Organización de Computadoras 2003

Apunte 1: Sistemas de Numeración: Sistemas Enteros y Punto Fijo

Los siquientes son ejercicios resueltos sobre sistemas enteros y punto fijo.

Conversiones entre los distintos sistemas

1) Convertir el número $(529)_{10}$ en su equivalente binario.

```
529 / 2 = 264  con resto 1 (LSD, dígito menos significativo)
264 / 2 = 132  con resto 0
132 / 2 =  66  con resto 0
 66 / 2 =  33  con resto 0
 33 / 2 =  16  con resto 1
 16 / 2 =   8  con resto 0
  8 / 2 =   4  con resto 0
  4 / 2 =   2  con resto 0
  2 / 2 =   1  con resto 0
  1 / 2 =   0  con resto 1 (MSD, dígito más significativo)
```

El número se lee de abajo hacia arriba, o sea 1000010001, de modo que $(529)_{10} = (1000010001)_2$

2) Convertir el número $(529)_{10}$ en su equivalente octal.

```
529 / 8 =  66  con resto 1 (LSD)
 66 / 8 =   8  con resto 2
  8 / 8 =   1  con resto 0
  1 / 8 =   0  con resto 1 (MSD)
```

El número se lee de abajo hacia arriba, o sea 1021, de modo que $(529)_{10} = (1021)_8$

3) Convertir el número $(529)_{10}$ en su equivalente hexadecimal.

```
529 / 16 =  33  con resto 1 (LSD)
 33 / 16 =   2  con resto 1
  2 / 16 =   0  con resto 2 (MSD)
```

El número se lee de abajo hacia arriba, o sea 211, de modo que $(529)_{10} = (211)_{16}$

4) Convertir la fracción $(0.371)_{10}$ en su equivalente binario.

```
0.371 x 2 = 0.742  con parte entera 0 (MSD, dígito más significativo)
0.742 x 2 = 1.484  con parte entera 1
0.484 x 2 = 0.962  con parte entera 0
0.962 x 2 = 1.936  con parte entera 1
0.936 x 2 = 1.872  con parte entera 1
0.872 x 2 = 1.744  con parte entera 1
0.744 x 2 = 1.488  con parte entera 1
0.488 x 2 = 0.976  con parte entera 0
0.976 x 2 = 1.952  con parte entera 1 (LSD, dígito menos significativo)
```

El número se lee de arriba hacia abajo, o sea 0.010111101, de modo que $(0.371)_{10} = (0.010111101)_2$

5) Convertir la fracción $(0.371)_{10}$ en su equivalente octal.

```
0.371 x 8 = 2.968  con parte entera 2 (MSD, dígito más significativo)
0.968 x 8 = 7.744  con parte entera 7
0.744 x 8 = 5.952  con parte entera 5 (LSD, dígito menos significativo)
```

El número se lee de arriba hacia abajo, o sea 0.275, de modo que $(0.371)_{10} = (0.275)_8$

6) Convertir la fracción $(0.371)_{10}$ en su equivalente hexadecimal.

```
0.371 x 16 =  5.936  con parte entera  5 (MSD)
0.936 x 16 = 14.976  con parte entera 14 (LSD)

El número se lee de arriba hacia abajo, o sea 5E (E es equivalente a 14), de modo que
(0.371)10 = (0.5E)16
```

7) Convertir el número binario 1001.101 en su equivalente decimal.

$$N = 1x2^3 + 0x2^2 + 0x2^1 + 1x2^0 + 1x2^{-1} + 0x2^{-2} + 1x2^{-3}$$
$$= 8 + 1 + 1/2 + 1/8$$
$$= 9.625$$

Por lo tanto, $(1001.101)_2 = (9.625)_{10}$

8) Convertir el número octal 1311.56 en su equivalente decimal.

$$N = 1x8^3 + 3x8^2 + 1x8^1 + 1x8^0 + 5x8^{-1} + 6x8^{-2}$$
$$= 512 + 192 + 8 + 1 + 0.625 + 0.09375$$
$$= 713.71875$$

Por lo tanto, $(1311.56)_8 = (713.71875)_{10}$

9) Convertir el número hexadecimal 2C9.B8 en su equivalente decimal.

$$N = 2x16^2 + Cx16^1 + 9x16^0 + Bx16^{-1} + 8x16^{-2}$$
$$= 2x16^2 + 12x16^1 + 9x16^0 + 11x16^{-1} + 8x16^{-2}$$
$$= 512 + 192 + 9 + 0.6875 + 0.03125$$
$$= 713.71875$$

Por lo tanto, $(2C9.B8)_{16} = (713.71875)_{10}$

10) Convertir el número hexadecimal 2E9.5C en su equivalente binario.

$(2E9.5C)_{16} = (0010\ 1110\ 1001.0101\ 1100)_{BCH}$ (BCH, hexadecimal codificado en binario)
$= (1011101001.010111)_2$

11) Convertir el número octal 631.47 en su equivalente binario.

$(631.47)_8 = (110\ 011\ 001.100\ 111)_{BCO}$ (BCO, octal codificado en binario)
$= (110011001.1)_2$

12) Convertir el número binario 111100011.101101 en su equivalente hexadecimal.

$(111100011.101101)_2 = (001\ 1110\ 0011.1011\ 0100)_{BCH} = (1E3.B4)_{16}$

13) Convertir el número binario 10111011.1011 en su equivalente octal.

$(10111011.1011)_2 = (010\ 111\ 011.101\ 100)_{BCO} = (273.54)_8$

14) Convertir el número octal 134.57 en su equivalente hexadecimal.

$(134.57)_8 = (001\ 011\ 100.101\ 111)_{BCO} = (1011100.101111)_2$
$= (0101\ 1100.1011\ 1100)_{BCH} = (5C.BC)_{16}$

15) Convertir el número hexadecimal F17.A6 en su equivalente octal.

$(F17.A6)_{16} = (1111\ 0001\ 0111.1010\ 0110)_{BCH} = (111100010111.1010011)_2$
$= (111\ 100\ 010\ 111.101\ 001\ 100)_{BCO} = (7427.514)_8$

Operaciones aritméticas

16) Realizar la suma entre $(101001011001.1111)_2$ y $(1111100.00011)_2$.

```
1111  11 111      Acarreos
```

```
   101001011001.1111
+       1111100.00011
   ------------------
   101011010110.00001
```

17) Realizar la suma entre $(5131.74)_8$ y $(174.06)_8$.

```
    1 1 1    Acarreos
   5131.74
+   174.06
   -------
   5326.02
```

18) Realizar la suma entre $(A59.F)_{16}$ y $(7C.18)_{16}$.

```
    12      Acarreos
   A59.F
+   7C.18
   ------
   AD6.08
```

19) Realizar la resta entre $(111010.001)_2$ y $(1111.00001)_2$.

```
        10  1 10                1
     0  0 10  0 10           0 10 10
   1  1  1  0  1  0  .  0  0  1  0  0
-        1  1  1  1  .  0  0  0  0  1
   ---------------------------------
   1  0  1  0  1  1  .  0  0  0  1  1
```

20) Realizar la resta entre $(72.1)_8$ y $(17.02)_8$.

```
   6 12     0 10
   7  2  .  1  0
-  1  7  .  0  2
   ------------
   5  3  .  0  6
```

21) Realizar la resta entre $(3A.2)_{16}$ y $(F.08)_{16}$.

```
   2 1A     1 10
   3  A  .  2  0
-     F  .  0  8
   ------------
   2  B  .  1  8
```

Representación en complemento a la base reducida en el sistema binario (Ca1)

22) Representar en Ca1 el número –68 tomando como base una computadora con palabra de 8 bits.

El número 68 sería 01000100, siendo el dígito más significativo el bit de signo. El Ca1 de 01000100 es 10111011 (se obtiene simplemente reemplazando los ceros por unos y los unos por ceros), entonces 10111011 es equivalente a –68.

23) Dar las representaciones posibles en Ca1 del 0 tomando como base una computadora con palabra de 8 bits.

En Ca1 hay dos representaciones posibles del 0, 00000000 y 11111111.

24) Averiguar qué número decimal expresa el binario 11010011 representado en Ca1 en una palabra de 8 bits.

Recomplementando el 11010011 se obtiene el 00101100, el cual representa el número decimal 44.

25) Sumar los números –28 y +122 representándolos en Ca1 en palabras de 8 bits.

```
  11100011    representación en Ca1 de –28
+ 01111010    representación en Ca1 de +122
  --------
 101011101
        1    se suma el acarreo
  --------
  01011110
```

Convirtiendo el resultado a decimal obtenemos que la suma da +94.

26) Representando en Ca1 en palabras de 8 bits efectuar la operación +43-93.

```
  00101011    representación en Ca1 de +43
+ 10011101    representación en Ca1 de -98
  --------
  11001000
```

Como el resultado es un número negativo (el bit de signo está en 1) para leerlo hay que recomplementarlo (obteniendo 00110111) y recién entonces convertirlo a decimal; en nuestro caso obtenemos como resultado –55.

Representación en complemento a la base en el sistema binario (Ca2)

27) Representar en Ca2 el número –56 tomando como base una computadora con palabra de 8 bits.

En el sistema binario el Ca2 de un número provee, al igual que en Ca1, el equivalente negativo del número que se está complementando.

```
56    00111000

Ca1   11000111
    +        1
      --------
Ca2   11001000     -56
```

28) Representar en Ca2 el número -88.

Una forma inmediata de obtener el Ca2 es recorrer los bits de derecha a izquierda y copiarlos tal cual están hasta el primer 1 inclusive, y los restantes bits tratarlos como al obtener el Ca1, o sea invertirlos, reemplazandos los ceros por unos y viceversa.

```
0101 1000     88
  <- ->
1010 1000    -88
```

29) Dar las representaciones posibles en Ca2 del 0 tomando como base una computadora con palabra de 8 bits.

