo escribimos como primer dígito del resultado el número 2 y debajo del dividendo el valor efectivo de la multiplicación entre el resultado y el divisor, en este caso 50:

$$\begin{array}{l} 60 : 25 = 2 \\ 50 \end{array}$$

- El siguiente paso consiste en restarle al dividendo el resultado de la multiplicación resultado-divisor. En este caso $60 - 50 = 10$:

$$\begin{array}{rl} & 60 : 25 = 2 \\ - & 50 \\ \hline & 10 \end{array}$$

- Ahora se repite el primer paso, pero esta vez ocupando como dividendo el resultado de la resta (10). En caso de que el dividendo sea menor que el divisor, agregamos 0s a la derecha del dividendo hasta que este sea mayor o igual que el divisor. Por cada 0 que se agrega en el dividendo, se compensa avanzando en un dígito fraccional del resultado a la derecha. En este caso basta agregar un 0 y como tal quedamos posicionados en el primer dígito fraccional del resultado

$$\begin{array}{rl} & 60 : 25 = 2. \\ - & 50 \\ \hline & 10\mathbf{0} \end{array}$$

- Finalmente dividimos ahora si el dividendo actual por el divisor, en este caso nos da como resultado 4. Como el divisor cabe exactamente en el dividendo, nos detenemos y el resultado final es 2,4.

$$\begin{array}{rl} & 60 : 25 = 2{,}4 \\ - & 50 \\ \hline & 100 \\ - & 100 \\ & 0 \end{array}$$

La divisón binaria es equivalente. La única diferencia es que las operaciones aritméticas intermedias deben ser realizadas ocupando aritmética binaria. Veremos el proceso con un ejemplo: $3 : 4 = (11)_2 : (100)_2$

- El primero paso corresponde a ver cuantas veces cabe completamente el divisor en el dividendo. En este caso no cabe, y por tanto debemos aplicar la técnica de agregar 0s al dividendo y desplazarnos en los dígitos fraccionales

$$11\mathbf{0} : 100 = 0.$$

- Al agregar un 0 el divisor (4) cabe una vez en el dividendo (6), por tanto agregamos un 1 al resultado, multiplicamos el resultado por el divisor y se lo restamos al dividendo, obteniendo en este caso como resto $(10)_2 = 2$.

$$\begin{array}{rl} & 110 : 100 = 0{,}1 \\ - & 100 \\ \hline & 010 \end{array}$$

- Repetimos el paso de agregar un 0 en el resto, que es ahora nuestro nuevo dividendo, quedando este con el valor $(100)_2 = 4$. Vemos que el divisor cabe exactamente 1 vez en el dividendo, y por tanto agregamos un 1 al resultado, y obtenemos resto 0 lo que nos indica que terminamos la división.

$$\begin{array}{rl} & 110 : 100 = 0{,}11 \\ - & 100 \\ \hline & 0100 \\ - & 0100 \\ \hline & 0 \end{array}$$

Podemos comprobar que el resultado es correcto convirtiéndolo a decimal:

$$0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 0 + 0{,}5 + 0{,}25 = 0{,}75 = \frac{3}{4} \tag{16}$$

Ahora que sabemos como dividir en binario, podemos completar el último paso de nuestro algoritmo de conversión, que era dividir $(1)_2 : (1010)_2$:

- Primero agregamos ceros hasta que el divisor quepa. Una vez conseguido esto multiplicamos por 1 el divisor y se lo restamos al dividendo.

$$\begin{array}{rl} & 1\mathbf{0000} : 1010 = 0{,}0001 \\ - & 01010 \\ \hline & 00110 \end{array}$$

- Repetimos el proceso con el nuevo divisor, agregando un 1 al resultado, restando la multiplicación resultado divisor en el dividendo, y actualizando el dividendo como el resto.

$$\begin{array}{rl} & 10000 : 1010 = 0{,}00011 \\ - & 01010 \\ \hline & 00110\mathbf{0} \\ - & 001010 \\ \hline & 000010 \end{array}$$

- Agregamos los 0s necesarios nuevamente para continuar el proceso

$$\begin{array}{rl} & 10000 : 1010 = 0{,}00011001 \\ - & 01010 \\ \hline & 00110\mathbf{0} \\ - & 001010 \\ \hline & 000010\mathbf{000} \\ - & 000001010 \\ \hline & 000000110 \end{array}$$

Podemos notar que a esta altura estamos repitiendo divisiones que ya hicimos exactamente igual, y por tanto sabemos que nunca terminaremos esta división. De hecho si continuamos dividiendo observaremos que el resultado es de la forma 0,00011001100110011... lo que corresponde a un número infinito semi-periódico: $0{,}0\overline{0011}$.

Este resultado es completamente contraintuitivo e inesperado, por que básicamente nos dice que **el número** 0,1 **tiene una representación infinita en binario**. Podemos ir incluso más allá: esto demuestra que dado un número fraccional cualquiera, el hecho de que su representación sea finita o infinita depende

exclusivamente de la base utilizada en la representación. Por ejemplo la fracción $\frac{1}{3}$ que en base decimal tiene la representación infinita periódica $0.\overline{3}$, en base ternaria (3) tendrá la representación finita 0,1.

La relevancia de esto es que dado que los computadores tienen un espacio finito para almacenar información, si ocupamos números binarios para representar números que en base decimal son finitos (como el 0,1), pero son infinitos en base binaria, obligatoriamente podemos almacenar sólo una aproximación de éste, lo que afectará en los resultados de operaciones que realicemos con este tipo de números.

3.2. Representaciones en un computador

Debido a que los computadores tienen espacio de almacenamiento limitado, los números que se almacenen en estos se guardan en porciones limitadas también. En general un número se almacenará mediante una cantidad fija de dígitos binarios (conocidos como **bits** de su nombre en inglés **bi**nary digi**ts**). Esto aplica tanto para números enteros como fraccionales, pero como se vio en la sección anterior es de particular importancia para los números fraccionales, que en muchos caso tendrá representación infinita, y como tal, al tener una cantidad fija de bits se deberán almacenar aproximaciones de los números.

La forma específica en que se guardan los números fraccionales en los bits del almacenamiento de un computador ha variado en el tiempo, pero son dos las principales representaciones usadas: punto fijo y punto flotante. La mayoría de los computadores actuales ocupa la segunda, pero ambas tienen posibles ventajas y desventajas que se deben considerar.

3.2.1. Representación de punto fijo (fixed point)

La representación de punto fijo consiste en que dado un espacio de n bits para almacenar un número, se reservan t bits para almacenar la parte entera del número y f bits para almacenar la parte fraccional, donde $n = t + f + 1$ (el bit extra se utiliza para almacenar el signo). De esta forma el punto (o coma) de la representación fraccional queda «fijo» en la t-ésima posición de la secuencia de bits.

Como ejemplo supongamos el número binario fraccional 10,111. Si tenemos $n = 8$ bits para almacenar todo el número, $t = 4$ bits para almacenar la parte entera y $f = 4$ bits para almacenar la parte fraccional, la representación almacenada del número sería:

0	010	1110
signo	t	f

3.2.2. Representación de punto flotante (floating point)

El problema que tiene la representación de punto fijo es que limita el **rango** posible de números. Para el ejemplo anterior ($n = 8$, $t = 3$ y $f = 4$) el máximo número positivo que podemos representar es el 111,1111 y el mínimo es 000,0001. Si pudiésemos mover o «flotar» el punto (o coma) libremente entre los 7 bits podríamos representar el número 1111111 y también el 0,000001, lo que nos daría un mayor rango, para así permitir trabajar tanto con números muy grandes como con números muy chicos. La representación usada para lograr esto se denomina representación de «punto flotante».

Para lograr que el punto «flote» se debe codificar de alguna forma para cada número la posición actual del punto. Una representación decimal que permite esto es la representación de notación científica, la cual codifica un número como una multiplicación entre un **significante** con una base (10) elevada a un **exponente**. En esta representación, el significante representa el valor del número y, dado que multiplicar por una potencia de 10 en representación decimal es equivalente a mover el punto, el valor del exponente está indicando la posición del punto.

Por ejemplo, el número 1023,456 se puede codificar en notación científica como: $1{,}023456 \times 10^3$. En este caso el significante sería 1,023456, la base 10 y el exponente 3, que se puede interpretar como «mover el punto 3 posiciones a la derecha». El número también podría codificarse como $10{,}23456 \times 10^2$ o $102345{,}6 \times 10^{-2}$, pero

en general se prefiere la que se uso inicialmente, con un sólo dígito a la izquierda de la coma del significante. Cuando un número en notación científica cumple con esta condición, se denomina **normalizado**.

La representación de punto flotante usada en el computador aplica la misma codificación de la notación científica, pero ahora con números binarios. De esta forma, ahora el significante y el exponente son representados como números binarios. Para mantener el hecho de que el exponente codifique la posición del punto, se utiliza como base el número 2 en vez de la base 10, ya que en representación binaria multiplicar por una potencia de 2 es equivalente a mover el punto.

De esta forma, por ejemplo, el número 10,111, lo podemos representar como $(1{,}0111)_2 \times 2^{(01)_2}$. El exponente $(01)_2 = 1$ al igual que en notación científica, lo interpretamos como «mover el punto 1 posición a la derecha».

Si queremos almacenar este número en $n = 8$ bits y definimos nuestra representación de manera de tener $s = 3$ bits de significante (normalizado), 1 bit de signo para el significante, $e = 3$ bits de exponente y 1 bit de signo para el exponente, podríamos almacenar el número de la siguiente forma:

0	101	0	001
signo s	s	signo e	e

Esto nos muestra de inmediato una clara desventaja de esta representación respecto a la de punto fijo: existe una pérdida de **precisión** es decir, de la cantidad de bits disponibles para almacenar un determinado valor. La precisión de un número de punto flotante está dada por la cantidad de bits de su significante, en este caso 3. La precisión de un número de punto fijo en cambio está dada por la cantidad de bits totales usadas por el número, en nuestro caso 7.

La ventaja es que aumentamos el **rango**: el máximo valor positivo representable en este caso es $(1{,}11)_2 \times 2^{(111)_2} = 11100000$ y el mínimo es $(0{,}01)_2 \times 2^{-(111)_2} = 0{,}00000001$. Este es un trade-off inevitable para una cantidad limitada de bits: para aumentar al rango, debemos reducir la precisión y vice-versa. La representación de punto flotante se prefiere por que la pérdida de precisión se puede compensar, en parte, aumentando la cantidad de bits.

El estándar IEEE754

La representación de punto flotante antes descrita es una de muchas que se podría utilizar. Los parámetros relevantes para una representación son: el número total de bits, el número de bits asignados al significante, la normalización o no del significante y el número de bits asignados al exponente. En 1985 se definió el estándar IEEE754 que especifica como representan los computadores un número de punto flotante. El estándar define varias representaciones siendo dos las principales: «**single** precision floating point» y «**double** precision floating point».

La representación «single» (conocida en los lenguajes de programación Java y C# como **float**) define un tamaño de 32 bits para los números, de los cuáles se ocupa 1 bit para el signo del significante, 23 bits para el valor del significante y 8 bits para el exponente:

1 bit	8 bits	23 bits
signo significante	exponente	significante

Esta representación tiene ciertas características especiales:

- El significante se almacena normalizado, pero sin el 1 que va a la izquierda de la coma. Por ejemplo el significante 1,101, se almacena como 10100000000000000000000, es decir se asume que todo significante comienza en 1. La ventaja de tener este dígito implícito es que aunque se almacenan 23 bits, la precisión dél número es de 24 bits.

- El exponente se almacena desfasado en 127, es decir en vez de almacenar un bit de signo aparte, o representar el número en complemento a 2, se desafasa el número de manera de tener sólo valores positivos. La ventaja de esto está en hacer más simple la aritmética.
- Dado que se tiene al significante se le agrega un 1 implícito a la izquierda del punto, es imposible representar directamente el número cero. Para representarlo, se reservó el exponente 00000000 y se definió que la representación del número 0 es la secuencia con ese exponente y con 0s en el significante. Dada esta definición existen dos posibles 0s: $+0 = 00000000000000000000000000000000$ y $-0 = 10000000000000000000000000000000$
- El exponente 11111111 también se reservó, para poder representar ciertos números especiales:
  - +Infinito : 01111111100000000000000000000000
  - -Infinito : 11111111100000000000000000000000
  - NaN: not a number : $01111111xxxxxxxxxxxxxxxxxxxxxxx$ donde alguno de los x debe cumplir con ser distinto de 0.

La representación «double» define un tamaño de 64 bits para los números, de los cuáles se ocupa 1 bit para el signo del significante, 52 bits para el valor del significante y 11 bits para el exponente:

1 bit	11 bits	52 bits
signo significante	exponente	significante

Las características especiales de la representación «single» también aplican a esta, diferenciándose en que: la precisión total, contando el bit implícito es de 53 bits; el exponente está desfasado en 1023.

Aritmética de punto flotante

A diferencia de los números enteros y de la representación de punto fijo, un número representado como punto flotante requiere un manejo aritmético distinto. Veremos que es esta aritmética especial, en particular el caso de la suma y resta, una de las causas principales de los problemas de esta representación.

Multiplicación y división

Vamos a comenzar con la multiplicación y división que en punto flotante son operaciones más simples. Revisemos primero estas operaciones en notación científica, veremos que son transferibles los algoritmos a punto flotante.

Tomemos como ejemplo los números $1{,}2 \times 10^2$ y 2×10^{-1}. El algoritmo de multiplicación es el siguiente:

- El significante del resultado se obtiene como la multiplicación de los significantes de los multiplicandos: $1{,}2 \times 2 = 2{,}4$.
- El exponente del resultado se obtiene como la suma de los exponentes de los multiplicandos: $2 + (-1) = 1$.
- Resultado final: $2{,}4 \times 10^1$

La explicación del algoritmo es simple:

- Si realizamos la multiplicación directamente obtenemos: $1{,}2 \times 10^2 \times 2 \times 10{-}1$.
- Luego, si agrupamos los significantes y las potencias obtenemos: $1{,}2 \times 2 \times 10^2 \times 10^{-1}$

- El primer paso entonces era multiplicar los significantes: $2{,}4 \times 10^2 \times 10^{-1}$
- Ahora multiplicamos las potencias, y sabemos que la regla para multiplicar potencias con base igual es sumar los exponentes: $2{,}4 \times 10^1$

El algoritmo para los números de punto flotante es equivalente. Supongamos nuestra representación previa ($n = 8$, $s = 3$, $e = 3$) y los números $(1{,}1)_2 \times 2^{-(1)_2} = (0\mathbf{1101001})_{float}$ y $(1{,}0)_2 \times 2^{(1)_2} = (0\mathbf{1000001})_{float}$:

- El significante del resultado se obtiene como la multiplicación de los significantes de los multiplicandos: $(1{,}1)_2 \times (1)_2 = (1{,}1)_2$.
- El exponente del resultado se obtiene como la suma de los exponentes de los multiplicandos: $(1)_2 + (-1)_2 = 0$.
- Resultado final: $(1{,}1)_2 \times 2^0 = (0\mathbf{1100000})_{float}$

Suma y resta

Para sumar dos fracciones binarias representadas como punto fijo, basta ir sumando bit a bit de derecha a izquierda, acarreando cuando corresponda, lo que corresponde al mismo algoritmo que la suma de enteros. En el caso de punto flotante es distinto, ya que para poder sumar dos números deben tener el mismo exponente, lo que implica que en caso de que esto no se cumpla, debemos modificar los números para que si tengan el mismo exponente y puedan ser sumados o restados.

Veamos un ejemplo, primero con notación científica que involucra los mismos elementos aritméticos que el punto flotante: Tenemos dos números para sumar: $1{,}23 \times 10^2$ y $5{,}12 \times 10^{-1}$. Los pasos para completar la suma son los siguientes:

- Equilibrar los exponentes: debemos ajustar el menor de los números para que queden con el mismo exponente que el primero. Si restamos los exponentes tenemos una diferencia de 3, que es el número de veces que hay que mover la coma a la izquierda en el significante del menor número, resultando en: $0{,}00512 \times 10^2$"
- Una vez equilibrados los exponentes, se procede a sumar directamente los significantes: $1{,}23512 \times 10^2$

El algoritmo para los números de punto flotante es equivalente. Supongamos nuestra representación previa ($n = 8$, $s = 3$, $e = 3$) y los números $(1{,}1)_2 \times 2^{-(1)_2} = (0\mathbf{1101001})_{float}$ y $(1{,}0)_2 \times 2^{(1)_2} = (0\mathbf{1000001})_{float}$:

- Equilibrar los exponentes: debemos ajustar el menor de los números para que queden con el mismo exponente que el primero. Si restamos los exponentes tenemos una diferencia de 2, que es el número de veces que hay que mover la coma a la izquierda en el significante del menor número, resultando en: $(0{,}011)_2 \times 2^{(1)_2}$"
- Una vez equilibrados los exponentes, se procede a sumar directamente los significantes: $(1{,}011)_2 \times 2^{(1)_2}$

Tenemos un problema: el significante tiene precisión = 4 bits, y nuestro formato soporta hasta 3 bits. Inevitablemente tendremos que perder **exactitud**, por ejemplo podríamos truncar el último bit y obtener el número $(1{,}01)_2 \times 2^{(1)_2} = (0\mathbf{1010001})_{float}$. Este es uno de los problemas principales de la suma en punto flotante: a diferencia de la multiplicación (y la división), con la suma (y la resta) es muy fácil que perdamos exactitud al realizar una operación, por lo que es importante tener esto en cuenta al momento de realizar operaciones aritméticas con números de punto flotante.

Redondeo

En el ejemplo anterior, cuando obtuvimos como resultado el número $(1{,}011)_2 \times 2^{(1)_2} = 1{,}375$ notamos que dada la precisión de la representación era imposible almacenar toda la información, ya que debíamos eliminar un bit. Sin embargo, hay distintas formas en que podíamos redondear el número de 4 a 3 bits, siendo algunas opciones mejores que otras.

El método más simple es el **redondeo hacia cero** que básicamente corresponde a truncar los bits que no caben en la representación. En el ejemplo anterior, aplicar este método resulta en $(1{,}01)_2 \times 2^{(1)_2} = 1{,}25$. El problema es que este es la peor forma de redondeo, ya que introduce el mayor error y un sesgo hacia el cero.

Un mejor método se denomina **redondeo de la mitad a cero** que es básicamente el método que tradicionalmente ocupamos para redondear números decimales. Por ejemplo, si el número decimal 3,95 se debe representar en dos dígitos, se redondea a 4,0 dado que el 5 está a mitad de camino del valor de la base (10). En el caso de los números binarios, se aplica el mismo criterio: si el dígito está a mitad de camino o más de la base, se aumenta en 1 el dígito siguiente. En nuestro ejemplo, como el último dígito es 1 que es la mitad de la base (2), debemos aumentar en 1 el siguiente dígito, lo que resulta finalmente en el número: $(1{,}10)_2 \times 2^{(1)_2} = 1{,}5$. En este caso particular el error es el mismo en este caso que con el método anterior, pero en términos generales conviene realizar este redondeo, ya que se evita el sesgo de truncar hacia a cero siempre, lo que a la larga compensa en parte los errores.

3.2.3. Alternativas a la representación de punto flotante

Debido a los problemas de exactitud que pueden ocurrir con la representación de punto flotante, en muchas circunstancias se deben ocupar otro tipo de representaciones que permitan manejar de mejor forma números fraccionales.

Enteros

Una posible alternativa para manejar números fraccionales es tratarlos como enteros en otra unidad. La frase que resume esto es: «hacer cálculos monetarios directamente en centavos y no en dólares», es decir, si es posible, convertir los valores numéricos en la unidad más pequeña, de manera de siempre trabajar con enteros.

Esta alternativa tiene la ventaja de ser simple y no requerir tipos especiales, pero tiene el problema de los números enteros no entregan tanto rango como los números de punto flotante: se puede pensar que un número entero es un número de punto fijo con el punto más allá del bit menos significativo. Dado esto los números enteros presentan las mismas limitaciones que los números de punto fijo y por tanto sólo conviene usarlos si no hay mejor alternativa.

Punto flotante con base decimal

Una alternativa mejor que los números enteros es usar representación de punto flotante, pero con base 10 en vez de base 2. Esta representación tiene la ventaja de que se eliminan los casos poco intuitivos en que una representación decimal finita (como el 0,1) tiene representacion infinita en binario. Al trabajar directamente en base 10, podemos representar 0,1 simplemente como: $(1{,}0)_2 \times 10^{(-1)_2}$, es decir el significante y exponente se almacenan en binario, pero al calcular el número completo se ocupa base 10 decimal.

El estándar IEEE754 en su versión del 2008 especificó tres representaciones de punto flotante decimales: decimal32 (32 bits), decimal64 (64 bits), decimal128 (128 bits), las cuales son implementadas en muchos de los computadores modernos . En particular, el lenguaje C# provee el tipo de datos `decimal` el cual corresponde a la especificación decimal128.

La principal desventaja de esta representación es que hace mucho más lentos los cálculos, y es por eso que no es la representación principalmente usada. En esta representación multiplicar el significante por una potencia de la base **no** se traduce en sólo mover el punto, lo que complica la aritmética. Además, como el resto de los números manejados en el computador si tienen base 2, es necesario estar realizando conversiones para operar entre estos números y los de punto flotante decimal. De todas maneras, si estás representaciones están disponibles **siempre es recomendable utilizar este tipo de datos en aplicaciones que trabajen con números usados por seres humanos (como aplicaciones financieras)** para evitar problemas de exactitud.

Punto flotante con base decimal y precisión arbitraria

La representación de punto flotante decimal, aunque eliminar los errores de representación de fracciones decimales como el 0,1 no elimina la limitación de espacio presente en todas las representaciones. Una mejor representación es la denominada de punto flotante decimal con precisión arbitraria, que va dinámicamente aumentando el espacio disponible para aumentar el significante, es decir, tiene precisión sólo limitada por el tamaño total de almacenamiento disponible en el computador. Aunque no existe soporte de hardware directo para este tipo de representaciones, algunos lenguajes de programación proveen clases que permiten trabajar con estos tipos. Por ejemplo en Java está la clase `BigDecimal` que permite trabajar con puntos flotantes decimales de precisión arbitraria.

La desventaja, al igual, que en el caso anterior, está en que las operaciones son más lentas. En este caso además de las conversiones decimal-binaria, se requiere ir aumentando dinámicamente el tamaño del significante lo que agrega un overhead adicional. Este tipo de representaciones conviene usarlos sólo en casos en que se requieran precisiones altísimas, como puede ser en cálculos científicos muy sofisticados.

4. Ejercicios

- Escriba un programa en Java que convierta un String de 1s y 0s, que representan un número binario, a un String que represente: un número decimal, un número hexadecimal.
- Describa el algoritmo para la divisón de números de notación científica y aplíquelo para restar números de punto flotante.
- Describa el algoritmo para la resta de números de notación científica y aplíquelo para restar números de punto flotante.

5. Referencias

- Morris Mano, M.; Computer System Architecture, 3 Ed., Prentice Hall, 1992. Capítulo 3: Representación de datos.
- The Floating Point Guide, http://floating-point-gui.de/
- Goldberg, D.; What Every Computer Scientist Should Know About Floating-Point Arithmetic, 1991, http://docs.sun.com/source/806-3568/ncg_goldberg.html
- Hyde, R. The Art of Assembly Language, 2003. Chapter 14: Floating Point Arithmetic http://webster.cs.ucr.edu/AoA/DOS/pdf/ch14.pdf

Apéndice: algoritmo de comparación de números de punto flotante

```
    public static boolean nearlyEqual(float a, float b, float epsilon)
    {
        final float absA = Math.abs(a);
        final float absB = Math.abs(b);
        final float diff = Math.abs(a - b);

        if (a * b == 0) { // a or b or both are zero
            // relative error is not meaningful here
            return diff < (epsilon * epsilon);
        } else { // use relative error
            return diff / (absA + absB) < epsilon;
        }
    }
```

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2343 Arquitectura de Computadores

Operaciones Aritméticas y Lógicas

1. Motivación

Para poder desarrollar un computador, el primer paso es implementar un mecanismo que automáticamente sea capaz de realizar operaciones básicas, como suma y resta. Dado que hay varias operaciones y además varios tipos de representaciones de datos (complemento a dos, punto flotante, etc.) necesitamos múltiples «máquinas» capaces de realizar estos cálculos e idealmente alguna técnica que nos permita diseñar estas máquinas de manera que sean matemáticamente correctas y que utilicen componentes físicamente construibles.

2. Representación física de números binarios

Para poder realizar cálculos sobre números binarios, debemos primero definir una representación física para estos. Dado que los números binarios trabajan con sólo dos símbolos (0 y 1) es sencillo definir representaciones física del tipo «tener X cantidad de algo = 1» y «no tener nada = 0». Por ejemplo, se podría pensar en ocupar una representación basada en líquidos: tener flujo del líquido se podría interpretar como un 1 no tener nada, como un 0.

Además de la representación se requiere un mecanismo de control que permita definir el valor de un bit. En el caso del líquido, por ejemplo, podría tenerse una válvula de control, que si esta cerrada deja pasar el líquido, representando un 1 binario, y si está abierta no deja pasar líquido, representando un 0 binario. El problema de esto es que ahora además de el líquido necesitamos un mecanismo de control, que podría implementarse a su vez con líquido, pero resulta complejo. Una mejor alternativa corresponde a usar flujo eléctrico en vez de ocupar flujo de un líquido. En esta representación, la presencia de una corriente eléctrica simboliza un 1; la ausencia de corriente un 0. La principal ventaja de esta representación es que la apertura de la válvula puede ser controlada a su vez por electricidad, y de esta forma simplificamos el funcionamiento del sistema.

Un componente eléctrico que cumple con las funciones que necesitamos para representar 1s y 0s son los relés. Los relés están formados por un cable de entrada, dos de salida y un cable de control. Si el cable de control no tiene corriente, la entrada está conectada a la 2da salida (este es el estado por defecto). Si se pasa corriente por el cable de control, la entrada estará conectada a la 1era salida. De esta forma, no tener corriente en el cable control, implica que no habrá corriente en la 1era salida (lo que podemos interpretar como un 0) y si tener corriente, implica que si habrá corriente en la 1era salida (lo que podemos interpretar como un 1).

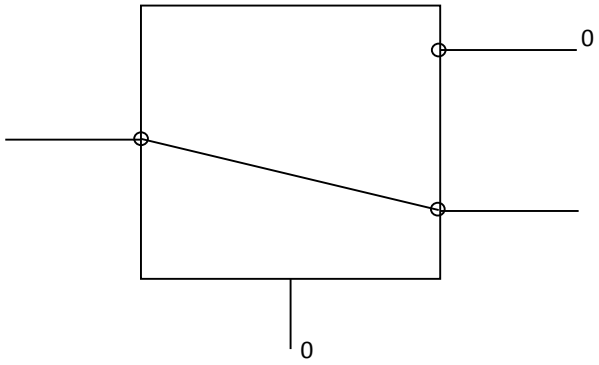


Figura 1: Relé abierto, representando un 0 binario.

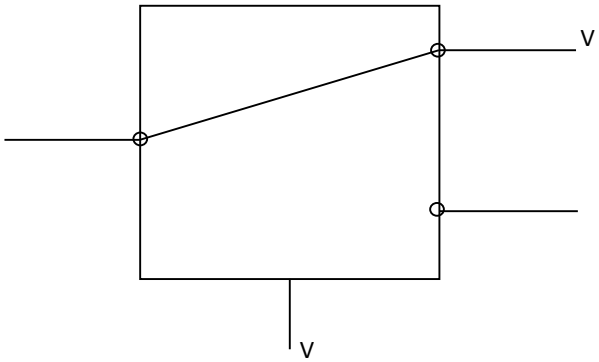


Figura 2: Relé cerrado, representando un 1 binario.

Saber la representación física de los números binarios, sin embargo, no nos ayuda a solucionar el problema de como realizar cálculo con ellos. Lo que sabemos es que debemos diseñar sistemas que sean capaces de funcionar con corriente eléctrica, pero necesitamos otras herramientas conceptuales para poder determinar que hacer con dicha corriente.

3. Lógica booleana

George Boole, un matemático inglés del siglo XIX, desarrolló un sistema formal para analizar la lógica. Este sistema lógico, denominado Lógica Booleana en su honor, fue creado sin ninguna relación directa con la problemática de automatizar el cálculo numérico, pero resultaría de suma relevancia en el siglo XX cuando se empezaron a construir los primeros computadores.

A continuación revisaremos los principales aspectos de la lógica de Boole, para luego hacer el vínculo con nuestro problema de cálculo numérico.

3.1. Proposiciones lógicas

La lógica booleana se basa en el concepto de proposiciones lógicas. Una proposición lógica es una oración o sentencia que puede ser verdadera o falsa. Por ejemplo, la sentencia: «*está lloviendo.*» será verdad si se dice un día que efectivamente está lloviendo, y será falsa si se dice en un día que no está lloviendo. La sentencia «*hoy es Viernes".*» será verdadera todos los días Viernes, y falsa todo el resto de los días.

Las proposiciones lógicas en el sistema definido por Boole, pueden tomar solamente uno de dos valores: verdadero (V) o falso (F). En este sistema no existe espacio para la ambigüedad («*Puede estar lloviendo.*» ni para las probabilidades «*Lloverá con un 60 % de probabilidad.*», sólo se consideran válidas sentencias que tiene una respuesta de verdad exacta.

3.2. Condiciones lógicos

La lógica booleana permite también conectar una secuencia de proposiciones ocupando una serie de condiciones lógicas básicas:

- podemos decir «*está lloviendo.*» **y** «*hoy es Viernes".*», lo cual forma una nueva proposición combinada que sólo será verdadera aquellos días Viernes que esté lloviendo (esta condición se conoce por su nombre en inglés como **and**);
- podríamos decir también «*está lloviendo.*» **o** «*hoy es Viernes".*», lo cual será verdadero todos los días que llueva, será verdadero todos los días Viernes y en particular también será verdadero todos los días Viernes que llueva (esta condición se conoce por su nombre en inglés como **or**);
- podemos tener también una condición que se aplique sobre una sola proposición: **no** «*está lloviendo.*» la cual será verdadera todos los días que no llueva y falsa sólo los días que llueva (esta condición se conoce por su nombre en inglés como **not**).

Cada una de estás condiciones lógicas define lo que se denomina una «tabla de verdad», es decir una tabla que asocia cuando será verdadera (V) o falsa (F) una proposición creada a partir de una o más proposiciones combinadas con estas condiciones, detallando para todas las posibles combinaciones de verdad o falsedad de éstas, cuál será el resultado.

La tabla de verdad de la condición lógica **not** al combinarla con una proposición A es:

A	not(A)
F	V
V	F

La tabla de verdad de la combinación de dos proposiciones lógicas A y B con la condición lógica **and** es:

A	B	A and B
F	F	F
F	V	F
V	F	F
V	V	V

La tabla de verdad de la condición lógica **or** es:

A	B	A or B
F	F	F
F	V	V
V	F	V
V	V	V

3.3. Álgebra booleana

Boole desarrolló un sistema algebraico a partir de su lógica, en la cual las condiciones lógicas representaban las operaciones posibles y las proposiciones, las variables. Con este sistema, se podían construir sentencias lógicas complejas que se ajustarán a un determinada tabla de verdad, construyendo nuevas condiciones lógicas basadas en las tres básicas.

Por ejemplo, podríamos querer definir una condición lógica similar al **or** pero que sea verdadera sólo si una de las dos proposiciones es verdad y no si ambas lo son (lo que se conoce como un o exclusivo, en inglés como **exclusive or** o **xor**). La tabla de verdad de la condición lógica **xor** es:

A	B	A xor B
F	F	F
F	V	V
V	F	V
V	V	F

A través del álgebra diseñada por Boole, podemos construir una expresión ocupando **and**, **or** y **not** que represente esta tabla de verdad:

$$\text{A } \textbf{xor} \text{ B} = \textbf{not}(\text{A}) \textbf{ and } \text{B } \textbf{or } \text{A } \textbf{and not } (\text{B})$$

4. Máquinas de cálculo basadas en lógica booleana

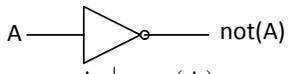
4.1. Compuertas binarias

En 1937, Claude Shannon, ingeniero del MIT, se dio cuenta que el álgebra booleana podía ser utilizada para trabajar con números binarios: bastaba reemplazar todos los valores «Verdadero» por 1s y los «Falso» por 0s, y todas las técnicas y procedimientos del álgebra descrita por Boole podían ser usados también para hacer álgebra con números binarios. El gran aporte de esta idea fue que entregó una herramienta poderosa para diseñar componentes que permitieran realizar cálculos con números binarios.

Shannon definió el concepto de «compuertas binarias» las cuales corresponden a componentes que implementan una de las condiciones lógicas definidas por Boole. Cada compuerta se comporta según las tablas de verdad de Boole, pero en vez de asociar valores de verdad, asocia valores binarios. Las compuertas, de esta manera, representan «cajas negras» que reciben como input valores binarios y entregan un determinado output. A cada compuerta se le asignó un símbolo, de manera de permitir diseñar gráficamente combinaciones de estás.

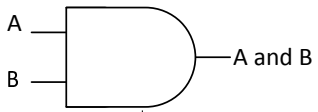
A continuación se muestran la tabla de valores y símbolo de las compuertas básicas:

- Tabla de valores y símbolo de la compuerta **not**:



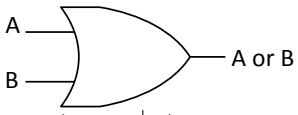
A	not(A)
0	1
1	0

- Tabla de valores y símbolo de la compuerta **and**:



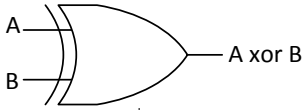
A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

- Tabla de valores y símbolo de la compuerta **or**:



A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

- Tabla de valores y símbolo de la compuerta **xor**:



A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

4.2. Implementación física de compuertas

El gran salto conceptual de Shannon de la lógica booleana a las compuertas binarias no tendría mayor utilidad si no hubiese también definido como implementar de manera física dichas compuertas. Shannon observó que una compuerta **and** se podía representar como un conjunto de dos relé ubicados en serie, dado que estos cumplen con la tabla de valores de la compuerta: habrá corriente en la salida, sólo si ambos relé dejan pasar corriente.

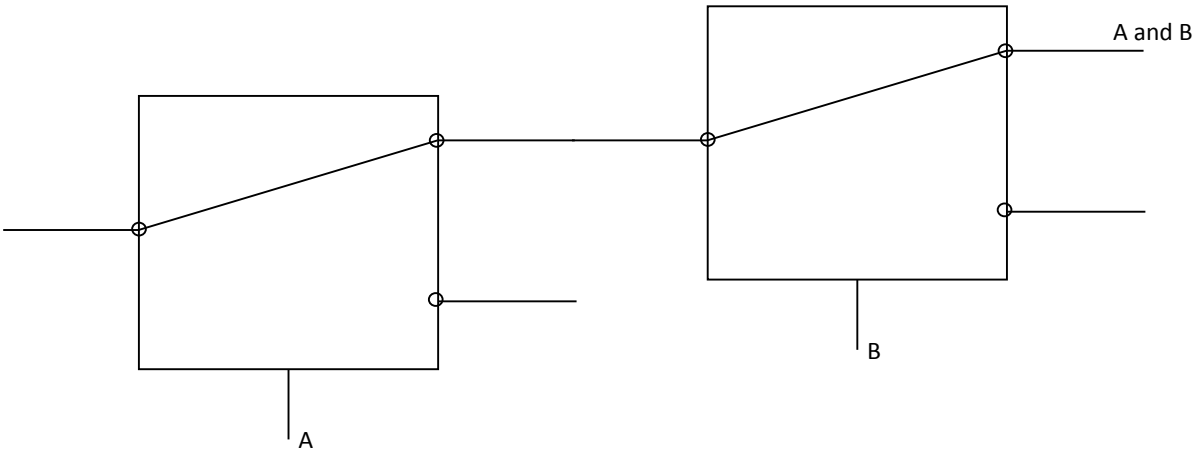


Figura 3: Compuerta **and** construida con relés. Se muestra el caso A=1, B=1.

De manera similar, una compuerta **or** se podía representar como un circuito en paralelo, es decir, como dos relés que funcionan independientes y que se conectan sus salidas, cumpliendo con que la corriente pasa al final, si pasa por cualquiera.

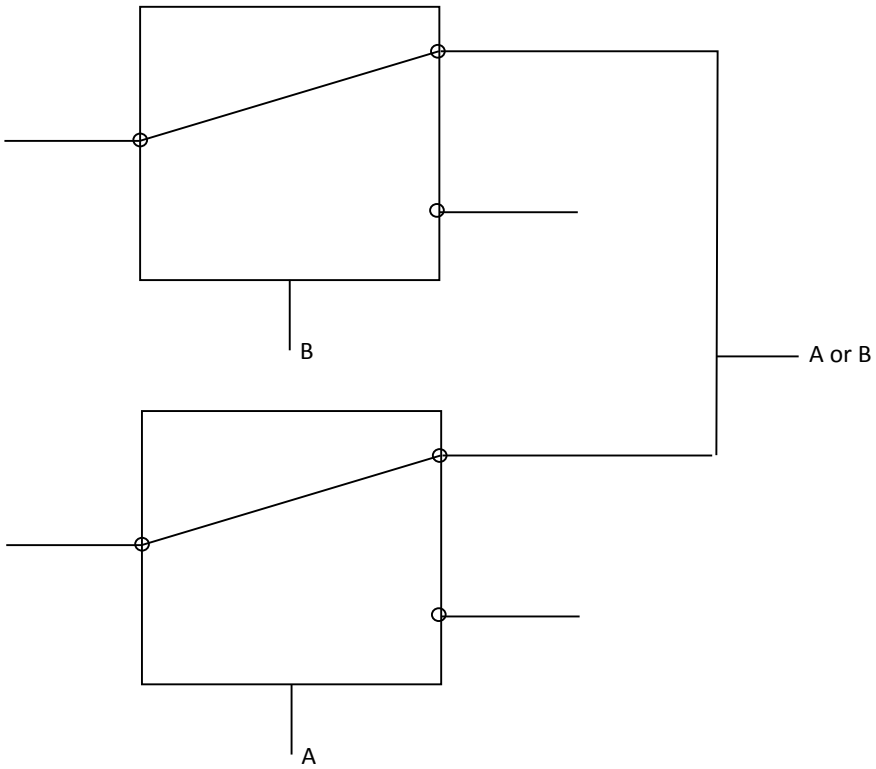


Figura 4: Compuerta **or** construida con relés. Se muestra el caso A=1, B=1.

Por último la compuerta **not** también podía implementarse con relés, básicamente considerando la 2da salida como la negación de la 1era.

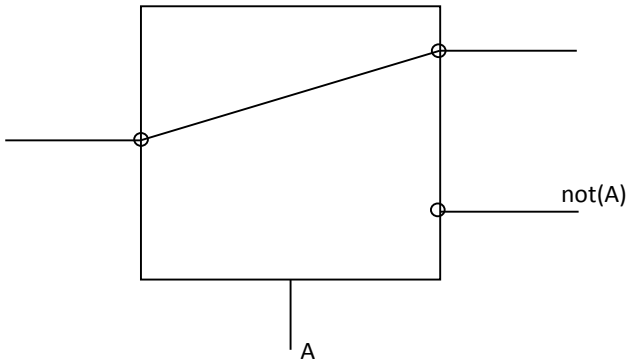


Figura 5: Compuerta **not** construida con relés. Se muestra el caso A=1.

La implementación física de estas tres compuertas nos permite construir sistemas más complejos, dado que al estar ocupando el álgebra de Boole, sabemos que podemos construir el resto de las combinaciones lógicas en base a estas tres compuertas.

4.3. Circuitos binarios

El diseño a través de compuertas binarias sumado a su implementación física, nos entrega todo lo necesario para comenzar a construir circuitos de cálculo numérico. Vamos a comenzar con la operación más sencilla: suma de 1 bit.

4.3.1. Sumador de 1 bit

Para ocupar las técnicas del diseño lógico, tenemos que pensar nuestras operaciones como tablas de verdad. La suma de 1 bit puede interpretarse como la siguiente tabla:

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Observamos que la suma de 1 bit resulta en un número de 2 bits, y por tanto debemos diseñar dos circuitos binarios. Los bits de salida de la suma se denominan habitualmente como S o bit de suma y C o bit de carry. Si observamos cada salida de manera independiente, se observa que el bit de carry tiene la misma tabla de verdad que la compuerta **and** y el bit de suma, la misma tabla que la compuerta **xor**. De esta forma el circuito de un sumador de 1 bit queda de la siguiente manera:

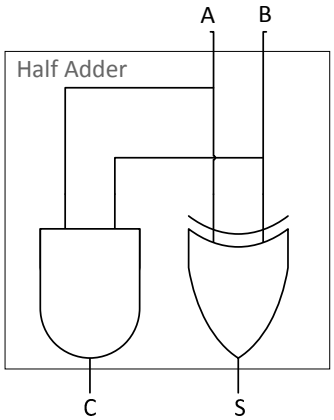


Figura 6: Medio sumador de 1 bit (half-adder)

4.3.2. Sumador de 4 bits

Nos gustaría ahora escalar nuestro circuito para poder realizar operaciones con más bits. Para eso, sin embargo, no nos basta con el circuito sumador antes visto, lo que podemos entender viendo el siguiente ejemplo de suma:

$$\begin{array}{cccccc} & & {\scriptstyle 1} & {\scriptstyle 1} & & \\ & 1 & 0 & 0 & 1 & \\ + & 0 & 0 & 1 & 1 & \\ \hline & 1 & 1 & 0 & 0 & \end{array}$$

El problema está en que para la suma del segundo bit, necesitamos sumar también el acarreo, osea se requieren sumar tres valores, y nuestro circuito previo sólo es capaz de sumar dos valores (por eso habitualmente se le denomina half-adder). Podemos utilizar el circuito anterior para construir nuestro sumador de 3 entradas (conocido como full-adder):

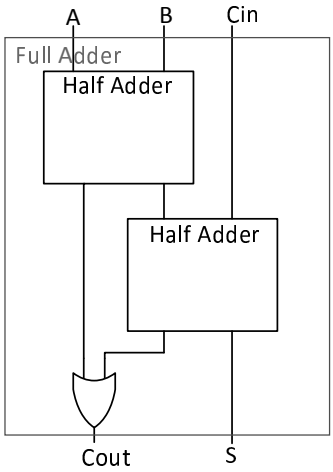


Figura 7: Sumador de 1 bit (full-adder)

El circuito completo del sumador de 4 bits consiste en simplemente unir 4 full-adders, conectando la salida de acarreo de cada uno, con la entrada de acarrero del siguiente, resultando en el siguiente circuito:

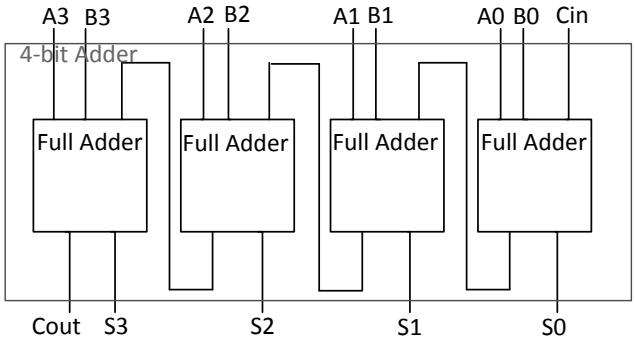


Figura 8: Sumador de 4 bits

4.3.3. Restador de 4 bits

El siguiente nivel de dificultad en operaciones está en la resta, para la cual debemos primero convertir el restando a complemento a dos, y luego hacer la suma normal. Con el circuito anterior ya tenemos lista esta última etapa, nos faltan la conversión a complemento a 2.

El algoritmo de conversión a complemento a 2 tenía dos pasos:

- Invertir los bits
- Sumar 1

Para invertir los bits, podemos ocupar la compuerta **not** que justamente realiza esto. Para la suma de 1, podríamos ocupar otro sumador más, pero una mejor solución es usar el mismo sumador que ya tenemos, e insertar el 1 como carry in para el primer bit. Con esto nos resulta el siguiente circuito:

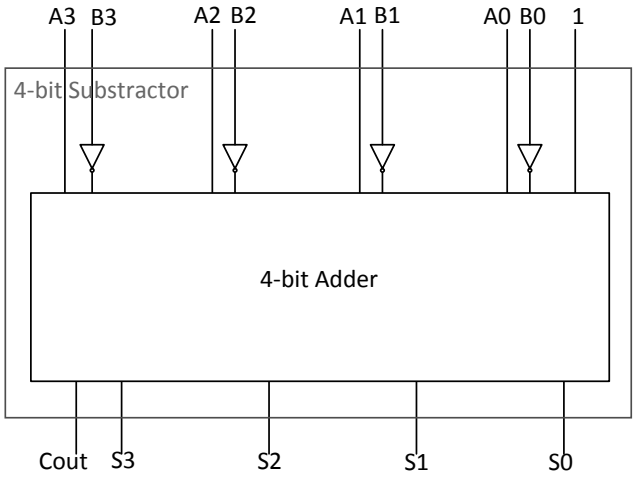


Figura 9: Restador de 4 bits

Podemos observar que ocupando las técnicas de la lógica booleana aplicadas a número binarios representados por circuitos eléctricos somos capaces de construir máquinas que realicen operaciones numéricas. Esta combinación de lógica con circuitos es el fundamento básico del diseño de los principales componentes de los computadores modernos. La única diferencia respecto a lo estudiado ahora, es que en vez de ocupar relés se utilizan transistores, que son componentes eléctricos conceptualmente equivalentes, pero de mucho menor tamaño y mayor velocidad.

5. Combinación de circuitos de operación

Los circuitos del sumador y restador de 4 bits vistos previamente representan dos ejemplos de máquinas de cálculo que pueden diseñarse para realizar operaciones dentro de un computador. Para permitir combinar estas distintas máquinas se requieren mecanismos especiales de control, y también abstracciones que permitan diseñar de manera más sencilla estas combinaciones.

5.1. Abstracción de componentes

Como se vio anteriormente, la base del diseño de componentes basados en compuertas binarias está en la modularización y abstracción. Para el sumador de 4 bits, por ejemplo, primero diseñamos el circuito del *Half-Adder* a nivel de compuertas, luego el *Full-Adder* como una combinación de *Half-Adders* y finalmente el sumador de 4 bits como un conjunto de *Full-Adders*. De esta manera pudimos ir aumentando el nivel de abstracción, lo que nos permite trabajar con módulos de más alto nivel que las compuertas básicas (por ejemplo el sumador de 4 bits) bastándonos saber que entradas y salidas tiene y que operación está realizando.

Otro elemento que permite abstraer el diseño de estos componentes es el concepto de **bus**. Un **bus** se puede definir como un conjunto de cables o líneas de corriente que agrupados representan un valor o unidad. Por ejemplo en el caso de los circuitos de sumador y restador antes vistos, podemos agrupar los 4 bits del operando A, los 4 bits del operando B y los bits del resultado S en tres **buses** de 4bits.

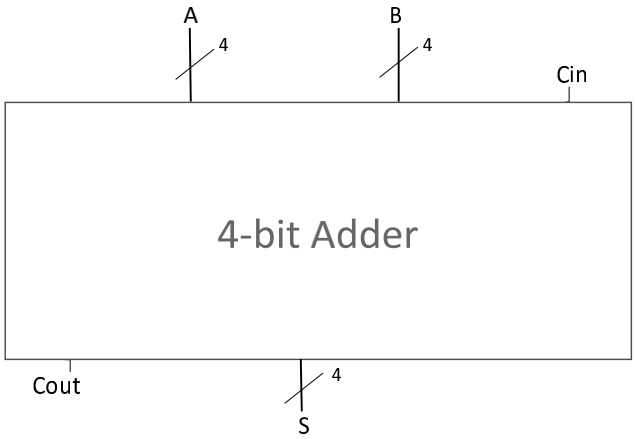


Figura 10: Sumador de 4 bits con notación de buses

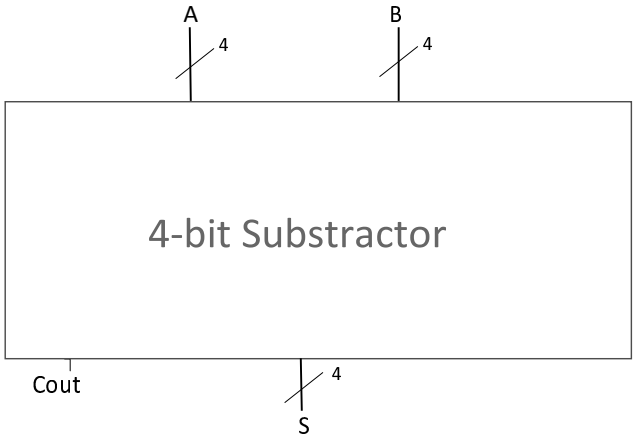


Figura 11: Restador de 4 bits con notación de buses

El concepto de buses nos entrega una notación que simplifica el diseño de estos circuitos, y nos permite trabajar con un nivel de abstracción mayor, conceptualizando las operaciones directamente sobre los operandos completos (como A y B en los casos anteriores) en vez de operaciones unitarias sobre bits. Hay que señalar, sin embargo, que también podemos hablar de buses de 1 bit, en los casos que corresponda, de manera de generalizar toda señal que interactúe con circuitos como buses.

Un aspecto interesante de la notación de buses es que nos permite escalar los circuitos a operaciones de mayor precisión, por ejemplo 8 o 32 bits, si tener que realizar modificaciones mayores **en el componente dibujado**. Es evidente que por debajo de la abstracción si será necesario realizar modificaciones para que dicho circuito funcione: en el caso del sumador de 8 bits necesitamos 8 Full-Adders conectados.

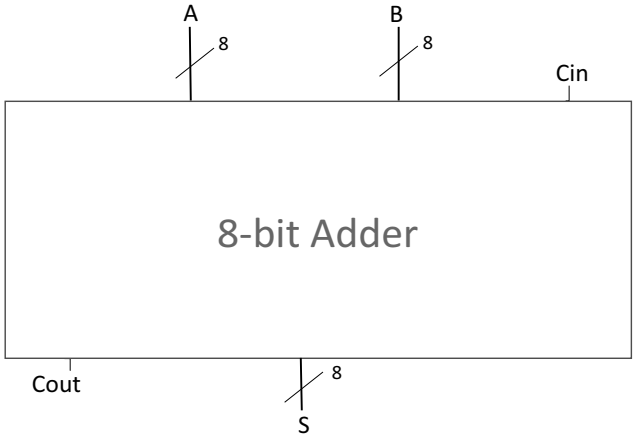


Figura 12: Sumador de 8 bits con notación de buses

5.2. Circuitos de control

Una vez definida las abstracciones de los componentes con las que trabajaremos, debemos agregar mecanismos que permitan controlar y combinar estos distintos componentes de cálculo para construir una unidad funcional de cálculo para el computador.

5.2.1. Enabler

Una primera funcionalidad que nos interesa agregar a nuestras máquinas de cálculo es la capacidad de controlar su salida. Los circuitos que hemos diseñado hasta ahora están siempre entregando en su salida el resultado, es decir el flujo de datos no está siendo controlado. En algunas circunstancias podría ser relevante tener el control para **habilitar** o **deshabilitar** que la salida del componente sea el resultado.

En particular nos interesaría que dado un bit de datos $A0$ podamos a través de otro valor E controlar si la salida $A0'$ ´tendrá el valor de $A0$, es decir, «se deja pasar el flujo», o tendrá un valor 0, es decir, «se corta el flujo», lo que podemos representar con la siguiente tabla:

E	A0’
0	0
1	A0

Una compuerta que cumple exactamente con lo anterior es la compuerta **and** la cual en caso de tener una de sus entradas en 0 tendrá siempre salida 0 y en caso de tener una de sus entradas en 1 tendrá como salida el valor de la segunda entrada. Si consideramos como una de las entradas de la compuerta a E y a la otra como $A0$ tenemos un circuito que cumple exactamente con la tabla anterior, el cual se denomina *Enabler* o *Habilitador* ya que permite o habilitar o deshabilitar el flujo de datos.

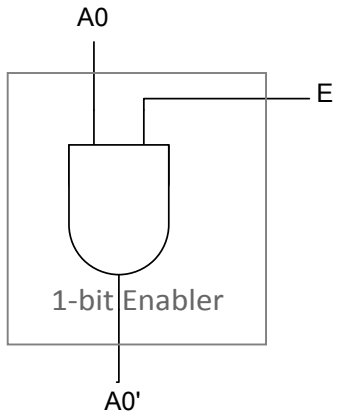


Figura 13: Enabler de 1 bit

Podemos extender este circuito a datos con mayor número de bits. Lo interesante está en que mantenemos una sola señal E para controlar la habilitación y deshabilitación.

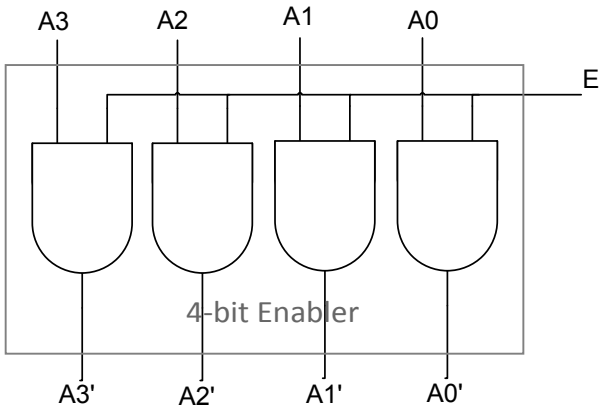


Figura 14: Enabler de 4 bits

5.2.2. Multiplexor

El circuito del enabler nos permite regular el flujo de datos de una máquina de cálculo. Sin embargo no nos ayuda en el objetivo de tener múltiples operaciones simultáneamente. Necesitamos otro circuito que nos permita controlar que operación vamos a realizar, de manera de que dada una salida común podamos obtener mediante esta tanto la suma o resta de dos operandos, dependiendo de un valor de control.

Veamos el problema primero a nivel de bits: supongamos que tenemos dos datos $A0$ y $B0$ y una salida común $M0$. Nos gustaría poder controlar que dato sale dependiendo de una señal de selección S que cuando sea 0 entregue el valor de $A0$ y cuando sea 1 entregue el valor de $B0$, lo que se representa en la siguiente tabla:

S	M0
0	A0
1	B0

Para construir el circuito podemos utilizar el circuito del enabler (es decir la compuerta and) de la siguiente forma: podemos ocupar la señal S como señal de habilitación/deshabilitación de manera de que cuando sea 1 habilite una de las dos salidas, por ejemplo $B0$ y **simultáneamente** deshabilite $A0$. De manera equivalente con un valor 0 habilitará $A0$ y desahbilitará $B0$. Con esto nos aseguramos que dado un valor de S estamos obteniendo en las dos salidas siempre un valor y un 0. Si combinamos estas dos salidas mediante una compuerta **or** obtendremos como única salida el valor que hayamos seleccionado, circuito que se denomina *Multiplexor* o *Mux*.

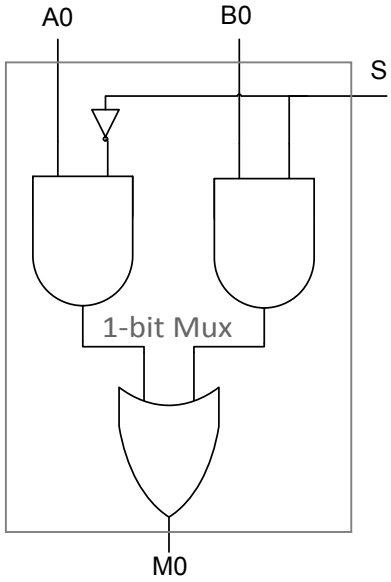


Figura 15: Mulitplexor de 2 entradas de 1 bit de datos

Al igual que en el enabler podemos escalar en el número de datos, usando varios multiplexores de 1 bit para obtener, por ejemplo, un multiplexor que elige entre dos valores de 4 bits.

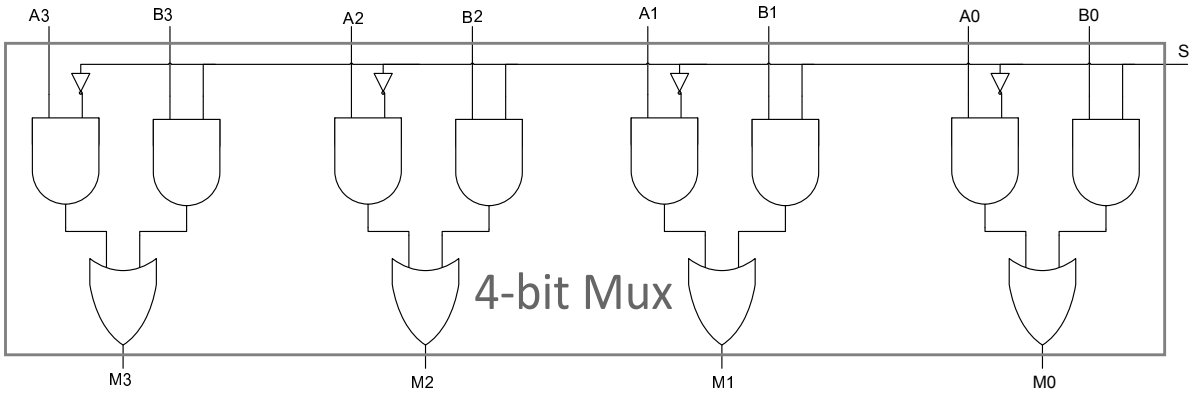


Figura 16: Mulitplexor de 2 entradas de 4 bit de datos

Podemos extender aún más la noción de multiplexor, pensando por ejemplo en un caso de que existan 4 entradas distintas que queramos seleccionar. La opción más simple sería escalonar 4 multiplexores, de manera que con 4 bits de control podemos controlar las 4 entradas. Podemos, sin embargo, optimizar este circuito de manera de ocupar sólo 2 bits de control: dado que 2 bits son capaces de representar 4 números ($0 = 00$, $1 = 01$, $2 = 10$, $3 = 11$), podemos diseñar un circuito que dependiendo de la combinación de los dos bits de control entregue una de las cuatro entradas, como se observa en la siguiente tabla:

S1	S1	M
0	0	A
0	1	B
1	0	C
1	1	D

La implementación específica de este circuito no es relevante. Sólo nos interesa que existe como componente, y podemos trabajar con este abstraiéndonos de los detalles de su implementación:

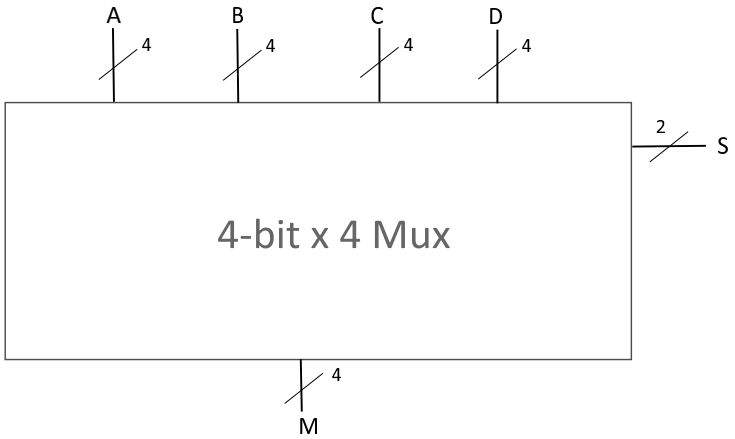


Figura 17: Mulitplexor de 4 entradas de 4 bit de datos

5.3. Buses de datos y control

Hay un elemento importante que surge como consecuencia del diseño de estos componentes. Los circuitos previamente estudiados (sumador y restador) los buses contenían los operandos y resultados, es decir eran **buses de datos**. En el caso del circuito enabler y multiplexor, en cambio, existen dos tipo de buses. Por una parte están los buses de datos, que transportan la información que sera habilitada/deshabilitada o seleccionada, según el circuito que se utilice. Sin embargo, además se cuentan con las señales de control o **buses de control**, como el bit E en el enabler o los bits de selección en el mux. La importancia de esta diferencia es que para ambos buses estamos ocupando la misma representación numérica (números binarios) y física, por lo que, como veremos más adelante, podemos combinar la información de los distintos buses si lo necesitamos.