La ventaja de la representación en Ca2 sobre Ca1 y BCS (binario con signo o representación en signo y módulo) es que el cero tiene una sola representación, no hay cero positivos y cero negativo como sucede en los otros dos casos, porque el Ca2 de 0 es 0, independientemente de la longitud de la palabra donde esté representado.

30) ¿Qué número decimal representa el número 11010010 representado en Ca2 en una palabra de 8 bits?.

Al igual que en Ca1, un número negativo no se puede leer directamente convirtiéndolo a decimal, ya que no obtendremos el equivalente decimal del número que está representando. Para saber de qué número se trata hay que recomplementarlo y recién entonces hacer la conversión.

En nuestro caso, si recomplementamos el 11010010 obtenemos 00101110, que representa el número decimal +46.

31) Efectuar la operación 117-36 representando los números en Ca2 en palabras de 8 bits.

Al sumar dos números en Ca2 también se incluye en la operación el bit de signo. El acarreo producido por el bit de signo se desprecia (no se tiene en cuenta).

```
   01110101   representación en Ca2 de 117
+  11011100   representación en Ca2 de –36
   --------
  101010001   el acarreo se desprecia
```

Como el acarreo se desprecia el resultado es 01010001, que convirtiéndolo a decimal nos da 81.

32) Sumar los números –115 y +87 representándolos en Ca2 en palabras de 8 bits.

```
   10001101   representación en Ca2 de -115
+  01010111   representación en Ca2 de   87
   --------
   11100100
```

Como el resultado es un número negativo (el bit de signo es 1) para leerlo primero hay que recomplementarlo, obteniendo 00011100, y recién entonces convertirlo a decimal; el resultado de esta operación es –28.

Representación en Exceso

En general los excesos son a la 2^{n-1}, para que haya igual cantidad de números positivos y negativos. El exceso, a diferencia del Ca1 y Ca2, si empieza con 0 es negativo y si empieza con 1 es positivo. En exc eso, al igual que en Ca2, existe una única representación del 0.

33) Interpretar el valor de 10110110 que se encuentra en exceso.

Si un número está en exceso y quiero saber su valor le resto el exceso (2^{n-1}). Cuando empieza con 1 es positivo y le resto 128, es decir, 10000000 (también conocido como universal/2 o U/2).

```
   10110110        188
-  10000000      - 128
   --------        ---
   00110110         60
```

Por lo tanto, obtenemos que 10110110 es el 60 sin exceso.

34) Interpretar el valor de 00011101 que se encuentra en exceso.

Si un número en exceso empieza con 0 es negativo, y para saber su valor se calcula –(U/2-a).

```
  10000000        128
- 00011101      -  29    El 29 es el número en exceso
  --------        ---
  01100011         99    El –99 es el número sin exceso
```

Por lo tanto, obtenemos que 00011101 es el –99 sin exceso.

35) Escriba el 15 en exceso con 8 bits.

```
  00001111         15
+ 10000000      + 128
  --------        ---
  10001111        143
```

Por lo tanto, el 15 en exceso es 10001111.

36) Escriba el -123 en exceso con 8 bits.

El 123 en binario es 01111011

```
  10000000        128
```

```
- 01111011      - 123
  --------        ---
  00000101          5
```

Por lo tanto, el –123 en exceso es 00000101.

Overflow y Carry

Tanto en la representación en Ca1 como en Ca2 una operación puede dar como resultado un número que excede la capacidad de la palabra de memoria, produciéndose así el overflow.

Al sumar dos números el overflow se puede dar sólo si los dos tienen el mismo signo; la suma de dos números de distinto signo nunca dará como resultado un número con módulo mayor al de mayor módulo de los dados, al máximo será igual (al sumarle 0 a otro número), pero en general será menor, por lo tanto no puede exceder la capacidad de la palabra de memoria.

El overflow se reconoce cuando los bits de signo de los dos números que se suman son iguales entre si pero distintos del bit de signo del resultado, o sea cuando los números son positivos y da resultado negativo o viceversa. En este caso el contenido de la palabra de memoria es incorrecta.

37) Sumar 5 y 3 en representación en BSS en palabras de 3 bits.

```
   101   representación en BSS de 5
+  011   representación en BSS de 3
  ----
  1000
```

Si tengo una suma en BSS y me da carry significa que el resultado es erróneo. Me da algo por afuera de lo que puedo escribir. El resultado no se puede escribir en el rango que hay.

38) Sumar 5 y 6 en representación en BCS en palabras de 4 bits.

```
  0101   representación en BCS de 5
+ 0110   representación en BCS de 6
  ----
  1011   representación en BCS de -2
```

La suma da overflow ya que al sumar los dos positivos dio negativo. El resultado es incorrecto.

39) Sumar 92 y 53 en representación en Ca1 en palabras de 8 bits.

```
   92          01011100   representación en Ca1 de 92
+  53       +  00110101   representación en Ca1 de 53
  ---          --------
  145          10010001
```

El bit más significativo es 1, por lo tanto, hay overflow.

En la suma representada en decimal se puede deducir que habrá overflow porque el resultado es mayor que 127, que es el mayor número representable en Ca1 en una palabra de 8 bits.

40) Sumar -3 y 3 en representación en Ca2 en palabras de 3 bits.

```
   101   representación en Ca2 de -3
+  011   representación en Ca2 de 3
  ----
  1000
```

Si me da carry cuando trabajo en Ca2 se desprecia, ya que el resultado queda bien.

41) Sumar -83 y -70 en representación en Ca2 en palabras de 8 bits.

```
  -83          10101101   representación en Ca2 de -83
+ -70       +  10111010   representación en Ca2 de -70
  ---          --------
 -153         101100111
```

El acarreo en Ca2 se desprecia. El bit más significativo es 1, por lo tanto, hay overflow. El overflow cuando trabajo en Ca2 significa que el resultado es erróneo.

Observando el resultado en decimal se puede asegurar que habrá overflow, ya que es más chico que –128, que es el menor número que se puede representar en Ca2 en una palabra de 8 bits.0

Capacidad de representación, resolución y rango en sistemas restringidos a n bits

Capacidad de representación: Es la cantidad de números que se pueden representar. Ya sea en punto fijo o no es b^n. Por ejemplo, si tengo un sistema restringido a 5 bits, sería 2^5 números, es decir, 32 números.

Resolución: Es la mínima diferencia entre un número representable y el siguiente. Se podría decir que es la diferencia entre el 0 y el siguiente. Por ejemplo, en binario con dos dígitos fraccionarios es 0.01.

Rango: El rango de un sistema está dado por el número mínimo representable y el número máximo representable. Por ejemplo, en binario con cinco dígitos es [0, 31] (donde el 0 es 00000 y el 31 es 11111). El número máximo representable en un sistema para la parte entera es b^n-1.

42) Indicar cuál es la capacidad de representación, la resolución y el rango de un sistema BSS de 5 bits.

El número máximo representable es 11111, es decir, el 31 decimal (o 2^5-1).
El número mínimo representable es 00000, es decir, el 0 decimal.
Por lo tanto, el rango es [0, 31]. Como podemos ver el rango en un sistema BSS es [0 … 2^n-1].
La capacidad de representación es 2^5, es decir, 32 números.
La resolución es 1, lo cual obtenemos al hacer la resta entre 00000 y su próximo número representable, es decir, 00001.

43) Indicar cuál es la capacidad de representación, la resolución y el rango de un sistema BCS de 5 bits.

El número máximo representable es 01111, es decir, el 15 decimal (o 2^4-1).
El número mínimo representable es 11111, es decir, el -15 decimal.
Por lo tanto, el rango es [-15, 15]. Como podemos ver el rango en un sistema BCS es [-(2^{n-1}-1) … 2^{n-1}-1].
La capacidad de representación sigue siendo b^n, es decir 2^5, 32 números (hay dos representaciones posibles del 0).
La resolución sigue siendo 1.

44) Indicar cuál es la capacidad de representación, la resolución y el rango de un sistema binario con 4 bits para la parte entera y 3 para la parte fraccionaria.

El número máximo representable es 1111.111.

$Vmax = 1x2^3 + 1x2^2 + 1x2^1 + 1x2^0 + 1x2^{-1} + 1x2^{-2} + 1x2^{-3}$
$= 8 + 4 + 2 + 1 + 1/2 + 1/4 + 1/8$
$= 15 + 0.875$
$= 15.875$

El número mínimo representable es 0000.000, es decir, 0.
Por lo tanto, el rango es [0, 15.875].
La capacidad de representación es 2^7, es decir, 128 números.
La resolución es de 0.125, obteniéndolo al interprentar el valor de 0000.001.
Como podemos ver, con los números fraccionarios perdemos rango pero podemos representar números con más precisión.

45) Indicar cuál es la capacidad de representación, la resolución y el rango de un sistema binario con 3 bits para la parte entera y 2 para la parte fraccionaria.

El número máximo representable es 111.11.

$Vmax = 1x2^{2} + 1x2^{1} + 1x2^{0} + 1x2^{-1} + 1x2^{-2}$
$= 4 + 2 + 1 + 1/2 + 1/4$
$= 6 + 0.75$
$= 6.75$

El número mínimo representable es 000.00, es decir, 0.
Por lo tanto, el rango es [0, 6.75].
La capacidad de representación es 2^{5}, es decir, 32 números.
La resolución es de 0.25, obteniéndolo al interprentar el valor de 000.01.

46) Especificar cuál es el rango de un sistema de representación binaria entera con n bits, con signo, en Ca1, Ca2 y en exceso.

El rango en Ca1 es $[-(2^{n-1}-1) \ldots (2^{n-1}-1)]$.
El rango en Ca2 es $[-(2^{n-1}-1)-1 \ldots (2^{n-1}-1)]$, o $[-2^{n-1} \ldots (2^{n-1}-1)]$.
El rango en exceso es $[-(2^{n-1}) \ldots (2^{n-1}-1)]$.