6. Unidades de ejecución

Ahora que conocemos los circuitos de control, podemos combinar las distintas máquinas de cálculo para armar una unidad de cálculo o **unidad de ejecución**. Nos enfocaremos en diseñar la unidad más básica de un computador, denominada **unidad aritmética y lógica** o **ALU** por sus siglas en inglés. Como su nombre lo señala, esta unidad permite realizar operaciones aritméticas básicas (suma y resta) y también operaciones lógicas simples (and, or, not, etc). Algunos computadores tienen otras unidades de ejecución además de la ALU, por ejemplo una unidad aritmética para números de punto flotante o **FPU**, pero por ahora nos enfocaremos sólo en la ALU.

6.1. Sumador/Restador

El primer paso para completar la ALU es armar la unidad aritmética, que consta de un sumador y un restador. Ocupando el multiplexor de 4 bits de datos definido previamente podemos diseñar un sumador/restador de una salida M, y con una señal de control S que cumpla con:

S	M
0	Suma
1	Resta

Combinando los circuitos de sumador y restador con el multiplexor obtenemos un circuito sumador/restador:

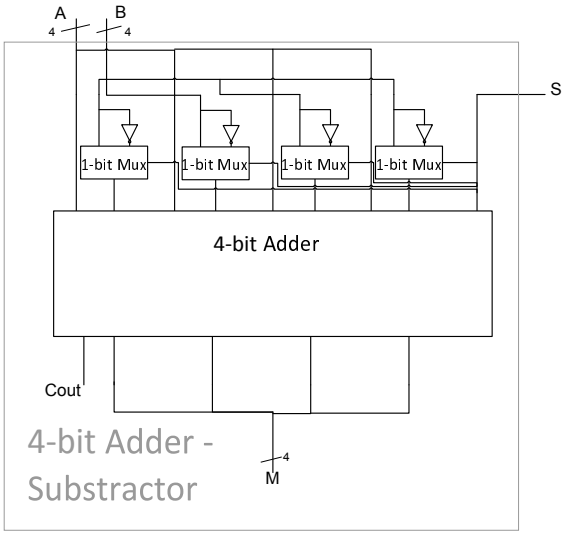


Figura 18: Sumador-Restador de 4 bits de datos

Este circuito cumple con permitir sumar y restar dos valores, entregando el resultado por la misma salida. Existe, sin embargo, una optimización que podemos hacer y tiene relación con el hecho de que el restador ocupa internamente un sumador, por lo que este circuito sumador/restador necesita de dos circuitos sumadores. Una forma en que se optimiza este circuito, es reutilizando un único sumador para ambas operaciones y agregando multiplexores a las entradas para seleccionar si entra directamente el segundo operando o su negación (para el caso de la resta). Este circuito es un ejemplo de lo que mencionamos previamente respecto a los buses de datos y control: en este caso el bus de control de selección se utiliza como dato: se agrega como carry in al sumador, de manera de que cuando tiene el valor 1, lo que representa una resta, se le suma 1 para completar la conversión a complemento a 2 del segundo valor.

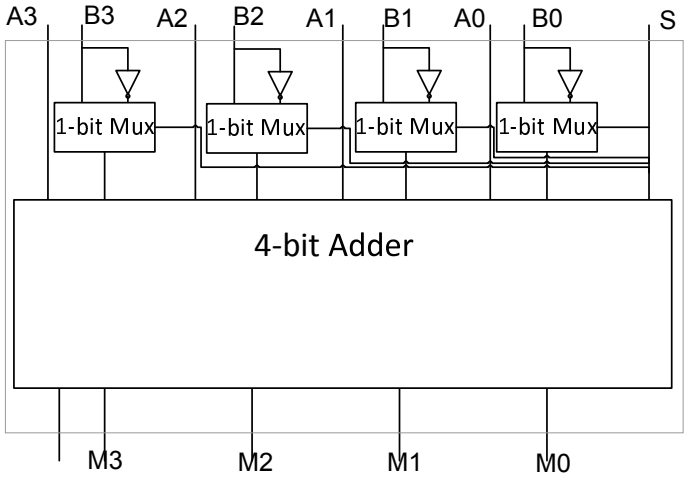


Figura 19: Sumador-Restador de 4 bits de datos optimizado

6.2. Operaciones lógicas

La ALU además de permitir operaciones aritméticas, permite operaciones lógicas entre dos valores como **and**, **or** y **xor** y operaciones lógicas sobre un valor como **not**.

Las operaciones lógicas entre números de varios bits se interpretan como operaciones bit a bit. Por ejemplo un **and** entre los valores 1001 y 0011 sería:

	1	0	0	1
and	0	0	1	1
	0	0	0	1

Un **or** entre los valores 1001 y 0011 sería:

	1	0	0	1
or	0	0	1	1
	1	0	1	1

Un **xor** entre los valores 1001 y 0011 sería:

	1	0	0	1
xor	0	0	1	1
	1	0	1	0

Por último un **not** del valor 0011 sería:

not	0	0	1	1
	1	1	0	0

Podemos ocupar la notación de buses para representar estas operaciones con los símbolos de las compuertas:

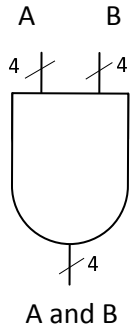


Figura 20: AND de 4 bit de datos

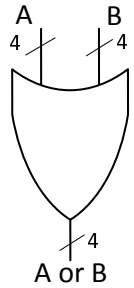


Figura 21: OR de 4 bit de datos

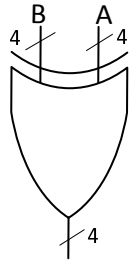


Figura 22: OR de 4 bit de datos

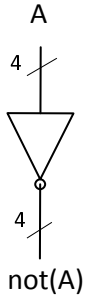


Figura 23: NOT de 4 bit de datos

6.3. Operaciones de desplazamiento o shifting

Otra operación relevante que es posible implementar con compuertas lógicas es la operación de desplazamiento o shifting. Esta operación permite, dado un cierto número binario, «desplazarlo» hacia la izquierda o la derecha. Por ejemplo si se tiene almacenado el número de 4 bits 0101, al hacer un *shift* o desplazamiento a la izquierda obtenemos el número 1010, es decir se movieron los bits una posición a la izquierda. Si hubiésemos hecho un *shift* a la derecha, habríamos obtenido el número 0010.

La relevancia de esta operación tiene que ver con la interpretación aritmética de la operación de shift. Un shift a la izquierda puede ser interpretado como una multiplicación por 2; Un shift a la derecha puede ser interpretado como una división por 2. Está característica hace que este circuito sea muchas veces agregado junto a la ALU como una unidad de ejecución adicional denominada **shifter**, permitiendo hacer shifts a la izquierda y a la derecha.

6.4. Unidad aritmética y lógica: ALU

Ya con todas las operaciones de la ALU definidas, nos interesa diseñar el componente completo. Dado que las operaciones que queremos agregar son 8: 2 operaciones aritméticas (suma y resta), 4 operaciones lógicas (and, or, xor, y not) y 2 operaciones de desplazamiento (shift left y shift right), necesitamos un multiplexor de 8 entradas y 3 señales de control, que cumpla con la siguiente tabla:

S2	S1	S0	M
0	0	0	Suma
0	0	1	Resta
0	1	0	And
0	1	1	Or
1	0	0	Not
1	0	1	Xor
1	1	0	Shift left
1	1	1	Shift right

El componente completo de la ALU lo podemos representar con la abstracción de la figura que aparece a continuación. Es importante destacar que para estos componentes ya no nos basta con el «dibujo» y el nombre del componente para saber la funcionalidad (como si era el caso del sumador). Acá necesitamos tener también la información de la tabla anterior, indicando que operaciones se realizan con que señales de control.

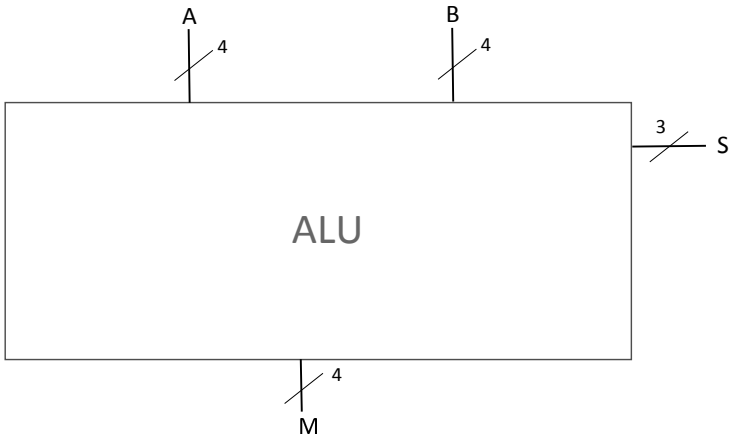


Figura 23: ALU de 4 bit de datos y 5 operaciones

7. Ejercicios

- Implemente un circuito 2 bit Multiplier, que realice la multiplicación entre dos valores de 2 bits.
- Escriba un programa que simule el funcionamiento de un multiplexor de 2 entradas 4 bits, ocupando variables boolean para representar bits, y los operadores lógicos &, —, $\wedge$ y para representar compuertas lógicas.

8. Referencias

- Shannon, C.; A Symbolic Analysis of Relays and Switching Circuits, MIT Press, 1937.

9. Apéndice: Operaciones Bitwise (lógicas y shifts) en lenguajes de programación

Las operaciones lógicas bit a bit **and**, **or**, **xor** y **not**, así como las operaciones de desplazamiento o **shift**, pueden ser ocupadas en los lenguajes de programación tradicionales como C, C++, Java y C#. A continuación se presentan los símbolos habitualmente usados:

Sean las variables `a`, `b` y `n`:

- la operación `a and b` se escribe como:
  `a & b`;
- la operación `a or b` se escribe como:
  `a | b`;
- la operación `a xor b` se escribe como:
  `a` $\wedge$ `b`;
- la operación not `a` se escribe como:

  ```
  ~ a
  ```

- la operación `shift left a en n bits` se escribe como:
  `a` $<<$ `n`;
- la operación `shift right a en n bits` se escribe como:
  `a` $>>$ `n`;

Las operaciones bitwise tienen la ventaja de permitir modificar valores en bits específicos de un determinado número, debido a esto permiten implementar una serie de funcionalidades útiles:

- **Máscaras**: La idea de una máscara numérica es similar al de una máscara física, en este caso una máscara numérica corresponde a un valor que al ser «superpuesto» en otro valor, «deja ver» solamente parte del número original. Esto es importante en casos que las distintas partes de un número representen distinta información. Un ejemplo práctico son las máscaras de subred, usadas en las configuraciones de red de los computadores. La utilidad en este caso es que dado un IP en un computador (por ejemplo 10.10.10.101), una parte de este IP se puede ocupar para enviar un mensaje a la subred completa de computadores en la cual se encuentra el computador. Una subred posible es la 10.10.10.0, otra la 10.10.0.0. En general el IP de subred se puede obtener al aplicar una máscara al IP, y dependiendo de la máscara se obtendrá la subred requerida.´ La operación de máscara numérica corresponde a realizar un and bit a bit entre la máscara y el valor a enmascarar. En el ejemplo anterior, una posible máscara de subred es 255.255.255.0, que en binario corresponde a una secuencia de 24 unos seguidas de 8 ceros (cada valor entre puntos de un IP es un número de 8 bits). Al aplicar esa máscara al IP 10.10.10.101, los primeros 24 bits se conservan, dado que un and con 1 conserva el valor. Los 8 bits finales en cambio se pasan a 0, obteniendo 10.10.10.0, que sería en este caso el IP de subred.

  A continuación se presenta un ejemplo en Java de la operación de enmascaramiento, ocupando representación hexadecimal para los números, lo que permite ver más fácilmente el resultado del enmascaramiento:

```
static void mascaras()
{
  int num1 = 0xF0;

  int num2 = 0xA1;

  int num3 = num1 & num2;

  System.out.println(String.format("%x", num3));
}
```

- **Flags**: Otra utilidad práctica de las operaciones bitwise es la posibilidad de guardar múltiple información en un solo número binario. Un ejemplo de esto son los flags, valores que pueden estar activos o no activos indicando con eso alguna propiedad. El sistema de permisos de archivos de Linux, por ejemplo, define tres posibles permisos: Lectura (Read), Escritura (Write), Ejecución (Execute). Para almacenar la información de si los flags están activos o no necesitamos solamente 1 bit por flag, por lo cual podemos ocupar un número binario de 3 bits para codificar esta información.

  Para ocupar flags de 1 bit son necesarias dos funcionalidades: poder setear si cada flag está activo o no, **de manera independiente**, y poder obtener la información del estado de cada flag. Para realizar lo primero, si se quiere setear como activo el flag, basta con hacer un or con una máscara que tenga un uno el bit en la posición del flag, y ceros en el resto de los bits. Con esto, al hacer or con ceros no se pierden los estados de los otros flags, y se modifica a uno el bit correspondiente. En el caso de necesitar setear el flag en 0, basta con hacer un and con una máscara que tenga unos en todos los bits, menos en el que se quiere dejar en 0.

  Para poder obtener información de cada flag, se requiere ocupar las mismas máscaras de seteo de flag activo, pero esta vez se aplica un and con el valor. Al hacer esto, el número será distinto de cero, solo si el bit del cual se quiere tener información estaba en cero, con lo cual se puede revisar esa condición para saber el estado.

  A continuación se muestran ejemplos de código que implementa el seteo y obtención de valores de flags.

```
static void setFlagsOn()
{
  int permisos = 0x0;

  int mascara0 = 0x1;
  int mascara1 = 0x2;
  int mascara2 = 0x4;

  permisos = permisos | mascara0 | mascara1 | mascara2;

  System.out.println(permisos);


}

static void setFlagsOff()
{
  int permisos = 0x7;

  int mascara0 = 0x6;
  int mascara1 = 0x5;
  int mascara2 = 0x3;

  permisos = permisos & mascara0 & mascara1 & mascara2;

  System.out.println(permisos);
```

```
}

static void getFlags()
{
  int permisos = 0x7;

  int mascara0 = 0x1;
  int mascara1 = 0x2;
  int mascara2 = 0x4;

  int get0 = permisos & mascara0;

  if(get0 == 0)
    System.out.println("No tiene permiso 0");
  else
    System.out.println("Tiene permiso 0");

  int get1 = permisos & mascara1;

  if(get1 == 0)
    System.out.println("No tiene permiso 1");
  else
    System.out.println("Tiene permiso 1");

  int get2 = permisos & mascara2;

  if(get2 == 0)
    System.out.println("No tiene permiso 2");
  else
    System.out.println("Tiene permiso 2");

}
```

- **Xor Clear**: El xor también es un operando bitwise que permite realizar funcionalidades útiles. Un ejemplo que se muestra continuación es que un número aplicado xor con sigo mismo resulta siempre en 0, por lo cual se puede esta operación como un «clear» para setear en 0 un cierto número:

```
static void xorClear()
{
  int num1 = 0x7;

  int num2 = num1 ^ num1;

  System.out.println(String.format("%x", num2));
}
```

- **División y Multiplicación por potencias de 2**:

  Las operaciones de desplazamiento tienen una importante utilidad, dado que un shift a la derecha de un número binario, por ejemplo $(1010)_2 = 10$ resulta en $(0101)_2 = 5$ el cual corresponde al número original dividido en 2. De manera similar, un shift a la izquierda corresponde a multiplicar por 2, en el ejemplo: $(10100)_2 = 20$. El hecho de que ocurra esto es evidente, si pensamos que en base de 10, desplazar un dígito a la derecha o izquierda se interpreta también como dividir o multiplicar por 10. En el caso de un número binario, como la base es 2, las operaciones que se logran son multiplicar y dividir por 2.

  El siguiente código muestra la operación de shift en Java:

```
static void shift()
{
  int num1 = 10;

  int num2 = num1 >> 1;

  int num3 = num1 << 1;

  System.out.println(num2);
  System.out.println(num3);
}
```

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2343 Arquitectura de Computadores

Almacenamiento de datos

1. Motivación

Una de las funciones básicas de un computador, además de realizar operaciones, es poder almacenar información. Existen distintas tecnologías para almacenar información que el computador puede ocupar, las cuales presentan distintas características que las hacen útiles para distintas funciones en el computador.

2. Variables y Arreglos

2.1. Variables

La unidad básica de almacenamiento de un programa se conoce como **variable**. Una variable corresponde a un contenedor que puede almacenar un valor que puede variar en el transcurso del programa (por eso su nombre). Las variables tienen tres características que las determinan: un **tipo de dato** que indica que información está almacenando; un **valor** que representa la información almacenada en un momento dado; y un **identificador** que se ocupa para reconocer a una determinada variable y diferenciarla de otras. Para poder representar variables de manera física en una máquina, por lo tanto, es necesario proveer estas tres características.

Como se ha visto anteriormente, el computador almacenará toda su información como números binarios (i.e. secuencias de bits). De esta forma, todas las variables se almacenarán finalmente como una secuencia de bits. La interpretación que se le de a esa secuencia de bits, sin embargo, variará, dependiendo del tipo de dato. De esta forma el valor de una variable es función tanto de la secuencia de bits como del tipo de dato. **Sin conocer el tipo de dato es imposible interpretar el valor de una determinada secuencia de bits.**

Los tipos de datos van a estar determinados por tres características principales: la **cantidad de bits** ocupados, y la **codificación** de la secuencia de bits y la **interpretación** que se le de a una codificación particular. La cantidad de bits determinará el rango de valores disponibles de un determinado tipo, es decir los valores máximo y mínimo que se pueden almacenar en un determinado tipo. La codificación determinara que está almacenando (la sintáxis del tipo) y la interpretación indicará que representa esa codificación particular para este tipo (la semántica del tipo).

Para entender mejor los diferentes elementos involucrados en un tipo de dato, analizaremos los tipos de datos básicos del lenguaje Java:

- Tipo `int`
  - Codificación: Número en base 2 con signo, ocupando representación de complemento a 2.
  - Interpretación: Número entero positivo o negativo.

    - Cantidad de bits: 32 bits (4 bytes), lo que permite un rango de valores entre los números -2147483648 y 2147483647.

- Tipo `char`
    - Codificación: Número en base 2 sin signo.
    - Interpretación: Caracter o letra obtenida del estándar Unicode.
    - Cantidad de bits: 16 bits (2 bytes), lo que permite un rango de valores entre los caracteres asociados a los números 0 y 65535.

- Tipo `byte`
    - Codificación: Número en base 2 con signo, ocupando representación de complemento a 2.
    - Interpretación: Número entero positivo o negativo.
    - Cantidad de bits: 8 bits (1 byte), lo que permite un rango de valores entre los números -128 y 127.

- Tipo `boolean`
    - Codificación: Número en base 2 sin signo.
    - Interpretación: Valor lógico o de verdad, pudiendo solo ser verdadero (1) o falso (0).
    - Cantidad de bits: 8 bits (1 byte), lo que permite potencialmente un rango de valores entre los números 0 y 255, aunque solo se permite ocupar los valores 0 y 1, asociados a los valores `true` y `false`.

- Tipo `float`
    - Codificación: Single precission floating point, según especificado por el estándar IEEE754.
    - Interpretación: Número fraccional positivo o negativo, además de algunos símbolos especiales (+Infinito, -Infinito, NaN).
    - Cantidad de bits: 32 bits (4 bytes), lo que permite un rango aproximado de valores entre los números $\pm 1{,}5 \times 10^{-45}$ y $\pm 3{,}4 \times 10^{38}$.

- Tipo `double`
    - Codificación: Double precission floating point, según especificado por el estándar IEEE754.
    - Interpretación: Número fraccional positivo o negativo, además de algunos símbolos especiales (+Infinito, -Infinito, NaN).
    - Cantidad de bits: 64 bits (8 bytes), lo que permite un rango aproximado de valores entre los números $\pm 5 \times 10^{-324}$ y $\pm 1{,}7 \times 10^{308}$.

2.2. Arreglos

Una segunda forma de almacenamiento que es relevante en un programa son los **arreglos**. Un arreglo se define como un conjunto de datos agrupados por un identificador único, y que permite acceder de manera indexada a una cierto dato para leer o modificar su valor. Los arreglos tienen cuatro características que las determinan: un **tipo de dato** que indica que información está almacenando en todos los datos; un **conjunto de valores** distintos que representa la información almacenada en un momento dado; un **identificador** que se ocupa para reconocer el arreglo y poder indexarlo para acceder a un valor; y un **largo** que indica cuantos valores se tienen almacenados.

Para poder representar arreglos de manera física en una máquina, por lo tanto, es necesario proveer estas tres características, lo que implica que además de los detalles descritos en la sección previa, para almacenar arreglos se requiere la capacidad de almacenar múltiples datos de manera ordenada.

3. Tecnologías de almacenamiento de números binarios

Una de las principales ventajas de usar números binarios es que su almacenamiento se puede realizar usando dos símbolos o estados por cada dígito o **bit**. Debido a esto existen diversas tecnologías que permiten almacenar números binarios, las cuales presentan distintas características.

3.1. Características

Los distintos tipos de almacenamiento se diferencia por una serie de características. A continuación se listan las más relevantes.

Tecnología

La primera característica diferenciadora es la tecnología que se ocupa para almacenar la información. Aunque en la actualidad existen una multitud de formas de guardar número binarios, son tres las tecnologías predominantes:

- Magnética: usada en los discos duros, se basa en modificar el estado magnético de una parte de una superficie para representar un 1. Las zonas no modificadas representa un 0. Para poder escribir y leer en esta superficie se utilizan cabezales especiales que son capaces de moverse en la posición donde se almacena la información y modificar el estado magnético para escribir o interpretar el estado para leer.
- Óptica: usada en los CD, DVD y discos Blu-ray, se basa en modificar la capacidad de reflectancia óptica de una sector superficie, de manera de que cuando se ilumine ocurra o no reflexión lo que se interpreta como un 1 o 0 lógico. Para leer se utilizan lasers de precisión que apuntan a una zona específica y contienen sensores que interpretan la reflectancia de la zona. Para escribir, se utilizan lasers más poderosos que son capaces de modificar las propiedades de reflectancia.
- Eléctrica: usada en las memorias flash, y otros tipos de memoria, se basa en almacenar la información usando componentes eléctricos, como transistores o condensadores que pueden ser leídos o escritos con señales eléctricas. Respecto a las anteriores presenta la ventaja de no requerir partes móviles para acceder a la información, ya que lo que se «mueve» es la corriente eléctrica.

Mutabilidad

Otra característica relevante es la «mutabilidad» o «posibilidad de cambiar». En dispositivos como los discos duros o memorias flash es posible tanto leer información como modificar información, es decir permiten **escritura y lectura**. En dispositivos como los CD y DVD no regrabables, sólo se puede obtener la información y no modificar, es decir son de **sólo lectura**.

Capacidad y Costo

La capacidad y el costo están directamente relacionado a la tecnología usada, y en general se cumple la regla de que a mayor capacidad, mayor costo. La razón $\frac{capacidad}{costo}$ es un índice relevante que indica para una tecnología cuanto cuesta almacenar una cierta cantidad de información lo que será un factor importante para determinar que tipo de almacenamiento ocupar en las distintas partes del computador.

Volatilidad

Los dispositivos habitualmente usados para almacenar nuestra información tienen la característica de ser **no volátiles**, es decir, si no estamos entregándoles alimentación eléctrica (i.e. no están enchufados, ni tiene baterías), son capaces de mantener la información guardada. Es evidente que esto es de suma relevancia, ya que en caso contrario podría ocurrir, por ejemplo, que al apagar el computador perderíamos toda nuestra información del disco duro, lo que no tendría mucho sentido práctico.

Existen, sin embargo, dispositivos de almacenamiento que no tienen esta capacidad, es decir, son **volátiles** y si pierden alimentación eléctrica, pierden la información. La volatilidad no presenta ninguna ventaja comparativa, pero ocurre que las tecnologías más rápidas de almacenamiento suelen tener esta característica, por lo que a pesar de contar con esta desventaja, son usadas.

Rendimiento

Una última característica relevante tiene relación con el rendimiento del dispositivo. Hay dos elementos principales para evaluar el rendimiento: la **latencia** que se refiere el tiempo en que se demora el acceso a una particular pieza de información almacenada, la cual se mide en segundos o nanosegundos, y el **throughput** que representa la cantidad de información que se puede sacar del dispositivo en un cierto tiempo, lo que se mide en $\frac{bytes}{segundos}$.

Como regla general los dispositivos de almacenamiento eléctrico tendrán menor latencia y mayor throughput, ya que no requieren estar moviendo partes mecánicas (como si ocurre en el almacenamiento magnético y óptico).

4. Circuitos de almacenamiento

Al considerar que dispositivo de almacenamiento usar como medio principal en un computador la principal característica a tomar en cuenta es el rendimiento. Los computadores realizan operaciones a altas velocidades y por tanto queremos que los dispositivos de almacenamiento ocupados para para estas operaciones sean también rápidos.

Como se señaló previamente, los dispositivos de almacenamiento eléctrico son los que presentan un mejor rendimiento, por lo que estos serán los elegidos para el almacenamiento principal del computador. A continuación se explicará como se pueden construir estos dispositivos de almacenamiento ocupando las mismas herramientas de diseño de circuitos basados en compuertas.

4.1. Flip-flops

El componente básico para almacenar información usando circuitos eléctricos se denomina flip-flop. A diferencia de los circuitos lógicos vistos previamente la salida de un flip-flop depende no sólo de las compuertas que lo componente, sino de su estado previo, lo que se denomina como circuito secuencial.

Existen distintos tipos de Flip-flops, que presentan sutiles variaciones en su diseño. Uno de esos diseños corresponde al flip-flop D con señal de control, que se observa a continuación:

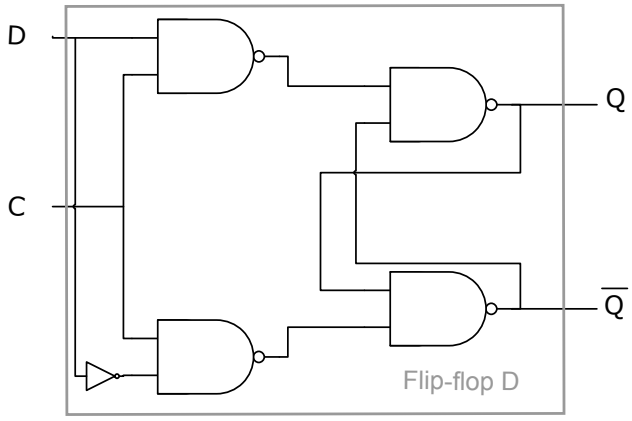


Figura 3: Flip-flop D con control

El funcionamiento de este circuito es el siguiente: Si en la entrada de datos D llega un determinado valor y además la señal de control C está activa, el valor de salida Q de flip-flop se actualiza con el valor de D. Si en cambio, la señal del control C está desactiva, el valor de Q **es el mismo que tenía previamente**.

La tabla de valores del flip-flop D se observa a continuación:

C	D	Q
0	x	Q
1	0	0
1	1	1

Lo relevante de este circuito es que **tiene la capacidad de almacenar 1 bit**, dado que mientras la señal de control sea cero, se mantiene el valor guardado. Además, el circuito **permite modificar el valor almacenado** al activar la señal de control, y enviar un valor por la señal de datos.

De aquí en adelante interpretaremos un flip-flop con su abstracción de alto nivel: una caja que es capaz de almacenar 1 bit de información, y modificar su valor.

4.2. Registros

Un flip-flop D representa una unidad de almacenamiento de un bit. Si combinamos varias unidades de almacenamiento de 1 bit, podemos obtener unidades capaces de almacenar más información, las que se denomina **registros**. En la figura 6 se observa un registro de 4 bits formado por 4 flip-flops D.

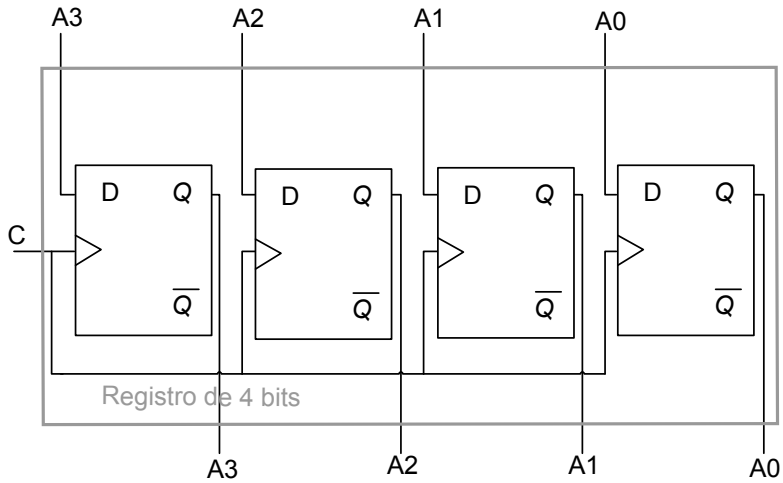


Figura 6: Registro de 4 bits formado por 4 flip-flops D

El registro más simple, como el que se observa en la figura, sólo permite cargar un valor cuando la señal de control se activa. Podemos agregar más funcionalidades al registro, por ejemplo un bit secundario de selección de carga L, y un bit que permita resetear el valor actual a 0 R, lo que se observa en la figura:

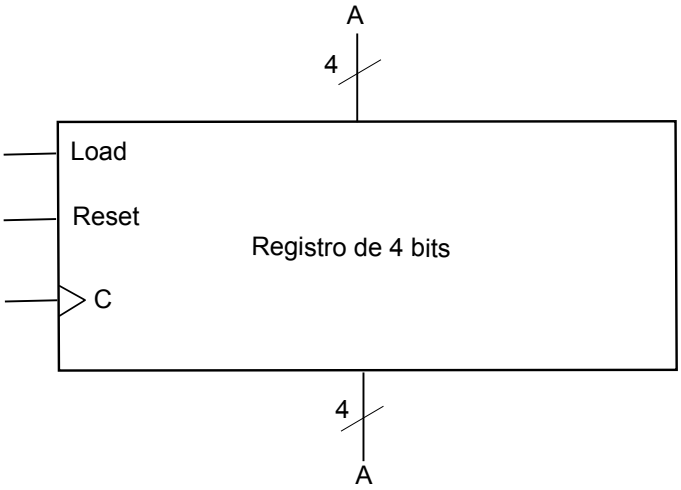


Figura 7: Registro de 4 bits con bit de carga y reset

4.2.1. Contadores

A partir de flip-flops y registros podemos construir circuitos de almacenamiento más complejos, como por ejemplo un contador. Un **contador incremental** es un circuito que almacena un cierto número y luego de recibir una señal de incremento aumenta en 1 el valor actual. Un **contador decremental** será equivalente pero en vez de aumentar en 1 disminuirá el valor en 1. Podemos combinar ambas opciones en un contador incremental/decremental o up/down.

Al igual con el registro, podemos agregar más funcionalidades al contador, como bit de carga y reset, obteniendo el componente que se observa en la figura 9.

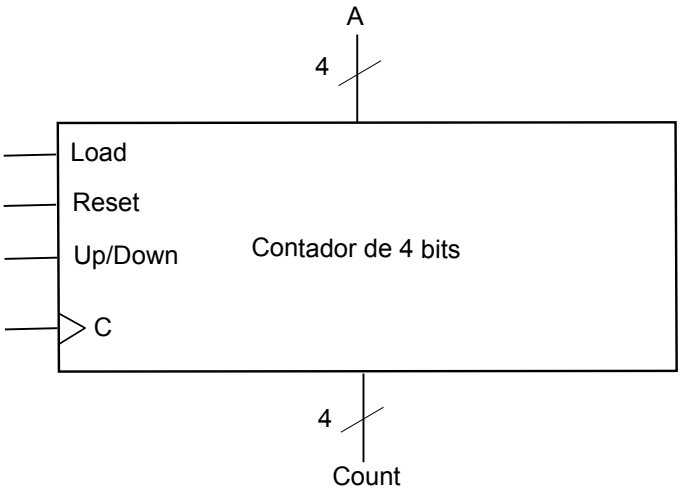


Figura 9: Contador incremental de 4 bits con carga y bit de reset.

4.3. Memorias

Los registros nos permiten almacenar cada uno una palabra o unidad de información. Nos gustaría poder escalar y tener un componente que almacene más información. Una opción simple es tener múltiples registros,

el problema esta en que tenemos que poder acceder a cada un de ellos, poder cargarlos y leer información, lo que hace complejo hacer el escalamiento con este método.

El componente que queremos, debe permitir almacenar varias palabras, a las cuales podamos acceder fácilmente, y también modificar fácilmente. El componente que cumple con esta función se denomina una **memoria** las cuales almacenan una cierta cantidad de palabras de un determinado tamaño y permite acceder a estas asociando a cada palabra un número o identificador representado por un número binario.

El identificador numérico usado para acceder a una palabra se denomina **dirección de memoria** y por consiguiente el proceso de acceder a una de las palabras se conoce como **direccionamiento**. Dado esto, la memoria puede ser pensada como una tabla de pares de valores: para cada dirección se asocia una palabra.

Dirección en decimal	Dirección en binario	Palabra
0	000	$Palabra_0$
1	001	$Palabra_1$
2	010	$Palabra_2$
3	011	$Palabra_3$
4	100	$Palabra_4$
5	101	$Palabra_5$
6	110	$Palabra_6$
7	111	$Palabra_7$

4.3.1. Tipos de memoria

Revisaremos dos tipos de memorias relevantes: la **random access memory** o **RAM** y la **read only memory** o **ROM**.

Memorias de escritura-lectura: RAM

La memoria RAM es una extensión de los registros. Al igual que estos ocupan como unida de almacenamiento de un bit un flip-flop, sin embargo a este se le agregan componentes para permitir cumplir con las funciones anteriormente descritas, formando lo que se denomina una **celda de memoria** o **memory cell**.

El diagrama de una memoria RAM se observa en la figura 13. La memoria RAM tiene tres buses de entrada: un bus de direccionamiento de k bits, que indica que palabra se quiere seleccionar (entrada *Address*); un bus de datos de n bits, que tiene el dato que se guardará en la posición seleccionada (entrada *Data In*); y un bus de control de 1 bit que indica si la memoria está en modo lectura (0) o escritura (1) (entrada *Write Enable* o *WE*). Adicionalmente cuenta con un bus salida de datos de n bits, que tiene el dato indicado por la dirección recibida (salida *Data Out*). El número n será el tamaño de la palabra, el número k indica la cantidad de bits disponibles para direccionar, y por tanto 2^k será la cantidad de palabras distintas que se pueden almacenar.

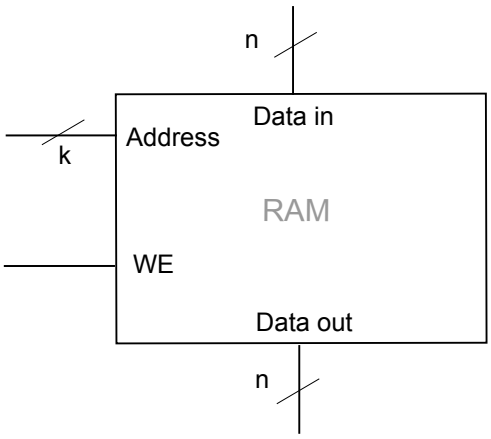


Figura 11: RAM de 2^k palabras de n bits

La RAM tiene dos modos de funcionamiento: escribir o leer. Veamos ambos modos a través de un ejemplo: supongamos que tenemos una RAM de 2 bits de direccionamiento, es decir $2^2 = 4$ palabras, y que cada palabra es de 8 bits = 1 byte. El diagrama de esta memoria es el siguiente:

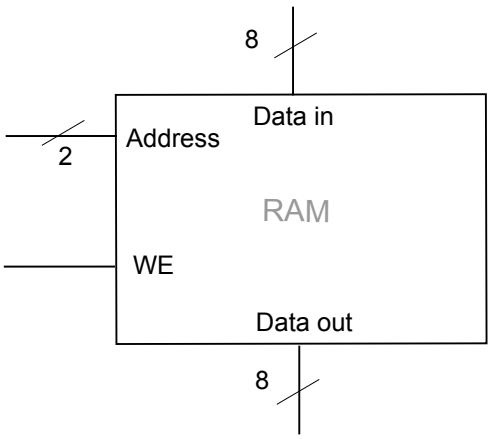


Figura 12: RAM de 2^2 palabras de 8 bits

Supongamos que la información inicial que tenemos en la RAM es la siguiente:

Dirección en decimal	Dirección en binario	Palabra en binario
0	00	00011010
1	01	01000010
2	10	00000000
3	11	00111111

Para leer la dirección 1 debemos colocar el valor 01 en la entrada *Address*, el valor 0 en la entrada *WE*, y obtendremos el valor 01000010 en la salida *Data Out*. Al igual que en los registros, en este modo lo que está en la entrada *Data In* no va a entrar.

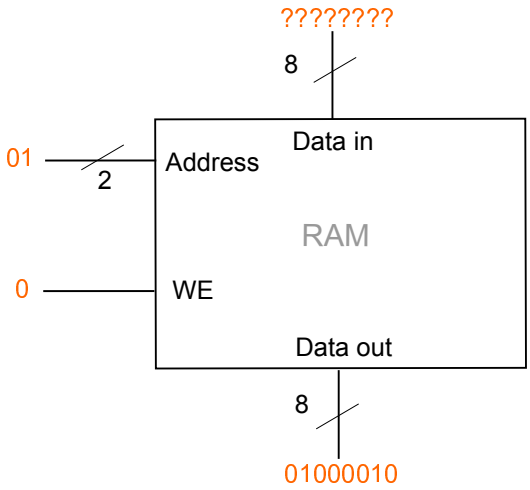


Figura 13: RAM en modo lectura

Para escribir en la dirección 2 debemos colocar el valor 10 en la entrada *Address*, el valor 1 en la entrada *WE*, y dato que queremos guardar en la entrada *Data In*, por ejemplo 11110000.

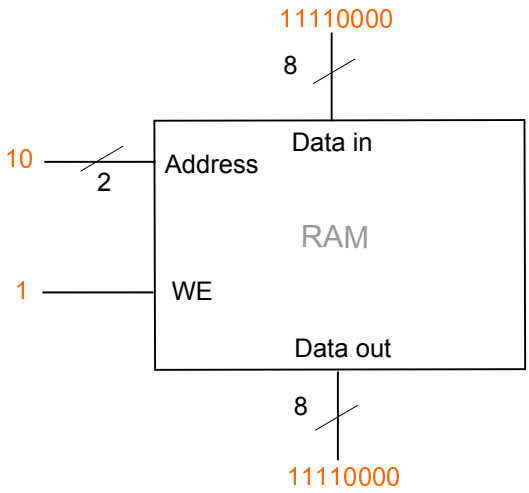


Figura 14: RAM en modo escritura

Memorias de sólo lectura: ROM

Un segundo tipo de memoria relevante son las memorias de sólo lectura o ROM. El concepto de memoria de «solo lectura» suena confuso, ya que si no se puede escribir información inicialmente, no tiene mayor sentido como memoria. Hablar de «solo lectura» en la práctica se interpreta como «se puede escribir sólo una vez» o también en algunos casos «se puede escribir más de una vez, pero a una velocidad y costo mucho mayor que leer».

Dada esta definición, la utilidad de una memoria de este tipo es para almacenar datos que no variarán en el tiempo o que variarán poco. El hecho de que una memoria sea de «sólo lectura» tiene que ver con la tecnología usada. A diferencia de la RAM, no se utilizan flip-flops para almacenar bits, sino que otros mecanismo. En la memoria ROM original y más simple el proceso de escritura consiste en soldar o dejar abiertos conexiones entre cables, lo que se interpreta como un 1 o 0.

El diagrama de la memoria ROM se observa en la figura 17. La memoria ROM es más simple que la RAM: tiene un sólo bus de entrada, el bus de direccionamiento de k bits, que indica que palabra se quiere seleccionar (entrada *Address*) y un bus salida de datos de n bits, que tiene el dato indicado por la dirección recibida (salida *Data Out*). Al igual que en la RAM el número n será el tamaño de la palabra, el número k

indica la cantidad de bits disponibles para direccionar, y por tanto 2^k será la cantidad de palabras distintas que se pueden almacenar.

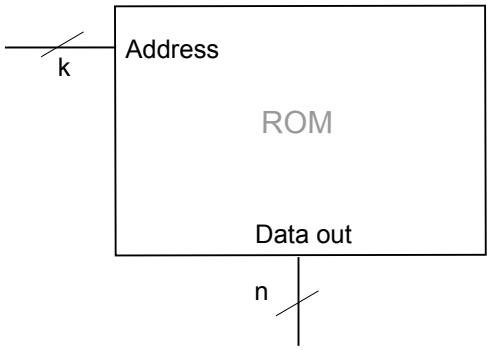


Figura 15: ROM de 2^k palabras de n bits

5. Representación de Variables y Arreglos en Registros y Memorias

5.1. Variables y Registros

Para almacenar una variable necesitábamos una representación física que permita almacenar una secuencia de bits, del tamaño indicado por el tipo de dato, además de poder identificar de alguna forma la representación para diferenciarla de otras. Los registros son una primera alternativa para lograr esto. Como se observó previamente un registros es básicamente un conjunto de flip-flops, es decir, cajas que almacenan bits. Dependiendo del tipo de dato, podríamos tener registros de distinto tamaño, que permitan representar los distintos tipos. Si queremos un tamaño único de registros, necesitamos que sea del tamaño del tipo de datos de mayor cantidad de bits. Por ejemplo, si queremos poder almacenar a lo más tipos de datos de 32 bits (como el tipo `int` de Java), necesitamos registros de 32 bits. Con estos registros también podemos representar tipos con menor cantidad de bits, por ejemplo de 8 bits (como el tipo `byte` de Java), para hacerlo basta con ocupar 8 bits de los 32, y no considerar el resto.

Para implementar la identificación, es necesario tener distintos registros para las distintas variables, de manera de poder decidir a que registro acceder conociendo el nombre de este. Por ejemplo si queremos tener dos variables, necesitamos que existan dos registros distintos al menos, que podríamos llamar A y B, y poder acceder a estos de manera diferenciada. Esto muestra una limitante del uso de registros para almacenar variables: necesitaríamos tantos registros como variables posibles para permitir la identificación.

En la práctica, los registros son ocupados para almacenar variables, pero solo de manera temporal, justo antes de realizar operaciones sobre éstas. Dado que los registros representarán solo variables temporales, no será necesario tener tantos registros como variables, y por tanto necesitaremos otro mecanismo mas escalable que permita almacenar una cantidad mayor de variables.

5.2. Variables y Memorias

Las memorias RAM representan un mecanismo ideal para evitar los problemas de los registros y permitir almacenar una mayor cantidad de variables. Como se detallé previamente, las memorias pueden ser pensadas como tablas, en que por un lado se tiene una dirección y por otra parte una palabra, que corresponde a un a secuencia de bits de un cierto tamaño. Esta definición nos indica directamente que las memorias poseen los dos elementos que necesitamos para almacenar variables: un identificador, que en este caso serían las direcciones; y una lugar donde almacenar una secuencia de bits, asociado al identificador.

Una elemento importante que se debe considerar es que un número almacenado en memoria puede estar guardado en varias palabras. Por ejemplo, si el tamaño de las palabras de la memoria es 1 byte = 8 bits (que es el tamaño que habitualmente se utiliza), y queremos almacenar el número de punto flotante de tipo

single precisión (`float`), siguiendo el estándar IEEE754 necesitamos 32 bits = 4 bytes = 4 palabras. De esta forma, para poder obtener un número de memoria, necesitamos saber a priori al menos de que tipo es el número, y cuantas palabras ocupa.

La información del tipo de dato y su tamaño puede no ser suficiente. Siguiendo el ejemplo anterior, tomemos un número de tipo `float`, por ejemplo el $0{,}5 = 0{,}1_2$ el cual se representa según el estándar con la secuencia de bits: 00111111000000000000000000000000. Como las palabras de nuestra memoria eran de tamaño 8 bits, debemos dividir nuestra secuencia en 4 partes: 00111111, 00000000 ,00000000 y 00000000, las cuales podríamos almacenar a partir de la dirección 0 de la siguiente forma:

Dirección en hexa	Dirección en binario	Palabra
0x00	000	00111111
0x01	001	00000000
0x02	010	00000000
0x03	011	00000000
0x04	100	$Palabra_4$
0x05	101	$Palabra_5$
0x06	110	$Palabra_6$
0x07	111	$Palabra_7$

El problema está en que esta es una de las formas en que podemos almacenar las 4 palabras que corresponde al número. De manera equivalente, podríamos almacenar las 4 palabras en el orden contrario:

Dirección en hexa	Dirección en binario	Palabra
0x00	000	00000000
0x01	001	00000000
0x02	010	00000000
0x03	011	00111111
0x04	100	$Palabra_4$
0x05	101	$Palabra_5$
0x06	110	$Palabra_6$
0x07	111	$Palabra_7$

Esto nos indica que además de saber el tipo del número y su tamaño asociado, debemos definir un orden en como se guardarán las palabras que compone el número. Este orden de las palabras se denomina **endianness**, y existen dos posibles formas de ordenar: **big endian**, en la que la palabra más significativa dentro del número (i.e. con los bits en las posiciones más altas) se almacena en la dirección menor (como el primer caso del ejemplo); **little endian**, en la que la palabra más significativa dentro del número (i.e. con los bits en las posiciones más altas) se almacena en la dirección mayor (como el segundo caso del ejemplo). Al diseñar un computador, se deberá definir cual de los dos órdenes se utiliza, y mantener la consistencia del orden para todos los tipos de datos.

5.3. Arreglos y Memorias

Las memorias también resultan útiles para almacenar estructuras de datos más complejas como los arreglos. Un arreglo requería un identificador de la lista de valores, la capacidad de guardar múltiples valores, y la posibilidad de acceder a estos múltiples valores de manera indexada. Veamos como almacenaríamos en memoria un arreglo unidimensional, comenzando con un arreglo de valores de tipo `byte` $0x00, 0x05, 0x0A, 0x0F$:

Dirección en hexadecimal	Dirección en binario	Palabra
0x00	000	$0x00$
0x01	001	$0x05$
0x02	010	$0x0A$
0x03	011	$0x0F$

Se observa que podemos ocupar distintas direcciones de memoria para guardar los distintos valores. El identificador del arreglo va a corresponder a la dirección del primer valor, en este caso, estamos guardando el arreglo a partir de la dirección 0x00. Se observa también que para poder acceder al i-ésimo elemento, nos basta con sumarle i a la dirección del primer valor. Por ejemplo, para acceder al valor arreglo[1], accedemos a la dirección de arreglo (0x00) sumada con la posición (0x01), es decir accedemos a la dirección 0x01. Esta es en parte la razón por la cual los arreglos comienzan con índice 0, dado que debido a eso es que funciona esta indexación.

El largo del arreglo en general se almacenará de manera independiente, y al acceder el arreglo habrá que estar al tanto de este largo para no pasarnos de entre los valores válidos.

Al igual que como se mencionó en el caso de las variables, un arreglo puede almacenar valores de tipos que tengan un tamaño mayor que la palabra. Revisemos un ejemplo de un arreglo de char (palabras de 2 bytes, 16 bits) $0x0000, 0x0005, 0x000A, 0x000F$:

Dirección en hexadecimal	Dirección en binario	Palabra
0x00	000	$0x00$
0x01	001	$0x00$
0x02	010	$0x00$
0x03	011	$0x05$
0x04	100	$0x00$
0x05	101	$0x0A$
0x06	110	$0x00$
0x07	111	$0x0F$

En este caso nuevamente surge el concepto de endianness: cada valor del arreglo deberá guardarse en un cierto orden (big o little endian), lo cual debe estar convenido previamente para poder interpretar correctamente el número. Otra diferencia de este caso es que ahora para poder indexar el i-ésimo valor del arreglo no basta con sumar la dirección del arreglo (0x00) con el índice. La fórmula correcta para el caso general de arreglos con tipos de tamaño distinto a 1 byte es: $dir(arreglo[i]) = dir(arreglo) + i \times sizeof(arreglo[i])$, donde $sizeof(arreglo[i])$ retorna el tamaño del tipo de dato del arreglo en bytes. Para el ejemplo del arreglo de valores de 2 bytes, si queremos acceder a la posición 2 del arreglo, debemos acceder a la dirección: $dir(arreglo[2] = 0x00 + 2 \times 2 = 0x04$.

5.3.1. Arreglos Multidimensionales

Un caso más complejo son los arreglos multidimensionales. Las memorias se prestan de manera natural para almacenar arreglos unidimensionales, como se mostró previamente, pero para arreglos de mayores dimensiones no es tan así, y se debe decidir de alguna forma como se guarda el arreglo en memoria.

Para el caso de matrices (arreglos de 2 dimensiones), existen dos convenciones: guardar la secuencia de filas o guardar la secuencia de columnas. Por ejemplo suponiendo la matriz de 2x3:

0x02	0x04	0x07
0x05	0xA	0x09

Guardándola con la convención de filas obtendríamos en memoria:

Dirección en hexadecimal	Dirección en binario	Palabra
0x00	000	$0x02$
0x01	001	$0x04$
0x02	010	$0x07$
0x03	011	$0x05$
0x04	100	$0x0A$
0x05	101	$0x09$

Guardándola con la convención de columnas obtendríamos en memoria:

Dirección en hexadecimal	Dirección en binario	Palabra
0x00	000	$0x02$
0x01	001	$0x05$
0x02	010	$0x04$
0x03	011	$0x0A$
0x04	100	$0x07$
0x05	101	$0x09$

La convención de filas es la más habitualmente usada. En este caso, para acceder a un valor matriz[i,j] se debe ocupar la fórmula: $dir(matriz[i,j] = dir(matriz) + i \times sizeof(matriz[i,j]) \times columnas + j \times sizeof(matriz[i,j])$

En nuestro ejemplo, para acceder al valor [1,1], en convención de filas, debemos acceder a la dirección: $dir(matriz[1,1] = 0x00 + 1 \times 1 \times 3 + 1 \times 1 = 0x04$

6. Referencias

- Shannon, C.; A Symbolic Analysis of Relays and Switching Circuits, MIT Press, 1937.

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2343 Arquitectura de Computadores

Interacción humano computador

1. Motivación

Construir una máquina que sea capaz de realizar operaciones y almacenar datos no tiene mayor utilidad a menos que una persona pueda interactuar con ella, para entregarle indicaciones e información y para obtener los resultados de los procesamientos realizados o la información almacenada. De esta forma para completar un computador funcional debemos agregar la capacidad de interacción entre humano y computador, y estudiar como implementar dicha interacción.

2. Interacción simple

La comunicación entre persona y computador debe ser idealmente en ambas direcciones. En general se dice que un dispositivo que permite que un humano interactúe con un computador se denomina dispositivo de **entrada** o **input**. Cuando el dispositivo es controlado por el computador para entregar información a un humano se denomina dispositivo de **salida** o **output**. Al conjunto de estos dispositivos se les denomina de Entrada y Salida, **E/S**, o Input y Output, **I/O**.

La forma más simple de interactuar con un computador es realizarlo directamente en el lenguaje que este maneja: números binarios. De manera similar la forma más simple para que un computador se comunique con nosotros es a través de mecanismos que fácilmente puedan ser controlados mediante representación binaria.

2.1. Dispositivos de entrada

2.1.1. Interruptores y botones

Un primer dispositivo de entrada que permite entregar información binaria al computador es el **interruptor**. Un interruptor es básicamente un mecanismo con dos estados, conectado y desconectado, que en caso de estar conectado permite el paso de corriente eléctrica entre sus extremos, y en caso de estar desconectado no lo permite. Estos dos estados pueden ser fácilmente interpretados como los dos posibles números binarios: si el interruptor está desconectado, se interpreta como un 0, si el interruptor está conectado, se interpreta como un 1.

Los interruptores nos permiten interactuar de manera directa con los componentes vistos previamente. Por ejemplo, podemos usar interruptores para cargar los bits de datos almacenados en un par de registros que estén conectados a una ALU, como se observa en la figura 1. Es importante señalar que el valor **5V** indicado en la figura conectado a los interruptores, significa *5 volts* y representa el voltaje eléctrico más comunmente usado para representar un 1 en circuitos binarios.

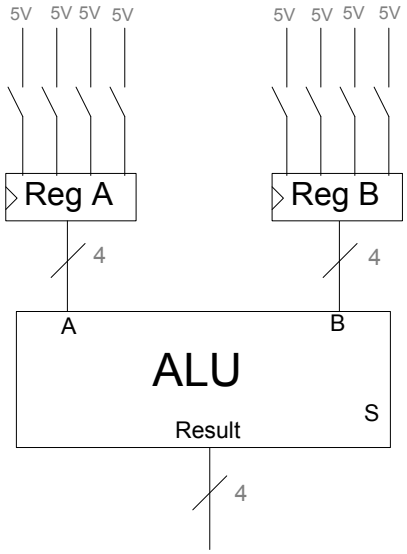


Figura 1: Interruptores usados para cargar valor en registro. En el diagrama los interruptores se representan como una línea cortada.

También es posible usar interruptores para representar bits de control: por ejemplo, podemos usar tres interruptores para indicar la operación seleccionada por la ALU, como se observa en la figura 2.

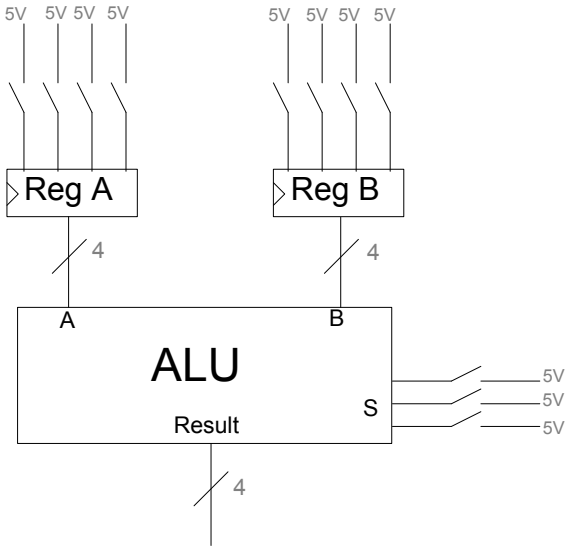


Figura 2: Interruptores usados para seleccionar operación de la ALU.

Para completar el circuito anterior, falta agregar la señal de control de los registros, para permitir el almacenamiento de los números entregados a tráves de los interruptores. Una opción sería agregar otro interruptor, que al estar conectado indique un 1 en la señal de control y permita el almacenamiento. Dado que los registros tienen señal de control activada por flanco de subida, no necesitamos tener la señal 1 todo el tiempo, y nos basta con enviar un «pulso» que indique un 1 para que deje almacenar, pero que luego la señal de control vuelva a 0. Para esto podemos ocupar otro dispositivo: un **botón**. Al igual que un interruptor, un botón puede tener dos estados, conectado y desconectado, pero la diferencia es que el botón debe permanecer presionado para mantener la conexión, mientras que el interruptor queda fijo en el estado de conexión, sin necesidad de tener que estar presionándolo.

Podemos agregar entonces un botón para controlar la señal de almacenamiento de los registros, como se observa en la figura 3.

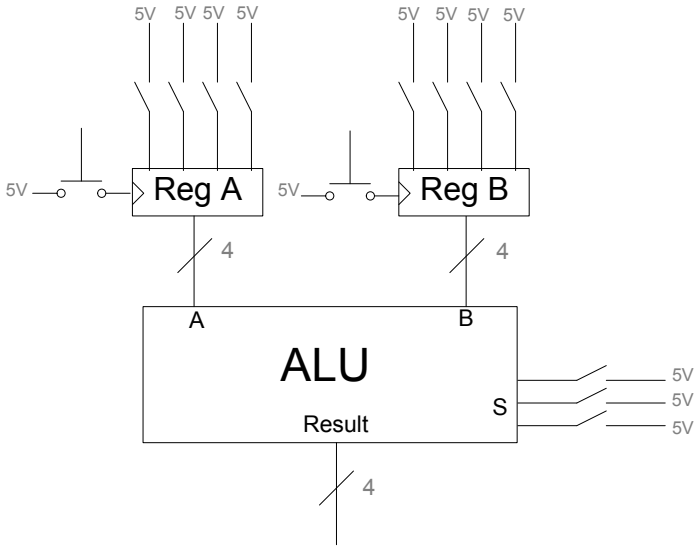


Figura 3: Botones usados para almacenar resultados en registros.

2.1.2. Problemas y limitaciones

Si los interruptores y botones fueran ideales, cuando se conectan deberían inmediatamente representar un 1, y cuando se desconectan, inmediatamente un 0, como se observa en la figura 4.

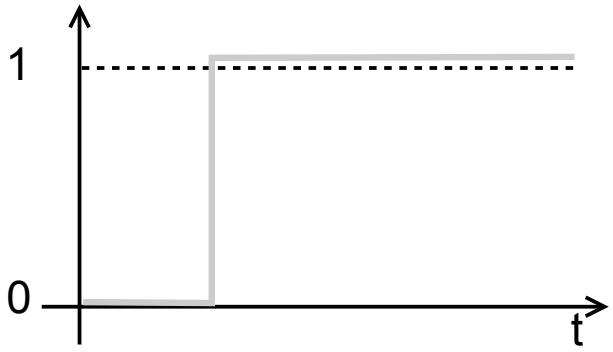


Figura 4: Interruptor ideal

Sin embargo, esto no ocurre en la práctica, debido a que los circuitos eléctricos no son ideales, y por tanto no tienen transiciones instantáneas. Lo que ocurre en la práctica es lo que se observa en la figura 5: al cerrar el interruptor, la señal eléctrica «**rebota**» varias veces en torno al valor de voltaje asociado al 1 binario, hasta que finalmente se regula en el estado correcto. Mientras la señal está rebotando se dice que esta está en **regimen transiente**, y cuando se estabiliza, entra en **regimen permanente**.

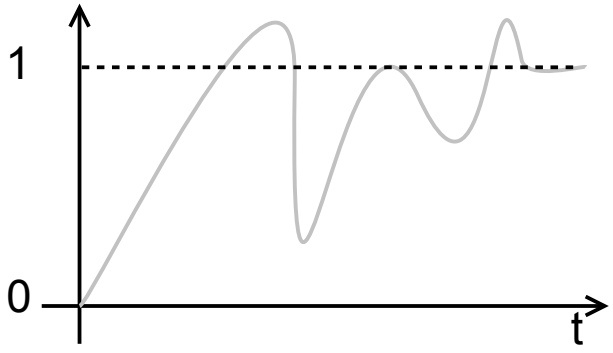


Figura 5: Interruptor real

El problema del período transiente es que la señal va a cambiar de 0 binario a 1 binario varias veces, ya que ambos valores son detectados para ciertos rangos, no para valores específicos. Así, presionar un botón una vez, lo que debería interpretarse como un sólo 1, podría percibirse como una secuencia de 1s seguidos, lo que dependiendo de que estemos haciendo con el dato ingresado puede presentar problemas.

La solución a este problema consiste en agregar entre el interruptor y el componente un circuito de retraso, que sólo dejará pasar la señal luego de un cierto tiempo. De esta forma, si este circuito se calibra para dejar pasar la señal sólo después del transiente, se evita el problema del rebote.

2.2. Dispositivos de salida

2.2.1. LEDs y displays

Para que el computador despliegue información de manera simple, necesitamos algo similar a los interruptores, es decir que funcionen con dos estados representables por los números 0 y 1, y que dependiendo del estado, se modifique alguna propiedad del dispositivo que sea percibible por el ser humano. Un dispositivo que cumple con lo anteior es el **led** (light emitting diode). Un led es un componente con dos conectores, que cumple que cuando pasa corriente entre estos se ilumina, y cuando no pasa corriente, no se ilumina. De esta forma podemos interpretar un 1 como una luz prendida, y un 0 como una luz apagada.

En la figura 4 se observan 4 leds conectados a la salida de la ALU para representar la información del resultado de la operación como un número binario, mediante leds encendidos y apagados. Los leds se representan con el símbolo de un diodo (triángulo) más dos flechas que indican que emite luz. El símbolo al cual se conecta el led en el diagrama se conoce como «tierra» y se interpreta como el punto en que se completa el circuito eléctrico.

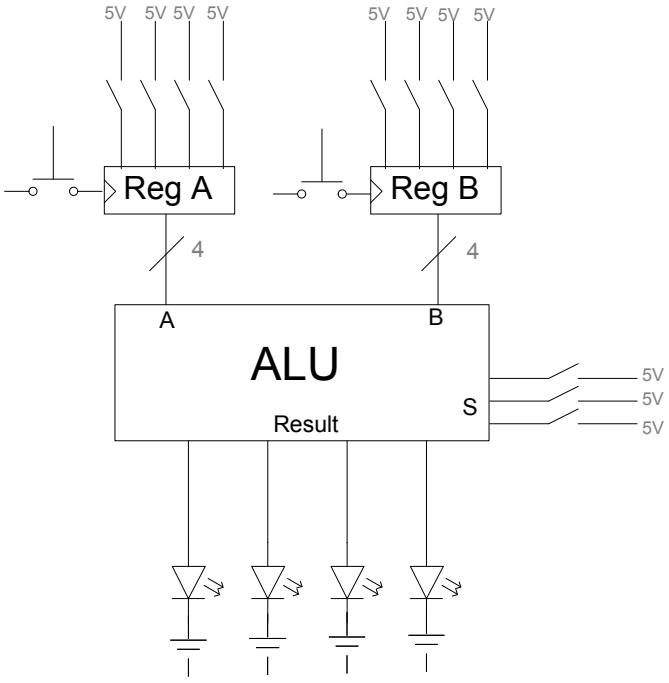


Figura 6: Display binario para mostrar operación realizada con la ALU.

Ocupando directamente los leds para representar números binarios ya nos permite percibir información desde el computador, sin embargo, nos gustaría que esta información estuviera en un formato más fácil de leer y entender que un número binario. Para lograr esto se pueden utilizar varios leds, que en conjunto formen símbolos más complejos, por ejemplo todos los dígitos decimales. Un componente que permite esto es el denominado **dispay de 7 segmentos**, que corresponde a un arreglo de 7 leds ubicados en forma de 8. Con

este display es posible, encendiendo ciertos leds específicos, formar los símbolos de los dígitos decimales. Más aún, podemos incluso representar algunas letras, lo que nos permite representar números en hexadecimal, como se observa en la figura 5.

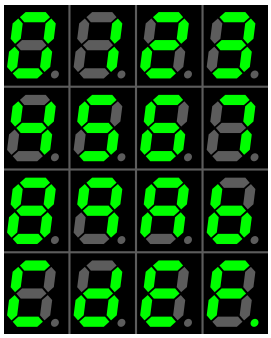


Figura 7: Dígitos hexadecimales representados con displays de 7 segmentos

Podemos entonces, reemplazar nuestro display binario del circuito anterior por un display de 7 segmentos que entregará el resultado en representación hexadecimal. Para lograr esto, sin embargo, debemos diseñar un circuito que convierta el número binario de 4 bits en los 7 bits necesarios para prender los leds que correspondan en el display representar el número correcto. Este circuito tendrá 4 entradas y 7 salidas y puede ser fácilmente diseñados con compuertas básicas. En la figura 6 se muestra una abstracción de este componente o controlador del display, completando el cicruito para desplegar el resultado de la ALU.

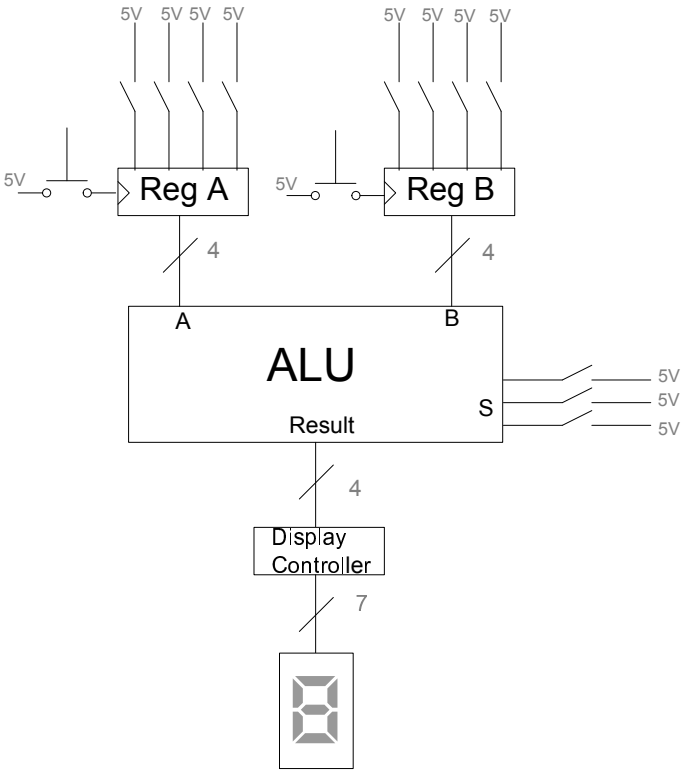


Figura 8: Display de 7 segmentos para mostrar operación realizada con la ALU

3. Interacción avanzada: Señales continuas

Los dispositivos mostrados en la sección previa permiten tener una interacción básica con los computadores. Sin embargo, no nos bastan para realizar interacciones más complejas. La comunicación de entrada

vista previamente, por ejemplo, obliga al humano a ingresar el número en binario, lo que no es lo ideal, ya que limita la complejidad de la información que se puede enviar.

La solución al problema de interactuar con información más compleja se basa en primer lugar en el uso de sensores y dispositivos que sean capaces de convertir información en señales eléctricas (y vice versa). Una vez convertida la información en señal eléctrica, se procederá a convertir en números binarios entendibles por el computador.

3.1. Dispositivos de entrada

Para explicar como ingresar información compleja a un computador, utilizaremos como ejemplo el sonido. El sonido se puede interpretar físicamente como el movimiento del aire, producido por ondas de presión. Como se señaló anteriormente, el primer paso para poder ingresar información al computador es convertirla a una señal eléctrica. En el caso del sonido, el dispositivo que convierte movimiento de aire en señal eléctrica es un micrófono. Un micrófono consiste en un imán que se mueve por la presión del aire y que al moverse induce corriente eléctrica proporcional al movimiento. De esta manera el movimiento del aire se traduce en una señal de corriente eléctrica.

Para convertir la señal eléctrica en información almacenable en el computador se requieren realizar diversos pasos que se describen en la siguiente sección.

3.1.1. Conversión de señal continua a información digital

Siguiendo el ejemplo anterior, luego de usar el micrófono obtenemos una señal eléctrica, la cual se puede graficar como una variación en la intensidad de corriente en el tiempo, como se observa en la figura 9. La señal en este momento se caracteriza por ser una **señal continua** o **análoga**, es decir para todo tiempo tiene un valor, no hay saltos entre instantes de tiempo.

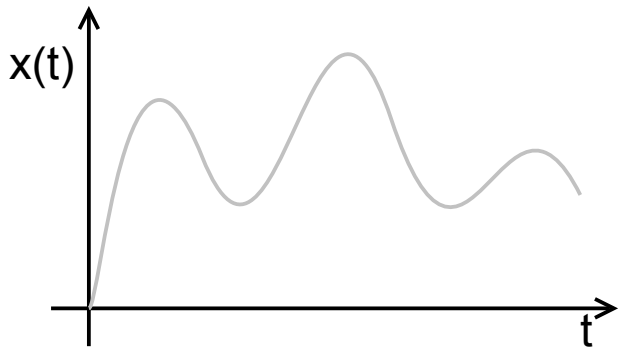


Figura 9: Señal continua

Podemos pensar la función descrita en la figura 9 como un conjunto de valores asociados a un determinado tiempo. Nos interesaría entonces poder almacenar estos números en el computador, y de esta forma almacenar la señal completa. Sin embargo, como la señal es continua este conjunto de valores es infinito, ya que para cada posible tiempo en el rango determinado, existirá un número particular que indicará el valor de la señal. Esto nos presenta un problema, ya que el computador no tiene espacio de almacenamiento infinito, y por tanto, debemos reducir la cantidad de números para poder almacenarlos.

Muestreo

Un proceso que permite reducir la información de una señal es el **muestreo** el cual consiste en obtener una muestra representativa de valores de la señal, separados por deltas de tiempo iguales, como se observa en la figura. Esta muestra de valores también se interpreta como una señal, denominada **señal de tiempo discreto** o **señal discreta**, y debido a esto este proceso también se denomina **discretizar** la señal.

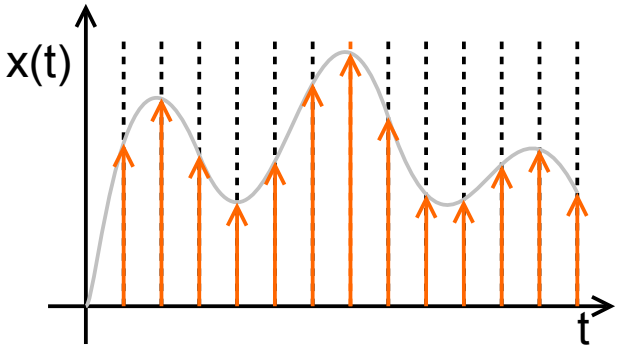


Figura 10: Señal discreta

El delta de tiempo entre cada muestra se denomina el **período de muestreo**, simbolizado por la letra T, que representa cada cuanto estamos obteniendo una nueva muestra. La **frecuencia de muestreo**, f, es una medida equivalente que representa el inverso multiplicativo del período ($T = \frac{1}{f}$).

Cuantización

Se podría pensar que luego de discretizar la señal estamos listos para almacenar la información en el computador. Esto no es así, debido a que aunque ahora tenemos un número finito de valores, cada uno de estos valores puede estar dentro de un rango infinito de posibilidades. El proceso que se utiliza para reducir este rango infinito se denomina **cuantización**, y consiste en definir que un valor en un determinado rango puede tomar sólo un valor de una serie finita. Al igual que en el muestreo la distancia entre estos posible valores es constante. La cantidad de valores disponible se denomina **resolución** y puede ser medida tanto como cantidad de números, como en bits (esto último es lo más habitual).

Si se realiza cuantización directamente sobre la señal continua se obtiene una señal como la de la figura 11, la cual tienen infinitos valores en el tiempo, pero cada uno de estos valores toma solo un valor dentro de un conjunto finito de posibilidades.

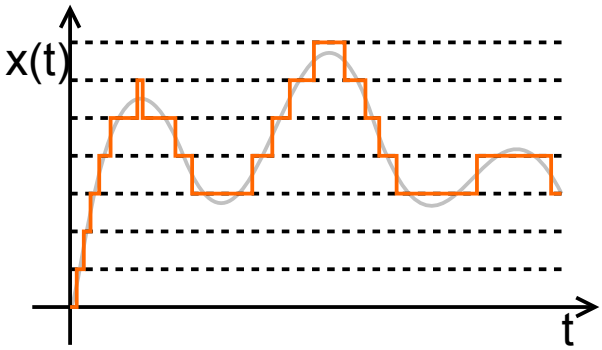


Figura 11: Señal cuantizada de tiempo continuo

Conversión análogo/digital

La conversión de una señal análoga a información almacenable por el computador involucra, como es de espear, la mezcla de los dos procesos visto anteriormente: muestreo y cuantización, en ese orden. Luego de realizar ambos procesos, se dice que se ha obtenido una **señal digital**, es decir un conjunto de valores numéricos que si puede ser almacenado por el computador, lo que se observa en la figura 12.

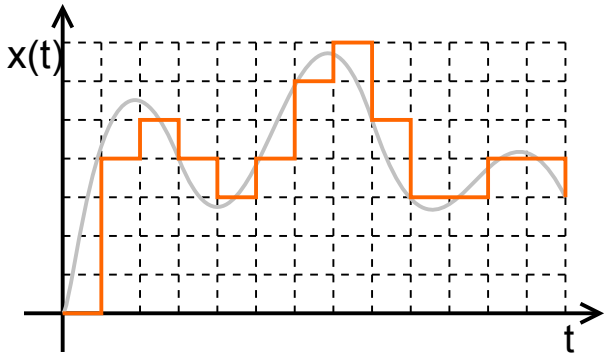


Figura 12: Señal digital

El componente que se utiliza para convertir una señal elétrica continua en una señal digital se denomina **conversor análogo digital**, el cual recibe una entrada análoga y entrega con un período T segundos valores representados por n bits de resolución.

Problemas y limitaciones

El principal problema de este proceso antes descrito es la pérdida de información. Dado que estamos reduciendo infinitos valores de rango infinito, a un número finito de valores de rango finito, se está reduciendo también la infomración. Para aumentar la cantidad de información obtenida, es necesario aumentar los dos parámetros de la conversión: frecuencia de muestreo y resolución. Si queremos almacenar más valores por segundo, debemos aumentar la frecuencia de muestreo; si queremos que cada valor tenga mayor precisión, aumentamos la resolución.

3.2. Dispositivos de salida

3.2.1. Conversión de información digital a señal continua

La conversión de una señal digital a una análoga corresponde al proceso inverso que el antes descrito. Para realizar este proceso se utiliza un componente denominado **conversor digital-análogo**, el cual recibe n bits cada T segundos y entrega una señal eléctrica continua.

El proceso consiste en utilizar lo que se denomina un **filtro de reconstrucción** que a partir de cada par de valores digitales, interpola los valores intermedios, completando la señal. Una vez reconstruida la señal continua, se pasa a convertirla nuevamente a al representación física que corresponda. Por ejemplo en el caso del audio, se ocupa un parlante, el cual recibe una señal de eléctrica, y la utiliza para mover un imán, el cual a su vez induce movimiento en el aire, generando el sonido que corresponde.

4. Referencias e información adicional

- Buxton, B., Human Input to Computer Systems: Theories, Techniques and Technology, http://billbuxton.com/inputManuscript.html
- Shannon, C., Communication in the Presence of Noise, 1949, http://www.stanford.edu/class/ee104/shannonpaper.pdf
- Englebart, D., A Research Center for Augmenting Human Inteligence, 1968, http://video.google.com/videoplay?docid=-8734787622017763097
- Wikipedia, Quantization (signal processing), http://en.wikipedia.org/wiki/Quantization_%28signal_processing%29

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2343 Arquitectura de Computadores

Programabilidad I: Automatización de operaciones y manejo de datos

©Alejandro Echeverría

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.

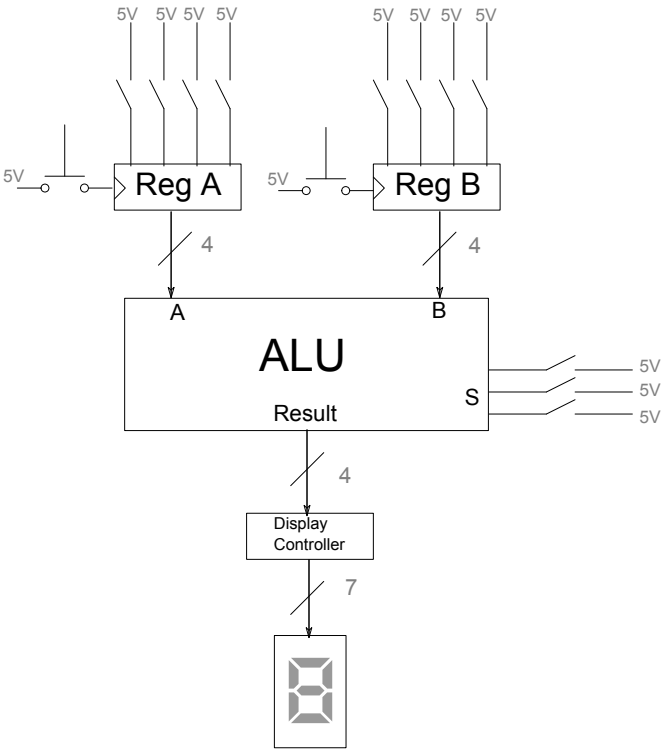


Figura 1: Calculadora de 4 bits.

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 se quisiera realizar ahora una operación que ocupe como operando el resultado obtenido (por ejemplo $6-4+2$) el usuario debe encargarse de colocar el resultado obtenido en los interruptores de manera manual. Para lograr eliminar al usuario del loop directo de ingreso de datos, debemos tener una forma de poder ir acumulando 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 estaremos desplegando el resultado en el display de 7 segmentos (que está limitado a números de 4 bits), podemos extender nuestro computador a 8 bits, aumentando el tamaño de los registros y la ALU.

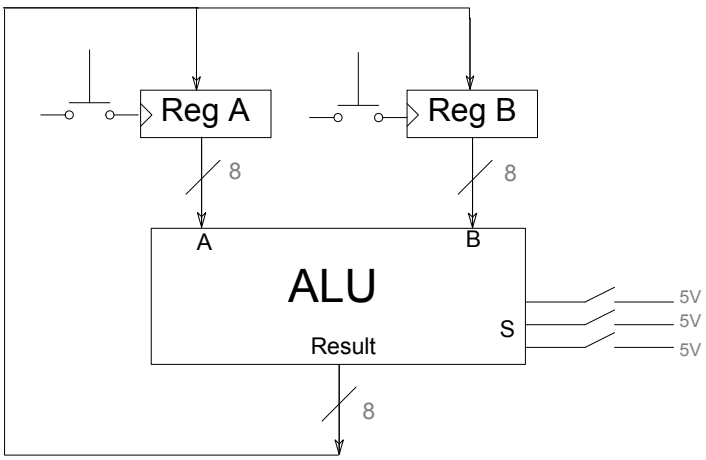


Figura 2: Se agregan registros acumuladores para almacenar secuencias de operaciones, y se modifica el tamaño de los registros y la ALU a 8 bits.

Se observa que para lograr esto se eliminaron tanto los interruptores de carga de datos como 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. Un capacidad importante que se pierde es la capacidad de cargar inicialmente los valores de los registros, lo que también será resuelto más adelante. Por ahora, la acumulación del resultado es suficientemente relevante como para eliminar esas otras capacidades.

Dada la capacidad de acumulación agregada a la máquina, 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 solucionar esto debemos agregar la capacidad de controlar la carga de los registros, lo que podemos lograr agregando una señal de control **Load** en ambos registros, como se ve en la figura 3.

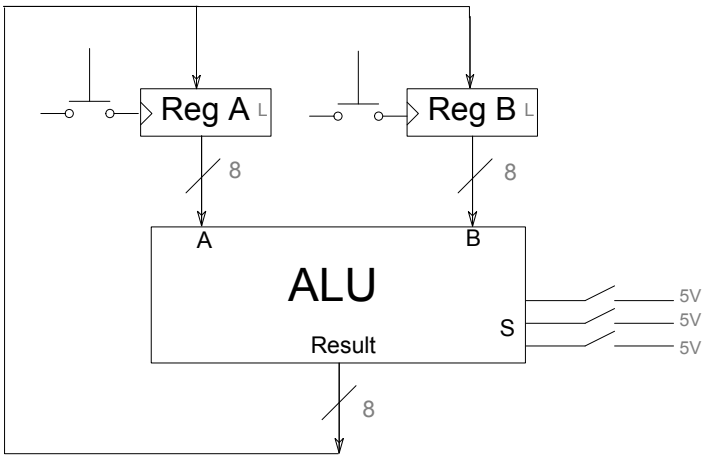


Figura 3: Se agregan señales de carga a los registros.

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=notA
0	1	1	0	0	B=notA
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.

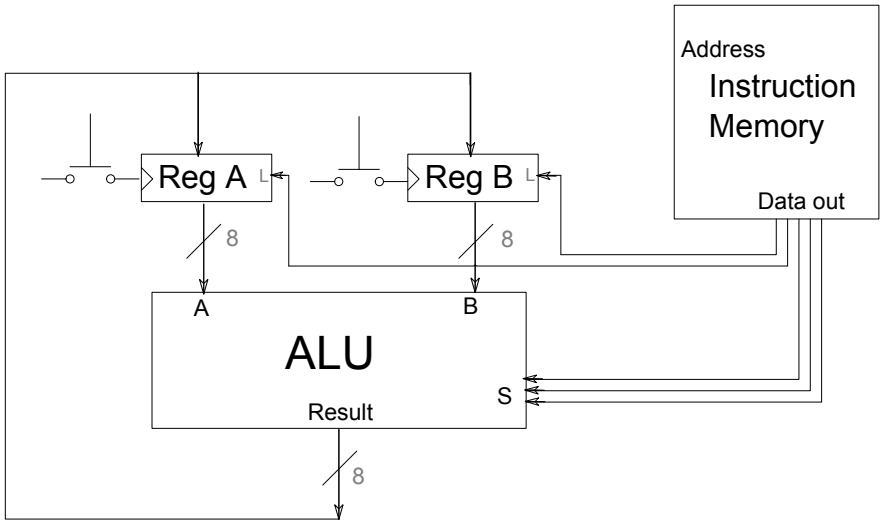


Figura 4: Se agrega una memoria ROM para almacenar instrucciones.

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.

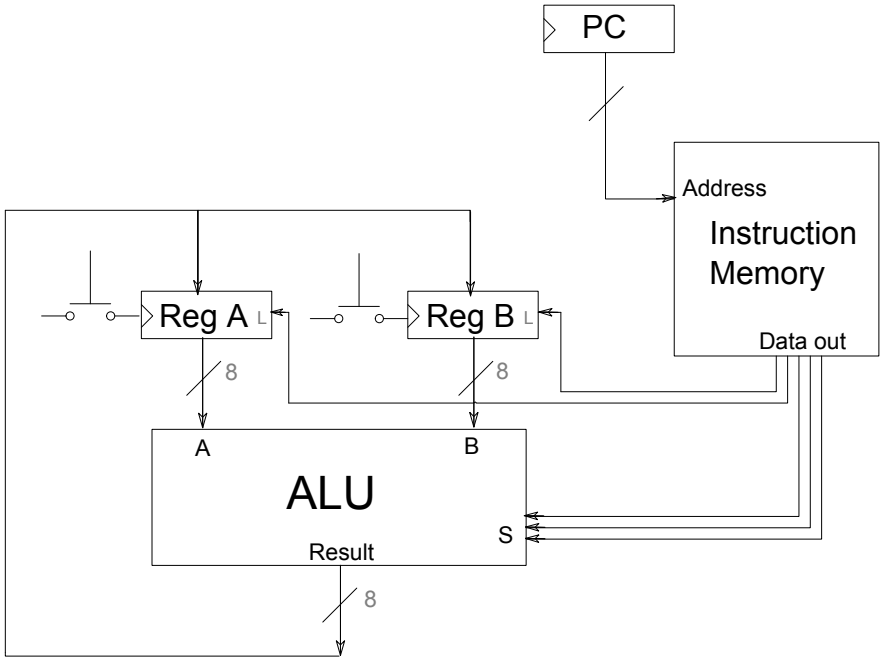


Figura 5: Se agrega un contador para direccionar secuencialmente la memoria de instrucciones: program counter.

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.

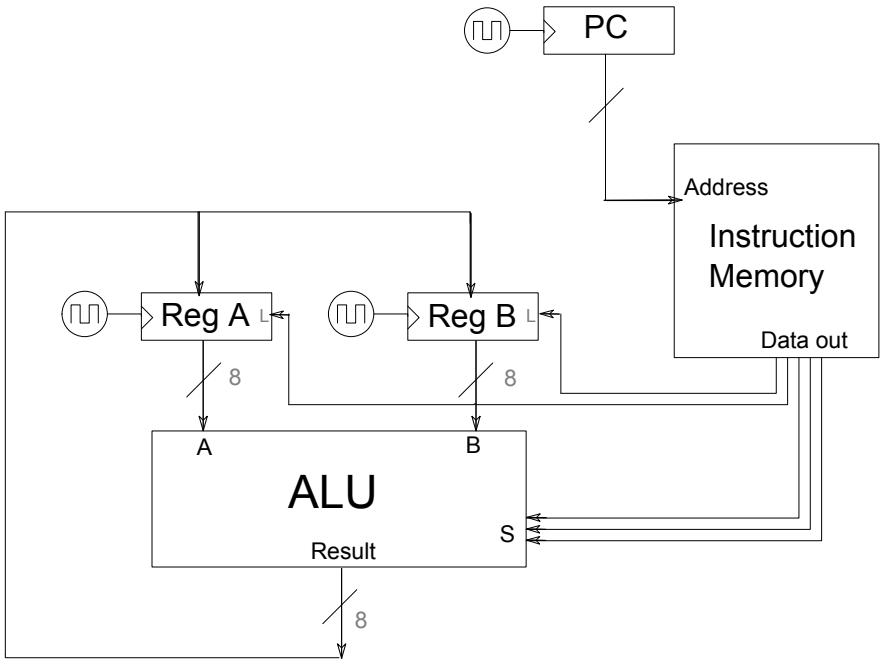


Figura 6: Se agrega un clock para sincronizar y automatizar el funcionamiento del computador.

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 $3 \times t_{gate} + t_{wire}$.

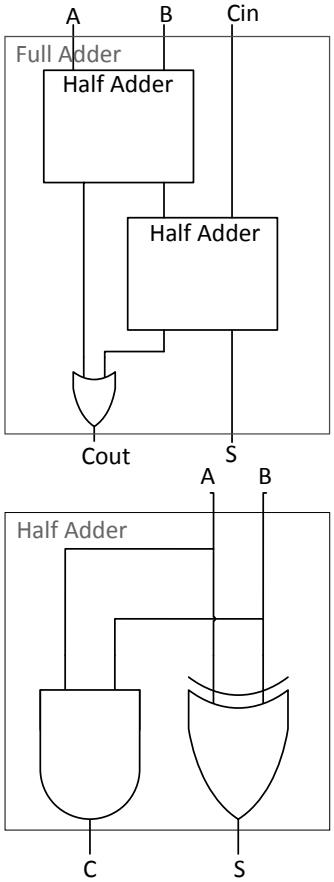


Figura 7: Circuitos de un full-adder y half-adder.

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.

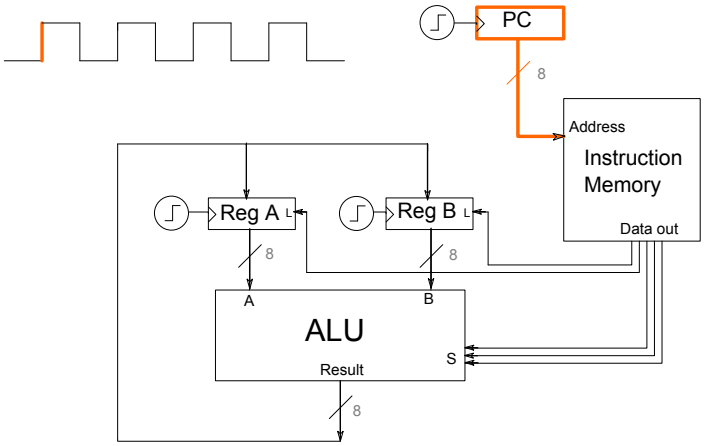


Figura 8: Program counter activado por flanco de subida.

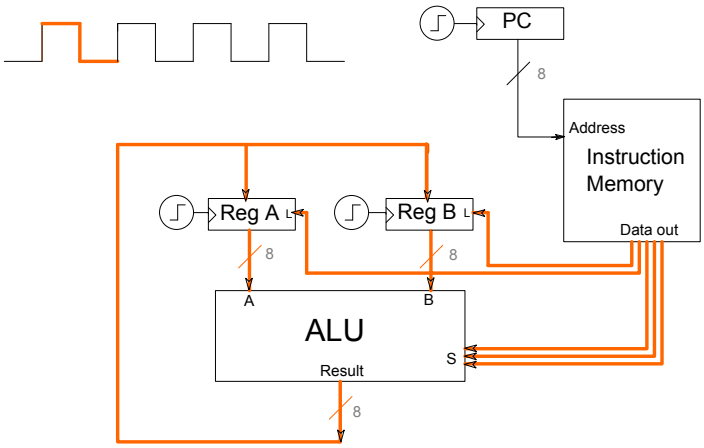


Figura 9: Cuando el clock subió y luego cuando baja, el program counter y los registros no se actualizan, dejando tiempo para procesar la instrucción y ejecutarla.

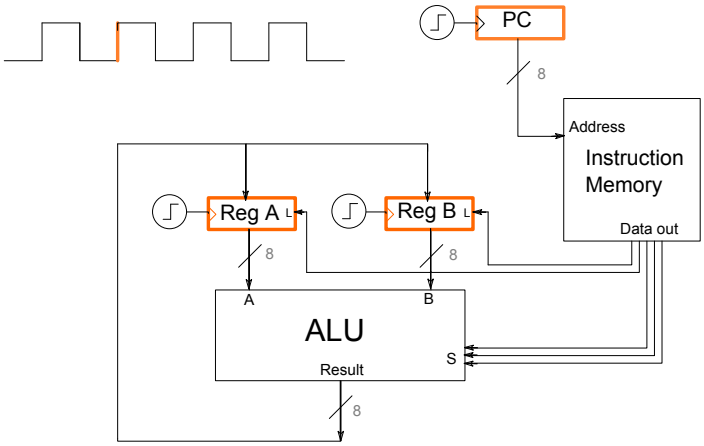


Figura 10: Los registros son activados en flanco de subida, guardando los resultados si correspondían.

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 1, 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.

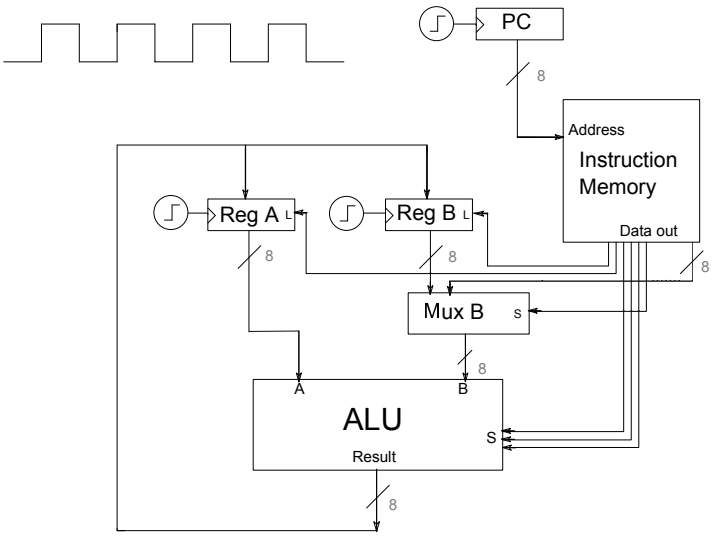


Figura 11: Computador con operaciones con literales.

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 2.

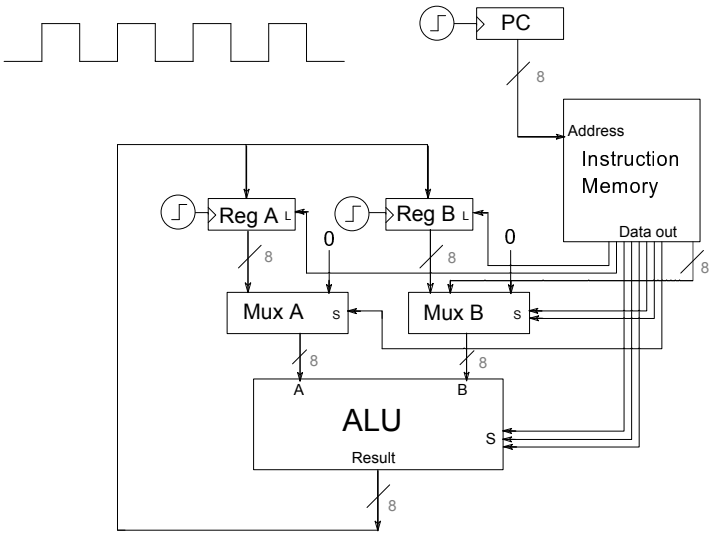


Figura 12: Computador con carga de valores de registros mediante literales.

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 1. 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	0	0	1	0	0	0	A=Lit
0	1	0	0	1	0	0	0	B=Lit
1	0	0	0	0	0	0	0	A=A+B
0	1	0	0	0	0	0	0	B=A+B
1	0	0	0	1	0	0	0	A=A+Lit
1	0	0	0	0	0	0	1	A=A-B
0	1	0	0	0	0	0	1	B=A-B
1	0	0	0	1	0	0	1	A=A-Lit
1	0	0	0	0	0	1	0	A=A and B
0	1	0	0	0	0	1	0	B=A and B
1	0	0	0	1	0	1	0	A=A and Lit
1	0	0	0	0	0	1	1	A=A or B
0	1	0	0	0	0	1	1	B=A or B
1	0	0	0	1	0	1	1	A=A or Lit
1	0	0	0	0	1	0	0	A=notA
0	1	0	0	0	1	0	0	B=notA
1	0	0	0	1	1	0	0	A=notLit
1	0	0	0	0	1	0	1	A=A xor B
0	1	0	0	0	1	0	1	B=A xor B
1	0	0	0	1	1	0	1	A=A xor Lit
1	0	0	0	0	1	1	0	A=shift left A
0	1	0	0	0	1	1	0	B=shift left A
1	0	0	0	1	1	1	0	A=shift left Lit
1	0	0	0	0	1	1	1	A=shift right A
0	1	0	0	0	1	1	1	B=shift right A
1	0	0	0	1	1	1	1	A=shift right Lit

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 2.

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

Tabla 4: Opcode y señales de control.

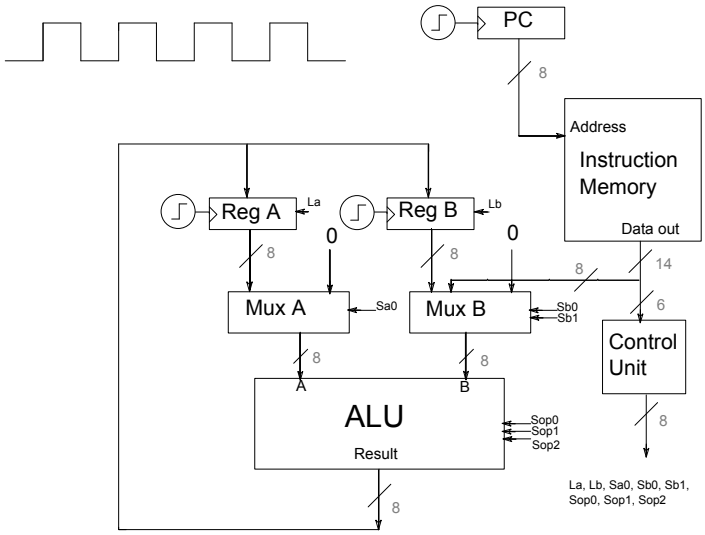


Figura 13: Computador con unidad de control.

7. Assembly y set de instrucciones

El uso de los opcodes descritos en la tabla 2 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 3. 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	0	0	1	0	0	0	A=Lit
MOVBL	000011	0	1	0	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=notA
NOTB	010001	0	1	0	0	0	1	0	0	B=notA
NOTL	010010	1	0	0	0	1	1	0	0	A=notLit
XORA	010011	1	0	0	0	0	1	0	1	A=A xor B
XORB	010100	0	1	0	0	0	1	0	1	B=A xor B
XORL	010101	1	0	0	0	1	1	0	1	A=A xor Lit
SHLA	010110	1	0	0	0	0	1	1	0	A=shift left A
SHLB	010111	0	1	0	0	0	1	1	0	B=shift left A
SHLL	011000	1	0	0	0	1	1	1	0	A=shift left Lit
SHRA	011001	1	0	0	0	0	1	1	1	A=shift right A
SHRB	011010	0	1	0	0	0	1	1	1	B=shift right A
SHRL	011011	1	0	0	0	1	1	1	1	A=shift right Lit

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 4. 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	0	0	1	0	0	0	A=Lit
	B,Lit	000011	0	1	0	0	1	0	0	0	B=Lit
ADD	A,B	000100	1	0	0	0	0	0	0	0	A=A+B
	B,A	000101	0	1	0	0	0	0	0	0	B=A+B
	A,Lit	000110	1	0	0	0	1	0	0	0	A=A+Lit
SUB	A,B	000111	1	0	0	0	0	0	0	1	A=A-B
	B,A	001000	0	1	0	0	0	0	0	1	B=A-B
	A,Lit	001001	1	0	0	0	1	0	0	1	A=A-Lit
AND	A,B	001010	1	0	0	0	0	0	1	0	A=A and B
	B,A	001011	0	1	0	0	0	0	1	0	B=A and B
	A,Lit	001100	1	0	0	0	1	0	1	0	A=A and Lit
OR	A,B	001101	1	0	0	0	0	0	1	1	A=A or B
	B,A	001110	0	1	0	0	0	0	1	1	B=A or B
	A,Lit	001111	1	0	0	0	1	0	1	1	A=A or Lit
NOT	A,A	010000	1	0	0	0	0	1	0	0	A=notA
	B,A	010001	0	1	0	0	0	1	0	0	B=notA
	A,Lit	010010	1	0	0	0	1	1	0	0	A=notLit
XOR	A,A	010011	1	0	0	0	0	1	0	1	A=A xor B
	B,A	010100	0	1	0	0	0	1	0	1	B=A xor B
	A,Lit	010101	1	0	0	0	1	1	0	1	A=A xor Lit
SHL	A,A	010110	1	0	0	0	0	1	1	0	A=shift left A
	B,A	010111	0	1	0	0	0	1	1	0	B=shift left A
	A,Lit	011000	1	0	0	0	1	1	1	0	A=shift left Lit
SHR	A,A	011001	1	0	0	0	0	1	1	1	A=shift right A
	B,A	011010	0	1	0	0	0	1	1	1	B=shift right A
	A,Lit	011011	1	0	0	0	1	1	1	1	A=shift right Lit

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 1 muestra el diagrama con la memoria de datos agregad 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.

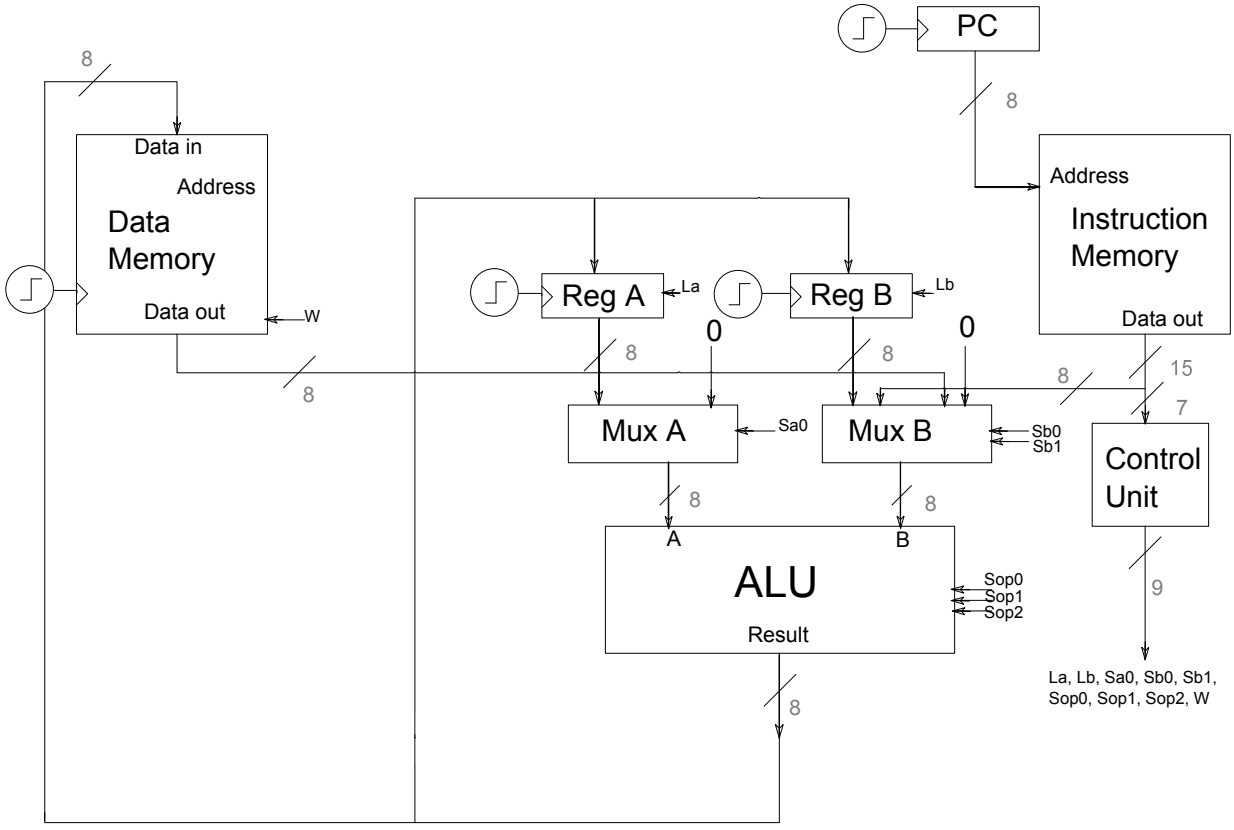


Figura 14: Memoria de datos agregada al computador básico

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 2 muestra el diagrama de las conexiones necesarias para realizar ambos modos de direccionamiento.

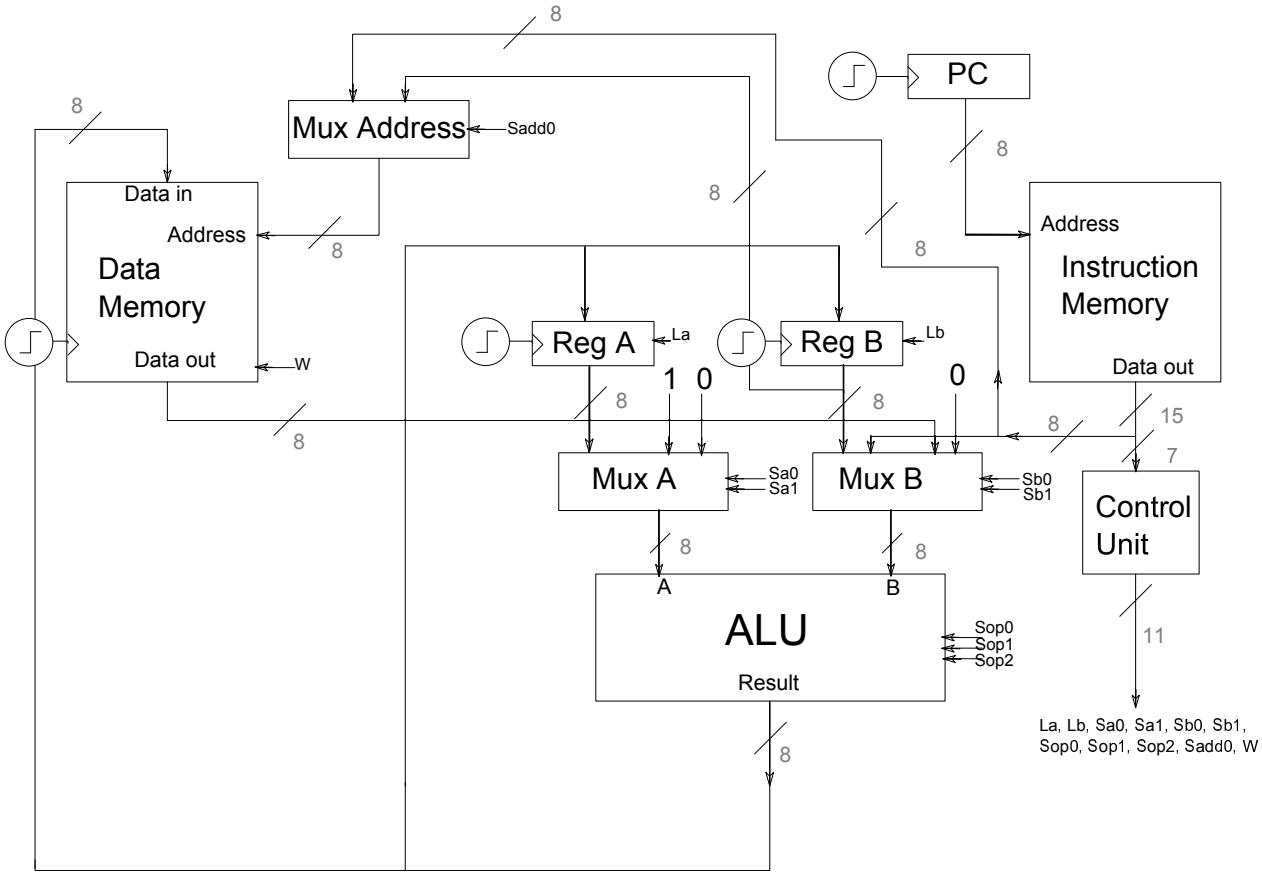


Figura 15: Modos de direccionamiento del computador básico

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	A,(Dir)	A=notMem[Dir]		NOT A,(var1)
	A,(B)	A=notMem[B]		-
	(Dir)	Mem[Dir]=not A		NOT (var1)
XOR	A,(Dir)	A=A xor Mem[Dir]		XOR A,(var1)
	A,(B)	A=A xor Mem[B]		-
	(Dir)	Mem[Dir]=A xor B		XOR (var1)
SHL	A,(Dir)	A=shift left Mem[Dir]		SHL A,(var1)
	A,(B)	A=shiflt left Mem[B]		-
	(Dir)	Mem[Dir]=shift left A		SHL (var1)
SHR	A,(Dir)	A=shift right Mem[Dir]		SHR A,(var1)
	A,(B)	A=shiflt right Mem[B]		-
	(Dir)	Mem[Dir]=shift right A		SHR(var1)
INC	B	B=B+1		-

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.

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2343 Arquitectura de Computadores

Computador básico

1. Motivación

Para que una máquina programable pueda ser referida como un computador, es necesario agregarle la capacidad de controlar el flujo de un programa para permitir que este realice decisiones e iteraciones. Debemos incorporar este elemento tanto de nivel del hardware del computador, como a nivel de software, para poder desarrollar programas complejos y completar un computador básico funcional.

2. Salto incondicional

2.1. Instrucción de salto incondicional

El código presentado en la tabla 1, genera la serie de Fibonacci almacenando los números alternadamente en los registros A y B. Para aumentar la cantidad de números de la secuencia, con las instrucciones y el computador que llevamos construido, la única alternativa es repetir las instrucciones de suma cuantas veces lo necesitemos.

Dirección	Instrucción	Operandos	A	B
0x00	MOV	A,0	**0**	?
0x01	MOV	B,1	0	**1**
0x02	ADD	A,B	**1**	1
0x03	ADD	B,A	1	**2**
0x04	ADD	A,B	**3**	2
0x05	ADD	B,A	3	**5**
0x06	ADD	A,B	**8**	5
0x07	ADD	B,A	8	**13**

Tabla 1: Programa que genera secuencia limitada de Fibonacci.

Una alternativa a esto sería tener la capacidad de saltar a una instrucción previa de manera de poder repetirla, sin perder el valor de los registros. Para poder lograr esto, debemos agregar una instrucción que permita saltar hacia otra instrucción, indicándole la dirección de memoria donde está almacenada. Esta instrucción se conoce como **salto incondicional** y en nuestro assembly ocuparemos el nombre `JMP` («jump» o salto) para referirnos a ella.

En la tabla 2, se observa el mismo programa antes descrito, pero esta vez con la instrucción de salto incondicional. Dado que el salto ocurre siempre, **este programa nunca se detiene** y por

tanto es capaz de generar la serie de Fibonacci hasta el límite del rango disponible por los registros (i.e. 255).

Dirección	Instrucción	Operandos
0x00	MOV	A,0
0x01	MOV	B,1
0x02	ADD	A,B
0x03	ADD	B,A
0x04	JMP	0x02

Tabla 2: Programa que genera secuencia «infinita» de Fibonacci.

Para utilizar la instrucción de salto incondicional, debemos conocer la dirección de memoria a la cual vamos a saltar, lo que en programas largos puede ser complejo. Para facilitar esto se utiliza el concepto de **label** que es un indicador que se puede agregar en una línea del código assembly para referirse a la dirección de memoria asociada a esa línea. De esta manera, podemos reescribir el programa anterior de la siguiente forma:

Dirección	Label	Instrucción	Operandos
0x00		MOV	A,0
0x01		MOV	B,1
0x02	start:	ADD	A,B
0x03		ADD	B,A
0x04		JMP	start

Tabla 3: Programa que genera secuencia «infinita» de Fibonacci con label para salto.

2.2. Implementación en hardware

Para implementar la instrucción de salto incondicional en hardware, se debe tener la capacidad de modificar la instrucción que se va ejecutar. La información de que instrucción se ejecuta está en el program counter, por lo que para agregar la capacidad de salto incondicional es necesario agregar la capacidad de cargar el program counter con el parámetro de la instrucción. Par esto es necesario agregar una nueva señal del control *Lpc* que indicará si el program counter está en modo carga $Lpc = 1$ o en modo incremento $Lpc = 0$. La figura 1 muestra el diagrama del computador incluyendo la capacidad de salto incondicional.

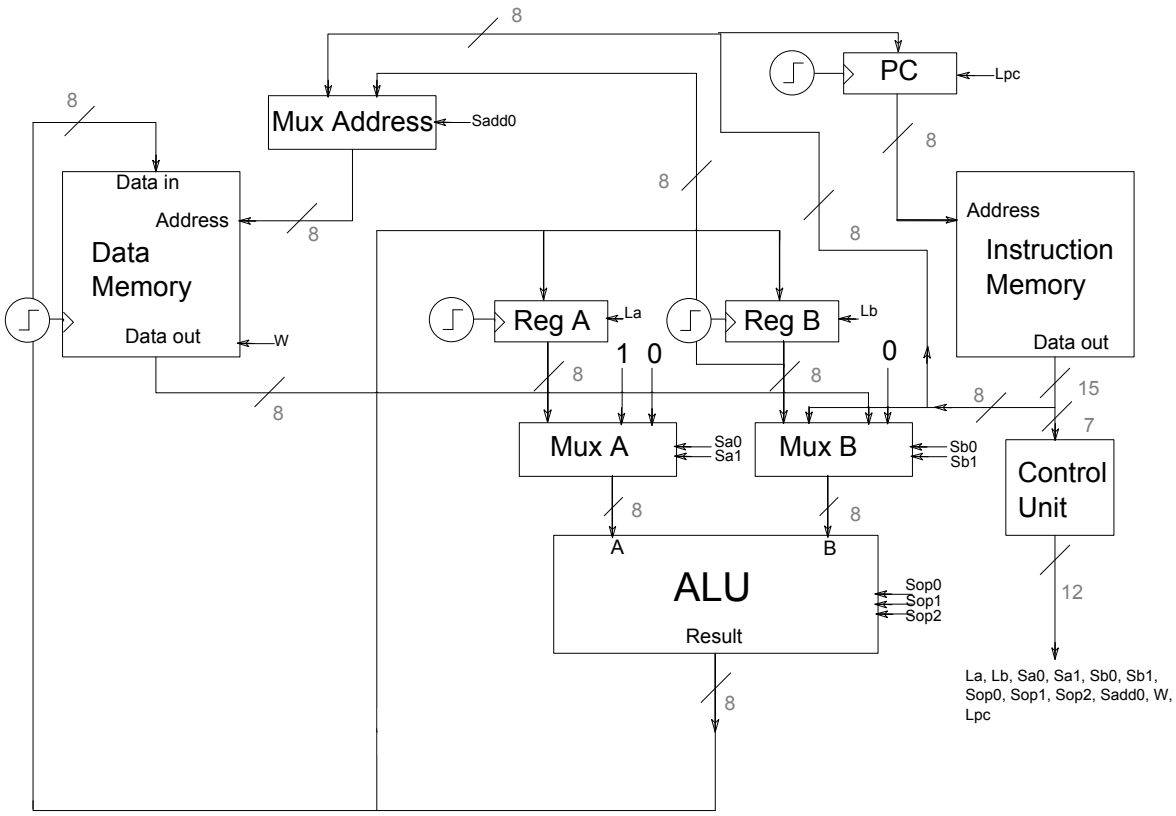


Figura 1: Computador con salto incondicional.

3. Salto condicional

3.1. Instrucciones de salto condicional para comparación

El salto incondicional no es suficiente para entregar la capacidad de decisión al computador. Para lograr esto es necesario agregar la opción de saltar condicionalmente, dependiendo de alguna condición. Un primer tipo de instrucciones de salto condicional corresponde a instrucciones que salten dependiendo del resultado de una comparación aritmética: $a == b$, $a! = b$, $a > b$, $a < b$,$a \geq b$,$a \leq b$.

Para lograr esto, es necesario primero agregar una instrucción que permita ejecutar la comparación, para lo cual es necesario reescribir las comparaciones aritméticas de la siguiente forma $a - b == 0$, $a - b! = 0$, $a - b > 0$, $a - b < 0$,$a - b \geq 0$,$a - b \leq 0$. Se observa que en todos los casos, la operación de comparación corresponde a restar el valor de a con el de b, por lo cual agregaremos la instrucción `CMP A,B` que ejecuta la resta entre los registros A y B y **no** almacena el resultado.

El siguiente paso luego de ejecutar la comparación es identificar si el resultado fue 0 o no, para los casos de igualdad, y si fue negativo o no, para los casos de mayor y menor que. De esta forma, se agregarán las siguientes instrucciones:

- `JEQ`: «Jump equal», salta en caso de igualdad $a == b$ es decir cuando el resultado de la ultima operación fue cero ($Z = 1$).
- `JNE`: «Jump not equal», salta en caso de desigualdad $a! = b$ es decir cuando el resultado de la

ultima operación fue distinto cero ($Z = 0$).

- `JGT`: «Jump greater than», salta en caso de que $a > b$ es decir cuando el resultado de la ultima operación no fue cero ni negativo ($Z = 0, N = 0$).
- `JLT`: «Jump less than», salta en caso de de que $a < b$ es decir cuando el resultado de la ultima operación fue negativo ($N = 1$).
- `JGE`: «Jump greater or equal than», salta en caso de $a \geq b$ es decir cuando el resultado de la ultima operación no fue negativa ($N = 0$).
- `JLE`: «Jump less or equal than», salta en caso de $a \leq b$ es decir cuando el resultado de la ultima operación fue cero o negativa ($Z = 1, N = 1$).

La tabla con las instrucciones en detalle se presenta a continuación:

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

Tabla 5: Instrucciones de salto condicional para comparación.

Con estas instrucciones, ya es posible desarrollar programas más complejos como por ejemplo un programa simple de multiplicación, como se observa en la tabla 6.

Dirección	Label	Instrucción	Operandos
0x00		MOV	A,3
0x01		MOV	B,5
0x02	mult:	ADD	B,5
0x03		SUB	A,1
0x04		CMP	A,1
0x05		JNE	mult

Tabla 6: Programa que multiplica 3×5.

3.2. Condition codes para comparación

Para poder ejecutar las comparaciones anteriores se observa que se requiere saber si el resultado de la operación anterior fue cero o si fue negativa. Esta es información que se puede obtener de la ALU, y se conoce como **condition codes** o códigos de condición:

- Zero (**Z**): El código de condición cero (Z por su nombre en ingles) se puede obtener a partir del resultado de la ALU, haciendo un **or** entre todos los bits y luego un **not** de la salida del or. De esta forma, sólo se obtendrá una salida final de 1 cuando todos los bits del resultado de la ALU sean 0.

- Negative (**N**): El código de condición negativo se puede obtener de manera simple, tomando el bit más significativo del resultado. Dado que las operaciones se realizan en complemento a 2, si el bit más significativo es 1, será un número negativo, si es 0, positivo.

3.3. Instrucciones de salto condicional por excepción

Un tipo particular de instrucciones de salto condicional son aquellas cuando ocurre algún tipo de caso de excepción. En particular nos interesa saber cuando la operación realizada resultó en un número que sobrepasa la representación válida por el tamaño del resultado. Hay dos casos a considerar: en operaciones de número si signos, cuando ocurre un **carry** se está indicando que el resultado real no es el que queda almacenado en la salida resultado; en operaciones de número con signo, cuando ocurre un **overflow** es decir cuando una operación que debía resultar en un número positivo resulta en un número negativo o viceversa. Las instrucciones para saltar en estos casos son:

- `JCR`: «Jump carry», salta en caso de que ocurra un carry ($C = 1$).
- `JOV`: «Jump overflow», salta en caso de que ocurra un overflow ($V = 1$).

El detalle se las instrucciones se presenta a continuación:

Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
JCR	Dir	PC = Dir	C=1	JCR label
JOV	Dir	PC = Dir	V=1	JOV label

Tabla 7: Instrucciones de salto condicional por excepción.

Con estas instrucciones es posible implementar programas que identifiquen cuando ocurrió un carry o un overflow y que realicen algún mecanismo que maneje estas condiciones excepcionales, como se observa en las tablas 8 y 9.

Dirección	Label	Instrucción	Operandos
0x00	start:	MOV	A,0
0x01		MOV	B,1
0x02	acum:	ADD	A,B
0x03		JOV	exc
0x04		JMP	acum
0x05	exc:	JMP	start

Tabla 8: Programa que maneja excepción de overflow.

Dirección	Label	Instrucción	Operandos
0x00	start:	MOV	A,0
0x01		MOV	B,1
0x02	acum:	ADD	A,B
0x03		JC	exc
0x04		JMP	acum
0x05	exc:	JMP	start

Tabla 9: Programa que maneja excepción de carry.