Organización de Computadoras 2004

Apunte 2: Sistemas de Numeración: Punto Flotante

La coma o punto flotante surge de la necesidad de representar números reales y enteros con un rango de representación mayor que el que nos ofrece punto fijo. En la representación en coma flotante, se dividen los n bits disponibles para representar un dato, en 2 partes llamadas mantisa M y exponente E. Considerando que la mantisa tiene una longitud de p bits y que el exponente la tiene de q bits, se cumple que n = p + q.

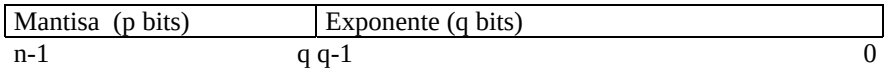
La mantisa contiene los dígitos significativos del dato, en tanto que el exponente indica el factor de escala, en forma de una potencia de base r. Por todo ello, el valor del número viene dado por:

$$V(X) = M * r^E$$

El número X viene representado por la cadena:

$$X = (m_{p-1}, ..., m_1, m_0, e_{q-1}, ... , e_1, e_0)$$

Cuya representación gráfica es:



La mantisa M y el exponente E se representan en alguno de los sistemas de punto fijo.

El rango de representación estará dado por:

Mínimo número negativo: (mantisa máxima) * base $^{\text{máximo exponente positivo}}$
Máximo número negativo: - (mantisa mínima) * base $^{\text{máximo exponente negativo}}$
Mínimo número positivo: (mantisa mínima) * base $^{\text{máximo exponente negativo}}$
Máximo número positivo: mantisa máxima) * base $^{\text{máximo exponente positivo}}$

Ejemplo:

Supongamos que tenemos un sistema de numeración con las siguientes características:

Mantisa de 5 bits, expresada en BSS.
Exponente de 3 bits, expresada en BCS.
Base 2.

Entonces, la cadena

10110 101

representará a $22 * 2^{-1} = 11$

Mantisa fraccionaria

En este caso, la mantisa en lugar de interpretarse como un número entero, se toma como un número real con el punto decimal implícito a la izquierda de sus bits.

Siguiendo con el ejemplo anterior de mantisa de 5 bits en BSS, exponente de 3 bits en BCS y base 2.

Entonces, la cadena

10110 101

representará a $(1/2 + 1/8 + 1/16) * 2^{-1} = 0.6875 * 2^{-1} = 0,34375$

y la cadena

00101 011

representará a $(1/8 + 1/32) * 2^{3} = (5/32) * 2^{3} = 5/4 = 1.25$

este valor coincide con el de la cadena

01010 010 = $(1/4 + 1/16) * 2^{2} = (5/16) * 2^{2} = 1.25$

y con el de la cadena

10100 001 = $(1/2 + 1/8)\ 2^{1} = (5/8) * 2^{1} = 1.25$

Para evitar tener múltiples representaciones de un mismo número, se puede representar la mantisa en forma ***normalizada***. La normalización consiste en hacer que su primer dígito tenga el valor 1. De esta manera, un número no puede tener mas de una representación.

En el ejemplo anterior, en un sistema con mantisa normalizada, de las 3 representaciones posibles para el número 1,25 solo sería valida 10100 001, ya que en ella la mantisa empieza con un 1.

La desventaja de este sistema de representación es que no se puede representar el número 0.

Dado que las mantisas normalizadas siempre empiezan con un 1, podemos no almacenar este dígito, con lo cual la mantisa tiene un bit de mas para almacenar. Por lo tanto,

$$p + q = n + 1$$

Este bit implícito se agrega en el momento de operar. A este sistema se lo llama ***mantisa fraccionaria normalizada con bit implicito.***

Dado que no siempre podemos representar exactamente el número que queremos, definimos *error absoluto* y *error relativo* de un número en un sistema de la siguiente forma:

$$EA(x) = | x' - x |$$
$$ER(x) = EA(x) / x$$

donde x' es el número representable del sistema más próximo a x.

Ejercicios

1. ***Mantisa entera en BSS de 8 bits.***
Exponente en Ca2 en 4 bits.
Donde los primeros 8 bits representan la mantisa, y los siguientes 4 al exponente.

Queremos obtener:

a) La representación del número 0,40625:

0,40625 * (2 * 2 * 2 * 2 * 2) = 13

$0,40625 * 2^5 = 13$
$0,40625 = 13/2^{-5}$
$0,40625 = 13 * 2^{-5}$
0,40625 = 00001101 1011

b) El número máximo:

mayor mantisa por mayor exponente: 11111111 0111 = $255 * 2^7 = 32640$

c) El número mínimo:

menor mantisa por menor exponente: 00000000 1000 = $0 * 2^{-8} = 0$

d) Resolución máxima:

mayor número representable:	11111111 0111	= 32640
anterior número representable:	11111110 0111	= 32512
diferencia:		= 128

e) Resolución mínima:

menor número representable:	00000000 1000	= 0
siguiente número representable:	00000001 1000	= 1/256
diferencia:		= 1/256

2. ***Mantisa fraccionaria en BCS de 6 bits.***
Exponente en Ex2 en 3 bits.
Donde el primer bit representa el signo, los 5 siguientes representan la mantisa, y los siguientes 3 al exponente.

Queremos obtener:

a) La representación del número 4,75:

$4,75 * (2 * 2) = 19$
$4,75 = 19 / 4$
tenemos que lograr que la mantisa sea una fracción, por tanto, multiplicamos y dividimos por 8
$4,75 = (19 * 8)/(4 * 8)$

reagrupando obtenemos
$4,75 = (19 / 32) * 8$
$4,75 = (19 / 32)* 2^3$
4,75 = 010011 111

b) El número máximo positivo:

mayor mantisa positiva por mayor exponente positivo: 011111 111 = $(31/32) * 2^3 = 7,75$

c) El número mínimo positivo:

menor mantisa positiva por mayor exponente negativo: 000000 000 = $0 * 2^{-4} = 0$

d) El número máximo negativo:

mayor mantisa negativa por mayor exponente positivo: 111111 111 = $-(31/32) * 2^3 = -7,75$

e) El número mínimo negativo:

menor mantisa negativa por menor exponente negativo: 100000 000 = 0 * 2^{-4} = 0

f) Resolución máxima positiva:

mayor número representable:	011111 111 = (31/32) * 2^3 = 7,75	
anterior número representable:	011110 111 = (30/32) * 2^3 = 7,50	
diferencia:		= 0,25

g) Resolución mínima positiva:

menor número representable positivo:	000000 000	= 0 * 2^{-4}
siguiente número representable:	000001 000	= (1/32) * 2^{-4}
diferencia:		= (1/32) * 2^{-4} = 2^{-9}

h) Resolución máxima negativa:

mayor número representable negativo:	111111 111 = (31/32) * 2^3= -7,75	
anterior número representable:	111110 111 = (30/32) * 2^3= -7,50	
diferencia:		= 0,25

i) Resolución mínima negativa:

menor número representable negativo:	100000 000	= 0 * 2^{-4}
siguiente número representable:	100001 000	= -(1/32) * 2^{-4}
diferencia:		= (1/32) * 2^{-4} = 2^{-9}

3. ***Mantisa fraccionaria normalizada en BCS de 6 bits.***
Exponente en Ex2 en 3 bits.
Donde el primer bit representa el signo, los 5 siguientes representan la mantisa, y los siguientes 3 al exponente.

Queremos obtener:

a) La representación del número -0,140625:

-0,140625 * (2 * 2 * 2 * 2 * 2 * 2) = -9
-0,140625 * 2^6 = -9
-0,140625 = -9 / 2^6
-0,140625 = (-9 / $2^{5)}$ * 2^{-1}
-0,140625 = 101001 011

como no esta normalizada la mantisa, hacemos un corrimiento
-0,140625 = 101001 011 = 110010 010

b) El número máximo positivo:

mayor mantisa positiva por mayor exponente positivo: 011111 111 = (31/32) * 2^3 = 7,75

c) El número mínimo positivo:

menor mantisa positiva por mayor exponente negativo: 010000 000 = (1/2) * 2^{-4} = 1/32

d) El número máximo negativo:

mayor mantisa negativa por mayor exponente positivo: 111111 111 = -(31/32) * 2^3 = -7,75

e) El número mínimo negativo:

menor mantisa negativa por menor exponente negativo: 110000 000 = -(1/2) * 2^{-4} = -1/32

f) Resolución máxima positiva:

mayor número representable positivo:	011111 111	= 7,75
anterior número representable:	011110 111	= 7,5
diferencia:		= 0,25

g) Resolución mínima positiva:

menor número representable positivo:	010000 000	= (1/2) * 2^{-4}
siguiente número representable:	010001 000	= (17/32) * 2^{-4}
diferencia:		= (1/32) * 2^{-4} = 2^{-9}

h) Resolución máxima negativa:

mayor número representable negativo:	111111 111	= -7,75
anterior número representable:	111110 111	= -7,5
diferencia:		= 0,25

i) Resolución mínima negativa:

menor número representable negativo:	110000 000	= -(1/2) * 2^{-4}
siguiente número representable:	110001 000	= -(17/32) * 2^{-4}
diferencia:		= (1/32) * 2^{-4} = 2^{-9}

4. ***Mantisa fraccionaria normalizada con bit implícito en BCS de 6 bits.***
Exponente en Ex2 en 3 bits.
Donde el primer bit representa el signo, los 5 siguientes representan la mantisa, y los siguientes 3 al exponente.