3.4. Condition Codes para excepciones

Al igual que en los casos de comparación, se necesita que la ALU entregue la información de carry u overflow:

- Carry (**C**): el bit de carry es simplemente la salida «carry out» del sumador restador, ya que solo con estas operaciones puede ocurrir un carry.
- Overflow (**V**): el bit de overflow es más complejo que el carry, ya que ocurre en distintas circunstancias dependiendo del signo de las entradas, la operación y de la salida. A continuación se presentan los casos posibles para que ocurra un overflow:

Operación	A	B	Resultado	Ejemplo (1 byte)
$A + B$	≥ 0	≥ 0	< 0	$127 + 4 = -125$
$A + B$	< 0	< 0	≥ 0	$-127 + -4 = 125$
$A - B$	≥ 0	< 0	< 0	$127 - -4 = -125$
$A - B$	< 0	≥ 0	≥ 0	$-127 - 4 = 125$

Tabla 10: Posibles casos de overflow.

Para implementar eso, es necesario agregar un circuito combinacional a la ALU que dependiendo de las entradas, operación y salidas entregue un 1 o un 0 para indicar el overflow.

3.5. Implementación en hardware

La implementación en hardware del salto condicional requiere, como se señaló anteriormente, que la ALU entregue los 4 bits de los condition codes, agregando las compuertas descritas previamente. Además de esto, dado que el salto se realiza con las condiciones ocurridas en la operación anterior, es necesario almacenar los condition codes en un registro, habitualmente denominado **status register**. Este registro se conecta a su vez a la unidad de control, la cual se encargará de determinar si se activa la señal *Lpc* dependiendo si se cumple o no la condición asociada a la instrucción de salto correspondiente.

El diagrama del computador con la capacidad de salto condicional se muestra a continuación.

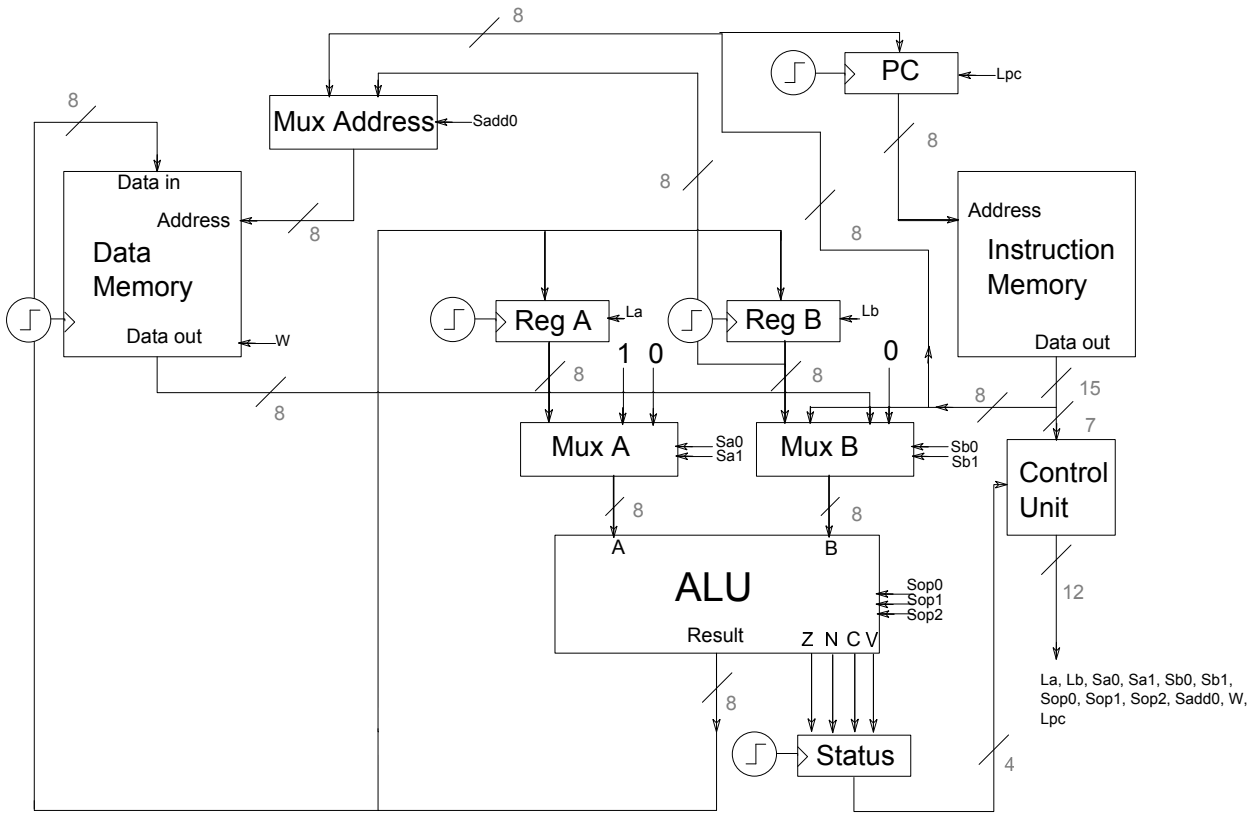


Figura 2: Computador con salto condicional.

4. Resumen instrucciones de salto

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

Tabla 11: Instrucciones de salto del computador básico.

5. Subrutinas

Supongamos que se desea escribir un código en assembly para calcular el producto punto entre un vector (a, b) y un vector (c, d). Por definición, el producto punto se calcula como: $(a, b) \cdot (c, d) = ac+bd$. Para realizar la función entonces necesitamos realizar dos multiplicaciones, un pseudocódigo en lenguaje de alto nivel para esta operación se muestra a continuación:

```
byte[] vector1 = new byte[]{2,3};
byte[] vector2 = new byte[]{4,5};
byte prodPunto = 0;

prodPunto += mult(vector1[0], vector2[0]);
prodPunto += mult(vector1[1], vector2[1]);
```

Como se vio anteriormente, con el computador básico y las instrucciones vistas ya es posible desarrollar un programa que multiplique dos números como el siguiente:

```
DATA:
    var1    3
    var2    4
    res     0
    i       0
CODE:
    start:  MOV A,(res)
            ADD A,(var2)
            MOV (res),A
            MOV A,(i)
            ADD A,1
            MOV (i),A
            MOV B,(var1)
            CMP A,B
            JLT start
```

Una opción para utilizar este código en el cálculo del producto punto sería reescribirlo dos veces, para las dos multiplicaciones. El problema de esto es que por un lado, debemos adaptar el código a cada multiplicación, y además escribirlo dos veces, es decir, desperdiciar espacio en al memoria de instrucciones. Una mejor alternativa sería poder reutilizar el mismo código de la multiplicación para las dos multiplicaciones del producto punto sin tener que reescribirlo, es decir poder llamar al trozo de código de la multiplicación durante la ejecución del programa del producto punto, lo que se conoce como una **subrutina**.

Existen tres elementos necesarios para poder implementar una subrutina: poder entregarle **parámetros**, poder recibir un **valor de retorno** y poder hacer la **llamada a la subrutina**, es decir poder saltar a la subrutina y volver luego de que termine a una **dirección de retorno** que apunte a la siguiente instrucción del programa.

```
DATA:
      vector1     2
                  3
      vector2     4
                  5
      prodPunto   0

CODE:
      ADD prodPunto, MULT(vector1[0],vector2[0])
      ADD prodPunto, MULT(vector1[0],vector2[0])
```

5.1. Parámetros

Para poder entregarle parámetros a la subrutina, es necesario almacenarlos en algún lugar al cual está pueda acceder. En el computador básico existen dos lugares donde se pueden almacenar datos: **registros** y **variables en memoria**.

- **Registros**: una primera opción para pasar parámetros es almacenarlos en los registros y que luego la subrutina se encargue de obtenerlos. Para lograr esto, se debe saber a priori que registros se ocuparán para que parámetros, de manera de cargar los que efectivamente ocupará la subrutina. Aunque está implementación es simple, la principal desventaja está en el número limitado de registros, que impide entregar más parámetros que los registros disponibles.
- **Variables**: otra opción para el paso de parámetros es almacenarlos en memoria y que la subrutina los obtenga de ahí. Al igual que en el caso de los registros, se debe saber a priori que variables se ocuparán para que parámetros, de manera de cargar las que efectivamente ocupará la subrutina. La ventaja respecto a los registros están en la mayor disponibilidad de espacio.

5.2. Valor de retorno

Al igual que con los parámetros, el valor de retorno debe ser almacenado en algún lugar que pueda ser accedido tanto por la subrutina, es decir, **registros** y **variables en memoria**.

- **Registros**: al igual que con los parámetros, para ocupar registros como valor de retorno hay que saber a priori que registro ocupará la subrutina. La diferencia con los parámetros es que como el retorno es sólo un valor, es factible realizar esto, a pesar de tener pocos registros.
- **Variables**: otra opción para el retorno es ocupar una variable en memoria. Al igual que en el caso de los registros, se debe saber a priori que variables se ocuparán para que parámetros, de manera de cargar las que efectivamente ocupará la subrutina.

5.3. Llamada y dirección de retorno

Además de poder entregar parámetros y recibir retorno del parte de la subrutina, debemos de alguna forma pasar de la ejecución del código principal a la ejecución del código de esta. Supongamos el siguiente código que realiza el producto punto y ocupa paso de parámetro y retorno ocupando variables:

```
DATA:
      vector1     2
                  3
      vector2     4
                  5
      prodPunto   0
      var1        0
      var2        0
      res         0
      i           0

CODE:
                  MOV A, (vector1)
                  MOV (var1), A
```

```
                    MOV A, (vector2)
                    MOV (var2), A
                    //Llamar a subrutina y retornar
                    MOV A, (prodPunto)
                    ADD A, (res)
                    MOV (prodPunto), A

                    MOV B, vector1
                    INC B
                    MOV A, (B)
                    MOV (var1), A
                    MOV B, vector2
                    INC B
                    MOV A, (B)
                    MOV (var2), A
                    //Llamar a subrutina y retornar
                    MOV A, (prodPunto)
                    ADD A, (res)
                    MOV (prodPunto), A

                    JMP end

mult:
start:      MOV A,(res)
            ADD A,(var2)
            MOV (res),A
            MOV A,(i)
            ADD A,1
            MOV (i),A
            MOV B,(var1)
            CMP A,B
            JLT start

end:
```

El primer paso para poder acceder al código de la subrutina es que se encuentre en la memoria de instrucciones. En el ejemplo anterior se observa que el código de la subrutina se incluyó al final, y se agregó un label `mult` para indicar el inicio de la subrutina. Llamar a la subrutina, entonces, se puede hacer simplemente saltando al label correspondiente con una instrucción `JMP mult`. El problema está en el retorno: al finalizar la ejecución de la subrutina, ¿cómo se sabe a donde volver? No es posible agregar otro JMP, ya que no se puede saber a que dirección se va a retornar (depende de donde se llamo a la subrutina). Para poder realizar correctamente el retorno es necesario **almacenar el valor del program counter** antes de realizar la llamada, para luego volver a cargarlo luego de retornar de la subrutina. En realidad, lo que nos interesa es guardar el valor del program counter incrementado en 1, dado que al volver de la subrutina queremos ejecutar la siguiente instrucción, no la actual.

Para lograr la llamada a la subrutina, entonces, debemos agregar el soporte de hardware necesario que permita almacenar el valor del program counter incrementado en 1 en memoria (al hacer la llamada) y para poder cargarlo desde memoria (al retornar), lo que se observa en la figura 3.

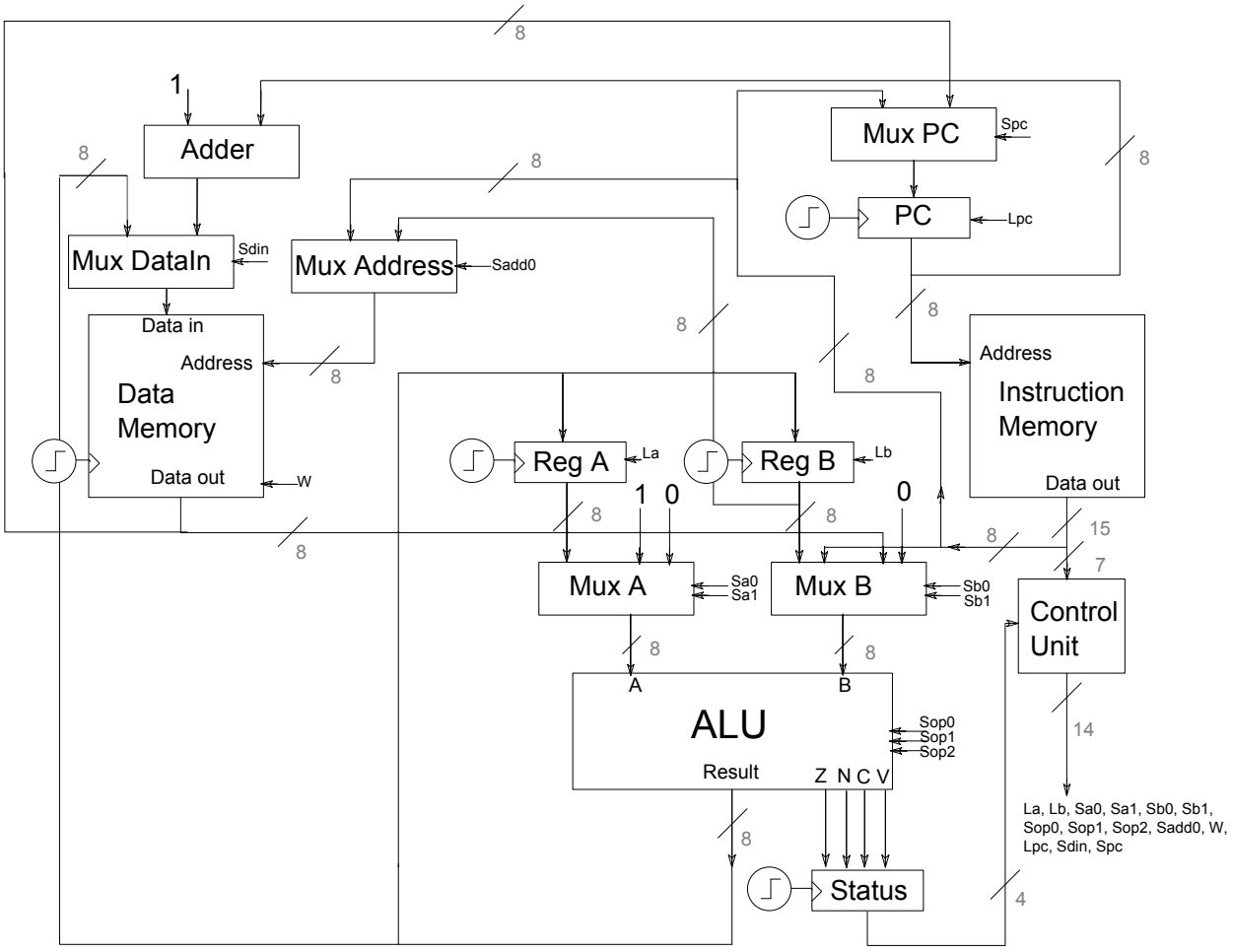


Figura 3: Almacenamiento del program counter

Con esta modificación de hardware podemos agregar dos instrucciones para realizar el llamado a subrutina y retorno de ellas: la instrucción `CALL dir` la cual almacena el PC en memoria y salta al label *dir* donde estará almacenada la subrutina; y la instrucción `RET` la cual se debe agregar al final de la subrutina y salta de vuelta a la dirección almacenada previamente del PC.

Con estas dos instrucciones se puede reescribir el código completo del programa que calcula el producto punto de la siguiente forma:

```
DATA:
      vector1     2
                  3
      vector2     4
                  5
      prodPunto   0
      var1        0
      var2        0
      res         0
      i           0

CODE:
                  JMP init
      mult:
                  MOV A, 0
```

```
                MOV (res), A
                MOV A, 0
                MOV (i), A
start:          MOV A,(res)
                ADD A,(var2)
                MOV (res),A
                MOV A,(i)
                ADD A,1
                MOV (i),A
                MOV B,(var1)
                CMP A,B
                JLT start
                RET

init:           MOV A, (vector1)
                MOV (var1), A
                MOV A, (vector2)
                MOV (var2), A
                CALL mult
                MOV A, (prodPunto)
                ADD A, (res)
                MOV (prodPunto), A

                MOV B, vector1
                INC B
                MOV A, (B)
                MOV (var1), A
                MOV B, vector2
                INC B
                MOV A, (B)
                MOV (var2), A
                CALL mult
                MOV A, (prodPunto)
                ADD A, (res)
                MOV (prodPunto), A
```

6. Almacenamiento de la dirección de retorno

6.1. Stack pointer

Para completar el manejo de subrutinas falta aún un elemento: en que parte de la memoria se guarda el valor del PC al hacer el llamado a la subrutina. Una opción es que el programador se encargara de indicar una dirección (por ejemplo almacenándola en el registro B de direccionamiento indirecto) previo a la llamada y previo al retorno. El problema de esto es que agrega complejidad a la programación.

Para solucionar este problema, se agrega un nuevo registro al computador denominado **stack pointer** o SP el cual comienza cargado con el valor de la **última dirección de la memoria**. La idea será entonces ocupar esa dirección para realizar la carga del program counter al ejecutar la instrucción `CALL`. Luego, al llamar a la instrucción `RET` se leerá el valor de memoria apuntado por el stack pointer y se cargará en el program counter para volver al flujo normal del programa.

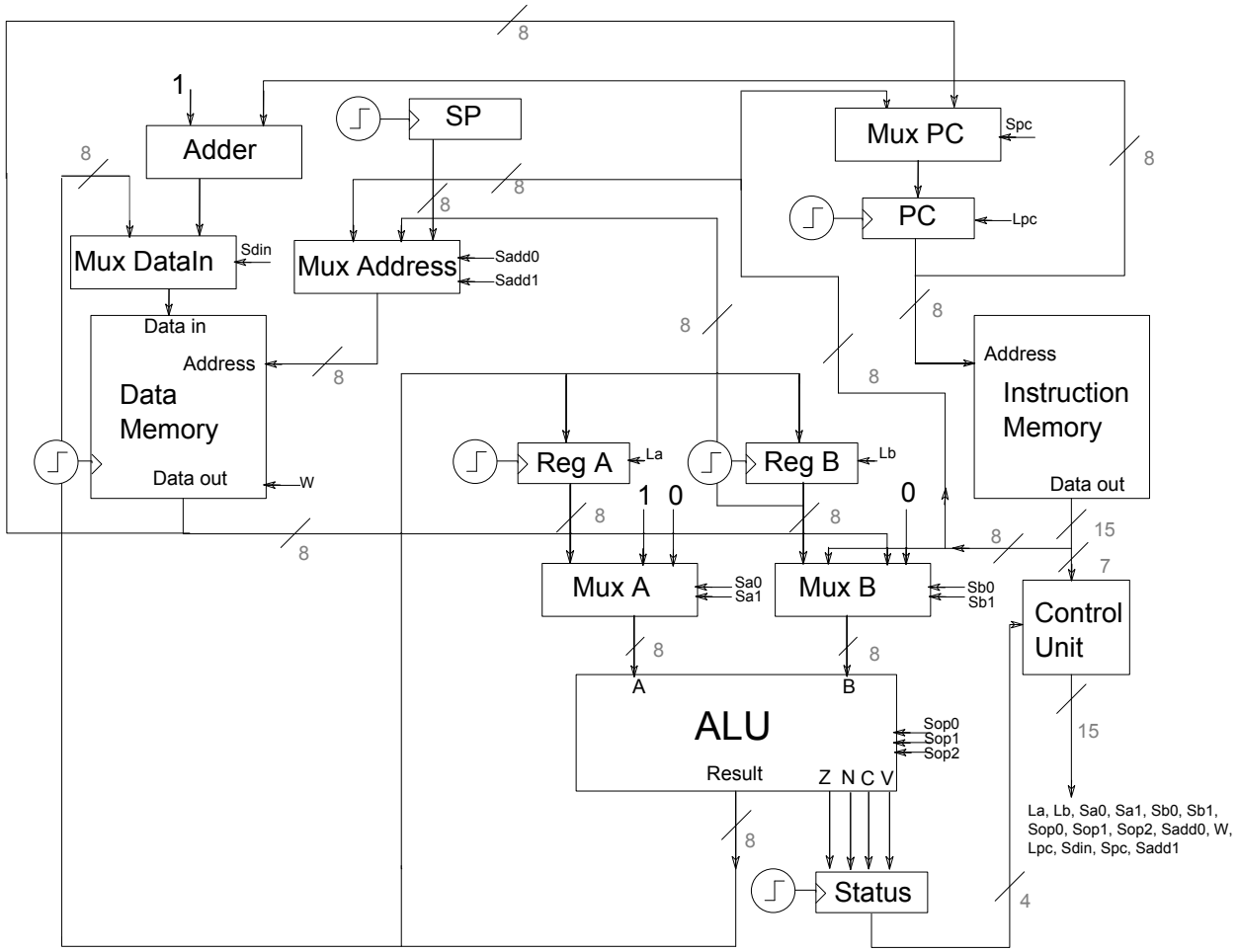


Figura 4: Registro stack pointer agregado al computador básico

6.2. Subrutinas anidadas

El tener un registro con un valor fijo para almacenar la dirección de retorno simplifica el manejo de llamadas a subrutinas simples. Sin embargo, persiste aún un problema, que se puede observar en un código como el siguiente:

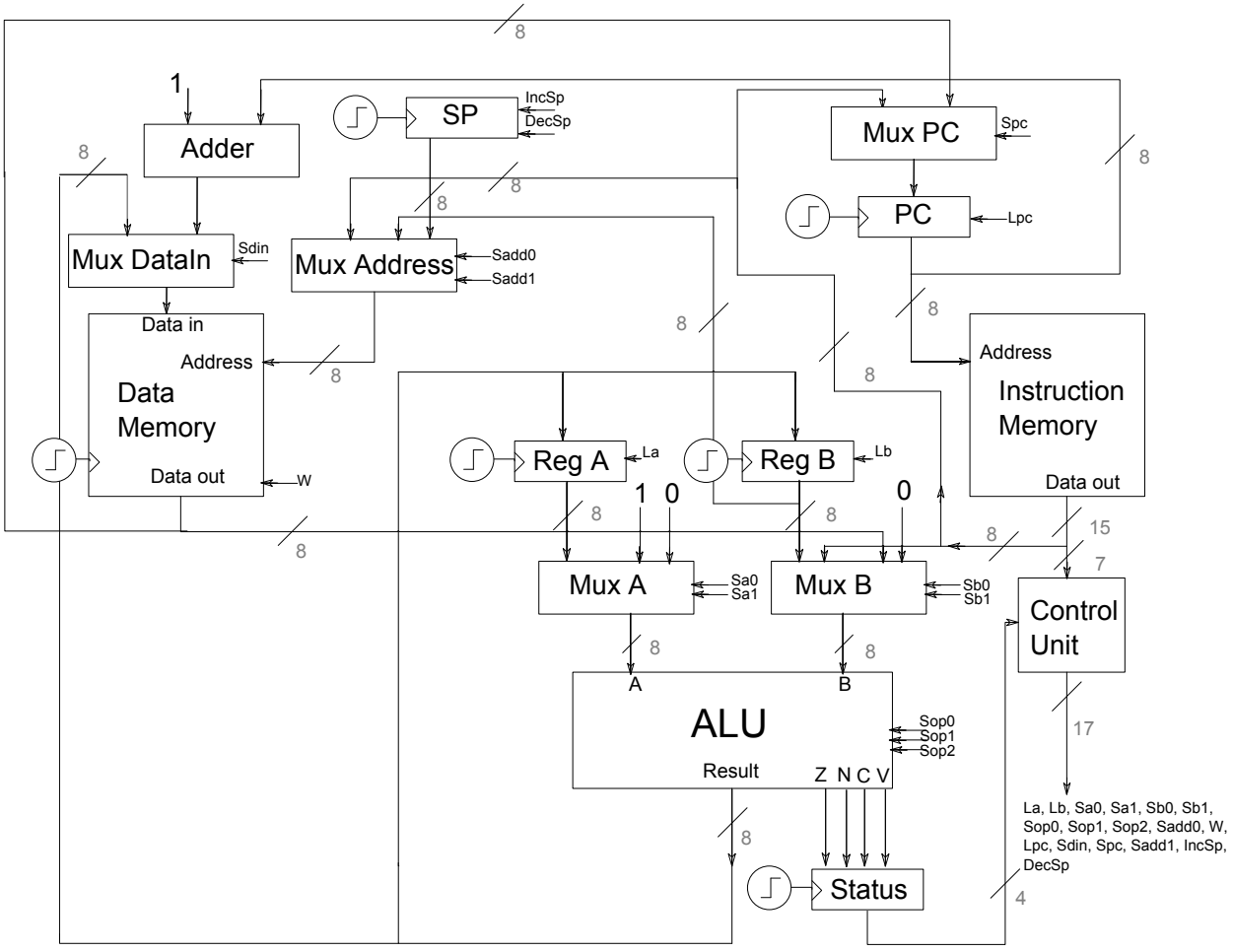
```
                ...
                ...
                CALL subrutina1
                ...
                ...
subrutina1:
                CALL subrutina2
                ...
                RET
                ...
                ...
subrutina2:     ...
                ...
                RET
```

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

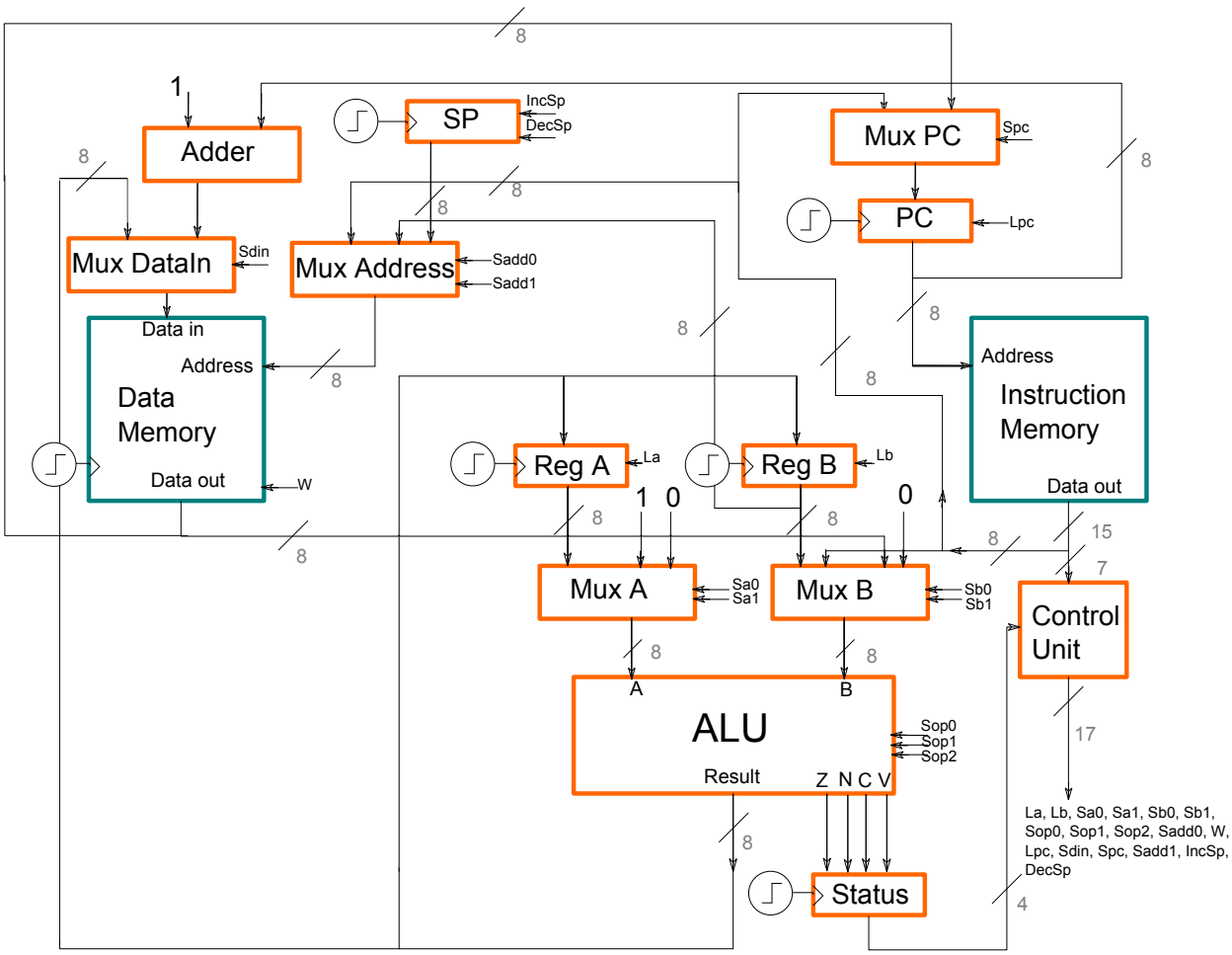
IIC2343 Arquitectura de Computadores

Arquitectura Computador Básico

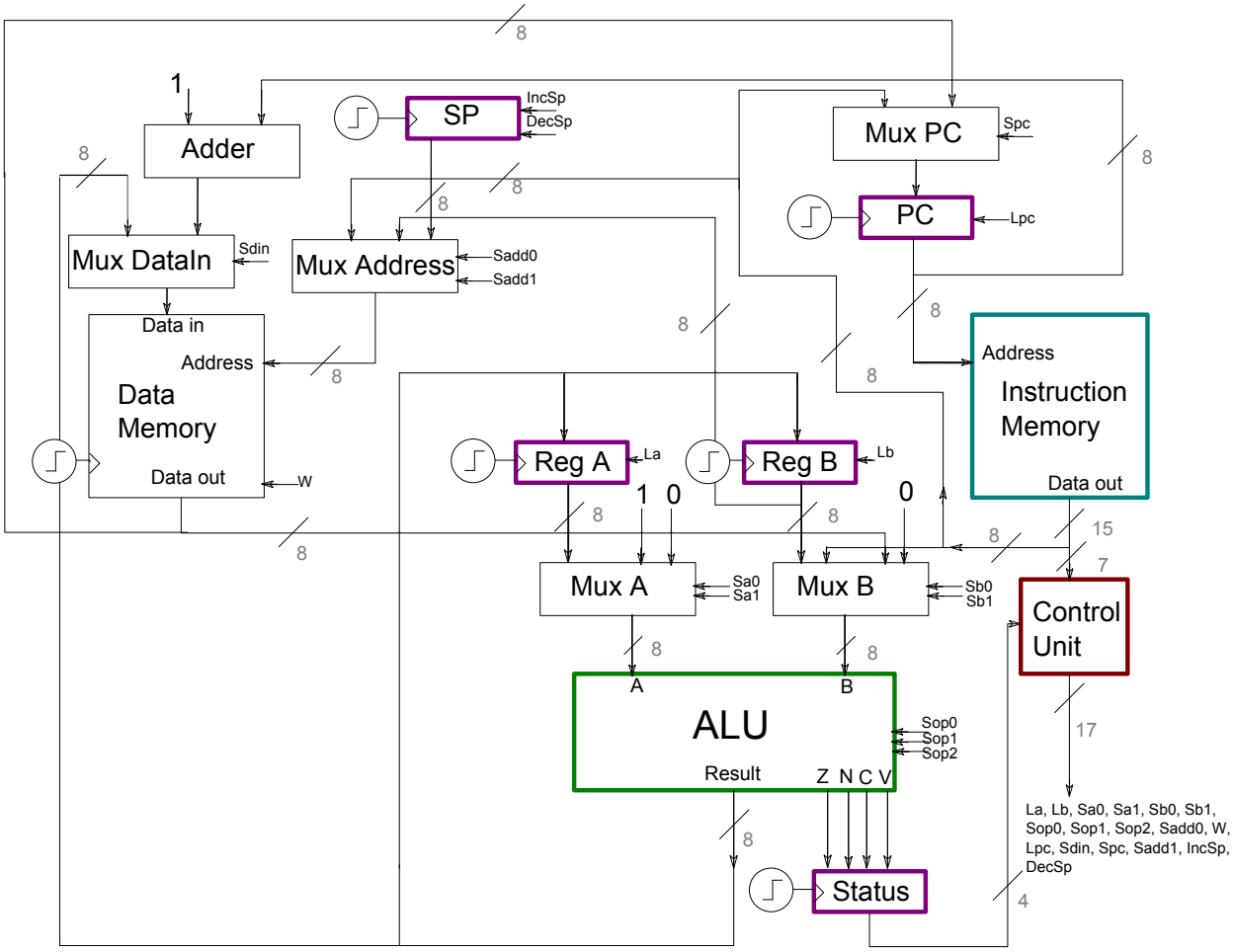
1. Microarquitectura Computador Básico



1.1. Partes del Computador Básico



Registros, Unidad de ejecución, Unidad de control



2. Set de instrucciones computador básico

2.1. Instrucciones de carga, aritméticas y lógicas

Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
MOV	A,B	A=B		-
	B,A	B=A		-
	A,Lit	A=Lit		MOV A,15
	B,Lit	B=Lit		MOV B,15
ADD	A,B	A=A+B		-
	B,A	B=A+B		-
	A,Lit	A=A+Lit		ADD A,5
SUB	A,B	A=A-B		-
	B,A	B=A-B		-
	A,Lit	A=A-Lit		SUB A, 2
AND	A,B	A=A and B		-
	B,A	B=A and B		-
	A,Lit	A=A and Lit		AND A,15
OR	A,B	A=A or B		-
	B,A	B=A or B		-
	A,Lit	A=A or Lit		OR A,5
NOT	A,A	A=notA		-
	B,A	B=notA		-
	A,Lit	A=notLit		NOT A,2
XOR	A,A	A=A xor B		-
	B,A	B=A xor B		-
	A,Lit	A=A xor Lit		XOR A,15
SHL	A,A	A=shift left A		-
	B,A	B=shift left A		-
	A,Lit	A=shift left Lit		SHL A,5
SHR	A,A	A=shift right A		-
	B,A	B=shift right A		-
	A,Lit	A=shift right Lit		SHR A,2

2.2. Instrucciones de salto y comparación

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

2.3. Instrucciones de memoria y direccionamiento

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		-

2.4. Instrucciones de subrutinas y stack

Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
CALL	Dir	Mem[SP] = PC + 1, SP–, PC = Dir		CALL func
RET		SP++		-
		PC = Mem[SP]		-
PUSH	A	Mem[SP] = A, SP–		-
PUSH	B	Mem[SP] = B, SP–		-
POP	A	SP++		-
		A = Mem[SP]		-
POP	B	SP++		-
POP		B = Mem[SP]		-

2.5. Set de instrucciones completo

Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
MOV	A,B	A=B		-
	B,A	B=A		-
	A,Lit	A=Lit		MOV A,15
	B,Lit	B=Lit		MOV B,15
	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,B	A=A+B		-
	B,A	B=A+B		-
	A,Lit	A=A+Lit		ADD A,5
	A,(Dir)	A=A+Mem[Dir]		ADD A,(var1)
	A,(B)	A=A+Mem[B]		-
	(Dir)	Mem[Dir]=A+B		ADD (var1)
SUB	A,B	A=A-B		-
	B,A	B=A-B		-
	A,Lit	A=A-Lit		SUB A, 2
	A,(Dir)	A=A-Mem[Dir]		SUB A,(var1)
	A,(B)	A=A-Mem[B]		-
	(Dir)	Mem[Dir]=A-B		SUB (var1)
AND	A,B	A=A and B		-
	B,A	B=A and B		-
	A,Lit	A=A and Lit		AND A,15
	A,(Dir)	A=A and Mem[Dir]		AND A,(var1)
	A,(B)	A=A and Mem[B]		-
	(Dir)	Mem[Dir]=A and B		AND (var1)
OR	A,B	A=A or B		-
	B,A	B=A or B		-
	A,Lit	A=A or Lit		OR A,5
	A,(Dir)	A=A or Mem[Dir]		OR A,(var1)
	A,(B)	A=A or Mem[B]		-
	(Dir)	Mem[Dir]=A or B		OR (var1)
NOT	A,A	A=notA		-
	B,A	B=notA		-
	(Dir)	Mem[Dir]=not A		NOT (var1)

Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
XOR	A,A	A=A xor B		-
	B,A	B=A xor B		-
	A,Lit	A=A xor Lit		XOR A,15
	A,(Dir)	A=A xor Mem[Dir]		XOR A,(var1)
	A,(B)	A=A xor Mem[B]		-
	(Dir)	Mem[Dir]=A xor B		XOR (var1)
SHL	A,A	A=shift left A		-
	B,A	B=shift left A		-
	(Dir)	Mem[Dir]=shift left A		SHL (var1)
SHR	A,A	A=shift right A		-
	B,A	B=shift right A		-
	(Dir)	Mem[Dir]=shift right A		SHR (var1)
INC	B	B=B+1		-
CMP	A,B	A-B		
	A,Lit	A-Lit		CMP A,0
JMP	Dir	PC = Dir		JMP end
JEQ	Dir	PC = Dir	Z=1	JEQ label
JNE	Dir	PC = Dir	Z=0	JNE label
JGT	Dir	PC = Dir	N=0 y Z=0	JGT label
JLT	Dir	PC = Dir	N=1	JLT label
JGE	Dir	PC = Dir	N=0	JGE label
JLE	Dir	PC = Dir	Z=1 o N=1	JLE label
JCR	Dir	PC = Dir	C=1	JCR label
JOV	Dir	PC = Dir	V=1	JOV label
CALL	Dir	Mem[SP] = PC + 1, SP–, PC = Dir		CALL func
RET		SP++		-
		PC = Mem[SP]		-
PUSH	A	Mem[SP] = A, SP–		-
PUSH	B	Mem[SP] = B, SP–		-
POP	A	SP++		-
		A = Mem[SP]		-
POP	B	SP++		-
		B = Mem[SP]		-

3. Señales de control

Instrucción	Operandos	Opcode	Condition	Lpc	La	Lb	Sa0,1	Sb0,1	Sop0,1,2	Sadd0,1	Sdin0	Spc0	W	IncSp	DecSp
MOV	A,B	0000000		0	1	0	ZERO	B	ADD	-	-	-	0	0	0
	B,A	0000001		0	0	1	A	ZERO	ADD	-	-	-	0	0	0
	A,Lit	0000010		0	1	0	ZERO	LIT	ADD	-	-	-	0	0	0
	B,Lit	0000011		0	0	1	ZERO	LIT	ADD	-	-	-	0	0	0
	A,(Dir)	0000100		0	1	0	ZERO	DOUT	ADD	LIT	-	-	0	0	0
	B,(Dir)	0000101		0	0	1	ZERO	DOUT	ADD	LIT	-	-	0	0	0
	(Dir),A	0000110		0	0	0	A	ZERO	ADD	LIT	ALU	-	1	0	0
	(Dir),B	0000111		0	0	0	ZERO	B	ADD	LIT	ALU	-	1	0	0
	A,(B)	0001000		0	1	0	ZERO	DOUT	ADD	B	-	-	0	0	0
	B,(B)	0001001		0	0	1	ZERO	DOUT	ADD	B	-	-	0	0	0
	(B),A	0001010		0	1	0	A	ZERO	ADD	B	ALU	-	1	0	0
ADD	A,B	0001011		0	1	0	A	B	ADD	-	-	-	0	0	0
	B,A	0001100		0	0	1	A	B	ADD	-	-	-	0	0	0
	A,Lit	0001101		0	1	0	A	LIT	ADD	-	-	-	0	0	0
	A,(Dir)	0001110		0	1	0	A	DOUT	ADD	LIT	-	-	0	0	0
	A,(B)	0001111		0	0	1	A	DOUT	ADD	B	-	-	0	0	0
	(Dir)	0010000		0	0	0	A	B	ADD	LIT	ALU	-	1	0	0
SUB	A,B	0010001		0	1	0	A	B	SUB	-	-	-	0	0	0
	B,A	0010010		0	0	1	A	B	SUB	-	-	-	0	0	0
	A,Lit	0010010		0	1	0	A	LIT	SUB	-	-	-	0	0	0
	A,(Dir)	0010011		0	1	0	A	DOUT	SUB	LIT	-	-	0	0	0
	A,(B)	0010100		0	1	0	A	DOUT	SUB	B	-	-	0	0	0
	(Dir)	0010101		0	0	0	A	B	SUB	LIT	ALU	-	1	0	0
AND	A,B	0010110		0	1	0	A	B	AND	-	-	-	0	0	0
	B,A	0010111		0	0	1	A	B	AND	-	-	-	0	0	0
	A,Lit	0011000		0	1	0	A	LIT	AND	-	-	-	0	0	0
	A,(Dir)	0011001		0	1	0	A	DOUT	AND	LIT	-	-	0	0	0
	A,(B)	0011010		0	1	0	A	DOUT	AND	B	-	-	0	0	0
	(Dir)	0011011		0	0	0	A	B	AND	LIT	ALU	-	1	0	0
OR	A,B	0011100		0	1	0	A	B	OR	-	-	-	0	0	0
	B,A	0011101		0	0	1	A	B	OR	-	-	-	0	0	0
	A,Lit	0011110		0	1	0	A	LIT	OR	-	-	-	0	0	0
	A,(Dir)	0011111		0	1	0	A	DOUT	OR	LIT	-	-	0	0	0
	A,(B)	0100000		0	1	0	A	DOUT	OR	B	-	-	0	0	0
	(Dir)	0100001		0	0	0	A	B	IR	LIT	ALU	-	1	0	0
NOT	A,A	0100010		0	1	0	A	-	NOT	-	-	-	0	0	0
	B,A	0100011		0	0	1	A	-	NOT	-	-	-	0	0	0
	(Dir)	0100111		0	0	0	A	B	NOT	LIT	ALU	-	1	0	0

Instrucción	Operandos	Opcode	Condition	Lpc	La	Lb	Sa0,1	Sb0,1	Sop0,1,2	Sadd0,1	Sdin0	Spc0	W	IncSp	DecSp
XOR	A,B	0101000		0	1	0	A	B	XOR	-	-	-	0	0	0
	B,A	0101001		0	0	1	A	B	XOR	-	-	-	0	0	0
	A,Lit	0101010		0	1	0	A	LIT	XOR	-	-	-	0	0	0
	A,(Dir)	0101011		0	1	0	A	DOUT	XOR	LIT	-	-	0	0	0
	A,(B)	0101100		0	1	0	A	DOUT	XOR	B	-	-	0	0	0
	(Dir)	0101101		0	0	0	A	B	XOR	LIT	ALU	-	1	0	0
SHL	A,A	0101110		0	1	0	A	-	SHL	-	-	-	0	0	0
	B,A	0101111		0	0	1	A	-	SHL	-	-	-	0	0	0
	(Dir)	0110011		0	0	0	A	B	SHL	LIT	ALU	-	1	0	0
SHR	A,A	0110100		0	1	0	A	-	SHR	-	-	-	0	0	0
	B,A	0110101		0	0	1	A	-	SHR	-	-	-	0	0	0
	(Dir)	0111001		0	0	0	A	B	SHR	LIT	ALU	-	1	0	0
INC	B	0111010		0	0	1	ONE	B	ADD	-	-	-	0	0	0
CMP	A,B	0111011		0	0	0	A	B	SUB	-	-	-	0	0	0
	A,Lit	0111100		0	0	0	A	LIT	SUB	-	-	-	0	0	0
JMP	Dir	0111101		1	0	0	-	-	-	-	-	LIT	0	0	0
JEQ	Dir	0111110	Z=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JNE	Dir	0111111	Z=0	1	0	0	-	-	-	-	-	LIT	0	0	0
JGT	Dir	1000000	N=0 y Z=0	1	0	0	-	-	-	-	-	LIT	0	0	0
JLT	Dir	1000001	N=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JGE	Dir	1000010	N=0	1	0	0	-	-	-	-	-	LIT	0	0	0
JLE	Dir	1000011	N=1 o Z=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JCR	Dir	1000100	C=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JOV	Dir	1000101	V=1	1	0	0	-	-	-	-	-	LIT	0	0	0
CALL	Dir	1000101		1	0	0	-	-	-	SP	PC	LIT	1	0	1
RET		1000110		0	0	0	-	-	-	-	-	-	0	1	0
		1000111		1	0	0	-	-	-	SP	-	DOUT	0	0	0
PUSH	A	1001000		0	0	0	A	ZERO	ADD	SP	ALU	-	1	0	1
PUSH	B	1001001		0	0	0	ZERO	B	ADD	SP	ALU	-	1	0	1
POP	A	1001010		0	0	0	-	-	-	-	-	-	0	1	0
		1001011		0	1	0	ZERO	DOUT	ADD	SP	ALU	-	0	0	0
POP	B	1001100		0	0	0	-	-	-	-	-	-	0	1	0
		1001101		0	0	1	ZERO	DOUT	ADD	SP	ALU	-	0	0	0

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2343 Arquitectura de Computadores

Arquitecturas de Computadores

1. Motivación

El computador básico visto hasta ahora contiene todos los elementos fundamentales de un computador. Sin embargo, existen una diversa variedad de computadores distintos que aunque mantienen este core de funcionalidades en común, presenta diferencias tanto a nivel de hardware como a nivel de software.

2. Arquitecturas de Computadores

Los distintos computadores existentes, aunque similares en los elementos fundamentales, presentan diversas diferencias en sus componentes de hardware y su definición de instrucciones. Las variaciones existentes dependerán del uso en que esté enfocado el computador particular, pudiendo incluir mayores funcionalidades o mejoras en la eficiencia. En general la arquitectura de un computador se puede caracterizar en base a dos elementos fundamentales: la **microarquitectura** y la **arquitectura del set de instrucciones o ISA**.

2.1. Microarquitecturas

La microarquitectura de un computador se refiere a los distintos componentes de hardware que estarán presente en un sistema computacional. Los elementos básicos de la microarquitectura de un computador se presentan a continuación. Además de estos elementos básicos, existe una variedad mayor de elementos diferenciadores que tienen que ver con mejorar la eficiencia de la comunicación entre las partes del computador, y en el procesamiento de información. Estos elementos irán siendo incluidos más adelante al modelo de computador visto hasta ahora.

- **Registros:** distintos computadores tendrán distinta cantidad y tamaño de registros. En el caso del computador básico visto hasta hora se tienen dos registros de 8 bits. Aumentar la cantidad de registros de un sistema permite reducir los traspasos de datos a memoria; aumentar el tamaño de los registros (y por consiguiente de la unidad de ejecución) permite realizar operaciones con números de mayor rango y precisión. Otro elemento diferenciador en el caso de los registros es la existencia de registros de propósito especial. En el caso del computador básico, el registro B es registro de operaciones y de direccionamiento, mientras que el registro A es sólo registro de operaciones.

- **Unidades de ejecución:** la unidad básica de ejecución de todo computador es la ALU y como tal estará presente en toda microarquitectura. Sin embargo, es posible modificar las funcionalidades de la ALU, por ejemplo eliminando los shifters, o agregando shifters con rotación, que en el caso de un **rotate left** vuelve a incluir el bit más significativo que fue desplazado en el bit menos significativo, y de manera inversa para un **rotate right**. También es posible agregar operaciones aritméticas más complejas, como multiplicación y división. Adicionalmente, se pueden incluir otras unidades de ejecución como una FPU e incluso varias unidades repetidas (varias ALU) por ejemplo para permitir realizar más cálculos de manera simultánea.

- **Unidad de control:** la unidad de control también puede presentar variaciones dependiendo del computador. En particular existen dos modelos de unidades de control principales: sistemas **hard-wired** en los cuales cada instrucción tiene asociado directamente las señales de control que ejecutan una función en el computador (como en el computador básico) y sistemas **micro-programados** o de **microcode**. En estos últimos sistemas, la unidad de control es de mayor complejidad, y cada instrucción del programa se traduce en un conjunto de microinstrucciones, que ejecutan subfunciones muy específicas.

- **Condition codes:** los condition codes presentes en un computador también variarán de sistema en sistema. Sistemas minimalistas trabajarán sólo con los condition codes Z y C por ejemplo. Condition codes adicionales a los del computador básico son los siguientes:

  - **Parity (P)**: indica si el resultado de la operación es par o impar, revisando el bit menos significativo (si ese bit es 1, el resultado es impar).
  - **Auxiliary carry (AC, DC o A)** indica si hubo carry en el primer *nibble* del número, es decir en los primeros 4 bits.

- **Stack:** la presencia de un stack para subrutinas también es un elemento que variará según el sistema computacional. Adicionalmente, algunos sistemas tendrán un stack manejado de manera independiente a la memoria de datos, y por tanto con una limitación de niveles de profundidad menor que si estuviese integrado en la memoria

- **Memorias:** la cantidad de palabras de las memorias y el tamaño de cada una de estas también será un elemento que variará entre distintos computadores. En el caso del computador básico visto hasta ahora, la memoria de datos tiene dirección de 8 bits es decir $2^8 = 256$ palabras, cada una de 8 bits. La memoria de instrucciones por su parte tiene dirección de 8 bits es decir $2^8 = 256$ palabras, cada una de 15 bits.

Una característica relevante del computador básico respecto a sus memorias, es la presencia de dos memorias separadas una de instrucciones y una de datos. Este paradigma arquitectónico se conoce como **arquitectura Harvard**, nombrado en honor al computador Havard Mark 1, que fue diseñado con esta arquitectura. Una alternativa a esto, es tener sólo una memoria, que tenga datos e instrucciones, lo que se conoce como **arquitectura Von Neumann**, nombrada en honor al matemático John Von Neumann quien ideo esta arquitectura. A continuación se presentan las características de ambos paradigmas, contextualizados en el computador básico y en microarquitecturas reales.

2.1.1. Arquitectura Harvard

La arquitectura Harvard se caracteriza por tener dos memorias, una de instrucciones y una de datos. Esta arquitectura presenta una serie de ventajas:

- Permite tener tamaños distintos de palabras de datos y de instrucciones.
- Permite tener tamaño de direcciones distintas para memoria de datos e instrucciones.
- Permite tener tecnologías distintas en las memorias. En el caso del computador básico, por ejemplo se ocupa una ROM para la memoria de instrucciones y una RAM para la de datos.

En general, esta arquitectura se utiliza principalmente en microcontroladores o en sistemas embebidos. A continuación se presenta un ejemplo de microarquitectura Harvard, el microcontrolador PIC16F877A:

Microcontrolador PIC16F877A

Características:

- **Registros:** el PIC16F877A tiene un solo registro de trabajo para realizar operaciones, denominado registro **W**. Adicionalmente cuenta con un registro de propósito específico para realizar direccionamiento indirecto, el registro **FSR**. Ambos registros son de 8 bits.
- **Unidades de ejecución:** el PIC16F877A tiene como única unidad de ejecución una *ALU* de 8 bits sin shift aritmético, pero con rotate left y right (shift lógico).
- **Unidad de control:** el mecanismo de control del PIC16F877A es hard-wired.
- **Condition codes:** los condition codes presentes en el PIC16F877A son el bit cero (Z), carry (C) y carry auxiliar (AC).
- **Stack:** el PIC16F877A tiene un stack independiente de la memoria de datos, de 8 niveles y palabras de 13 bits.
- **Memorias:** la memoria de instrucciones del PIC16F877A tiene dirección de 13 bits, es decir contiene $2^{13} = 8K$ palabras, cada una de 14 bits. La memoria de datos tiene dirección de 9 bits, es decir $2^9 = 512$ palabras de 8 bits. Parte de la memoria está reservada para funciones especiales, por lo que el tamaño efectivo es de 368 palabras. Esta memoria se caracteriza por estar subdividida en 4 regiones lógicas denominadas bancos. La ventaja de esto es que ocupando un selector de banco de 2 bits es posible direccionar la memoria con sólo 7 bits.

2.1.2. Arquitectura Von Neumann

El segundo paradigma arquitectónico predominante corresponde a la arquitectura Von Neumann. En esta arquitectura, tanto instrucciones como datos están almacenados juntos en una misma memoria. La gran ventaja de esto, es la capacidad de **poder crear programas para el computador, ocupando el propio computador**. En un computador con arquitectura Harvard los programas deben ser creados externamente y escritos en la ROM con mecanismos especiales. En un computador con arquitectura Von Neumann, en cambio, al poder trabajar los programas

como si fueran datos, permite crearlos directamente en el mismo equipo, y es por esto que esta es la arquitectura predominante en computadores de uso personal.

La arquitectura de Von Neumann tiene una implementación más simple que la arquitectura de Harvard, gracias a tener una sola vía de acceso para datos e instrucciones. Una desventaja, sin embargo, es el «cuello de botella» que se genera debido a no poder acceder a las instrucciones y datos en paralelo. Otra desventaja pasa por la seguridad: al tener las instrucciones y datos juntos, es teóricamente posible modificar en tiempo de ejecución un programa, pudiendo insertar código malicioso. Estás desventajas, no obstante, fueron considerada menores respecto a las ventajas de simpleza y programabilidad, por lo cual esta arquitectura es la más usada hoy en día.

La microarquitectura más importante que sigue el paradigma de Von Neumann es la de los procesadores Intel. A continuación se detallan las características del primer microporcesador Intel usado en computadores personales, el Intel 80186.

Intel 80186

Características:

- **Registros:** El Intel 80186 posee 4 registros de uso general: AX, BX, CX, DX, todos de 16 bits. Para estos registros, se permite también acceder a los sub registros *high* y low que corresponden a los 8 bits más y menos significativos respectivamente. De esta forma, el registro AX, puede ser trabajado como dos registros de 8 bits, AH y AL, lo mismo para BX (BH y BL), CX (CH y CL) y DX (DH y DL). Además cuenta con registros de uso específico también de 16 bits: SP y BP, para manejo del stack; SI y DI para direccionamiento indexado; y registros especiales para acceder a distintos segmentos de la memoria: CS, DS, ES y SS.
- **Unidades de ejecución:** El Intel 80186 cuenta con una ALU de 16 bits que tiene operaciones aritméticas y lógicas básicas, así como también multiplicación, división y shifts lógicos y aritméticos.
- **Unidad de control:** el mecanismos de control del Intel 80186 es una mezcla entre hard-wired y microcode.
- **Condition codes:** el Intel 80186 tiene los siguientes condition codes: carry (C), overflow (O), cero (Z), signo o negativo (S), carry auxiliar (A) y paridad (P).
- **Stack:** El Intel 80186 tiene stack integrado en la memoria y controlado con dos registros: el stack pointer (SP) y el base pointer (BP).
- **Memorias:** El Intel 80186 tiene un bus de direcciones de 16 bits y bus de datos de 16 bits.

2.2. Arquitectura del Set de Instrucciones (ISA)

El segundo elemento que define la arquitectura de un computador es la arquitectura del set de instrucciones o ISA. Las instrucciones de un computador definirán como se deben escribir programas, y se diferenciarán por los siguientes factores.

- **Tipos de instrucciones:** Dependiendo del ISA, existirán distintos tipos de instrucciones disponibles. En general los tipos de instrucciones mínimos que se soportan en un ISA son:

- Instrucciones de carga
- Instrucciones aritméticas
- Instrucciones lógicas y shifts
- Instrucciones de salto
- Instrucciones de subrutina

- **Tipos de datos:** Distintos ISA pueden definir distintos tipos de datos que son soportados.
- **Modos de direccionamiento:** Además de las instrucciones, el ISA debe definir que modos de direccionamiento son soportados por el sistema para generar la dirección efectiva (**effective address**) con la que se accederá a memoria. Algunos de los posibles modos de direccionamiento son los siguientes:

  - **Direccionamiento inmediato o literal:** la forma más simple de acceder a información en un computador es almacenar el dato en la instrucción misma como un literal que puede ser cargado en un registro o ocupando como operando, lo que se conoce como direccionamiento inmediato o literal.
  - **Direccionamiento por registros**: otra forma de acceder a la información del computador es accediendo a la información almacenada en los registros

  Aunque los modos anteriores son clasificados como modos de direccionamiento, el concepto se utiliza principalmente para referirse a como acceder a la información en la memoria de datos. Distintos computadores tendrán distintos modos de acceder a las palabras de la memoria, a continuación se presentan los modos principales usados:

  - **Direccionamiento directo:** el direccionamiento directo corresponde a indicar en la instrucción la dirección de memoria específica donde se encuentra el dato. Una instrucción que carga un valor de memoria en un registro, por ejemplo, contendrá el opcode que indica la carga, y además como parámetro, la dirección de la memoria de datos desde donde se obtendrá el valor. De esta forma, se está indicando **directamente** en la instrucción desde donde se obtendrá el valor de memoria:

Memoria de instrucciones

Dirección	Opcode	Parámetro
0x00	000100	**0x02**
0x01	...	...
0x02	...	...
0x03	...	...
0x04	...	...

Memoria de datos

Dirección	Palabra
0x00	Dato 0
0x01	Dato 1
0x02	Dato 2
0x03	Dato 3
0x04	Dato 4

- **Direccionamiento indirecto por registro:** el siguiente nivel de direccionamiento corresponde a no indicar directamente la dirección sino indicar en que registro se encuentra almacenada la dirección que se utilizará para acceder a un valor de memoria. Por ejemplo, si el registro B tiene almacenado el valor $0x03$, al ocupar direccionamiento indirecto por registro con el registro B estamos diciendo que el dato de memoria que queremos ocupar es el que está almacenado en la dirección que tiene almacenado el registro B, es decir, el dato en la dirección $0x03$:

Registros

Registro	Valor
A	...
B	**0x03**

Memoria de datos

Dirección	Palabra
0x00	Dato 0
0x01	Dato 1
0x02	Dato 2
0x03	Dato 3
0x04	Dato 4

- **Direccionamiento indirecto por registro base + offset:** el direccionamiento indirecto por registro puede ser extendido de distintas maneras. Una de estas maneras incluye que la dirección para obtener el dato se calcule ocupando el valor del registro como **base** y a esa base sumarle un **offset** o desplazamiento que puede venir incluido como parámetro en la instrucción. Por ejemplo si el registro B tiene almacenado el valor $0x01$ y el parámetro de la instrucción es $0x02$, la dirección apuntada es $0x01 + 0x02 = 0x03$:

Registros

Registro	Valor
A	...
B	**0x01**

Memoria de instrucciones

Dirección	Opcode	Parámetro
0x00	000101	**0x02**
0x01	...	...
0x02	...	...
0x03	...	...
0x04	...	...

Memoria de datos

Dirección	Palabra
0x00	Dato 0
0x01	Dato 1
0x02	Dato 2
0x03	Dato 3
0x04	Dato 4

- **Direccionamiento indirecto por registro base + registro índice:** otra manera de extender el direccionamiento indirecto por registro es ocupar dos registros para almacenar la direción: un registro base y un registro que almacena el offset, conocido como registro **índice**. Por ejemplo, si el registro B se ocupa como base y tiene el valor $0x01$ y el registro A se ocupa como índice y tiene el valor $0x03$, al dirección indicada es $0x01 + 0x03 = 0x04$:

Registros

Registro	Valor
A	**0x03**
B	**0x01**

Memoria de datos

Dirección	Palabra
0x00	Dato 0
0x01	Dato 1
0x02	Dato 2
0x03	Dato 3
0x04	Dato 4

- **Direccionamiento indirecto por registro con post incremento:** el registro indirecto por registro puede ser extendido para automáticamente incrementar el valor del registro base en 1 lo que es útil para ir recorriendo una secuencia de valores relacionados de palabras de memoria, lo que se conoce como un **arreglo** de datos:

Registros

Registro	Valor actual	Valor siguiente
A	...	...
B	**0x00**	0x01

Memoria de datos

Dirección	Palabra
0x00	Dato Arreglo 0
0x01	Dato Arreglo 1
0x02	Dato Arreglo 2
0x03	Dato Arreglo 3
0x04	Dato Arreglo 4

- **Direccionamiento indirecto por registro con post decremento**: Similar al anterior, pero el valor del registro de dirección se decrementa luego de ser usado:

Registros

Registro	Valor actual	Valor siguiente
A	...	...
B	**0x04**	0x03

Memoria de datos

Dirección	Palabra
0x00	Dato Arreglo 0
0x01	Dato Arreglo 1
0x02	Dato Arreglo 2
0x03	Dato Arreglo 3
0x04	Dato Arreglo 4

- **Direccionamiento indirecto:** El direccionamiento indirecto extiende la idea del indirecto por registro, pero ocupando una palabra de memoria para almacenar la dirección. Este modo es similar al modo directo, en que se utiliza el parámetro de la instrucción para direccionar, pero la diferencia es que el parámetro indica la palabra en memoria que tiene la dirección del dato, es decir indica **indirectamente** como acceder al dato. Por ejemplo, si la instrucción tiene el parámetro $0x02$, ocupando direccionamiento indirecto indica que la palabra almacena en la dirección $0x02$ (por ejemplo $0x04$) corresponde a al dirección del dato requerido en la memoria:

Memoria de instrucciones

Dirección	Opcode	Parámetro
0x00	000101	**0x02**
0x01	...	...
0x02	...	...
0x03	...	...
0x04	...	...

Memoria de datos

Dirección	Palabra
0x00	Dato 0
0x01	Dato 1
0x02	0x04
0x03	Dato 3
0x04	Dato 4

- **Manejo del stack:** El manejo del stack variará según cada computador, lo que se ve reflejado en las instrucciones disponibles en el ISA. En general, el mínimo manejo de stack corresponde a usarlo para almacenar el valor del PC en los llamados a subrutinas. Adicionalmente, el stack puede ser usado como almacenamiento general para los registros y también como mecanismo de paso de parámetros y retorno de subrutinas.
- **Formato de la instrucción:** El formato de la instrucción específica la traducción en lenguaje

de máquina de una instrucción particular del set. Dependiendo del ISA, este formato será general para todas las instrucciones, o existirán distintos formatos específicos.

- **Palabras por instrucción:** La cantidad de palabras que se requieran para almacenar una instrucción variará en distintos ISAs, y también en un mismo ISA, distintas instrucciones puede tener distinto largo.
- **Ciclos por instrucción:** La cantidad de ciclos que se demora en ejecutar cada instrucción también variará en distintos ISAs, y también en un mismo ISA, distintas instrucciones puede tener distinto largo.

El ISA está estrechamente vinculado con el **lenguaje assembly**, pero no son equivalentes. Es posible definir varios assembly para un mismo ISA, ya que elementos como las secciones de datos, uso de labels y sintáxis numérica pueden ser definidos de manera distinta en distintos assembly.

Existen dos paradigmas principales para implementar ISAs: Reduce Instruccion Set Computer (RISC) y Complex Instruction Set Computer (CISC), los cuales se presentan a continuación.

2.2.1. Reduce Instruction Set Computer (RISC)

Los sets de instrucciones RISC fueron desarrollados con el objetivo de minimizar la complejidad del hardware del computador, simplificando la arquitectura y diseño de este. En general un computador con ISA RISC se califica como que tiene «énfasis en el software» ya que el hardware del computador sólo provee funcionalidades básicas, y las funcionalidades avanzadas son implementadas por software. Sus características principales son las siguientes:

- Instrucciones de un sólo ciclo de clock.
- Unidad de control hard-wired
- Formato de instrucción uniforme e idealmente almacenado en sólo una palabra.
- Registros de propósito general idénticos, que permiten todos realizar las mismas funciones.
- Modos de direccionamiento simples.
- Códigos en assembly largos.
- Pocos tipos de datos soportados directamente en hardware.

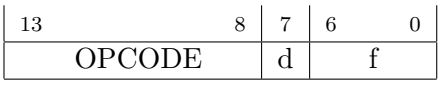
El computador básico tiene un ISA se puede clasificar como RISC, cumpliendo con todos los requisitos anteriores, salvo el hecho de tener registros de propósito general con las mismas funciones. El microcontrolador PIC16F877A también tiene un ISA que se puede clasificar como RISC. Sus características se describen a continuación.

ISA PIC16F877A

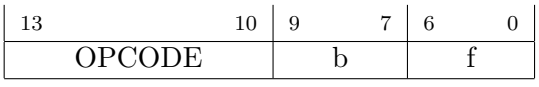
- **Tipos de instrucciones:**
  - Instrucciones de carga: Instrucciones especiales para cargar entre el registro W y memoria y para cargar un literal al registro W (ver modos de direccionamiento).

- Instrucciones aritméticas: Instrucciones para sumar y restar.
- Instrucciones lógicas y shifts: Instrucciones para implementar las operaciones lógicas AND, OR y XOR. Además cuaenta con instrucciones para realizar shift rotates.
- Instrucciones de salto: El PIC tiene dos tipos de instrucciones de salto: salto con comparación en un ciclo, implementado en las instrucciones `DECFSZ f,d` y `INCFSZ f,d` que decrementan o incrementan el valor de memoria guardado en f y se saltan la siguiente instrucción si $Z = 0$; salto con bit test, implementado en las instrucciones `BTFSC f,b` y `BTFSS f,b` que se saltan la siguiente instrucción si el bit b del valor de memoria guardado en f está en 0 («Clear») o están en 1 («Set») respectivamente.
- Instrucciones de subrutina:

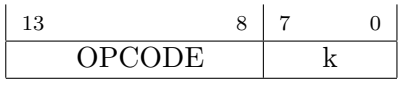
- **Tipos de datos:** Las variable sólo pueden ser de 8 bits, es decir de 1 byte, que es el tamaño de palabras de la memoria de datos.
- **Modos de direccionamiento:** El ISA implementa los siguientes modos de direccionamiento:
  - Direccionamiento inmediato: realizado mediante instrucciones especiales de la forma «inst»`LW`, como por ejemplo `MOVLW k`, que copia en el registro W el valor de k. Soporta literales de 8 bits.
  - Direccionamiento directo: realizado mediante instrucciones especiales de la forma «inst»`WF` para mover de W a f, como `MOVWF f`, e instrucciones de la forma «inst»`WF` para mover de f a W, como `MOVF f,d`. Debido a que se utiliza una instrucción especial, distinta a la del literal, no se ocupan elementos de sintáxis especial (como los paréntesis) para diferenciar la dirección de memoria del valor asociado a esta.
  - Direccionamiento indirecto por registro: el direccionamiento indirecto se implementa ocupando el registro especial FSR.
- **Manejo del stack:** Sólo para almacenar PC al hacer llamados a subrutinas con `CALL`, no se puede ocupar para almacenamiento general.
- **Formato de la instrucción:** Distintos tipos de instrucción tienen distintos formatos:
  - Byte-oriented:



  - Bit-oriented:



  - Literal:



- Saltos:

13 — 11	10 — 0
OPCODE	k

  Donde:
  - d representa el destino: $d = 0$ representa el registro W; $d = 1$ representa la dirección de memoria f
  - f representa la dirección de memoria de datos.
  - b representa un bit, comenzando en 0 desde el menos significativo.
  - k representa un literal.

- **Palabras por instrucción:** Una palabra de 14 bits por instrucción.
- **Ciclos por instrucción:** Salvo algunas excepciones, las instrucciones son de 1 ciclo.

A modo de ejemplo, el siguiente código de multiplicación en lenguaje de alto nivel:

```
byte var1 = 2;
byte var2 = 3;
byte res = 0;
byte i = var1;
do
{
  res += var2;
  i--;
}
while(i != 0);
```

que se escribe así en el assembly del computador básico:

```
DATA:
    var1    2
    var2    3
    res     0
    i       0
CODE:
            MOV A, (var1)
            MOV (i), A
    start:  MOV A,(res)
            ADD A,(var2)
            MOV (res),A
            MOV A,(i)
            SUB A,1
            MOV (i),A
            CMP A,0
            JNE start
```

se escribe de la siguiente forma en el assembly de PIC16F877A:

```
    var1    EQU H'20'
    var2    EQU H'21'
    res     EQU H'22'
    i       EQU H'23'
```

```
            MOVLW 2
            MOVWF var1
            MOVLW 3
            MOVWF var2
            MOVF var1,0
            MOVWF i
start:      MOVF  res,0
            ADDWF var2,0
            MOVWF res
            DECFSZ i,1
            GOTO start
```

2.2.2. Complex Instruction Set Computer (CISC)

Los sets de instrucciones CISC se caracterizan por tener muchas instrucciones y de alta complejidad, enfocándose en tener componentes de hardware específicos para implementar distintas funcionalidades. Los computadores con ISA CISC se califican con «énfasis en el hardware» ya que poseen diversos componentes de hardware para soportar instrucciones avanzadas, y habitualmente incluyen unidad de control con microcode.

Características

- Instrucciones de múltiples clock.
- Unidad de control con algún grado de microcode.
- Códigos cortos.
- Tipos de datos complejos implementados en hardware.

El ejemplo cásico de un set ISA es el set de instrucciones de las arquitecturas de Intel, denominado **x86**. A continuación se presentan sus características principales.

ISA Intel x86

- **Tipos de instrucciones:**
  - Instrucciones de carga: Instrucción general `MOV` para realizar transferencias ocupando distintos tipos de direccionamiento.
  - Instrucciones aritméticas: Instrucciones de suma y resta, además de instrucciones de multiplicación con y sin signo (`IMUL` y `MUL`) y división con y sin signo (`IDIV` y `DIV`).
  - Instrucciones lógicas: Instrucciones para implementar las operaciones lógicas AND, OR, XOR NOT. Además cuaenta con instrucciones para realizar shift rotates y shift normales.
  - Instrucciones de salto y subrutina: saltos condicionales, sin signo y con signo, además de saltos por excepciones (overlfow y carry). .
- **Tipos de datos:** Soporta variables del tamaño de la palabra de memoria (16 bits), lo que se conoce como tipo **word** y también variable de 1 byte (8 bits). Esto se puede debido a que los registros generales se pueden ocupar completos (e.g AX) o como dos registros de 8 bits (e.g

AH y AL). Un elemento importante a considerar debido a lo anterior, es que al definir una variable de un cierto tipo (byte o word), esa variable solo se podrá almacenar en un registro de tamaño equivalente.

- **Modos de direccionamiento:** El ISA x86 soporta una gran variedad de modos de direccionamiento, los cuales no requieren instrucciones especiales.
  - Direccionamiento inmediato: se implementa indicando directamente el literal como segundo parámetro: `MOV AX, 10`
  - Direccionamiento por registros: se implementa indicando los registros fuente como segundo parámetro y destino como primer parámetro: `MOV AX, BX`
  - Direccionamiento directo: se implementa usando la sintáxis especial de paréntesis cuadrados: `MOV AX, [var1]`. En este caso la variable `var1` tiene que haber sido definida del tipo word para que la instrucción sea válida.
  - Direccionamiento indirecto por registros: se implementa usando la sintáxis especial de paréntesis cuadrados: `MOV AX, [BX]`
  - Direccionamiento indexado con registro base y offset: se implementa usando la sintáxis especial de paréntesis cuadrados e indicando con una suma los registros a usar: `MOV AX, [BX + SI]`
  - Direccionamiento indexado con registro base, registro índice y offset: se implementa usando la sintáxis especial de paréntesis cuadrados e indicando con una suma los registros a usar y el offset: `MOV AX, [BX + SI + 2]`
- **Manejo del stack y sburutinas:** Stack usado para almacenar PC luego de llamado a subrutinas, para carga de registros generales individuales, carga de todos los registros de propósito general (instrucciones `PUSHA` y `POPA`) del registro de status (instrucciones `PUSHF` y `POPF`) y usado para paso de parámetro.
- **Formato de la instrucción:** El formato depende del tipo de instrucción. Los formatos más relevantes son:
  - Operaciones entre dos registros o registro y memoria con direccionamiento indirecto:

Opcode	Size	Opmode	Reg Dest/Address mode	Reg Src/Address mode

  - Operaciones entre registro y literal:

Opcode	Size	Opmode	Reg Dest	Literal

  - Operaciones entre registro y memoria con direccionamiento directo:

Opcode	Size	Address

  - Saltos:

Opcode	Address

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2343 Arquitectura de Computadores

Programación en Arquitectura x86 (16 bits)

1. Motivación

La arquitectura x86 es la más usada hoy en día en los computadores personales, y por tanto es relevante entender como se desarrollan programas en el assembly de esta.

2. Programación en Arquitectura x86 de 16 bits

Para poder desarrollar programas en el assembly de la arquitectura x86, es necesario entender los elementos básicos de esta que pueden ser usados en los programas, en particular: registros, variables e instrucciones avanzadas.

2.1. Registros

A continuación se listan los registros de la arquitectura que pueden ser usados en un determinado programa:

- Registro AX: registro de 16 bits, de propósito general. Puede ser usado como dos subregistros de 8 bits:

AH	AL
8 bits	8 bits

- Registro BX: registro de 16 bits, de propósito general y direccionamiento indirecto por registros (registro base). Puede ser usado como dos subregistros de 8 bits:

BH	BL
8 bits	8 bits

- Registro CX: registro de 16 bits, de propósito general. Puede ser usado como dos subregistros de 8 bits:

CH	CL
8 bits	8 bits

- Registro DX: registro de 16 bits, de propósito general. Puede ser usado como dos subregistros de 8 bits:

DH	DL
8 bits	8 bits

- Registro SI: registro de 16 bits, de propósito general y direccionamiento indirecto por registros (registro índice)
- Registro DI: registro de 16 bits, de propósito general y direccionamiento indirecto por registros (registro índice)
- Registro SP: (stack pointer) registro de 16 bits, indica la posición tope del stack.
- Registro BP: (base pointer) registro de 16 bits, indica la posición base del frame de la subrutina. Puede ser usado para direccionar indirectamente el stack.

2.2. Variables

El assembly de la arquitectura x86 de 16 bits soporta dos tipos de datos principales: tipo byte, de 8 bits representado por el símbolo `db` y tipo word, de 16 bits representado por el símbolo `dw`. La declaración de variables en el asssembly tiene la siguiente sintáxis:
Identificador Tipo Valor.

Adicionalmente el assembly soporta definición de arreglos, los cuales pueden ser tanto de tipo byte como de tipo word. La sintáxis para declarar arreglos es la siguiente:
Identificador Tipo Valor1, *Valor2*, *Valor3*,...

Es importante analizar la relación entre las variables y su almacenamiento en memoria. Suponiendo que se han declarado dos variables: `var1 db 0x0A` y `var2 dw 0x07D0`, y que las direcciones asociadas a las variables son: var1 = 100 y var2 = 101, y se han declarado dos arreglos: `arr1 db 0x01, 0x02, 0x03` y `arr2 dw 0x0A0B, 0x0C0D` , y que las direcciones asociadas a los arreglos son: arr1 = 103 y arr2 = 106, el estado de la memoria sería el siguiente

Variable	Dirección (16 bits)	Palabra (8 bits)
var1	100	0x0A
var2	101	0xD0
	102	0x07
arr1	103	0x01
	104	0x02
	105	0x03
arr2	106	0x0B
	107	0x0A
	108	0x0D
	109	0x0C

De esta tabla de memoria se pueden observar varios elementos relevantes:

- El tamaño de las palabras de memoria es de 8 bits (1 byte).

- Las variables de tipo word se guardan en dos palabras de memoria en orden **little endian**.
- El tamaño de las direcciones de memoria es de 16 bits (1 byte).

A continuación se presenta un ejemplo de programación en assembly ocupando los registros y variables antes descritos. Es importante tomar en cuenta que las instrucciones de copia entre registro y memoria permiten copiar variables tanto de tipo byte como de tipo word de manera directa, pero se debe tener en consideración el registro que se utiliza para la carga: registros de 8 bits para variables tipo byte (como AH, AL) y registros de 16 bits para variables tipo word (como AX).

Ejemplo Multiplicación: Código Java

```java
public static void mult()
{
  int a = 10;
  int b = 200;
  int res = 0;
  while(a > 0)
  {
    res += b;
    a--;
  }
  System.out.println(res);
}
```

Ejemplo Multiplicación: Código Assembly x86

```
;Calculo de la multiplicacion res = a*b

org 100h

MOV AX, 0
MOV CX, 0
MOV DX, 0

MOV CL, [a]          ;CL guarda el valor de a
MOV DL, [b]          ;DL guarda el valor de b

start:
CMP CL, 0            ;IF a <= 0 GOTO end
JLE endprog

ADD AX, DX           ;AX += b

DEC CL               ;a--
JMP start

endprog:

MOV [res], AX        ;res = AX

RET

a         db 10
b         db 200
res       dw 0
```

2.3. Instrucciones avanzadas

La arquitectura x86 tiene un ISA CISC, y por tanto implementa instrucciones más complejas que pueden simplificar la programación. En particular en la arquitectura de 16 bits vienen implementadas las instrucciones `MUL op` y `DIV op` las cuales implementan las operaciones de multiplicación y división entera respectivamente, ocupando el registro AX como operando y resultado, de la siguiente forma: `MUL op =>AX = AL*op` y `DIV op =>AL = AX/op`.

Con estas instrucciones se puede reeimplementar el código de la multiplicación visto previamente de una manera mucho más simple:

```
;Calculo de la multiplicacion res = a*b

org 100h

MOV AX, 0

MOV AL, [a]          ;AL = a
MUL [b]              ;AX = AL*b

MOV [res], AX        ;res = AX

RET

a         db 10
b         db 200
res       dw 0
```

Otra instrucción especial que provee el ISA es la instrucción `LEA` Load Effective Address. El propósito de esta instrucción es poder obtener la dirección en que efectivamente está almacenada una determinada variable o arreglo. Esto es más complejo de determinar en la arquitectura x86 que en el computador básico debido a que esta arquitectura soporta la presencia de múltiples programas, y por tanto la ubicación de un determinado programa no se sabe necesariamente antes de ejecutarlo.

La sintáxis de la instrucción es la siguiente `LEA reg, var`, lo cual está copiando la dirección de la variable `var` en el registros de 16 bits `reg`. A continuación se muestra un ejemplo en que se usa la instrucción para recorrer un arreglo al calcular el promedio:

Ejemplo Promedio: Código Java

```
public static void promedio()
{
  int[] arreglo = new int[]{6,4,2,3,5};
  int n = 5;
  int i = 0;
  int promedio = 0;

  while(i < n)
  {
    promedio += arreglo[i];
    i++;
  }
  promedio /= n;
  System.out.println(promedio);
}
```

Ejemplo Promedio: Código Assembly x86

```
;Calculo del promedio de un arreglo

org 100h

MOV CL, [n]          ;CL tendra el valor del tamano completo del arreglo
MOV SI, 0            ;SI se usara para iterar sobre el arreglo

MOV AX, 0            ;AL se usara para guardar el promedio
MOV DX, 0            ;DL se usara para guardar los valores del arreglo

start:               ;Loop principal: recorrer el arreglo completo
CMP SI, CX
JGE endprog

LEA BX, arreglo      ;En BX se almacena posicion inicial del arreglo comprimido

MOV DL, [BX + SI]    ;Se almacena en AL el primer valor que indica las repeticiones

ADD AL, DL           ;Se acumula el valor del arreglo

INC SI               ;Se incrementa el contador del loop principal
JMP start

endprog:

DIV [n]
MOV [promedio], AL

RET

n        db 5
arreglo  db 6,4,2,3,5
promedio db 0
```

2.4. Subrutinas

El manejo de subrutinas en la arquitectura x86 es más complejo que en el computador básico, principalmente porque el stack (i.e. el segmento inferior de la memoria) tiene una mayor relevancia, y es usado para guardar más que solamente la dirección de retorno (como es el caso del computador básico).

La estructura general de un stack luego de llamar e inicializar una subrutina en la arquitectura x86 es la siguiente:

SP →	
BP →	Variables locales
	Base Pointer anterior a la llamada
	Dirección de retorno
	Parámetros de la subrutina

Se puede observar que en el stack se está almacenando:

- Los parámetros de la subrutina: a diferencia del computador básico en que se ocupaban variables para el paso de parámetros, en la arquitectura x86 se ocupa el stack, ocupando las instrucciones `PUSH` antes del llamado para agregarlos.
- La dirección de retorno: al igual que en el computador básico, en la arquitectura x86 al momento de ejecutar la instrucción `CALL` se almacena la dirección de retorno (la que viene después de la llamada) en el stack, para al momento de retornar de la subrutina saber a donde se debe ir.
- El valor anterior del base pointer: que es importante en el caso de una secuencia de llamadas anidadas (más adelante se verá porque).
- Las variables locales: la arquitectura x86 agrega este concepto de variables locales, variables que solo viven en el scope de la subrutina, las cuales son definidas en el stack, y por tanto dejarán de ser accesibles luego del retorno de la subrutina.

Adicionalmente, la forma en que son manejados los parámetros y retorno en la arquitectura x86 es especial, y tiene distintas variaciones. Cada una de estas variaciones se denomina una **convención de llamada**, en nuestro caso ocuparemos la convención de Windows, que se denomina **stdcall**. Esta convención especifica lo siguiente:

- Los parámetros son pasados de derecha e izquierda
- El retorno se almacenará en el registro AX
- La subrutina se debe encargar de dejar el SP apuntando en la misma posición que estaba antes de pasar los parámetros.

Considerando estos dos elementos, la secuencia completa de pasos que se debe realizar para llamar una subrutina es la siguiente:

1. Estando en la parte del programa que va a llamar a la subrutina:

a) Paso de parámetros: se deben agregar al stack los parámetros que se necesiten en la subrutina, ocupando la instrucción `PUSH`. Es importante señalar que esta instrucción solo permite agregar valores de 16 bits, es decir 16 bits es la unidad mínima de dato del stack.

b) Llamada a la subrutina: ocupando la instrucción `CALL` se llama a la subrutina, lo que almacena en el stack la dirección de retorno, y ejecuta el salto a la dirección de la subrutina.

2. Estando dentro de la subrutina:

a) Guardar el valor actual del base pointer (BP) en el stack, y cargar el valor del stack pointer en el base pointer, lo que se implementa con la secuencia de instrucciones: `PUSH BP` y `MOV BP, SP`

Opcional En caso de ocupar variables locales, se debe reservar el espacio para estas, moviendo el SP n posiciones hacia arriba, donde n es el número de bytes que se ocupará para las variables. Para esto se utiliza la instrucción `ADD SP, n`

b) Ejecución de la subrutina: se ejecuta lo que corresponda. Para acceder a los parámetros se ocupa direccionamiento mediante el registro BP: el primer parámetro estará en la dirección BP + 4 (dirección del base pointer + 4 bytes), el segundo parámetro en la dirección BP + 6, y así. Para acceder a las variables locales, también se ocupa direccionamiento con el registro BP pero con offsets negativos (las variables locales están «arriba» de donde está apuntando BP). Por ejemplo se puede acceder a una primera variable de 16 bits, apuntando a la dirección BP - 2.

c) Al finalizar la ejecución de la subrutina se debe primero recuperar el espacio asignado a las variables locales, bajando el SP con un llamado a la instrucción `SUB SP, n` donde n es el mismo número de bytes asignados al comienzo.

d) Luego se debe rescatar el valor previo del BP, con la instrucción `POP BP`

e) Finalmente se debe retornar, pero para que el SP quede ubicado al final de la memoria, es necesario indicarle que se mueva el espacio ocupado por los parámetros, para lo cual se le agrega un parámetro a la instrucción de retorno de la siguiente forma: `RET n` donde n indica el número de bytes ocupados por los parámetros.

A continuación se muestran dos ejemplos de subrutinas. Ambas implementan una subrutina de potencia ($base^{exponente}$ pero la primera no ocupa variables locales, la segunda si:

Ejemplo Potencia sin variables locales: Código Java

```
public static int potencia(int base, int exp)
{
  int res = 1;
  while(exp > 0)
  {
    res *= base;
    exp--;
  }
  return res;
}
```

Ejemplo Potencia sin variables locales: Código Assembly x86

```
;Calculo de la potencia pow = base*exp

org 100h

MOV BL, [exp]
MOV CL, [base]

PUSH BX             ;Paso de parmetros (de derecha a izquierda)
PUSH CX

CALL potencia   ;potencia(base,exp)

MOV [pow], AL       ;Retorno viene en AX

RET

potencia:     ;Subrutina para el calculo de la potencia

PUSH BP
MOV BP, SP      ;Actualizamos BP con valor del SP

MOV CL, [BP + 4]  ;Recuperamos los dos parámetros
MOV BL, [BP + 6]

MOV AX, 1     ;AX = 1

start:
CMP BL,0      ;if exp <= 0 goto endpotencia
JLE endpotencia

MUL CL        ;AX = AL * base

DEC BL        ;exp--
JMP start

endpotencia:

POP BP
RET 4       ;Retornar, desplazando el SP en 4 bytes (2 por cada parametro)

base     db 2
exp      db 7
pow      db 0
```

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2343 Arquitectura de Computadores

Comunicación de la CPU y la Memoria con I/O

1. Motivación

Una vez entendido las bases de la programabilidad de un computador, el siguiente paso corresponde a estudiar la comunicación entre las distintas partes del computador, la comunicación con los usuarios y la comunicación con otros computadores. Un primer aspecto a analizar es como la CPU y la Memoria se comunican con el resto de los componentes que pueda tener un computador.

2. Comunicación entre las partes del computador

El modelo de computador visto hasta el momento incluye principalmente dos componentes: Memoria y CPU (Central Processing Unit). La memoria corresponde al componente donde se almacenan datos e instrucciones (juntos si es la arquitectura Von Neumann, separados si es Harvard). La CPU corresponde a todos los elementos adicionales que permiten implementar programas en la máquina: registros de uso general, registros de uso específico, unidades de ejecución, unidad de control, program counter y todos los componentes de digitales que permiten la conexión entre los elementos.

En las figura 1 y 2 se observa el diagrama de un computador básico (con arquitecturas Harvard y Von Neumann, respectivamente), agrupando todos los elementos de la CPU en un sólo componente. Se puede observar que existen distintos tipos de comunicación, los cuales pueden ser categorizados en tres grupos:

- **Comunicación de direcciones:** Tanto en el computador Harvard como en el Von Neumann existe una conexión entre la CPU y la dirección de la memoria RAM, que permite que el computador indique que dato (o instrucción en el caso Von Neumann) se va a leer o escribir. En el computador Harvard, adicionalmente se tiene una conexión hacia la dirección de la ROM, indicando la instrucción a ejecutar.
- **Comunicación de datos:** Nuevamente, en ambos modelos existen canales de comunicación de datos con la RAM: uno para enviar datos que entren a la memoria, y otro para recibir datos que salgan de esta. En Harvard adicionalmente existe una conexión de datos recibidos de la ROM, que corresponde a las instrucciones del programa en ejecución.
- **Comunicación de control:** Por último, para controlar la RAM ambas arquitecturas tiene una señal de control que indica si la memoria RAM debe leer o escribir en la dirección indicada.

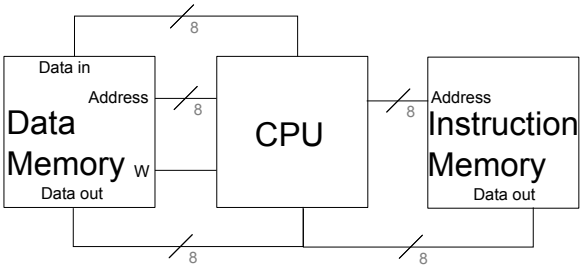


Figura 1: Abstracción de computador básico con microarquitecura Harvard

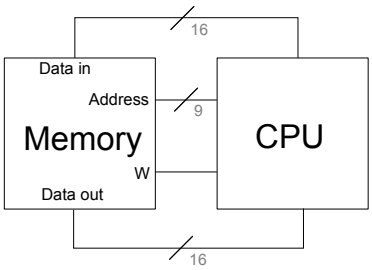


Figura 2: Abstracción de computador básico con microarquitecura Von Neumann

Dado que los tipos de comunicación para ambas arquitecturas son similares, de ahora en adelante se trabajará con la arquitectura Von Neumann, pero todo los conceptos serán aplicables también a la arquitectura Harvard. Para completar un modelo de computador general Von Neumann con una memoria y una CPU, es necesario considerar la implementación en detalle de los canales de comunicación entre estos dos elementos, y además especificar como se comunican los otros dispositivos externos que permiten ingresar y extraer datos desde el computador.

2.1. Canales de comunicación: Buses compartidos

Los canales de comunicación que habitualmente se ocupan para los distintos tipos de comunicación en un computador corresponden a **buses compartidos**. Un bus compartido consiste en un canal de cables eléctricos que puede ser accedido por más de un dispositivo a la vez. La ventaja de este modelo con respecto a buses no compartidos, es que permite reducir la cantidad de conexiones entre los distintos componentes del computador. La desventaja es que si dos dispositivos transmiten información al mismo tiempo se puede producir una falla, ya que se estará tratando de definir voltajes distintos para las mismas líneas eléctricas.

En la figura 3 se observa un computador Von Neumann con buses compartidos. Se puede ver que los canales de datos de entrada y salida fueron reducidos a un **bus compartido de datos**, en el cual habrán datos de salida o entrada dependiendo de lo que se requiera. El **bus de dirección** y **bus de control** completan el modelo de computador con buses compartidos.

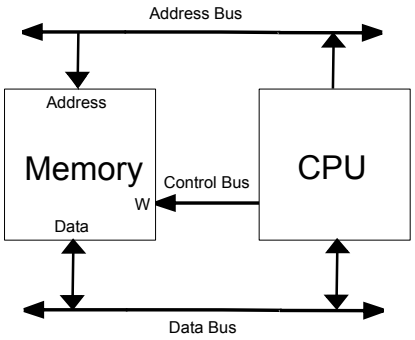


Figura 3: Abstracción de computador Von Neumann con buses compartidos

2.2. Dispositivos de Entrada y Salida: I/O

El modelo general de un computador incluye una gran variedad de dispositivos que pueden ser conectados éste, para ingresar y recibir datos. A este conjunto de dispositivos se les conoce como dispositivos de entrada y salida (E/S) o input /outuput (I/O). Considerando los dispositivos de I/O, el modelo del computador Von Neumann se extiende de la siguiente forma:

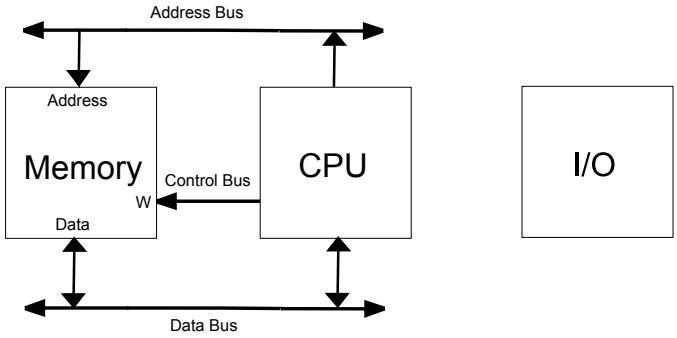


Figura 4: Abstracción de computador Von Neuman con I/O

Antes de detallar la comunicación entre los dispositivos de I/O y el resto del computador, es necesario caracterizar a estos dispositivos.

2.2.1. Características

Los dispostivos de I/O son muy diversos en sus funcionalidades y estructuras internas. Sin embargo, existen tres características relevantes que sirven para organizarlos:

- **Comportamiento**: El comportamiento de un dispositivo I/O indica que puede hacer el computador con el dispositivo. Las opciones son:
  - Entrada (Input): entregan datos al computador, pero no permiten que el computador envíe datos.
  - Salida (Output): reciben datos del computador, pero no entregan datos.
  - Entrada o salida (Input or Output): permiten entregar datos al computador o recibir datos, pero no de manera simultánea.

    - Almacenamiento (Input and Output): entregar y reciben datos almacenados en el dispositivo.

- **Comunicante**: La segunda característica indica con quien se está comunicando el dispositivo:
    - Humano: dispositivos como el mouse, teclado y la pantalla, son dispositivos que se comunican con un ser humano.
    - Máquina: dispositivos como la tarjeta de red, o los discos duros se comunican con máquinas, sin intervención de un ser humano.
- **Velocidad de transferencia**: La velocidad de transferencia de los dispositivos de I/O es muy variable entre dispositivos, y a veces en un mismo dispositivo. En general se utiliza la velocidad **peak** como referencia para cada dispositivo.

Dispositivo	Comportamiento	Comunicante	Velocidad de transferencia (Mbit/seg)
Teclado	Input	Humano	0.0001
Mouse	Input	Humano	0.0038
Voice in	Input	Humano	0.2640
Scanner	Input	Humano	3.2
Voice out	Output	Humano	0.2640
Impresora laser	Output	Humano	3.2
Display	Output	Humano	800
Red por cable	Input o Output	Máquina	100-1000
Red inalámbrica	Input o Output	Máquina	11-54
Disco óptico	Almacenamiento	Máquina	80-220
Disco magnético	Almacenamiento	Máquina	800-3000

Tabla 1: Ejemplos de dispositivos I/O y sus características.

2.2.2. Componentes de los dispositivos I/O

A pesar de las diferentes características que tiene los dispositivos de I/O, la mayoría de estos contienen dos tipos de componentes definidos: **elementos electromecánicos** que realizan las operaciones de interacción, y **controladores** electrónicos que regulan el funcionamiento de los componentes mecánicos y se comunican con el computador.

Controladores

Los controladores de dispositivos de I/O son circuitos digitales encargados de controlar las partes electro/mecánicas del dispositivo según la comunicación realizada con el computador. En la mayoría de los casos, los controladores son microprocesadores completos, que ejecutarán programas especialmente diseñados para controlar al dispositivo. Las funciones principales del controlador son las siguientes:

- Comunicación con el computador.
- Comunicación con el dispositivo.
- Almacenamiento temporal.

- Detección de errores.
- Control de elementos mecánicos/físicos.
- Conversión de señales continuas en digitales o vice-versa.

Para cumplir las funciones antes descritas, los controladores cuenta con los siguientes componentes:

- **Circuitos de control:** Dependiendo de la complejidad del dispositivo será la complejidad de estos circuitos, que en muchos casos corresponde a microprocesadores completos, y estarán encargados de regular el funcionamiento del dispositivo, coordinar la comunicación con el computador y ejecutar la detección y/o corrección de errores.
- **Conversores ADC o DAC:** Dependiendo de la funcionalidad y complejidad, los controladores puede incluir conversores análogo-digital o digital-análogo para traducir señales eléctricas continuas en información para el computador o vice-versa.
- **Memoria:** Los controladores contendrán una memoria la cual será el mecanismo fundamental para comunicarse con el computador. Algunas direcciones de esta memoria se reservarán para funciones específicas, y se denominan registros:
  - **Buffer:** parte de la memoria, que almacena los datos que el dispositivo este entregando o recibiendo del computador.
  - **Registros de control:** direcciones específicas de la memoria que son escritos por el computador para indicar comandos que debe ejecutar el dispositivo.
  - **Registros de status:** direcciones específicas de la memoria que son escritos por el dispositivo para indicar información al computador.
  - **Registros de datos:** direcciones específicas de la memoria para leer o escribir datos individuales o asociados a la memoria local (buffer) del dispositivo.
  - **Registros de dirección:** direcciones específicas de la memoria para direccionar la memoria local (buffer) del dispositivo.

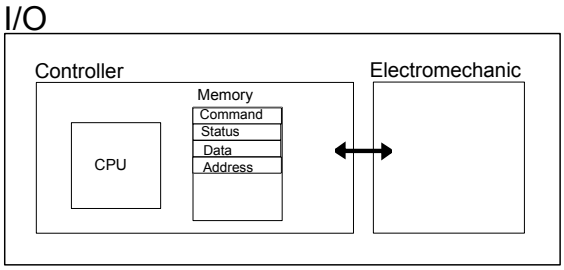


Figura 5: Diagrama genérico de I/O

2.3. Comunicación entre los dispositivos de I/O, la CPU y la Memoria

Para poder interactuar con los dispositivos de I/O es necesario que la CPU pueda comunicarse con estos. Como se señaló anteriormente, el controlador de cada dispositivo será el encargado de realizar está comunicación con la CPU, para lo cual el procedimiento que se utiliza es que la CPU escriba o lea directamente información de los registros o el buffer del controlador. En base a este procedimiento, existen tres tipos de comunicaciones que pueden ocurrir entre la CPU y los dispositivos de I/O:

- **Comunicación de comandos:** Cuando la CPU quiere indicarle a un dispositivo que realice una determinada acción, debe hacerlo enviándole comandos a los registros de control. Por ejemplo si la CPU quiere avisarle a un cierto dispositivo (e.g cámara web) que se encienda, deberá escribir en un determinado registro de control del controlador del dispositivo un cierto número, que este interpretará como el comando de activación.
- **Comunicación de estado:** Cuando la CPU quiere obtener información sobre el estado del dispositivo, dee hacerlo leyendo uno de los registros de status de éste. Por ejemplo si la CPU quiere saber si un dispositivo tiene nueva información que enviar (e.g el mouse se movió, y tiene nuevas posiciones que enviar), la CPU deberá leer el valor almacenado en el registro específico e interpretarlo según corresponda.
- **Transferencia de datos:** Cuando la CPU quiere enviarle datos a un dispositivo o leer datos de él, si es poca información esto se podrá hacer ocupando registros de datos que tenga el controlador del dispositivo. Si es más información, esto se realizará escribiendo o leyendo desde el buffer (memoria) del dispositivo. Por ejemplo, si el disco duro quiere enviarle información a la CPU, se accederá a ésta mediante el buffer de datos del disco.

2.3.1. Acceso a los dispositivos de I/O y transferencia de datos

Para poder realizar los tipos de comunicación antes descrito es necesario que la CPU pueda acceder a los registros y memorias de los distintos dispositivos de I/O. Para esto es necesario definir un mecanismo general que permita acceder de manera equivalente a todos los dispositivos y enviarle datos, es decir hay que poder **direccionar** los distintos registros o buffers y **transferir datos** a ellos.

El modelo de direccionamiento + transferencia de datos ya existe en el computador, en la comunicación de la CPU con la memoria de datos. En ese caso, si se tiene una instrucción de transferencia, por ejemplo `MOV A, (120)`, lo que ocurrirá en el sistema es:

- La CPU coloca el valor 120 en el bus de direcciones.
- La CPU envia la señal de control $W = 0$ a la RAM indicando que va a leer y no escribir de la RAM
- La RAM recibe desde el bus de direcciones en su entrada de dirección el valor 120.
- La RAM coloca en el bus de datos el valor $Mem[120]$, es decir, el valor de memoria apuntado por la dirección 120.
- La CPU recibe desde el bus de datos el valor $Mem[120]$ y lo almacena en el registro A.

Para acceder a los dispositivos de I/O se necesitan ejecutar pasos similares a los de comunicación con la memoria. Por ejemplo si la CPU quiere leer el estado a un cierto dispositivo, los pasos a seguir serían:

- La CPU coloca en el bus de direcciones algún valor que «direcciona» al registro de status de ese dispositivo.
- El dispositivo recibe desde el bus de direcciones ese valor y lo interpreta apuntando el valor del registro de status.
- El dispositivo coloca el valor del registro de status en el bus de datos.
- La CPU recibe desde el bus de datos el valor del registro de status del dispositivo.

Existen dos mecanismos principales para realizar el proceso de direccionamiento y transferencia de datos: mapeo a memoria (**memory mapped I/O**) y usando el mecanismo de puertos (**port I/O**).

Memory mapped I/O

La idea central de memory mapped I/O es la siguiente: si el computador ya tiene soporte a nivel de instrucciones para direccionar y transferir datos desde y hacia la memoria de datos, por que no aprovechar esos mismos mecanismos para acceder a los registros y buffers de los dispositivos de I/O. La implementación de esta idea consiste en reservar un espacio de direcciones de memoria para ser ocupados para «mapear» los registros y buffer de los dispositivos de I/O. De esta forma es posible acceder directamente ocupando una instrucción como `MOV A, (dir)` a un dispositivo.

El mapeo explícito a los distintos registros y buffers de los dispositivos los coordina el **address decoder**, una pieza de hardware especializada que estará «vigilando» el bus de dirección para determinar si la dirección colocada corresponde a la RAM o alguno de los dispositivos.

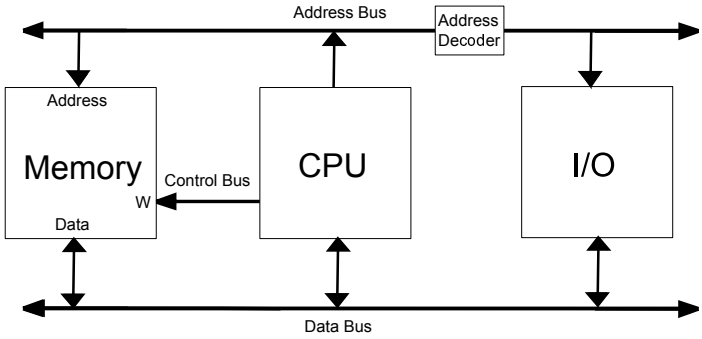


Figura 6: Computador ocupando memory mapped I/O

Los buffer de los dispositivos pueden estar mapeados completamente en memoria o no. En caso de estar mapeados completamente, el acceso a cada dirección de memoria del buffer es directo. En caso contrario, el mecanismo que se ocupa es tener mapeados los registros de dirección y datos del dispositivo, y ocupar esos registros para direccionar y escribir/leer datos, respectivamente.

El mecanismo de mapeo a memoria tiene varias ventajas: su implementación es simple a nivel de hardware, es posible ocupar las mismas instrucciones de transferencia y modos de direccionamiento

que para trabajar con memoria y también es posible realizar operaciones en la ALU, de la misma forma que se realizaban con datos de memoria.

La desventaja de este sistema es que limita el espacio direccionable de la memoria, lo que se conoce como **memory barrier**. Debido a que se deben reservar un grupo de direcciones para apuntar a los dispositivos, estas direcciones no pueden ser ocupadas para almacenar en memoria.

En la arquitectura x86 el acceso a muchos de los dispositivos se realiza mediante mapeo a memoria. En el IBM PC original (primer computador con arquitectura x86), se tenía un espacio direccionable de memoria de 1MB, del cual el espacio entre las direcciones 640KB y 1MB se reservó para mapeo de dispositivos de I/O. Uno de los dispositivos que ocupaban mayor parte de ese espacio de direcciones era la tarjeta de video, la cual tenía mapeada su memoria de video de manera de permitir pintar pixeles en el display ocupando instrucciones de copia a memoria.

Port I/O

El segundo mecanismo para acceder a los dispositivos es mediante port I/O. En este mecanismo se definen instrucciones específicas y un espacio de direccionamiento propio para acceder a los dispositivos. En el caso de la arquitectura Intel, por ejemplo, se definen dos instrucciones: `IN` que se ocupa para leer datos desde un dispositivo y `OUT` que se ocupa para escribir datos en un dispositivo.

Para direccionar, se ocupa un espacio de direcciones separado del de la RAM, por ejemplo, la instrucción `IN A, (120)` copiaría el valor de I/O asociado a la dirección 120 en el registro A. Esa dirección 120 no tendría relación alguna con la dirección 120 de memoria RAM, y por tanto no se limita el espacio direccionable con este método.

Para efectivamente por implementar este direccionamiento especial, existen dos mecanismos. El primero es utilizar un bus de direcciones especial sólo para I/O (figura 7) con lo que se asegura que no habrá choque entre las direcciones. El segundo método consiste en ocupar el mismo bus que el de la RAM, pero indicarle mediante una señal de control al bus, que se está refiriendo a una dirección de I/O, y por tanto la RAM no debe considerar ese valor.

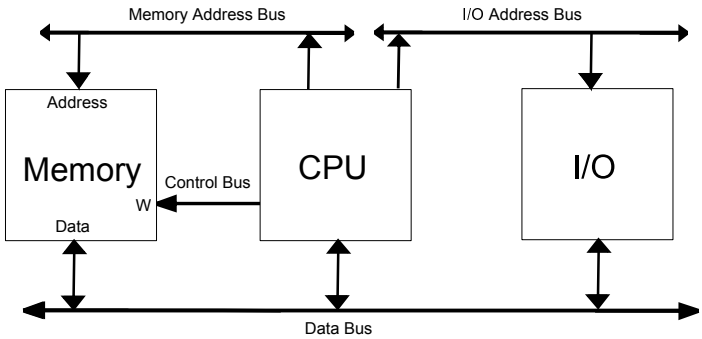


Figura 7: Computador ocupando port I/O con bus de direcciones separado

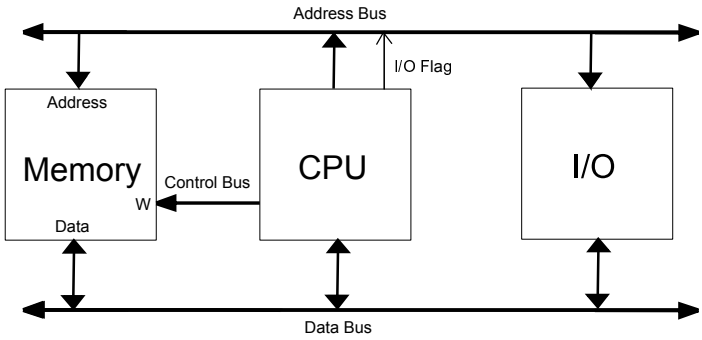


Figura 8: Computador ocupando port I/O con bus de direcciones compartido

Para utilizar Port I/O en la arquitectura x86, existe un espacio direccionable de 16 bits, permitiendo acceder a 65536 puertos, que van desde la dirección `0000h` hasta la `FFFFh`. Existen dos instrucciones especializadas para acceder a los puertos: `IN` y `OUT`. Ambas instrucciones trabajan con dos parámetros: el número (dirección) del puerto al cual se quiere acceder y el registro `AX` o `AL`, que se ocupará para recibir o enviar datos al dispositivo. El formato de ambas instrucciones es el siguiente:

- `IN Reg, Port` donde `Reg` es `AX` o `AL` y `Port` es un número que va desde 0 hasta 65535.
- `OUT Port, Reg` donde `Reg` es `AX` o `AL` y `Port` es un número que va desde 0 hasta 65535.

A continuación se muestra un ejemplo ocupando uno de los dispositivos I/O de prueba asociados al software emu8086, el cual simula el control de un robot. El controlador del robot tiene los siguientes registros: registro de comandos (asociado al puerto 9), registro de datos (asociado al puerto 10) y registro de estado (asociado al puerto 11). Los detalles de los valores asociados a cada registros se encuentran en:
http://www.emu8086.com/assembler_tutorial/io.html

```
#start=robot.exe#

org 100h

start:
        CALL waitcommand
        MOV AL, 1   ; move forward.
        OUT 9, AL   ;

        CALL waitcommand
        MOV AL, 4   ; examine.
        OUT 9, AL   ;

        CALL waitdata
        IN AL, 10
        CMP AL, 0
        JE start

        CALL TURN

        JMP start

        RET
```

```
waitcommand:
loop1:  IN AL, 11
        AND AL, 10b
        JNE loop1
        RET

waitdata:
loop2:  IN AL, 11
        AND AL, 01b
        JE loop2
        RET

turn:
        CALL waitcommand
        MOV AL, 3   ; turn right.
        OUT 9, AL   ;


        RET
```

2.3.2. Comunicación I/O → CPU

Polling

Ocupando los mecanismo de direccionamiento y transferencia previamente descritos, es posible implementar toda la comunicación necesaria entre la CPU, memoria e I/O. En el caso particular de la comunicación que va desde el dispositivo a la CPU, por ejemplo cuando un dispositivo quiere avisar que hizo algo (e.g. el mouse se movió) el algoritmo que maneja esa comunicación sería de la siguiente forma:

1. Ejecutar instrucciones del programa.
2. Revisar el estado del dispositivo leyendo su registro de status
3. Si el dispositivo tiene algo nuevo que reportar, se pasa a 4. Si no, se vuelve a 1.
4. Leer un dato del dispositivo
5. Si quedan datos por leer, volver a 4. Si se termino de leer, se vuelve a 1.

Este mecanismo se conoce como **polling** por que la CPU debe estar preguntándole o encuestando al dispositivo continuamente para saber si este tiene algo nuevo que reportar. El problema de esto es que es muy ineficiente, ya que la CPU debe gastar varias instrucciones para determinar esto, y si fueran varios dispositivos los que hay que revisar, el problema es aún peor.

La solución a esta situación es desarrollar un mecanismo especial con soporte de hardware para que los dispositivos le avisen al computador cuando tienen algo nuevo que reportar. Este mecanismo se conoce como **interrupciones**.

Interrupciones

En un sistema de I/O basado en interrupciones, el dispositivo de I/O será el encargado de avisar cuando ocurre un suceso relevante a la CPU, liberando a ésta de tener que estar preguntándole a cada dispositivo si ha habido algún evento relevante. Para lograr esto, el dispositivo de I/O se

debe conectar a la CPU con una señal de control dedica denominada **interrupt request** o IRQ. Mediante esta señal de 1 bit el dispositivo le notificará a la CPU si ocurrió algo relevante.

Para atender la solicitud del dispositivo, se definen subrutinas especiales, denominadas **Interrupt Service Routine (ISR)**. En estas rutinas se debe definir el código encargado de atender la solicitud del dispositivo.

Cuando la CPU detecta que ocurrió la interrupción los pasos que se deben realizar son los siguientes:

1. Dispositivo interrumpe (IRQ).
2. CPU termina de ejecutar la instrucción actual, y guarda en el stack los condition codes.
3. CPU deshabilita la atención de más interrupciones.
4. CPU llamar a la ISR asociada al dispositivo.
5. ISR respalda el estado actual de la CPU.
6. ISR ejecuta su código.
7. ISR devuelve el estado previo a la CPU.
8. ISR retorna.
9. CPU rehabilita la atención de interrupciones.
10. CPU recuperar condition codes desde el stack.

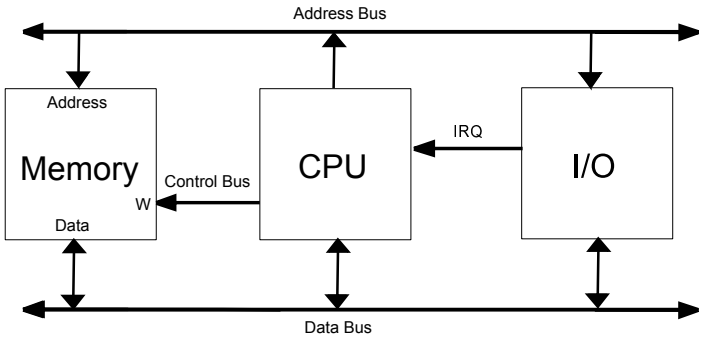


Figura 9: Computador con un dispositivo I/O ocupando interrupciones

En la práctica, un computador cuenta con múltiples dispositivos de I/O que pueden interrumpir. Debido a esto el manejo de interrupciones se complica, ya que todos los dispositivos deben poder notificarle a la CPU algún evento. Para lograr esto una opción sería que a la CPU llegara una señal IRQ por cada dispositivo. El problema de esto es que no es escalable. Para solucionar este problema se agrega un **controlador de interrupciones** al cual se conectan todos los dispositivos, y el cual se comunica directamente con la CPU. Cuando ocurre una interrupción, el controlador notificará a la CPU. Está, para saber quien fue el que interrumpió, le envía una señal **Interrupt Acknowledge (INTA)** al controlador, para que le envíe el id del dispositivo por el bus de datos.

Como existen distintos dispositivos, existirá una ISR asociado al id de cada dispositivo. Las direcciones donde se encuentran almacenadas las ISR se encuentran en lo que se conoce como **vector**

de interrupciones. El vector de interrupciones es un conjunto de palabras de memoria (habitualmente almacenados al comienzo de ésta), en el cual cada palabra tiene almacenada la dirección de un ISR de un dispositivo. De esta manera, el id que entrega el controlador de interrupciones luego de que la CPU le envíe la señal INTA, va a corresponder a un dirección dentro del vector de interrupciones, que indicara a su vez la dirección de memoria donde comienza la ISR asociada, y por tanto esa dirección se cargará en el program counter, para que la CPU ejecute las subrutina.

Una vez que se termina de ejecutar la ISR, esta se encarga de avisarle al controlador de interrupciones que ya se completo la atención, enviando un comando **End of Interrupt (EOI)**. Cuando el controlador recibe este comando

Para deshabilitar una interrupción específica que no se quiere recibir, se utiliza el concepto de **enmascaramiento o masking**, que consiste en poder especificar que algunas interrupciones se desestimen. Para eso, el controlador de interrupciones tiene un registro especial, el Interrupt Masking Register o **IMR** que permite definir que interrupciones se atenderán y cuales no.

Otro potencial problema de trabajar con múltiples dispositivos es que pueden ocurrir múltiples interrupciones mientras se esté atendiendo ya una. Para manejar esto, lo que se hace es que el controlador de interrupciones, mientras no recibe una señal INTA, encola las solicitudes. Cuando se recibe un EOI, se le envía a la CPU la señal de la primera interrupción en la cola a ser atendidas. Si hay más de una interrupción en cola, el controlador las ordenara de acuerdo a **prioridades** que estarán preestablecidas para los distintos dispositivos.

En algunos casos particulares, algunas interrupciones son demasiado relevantes como para encolarlas. Estas interrupciones se denominan **no enmascarables** y son capaces de interrumpir una ejecución de un ISR.

Interrupciones en x86

El manejo de las interrupciones en la arquitectura x86 esta controlado por el **Programmable Interrupt Controller 8259** (PIC8259), una pieza de hardware especialmente diseñada para manejar interrupciones de dispositivos de I/O. El PIC está compuesto por los siguientes componentes:

- IRQs: Cada PIC8259 maneja hasta 8 IRQs (IRQ0-IRQ7), cada uno de los cuales estará asociado a un dispositivo I/O.
- Interrupt Request Register: Registro de 8 bits mantiene la información de las interrupciones que están actualmente esperando un acknowledge de la CPU (INTA). Cada bit se asocia a un IRQ, donde un 1 en ese bit representa que el dispositivo asociado a ese IRQ interrumpió y espera INTA.
- In-Service Register: Registro de 8 bits mantiene la información de las interrupciones que están siendo atendidas. Cada bit se asocia a un IRQ, donde un 1 en ese bit representa que el dispositivo asociado a ese IRQ está siendo atendido y espera EOI.
- Interrupt Mask Register: Registro de 8 bits mantiene la información de las interrupciones que deben ser consideradas. Cada bit se asocia a un IRQ, donde un 1 en ese bit representa que si el dispositivo interrumpe debe ser atendido; un 0 indicará que una interrupción de ese dispositivo no será considerada. Este registro puede ser modificado en la ISR usando el port I/O mediante el puerto 0x21 para el primer PIC, y 0xA1 para el segundo PIC.

- Priority Solver: Circuito que define que interrupción notificar primero a la CPU, en caso de existir varias interrupciones pendientes.

El PIC8259 tiene la capacidad de trabajar en modo «maestro» o «esclavo», lo que permite que a un PIC «maestro», se le puedan conectar las salidas IRQ de otros PIC «esclavos» a alguna de sus entradas IRQ. En la arquitectura x86 tradicional se utilizan dos PIC8259, uno maestro y uno esclavo. El PIC esclavo se conecta al IRQ2 del PIC maestro, totalizando 15 posibles IRQ para conectar dispositivos. El diagrama de conexión entre los PIC y la CPU se observa en la figura 1.

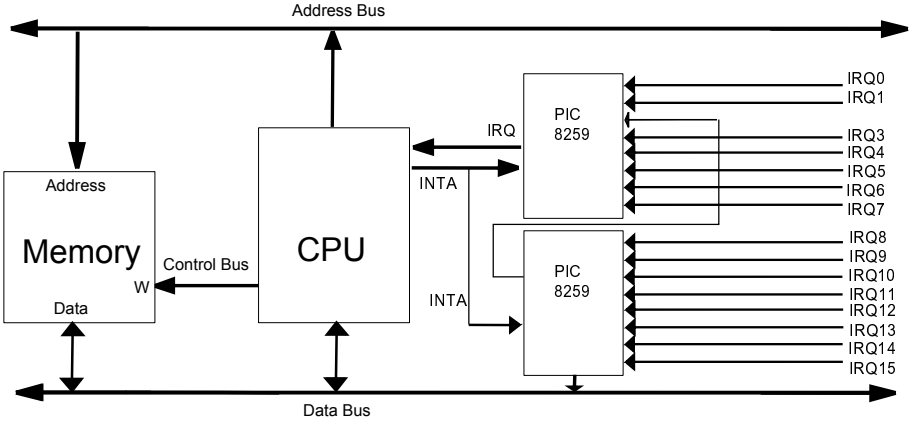


Figura 10: Conexión entre controladores de interrupciones y la CPU en la arquitectura x86 tradicional.

A nivel de la CPU, la habilitación y deshabilitación de las interrupciones está controlada mediante un flag del registro de status denominado **Interrupt Flag (IF)** el cual cuando está en 0 indica que no se atenderán interrupciones, y cuando están en 1 indica que si se atenderán.

Para enviar una señal EOI, los dos PIC tiene un registro de comando, asociados a los puertos 0x20 y 0xA0 respectivamente. La señal EOI se envía mediante un comando 0x20 a el puerto que corresponda.

A continuación se presenta la asociación habitualmente ocupada en la arquitectura x86 entre IRQs y dispositivos:

IRQ	Dispositivo	Vector de interrupción
IRQ0	Timer del sistema	08
IRQ1	Puerto PS/2: Teclado	09
IRQ2	Conectada al PIC esclavo	0A
IRQ3	Puerto serial	0B
IRQ4	Puerto serial	0C
IRQ5	Puerto paralelo	0D
IRQ6	Floppy disk	0E
IRQ7	Puerto paralelo	0F
IRQ8	Real time clock (RTC)	70
IRQ9-11	No tienen asociación estándar, libre uso.	71-73
IRQ12	Puerto PS/2: Mouse	74
IRQ13	Coprocesador matemático	75
IRQ14	Controlador de disco 1	76
IRQ15	Controlador de disco 2	77

Tabla 2: IRQs en arquitectura x86 tradicional.

Considerando todos estos elementos, el flujo completo de manejo de interrupciones es el siguiente:

1. Dispositivo envía señal IRQ al controlador.
2. PIC revisa su registro IMR, si la interrupción no está enmascarada la atiende, marcando un 1 en el bit correspondiente del Interrupt Request Register.
3. PIC decide cual de las interrupciones actuales es prioritaria (menor IRQ) y marca un 1 en el bit correspondiente del In-Service Register.
4. PIC envía interrupción (INT) a la CPU.
5. CPU termina de ejecutar la instrucción actual.
6. CPU revisa si el flag de interrupciones está activo (IF = 1), en cuyo caso va a atender a la interrupción.
7. CPU deshabilita la atención de más interrupciones (IF=0).
8. CPU guarda en el stack los condition codes.
9. CPU envía INTA para saber quien interrumpió.
10. PIC revisa In-Service Register para saber el id del IRQ que está siendo atendido, y envía mediante bus de datos.
11. CPU usa el id para buscar dirección de ISR en el vector de interrupciones.
12. CPU llama a la ISR asociada al dispositivo (CALL Mem[id]).