Queremos obtener:

a) La representación del número 1,96875:

1,96875 * (2 * 2 * 2 * 2 * 2) = 63
1,96875 * 2^5 = 63
1,96875 = 63 / 2^5
1,96875 = (63 / 2^5) * 2^0 = (63 / 2^6) * 2^1
1,96875 = 011111 101

b) El número máximo positivo:

mayor mantisa positiva por mayor exponente positivo: 011111 111 = (63/64) * 2^3 = 7,875

c) El número mínimo positivo:

menor mantisa positiva por mayor exponente negativo: 000000 000 = (1/2) * 2^{-4} = 1/32

d) El número máximo negativo:

mayor mantisa negativa por mayor exponente positivo: 111111 111 = -(63/64) * 2^{3} = -7,875

e) El número mínimo negativo:

menor mantisa negativa por menor exponente negativo: 100000 000 = -(1/2) * 2^{-4} = -1/32

f) Resolución máxima positiva:

mayor número representable positivo:	011111 111	= 7,875
anterior número representable:	011110 111	= 7,75
diferencia:		= 0,125

g) Resolución mínima positiva:

menor número representable positivo:	000000 000	= (1/2) * 2^{-4}
siguiente número representable:	000001 000	= (33/64) * 2^{-4}
diferencia:		= (1/64) * 2^{-4} = 2^{-10}

h) Resolución máxima negativa:

mayor número representable negativo:	111111 111	= -7,875
anterior número representable:	111110 111	= -7,75
diferencia:		= 0,125

i) Resolución mínima negativa:

menor número representable negativo:	100000 000	= -1/32
siguiente número representable:	100001 000	= -(33/64) * 2^{-4}
diferencia:		= (1/64) * 2^{-4} = 2^{-10}

5. ***Mantisa fraccionaria entera en Ca1 de 8 bits.***
Exponente en BSS en 4 bits.
Donde los primeros 8 bits representan la mantisa, y los siguientes 4 al exponente.

Dadas las siguientes representaciones:

p = 00011010 1101
q = 00001001 1111

Queremos hallar el resultado de la suma de las mismos.

Para esto, tenemos que igualar los exponentes. Dado que el exponente de p es menor que el del q, o bien subimos el exponente de p, o bien bajamos el de q. Subir el exponente de un número implica hacer un corrimiento a derecha de la mantisa; bajar el exponente implica hacer un corrimiento a izquierda de la mantisa.

- Subir el exponente de p:

Subiendo en 1 el exponente de p obtenemos:

00001101 1110

Todavía el exponente sigue siendo menor que el de q, con lo cual tendríamos que hacer otro corrimiento. Pero dado que el ultimo bit de la mantisa es un 1, no lo podemos hacer.

- Bajar el exponente de q:

Bajando en 1 el exponente de q obtenemos:

00010010 1110

Todavía el exponente sigue siendo mayor que el de p, con lo cual tendríamos que hacer otro corrimiento.

Bajando en 1 el exponente obtenemos:

00100100 1101

Con lo cual conseguimos igualar los exponentes.

Ahora realizamos la suma de las mantisas:

00011010
00100100
00111110

Por tanto, el resultado de la suma es: 00111110 1101

IEEE 754

El IEEE 754 es un estándar de aritmética en coma flotante. Este estándar especifica como deben representarse los números en coma flotante con simple precisión (32 bits) o doble precisión (64 bits), y también cómo deben realizarse las operaciones aritméticas con ellos.

Emplea mantisa fraccionaria, normalizada y en representación signo magnitud, sin almacenar el primer dígito, que es igual a 1. El exponente se representa en exceso, que en este caso no se toma como 2^{n-1}, sino como $2^{n-1} - 1$

Simple Precisión
El estándar IEEE-754 para la representación en simple precisión de números en coma flotante exige una cadena de 32 bits. El primer bit es el bit de signo (S), los siguientes 8 son los bits del exponente (E) y los restantes 23 son la mantisa (M):

Simple precisión

S	Exponente	mantisa
0	1 8	9 31

En los formatos IEEE no todas lasa cadenas se interpretan de la misma manera. Algunas combinaciones se utilizan para representar valores especiales.

El valor V representado por esta cadena puede ser determinado como sigue:

Exponente	Mantisa	Signo	Valor
0 < E < 255	Cualquiera	1	$-1 * 2^{E-127} * 1.M$ donde "1.M" se emplea para representar el número binario creado por la anteposición a M de un 1 y un punto binario
0 < E < 255	Cualquiera	0	$2^{E-127} * 1.M$ donde "1.M" se emplea para representar el número binario creado

			por la anteposición a M de un 1 y un punto binario
255	No nulo	Cualq.	NaN ("Not a number"). Se aplica para señalar varias condiciones de error.
255	0	1	-Infinito
255	0	0	Infinito
0	No nulo	1	$-(2^{-126})*(0.M)$ (Números sin normalizar)
0	No nulo	0	$2^{-126} * (0.M)$ (Números sin normalizar)
0	0	1	-0
0	0	0	0

En particular,

0 00000000 00000000000000000000000 = 0
1 00000000 00000000000000000000000 = -0

0 11111111 00000000000000000000000 = Infinito
1 11111111 00000000000000000000000 = -Infinito

0 11111111 00000100000000000000000 = NaN
1 11111111 00100010001001010101010 = NaN

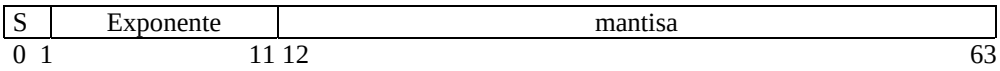
0 10000000 00000000000000000000000 = $2^{(128-127)} * 1.0 = 2$
0 10000001 10100000000000000000000 = $2^{(129-127)} * 1.101 = 6.5$
1 10000001 10100000000000000000000 = $-1 * 2^{(129-127)} * 1.101 = -6.5$

0 00000001 00000000000000000000000 = $2^{(1-127)} * 1.0 = 2^{(-126)}$
0 00000000 10000000000000000000000 = $2^{(-126)} * 0.1$

Doble precisión

El estándar IEEE-754 para la representación en doble precisión de números en coma flotante exige una cadena de 64 bits. El primer bit es el bit de signo (S), los siguientes 11 son los bits del exponente (E) y los restantes 52 son la mantisa (M):

Doble Precisión



El valor V representado por esta cadena puede ser determinado como sigue:

Exponente	Mantisa	Signo	Valor
$0 < E < 2047$	Cualquiera	1	$-1 * 2^{E-1023} * 1.M$ donde "1.M" se emplea para representar el número binario creado por la anteposición a M de un 1 y un punto binario
$0 < E < 2047$	Cualquiera	0	$2^{E-1023} * 1.M$ donde "1.M" se emplea para representar el número binario creado por la anteposición a M de un 1 y un punto binario
2047	No nulo	Cualq.	NaN ("Not a number"). Se aplica para señalar varias condiciones de error.
2047	0	1	-Infinito
2047	0	0	Infinito
0	No nulo	1	$-(2^{-1022})*(0.M)$ (Números sin normalizar)
0	No nulo	0	$2^{-1022} * (0.M)$ (Números sin normalizar)
0	0	1	-0
0	0	0	0

En particular,

0 00000000000 0000000000000000000000000000000000000000000000000000 = 0
1 00000000000 0000000000000000000000000000000000000000000000000000 = -0

0 11111111111 0000000000000000000000000000000000000000000000000000 = Infinito
1 11111111111 0000000000000000000000000000000000000000000000000000 = -Infinito

0 11111111111 0000010000000000000000000000000000000000000000000000 = NaN
1 11111111111 0010001000100101010101000000000000000000000000000000 = NaN

0 10000000000 0000000000000000000000000000000000000000000000000000 = $2^{(1024-1023)} * 1.0 = 2$
0 10000000001 1010000000000000000000000000000000000000000000000000 = $2^{(1025-1023)} * 1.101 = 6.5$
1 10000000001 1010000000000000000000000000000000000000000000000000 = $-1 * 2^{(1025-1023)} * 1.101 = -6.5$

0 00000000001 0000000000000000000000000000000000000000000000000000 = $2^{(1-1023)} * 1.0 = 2^{(-1022)}$
0 00000000000 1000000000000000000000000000000000000000000000000000 = $2^{(-1022)} * 0.1$

Organización de Computadoras 2003

Apunte 3: Sistemas de Numeración: Operaciones Lógicas

Para comprender este tema, me parece apropiado que repasen el tema de cálculo proposicional introducido en el curso de ingreso (en este apunte hay una breve introducción al respecto). De esta manera, con el concepto de conectivos lógicos firme, vamos a entender los distintos usos que podemos dar a las operaciones lógicas en Informática. Como siguiente paso voy a describir el concepto básico de operador, para finalizar con las operaciones lógicas que podemos realizar en lenguaje Assembler. El apunte finaliza con una serie de ejercicios prácticos sobre el tema.

Espero que este apunte les sea de utilidad, y si tienen dudas sobre los temas expuestos o quieren profundizar en alguno de ellos, al final se agrega la bibliografía utilizada.

Introducción: "Cálculo Proposicional"

El cálculo proposicional es el estudio de las relaciones lógicas entre objetos llamados proposiciones, que generalmente pueden interpretarse como afirmaciones que tienen algún significado en contextos de la vida real. Para nosotros, una proposición será cualquier frase que sea verdadera o falsa, pero no ambas.