13. ISR respalda el estado actual de la CPU.
14. ISR ejecuta su código.
15. ISR envía un comando EOI al PIC, con lo cual éste setea en 0 el In-Service Register, indicando que terminó la atención de la interrupción.
16. ISR devuelve el estado previo a la CPU.
17. ISR retorna.
18. CPU recupera condition codes desde el stack.
19. CPU rehabilita la atención de interrupciones (IF=1).

A continuación se muestra un código que implementa interrupciones de hardware. En este código, primero se inscriben las ISR asociándolas a direcciones del vector de interrupción (no es relevante entender el detalle de esta parte en código). Luego de esto el programa principal entra en un loop, esperando el llamado de una ISR. En cada ISR se observa la estructura básica de una subrutina de este tipo:

1. Backup de todos los registros mediante la instrucción `POPA`
2. Ejecución del código de la subrutina
3. Envío de señal EOI (0x20) al PIC (en este caso el primer PIC, asociado al puerto 0x20).
4. Devolución de los valores de los registros mediante la instrucción `PUSHA`
5. Retorno mediante la instrucción `IRET`.

```
    #start=robot.exe#
    #start=InterruptGenerator.exe#


org 100h


start:
        MOV AX, 0
        MOV ES, AX
        MOV AL, 90H ; EN LA DIRECCION 0X90 DEL VECTOR DE INTERRUPCIONES
        MOV BL, 4H
        MUL BL
        MOV BX, AX
        MOV SI, OFFSET [turnleft] ;REFERENCIAMOS LA ISR TURNLEFT
        MOV ES:[BX], SI
        ADD BX, 2
        MOV AX, CS
        MOV ES:[BX], AX

        MOV AX, 0
        MOV ES, AX
        MOV AL, 91H  ; EN LA DIRECCION 0X91 DEL VECTOR DE INTERRUPCIONES
        MOV BL, 4H
```

```
  MUL BL
  MOV BX, AX
  MOV SI, OFFSET [forward]  ;REFERENCIAMOS LA ISR FORWARD
  MOV ES:[BX], SI
  ADD BX, 2
  MOV AX, CS
  MOV ES:[BX], AX

  MOV AX, 0
  MOV ES, AX
  MOV AL, 92H   ; EN LA DIRECCION 0X92 DEL VECTOR DE INTERRUPCIONES
  MOV BL, 4H
  MUL BL
  MOV BX, AX
  MOV SI, OFFSET [turnright]  ;REFERENCIAMOS LA ISR TURNRIGHT
  MOV ES:[BX], SI
  ADD BX, 2
  MOV AX, CS
  MOV ES:[BX], AX

  MOV AX, 0
  MOV ES, AX
  MOV AL, 93H   ; EN LA DIRECCION 0X93 DEL VECTOR DE INTERRUPCIONES
  MOV BL, 4H
  MUL BL
  MOV BX, AX
  MOV SI, OFFSET [examine]    ;REFERENCIAMOS LA ISR EXAMINE
  MOV ES:[BX], SI
  ADD BX, 2
  MOV AX, CS
  MOV ES:[BX], AX

  MOV AX, 0
  MOV ES, AX
  MOV AL, 94H   ; EN LA DIRECCION 0X94 DEL VECTOR DE INTERRUPCIONES
  MOV BL, 4H
  MUL BL
  MOV BX, AX
  MOV SI, OFFSET [turnon]     ;REFERENCIAMOS LA ISR TURNON
  MOV ES:[BX], SI
  ADD BX, 2
  MOV AX, CS
  MOV ES:[BX], AX

  MOV AX, 0     ; EN LA DIRECCION 0X95 DEL VECTOR DE INTERRUPCIONES
  MOV ES, AX
  MOV AL, 95H
  MOV BL, 4H
  MUL BL
  MOV BX, AX
  MOV SI, OFFSET [turnoff]     ;REFERENCIAMOS LA ISR TURNOFF
  MOV ES:[BX], SI
  ADD BX, 2
  MOV AX, CS
  MOV ES:[BX], AX


;Programa principal
waiting:
        MOV AL, 7
        JMP waiting

        RET

;Subrutinas
```

```
waitcommand:
loop1:  IN AL, 11
        AND AL, 10b
        JNE loop1
        RET

waitdata:
loop2:  IN AL, 11
        AND AL, 01b
        JE loop2
        RET

;ISRs

forward:
    PUSHA  ;Backup de todos los registros

        CALL waitcommand
        MOV AL, 1  ; forward.
        OUT 9, AL  ;

        MOV AL, 20h   ;EOI al PIC1
        OUT 20h, AL

        POPA   ;Devolver todos los registros
        IRET

turnright:
    PUSHA

        CALL waitcommand
        MOV AL, 3  ; turn right.
        OUT 9, AL  ;

        MOV AL, 20h   ;EOI al PIC1
        OUT 20h, AL

        POPA
        IRET

turnleft:
    PUSHA

        CALL waitcommand
        MOV AL, 2  ; turn left.
        OUT 9, AL  ;

        MOV AL, 20h   ;EOI al PIC1
        OUT 20h, AL

        POPA
        IRET

examine:
        PUSHA

        CALL waitcommand
        MOV AL, 4  ; examine.
        OUT 9, AL  ;
        CALL waitdata

        MOV AL, 20h   ;EOI al PIC1
        OUT 20h, AL
```

```
        POPA
            IRET

    turnon:
        PUSHA

            CALL waitcommand
            MOV AL, 5   ; on.
            OUT 9, AL   ;

            MOV AL, 20h    ;EOI al PIC1
            OUT 20h, AL


            POPA
            IRET

    turnoff:
        PUSHA

            CALL waitcommand
            MOV AL, 6   ; off.
            OUT 9, AL   ;

            MOV AL, 20h    ;EOI al PIC1
            OUT 20h, AL


            POPA
            IRET
```

2.3.3. Excepciones e interrupciones de software

Además de existir interrupciones gatilladas por dispositivos de I/O, existen otros eventos que son capaces de gatillar interrupciones. Un posible tipo de interrupción son las **excepciones**. Las excepciones ocurren cuando la CPU detecta alguna condición de error al ejecutar alguna instrucción. Una de las excepciones habituales es la división por 0, que ocurre cuando el dividendo de una instrucción de división tiene el valor 0, y por tanto no es posible realizar el cálculo de la operación. Un código que gatilla esta excepción se observa a continuación:

```
org 100h

MOV BX, 0
DIV BX

RET
```

Las excepciones, al igual que las interrupciones de hardware, serán atendidas por una ISR especializada, que habitualmente se denominan **exception handlers**. A diferencia de las interrupciones de hardware, las excepciones las controla directamente la CPU, y no un elemento externo (como el PIC).

Un segundo tipo de interrupción que no es gatillado por un dispositivo de I/O son las **interrupciones de software**. Las interrupciones de software corresponde a interrupciones que son gatilladas explícitamente en un programa ejecutando una instrucción especial. En el caso de la arquitectura x86, la instrucción usada para gatillar una interrupción de software es `INT dir` donde

dir corresponde a una dirección de memoria dentro del vector de interrupciones.

Una interrupción de software puede ser pensada como una llamada directa a un ISR particular. A diferencia de las interrupciones de hardware al llamar al ISR de una interrupción de hardware, las interrupciones de hardware no se deshabilitan. Para realizar esto es necesario ejecutar instrucciones explicitas que modifiquen el valor del Interrupt Flag: para deshabilitar las interrupciones se cuenta con la instrucción `CLI` (clear interrupt flag); para setear en 1 el flag y habilitar las interrupciones se cuenta con la instrucción `STI`.

Las interrupciones de software son habitualmente usadas para acceder a un dispositivo I/O. Para lograr esto, los ISR asociados a estas interrupciones tienen implementada la comunicación con un determinado dispositivo (ya sea mediante memory mapped I/O o port I/O), lo que permite ahorrarse tener que implementar esa comunicación cada vez que se quiera acceder a un dispositivo de I/O. Adicionalmente, como se verá más adelante, en los computadores con múltiples programas y un sistema operativo, el acceso a los dispositivos de I/O es posible de realizar sólo cuando la CPU está en un modo especial, por lo que directamente los programas no podrían acceder, y el mecanismo de interrupciones de software permite este acceso, protegiendo la comunicación.

A continuación se muestra un ejemplo de uso de interrupciones de software para acceder a la tarjeta de video del computador y pintar franjas de distintos colores. En este caso, la llamada `INT 10h` ejecutará una ISR asociada al manejo de la tarjeta de video. Dependiendo del parámetro almacenado en el registro AH antes de hacer la llamda, se ejecutará una ISR distinta. En este caso con $AH = 0$ se ejecuta la ISR de selección de modo de la tarjeta de video (texto o gráfico). El valor almacenado en el registro AL corresponderá al parámetro que indique efectivamente que modo elegir (en este caso el modo 13h). En el segundo llamado `INT 10h` se utiliza el parámetro $AH = 0Ch$ el cual indica que se pintará un pixel. La posición del pixel estáda dada por los valores almacenados en los registros DX (fila) y CX (columna). El color estará almacenado en los 4 bits menos significativos del registro AL en formato IRGB: el bit 3 representa la intensidad (1 alta, 0 baja); el bit 2 representa si hay componente rojo (1) o no(0), el bit 1 si hay componente verde o no, y el bit 0 si hay componente azul o no, totalizando 16 posibles colores.

```
    ORG 100H

    MOV AL, 13H

    MOV AH, 0
    INT 10H     ; MODO VIDEO
    MOV BX, COLORES
    MOV CX, 10  ; COLUMNA
FOR1:
    MOV AL, [BX]
    MOV DX, 20  ; FILA
FOR2:
    MOV AH, 0CH
    INT 10H     ; SET PIXEL(CX,DX)=AL
    INC DX
    CMP DX, 100
    JNE FOR2

    INC BX
    ADD CX,5
    CMP CX, 40
    JNE FOR1

    RET
```

```
COLORES:    DB 1100B
            DB 1010B
            DB 1001B
            DB 1110B
            DB 1111B
            DB 0111B
```

2.3.4. Tabla de vectores de interrupciones

A continuación se presenta la tabla con los vectores de interrupciones en la arquitectura x86 tradicional:

Dirección del vector	Tipo	Función asociada
00-01	Excepción	Exception handlers
02	Excepción	Usada para errores críticos del sistema, no enmascarable
03-07	Excepción	Exception handlers
08	IRQ0	Timer del sistema
09	IRQ1	Puerto PS/2: Teclado
0A	IRQ2	Conectada al PIC esclavo
0B	IRQ3	Puerto serial
0C	IRQ4	Puerto serial
0D	IRQ5	Puerto paralelo
0E	IRQ6	Floppy disk
0F	IRQ7	Puerto paralelo
10	Int. de Software	Funciones de video
11-6F	Int. de Software	Funciones varias
70	IRQ8	Real time clock (RTC)
71 - 73	IRQ9-11	No tienen asociación estándar, libre uso
74	IRQ12	Puerto PS/2: Mouse
75	IRQ13	Coprocesador matemático
76	IRQ14	Controlador de disco 1
77	IRQ15	Controlador de disco 2
78-FF	Int. de Software	Funciones varias

Tabla 3: Tabla de los vectores de interrupción en arquitectura x86 tradicional.

2.3.5. Transferencia de datos entre I/O y memoria

Programmed I/O (PIO)

Aun considerando el uso de interrupciones, todavía existe una fuente de ineficiencia en la comunicación con los dispositivos de I/O. El problema ocurre cuando un dispositivo tiene copiar datos a memoria (por ejemplo el disco duro). Supongamos que la CPU iniciara la comunicación y le solicita al dispositivo copiar ciertos datos a memoria, el algoritmo sería de la siguiente forma:

1. Configurar el dispositivo para ser leído, enviando señales de control correspondientes.
2. Leer un dato del dispositivo, ocupando memory map o port I/O. El dato queda guardado en un registro de la CPU.

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2343 Arquitectura de Computadores

Memoria Caché

1. Motivación

En la mayoría de los computadores modernos, la CPU no se comunica directamente con la memoria principal, sino con una memoria más rápida y de menor tamaño denominada memoria caché. A continuación se explica porqué se utilizan las memorias cachés y como funcionan.

2. Jerarquía de Memoria

La memoria principal de un computador juega un rol fundamental, ya que será en ésta donde se almacenen tanto los datos como las instrucciones de los programas y por tanto la CPU deberá comunicarse continuamente con está para efectuar sus operaciones. Una memoria ideal sería una con una gran capacidad de almacenamiento, que nos permita almacenar muchos datos al mismo tiempo, y con una velocidad de acceso muy rápida, que permita que la CPU pueda obtener la información rápidamente, sin perder mucho tiempo de procesamiento. El problema es que dadas las tecnologías que existen en la actualidad, hay un trade-off entre la velocidad y la capacidad de una memoria: para un costo fijo, una memoria muy rápida puede tener solo una capacidad limitada de almacenamiento, y una memoria de mucha capacidad será de una velocidad reducida. Si ocupáramos solo la memoria más rápida, el costo de tener capacidades altas de almacenamiento sería demasiado; si ocupáramos solo la memoria de gran capacidad la velocidad sería demasiado baja.

La solución a este problema es utilizar una combinación de memorias de distinto tipo, en lo que se denomina la **jerarquía de memoria** de un computador. La jerarquía de memoria de un computador se basa en el uso de diversos niveles de memoria, cada uno de los cuales ocupa una tecnología distinta, aprovechando las ventajas de cada tecnología: para almacenar mucha información se utilizan tecnologías que tienen un costo bajo por Byte, como los discos magnéticos; para acceder rápidamente a información se utilizan tecnologías electrónicas de rápido acceso, como SRAM (static RAM); para niveles intermedios se utilizan tecnologías con velocidades de acceso y costos intermedios, como DRAM (dynamic RAM).

La jerarquía de memoria comienza a partir de la CPU, la cual tendrá acceso al primer nivel de memoria, el cual será rápido, pero de poca capacidad. En el siguiente nivel, existirá una memoria de segundo nivel, la cual tendrá una velocidad menor, pero mayor capacidad. La memoria del primer nivel, entonces, será un subconjunto de la memoria del segundo nivel. En el tercer nivel se repite el proceso, está vez con una siguiente memoria cada vez más lenta pero de mayor capacidad. Los

niveles puede continuar mientras sea necesario, pero en la práctica los computadores modernos se basan en jerarquías de 3 niveles.

En la figura 1 se detalla las tecnologías, costos y velocidades de los tres niveles de memoria actualmente usados. El segundo nivel corresponde a la memoria principal («memoria RAM») del computador, la que hasta ahora ha sido el único nivel de memoria que hemos considerado. El tercer nivel corresponde al disco duro, que a pesar de ser un dispositivo de I/O y comunicarse de manera distinta con la CPU, a nivel lógico se considera como parte de la jerarquía de memoria, dado que es en el disco donde está contenida toda la información (programas y datos) del computador. Por último el primer nivel corresponde a una memoria de alta velocidad conocida como **memoria caché** de la cual hablaremos más adelante en este capítulo.

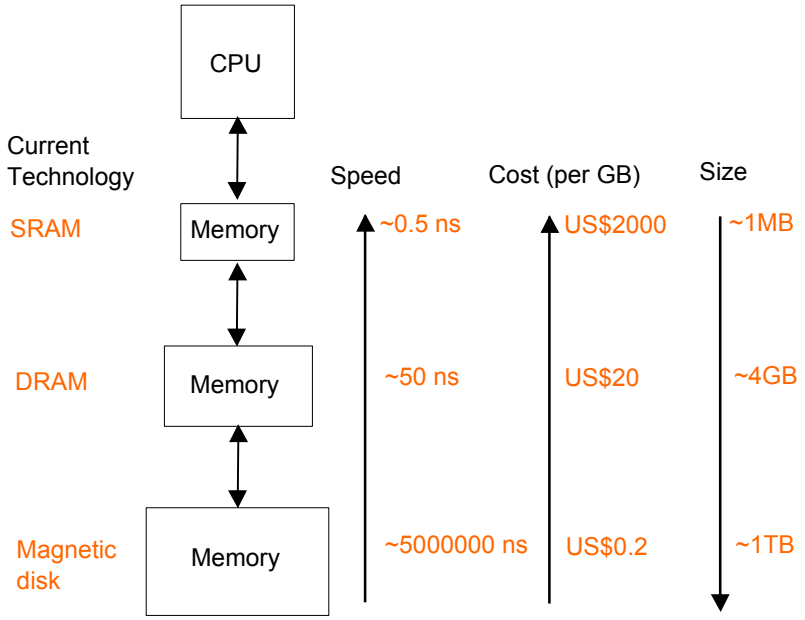


Figura 1: La jerarquía de memoria de los computadores actuales se basa en tres niveles ocupando tres tecnologías distintas.

Si cada vez que se quisiera acceder a un dato hubiese que bajar hasta el último nivel de la jerarquía, no se ganaría mucho, de hecho sería peor que si no hubiera jerarquía: el tiempo total de acceso al dato sería el tiempo acumulado de acceso a los tres niveles. La jerarquía de memoria es útil debido a lo que se conoce como el **principio de localidad**. Este principio se basa en dos tipos de localidad: **localidad temporal** y **localidad espacial**. La localidad temporal plantea que un dato recientemente obtenido de memoria, es muy probable que vuelva a ser usado en el corto plazo. La localidad espacial plantea que si un dato se necesita, es probable que datos ubicados en posiciones cercanas a este se van a necesitar en el corto plazo. De esta forma si la CPU necesita un dato y lo va a buscar al último nivel de la jerarquía, lo que conviene es que además de buscar ese dato, tome también datos contiguos y los vaya copiando en los niveles superiores de la jerarquía. Este conjunto de datos contiguos que se copia entre dos niveles de memoria se conoce como **bloque** o **línea**.

En el siguiente código se muestra una rutina de multiplicación en la arquitectura x86 que ejemplifica el principio de localidad temporal. En este caso, la variable **var2** va a ser accedida varias veces dentro del loop de multiplicación. Si la primera vez que accedemos dejamos una copia

en el nivel más alto de la jerarquía, al acceder las siguientes veces será mucho más rápido, ya que la tendremos en el nivel más rápido de memoria.

Dirección	Label	Instrucción/Dato
	CODE:	
0x00	start:	MOV CL, [var1]
0x01	while:	MOV AL,[res]
0x02		ADD AL,**[var2]**
0x03		MOV [res],AL
0x04		SUB CL,1
0x05		CMP CL,0
0x06		JNE while
	DATA:	
0x07	var1	3
0x08	**var2**	**2**
0x09	res	0

En el siguiente código se muestra una rutina que calcula el promedio de un arreglo en la arquitectura x86 que ejemplifica el principio de localidad espacial. En este caso, los distintos valores del arreglo van a ser accedidos todos de manera secuencial. Si al ir a buscar el primer valor del arreglo a un nivel bajo en la jerarquía traemos también el resto de los valores del arreglo (es decir traemos un bloque que contenga los demás valores), cuando se quiera acceder a los siguiente valores, estos estarán en el nivel alto de la jerarquía, y por tanto serán accedidos con mayor velocidad.

Dirección	Label	Instrucción/Dato
	CODE:	
0x00	start:	MOV SI, 0
0x01		MOV AX, 0
0x02		MOV BX, arreglo
0x03		MOV CL, [n]
0x04	while:	CMP SI, CX
0x05		JGE end
0x06		MOV DX, **[BX + SI]**
0x07		ADD AL, DL
0x08		INC SI
0x09		JMP while
0x0A	end:	DIV CL
0x0B		MOV [prom], AL
	DATA:	
0x0C	**arreglo**	**6**
0x0D		**7**
0x0E		**4**
0x0F		**5**
0x10		**3**
0x11	n	5
0x12	prom	0

Cuando el procesador al ir a buscar un dato lo encuentra en el primer nivel de la jerarquía, se denomina un **hit**; si no lo encuentra, se denomina un **miss**. En caso de un miss, se debe acceder al siguiente nivel de la jerarquía a buscar el dato. El **hit rate** es la proporción de accesos a memoria en los cuales se encontró el dato en el primer nivel (osea hubo un hit); el **miss rate** es la proporción de accesos a memoria en los cuales no se encontró el dato en el primer nivel (osea hubo un miss), y es equivalente a 1 - hit rate.

La razón para definir la jerarquía de memoria era reducir los tiempos de accesos, por lo que es importante definir métricas para medir esto. El tiempo que le toma la CPU acceder a un dato encontrado en el primer nivel (un hit) se conoce como **hit time**, que incluye además del acceso determinar si hubo hit o miss. El tiempo que se necesita para reemplazar un bloque del nivel más alto por un bloque del siguiente nivel, en el caso de haber un miss, más el tiempo de acceso de la CPU al bloque agregado se conoce como **miss penalty**.

3. Memoria caché

3.1. Fundamentos de la memoria caché

La memoria que ocupa el primer nivel de la jerarquía en un computador se conoce como **memoria caché**. La palabra caché se utiliza no solo en este contexto sino también en cualquier situación en la cual se tiene un lugar de almacenamiento intermedio, el cual se utiliza aprovechando el principio de localidad (por ejemplo, el caché del browser, que almacena archivos y páginas web descargadas recientemente). En el contexto del computador la memoria caché será la memoria que se comunique directamente con la CPU: todo dato que pase por la CPU, pasará también por caché. De esta forma, en los diagramas de computador antes observados basta reemplazar la RAM por caché y el resto del funcionamiento del computador se mantiene igual.

Para entender el funcionamiento de la memoria caché, basta entender como funciona su comunicación con la CPU y con el siguiente nivel en la jerarquía (i.e. la memoria principal). Lo que ocurra más abajo en la jerarquías (e.g. comunicación entre la memoria y el disco) es irrelevante para la caché, dado que esto será transparente. De esta forma la caché sabe que en caso de no encontrar un dato solicitado, debe ir a pedirlo al siguiente nivel y luego almacenarlo.

A nivel de la CPU, lo único que conoce de la jerarquía de memoria es el espacio de direcciones de la memoria principal (segundo nivel de la jerarquía). De esta forma, una instrucción como `MOV AX, [120]` siempre se referirá a la dirección 120 de la memoria principal. Para poder lograr este nivel de transparencia la caché cuenta con un **controlador de caché**, encargado de gestionar la transferencia entre la caché y la memoria principal. Son dos funciones las principales del controlador:

- Mecanismo de acceso a los datos: Ante una solicitud de lectura de un dato de memoria, la caché debe saber ubicar una determinada dirección de la memoria principal (e.g 120) entre sus datos almacenados. Es decir, para una dirección de memoria, la caché debe saber tanto si tiene ese dato almacenado como donde lo tiene. En caso de no tener un cierto dato requerido, la cache debe encargarse de ir a buscar el dato a la memoria principal y guardarlo en una determinada posición, que después le permita ubicarlo si se solicita de nuevo.
- Políticas de escritura: Cuando la CPU escribe un dato en memoria (e.g `MOV [120], AX`), la caché debe en algún momento actualizar ese valor en la memoria principal, para asegurar consistencia, y por tanto debe tener definida una política de escritura que indica cuando se realizará dicha actualización.

3.1.1. Mecanismo de acceso a los datos

Para poder almacenar bloques desde la memoria principal, la caché debe definir una **función de correspondencia** entre las direcciones de la memoria principal y su propia memoria. La función de correspondencia más simple se conoce como **direcly mapped**. Con este mecanismo, cada **bloque** de la memoria principal tiene asociado un solo **bloque** en la memoria caché. La caché identificará cada bloque interno con un **índice** (equivalente a una dirección, pero para bloques). La fórmula de asociación entre un bloque de memoria y un índice de caché depende de los siguientes factores:

- Tamaño de cada bloque de caché. Por simplicidad siempre será un potencia de 2.
- Cantidad de bloques en caché. Por simplicidad siempre será un potencia de 2.
- Dirección del bloque en memoria. La dirección del bloque de memoria será la dirección de la primera palabra del bloque.

Para observar como se realiza la asociación entre memoria y caché, revisemos el siguiente ejemplo: se tiene una memoria principal de 32 bytes (32 palabras), una caché de 8 bytes, con 4 bloques cada uno de 2 bytes (2 palabras). El 5to bloque de memoria está asociado a la dirección $8 = 01000$. Para obtener su asociación en caché hay que descomponer la dirección de la siguiente forma:

- Como la caché tiene bloques de 2 palabras $= 2^{\mathbf{1}}$, se requiere **1** bit para determinar la posición de la palabra dentro del bloque. En este ejemplo entonces, el bit menos significativo de la dirección se usará para indicar la posición dentro del bloque, osea 0100**0**, por lo que en este caso, la palabra asociada a la dirección $8 = 01000$ estará almacenada en la palabra 0 del bloque.
- Como la caché tiene 4 bloques $= 2^{\mathbf{2}}$, se requieren **2** bits para determinar el índice del bloque dentro de la caché. Se usarán los siguientes dos bits de la dirección para determinar el índice, osea 01**00**0, por lo que en este caso, el bloque asociado a la dirección $8 = 01000$ se almacenará en el bloque 00 de caché.

En el diagrama de la figura 2 se observa la memoria y caché del ejemplo y distintas asociaciones de bloques de memoria con bloques de caché. Se puede observar que un mismo bloque de caché será usado para varios bloques de memoria. El problema de esto, es que para que la CPU sepa que palabra de caché corresponde a que palabra de memoria, se requiere almacenar información adicional. Esta información adicional que se almacena se conoce como **tag**. El tag corresponderá al resto de los bits de la dirección del bloque, que no fueron ocupados para determinar el índice o palabra en caché. En el caso del ejemplo anterior, el tag para el bloque $8 = 01000$ sería **01**000. Este valor se almacena en un lugar especial de la caché reservado para los tags.

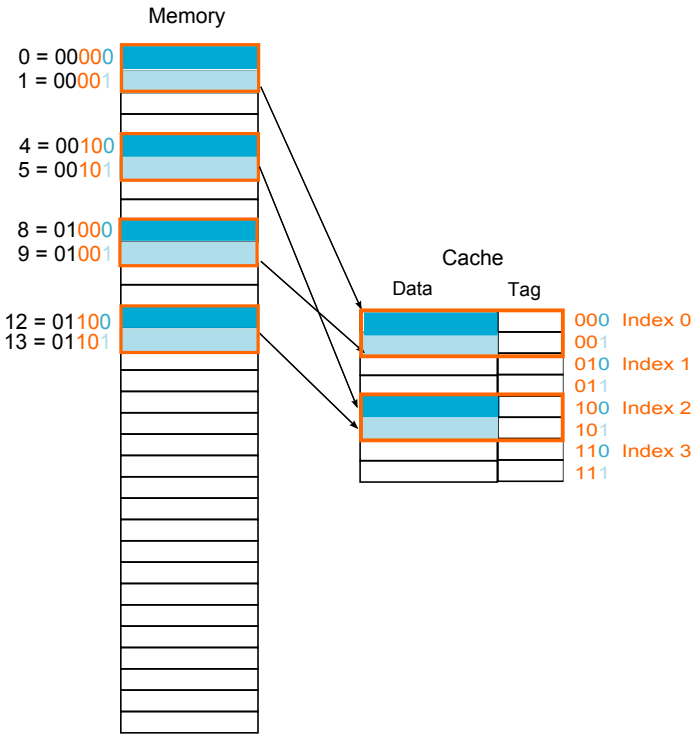


Figura 2: En una caché directly mapped, cada bloque de memoria tendrá una ubicación fija en la caché. Cada posición de caché, por su parte, podrá tener uno de varios bloques de memoria.

Un elemento adicional que se debe almacenar para cada palabra es un bit de validez o **valid bit**. Este bit indica si los valores almacenados en caché son válidos, lo que es importante al comenzar a llenar la caché, ya que cuando la caché está vacía, las palabras que tienen almacenadas no son válidas.

A continuación se muestra un ejemplo del funcionamiento de una caché de 4 bloques de 2 palabras para un programa que accede a la siguiente secuencia de direcciones de memoria: 12, 13, 14, 4, 12, 0.

- En un comienzo la caché comienza vacía, y por tanto todos los bits de validez están en 0, indicando que los datos actuales no son válidos.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	0		
	1	0		
01	0	0		
	1	0		
10	0	0		
	1	0		
11	0	0		
	1	0		

- El primer acceso es a la dirección 12 = 01100, por lo tanto se almacenará el bloque de memoria en el bloque de caché con índice 10. La palabra asociada a la dirección 01100 se guardará en la ubicación 0. Adicionalmente se guardará también el dato de la dirección 13 = 01101 que es parte del bloque de memoria, y se almacenará en el mismo bloque de caché, pero en la ubicación 1. El tag para ambas palabras es 01.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	0		
	1	0		
01	0	0		
	1	0		
10	0	1	01	Mem[12]
	1	1	01	Mem[13]
11	0	0		
	1	0		

- El siguiente acceso es a la dirección 13 = 01100. Como en el paso anterior el bloque que se copió contenía ya este valor, no hay que ir a buscarlo a memoria y la CPU puede obtener el dato directamente de caché.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	0		
	1	0		
01	0	0		
	1	0		
10	0	1	01	**Mem[12]**
	1	1	01	**Mem[13]**
11	0	0		
	1	0		

- El siguiente acceso es a la dirección 14 = 01110, por lo tanto se almacenará el bloque de memoria en el bloque de caché con índice 11. La palabra asociada a la dirección 01110 se guardará en la ubicación 0. Adicionalmente se guardará también el dato de la dirección 15 = 01111 que es parte del bloque de memoria, y se almacenará en el mismo bloque de caché, pero en la ubicación 1. El tag para ambas palabras es 01.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	0		
	1	0		
01	0	0		
	1	0		
10	0	1	01	Mem[12]
	1	1	01	Mem[13]
11	0	1	01	**Mem[14]**
	1	1	01	**Mem[15]**

- El siguiente acceso es a la dirección 4 = 00100, por lo tanto se almacenará el bloque de memoria en el bloque de caché con índice 10. Como este bloque ya está ocupado, hay que eliminar lo que se tenía guardado y reemplazarlo por el nuevo bloque traído de memoria, La palabra asociada a la dirección 00100 se guardará en la ubicación 0. Adicionalmente se guardará también el dato de la dirección 5 = 00101 que es parte del bloque de memoria, y se almacenará en el mismo bloque de caché, pero en la ubicación 1. El tag para ambas palabras es 00.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	0		
	1	0		
01	0	0		
	1	0		
10	0	1	00	**Mem[4]**
	1	1	00	**Mem[5]**
11	0	1	01	Mem[14]
	1	1	01	Mem[15]

- El siguiente acceso es a la dirección 12 = 01100, por lo tanto se almacenará el bloque de memoria en el bloque de caché con índice 10. Como este bloque ya está ocupado, hay que eliminar lo que se tenía guardado y reemplazarlo por el nuevo bloque traído de memoria, La palabra asociada a la dirección 01100 se guardará en la ubicación 0. Adicionalmente se guardará también el dato de la dirección 13 = 01101 que es parte del bloque de memoria, y se almacenará en el mismo bloque de caché, pero en la ubicación 1. El tag para ambas palabras es 01.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	0		
	1	0		
01	0	0		
	1	0		
10	0	1	01	**Mem[12]**
	1	1	01	**Mem[13]**
11	0	1	01	Mem[14]
	1	1	01	Mem[15]

- El último acceso es a la dirección 0 = 00000, por lo tanto se almacenará el bloque de memoria en el bloque de caché con índice 00. La palabra asociada a la dirección 00000 se guardará en la ubicación 0. Adicionalmente se guardará también el dato de la dirección 1 = 00001 que es parte del bloque de memoria, y se almacenará en el mismo bloque de caché, pero en la ubicación 1. El tag para ambas palabras es 00.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	1	00	**Mem[0]**
	1	1	00	**Mem[1]**
01	0	0		
	1	0		
10	0	1	01	Mem[12]
	1	1	01	Mem[13]
11	0	1	01	Mem[14]
	1	1	01	Mem[15]

3.2. Políticas de escritura

En el caso que la CPU quiera escribir un dato en memoria además de preocuparse del acceso, debe encargarse de actualizar la memoria principal para que está mantenga consistencia en la información. Existen dos políticas usadas para realizar esta actualización

- Write-through: en esta política cada escritura en caché, se actualiza inmediatamente en memoria. La ventaja de esta política es que la memoria principal nunca tendrá datos inconsistentes. La desventaja es que cada escritura involucrará una espera considerable para la CPU ya que deberá esperar no tanto la escritura en caché como la escritura en memoria principal.
- Write-back: en esta política las escrituras se realizan en principio sólo en caché. El valor se actualizará en memoria sólo cuando el bloque que estaba en caché vaya a ser reemplazado. La ventaja de este mecanismo es que es mucho más eficiente, ya que las escrituras se realizan solo en los reemplazos y no siempre. La desventaja es que la memoria principal puede quedar con datos inconsistentes, lo que puede afectar si un dispositivo de I/O intenta acceder a esta mediante DMA. Para que esta política no genere errores de consistencia se deben agregar mecanismos que realicen la actualización ante un posible caso de inconsistencia, como en un acceso por DMA.

3.3. Mejoras al rendimiento de la memoria caché

3.3.1. Funciones de correspondencia

Una caché con función de correspondencia **directly mapped** presenta la desventaja de que al tener un mapeo fijo entre bloques de memoria y caché no se está aprovechando al máximo la disponibilidad de la caché. El problema ocurre por la posible contención que pueda ocurrir entre dos bloques de memoria mapeados a un mismo bloque de caché, los cuales aunque haya espacio disponible en otro bloque, estarán compitiendo por el espacio del mismo bloque. Este factor afecta directamente al hit rate de la caché, y por tanto es necesario definir funciones de correspondencia más eficientes.

Existen dos funciones de correspondencia además de directly mapped, que intentan solucionar el problema de la contención: **fully associative** y n-way associative:

Fully associative

En una memoria con función de correspondencia fully associative, cada bloque de memoria puede asociarse a cualquier bloque de la caché. De esta manera se aprovecha al máximo la disponibilidad

de espacio en la caché y se evita el problema de la contención. El problema de esta función de correspondencia, es que ante un requerimiento de memoria de la CPU, ahora se vuelve más difícil encontrar la palabra en caché. En el caso de direclty mapped, ocupando los bits de la dirección era posible llegar a la única posible ubicación de la palabra en caché. Una vez en esa posición se podía revisar el tag y con eso determinar si estaba o no la palabra. En el caso de fully associative, no es posible ir directamente a la posición dado que puede ser cualquiera. Para saber si una palabra está en caché es necesario buscar en todas las posiciones, y además es necesario guardar un tag más grande, que contenga toda la dirección menos los bits de ubicación de la palabra en el bloque.

Para que la detección de la palabra no sea tan lenta, se utilizan circuitos comparadores que funcionan en paralelo, lo que complejiza el hardware de la caché, haciéndola más cara. Además, a pesar de hacer la comparación en paralelo, el hecho de realizarla involucra un gasto de tiempo, por lo que está función, aun cuando aumenta el hit rate, aumenta también el hit time. De todas maneras la ganancia en velocidad por el aumento del hit rate supera lo que se pierda por aumentar el hit time. La principal desventaja de esta función es la necesidad del hardware complejo de comparación.

A continuación se muestra un ejemplo del funcionamiento de una caché de 4 bloques de 2 palabras con función de correspondencia fully associative para un programa que accede a la siguiente secuencia de direcciones de memoria: 12, 13, 14, 4, 12, 0.

- En un comienzo la caché comienza vacía, y por tanto todos los bits de validez están en 0, indicando que los datos actuales no son válidos.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	0		
	1	0		
01	0	0		
	1	0		
10	0	0		
	1	0		
11	0	0		
	1	0		

- El primer acceso es a la dirección 12 = 01100, y como la caché es fully associative, el bloque asociado a esa palabra (Mem[12] y Mem[13]) se guardará en el primer bloque disponible de la caché (Índice 00). El tag para ambas palabras es 0110, es decir todas la dirección menos el bit de ubicación de la palabra. Como no se encontró el valor en memoria, este acceso es un miss.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	1	0110	**Mem[12]**
	1	1	0110	**Mem[13]**
01	0	0		
	1	0		
10	0	0		
	1	0		
11	0	0		
	1	0		

- El siguiente acceso es a la dirección 13 = 01100. Como en el paso anterior el bloque que se copió contenía ya este valor, no hay que ir a buscarlo a memoria y la CPU puede obtener el dato directamente de caché. Como si se encontró el valor en memoria, este acceso es un hit.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	1	0110	Mem[12]
	1	1	0110	Mem[13]
01	0	0		
	1	0		
10	0	0		
	1	0		
11	0	0		
	1	0		

- El siguiente acceso es a la dirección 14 = 01110, por lo tanto se almacenará el bloque de memoria (Mem[14 y Mem[15]) en el siguiente bloque de caché disponible (índice 01). El tag para ambas palabras es 0111. Como no se encontró el valor en memoria, este acceso es un miss.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	1	0110	Mem[12]
	1	1	0110	Mem[13]
01	0	1	0111	**Mem[14]**
	1	1	0111	**Mem[15]**
10	0	0		
	1	0		
11	0	0		
	1	0		

- El siguiente acceso es a la dirección 4 = 00100, por lo tanto se almacenará el bloque de memoria (Mem[4] y Mem[5] en el siguiente bloque de caché disponible. El tag para ambas palabras es 0010. Como no se encontró el valor en memoria, este acceso es un miss.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	1	0110	Mem[12]
	1	1	0110	Mem[13]
01	0	1	0111	Mem[14]
	1	1	0111	Mem[15]
10	0	1	0010	**Mem[4]**
	1	1	0010	**Mem[5]**
11	0	0		
	1	0		

- El siguiente acceso es a la dirección 12 = 01100. Como el bloque que se copió ya tenía ese valor, no hay que ir a buscarlo a memoria y la CPU puede obtener el dato directamente de caché. Como si se encontró el valor en memoria, este acceso es un hit.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	1	0110	Mem[12]
	1	1	0110	Mem[13]
01	0	1	0111	Mem[14]
	1	1	0111	Mem[15]
10	0	1	0010	Mem[4]
	1	1	0010	Mem[5]
11	0	0		
	1	0		

- El siguiente acceso es a la dirección 0 = 00000, por lo tanto se almacenará el bloque de memoria (Mem[0] y Mem[1] en el siguiente bloque de caché disponible. El tag para ambas palabras es 0000. Como no se encontró el valor en memoria, este acceso es un miss.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	1	0110	Mem[12]
	1	1	0110	Mem[13]
01	0	1	0111	Mem[14]
	1	1	0111	Mem[15]
10	0	1	0010	Mem[4]
	1	1	0010	Mem[5]
11	0	1	0000	**Mem[0]**
	1	1	0000	**Mem[1]**

El hit rate de esta secuencia de accesos fue de $\frac{2}{6}$ que es mejor que en el caso directly mapped donde había sido $\frac{1}{6}$.

N-way associative

Con directly mapped, la caché es simple, pero a costa de un bajo hit rate. Con fully associative, la caché es compleja, pero logra un alto hit rate. La función de correspondencia n-way associative busca ser un punto medio entre estos dos esquemas, tratando de simplificar el hardware de la caché, pero sin perder las ventajas de la fully associative respecto al hit rate. Para esto, la caché se divide lógicamente en **conjuntos** o sets, siendo cada uno un group de bloques. Cada bloque de memoria tendrá un mapeo directo a un sólo conjunto, pero dentro del conjunto podrá ubicarse en cualquier bloque disponible (Figura 3). De esta forma, al tener el mapeo directo al conjunto, se reducen las comparaciones al momento de buscar si está un dato o no en caché, y al tener libertad de elegir en que bloque del conjunto almacenar, se mantienen niveles de hit rate altos.

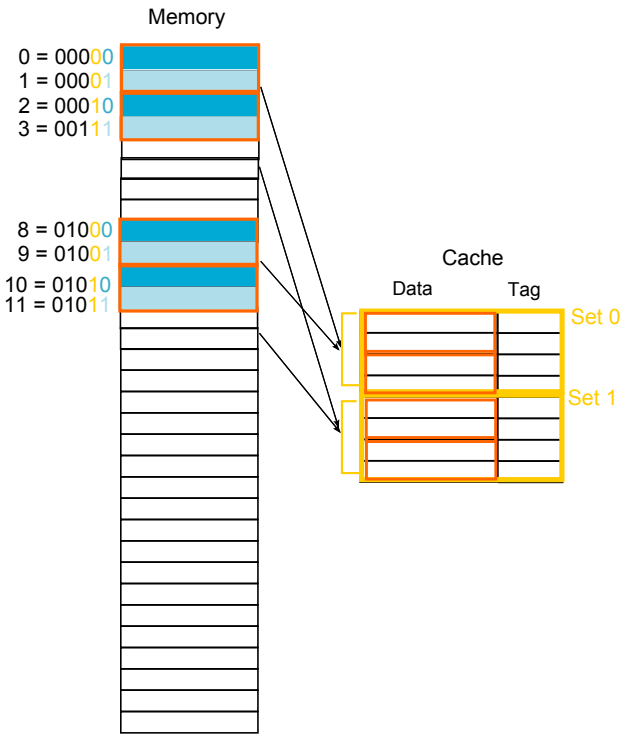


Figura 3: En una caché n-way associative, cada bloque de memoria estará asociado a un conjunto de bloques de la caché, y dentro del conjunto se podrá ubicar en cualquier lugar. En este caso, la caché es 2-way associative, es decir, tiene conjuntos de 2 bloques cada uno.

Dependiendo de cuantos bloques tenga cada conjunto, se tendrán distintos tipos de funciones n-way associative. Si se tienen 4 bloques por conjunto, se denominará 4-way associative; 2 bloques por conjunto, 2-way associative y así. A continuación se muestra un ejemplo del funcionamiento de una caché de 4 bloques de 2 palabras con función de correspondencia 2-way associative para un programa que accede a la siguiente secuencia de direcciones de memoria: 12, 13, 14, 4, 12, 0.

- En un comienzo la caché comienza vacía, y por tanto todos los bits de validez están en 0, indicando que los datos actuales no son válidos.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	0		
	1	0		
01	0	0		
	1	0		
10	0	0		
	1	0		
11	0	0		
	1	0		

- El primer acceso es a la dirección 12 = 01100, y como la caché es 2-way associative, el bloque asociado a esa palabra (Mem[12] y Mem[13]) se guardará en el primer bloque disponible del

conjunto 0. El tag para ambas palabras es 011. Como no se encontró el valor en memoria, este acceso es un miss.

Conjunto	Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
0	00	0	1	011	**Mem[12]**
		1	1	011	**Mem[13]**
	01	0	0		
		1	0		
1	10	0	0		
		1	0		
	11	0	0		
		1	0		

- El siguiente acceso es a la dirección 13 = 01100. Como en el paso anterior el bloque que se copió contenía ya este valor, no hay que ir a buscarlo a memoria y la CPU puede obtener el dato directamente de caché. Como si se encontró el valor en memoria, este acceso es un hit.

Conjunto	Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
0	00	0	1	011	Mem[12]
		1	1	011	Mem[13]
	01	0	0		
		1	0		
1	10	0	0		
		1	0		
	11	0	0		
		1	0		

- El siguiente acceso es a la dirección 14 = 01110, por lo tanto se almacenará el bloque de memoria (Mem[14 y Mem[15]) en el siguiente bloque de caché disponible del conjunto 1. El tag para ambas palabras es 011. Como no se encontró el valor en memoria, este acceso es un miss.

Conjunto	Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
0	00	0	1	011	Mem[12]
		1	1	011	Mem[13]
	01	0	0		
		1	0		
1	10	0	1	011	**Mem[14]**
		1	1	011	**Mem[15]**
	11	0	0		
		1	0		

- El siguiente acceso es a la dirección 4 = 00100, por lo tanto se almacenará el bloque de memoria (Mem[4] y Mem[5] en el siguiente bloque de caché disponible del conjunto 0. El tag para ambas palabras es 001. Como no se encontró el valor en memoria, este acceso es un miss.

Conjunto	Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
0	00	0	1	011	Mem[12]
		1	1	011	Mem[13]
	01	0	0	001	**Mem[4]**
		1	0	001	**Mem[5]**
1	10	0	1	011	Mem[14]
		1	1	011	Mem[15]
	11	0	0		
		1	0		

- El siguiente acceso es a la dirección 12 = 01100. Como el bloque que se copió ya tenía ese valor, no hay que ir a buscarlo a memoria y la CPU puede obtener el dato directamente de caché. Como si se encontró el valor en memoria, este acceso es un hit.

Conjunto	Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
0	00	0	1	011	Mem[12]
		1	1	011	Mem[13]
	01	0	0	001	Mem[4]
		1	0	001	Mem[5]
1	10	0	1	011	Mem[14]
		1	1	011	Mem[15]
	11	0	0		
		1	0		

- El siguiente acceso es a la dirección 0 = 00000, por lo tanto se almacenará el bloque de memoria (Mem[0] y Mem[1] en el siguiente bloque de caché disponible del conjunto 0. Como el conjunto está lleno, se debe reemplazar un bloque, en este casos s reemplazará el primero que ingresó. El tag para ambas palabras es 000. Como no se encontró el valor en memoria, este acceso es un miss.

Conjunto	Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
0	00	0	1	000	**Mem[0]**
		1	1	000	**Mem[1]**
	01	0	0	001	Mem[4]
		1	0	001	Mem[5]
1	10	0	1	011	Mem[14]
		1	1	011	Mem[15]
	11	0	0		
		1	0		

El hit rate de esta secuencia de accesos fue de $\frac{2}{6}$ que es equivalente al ejemplo con fully associative.

3.3.2. Algoritmos de reemplazo

Para las funciones de correspondencia fully associative y n-way associative, hay que definir una política de que bloque reemplazar en caso de no haber más espacio, lo que se conoce como **algoritmos de reemplazo**. Existen diversos algoritmos de reemplazo, que en distintas circunstancias pueden representar una mejor alternativa:

First-in First-out (FIFO)

El algoritmo de reemplazo más simple se conoce como first-in first-out o FIFO, en el cual, el primer bloque que entró va a ser el primero en salir. Para implementar este algoritmo, se debe agregar información que indique el orden con que entraron los bloques para ir sacándolos acordemente.

A continuación se muestra un ejemplo del funcionamiento del algoritmo de reemplazo en una caché fully associative, llenada inicialmente con la secuencia: 12, 13, 14, 4, 12, 0 (el desglose de la secuencia está en el ejemplo de la sección anterior). Supongamos ahora que luego de esa secuencia viene el acceso a la secuencia 14, 7, 17:

- Lo primero es determinar el orden actual de los bloques de caché para saber a cual le corresponde salir, lo que se observa a continuación:

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato	Orden
00	0	1	0110	Mem[12]	1
	1	1	0110	Mem[13]	
01	0	1	0111	Mem[14]	2
	1	1	0111	Mem[15]	
10	0	1	0010	Mem[4]	3
	1	1	0010	Mem[5]	
11	0	1	0000	Mem[0]	4
	1	1	0000	Mem[1]	

- El primer acceso es al bloque 14-15, el cual ya está en caché por lo tanto no hay reemplazo. El acceso al bloque no afecta al orden FIFO:

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato	Orden
00	0	1	0110	Mem[12]	1
	1	1	0110	Mem[13]	
01	0	1	0111	Mem[14]	2
	1	1	0111	Mem[15]	
10	0	1	0010	Mem[4]	3
	1	1	0010	Mem[5]	
11	0	1	0000	Mem[0]	4
	1	1	0000	Mem[1]	

- Como el primer bloque en entrar fue el de las direcciones 12 y 13, ese bloque se reemplazará por el bloque de las direcciones 6 y 7. Luego del reemplazo, se actualiza el orden:

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato	Orden
00	0	1	0011	**Mem[6]**	4
	1	1	0011	**Mem[7]**	
01	0	1	0111	Mem[14]	1
	1	1	0111	Mem[15]	
10	0	1	0010	Mem[4]	2
	1	1	0010	Mem[5]	
11	0	1	0000	Mem[0]	3
	1	1	0000	Mem[1]	

- Como el primer bloque en entrar de los que están fue el de las direcciones 14 y 15, ese bloque se reemplazará por el bloque de las direcciones 16 y 17. Luego del reemplazo, se actualiza el orden:

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato	Orden
00	0	1	0011	Mem[6]	3
	1	1	0011	Mem[7]	
01	0	1	1000	**Mem[16]**	4
	1	1	1000	**Mem[17]**	
10	0	1	0010	Mem[4]	1
	1	1	0010	Mem[5]	
11	0	1	0000	Mem[0]	2
	1	1	0000	Mem[1]	

Least-frequently used (LFU)

El algoritmo least-frequently used (LFU) reemplazará el bloque que ha sido accedido menos frecuentemente. La idea es que si un bloque ha sido accedido varias veces, es probable que vuelva a ser accedido (principio de localidad temporal). Para implementar este algoritmo, se debe almacenar un contador de accesos a cada bloque, de manera de saber que bloque ha sido accedido menos. En caso de haber dos bloques empatados en menor cantidad de acceso, se ocupará otro algoritmo para desempatar, generalmente FIFO.

A continuación se muestra un ejemplo del funcionamiento del algoritmo de reemplazo en una caché fully associative, llenada inicialmente con la secuencia: 12, 13, 14, 4, 12, 0 (el desglose de la secuencia está en el ejemplo de la sección anterior). Supongamos ahora que luego de esa secuencia viene el acceso a la secuencia 14, 7, 17:

- Lo primero es determinar el orden actual de los bloques y cuantos accesos han tenido, lo que se observa a continuación:

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2343 Arquitectura de Computadores

Multiprogramación

1. Motivación

Una de las ventajas de tener una máquina multipropósito como un computador es la posibilidad de correr múltiples programas que realicen distintas operaciones. Para lograr que un computador ejecute y almacene múltiples programas son necesarias diversas mejoras y modificaciones al hardware.

2. Multiprogramación

El concepto de multiprogramación se refiere a la idea general de poder cargar múltiples programas dentro de un mismo computador para que sean ejecutados en un determinado momento. Para lograr manejar múltiples programas, es necesario primero definir que compone a un programa. En general, podemos decir que un programa está compuesto por dos partes: su **representación en memoria** que incluye el código, datos y stack del programa, y su **estado de ejecución** que incluye los valores almacenados en los registros de la CPU (PC, registros acumuladores, SP, Status register, etc.) que indican el estado actual del programa en la máquina. Para lograr trabajar con múltiples programas, entonces, es necesario permitir el manejo de múltiples representaciones en memoria y de múltiples estados de ejecución.

2.1. Manejo de múltiples representaciones en memoria

Supongamos que queremos representar en memoria dos programas que originalmente funcionaban como programa único de un determinado computador. Los códigos de ambos programas (P1 y P2) se muestran a continuación:

P1:

```
MOV A, (100)
MOV B, (101)
ADD A,B
MOV (102), A
```

P2:

```
MOV A, (100)
MOV B, (101)
SUB A,B
MOV (102), A
```

Ambos programas tienen sus instrucciones y datos almacenados en memoria. En este caso particular, además, ambos programas ocupan las mismas direcciones de memoria para almacenar sus tres variables. Es claro que no es posible utilizar el mismo espacio físico de memoria para ambos programas, por lo que es necesario desarrollar alguna modificación para poder almacenar los dos programas.

Una primera solución que se puede pensar es modificar los programas de manera de que las instrucciones y datos estén en ubicaciones distintas de memoria. Con este esquema, se podría almacenar en una parte de la memoria un programa completo, y en otra parte otro programa, como se observa en el diagrama de la figura 1

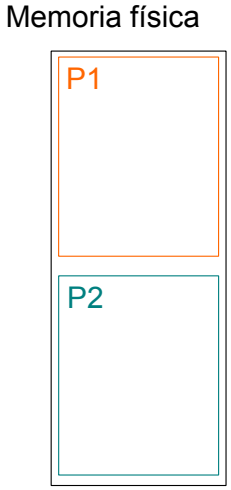


Figura 1: Esquema simple de almacenamiento de dos programas en memoria.

Para lograr que este esquema funcione, es necesario colocar el código del segundo programa luego del espacio del primer programa y además es necesario modificar las posiciones de las direcciones de memoria de las variables usadas por el programa, por ejemplo a las direcciones 200, 201 y 202, como se observa en el siguiente código del programa 2 modificado:

P2:

```
MOV A, (200)
MOV B, (201)
SUB A,B
MOV (202), A
```

Con este esquema entonces, es necesario modificar las direcciones de memoria originales de al menos uno de los programas, como se observa en el siguiente diagrama:

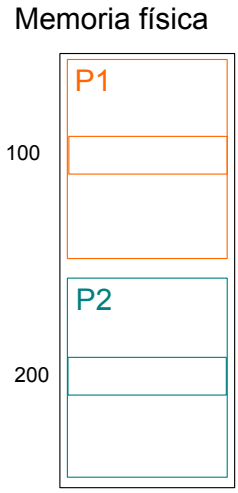


Figura 2: En el esquema simple de manejo de múltiples programas en memoria, es necesario modificar la ubicación en memoria del código y los datos de al menos uno de los programas.

Este esquema presente tres problemas importantes:

- Es necesario modificar uno de los programas para poder hacerlo calzar en su nueva ubicación de memoria.
- No existe protección de la memoria: el programa 1 puede escribir en los datos del programa 2 y vice-versa, lo que claramente no es deseado.
- Cada programa está limitado solo a ocupar una parte de la memoria y no a todo el espacio direccionable.

Para solucionar estas tres dificultades presentados por este esquema simple, los sistemas de multiprogramación utilizan el concepto de **memoria virtual**. En un sistema de memoria virtual, se distinguen las direcciones que ocupa internamente un programa (direcciones virtuales) de las direcciones físicas donde se almacena la información. Cada programa tendrá un espacio virtual de direcciones, que corresponde al tamaño de la espacio de direcciones de la memoria completa habitualmente. De esta forma, cada programa tiene la percepción de ser el único programa en la máquina.

En la práctica sin embargo, cada programa sigue estando mapeado a un lugar físico distinto en la memoria física, pero a nivel del programa esto es transparente y por tanto el programa no se preocupa de saber en que ubicación física está, ya que solo le interesa su espacio virtual (figura 3).

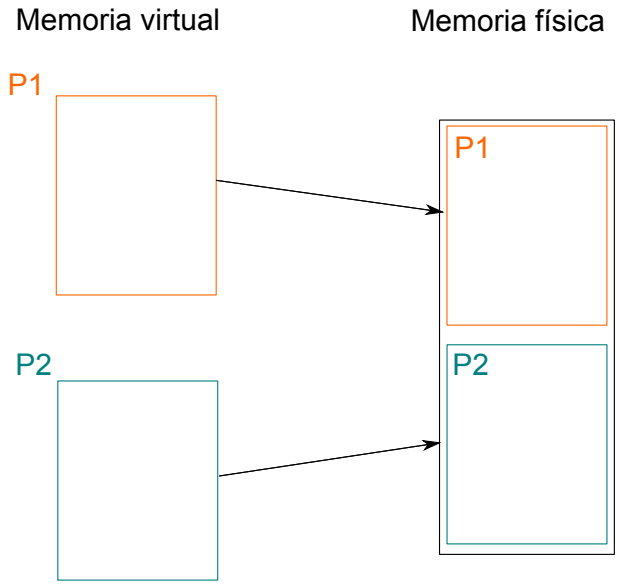


Figura 3: Cada programa tiene un espacio virtual de memoria que se mapea a una ubicación física.

De esta forma, en el ejemplo de los programas anteriores no es necesario modificar las direcciones del programa 2, ya que para éste, sus variables siguen estando ubicadas en las direcciones 100, 101 y 102, de su espacio virtual. En la práctica, estas direcciones son mapeadas internamete a otras ubicaciones físicas (200,201 y 202 por ejemplo), pero a nivel del código esto no afecta (figura 4).

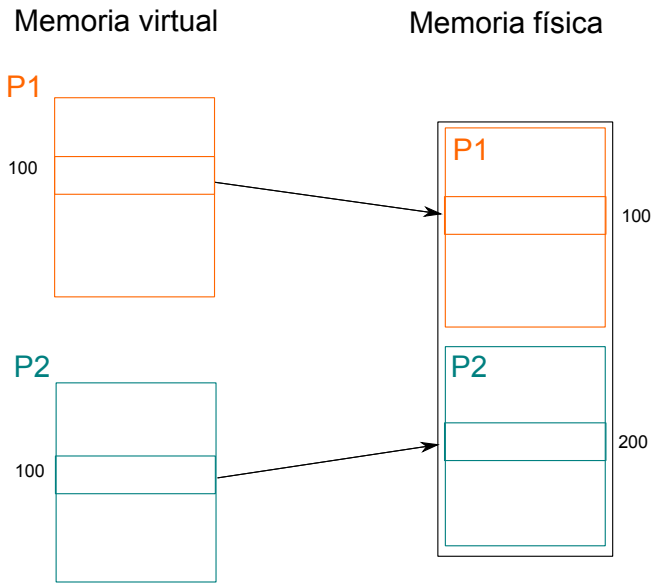


Figura 4: Cada programa puede acceder a las direcciones de memoria del espacio virtual completo, sin preocuparse de topar con direcciones de otro programa.

El sistema de memoria virtual es capaz de solucionar los tres problemas antes descritos:

- No es necesario modificar los programas ya que cada programa tiene la impresión de estar sólo en el computador.

- Al poder acceder solo a su espacio virtual, un programa no puede escribir en los datos de otro, agregando protección de memoria.
- Cada programa puede direccionar el espacio completo de memoria, y por tanto no hay problemas de limitación.

Para implementar un sistema de memoria virtual, es necesario agregar a la CPU un componente de hardware que realice la traducción de una dirección virtual a una dirección física. Este componente se conoce como **Memory Management Unit (MMU)** y es parte de la CPU. La MMU se encargará de traducir las solicitudes de memoria que vengan desde la CPU, mapeando para el programa actual la dirección física correspondiente.

En el caso de los programas anteriores, una solicitud a la dirección virtual 100 de parte del programa 2, será traducida como solicitud a la dirección física 200:

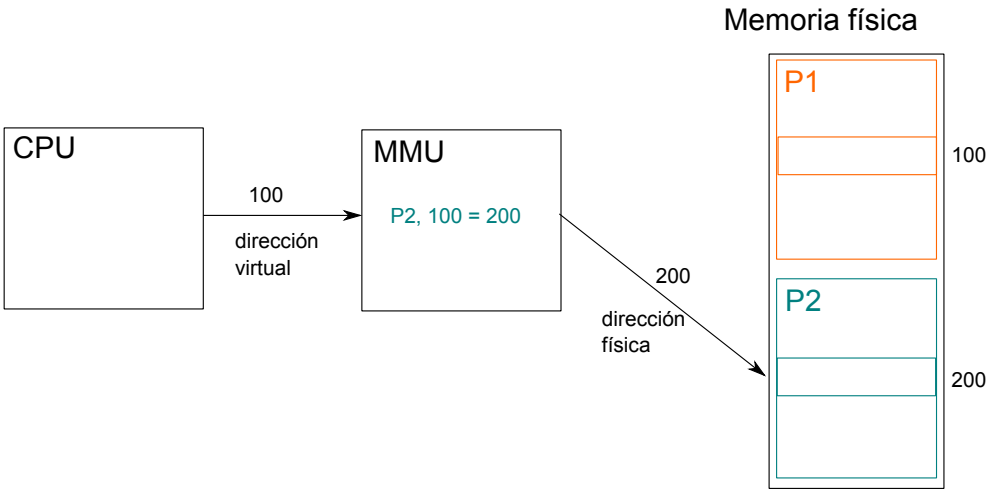


Figura 5: Traducción de dirección virtual a física.