Recordemos del curso de ingreso y de programación de computadoras que en el cálculo proposicional se utilizan letras minúsculas (ej: p, q, r) para simbolizar proposiciones, que se pueden combinar utilizando *conectivos lógicos*:

$\neg$	para "no" o negación
$\wedge$	Para "y"
$\vee$	para "o"
$\rightarrow$	para "entonces" o implicación condicional
$\leftrightarrow$	para "si y sólo si" o la bicondicional

Repasemos con un ejemplo. Proposiciones:

p = "está lloviendo"
q = "el sol está brillando"
r = "hay nubes en el cielo"

simbolizamos las siguientes frases:

Proposición	*Simbolización*
Está lloviendo y el sol está brillando	$p \wedge q$
Si está lloviendo, entonces hay nubes en el cielo	$p \rightarrow r$
Si no está lloviendo, entonces el sol no está brillando y hay nubes en el cielo	$\neg p \rightarrow (\neg q \wedge r)$
El sol está brillando si y sólo si no está lloviendo	$q \leftrightarrow \neg p$

La suposición fundamental del cálculo proposicional consiste en que los valores de verdad de una proposición construida a partir de otras proposiciones quedan completamente determinados por los valores de verdad de las proposiciones originales. Para ello se establecen los valores de verdad según las posibles combinaciones de valores de verdad de las proposiciones originales, basándonos en las siguientes tablas:

Negación ("no"):

p	¬ p
V	F
F	V

Evidentemente, si la proposición p es verdadera, su negación será falsa y viceversa.

Conjunción ("y"):

p	q	p ∧ q
V	V	V
V	F	F
F	V	F
F	F	F

La tabla indica que, el conectivo lógico "y" sólo será verdadero cuando ambas proposiciones p y q sean verdaderas.

Disyunción ("o"):

p	q	p ∨ q
V	V	V
V	F	V
F	V	V
F	F	F

La tabla indica que, si al menos una de las proposiciones es verdadera, la proposición formada por el conectivo "o" será verdadera.

Condicional ("entonces"):

p	q	p → q
V	V	V
V	F	F
F	V	V
F	F	V

Naturalmente, si el antecedente es verdadero y el consecuente es falso, la proposición formada por el conectivo "→" será falsa.

Bicondicional ("si y sólo si"):

p	q	p ↔ q
V	V	V
V	F	F
F	V	F
F	F	V

El bicondicional establece que, para que la proposición formada por el conectivo "↔" sea verdadera ambas proposiciones deben tener el mismo valor de verdad.

Observaciones importantes:

- Nótese que, el operador "¬" es unario (se utiliza con una única proposición), mientras que los demás operadores son binarios (utilizan dos proposiciones).
- Otra observación importante es que los operadores condicionales "→" y bicondicionales "↔" en realidad no son necesarios, ya que pueden ser reemplazados por combinaciones de "¬", "∧" y "∨ " a través de las siguientes equivalencias lógicas:

p → q	es equivalente a	(¬ p) ∨ q
p ↔ q	es equivalente a	(p → q) ∧ (q → p)

(donde los condicionales pueden reemplazarse por la equivalencia anterior)

Relacionemos este tema con la Informática:

La lógica utilizada en informática a bajo nivel admite sólo dos estados para cada unidad mínima de información: (1,0), (on, off) o (Verdadero, Falso). Podemos reemplazar los valores V y F de las tablas anteriores por los valores 1 y 0 respectivamente, para formar el álgebra booleana, consistente de dos operaciones binarias "∧" y "∨ ", y una operación unaria "¬".

De esta manera, reconstruimos las tablas de los conectivos lógicos considerando que las operaciones se efectúan a nivel de bits:

b1	b2	b1 AND b2
1	1	1
1	0	0
0	1	0
0	0	0

b1	b2	b1 OR b2
1	1	1
1	0	1
0	1	1
0	0	0

b1	NOT b1
1	0
0	1

Se puede observar que el conectivo lógico OR es inclusivo (la operación retorna 1 donde al menos uno de los operandos sea 1). También resulta de utilidad el conectivo lógico OR exclusivo (XOR), donde la operación retorna 1 en caso que uno de los operandos sea 1, pero no ambos. A continuación se observa la tabla del XOR:

b1	b2	b1 XOR b2
1	1	0
1	0	1
0	1	1
0	0	0

Operaciones comunes en lenguaje Assembler

Podemos categorizar las operaciones más comunes que realiza la unidad Aritmético/Lógica (ALU) en la siguiente tabla:

Desplazamientos:	Lógicos
	Circulares
	Aritméticos
Lógicas:	NOT
	AND
	OR
	XOR
Aritméticas:	Negación
	Suma
	Resta
	Multiplicación
	División

En este apunte nos centramos en las operaciones lógicas, las que por su propia naturaleza son a nivel de bit. Aunque se pueden realizar en forma paralela, no existe ninguna interacción entre bits de posiciones diferentes. Las operaciones lógicas sobre dos secuencias de bits realizan la operación sobre cada par de bits de igual posición entre ambas secuencias, de manera independiente.

Detallemos cómo funcionan las operaciones lógicas con grupos de bits mediante ejemplos:

	1101
AND	
	1011
	1001

	1001
OR	
	1011
	1011

	1101
XOR	
	1011
	0110

NOT 1011
0100

Observación: Estos ejemplos consideran para el AND, OR y XOR dos secuencias de cuatro bits c/u.

Se puede notar que la operación NOT equivale al complemento lógico (complemento a 1 -Ca1).

Implementación de máscaras

Dada una secuencia de bits, a veces resulta útil "jugar" con ciertas posiciones (ej: forzar los bits de posiciones impares a 1, o averiguar el estado del 3er bit en la secuencia, o invertir los valores de ciertas posiciones dejando intactas las demás, etc). Para esta tarea se pueden utilizar operaciones lógicas adecuadas que reciban como primer secuencia de bits la secuencia dada y, como segunda secuencia de bits, una secuencia predeterminada, denominada "máscara", que servirá como segundo operando para obtener el resultado deseado.

Tengamos en cuenta lo siguiente para definir el operador adecuado y la máscara:

- De la tabla del AND se deduce que:
  - Al utilizar un operando con valor 1, el resultado de la operación coincidirá con el valor del otro operando (1 AND 1 = 1, 1 AND 0 = 0). Genéricamente: 1 AND X = X.
  - Al utilizar un operando con valor 0, el resultado de la operación será 0 independientemente del valor del otro operando. (0 AND 0 = 0, 0 AND 1 =0). Genéricamente: 0 AND X = 0.
- De la tabla del OR se deduce que:
  - Al utilizar un operando con valor 1, el resultado de la operación será 1 independientemente del valor del otro operando (1 OR 0 = 1, 1 OR 1 = 1). Genéricamente: 1 OR X = 1.
  - En caso de realizar un OR entre 0 y cualquier operando, el resultado coincidirá con el valor de ese operando (0 OR 0 = 0, 0 OR 1 = 1). Genéricamente 0 OR X = X.
- De la tabla del XOR se deduce que:
  - Al utilizar como primer operando un 0, el resultado coincidirá con el valor del segundo operando (0 XOR 0 = 0 , 0 XOR 1 = 1). Genéricamente: 0 XOR X = X.
  - En caso de ralizar un XOR entre 1 y cualquier operando, el resultado será el opuesto del operando (1 XOR 0 = 1, 1 XOR 1 = 0). Genéricamente: 1 XOR X = NOT X.

A modo de ejemplo, se plantean los siguientes ejercicios:

Ej. 1) Dada una secuencia de 4 bits, forzar el 1er bit a cero, dejando el resto sin modificar.

Solución: Usamos el operador AND y como máscara definimos una que tenga el valor 0 en la primer posición (dado que 0 AND X = 0) y completamos la máscara con 1 (dado que 1 AND X = X).

```
        b3 b2 b1 b0
AND
         1  1  1  0    (máscara = 1110)
-------------------
        b3 b2 b1 0
```

Ej. 2) Dada una secuencia de 4 bits, invertir el valor de las posiciones impares, dejando el resto sin modificar.

Solución: Usamos el operador XOR y como máscara definimos una que tenga el valor 1 en las posiciones que queremos invertir (1 XOR X = NOT X) y un 0 en las posiciones que deben permanecer iguales (0 XOR X = X).

```
        b3 b2 b1 b0
XOR
         1  0  1  0    (máscara = 1010)
-------------------
       ¬b3 b2 ¬b1 b0
```

Ej. 3) Dada una secuencia de 4 bits, forzar a uno el segundo bit, dejando el resto sin modificar.

Solución: Usamos el operador OR y como máscara definimos una que tenga el valor 1 en la segunda posición (1 OR X = 1) y un 0 en las posiciones que deben permanecer iguales (0 OR X = X).

```
        b3 b2 b1 b0
OR
      0  0  1  0    (máscara = 0010)
      ---------
        b3 b2 1 b0
```

En ciertos casos, puede ser necesario pasar la secuencia de bits de entrada por mas de una máscara y operador:

Ej. 4) Dada una secuencia de 4 bits, invertir el valor de las posiciones impares y forzar a uno las posiciones pares.

Solución: En el 2do ejemplo observamos que el operador XOR resulta adecuado para invertir dígitos binarios utilizando como máscara un 1 en las posiciones que deseamos invertir y un 0 en las posiciones que deseamos queden intactas. Luego realizamos un OR sobre el resultado parcial para forzar a 1 las posiciones pares, utilizando así una segunda máscara con un 1 en las posiciones que deseamos forzar a 1 y un 0 en las posiciones que deseamos se mantengan intactas:

```
          b3 b2 b1 b0  (secuencia de entrada)
XOR
        1  0  1  0     (1er máscara = 1010)
      ---------------
      ¬b3 b2 ¬b1 b0    (resultado parcial, se invirtieron las posiciones impares)
OR
        0  1  0  1     (2da máscara = 0101)
      ---------------
      ¬b3 1  ¬b1 1     (resultado final)
```

Bibliografía consultada para elaborar este apunte

- "Matemáticas Discretas", Kenneth A. Ross, Charles R.B. Wright. Ed. Prentice Hall.
- "Arquitectura de Computadores", Pedro de Miguel, José Angulo. Ed. Paraninfo.
- "Fundamentos de los Computadores", Pedro de Miguel Anasagasti. Ed. Paraninfo.
- "Programmer's Technical Reference: The Processor and Coprocessor", Robert Hummel. Ed. ZD Press.

Ejercicios prácticos

Ej. 1) ¿Cuál es el resultado de las siguientes operaciones?.

1101 AND 0111 = ...
0101 OR 1001 = ...
NOT 0100 = ...
1011 XOR 1110 = ...
(((1010 AND 1100) OR 0101) XOR 1100) = ...

Ej. 2) Completar los bits *X*. Aclaración: puede haber más de una combinación posible.

1001 AND *XXXX* = 1000
0110 OR *XXXX* = 1110
1010 XOR *XXXX* = 1010
NOT *XXXX* = 0110

Ej. 3) ¿Tienen solución las siguientes operaciones?. ¿Por qué?.

1011 OR *XXXX* = 0001
0101 AND *XXXX* = 1100

0011 XOR *XXXX* = 0000

Ej. 4) Dada una secuencia de 4 bits (b_3 b_2 b_1 b_0), encuentre las máscaras apropiadas para:

- Poner en 0 el bit mas significativo, dejando el resto sin modificar.
- Poner en 1 las posiciones impares, dejando el resto sin modificar.
- Invertir todos los bits.
- Invertir las posiciones pares, dejando el resto sin modificar.
- Poner en 1 el bit b_0 y en 0 el bit b_3,dejando el resto sin modificar.
- Invertir las posiciones impares y forzar a 0 las posiciones pares.

Ej. 5) Complete con el operador adecuado (AND, OR, XOR, NOT) las siguientes operaciones:

1000 ..... 1011 = 1000
0110 ..... 1000 = 1110
1101 ..... 1001 = 0100
1111 ..... 0011 = 1100

Organización de Computadoras 2004

Apunte 4: Lenguaje Assembly

Introducción

Las computadoras son máquinas diseñadas para ejecutar las instrucciones que se le indican, de manera de resolver problemas o hacer algún otro tipo de cosas. Dichas instrucciones básicas no suelen ser más complicadas que realizar una suma o resta, una comparación, leer un dato o escribir un resultado en la memoria. Además, lo normal es que cada instrucción esté almacenada mediante un código en la memoria, de forma que la computadora "ve" esos números y los entiende como instrucciones. También, estos códigos son específicos de cada tipo de CPU. A este conjunto de instrucciones codificadas se le llama lenguaje de máquina y es el único lenguaje que una computadora entiende.

Ahora bien, como programador, no resulta reconfortante pensar y escribir programas como una secuencia de números, de la manera en que la computadora los quiere ver, sino que es natural verlos como una secuencia de instrucciones. Para ello, se define un lenguaje, en principio, que es un mapeo directo de esos códigos a instrucciones comprensibles por un humano. A este lenguaje se lo denomina ***assembly*** o ***lenguaje de ensamble*** y al programa encargado de tomar programas escritos en assembly y generar los códigos que la computadora entenderá se lo denomina ***assembler*** o ***ensamblador***. Como veremos, no todas las instrucciones de assembly se corresponden directamente con instrucciones de lenguaje de máquina, sino que varias son instrucciones para el ensamblador mismo.

El Simulador

Antes de comenzar con algunos ejemplos de programas escritos en assembly, veremos una breve descripción de un programa cuyo propósito es simular una CPU y mostrar los pormenores de su funcionamiento. Este programa se llama **MSX88** y simula una computadora basada en una versión simplificada del célebre procesador 8086 de Intel, denominada **SX88**. Gráficamente se muestran los flujos de información existentes entre los diversos elementos que lo componen. Utiliza como plataforma de sistema operativo DOS, pudiéndose utilizar desde una ventana DOS bajo Windows. En la Figura 1 se muestra la pantalla inicial.

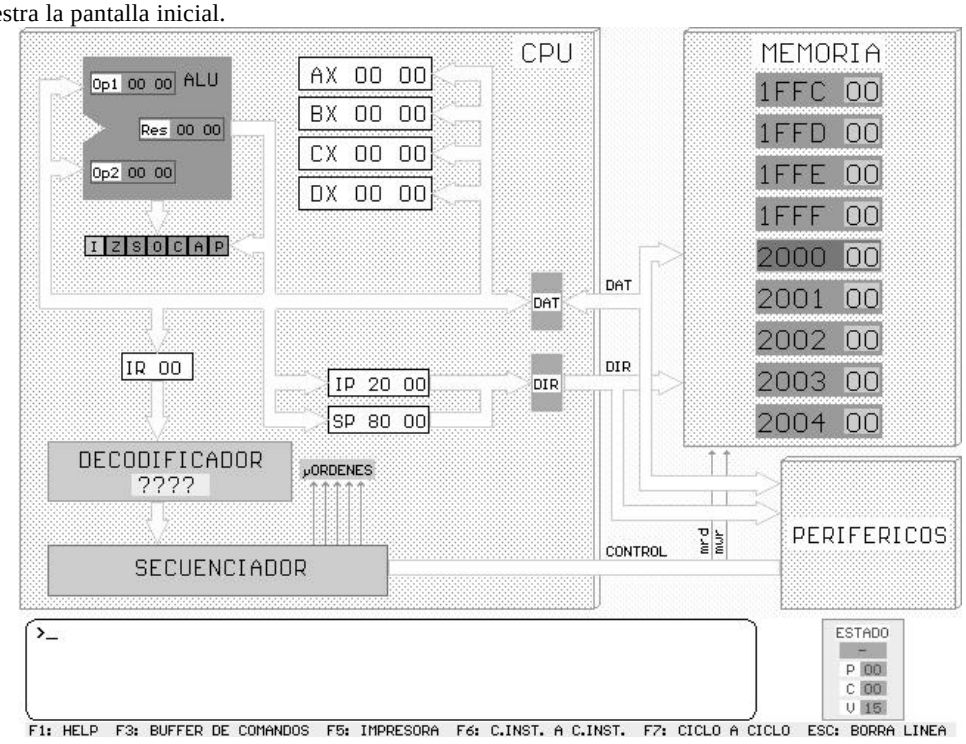


Figura 1

En la Figura 1 podemos distinguir los siguientes bloques:

- CPU
  - ALU
  - Registros AX, BX, CX, DX, SP, IP, IR.
  - Registro de Flags
  - Decodificador
  - Secuenciador
- Memoria principal
- Periféricos
- Bus de datos y de direcciones
- Toma de entrada de comandos

La CPU es la encargada de ejecutar un programa contenido en la memoria instrucción por instrucción.
La ALU es la encargada de ejecutar las operaciones aritméticas y lógicas entre los registros temporarios Op1 y Op2, dejando el resultado en Res. Dichas operaciones serán las dispuestas por la instrucción en ejecución
Los registros AX, BX, CX y DX son de uso general, 16 bits de longitud, se pueden dividir en 2 partes de 8 bits cada uno. Ejemplo: AX en AH y AL.
El registro IP (*instrucción pointer*) contiene la dirección de memoria de la próxima instrucción a ser ejecutada.
El registro SP (*stack pointer*) contiene la dirección de memoria del tope de la pila.
El registro de flags nos mostrará el estado de las banderas o flags luego de cada operación. Son 8 bits que indican el estado de las correspondientes 8 banderas. De estas 8 se utilizaran en la práctica las correspondientes a:

- Bandera de cero: identificada por la letra Z.
- Bandera de overflow: identificada por la letra O.
- Bandera de carry/borrow: identificada por la letra C.
- Bandera de signo del número: identificada por la letra S.

La entrada de comandos es el lugar donde se introducirán los comandos del simulador una vez iniciada la aplicación.
El bloque de periféricos engloba la entrada-salida del simulador.

Modo de uso

Par tener una primera toma de contacto con el programa se recomienda arrancar la demostración incluida en el archivo de distribución de MSX88. Asegúrese que los archivos DEMOMEMO.EJE, DEMOPIO.EJE y DEMODMA.EJE estén en el mismo directorio que DEMO.EXE. Se supone que todos los archivos del MSX88 se encuentran en el directorio SIMULADOR dentro del directorio raíz del disco C. Para arrancar la demo se debe teclear:

```
C:\SIMULADOR> DEMO <Enter>
```

Este programa ofrece una introducción muy detallada a las características principales de MSX88.
Ejecutando el programa DEMO, puede finalizar la ejecución pulsando varias veces la tecla 2.
Para arrancar el programa MSX88 ejecute el comando DOS:

```
C:\SIMULADOR> MSX88 <Enter>
```

Aparecerá la pantalla de la Figura 1. En la parte inferior, solo aparecerá el prompt (>), en donde se introducirán los comandos. MSX88 tiene un programa monitor que controla todas las partes de la máquina virtual. El primer comando que se debe conocer es, como en la mayoría de los sistemas operativos, el de help. Para invocar al help del monitor puede pulsar la tecla F1, teclear "help", o simplemente "h" que aparecerán escritos en la ventana de comandos del monitor.