Una solicitud a la dirección virtual 100 de parte del programa 1, en cambio, será traducida como solicitud a la dirección física 100.

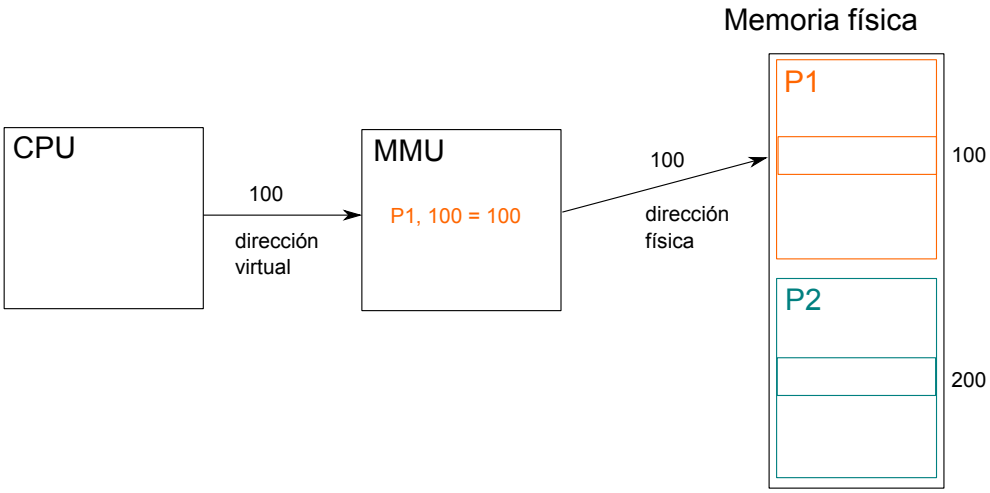


Figura 6: Traducción de dirección virtual a física.

Para realizar la traducción la MMU debe almacenar una tabla que tenga la asociación virtual - física. La opción más simple para esto es almacenar todos los pares de direcciones virtual-física, lo que en el caso de los programas 1 y 2 serían los siguientes:

Dirección virtual	Dirección física
100	100
101	101
102	102

Tabla 1: Tabla de mapeo virtual-físico del programa 1.

Dirección virtual	Dirección física
100	200
101	201
102	202

Tabla 2: Tabla de mapeo virtual-físico del programa 2.

El problema de almacenar todos los mapeos posible está en que se necesita para cada programa una tabla con tantas entradas como direcciones virtuales disponibles hay, y por tanto se requeriría para cada programa un espacio de almacenamiento igual al tamaño dela memoria física, lo que claramente no es implementable en la práctica.

Para solucionar el problema del tamaño de la tabla de mapeo virtual-físico, se agrega el concepto de **paginación**. La paginación corresponde a dividir la memoria en bloques de palabras contiguos conocidos como **páginas** en el espacio virtual o **marcos** en el espacio físico. De esta manera, cada programa tendrá asociada una cierta cantidad de páginas de memoria virtual, las cuales estarán mapeadas a marcos físicos (figura 7). Con este esquema, las tablas de mapeo, denominadas **tablas de páginas**, pueden tener un tamaño razonable lo que permite implementar el sistema en la práctica.

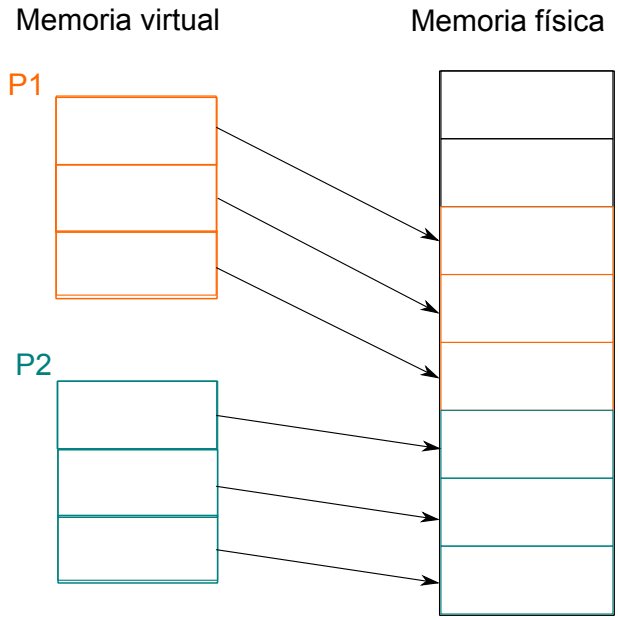


Figura 7: En un sistema de paginación, cada programa utiliza páginas de memoria virtual, las cuales están mapeadas a marcos de memoria física.

Suponiendo por ejemplo una memoria física de 256 bytes y páginas de 32 bytes, se tendría 8 marcos físicos posibles. Si los programas 1 y 2 del ejemplo anterior se quieren almacenar en esta memoria, y suponiendo que ocupan tres páginas virtuales, las posibles tablas de página serían las siguientes:

Página virtual	Marco físico
0	2
1	3
2	4

Tabla 3: Tabla de página del programa 1.

Página virtual	Marco físico
0	5
1	6
2	7

Tabla 4: Tabla de página del programa 2.

En un sistema de memoria virtual con paginación, la dirección de memoria sera interpretada como dos partes: una que indica el número de página, y otra que indica el offset dentro de la página. En el caso anterior, al ser el espacio de direccionamiento de 8 bits ($2^8 = 256$ palabras en memoria), la dirección de 8 bits se dividirá en los 3 bits más significativos para indicar el número de página ($2^3 = 8$ páginas), y los 5 bits restantes para indicar el offset ($2^5 = 32$ palabras por página).

Para determinar la ubicación física de la dirección virtual 70 del programa 2, por ejemplo, se debe realizar los siguientes pasos:

- Primero es necesario obtener la dirección en binario: 70 = 01000110.
- Los tres primeros bits de la dirección (010 = 2) indican el número de página, los siguientes cinco bits (00110 = 6) indican el offset dentro de la página.
- La página virtual debe ser traducida a un marco físico ocupando la tabla de página correspondiente. En este caso la página 2 esta mapeada al marco 7 = 111.
- Ocupando el número del marco físico (111) más el offset original (00110) se obtiene la dirección física real: 11100110 = 230.

Como se señaló previamente, la paginación se utiliza para reducir el tamaño de las tablas de mapeo. En los computadores personales el tamaño de página es habitualmente de 1KB (2^{10} bytes). Pensando por ejemplo en un computador con 1GB (2^{30} bytes) de memoria , se tendrían $2^{30-10} = 2^{20} = 1M$ páginas, es decir el tamaño de la tabla de página de cada programa sería alrededor de 1MB, lo que es una proporción razonable considerando el tamaño de la memoria.

Las tablas de página de cada programa contienen la información completa del mapeo del espacio virtual del programa al espacio físico. Esto quiere decir que se deben tener almacenadas entradas para todas las posibles páginas, aunque estas no estén utilizadas. Para diferenciar las páginas que está correctamente mapeadas a un marco físico de las que no, se agrega a la tabla de páginas un **bit de validez** el cual cuando tiene el valor 1 indica que esa entrada es válida, y cuando tiene el valor 0 indica que no lo es.

Para el ejemplo de los programas 1 y 2 antes vistos, las tablas de páginas completas serían las siguientes:

Página virtual	Marco físico	Validez
0	2	1
1	3	1
2	4	1
3	x	0
4	x	0
5	x	0
6	x	0
7	x	0

Tabla 5: Tabla de página con bit de validez del programa 1.

Página virtual	Marco físico	
0	5	1
1	6	1
2	7	1
3	x	0
4	x	0
5	x	0
6	x	0
7	x	0

Tabla 6: Tabla de página con bit de validez del programa 2.

Las tablas de páginas de todos los programas son almacenadas en la memoria principal, en el espacio correspondiente al **sistema operativo** que será el programa de controlar que programa está activo, como se verá más adelante. El problema de tener las tablas de página en memoria, es que para cada acceso a memoria de un programa dado, se requieren dos accesos en la práctica: uno para ir a buscar el mapeo virtual-físico en la tabla de página y otro para realizar el acceso real.

Para mejorar el rendimiento en los accesos a memoria, y evitar que siempre ocurra este doble acceso, se agrega una caché especialmente dedicada a almacenar entradas de tabla de página, la cual se conoce como **Translation Lookaside Buffer (TLB)**. La TLB será habitualmente una caché split, fully associative y con algoritmo de reemplazo LRU, y almacenará algunas de las entradas de la tabla de página del programa que actualmente está en ejecución. En el ejemplo anterior, un posible estado de una TLB de dos entradas para la tabla de páginas del programa 2 sería el siguiente:

Página virtual	Marco físico	
0	5	1
2	6	1

Tabla 7: TLB para el programa 2.

La TLB tendrá el mismo funcionamiento que las cachés tradicionales, basándose en los principios de localidad de referencia espacial y temporal para mejorar el rendimiento de los accesos. En caso de ocurrir un hit en la caché, el acceso a memoria tomará el tiempo que se necesite en acceder a la TLB y el tiempo de acceso a memoria. En caso de ocurrir un miss en la caché, el acceso a memoria tomará el tiempo de acceso a la TLB, más el tiempo de ir a memoria a buscar la entrada a la tabla de página y traerla a caché, más el tiempo de acceso al valor real en memoria, es decir, al igual que

con cualquier caché, el miss penalty será mayor que un accesos sin caché, por lo que es fundamental que el hit rate sea alto para que sea efectiva.

Una última situación a considerar en el manejo de memoria virtual corresponde a que ocurre cuando un programa solicita acceso a una página de manera dinámica, mientras se está ejecutando. Al ocurrir esto, el programa le cede el control de la CPU al sistema operativo, el cual se encargará de mapear un marco físico disponible a una nueva página del programa, actualizando la tabla de página correspondiente.

Con este esquema un programa puede solicitar más memoria mientras está siendo ejecutado. Por ejemplo, en el caso anterior de los programas 1 y 2, podría ocurrir que el programa 1 solicite acceso a su página 3, y esta se mapee al marco 0, y que el programa 2 solicite acceso a su página 3, y está se mapee al marco 1, como se observa en el siguiente diagrama y las siguientes tablas de página:

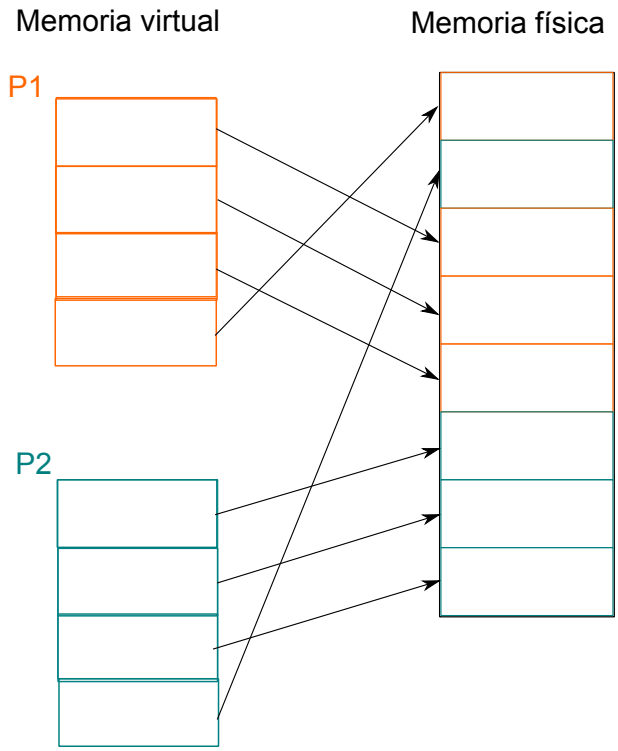


Figura 8: Mapeo de memoria virtual al agregar páginas 3 a ambos programas.

Página virtual	Marco físico	Validez
0	2	1
1	3	1
2	4	1
3	0	1
4	x	0
5	x	0
6	x	0
7	x	0

Tabla 8: Tabla de página del programa 1 luego de acceder a la página 3.

Página virtual	Marco físico	
0	5	1
1	6	1
2	7	1
3	1	1
4	x	0
5	x	0
6	x	0
7	x	0

Tabla 9: Tabla de página del programa 2 luego de acceder a la página 3.

El problema ocurre si ahora uno de los programas quiere acceder a otra página, por ejemplo el programa 1 a la página 4. En este momento, la memoria física está ocupada completamente, al momento de ir a asociar un marco a la página 4 del programa 1, el sistema operativo se encuentra con que no hay marcos físicos disponibles. Esta situación se conoce como una **falta de página** o **page fault**:

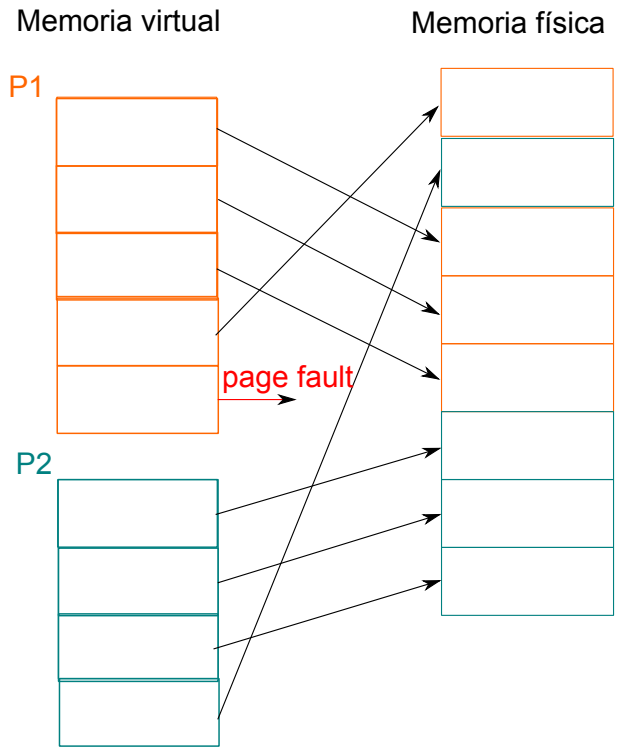


Figura 9: Programa 1 necesita espacio para su página 4, pero la memoria física está completa.

Para solucionar esta situación, se utiliza el disco duro como almacenamiento de respaldo para los marcos de memoria. Para lograr esto se reserva un espacio especial en el disco, denominado **swap file**, el cual será utilizado para respaldar marcos. De esta forma cuando un programa requiere un marco, pero la memoria está completamente ocupada, se copiará un marco de memoria al disco para dejar espacio para la nueva solicitud. Este proceso de respaldo de memoria a disco se conoce como **swap out**. Para determinar que marco reemplazar, se utiliza un algoritmo de reemplazo,

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2343 Arquitectura de Computadores

Pipeline: Paralelismo a nivel de instrucción

©Alejandro Echeverría

1. Motivación

Las principales mejoras en eficiencia de los computadores no ocurren sólo por mejoras en la tecnología de construcción de estos, sino también por el desarrollo de técnicas que permiten aprovechar procesamiento paralelo en el computador. Existen distintos niveles a los cuales se puede aprovechar el paralelismo, siendo el más básico el paralelismo a nivel de la instrucción.

2. Ciclo de la instrucción

Cada instrucción que se ejecuta un computador pasa por un ciclo, que va desde que es seleccionada desde memoria hasta que completa su objetivo. Este ciclo aunque es similar en todas las arquitecturas de computadores, presenta algunas variaciones, dependiendo de la complejidad de la microarquitectura y de decisiones de diseño. El computador básico Harvard (figura 1), por ejemplo, tendrá un cierto ciclo, pero este no será exactamente igual al de una arquitectura x86 o de un PIC16F87AA

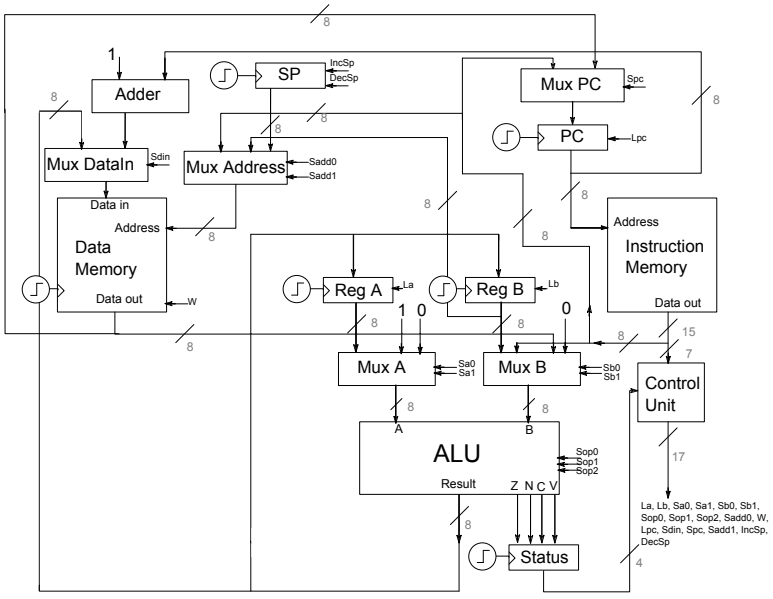


Figura 1: Diagrama computador básico.

El ciclo de la instrucción de computador básico Harvard, aunque es simple, presenta algunas complejidades que harán difíciles el análisis en detalle de éste posteriormente requerido. Para evitar estos problemas y trabajar con un ciclo más simple, se trabajará con una versión simplificada del computador básico, la cual se muestra en la figura 2. Las principales diferencias son las siguientes:

- Se elimina el soporte de stack, eliminando el Stack Pointer y las conexión del PC con memoria.
- Se simplifican los saltos condicionales, soportando ahora solamente el condition code Z, y la instrucción `JEQ`. Adicionalmente, esta instrucción realizará la comparación $(A - B)$ y el salto en un mismo ciclo, por lo que se elimina también el Status Register.
- Se elimina la conexión entre la salida de memoria y el MUX B. Ahora la salida de memoria no se puede ocupar como parámetro en la ALU, y solo puede ser usada para cargar los registros A y B, es decir sólo se soportan instrucciones de transferencia desde la memoria.
- La entrada de datos de la memoria, ahora sólo puede provenir de los registros A y B, y no de la ALU, es decir sólo se soportan instrucciones de transferencia hacia la memoria
- La dirección de la memoria ahora proviene de la ALU, lo que permite los mismos direccionamientos que antes (directo, indirecto por registro B), y agrega otros tipos de direccionamiento (indirecto por registro A, indirecto con índice, etc.)
- La unidad de salto, parte de la unidad de control encargada de determinar si corresponde un salto condicional, se separó de la unidad de control.

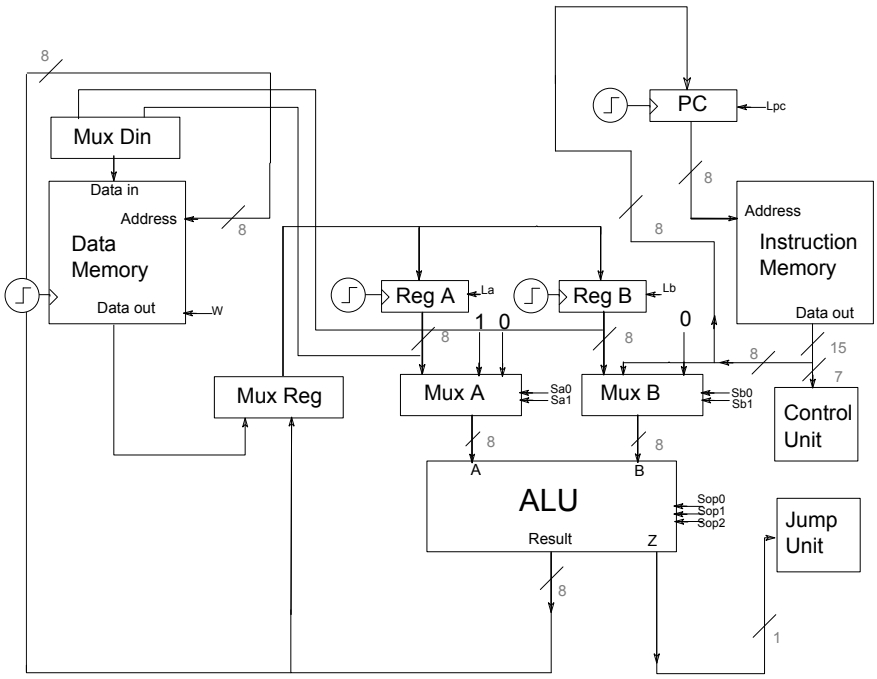


Figura 2: Diagrama computador básico modificado.

La idea detrás de estas modificaciones es por un lado simplificar el computador, pero también lograr que el ciclo de las instrucciones se más uniforme que en la versión original. Con estas modificaciones es posible reorganizar las partes del computador, para mostrar con más claridad las distintas partes del ciclo, lo que se observa en la figura 3.

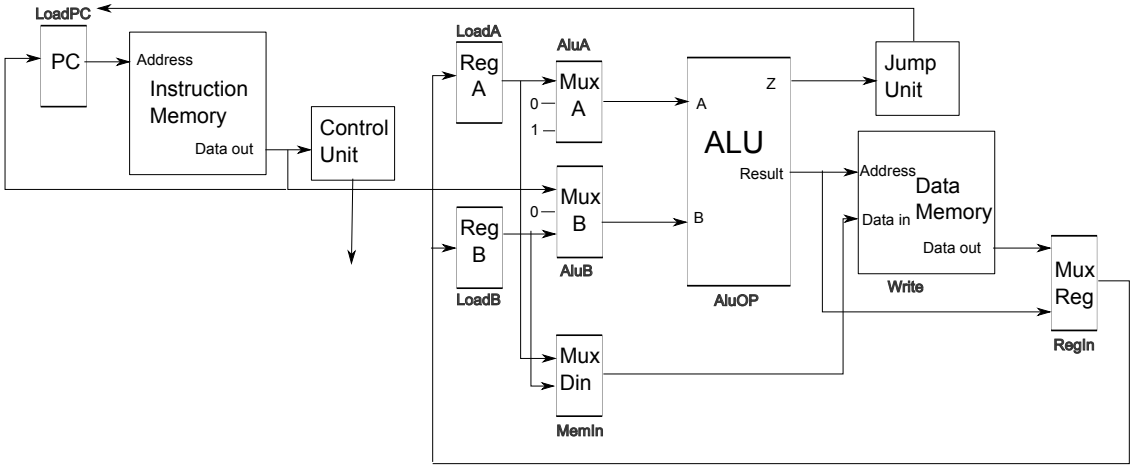


Figura 3: Diagrama computador básico modificado reordenado.

El ciclo de la instrucción para este computador, contará de 5 etapas: **instruction fetch**, **instruction decode**, **execute**, **memory** y **writeback**.

Instruction fetch (IF)

Corresponde a la primera etapa del ciclo, en la cual se va a buscar a la memoria de instrucciones la siguiente instrucción, apuntada por el valor actual del Program Counter. La salida de esta etapa es la instrucción obtenida desde memoria, la cual se separará en un opcode y en el parámetro.

Instruction decode (ID)

La segunda etapa del ciclo corresponde al instruction decode, es decir la decodificación de la instrucción y la generación de las señales de control. La unidad de control se encargará de transformar el opcode de la instrucción obtenida en las señales de control específicas que le indicarán al procesador que tarea ejecutar.

Execute (EX)

La tercera etapa corresponde a la ejecución, la cual es realizada mediante la ALU. La ALU, como única unidad de ejecución en este computador, se encargará de realizar la operación seleccionada en la etapa previa, con los parámetros seleccionados también por las señales de control, obteniendo un resultado. Adicionalmente, la ALU también generará el condition code Z, dependiendo del resultado de la operación obtenida.

Memory (MEM)

La cuarta etapa corresponde a la lectura o escritura en memoria de datos. Esta etapa solo estará presente si a la instrucción ejecutada le correspondía transferencia hacia o desde memoria (por ejemplo la instrucción `MOV (var1), A`). En esta etapa se podrá haber escrito en memoria, u obtenido un dato de está para luego ser almacenado en un registro.

Adicionalmente al manejo de memoria, en esta etapa también se incluirá la definición si corresponde saltar o no, por parte de la unidad de salto. Aunque este proceso no tiene relación con la

memoria de datos, se incluye en esta etapa debido a que solo luego de ejecutar la operación en la ALU se podrá tener la información necesaria para esta decisión.

Writeback (WB)

La última etapa del ciclo también estará presente sólo en algunas instrucciones. En esta etapa de writeback, corresponderá escribir en los registros ya sea un resultado de la ALU o un dato obtenido desde memoria (por ejemplo las instrucciones `ADD A,B` y `MOV A, (var1)`

El resumen de las 5 etapas, indicando las unidades funcionales asociadas a cada una, se observa en la figura 4.

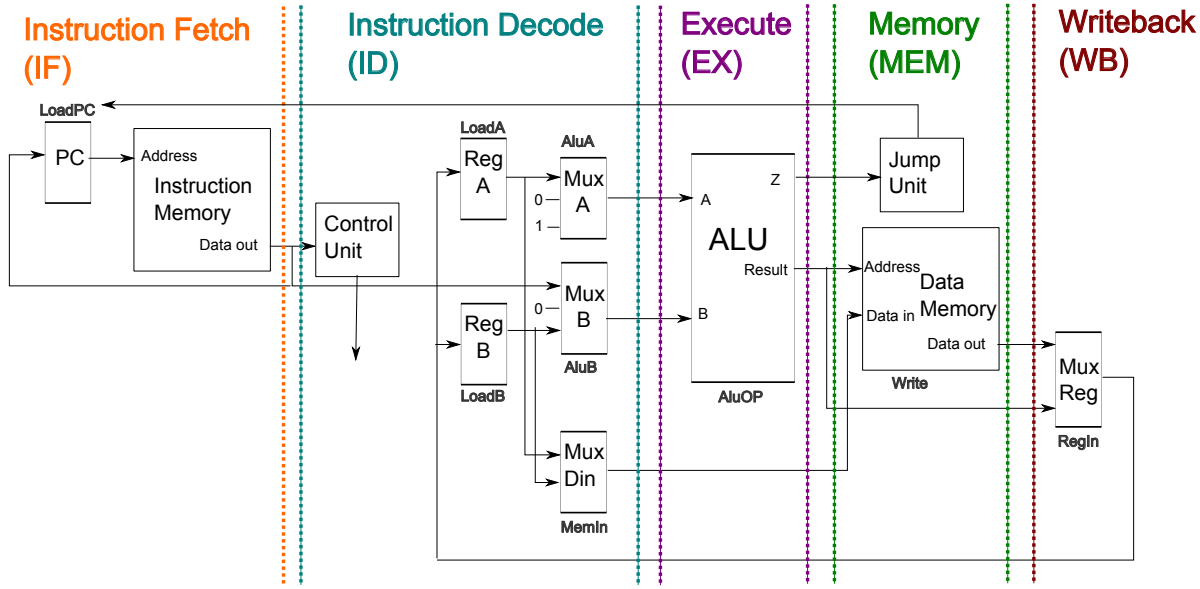


Figura 4: Ciclo de la instrucción.

A continuación se presentan una serie de ejemplos con los ciclos de distintas instrucciones. En cada caso se muestra el tiempo que tomaría hipotéticamente en completarse cada fase, y el tiempo total que tomaría la instrucción:

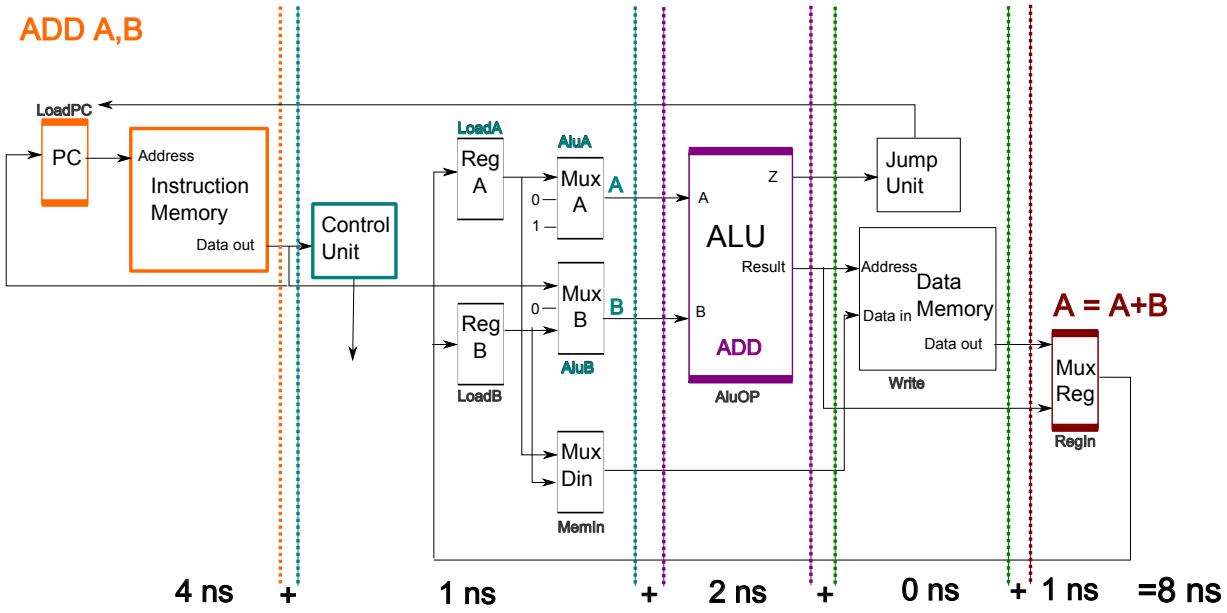


Figura 5: Ciclo de instrucción ADD A,B

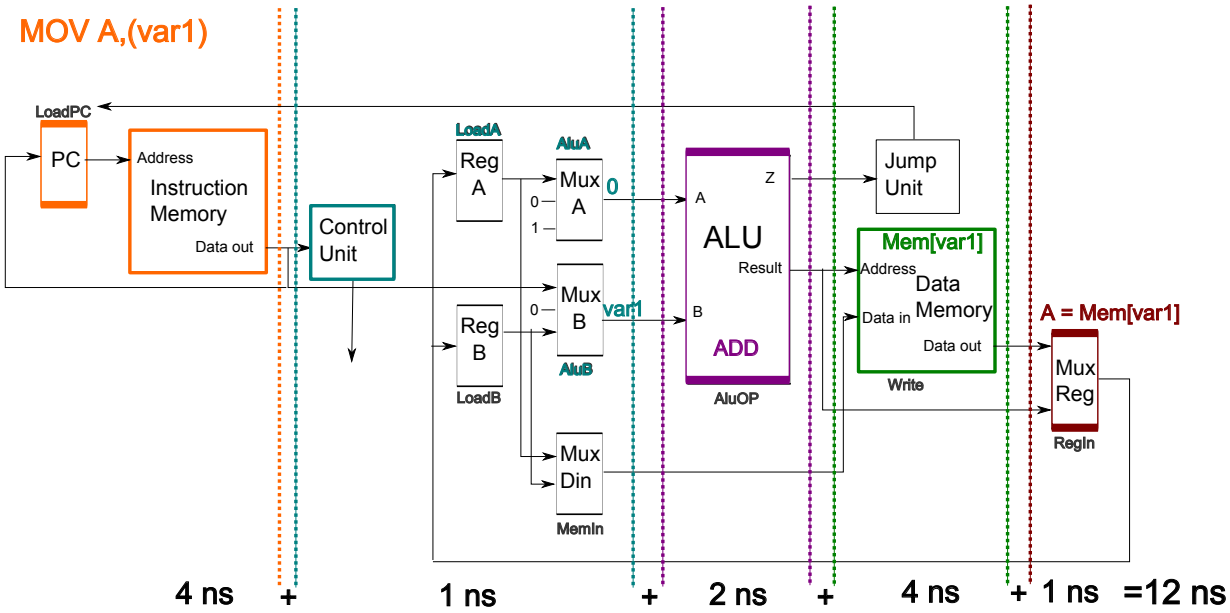


Figura 6: Ciclo de instrucción MOV A, (var1)

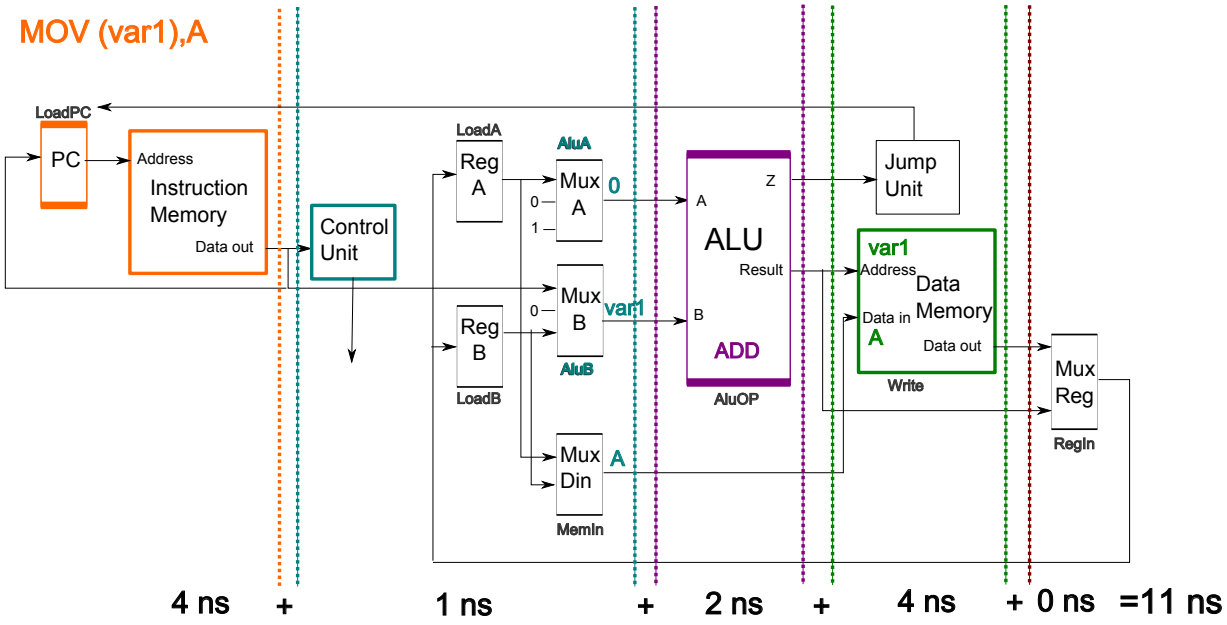


Figura 7: Ciclo de instrucción MOV (var1), A

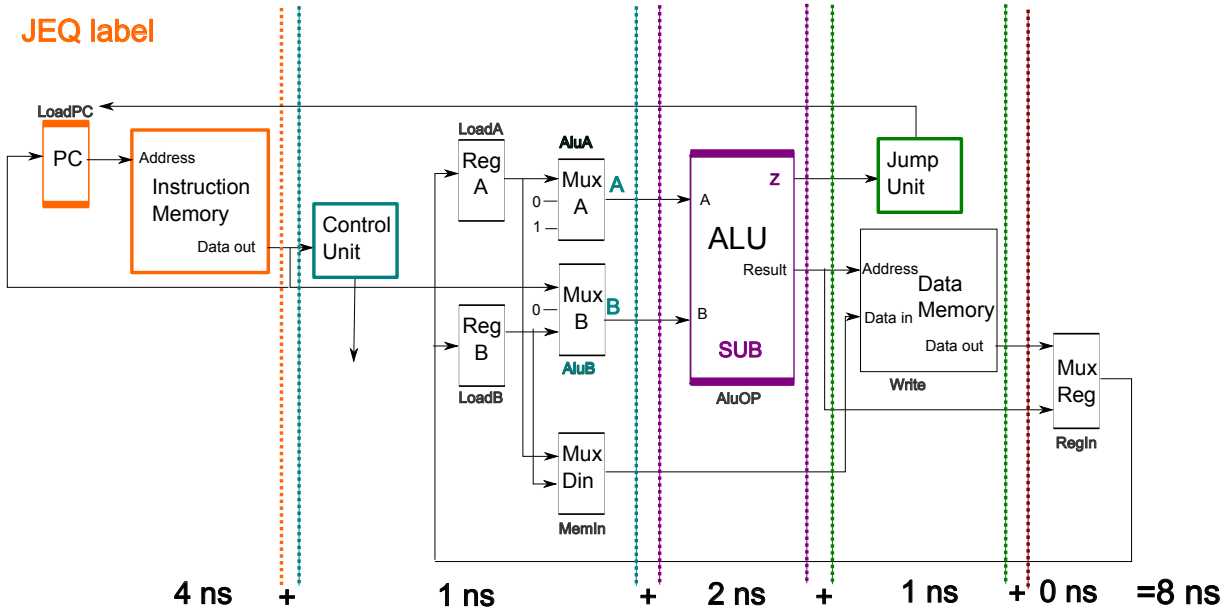


Figura 8: Ciclo de instrucción JEQ label

Como se observa en los ejemplos anteriores, cada una de las etapas del ciclo de la instrucción tendrá una duración determinada, dependiendo de que instrucción se ejecute. De esta forma el tiempo total de ejecución de una instrucción variará de acuerdo a que etapas estén involucrados. En un computador de un ciclo por instrucción, como el computador básico, se debe cumplir que el tiempo que dura el ciclo (indicado por la frecuencia del clock) no puede ser menor que el tiempo que toma la instrucción más lenta (en este ejemplo `MOV A, (var1)`). Debido a esto en un computador de una instrucción por ciclo, la frecuencia del clock queda limitada por el tiempo que se demora en procesar la instrucción más lenta.

3. Instruction pipeline

3.1. Fundamentos

El ciclo de una instrucción se caracteriza por corresponder a una secuencia de pasos que debe recorrer la instrucción, cada uno de los cuales estará a cargo de una unidad funcional distinta. Este tipo de procesos secuenciales con etapas independientes se conoce como **pipeline**. Existen diversos procesos que pueden ser modelados como un pipeline, siendo los más comunes las líneas de ensamblaje industriales, en las cuales distintas tareas son ejecutadas sobre un producto (e.g. automóvil) y luego de ejecutar todas las tareas se obtiene un producto final terminado.

En el caso del procesamiento de la instrucción, esta secuencia de etapas se conoce como **instruction pipeline**, que contiene las distintas partes del ciclo de la instrucción antes descrito. A diferencia de las líneas de ensamblaje, en el instruction pipeline no se obtiene un producto final, sino que el resultado corresponde a lo que haya ocurrido a lo largo del proceso.

La gran ventaja de un pipeline de ensamblaje está en que es posible paralelizar la producción de un producto (e.g auto), dado que al mismo tiempo que se le esté aplicando un proceso a un elemento (e.g agregar las puertas), es posible estar ejecutando otra tarea sobre otro elemento (e.g. construir el chasis). De esta forma, al dividir la producción completa en tareas menores se logra reducir el tiempo, ya que se aprovecha que cada parte del proceso es independiente, y que por tanto queda «libre» una vez que un determinado elemento a pasado por esta.

En el instruction pipeline ocurre lo mismo que en el pipeline de ensamblaje: la idea es poder reutilizar cada una de las subetapas para ir procesando otras instrucciones de manera simultánea. Supongamos el siguiente ejemplo: se quieren ejecutar 3 instrucciones seguidas en el computador básico modificado. Como señalamos anteriormente, dado que la instrucción más lenta toma 12 nanosegundos (ns), el ciclo del clock debe tardar ese tiempo como mínimo siempre. Considerando esto, las tres instrucciones tomarían 36 ns (figura 9).

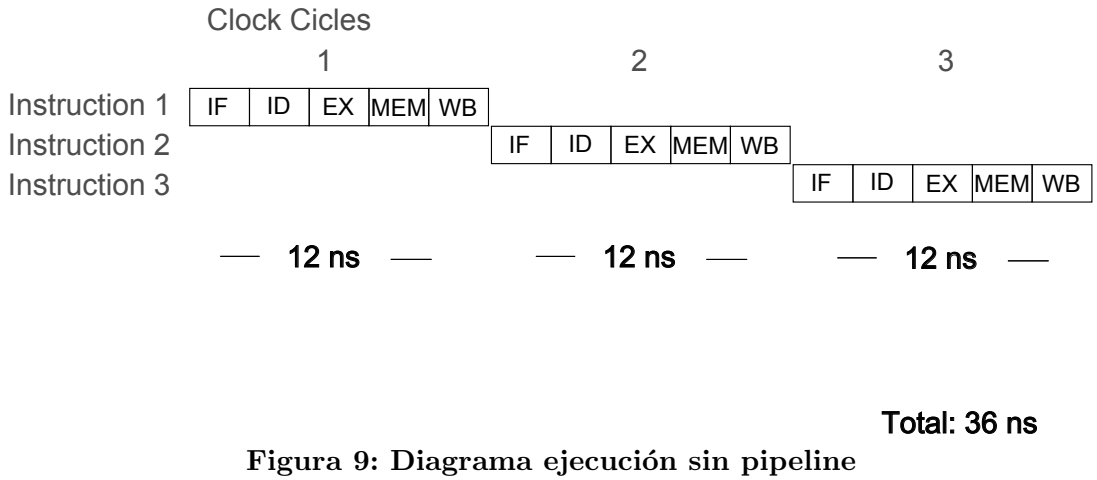


Figura 9: Diagrama ejecución sin pipeline

Para poder aprovechar el instruction pipeline, hay que considerar que cada instrucción se ejecuta en una secuencia de etapas, cada una de las cuales ocupa una unidad funcional distinta (al igual que en la línea de ensamblaje) y por tanto, una vez que se desocupa una etapa de procesar una instrucción puede inmediatamente comenzar a procesar la siguiente. Para lograr esto es necesario que en cada ciclo del clock no se procese toda una instrucción, sino sólo una etapa del ciclo. Con esto podemos tener ciclos de clocks más rápidos. El tiempo del ciclo estará limitado ahora no por

la instrucción mas lenta, sino por la etapa más lenta, lo que continuando con el ejemplo anterior serían 4 ns para los etapas de acceso a memoria.

Con esta modificación, el procesamiento de las 3 instrucciones quedaría de la siguiente manera:

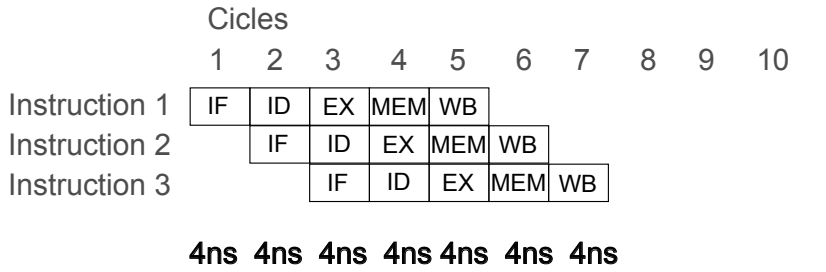


Figura 10: Diagrama ejecución con pipeline

Se observa una reducción de tiempo de los 36ns originales a 28ns. Esta reducción puede parecer menor, pero hay que considerar que un programa contiene millones de instrucciones. Si se considera ahora un ejemplo con 1 millón de instrucciones, el tiempo que tomaría procesarlas sin pipeline sería: 12 millones de nanosegundos (12×1 millon). En el caso del procesador con pipeline, el cálculo es más complicado. Hay que observar que pasa con más instrucciones en el diagrama:

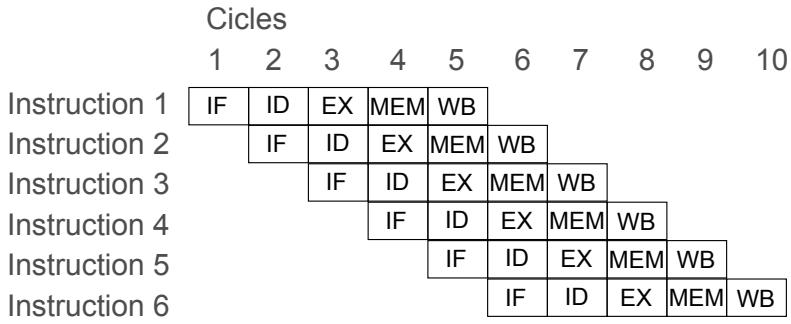


Figura 11: Diagrama ejecución con pipeline y 6 instrucciones

Se puede observar que a partir de la 5ta instrucción, el pipeline está «lleno» es decir, todas las etapas están procesando algo. Con esto se logra que a partir de esta instrucción se logra estar procesando **una instrucción por ciclo**, pero un ciclo mucho más rápido que en el caso anterior 4ns. Sólo en las 4 primeras instrucciones el pipeline está funcionando parcialmente, por lo que la regla para determinar cuanto se demoraría para este pipeline de 5 etapas es $Tiempociclo \times N°instrucciones + Tiempociclo \times 4$. En este ejemplo serían entonces 4 millones 16 nanosegundos, es decir se reduce prácticamente un tercio del tiempo original.

La gran paradoja del instruction pipeline, es que el tiempo para procesar cada instrucción no disminuye, sino que aumenta: en este caso el tiempo de procesar cada instrucción es de 20 ns, mayor que los 12 ns anteriores. Esto se debe a que la etapa más lenta indicará la velocidad del clock y por tanto ese tiempo se multiplicará por todas las etapas. Lo que mejora considerablemente con un sistema que ocupa pipeline es el **throughput** de las instrucciones de un programa, es decir, cuantas

instrucciones son procesadas en un cierto tiempo. En el ejemplo anterior, en los mismos 12 millones de nanosegundos que toma el computador sin pipeline en ejecutar 1 millón de instrucciones, el computador con pipeline ejecutará 3 millones de instrucciones.

La siguiente es una lista de términos relevantes en el contexto de pipeline (figura 12):

- La **profundidad (depth)** del pipeline corresponde al número de etapas de este. En nuestro ejemplo son 5 etapas.
- El período al comenzar el procesamiento de una secuencia de instrucciones, en el cual no todas las etapas del pipeline están ocupadas, se conoce como **llenado o filling**.
- El período en el cual todas las etapas están ocupadas se dice que el pipeline está **lleno o full**.
- El período al terminar el procesamiento de una secuencia de instrucciones, en el cual no todas las etapas del pipeline están ocupadas, se conoce como **vaciado o emptying**.

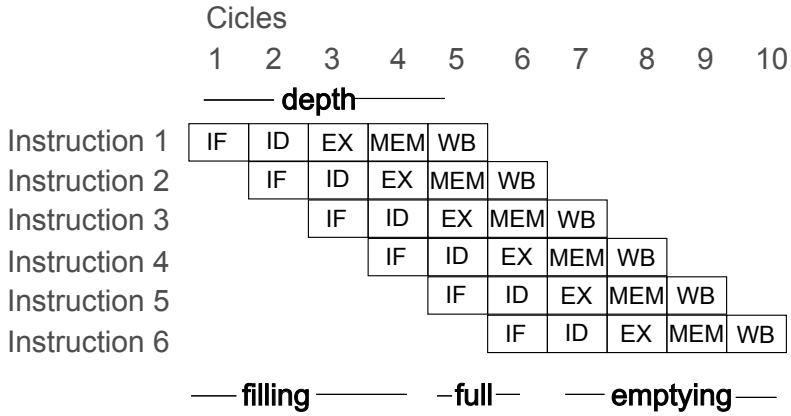


Figura 12: Terminología de pipeline

3.2. Soporte de Hardware

Para implementar un procesador con pipeline es necesario ir almacenando los resultados de las distintas etapas y propagándolos, de manera que en el siguiente ciclo se puedan utilizar para la siguiente etapa. Para lograr esto, se utilizan una serie de registros que se ubican entre las distintas etapas. Los conjuntos de registros se denominan de acuerdo al par de etapas que comuniquen: registros IF/ID, ID/EX, EX/MEM y MEM/WB (figura 13). Para la etapa de write back no se requieren registros adicionales, ya que el resultado de esta etapa se almacena en los registros propios de la CPU.

En cada uno de estos registros se almacenará los distintos resultados de cada etapa, para el computador básico estos serían:

- IF/ID: almacena la instrucción obtenida desde la memoria de instrucciones, incluyendo el opcode y el parámetro
- ID/EX: almacena los dos parámetros de la ALU, el valor del registro seleccionado para ser copiado a memoria (si corresponde) y el parámetro de la instrucción. Este último se debe ir propagando para el caso de las instrucciones de salto, ya que es necesario que sea cuando se revise si va a haber salto (en la etapa MEM) que se cargue el PC con la dirección de memoria a saltar.

- EX/MEM: resultado de la ALU, el valor del registro seleccionado para escribir en memoria y el parámetro de la instrucción ambos que se propagaron de la etapa anterior.
- MEM/WB: resultado de la ALU y el valor leído de memoria, los cuales serán utilizados en WB para escribir (si corresponde) en los registros.
- El resultado de la etapa WB se almacena en los registros A o B, por lo que no se requieren registros especiales.

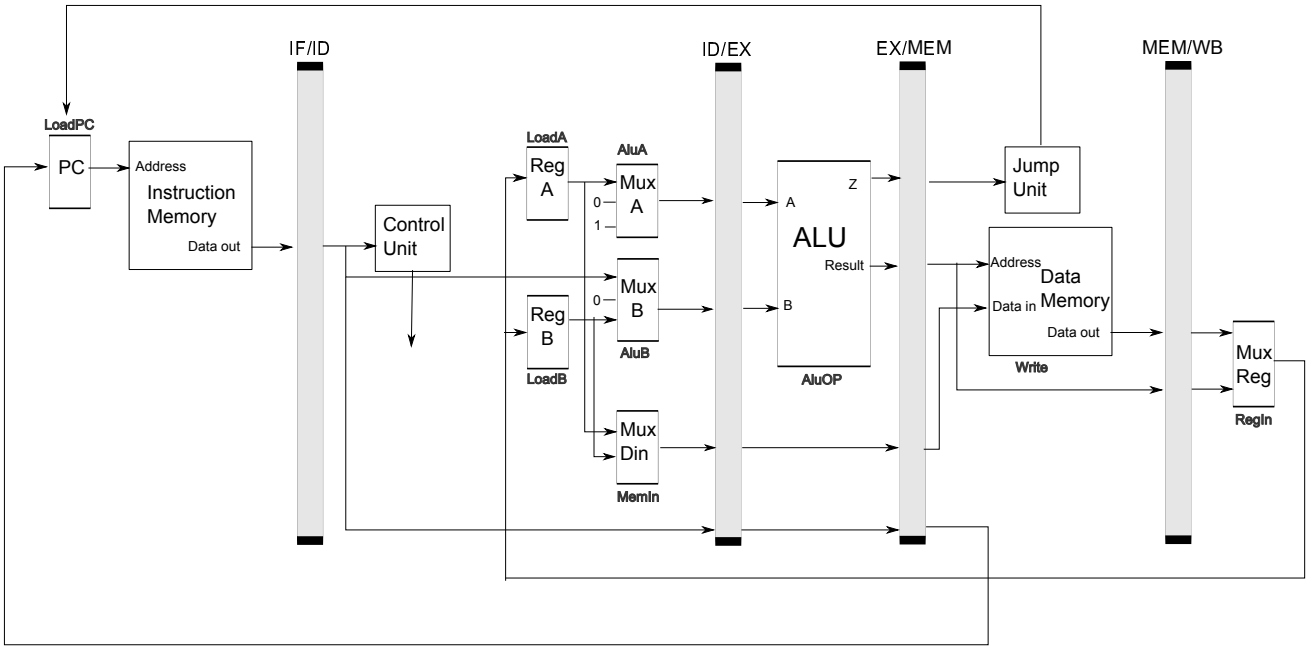


Figura 13: Soporte de hardware para propagación de datos

Además de almacenar datos, los registros intermedios deben almacenar las señales de control, generadas en la etapa ID, que son necesarias en etapas siguientes (figura 14):

- ID/EX: almacena todas las señales de control necesarias para todas las siguientes etapas: operación de la ALU (AluOP), señal de escritura en Memoria (W), señales de carga en los registros (LoadA y LoadB), señal de selección de que se va a escibir en los registros (RegIn) y señal que indica si la instrucción actual es de salto (Jmp).
- EX/MEM: en la etapa EX se utilizó la señal AluOp, por lo que no es necesario propagarla, se propagan todas las demás.
- MEM/WB: En la etapa MEM se utilizan las señales Jmp y W, solo se requiere propagar la señal RegIn y las señales LoadA y LoadB.

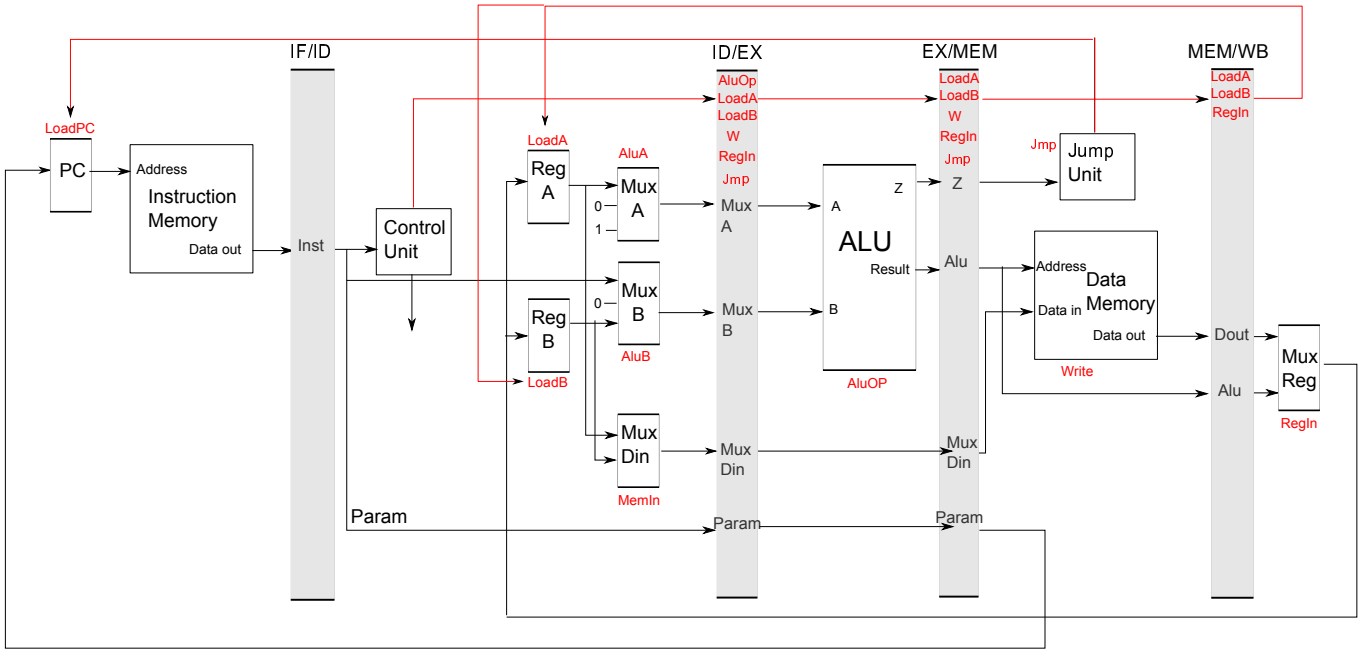


Figura 14: Soporte de hardware para propagación de control

3.3. Hazards

Al incorporar un pipeline en el computador, se generan una serie de problemas que son conocidos como **hazards**. Estos problemas ocurren por distintas razones, pero en general se deben al hecho de empezar a procesar una instrucción antes de que se haya procesado la anterior. Existen tres tipos de hazards: hazards de datos (data hazards), hazards de control (control hazards) y hazards estructurales (structural hazards).

3.3.1. Hazards de datos

Para entender los data hazards, revisemos el siguiente ejemplo de un programa en el que se ejecutan la instrucción `ADD A,B` y luego la instrucción `AND B,A`. Se puede observar que la segunda instrucción ocupa como parámetro el registro A. La primera instrucción va a modificar el valor de A, por lo que es necesario que haya terminado de ejecutarse para que las siguientes instrucciones tengan el valor correcto. El diagrama de pipeline de estas dos instrucciones es el siguiente:

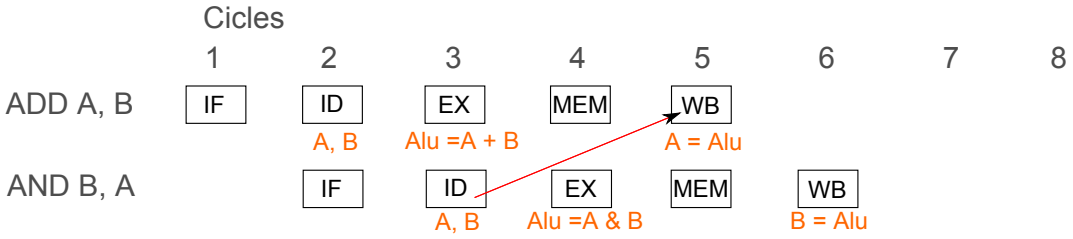


Figura 15: Ejemplo de data hazard, las flechas rojas indican las dependencias de datos

Como se observa en el diagrama, la primera instrucción `ADD A,B` va a actualizar el valor del registro A en su etapa de **WB**, es decir en el ciclo 5. El problema es que la siguiente instrucción

`AND B,A`, necesita obtener el valor de A en su etapa **ID**, la que ocurre en el ciclo 3. La causa de este problema es que la secuencia de instrucciones del ejemplo presenta **dependencia entre los datos** que utiliza. Si la instrucción 2 no ocuparán el registro A, no habría problema, pero como depende de tener el valor que se genera en la primera instrucción, ocurre un error.

Si observamos nuevamente el diagrama de la figura 16 observamos que aunque efectivamente es en la etapa WB de la instrucción 1 cuando el registro A se actualiza, el nuevo valor de A se generó antes, en la etapa EX, la cual ocurre en el ciclo 3 de ejecución. Para la instrucción 2 además, tenemos que aunque el valor de A se capta en la etapa ID, este se necesitará en la etapa EX, es decir en el ciclo 4. De esta forma, cuando el nuevo valor de A se necesita (ciclo 4), este ya se generó (ciclo 3), y está almacenado en los registros de la etapa ID/EX. Lo que necesitamos entonces es una forma de propagar el valor que se generó en la ALU al procesar la primera instrucción luego de la ejecución para que pueda ser ocupado como parámetro en la ejecución de las siguientes etapas EX, de la segunda instrucción (figura 17). Este mecanismo de propagación que se agrega a un pipeline se conoce como **forwarding**.