Elaboración y ejecución de un programa de prueba

Primero se deberá editar el programa, escrito en el lenguaje de ensamble de MSX88 en un editor de texto como el EDIT o el NOTEPAD. En caso de utilizar el NOTEPAD, luego de la última línea del programa (END) se debe teclear un Enter, si no dará error al ensamblarlo. El programa fuente así generado se guardara con extensión .asm (prueba.asm por ejemplo). Se deberá usar una sintaxis determinada.
Luego habrá que ensamblarlo con el ensamblador del entorno: el ASM88. Esto se realiza con el comando

```
C:\SIMULADOR> ASM88 prueba.asm <Enter>
```

ASM88 producirá dos archivos: objeto con extensión ".o" (prueba.o) y, opcionalmente un archivo listado con extensión ".LST" (prueba.lst). Este proceso podrá dar errores en caso de que el programa fuente los tenga, por lo que habrá que editarlo nuevamente, corregirlos, volver a ensamblar, y así hasta que se eliminen los mismos.
Cuando haya conseguido ensamblar sin errores deberá utilizar el programa enlazador para generar el programa ejecutable de extensión ".EJE". LINK88 no es realmente un enlazador, dado que no presenta funcionalidades de tal, ya que no enlaza otros programas objeto al que tenemos, pero se ha considerado interesante que el alumno vaya adquiriendo unos hábitos que le ayuden en su trabajo con las herramientas profesionales que si cuentan con él. Esto se realiza con el comando

```
C:\SIMULADOR> LINK88 prueba.o <Enter>
```

Si no se cometieron errores, debemos tener en el mismo directorio el archivo prueba.eje. Para cargarlo en la memoria de MSX88 se ha de ejecutar el siguiente comando dentro del área de comandos del MSX88 (ojo, no es un comando de DOS):

```
> L prueba <Enter>
```

Por último, para ejecutar el programa, se utiliza el comando

```
> G <Enter>
```

Podremos observar en la pantalla como se obtienen instrucciones, se las decodifican, como se producen movimientos de datos, operaciones aritméticas, etc. Si se desea volver a ejecutar el programa, se deberá modificar el registro IP para que apunte a la dirección de comienzo del programa con el comando

```
> R IP 2000 <Enter>
```

que coloca en el registro IP (dirección de la próxima instrucción) la dirección de la celda donde está la primer instrucción del programa (por defecto, 2000h en el simulador).
Éstos y otros comandos más se encuentran detallados en el manual del simulador.

Directivas para el ensamblador

Como se había mencionado en un principio, un programa en assembly no está comprendido únicamente por instrucciones del lenguaje de máquina sino que existen otro tipo de instrucciones que le indican al ensamblador como realizar algunas tareas.
Un hecho notable cuando se trata del uso de variables es que en las instrucciones del lenguaje de máquina no se hace referencia a ellas por un nombre sino por su dirección en la memoria. Para evitarnos la muy poco grata tarea de recordar que tal variable está almacenada en una dirección particular (que, de hecho, podría cambiar a lo largo del desarrollo de un programa, forzándonos a revisar el programa entero para realizar los ajustes necesarios), en assembly existen instrucciones para definir variables. La sintaxis para definir una variable es la siguiente:

```
nombre_variable  especificador_tipo  valor_inicial
```

El nombre de la variable debe comenzar con una letra o un underscore (_) seguido por números, letras o underscores. El tipo indica el tamaño que ocupa en memoria dicha variable y puede ser alguno de los enumerados en la Tabla 1. El valor inicial puede no especificarse, usando el carácter '?', lo que le informa al ensamblador que dicho valor será el que se encuentre en la memoria del simulador en el momento de cargar el programa.

Especificador	Tipo	Tamaño
DB	Byte	8 bits
DW	Word	16 bits

Tabla 1: Especificadores de los tipos de variables.

Ejemplos de definición de variables:

```
Var1 DB 10
Var2 DW 0A000h
```

Podemos observar que los valores numéricos se interpretan en decimal, a menos que terminen con una letra 'h', que en cuyo caso se interpretarán como valores en hexadecimal. Además, como los números deben comenzar con un dígito decimal, en el caso del A000h, se antepone un cero para evitar que se la confunda con una variable llamada A000h.

A veces resulta bueno poder definir valores constantes. Esto se hace del siguiente modo:

```
nombre  EQU  valor
```

Debe notarse que en este caso el ensamblador reemplazará cualquier ocurrencia de 'nombre' por el valor indicado, pero que dicho valor no va a ocupar ninguna dirección de memoria, como lo hacen las variables.

Es posible extender la definición de variables para incluir la idea de tablas, de la siguiente manera:

```
nombre_variable  especificador_tipo  valores
```

En la definición anterior, *valores* es una lista de datos del mismo tipo de la variable separados por coma:

```
tabla DB  1,  2,  4,  8, 16, 32, 64, 128
```

Esto genera una tabla con los ocho valores especificados, uno a continuación del otro. Esto se puede ver como un arreglo de ocho bytes pero en el que se inicializaron sus celdas con dichos valores. Por otro lado, si quisiéramos definir algo equivalente a un string, podemos aplicar la misma idea de la tabla anterior, en donde en cada celda se almacenaría cada carácter del string. Sin embargo, escribir los códigos ASCII de cada carácter no simplifica mucho las cosas, así que existe una sintaxis alternativa:

```
string DB "Esto es un String."
```

Si quisiéramos definir una tabla en la que los datos que contienen son iguales o cumplen algún patrón repetitivo, es posible utilizar el modificador DUP en la lista de valores, de la siguiente manera:

```
cantidad DUP (valores)
```

En este caso, *cantidad* indica la cantidad de veces que se repiten el o los valores indicados entre paréntesis. Por ejemplo, para definir un arreglo de 20 palabras, inicialmente conteniendo 1234h y 4321h alternadamente, se le indica al ensamblador lo siguiente:

```
cantidad EQU 10
arreglo  DW cantidad DUP (1234h, 4321h)
```

Otra cuestión que se obvió hasta el momento es que dirección se le asigna a cada variable. El ensamblador lleva cuenta de las variables e instrucciones que va procesador y de la dirección que le corresponde a cada una, dependiendo de una dirección inicial y del tamaño correspondiente. Existe una directiva del ensamblador, llamada ORG, que permite cambiar sobre la marcha la dirección a partir de la cual se colocarán las cosas que estén a continuación de la misma. La sintaxis de la misma es:

```
ORG dirección
```

El uso de ORG se ejemplifica a continuación:

```
ORG 1000h

      contador DW 1234h
      cantidad DB 0

ORG 2000h

      arreglo DB 0A0h, 3 DUP (15)

      cadena  DB
"Un string es un
arreglo de
bytes."

END
```

1000h	contador	34h
1001h	contador	12h

1002h	cantidad	00h
2000h	arreglo	A0h
2001h	arreglo	0Fh
2002h	arreglo	0Fh
2003h	arreglo	0Fh
2004h	cadena	U
2005h	cadena	n

2006h	cadena	
2007h	cadena	s
2008h	cadena	t

En el ejemplo anterior, el primer ORG le indicará al ensamblador que todo lo que esté a continuación deberá ubicarse a partir de la dirección 1000h. Por eso, el contenido de la variable "contador" estará almacenado en las direcciones 1000h y 1001h, pues es un valor de 16 bits. A continuación de "contador", se almacenará "cantidad", a la que le toca la dirección 1002h.
El siguiente ORG indica que lo que se encuentre a continuación de él, será ubicado a partir de la dirección 2000h. Por ello, "arreglo" se almacena desde 2000h hasta 2003h y "cadena" comienza en la dirección 2004h. En la tabla a la derecha del ejemplo podemos ver como se ubican las variables definidas en la memoria: "arreglo" ocupa 4 bytes, uno por el primer valor de la lista y tres más por duplicar tres veces al valor entre paréntesis; "cadena" ocupa tantos bytes como caracteres contenga el string.

Hasta aquí vimos como definir constantes y variables de distintos tipos y como indicarle al ensamblador en que dirección ubicarlas. Ahora veremos como escribir las instrucciones que darán forma al programa.

Instrucciones

Recordemos que el procesador que utiliza el simulador MSX88, llamado SX88, cuenta con varios registros de propósito general (AX, BX, CX y DX), a los que podemos tratar como registros de 16 bits o como un par de registros de 8 bits, tomando la parte baja separada de la parte alta. En el caso de AX, tendríamos AH (parte alta o *high* de AX) y AL (parte baja o *low* de AX), a BX como BH y BL, etc. Además existe un registro llamado IP, que contiene la dirección de la próxima instrucción a ejecutar, y otro llamado SP, que contiene la dirección del tope de la pila.

Instrucciones de transferencia de datos

La primera instrucción que veremos es la que permite realizar movimientos de datos, ya sea entre registros o desde un registro a la memoria y viceversa. Debe notarse que no es posible realizar movimientos de datos desde una parte a otra de la memoria mediante una única instrucción. La instrucción en cuestión se llama MOV (abreviatura de MOVE, *mover* en inglés) y tiene dos operandos:

```
MOV destino, origen
```

El valor contenido en *origen* será asignado a *destino*. La única restricción es que tanto *origen* como *destino* sean del mismo tipo de datos: bytes, words, etc. Ejemplo:

```
ORG 1000h

      var_byte DB 20h
      var_word DW ?

ORG 2000h

      MOV AX, 1000h
      MOV BX, AX
      MOV BL, var_byte
      MOV var_word, BX

END
```

Instante	AX		BX	
	AH	**AL**	**BH**	**BL**
0	00	00	00	00
1	**10**	**00**	00	00
2	10	00	**10**	**00**
3	10	00	10	**20**
4	10	00	10	20

Aquí vemos el uso más común de la directiva ORGL: La idea es separar las variables del programa. En el ejemplo, las variables serán almacenadas a partir de la dirección 1000h mientras que las instrucciones del programa estarán a partir de la dirección 2000h. Además, como el simulador, por defecto, comienza ejecutando lo que está contenido en la dirección 2000h, eso hará que nuestro programa comience en la primera instrucción MOV.

Dicha instrucción asigna el valor inmediato 1000h al registro AX. Esta instrucción emplea el modo de direccionamiento conocido como "inmediato". Como ambos son valores de 16 bits, no hay inconveniente en esa asignación. En la tabla a la derecha del ejemplo se muestran el contenido de los registros AX y BX. Inicialmente, ambos estaban en cero (instante 0) y luego de la primer instrucción MOV, el registro AX cambia (instante 1, resaltado el cambio en negrita).

El siguiente MOV asigna el contenido del registro AX al registro BX (instante 2). De nuevo, como ambos son de 16 bits, es una asignación válida. El modo de direccionamiento que usa es el denominado "registro".
El tercer MOV asigna el contenido de la variable "var_byte" (que es 20h) al registro BL. Como BL es la parte baja de BX, el cambio que se produce en BX es el indicado en la tabla: pasa de 1000h a 1020h (instante 3). Como BL y "var_byte" son ambos de 8 bits, es una asignación permitida. Además, se emplea el modo de direccionamiento "directo".