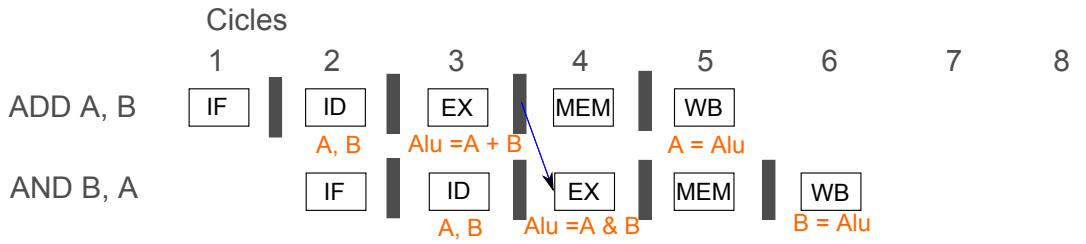


Figura 16: Ejemplo de data hazard, las flechas azules indican cuando hay que hacer forwarding

La idea de forwarding es, al estar ejecutando una instrucción, detectar si va a ocurrir un data hazard, y en ese caso realizar la propagación correspondiente. Esta detección debe realizarse en la etapa EX de la instrucción actual, antes de que se ejecute la operación de la ALU, que es lo primero que va a necesitar algún parámetro.

Existen varias situaciones en las cuales puede ocurrir esta detección, dependiendo de que instrucción previa genera dependencia, y en que etapa se genera:

1. Instrucción previa escribe en registro; instrucción actual utiliza registro en la ALU:

   La primera situación que requiere forwarding es en el caso de que una instrucción vaya a realizar una operación con la ALU, y requiera como parámetro un registro que está siendo modificado por la instrucción anterior. En el ejemplo anterior esto es lo que ocurre con la segunda instrucción, `AND B,A` .

   Para detectar este tipo de hazards hay que obtener la siguiente información:

   a) La instrucción previa tiene que estar escribiendo en registro, es decir las señales LoadA o LoadB deben haberse activadas en el decode de dicha instrucción. Como la instrucción previa esta en la etapa MEM en este momento, las señales de control correspondientes estarán en el registro EX/MEM. La nomenclatura para referirse a estas señales es EX/MEM.LoadA y EX/MEM.Loadb.

b) La instrucción actual ocupa como uno de sus parámetros el mismo registro que se escribió previamente. Para saber esto necesitamos saber que valores se eligieron como parámetros de la ALU en los MuxA y MuxB, lo cual estará almacenado en las señales de control AluA y AluB asociados a la instrucción actual, las cuales ahora deben ser propagadas entre las etapas ID y EX. Como la instrucción actual está en la etapa EX, las señales de control que nos interesan están en los registros ID/EX, y por tanto la nomenclatura para estas señales es ID/EX.AluA y ID/EX.AluB.

En base a esta información se puede decidir si corresponde hacer forwarding a la entrada A o B de la ALU. Lo que se tiene que propagar es el resultado de la ALU que se tenga almacenado en la etapa MEM. Para saber si ese resultado debe ingresar a la entrada A o B, se revisa lo siguiente:

- Si EX/MEM.LoadA == 1 y ID/EX.AluA == A (se eligió el registro A como parámetro), corresponde hacer forwarding del resultado de la ALU de la etapa MEM a la entrada A de la ALU.
- Si EX/MEM.LoadB == 1 y ID/EX.AluB == B (se eligió el registro B como parámetro), corresponde hacer forwarding del resultado de la ALU de la etapa MEM a la entrada B de la ALU.

2. Instrucción previa a la anterior escribe en registro; instrucción actual utiliza registro en la ALU:

La segunda situación en que es necesario propagar ocurre cuando una instrucción que va a operar con la ALU requiere como parámetro un registro que está siendo modificado por la instrucción de dos ciclos atrás:

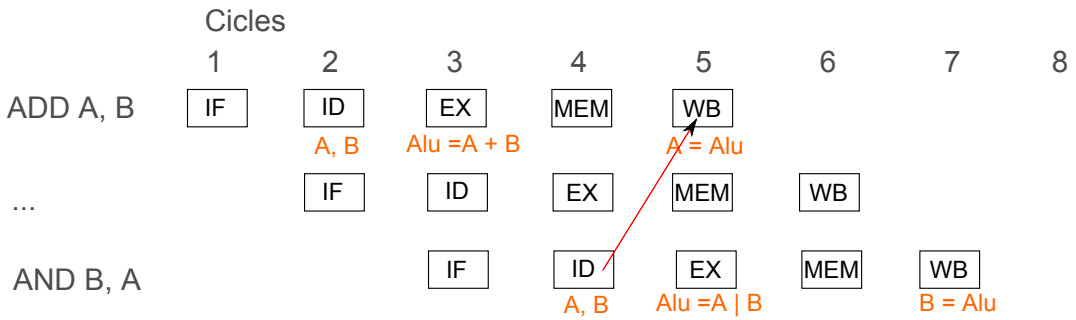


Figura 17: Ejemplo de data hazard en que una instrucción afecta a la subsiguiente.

Aunque este caso es similar al anterior, y también es posible detectarlo ocupando el mismo tipo de condiciones antes descritas, existen un par de diferencias. En primer lugar, cuando la instrucción que va a sufrir un data hazard esté en la etapa EX (`AND B,A` en el ejemplo), la instrucción que generará el hazard (`AND B,A`) estará en la etapa WB, y por tanto las señales LoadA y LoadB que interesa revisar estarán en los registros MEM/WB (MEM/WB.LoadA y MEM/WB.LoadB). Lo segundo es qué ocurre si la instrucción intermedia genera también dependencia de datos con la tercera instrucción, como en el ejemplo de la figura 19. En este

caso, no es necesario hacer el forwarding desde la instrucción 1 a la 3, dado que para la instrucción 3 lo que importa es el último valor de A, que será obtenido en la instrucción 2.

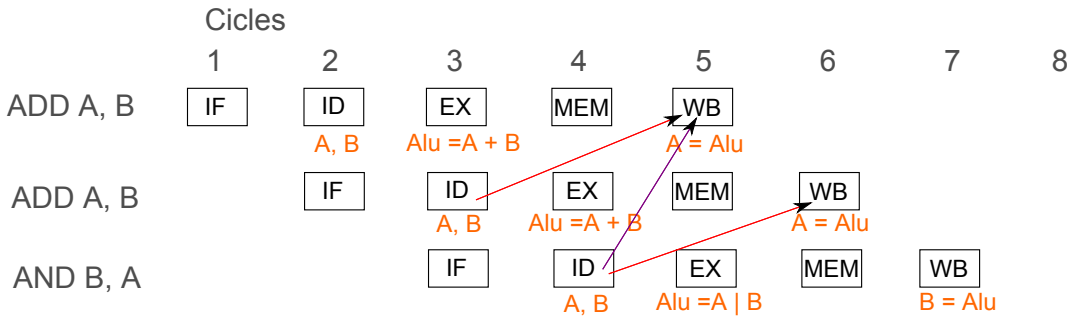


Figura 18: Ejemplo de data hazard: para la tercera instrucción sólo importa la última modificación sobre el registro A, por lo que la dependencia entre la 1era y 3era instrucción (flecha morada) no necesita forwarding.

Considerando esto, la información para detectar este tipo de hazards es:

a) La instrucción anterior a la previa tiene que estar escribiendo en registro, es decir las señales LoadA o LoadB deben haberse activadas en el decode de dicha instrucción. Como la instrucción anterior a la previa esta en la etapa WB en este momento, las señales de control correspondientes estarán en el registro MEM/WB: MEM/WB.LoadA y MEM/WB.Loadb.

b) La instrucción actual ocupa como uno de sus parámetros el mismo registro que se escribió previamente. Para saber esto necesitamos saber que valores se eligieron como parámetros de la ALU en los MuxA y MuxB, lo cual estará almacenado en las señales de control AluA y AluB asociados a la instrucción actual: ID/EX.AluA y ID/EX.AluB.

c) Si la instrucción previa también modificó los registros A o B, esta información estará en los registros EX/MEM: EX/MEM.LoadA y EX/MEM.LoadB.

En base a esta información se puede decidir si corresponde hacer forwarding a la entrada A o B de la ALU. Lo que se tiene que propagar es el resultado de la ALU que se tenga almacenado en la etapa WB. Para saber si ese resultado debe ingresar a la entrada A o B, se revisa lo siguiente:

- Si MEM/WB.LoadA == 1, ID/EX.AluA == A y EX/MEM.LoadA == 0 (es decir, no hubo escritura en la instrucción anterior), corresponde hacer forwarding del resultado de la ALU de la etapa WB a la entrada A de la ALU.
- Si MEM/WB.LoadB == 1, ID/EX.AluB == B y EX/MEM.LoadB == 0 (es decir, no hubo escritura en la instrucción anterior), corresponde hacer forwarding del resultado de la ALU de la etapa WB a la entrada B de la ALU.

3. Instrucción previa escribe en registro; instrucción actual escribe en memoria valor del registro modificado:

Los últimos casos en que es necesario propagar son cuando la instrucción actual está escribiendo en memoria el valor de un registro que está siendo modificado por la instrucción previa:

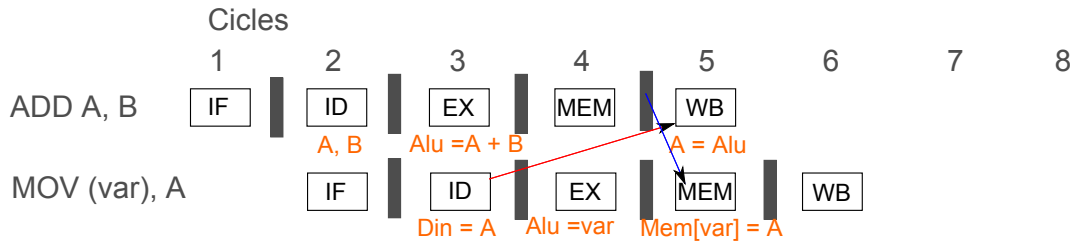


Figura 19: Ejemplo de data hazard: la segunda instrucción necesita el nuevo valor de A para escribir en memoria. Se debe realizar forwarding de la salida de la etapa WB previa a la etapa MEM actual.

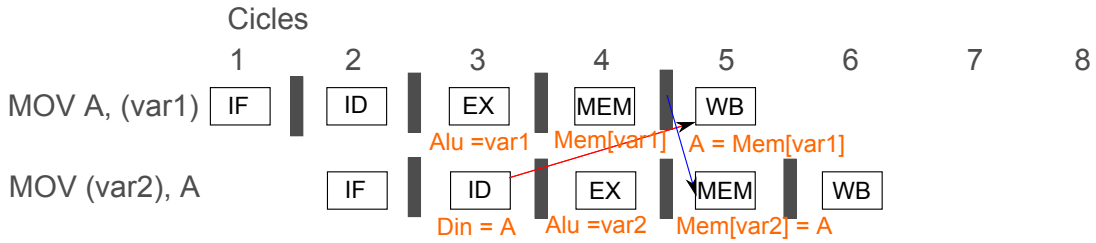


Figura 20: Ejemplo de data hazard: la segunda instrucción necesita el nuevo valor de A para escribir en memoria. Se debe realizar forwarding de la salida de la etapa WB previa a la etapa MEM actual.

En este caso, la dependencia de datos está entre las etapas ID de la instrucción actual, donde se elige el registro que será escrito en memoria, y la etapa WB de la instrucción anterior, donde se está modificando ese registro. A diferencia de los casos anteriores, la etapa a la cual es necesario propagarle datos es la etapa MEM. La información para detectar este tipo de hazards es:

a) La instrucción anterior tiene que estar escribiendo en registro, es decir las señales LoadA o LoadB deben haberse activadas en el decode de dicha instrucción. Como la instrucción anterior está en la etapa WB en este momento, las señales de control correspondientes estarán en el registro MEM/WB: MEM/WB.LoadA y MEM/WB.Loadb.

b) La instrucción actual está escribiendo en memoria (EX/MEM.W = 1) y ocupa como dato de entrada en memoria el mismo registro que fue modificado (EX/MEM.MuxDin = A o EX/MEM.MuxDin = B).

En base a esta información se puede decidir si corresponde hacer forwarding al valor de entrada de la memoria. Lo que se tiene que propagar en este caso es la salida del Mux RegIn de la etapa WB, ya que esta situación puede ocurrir tanto si la instrucción previa fue ocupando la ALU o leyendo de memoria. Las codiciones para revisar este caso son:

- Si MEM/WB.LoadA == 1, EX/MEM.MemDin == A, EX/MEM.W == 1 corresponde hacer forwarding del RegIn de la etapa WB a la entrada de memoria.

- Si MEM/WB.LoadB == 1, EX/MEM.MemDin == B, EX/MEM.W == 1 corresponde hacer forwarding del RegIn de la etapa WB a la entrada de memoria.

Para lograr detectar el forwarding e implementarlo cuando corresponda, se agrega una pieza de hardware denominada **forwarding unit**, la cual se encarga de revisar si ocurrió un data hazard y en caso de haber ocurrido uno, propaga los valores a las entradas de la ALU correspondiente, a través de dos multiplexores que se agregan a estas entradas (figura 21).

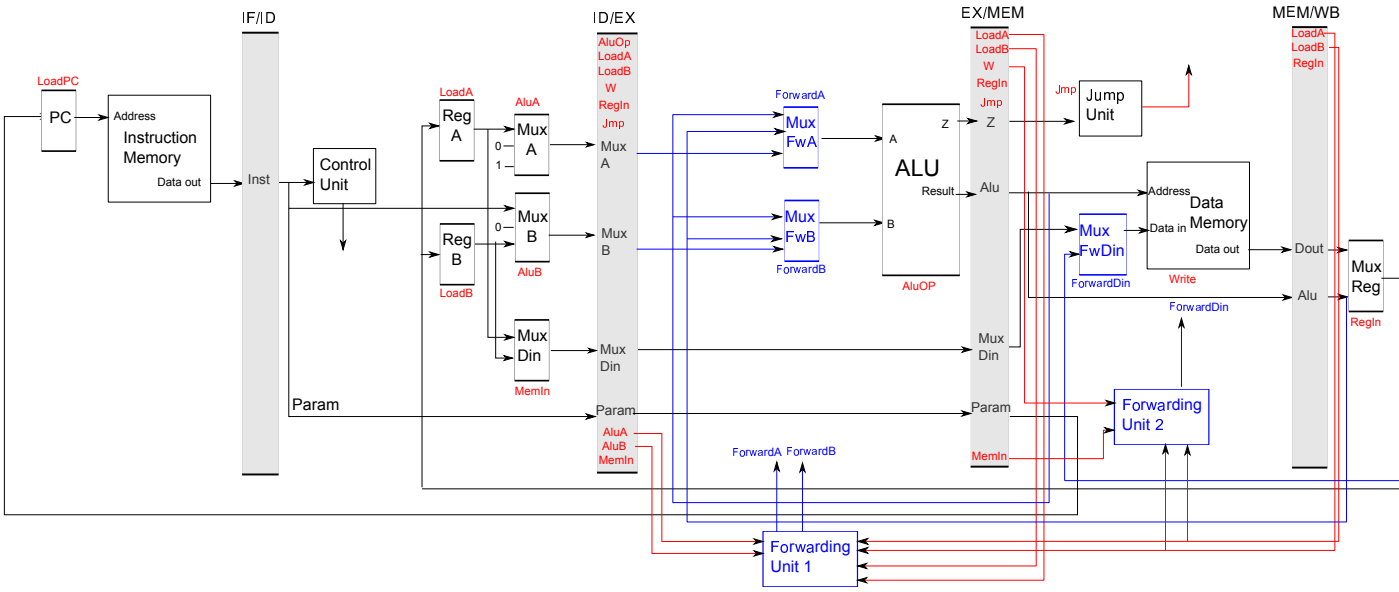


Figura 21: Soporte de hardware para forwarding.

Al agregar forwarding al procesador se solucionan los casos de data hazard antes descritos, pero no es suficiente para manejar todos los posibles casos. En particular, existe otro tipo de data hazard, que ocurre cuando un valor se lee de memoria en una instrucción, y luego se utiliza como parámetro en la ALU de la instrucción siguiente, como se observa en el ejemplo de la figura 22.

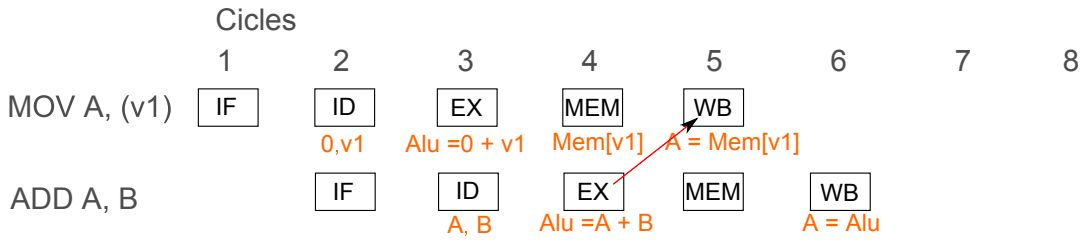


Figura 22: Ejemplo de data hazard, producido por una lectura de memoria.

El problema en este caso es que a diferencia de los hazards anteriores, acá el valor modificado no se obtiene en la etapa EX, sino en la etapa MEM. Si intentamos hacer forwarding ocurre lo que se observa en la figura 23: el valor que requerimos propagar solo se obtendrá al final del ciclo en que se ejecuta MEM, pero es requerido al principio de ese mismo ciclo por la etapa EX de la siguiente instrucción. Esto hace imposible lograr el forwarding ya que en el momento que tenemos el valor ya es muy tarde y se ejecutó la operación siguiente con el valor erróneo. En estos casos

se requiere agregar un mecanismo que permita que la siguiente instrucción «espere» mientras se obtiene el valor necesario, lo que se conoce como **stalling**.

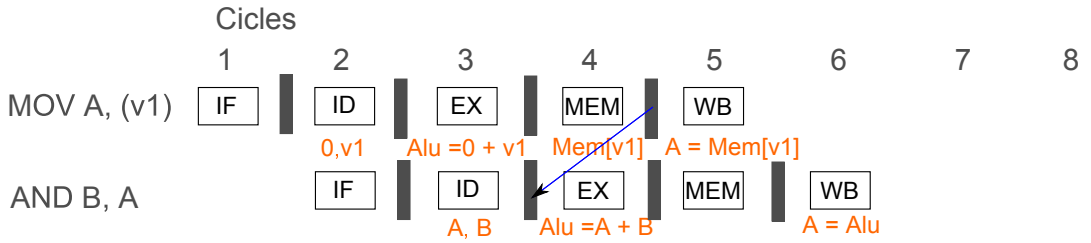


Figura 23: Ejemplo de data hazard, producido por una lectura de memoria.

En el mecanismo de stalling, la idea es hacer que la instrucción siguiente no entré en la etapa EX hasta que se haya obtenido el dato desde memoria. Para lograr esto hay dos mecanismos: uno implementado por hardware y otro por software. En el mecanismo por hardware, el procesador debe detectar esta situación y detener el avance del pipeline por un ciclo, insertando lo que se conoce como una «burbuja» en el pipeline (figura 24). Esta detención permite que se mantenga la consistencia de los datos, pero a costo de perder un ciclo de CPU.

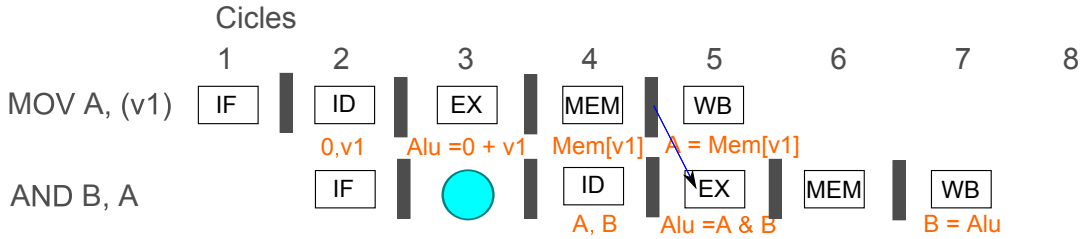


Figura 24: Stalling del pipeline para solucionar data hazard de lectura. Se agrega una «burbuja» al pipeline.

Las condiciones que se deben dar para que haya stalling para este computador, son las siguientes:

1. La instrucción anterior tiene que estar escribiendo en registro desde memoria, lo que se puede observar en la señal de control RegIn.
2. La instrucción actual ocupa como parámetro de la ALU el registro escrito.

En este caso, la idea es detectar lo antes posible el hazard para perder la menor cantidad de ciclos. Debido a esto, la detección de este tipo de hazards se realiza en la etapa ID, cuando son generadas las señales de control de la instrucción actual, la cual debemos detener en caso de que haya hazard. Los valores que se deben revisar para estos casos son:

- Si ID/EX.LoadA == 1 y ID/EX.RegIn == Dout (se va a leer de memoria), y la instrucción actual tienen AluA == A, corresponde hacer stalling para evitar inconsistencias con el registro A.
- Si ID/EX.LoadB == 1 y ID/EX.RegIn == Dout (se va a leer de memoria), y la instrucción actual tienen AluB == B, corresponde hacer stalling para evitar inconsistencias con el registro B.

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2343 Arquitectura de Computadores

Paralelismo avanzado

1. Motivación

El enfoque que tradicionalmente se utilizó para lograr que un computador realice más instrucciones por unidad tiempo, y que por tanto el computador mejore su rendimiento, fue ir logrando que el computador pudiese correr a clocks más rápidos. Ya sea mediante la disminución de tamaño de transistores u otras técnicas como pipeline, el aumentó velocidad del clock de la CPU logró por muchos años permitir las mejoras continuas de rendimiento en los computadores. Sin embargo, el aumento de velocidad trajo consigo un aumento de consumo de energía y disipación de temperatura, llegándose a niveles que hicieron imposible seguir aumentando más la velocidad.

Para lograr que los computadores sigan mejorando su rendimiento se comenzaron a implementar distintas técnicas de paralelismo en las arquitecturas de los computadores, las cuales permiten que un computador ejecute más instrucciones por unidad de tiempo, pero sin necesariamente aumentar la velocidad del clock.

2. Arquitecturas paralelas

Existen distintas formas en que se puede paralelizar el procesamiento en un computador, las que incluyen paralelismo dentro y fuera del procesador. Una forma de categorizar las arquitecturas paralelas es mediante la **Taxonomía de Flynn** (tabla 1) la cual define cuatro categorías de paralelismo según si se utiliza uno o múltiples streams de instrucciones (i.e. programas) y uno o múltiples streams de datos.

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Tabla 1: Taxonomía de Flynn.

De estas cuatro categorías, las tres más relevantes y que tienen aplicación práctica son: SISD, paralelismo dentro de un procesador sobre un programa, MIMD, paralelismo de múltiples procesadores sobre múltiples programas, y SIMD, paralelismo que puede ser en uno o múltiples procesadores, y que realiza un mismo procesamiento sobre un conjunto de datos. A continuación se analizaran distintos ejemplos de implementación de estos sistemas

2.1. Paralelismo SISD

El paralelismo SISD nace de una idea simple: que pasa si agregamos una unidad de ejecución secundaria al procesador, y así permitir que este ejecute más de una instrucción al mismo tiempo.

En la figura 1 se observa un diagrama simplificado del computador básico con pipeline, en la cual los registros han sido agrupados en un **register file** o conjunto de registros, y se renombran los registros A y B como R1 y R2. Supongamos ahora que se le agrega una FPU a este computador. Para poder realizar operaciones con esta unidad, es necesario agregar registros de punto flotante al computador, como se observa en la figura 2.

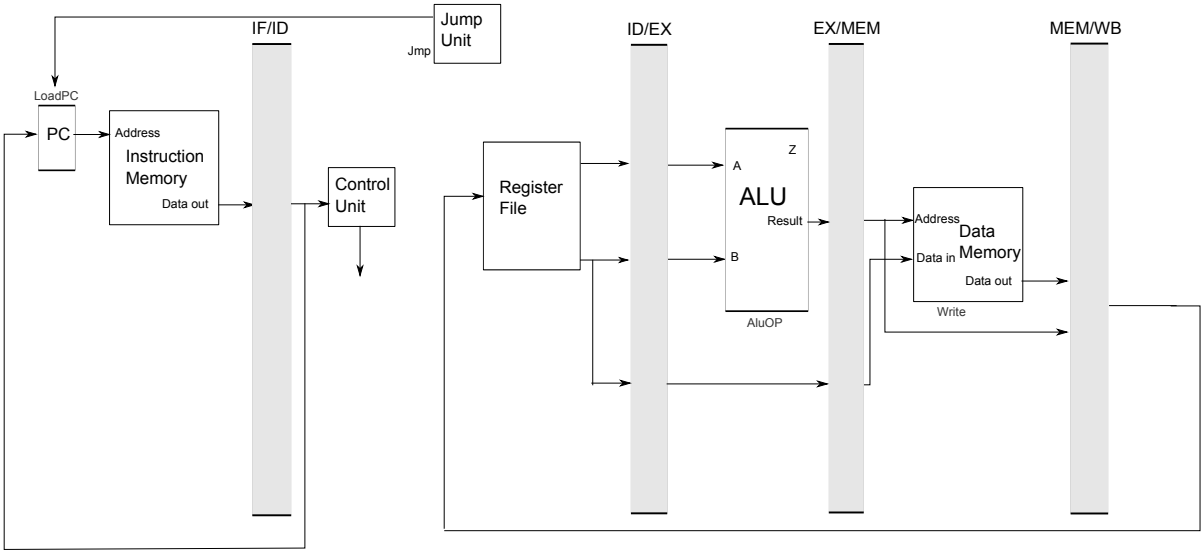


Figura 1: Diagrama simplificado del computador básico con pipeline, agrupando los registros en un register file.

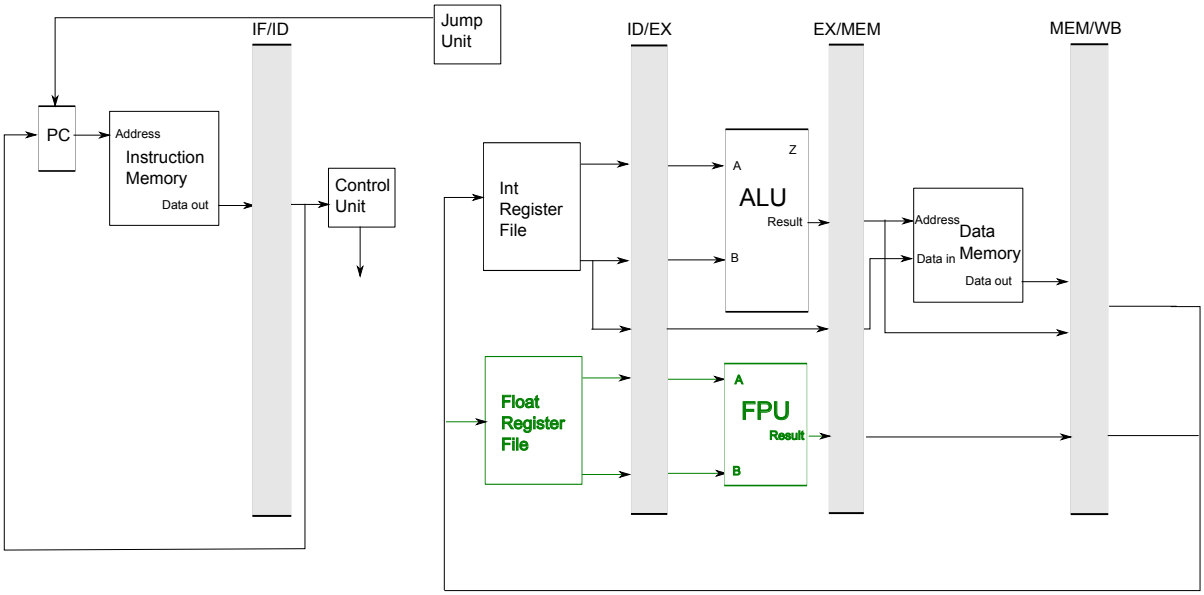


Figura 2: Al agregar una FPU y registros de punto flotantes agregados al computador básico, el flujo de la instrucción puede tomar dos caminos en la etapa de ejecución.

Se puede observar que ahora una instrucción puede tomar dos caminos, dependiendo de si se ejecuta una operación en la ALU o en la FPU. Para lograr ejecutar operaciones en la FPU es necesario agregar instrucciones especiales de punto flotante y permitir el acceso a los registros de punto flotante. Supongamos en este caso que se agrega la instrucción `FADD` para realizar la suma y los registros de punto flotante `F1` y `F2`. Como en general las operaciones aritméticas de punto flotante son más complejas que las con números enteros, la ejecución de una operación de la FPU tomará más tiempo que la ejecución de una operación de la ALU. Supongamos que la etapa de ejecución de la FPU se puede dividir en tres subetapas: `FEX1`, `FEX2` y `FEX3`. El diagrama del pipeline para la ejecución de las instrucciones `ADD R1, R2` seguida de `FADD, F1, F2` se observa en la figura 3.

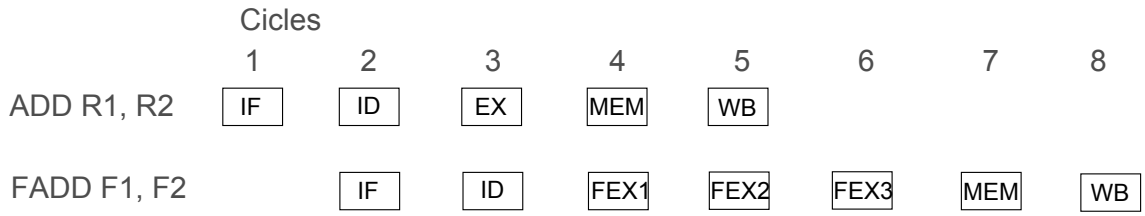


Figura 3: Las instrucciones `ADD R1, R2` y `FADD F1, F2` utilizan registros y unidades de ejecución distintas.

Lo interesante de este par de instrucciones es que están ocupando unidades de ejecución distintas (una la ALU y otra la FPU) y registros de operación y resultado distintas (R1 y R2 la primera, F1 y F2 la segunda). De esta forma las etapas de ejecución y write back de ambas instrucciones son completamente independientes. A partir de esto se intuye que sería posible paralelizar estas instrucciones si las etapas de fetch y decode fueran capaces de trabajar con más de una instrucción al mismo tiempo, como se observa en el diagrama del computador de la figura 4 y el diagrama de pipeline de la figura 5.

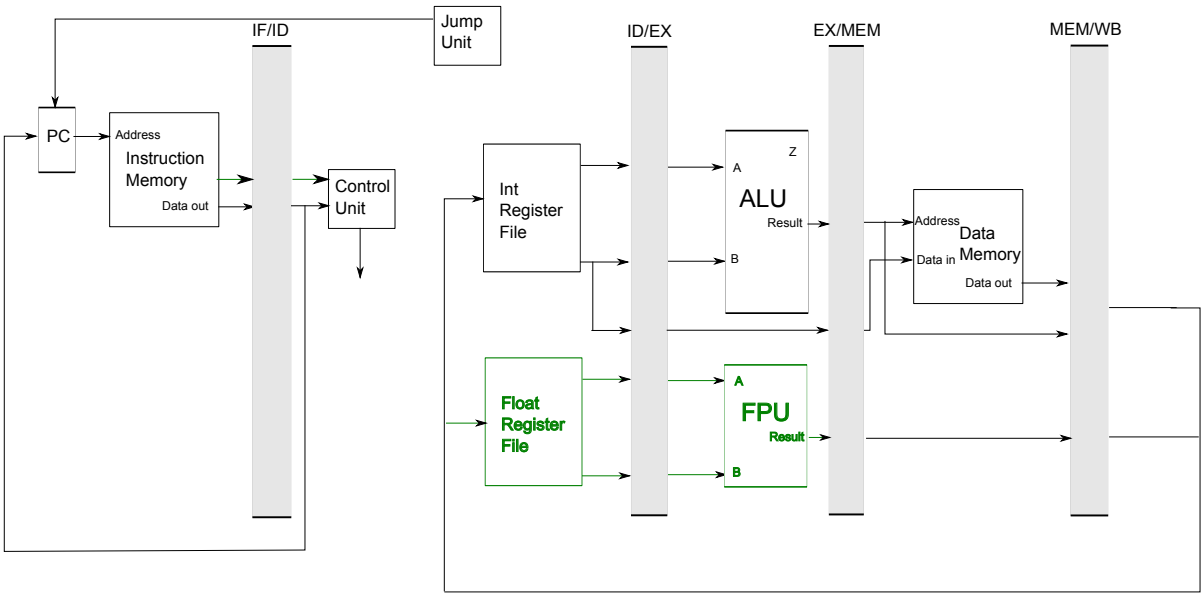


Figura 4: Computador capaz de ejecutar dos instrucciones al mismo tiempo, una ocupando la ALU, la otra la FPU.

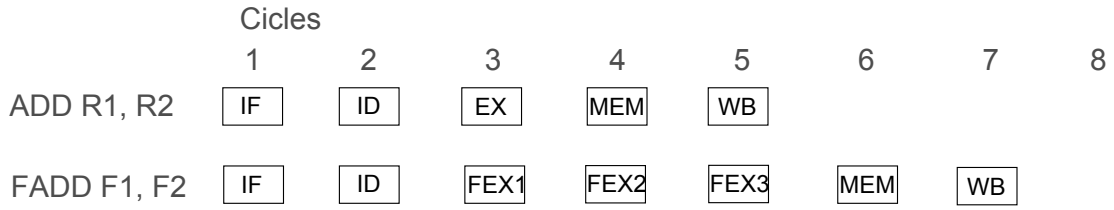


Figura 5: Diagrama de pipeline de dos instrucciones paralelizables.

Podemos ir más allá y extender esta idea, agregando ahora múltiples ALUs, y más registros, lo que permite ejecutar más instrucciones al mismo tiempo, como se observa en el ejemplo de la figura 6. Un procesador que permite obtener, decodificar y ejecutar múltiples instrucciones al mismo tiempo se conoce como **multiple-issue**. Si es capaz de procesar dos instrucciones al mismo tiempo, será un procesador 2-issue, si procesa 4, un 4-issue, y así. En la figura 7 se observa el esquemea del flujo de las instrucciones en un computador 4-issue.

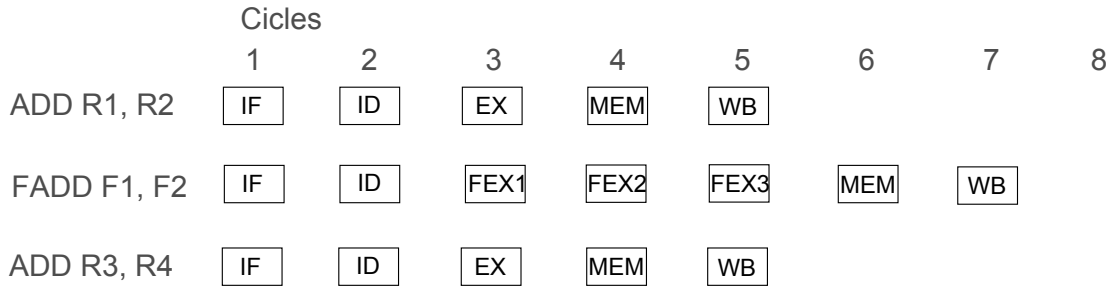


Figura 6: Diagrama de pipeline de un computador con múltiples ALUs y FPU.

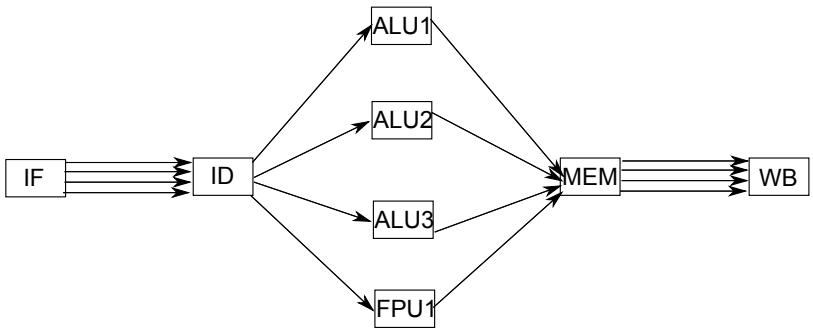


Figura 7: Esquema del flujo de instrucciones en un computador 4-issue.

En un procesador multiple-issue es necesario agregar elementos adicionales que permitan determinar que instrucciones pueden ejecutarse efectivamente en paralelo y cuales son. Existen dos tipos de técnicas para realizar esto: **estáticas** y **dinámicas**. Las técnicas estáticas se basan en que el compilador sea capaz de agrupar los conjuntos de instrucciones paralelizables y enviárselas en conjunto al procesador. Las técnicas dinámicas se basan en que el procesador sea capaz de determinar en tiempo de ejecución que instrucciones paralelizar, pudiendo despachar instrucciones a distintas unidades de ejecución dinámicamente.

Dentro de las técnicas dinámicas, la más utilizada se conoce como **Arquitectura Superescalar**. En este tipo de arquitecturas, antes de cada unidad de ejecución se agrega una estación de reserva o **reservation station** la cual será encargada de ir almacenando los operandos necesarios para una determinada operación, reservándolos hasta que estén todos y solo en ese momento se enviarán a la

unidad de ejecución. Luego de que se ejecute la operación, el resultado se almacena temporalmente en una **commit unit** (también llamada reorder buffer) la cual se encargará de enviar los datos a memoria o a los registros sólo cuando haya seguridad de que no hábrán problemas de dependencia de datos (figura 8).

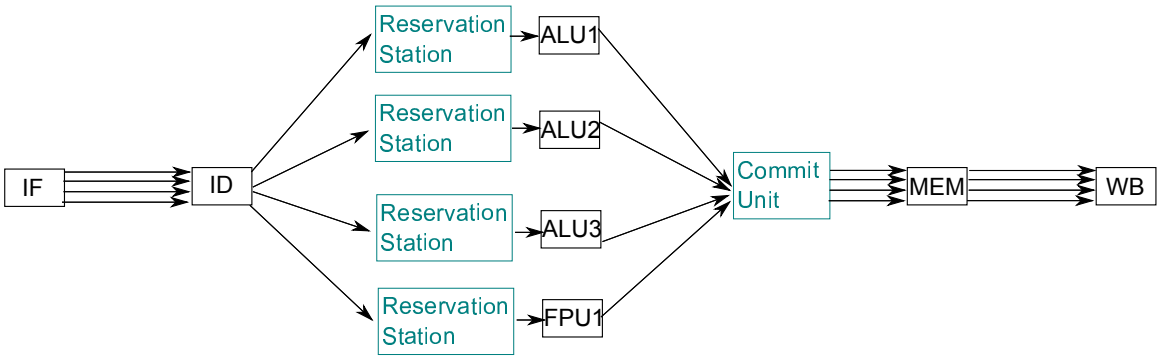


Figura 8: Esquema del flujo de instrucciones en un computador 4-issue superescalar.

Dentro de las técnicas estáticas, la más utilizada se conoce como **Very Long Instruction Word (VLIW)**, la cual consiste en que el compilador agrupe un paquete o **bundle** de instrucciones juntas, las cuales pueden ejecutarse en paralelo, y envíe al procesador este bundle como una «instrucción muy larga» que el procesador directamente envié a distintas unidades de ejecución en paralelo. Para lograr esto, el compilador puede implementar técnicas de **code motion**, es decir, reordenar el código de manera de permitir paralelizar instrucciones que no tienen dependencia de datos, pero sin perder la lógica original del algoritmo, como se observa en las tablas 1 y 2.

Dirección	Instrucción
0x00	Instrucción 1
0x01	Instrucción 2
0x02	Instrucción 3
0x03	Instrucción 4
0x04	Instrucción 5
0x05	Instrucción 6
0x06	Instrucción 7
0x07	Instrucción 8
0x08	Instrucción 9

Tabla 2: Secuencia de instrucciones sin VLIW.

Dirección	Bundle			
0x00	Instrucción 1	Instrucción 6	Instrucción 7	NOP
0x01	NOP	NOP	Instrucción 3	Instrucción 4
0x02	NOP	Instrucción 2	NOP	NOP
0x03	NOP	Instrucción 5	Instrucción 9	NOP
0x04	NOP	NOP	NOP	Instrucción 8

Tabla 3: Secuencia de bundles con VLIW.

Para lograr paralelismo en en arquitecturas multiple-issue es necesario explotar el paralelismo a nivel de instrucción que pueda tener disponible un determinado programa. Existen distintas métricas que de pueden utilizar sobre un programa para determinar su grado de paralelismo. Una de esas métricas se conoce como **Instruction Level Parallelism (ILP)**, la cual mide a nivel de las instrucciones del programa que tan paralelizable es un cierto código. La idea de esta métrica es determinar que tanto más rápido se puede ejecutar un código si se pudiera paralelizar lo más posible.

Para calcular el ILP se asume que cada instrucción toma un ciclo. El ILP se obtiene como la razón entre el número de ciclos que toma el programa sin nada de paralelismo, dividido por el número de ciclos que toma el programa aprovechando al máximo el paralelimso. Por ejemplo el siguiente código toma 4 ciclos en completarse sin paralelismo. Para determinar el número de ciclos con paralelismo hay que revisar la dependencia entre las instrucciones. En este caso, la instrucción 3 depende de la 1, por lo cual no pueden ser ejecutadas en paralelo, entonces las instrucciones 1, 2 y 4 pueden ejecutarse en paralelo (un ciclo), pero la instrucción 3 debe ejecutarse después de la 1, por lo tanto la mínima cantidad de ciclos en que se puede ejecutar el programa considerando paralelismo es 2 ciclos. El ILP en este caso entonces sería $\frac{4}{2} = 2$ lo que se interpreta como que el código puede correr hasta 2 veces más rápido si se tiene todo el paralelimso posible.

```
ADD [var1], 2           /*Instrucción 1*/
SUB [var2], 3           /*Instrucción 2*/
ADD [var1], 5   /*Instrucción 3*/
INC [var3]              /*Instrucción 4*/

RET

Data:

var1    db      2
var2    db      4
var3    db      5
```

Para determinar el ILP de programas más extensos, una herramienta útil es el **grafo de dependencia** del programa, el cual mapea visualmente todas las dependencias entre las instrucciones de un programa. Una vez obtenido el grafo de dependencias, es fácil determinar la cantidad de ciclos del código al ser ejecutado en paralelo, ya que corresponderá a la cantidad de instrucciones de la rama más profunda del grafo.

Para el siguiente código que originalmente toma 9 ciclos, se muestra su grafo de dependencia en la figura 14. Se puede observar que la rama mas profunda tiene 4 instrucciones, y por tanto al paralelizar el programa se puede reducir el tiempo a 4 ciclos. Con esto, el ILP de esté codigo sería $\frac{9}{4}$.

```
ADD [var1], 2
SUB [var2], 3
ADD BL, [var1]
ADD [var3], BL
MOV [var5], 0
ADD AL, [var5]
AND [var4], AL
XOR [var6], 7
```

```
INC [var3]

RET

Data:

var1    db      2
var2    db      4
var3    db      5
var4    db      6
var5    db      7
var6    db      1
```

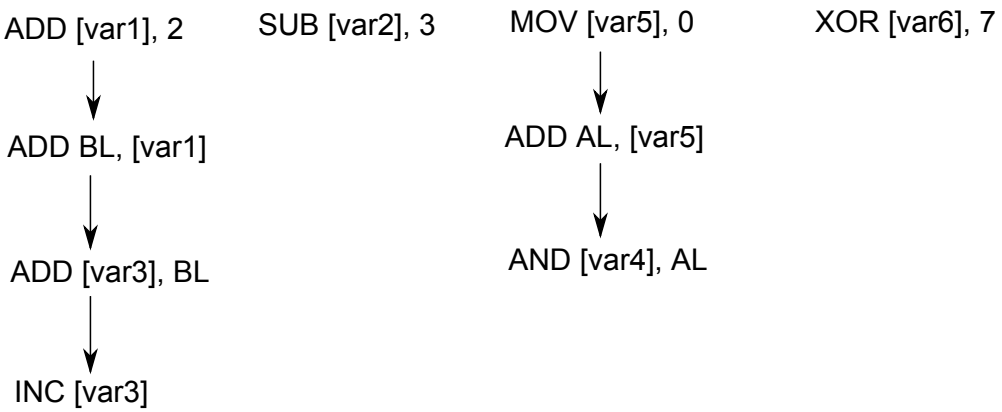


Figura 9: Ejemplo de un grafo de dependencias de instrucciones.

2.2. Paralelismo MIMD

Aunque el paralelismo dentro de un procesador logra mejoras de desempeño, tiene el costo asociado de aumentar la complejidad del procesador, lo que también repercute en el consumo de energía de éste. Una alternativa para aumentar el rendimiento del computador es en vez de complejizar más el procesador, trabajar con múltiples procesadores simples, repartiendo el procesamiento de múltiples programas independientes entre éstos, lo que se conoce como un sistema **multiprocesador**.

2.2.1. Tipos de multiprocesador

Existen dos tipos principales de multiprocesadores: multiprocesador por **paso de mensaje** y multiprocesador de **memoria compartida**. En un multiprocesador por paso de mensaje, cada procesador tiene una memoria y espacio de direccionamiento propio, y la comunicación con los demás procesadores se realiza a través de algún sistema de red que los interconecte (figura 10).

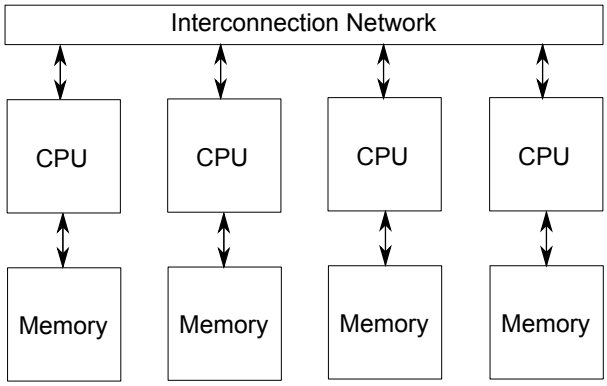


Figura 10: Diagrama multiprocesador de paso de mensajes

Dependiendo de la cercanía que tengan los distintos procesadores, existirán distintos tipos de multiprocesadores de paso de mensaje. Un **cluster** corresponde a un multiprocesador de paso de mensaje formado por distintos computadores comunicados por red local ubicados en una misma locación física. Un **sistema distribuido** corresponde a un multiprocesador de paso de mensajes en el cual los distintos computadores se encuentra ubicados físicamente distantes, pudiendo estar en países distintos, y realizando la comunicación mediante Internet.

El segundo tipo de multiprocesador, los multiprocesadores de memoria compartida se caracterizan por tener un nivel de memoria compartida entre los distintos procesadores (figura 11). Debido a esto es más simple compartir información entre los distintos procesadores, ya que no se requiere estar enviándola a través de la red, pero se generan problemas de sincronización, ya que se debe evitar que ambos procesadores modifiquen un mismo dato de memoria al mismo tiempo.

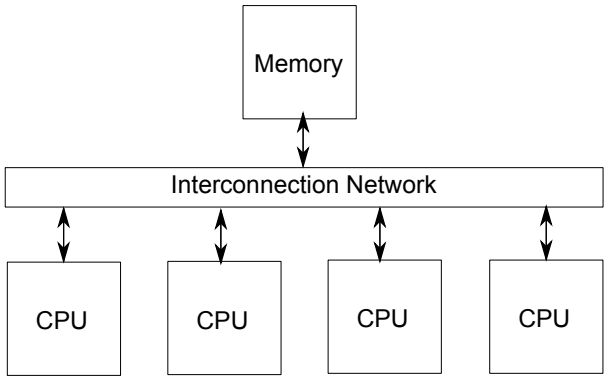


Figura 11: Diagrama multiprocesador de memoria compartida

Un problema particular en la sincronización de dos procesadores ocurre por el hecho de que a pesar de tener un nivel de memoria compartida, es probable que cada procesador tenga un caché no compartido (figura 12), lo que puede generar problemas de **coherencia de caché**, ya que si un procesador escribe un valor en su caché y esto no se le notifica al otro procesador, la información sobre una misma variable puede quedar almacenada de manera inconsistente en el sistema.

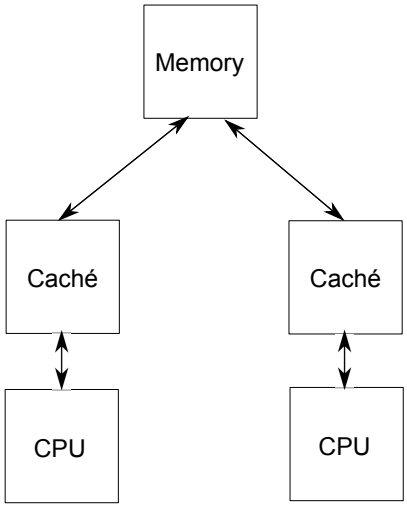


Figura 12: Diagrama de multiprocesador de memoria compartida, con cachés individuales

Un mecanismo para solucionar los problemas de coherencia de caché es el protocolo **MSI**. En este sistema, cada bloque puede estar en uno de cuatro estados:

- Modified: bloque distinto de memoria principal y no existe en otras caches.
- Shared: bloque coincide con memoria principal y puede estar presente en otras caches.
- Invalid: datos no son válidos.

La idea del protocolo es la siguiente (se asume que las cachés tienen política de escritura write-back):

- En un comienzo todos los bloques comienzan en estado Invalid.
- Si un procesador realiza una lectura de un valor de memoria, y lo almacena en su caché, ese bloque pasa a estar en estado Shared en el caché de ese procesador.
- Luego de esto pueden ocurrir distintas alternativas:
  - Si el mismo procesador escribe en ese bloque de caché, el estado del bloque pasa a Modified.
  - Si el segundo procesador lee esa dirección de memoria y lo almacena en su caché, ese bloque pasa a estar en estado Shared en la caché del segundo procesador.
- En caso de que el estado de bloque quedó en Modified, si ahora el segundo procesador intenta leer esa dirección de memoria, ocurrirá un problema de coherencia. Para evitar esto, el controlador de la caché con el bloque Modified, debe encargarse de hacer **bus snooping**, es decir «espiar» el bus de direcciones, para determinar si hubo un acceso a la dirección de memoria asociada al bloque Modified. En caso de detectar un acceso, el controlador le envía una señal a la segunda caché, indicando que el segundo procesador debe esperar (stall), por que si lee ahora habrá problema de inconsistencia. En ese momento el controlador de la primera caché pasa a escribir el bloque en la memoria principal compartida, para que cuando

el segundo procesador reintente el acceso, ahora el bloque ya no esté modificado, sino que se haya cambiado a estado Exclusive y no haya problemas de coherencia.

Supongamos los siguientes ejemplos de accesos a memoria de parte de dos procesadores:

Ejemplo 1:

1. El primer procesador ejecuta la instrucción `MOV A, (120)`, copiando el dato de memoria en la dirección 120 a un bloque de la caché. El bloque de caché en este procesador pasa de estado Invalid a Shared.
2. El segundo procesador ejecuta la instrucción `MOV B, (120)`, copiando el dato de memoria en la dirección 120 a un bloque de la caché. El bloque de caché en ambos procesadores pasa a Shared.
3. El primer procesador ejecuta la instrucción `MOV (120), B`, escribiendo en el bloque de caché, el cual pasa a estado Modified.

Ejemplo 2:

1. El primer procesador ejecuta la instrucción `MOV A, (120)`, copiando el dato de memoria en la dirección 120 a un bloque de la caché. El bloque de caché en este procesador pasa de estado Invalid a Shared.
2. El primer procesador ejecuta la instrucción `MOV (120), B`, escribiendo en el bloque de caché, el cual pasa a estado Modified.
3. El segundo procesador ejecuta la instrucción `MOV B, (120)`, intentando copiar el dato de memoria en la dirección 120 a un bloque de la caché. El controlador de caché del primer procesador está haciendo bus snooping y por tanto detecta que se quiere leer un bloque modificado y le envía una señal para que espere. Este controlador se preocupará de escribir inmediatamente en memoria, y actualizar el estado a Shared. Cuando el segundo procesado reintente el acceso, lo podrá hacer, y ahora ambos bloques quedarán en estado Shared.

2.3. Arquitecturas Multicore

Una arquitectura particular de multiprocesadores son las arquitecturas multicore, las cuales no corresponde a un tipo de multiprocesador, sino a una tecnología en la cual los distintos procesadores del sistema están incorporados en el mismo chip. El gran atractivo de estas arquitecturas es que al estar en el mismo chip la comunicación entre los procesadores será mucho más rápida.

Los sistemas multicore pueden ser tanto de paso de mensajes como memoria compartida, aunque habitualmente son de este último tipo, por ejemplo los procesadores multicore usados en los computadores personales. En éstos, en general el nivel de memoria que comparten los procesadores corresponde al nivel L2 de caché, teniendo cada uno una caché L1 independiente (figura 13).

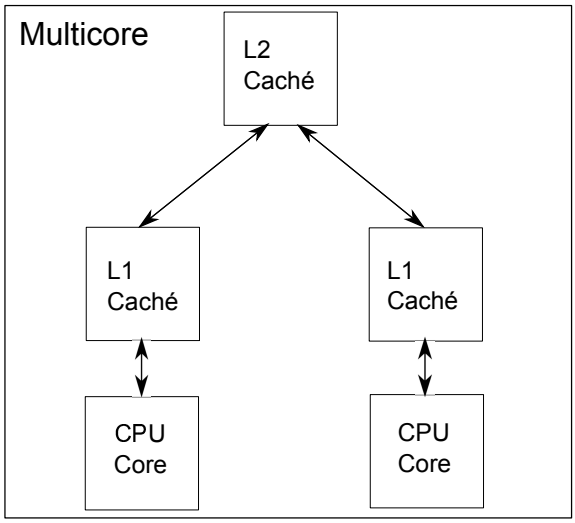


Figura 13: Diagrama de multiprocesador multicore

2.4. Paralelismo SIMD

Una categoría importante dentro de la taxonomía de Flynn son los sistemas SIMD, Single Instruction Multiple Data. La idea de estos sistemas es que en muchas circunstancias se requiere ejecutar la misma operación sobre distintos datos, por lo que es posible implementar naturalmente cierto grado de paralelismo en estos casos. Este paralelismo es particularmente importante para el procesamiento de operaciones vectoriales, que por su naturaleza suelen requerir operaciones de tipo SIMD: sumar un vector con otro, por ejemplo, corresponde a sumar (una instrucción) los distintos pares de componentes de ambos vectores (múltiples datos).

2.4.1. Extensiones Multimedia

Un ejemplo de un sistema SIMD implementado para el procesamiento de operaciones vectoriales corresponde a la extensión para procesamiento de datos multimedia que se incorporó en los computadores x86, denominada **Streaming SIMD Extension (SSE)**. Está extensión se agregó como una unidad de ejecución adicional al procesador, capaz de realizar operaciones sobre 8 registros vectoriales especialmente agregados (`XMM0` a `XMM7`). Estos registros son de 128 bits cada uno, pero para la unidad SSE pueden ser vistos tanto como un único registro, pero también como 4 registros de 32 bits, sobre los cuales se puede ejecutar una misma instrucción. En la figura 9 se observa como ejemplo la instrucción `MULPS XMM1, XMM0`, la cual multiplica cada uno de los 4 subregistros de 32 bits del registros `XMM0` con los del registro `XMM1` y los almacena en `XMM1`.

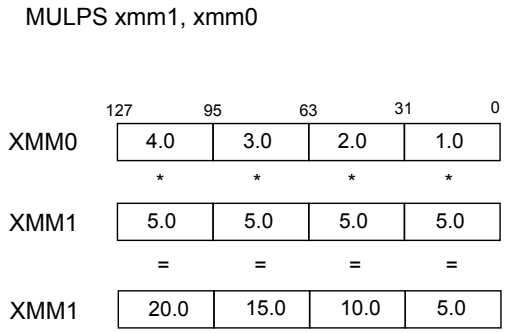


Figura 14: Ejemplo instrucción SSE.

2.4.2. Graphics Processing Unit

Una segunda arquitectura SIMD son las unidades de procesamientos gráficos o GPU. Las GPU son sistemas de multiprocesadores en los cuales cada procesador tiene funcionalidades específicas. En su conjunto, los procesadores de la GPU realizan el proceso de transformar representaciones de objetos 3D (vértices), primero a polígonos, y finalmente a una imagen 2D (figura 15).

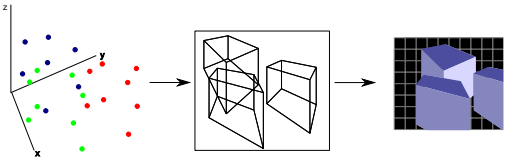


Figura 15: Etapas del procesamiento gráfico.

El procesamiento gráfico tiene la ventaja de ser extremadamente paralelizable, dado que en cada etapa se puede procesar de manera independiente las distinta unidades de información (vértices, polígonos, pixeles), lo que lo hace ideal para una arquitectura paralela de tipo SIMD, ya que se está realizando el mismo procesamiento sobre distintos datos al mismo tiempo. Además, el procesamiento es de tipo streaming, es decir, es en una dirección, y una vez que un dato pasa por una etapa, ya no vuelve a esta, por lo que no hay dependencia de datos, y se puede «pipelinizar» el proceso (figura 16).

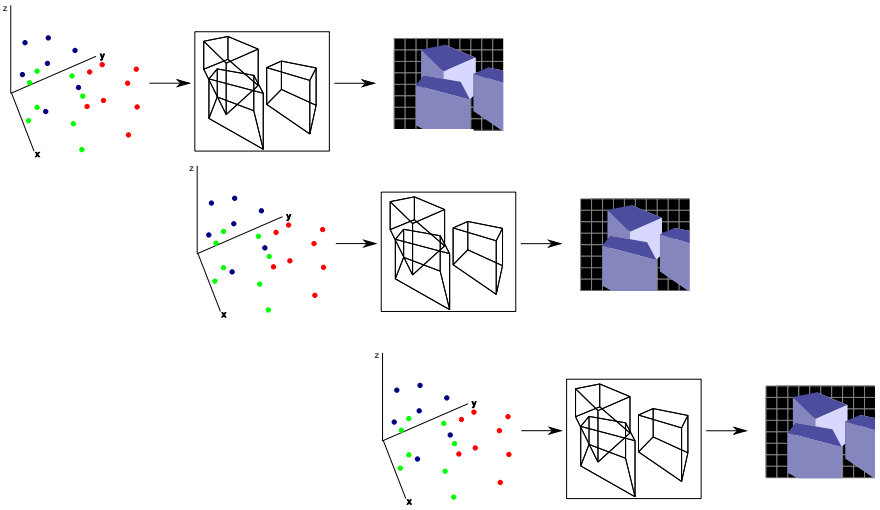


Figura 16: Graphics pipeline.

Las GPUs modernas llegan a tener cientos de procesadores, lo que permite que ejecuten muchas más operaciones por segundo que una CPU, lo que ha impulsado que en los últimos años se hayan comenzado a utilizar para realizar procesamiento general, no solamente gráfico. A pesar de esto, la CPU sigue siendo necesaria, ya que la GPU está optimizada para resolver un tipo particular de problema, pero no es óptima para problemas generales, en los que hayan dependencias de datos, saltos, y procesamiento lógico mas avanzado.

3. Referencias e información adicional

- Hennessy, J.; Patterson, D.: Computer Organization and Design: The Hardware/Software Interface, 4 Ed., Morgan-Kaufmann, 2008. Chapter 7: Multicores, multiprocessors and clusters.