El último MOV asigna el valor contenido en el registro BX a la dirección de memoria a la que "var_word" hace referencia. Ahora, "var_word" contiene el valor 1020h.

Mover datos de un lugar a otro de la memoria resulta poco útil si no podemos trabajar sobre ellos. Si recordamos que la CPU cuenta con una ALU capaz de realizar operaciones aritméticas y lógicas, deben existir instrucciones para utilizarlas, las cuales se detallarán a continuación.

Instrucciones aritméticas

Existen dos operaciones aritméticas básicas que puede realizar la CPU: sumar y restar. Para cada una de ellas existen instrucciones correspondientes: ADD (*sumar* en inglés) y SUB (*restar* en inglés). El formato de estas instrucciones es el siguiente:

```
ADD operando1, operando2
SUB operando1, operando2
```

En principio, parecería que falta algo, ya que se indican con que valores se va a operar pero no en donde va a quedar almacenado el resultado de la operación. En realidad, esto está implícito en la instrucción, puesto que el resultado se almacenará en *operando1* (recordar la idea de máquina de dos direcciones). En otras palabras, estas instrucciones hacen lo siguiente (los paréntesis indican *"el contenido de"*):

```
ADD: (operando1) ← (operando1) + (operando2)
SUB: (operando1) ← (operando1) - (operando2)
```

Esto implica que el valor contenido en operando1 es reemplazado por el resultado de la operación, por lo que si dicho valor se utilizará más tarde, es necesario operar sobre una copia del mismo. Por ejemplo:

```
ORG 1000h
      dato1     DW 10
      dato2     DW 20
      resultado DW ?

ORG 2000h
      MOV AX, dato1
      ADD AX, dato2
      MOV resultado, AX
END
```

En el ejemplo anterior, se utiliza el registro AX como variable auxiliar, de manera de no afectar a *dato1*.

Existen dos instrucciones más similares a las anteriores pero que tienen en cuenta al flag de *carry/borrow*:

```
ADC operando1, operando2
SBB operando1, operando2
```

La semántica de dichas instrucciones es la siguiente:

```
ADC: (operando1) ← (operando1) + (operando2) + (C)
SBB: (operando1) ← (operando1) - (operando2) - (C)
```

C es el valor del flag de *carry/borrow*, que puede ser 0 o 1. En otras palabras, estas instrucciones suman o restan incluyendo a dicho flag, por lo que resultan útiles para encadenar varias operaciones de suma.
Supongamos que queremos sumar valores de 32 bits. Dado que nuestra CPU opera con valores de 8 o 16 bits, no sería posible hacerlo en un solo paso. Sin embargo, podríamos sumar la parte baja (los 16 bits menos significativos) por un lado y la parte alta (los 16 bits más significativos) por otro usando dos instrucciones ADD. El problema se presenta cuando se produce un acarreo al realizar la suma en la parte baja, ya que no podemos simplemente ignorarlo pues el resultado no sería el correcto, como se muestra en este ejemplo:

```
  Correcto:          Incorrecto:
  0015 FFFF            0015   FFFF
+ 0002 0011          + 0002 + 0011
  0018 0010            0017   1 0010   ———— 0017 0010
```

Para resolver ese problema, utilizamos estas instrucciones que acabamos de ver de la siguiente manera:

```
ORG 1000h
      dato1_l   DW 0FFFFh
      dato1_h   DW  0015h
      dato2_l   DW  0011h
      dato2_h   DW  0002h

ORG 2000h
      MOV AX, dato1_l
      ADD AX, dato2_l
      MOV BX, dato1_h
      ADC BX, dato2_h
END
```

Instante	AX	BX	Flag C
0	0000	0000	0
1	**FFFF**	0000	0
2	**0010**	0000	**1**
3	0010	**0015**	1
4	0010	**0018**	**0**

Nótese que la segunda suma es realizada usando un ADC y no un ADD. De esta manera, si en el ADD se produce un acarreo, éste es sumado junto a *dato1_h* y *dato2_h* durante el ADC, produciendo el resultado correcto. Esta misma idea puede aplicarse a la resta usando SUB y SBB.

Existen un par de instrucciones que son casos particulares del ADD y del SUB, llamadas INC y DEC, que poseen un solo operando y simplemente le suman o restan 1 a dicho operando.

```
INC operando
DEC operando

INC: (operando) ← (operando) + 1
DEC: (operando) ← (operando) - 1
```

Estas instrucciones existen porque ocupan menos bits en memoria que su contrapartida empleando ADD o SUB, por lo que suelen ejecutarse más velozmente.

Finalmente, las mismas restricciones que se imponen al MOV se aplican a estas instrucciones: Los operandos deber ser del mismo tipo y no es posible hacer referencias a memoria en los dos operandos al mismo tiempo; siempre deben moverse datos entre registros o entre un registro y la memoria.

Instrucciones lógicas

Las operaciones lógicas que posee la CPU del simulador son las siguientes:

```
AND operando1, operando2
OR  operando1, operando2
XOR operando1, operando2
NOT operando
NEG operando
```

Al igual que el ADD y el SUB, el resultado de las instrucciones AND, OR y XOR queda almacenado en el primer operando. Las primeras tres instrucciones realizan la operación lógica correspondiente aplicada bit a bit sobre todos los bits de ambos operandos. NOT y NEG calculan sobre el operando el complemento a 1 y a 2 respectivamente.

Instrucciones de comparación

Esta CPU cuenta con una instrucción de comparación llamada CMP (abreviatura de COMPARE, *comparar* en inglés). CMP es esencialmente equivalente en funcionamiento a SUB, con excepción de que el resultado de la resta no se almacena en ninguna parte, por lo que ninguno de los operandos se ve alterado. Esta instrucción da la impresión de que no hace nada, pero en realidad, al hacer la resta, causa que se modifiquen los flags. Ver como quedaron los flags luego de una resta nos dice muchas cosas sobre los números que se restaron. Por ejemplo, si el flag Z queda en 1, el resultado de la resta fue cero, con lo que podemos concluir que los dos números que se restaron eran iguales. Si, en cambio, el flag S (signo) quedó en 1 y vemos a los operandos como números en CA2, podemos inferir que el segundo operando era mayor que el primero, puesto que al restarlos nos da negativo.
Existen combinaciones de los flags que nos indican si un número es mayor, mayor o igual, menor, menor o igual, igual o distinto a otro. Estas combinaciones pueden depender del sistema en el que interpretamos a los números, normalmente, en BSS o CA2.

Una instrucción CMP por si sola es bastante inútil pero adquiere su poder al combinarla con instrucciones de salto condicionales, que se verán a continuación.

Instrucciones de salto

Las instrucciones de salto, como indica su nombre, permiten realizar saltos alterando el flujo de control a lo largo de la ejecución de un programa. Estas instrucciones tienen un operando que indica la dirección que se le asignará al registro IP. Al alterar este registro, se modifica cual va a ser la próxima instrucción que va a ejecutar la CPU. Por otro lado, tenemos dos tipos de instrucciones de salto: los saltos incondicionales y los saltos condicionales. Los primeros se producen siempre que se ejecuta la instrucción mientras que los segundos dependen de alguna condición para que se produzca o no dicho salto.
A continuación se detallan las instrucciones de salto que el SX88 posee, junto con la condición por la cual se realiza el salto:

```
      JMP dirección     ; Salta siempre

      JZ  dirección     ; Salta si el flag Z=1
      JNZ dirección     ; Salta si el flag Z=0
      JS  dirección     ; Salta si el flag S=1
      JNS dirección     ; Salta si el flag S=0
      JC  dirección     ; Salta si el flag C=1
      JNC dirección     ; Salta si el flag C=0
      JO  dirección     ; Salta si el flag O=1
      JNO dirección     ; Salta si el flag O=0
```

Al igual que lo que ocurría con las variables, el ensamblador nos facilita indicar la dirección a la que se quiere saltar mediante el uso de una etiqueta, la cual se define como un nombre de identificador válido (con las mismas características que los nombres de las variables) seguido de dos puntos, como se muestra en este ejemplo:

```
ORG 2000h
      MOV AX, 10

Lazo: ... ;
      ... ; <Instrucciones a repetir>
      ... ;

      DEC AX
      JNZ Lazo
```

```
Fin:  JMP Fin

END
```

En el ejemplo podemos ver que se definen dos etiquetas: *Lazo* y *Fin*. También se ve en acción a un par de instrucciones de salto: JMP y JNZ. El programa inicializa AX en 10, hace lo que tenga que hacer dentro del bucle, decrementa AX en 1 y salta a la instrucción apuntada por la etiqueta *Lazo* (el DEC) siempre y cuando el flag Z quede en 0, o sea, que el resultado de la operación anterior no haya dado como resultado 0. Como AX se decrementa en cada iteración, llegará un momento en que AX será 0, por lo que el salto condicional no se producirá y continuará la ejecución del programa en la siguiente instrucción luego de dicho salto. La siguiente instrucción es un salto incondicional a la etiqueta *Fin*, que casualmente apunta a esa misma instrucción, por lo que la CPU entrará en un ciclo infinito, evitando que continúe la ejecución del programa. Nótese que sin esta precaución la CPU continuará ejecutando lo que hubiera en la memoria a continuación del fin del programa. La palabra END al final del programa le indica al ensamblador que ahí finaliza el código que tiene que compilar, pero eso no le dice nada a la CPU. Más adelante, veremos como subsanar este inconveniente.
El ejemplo anterior plantea un esquema básico de iteración usado comúnmente como algo equivalente a un "`for i := N downto 1`" del pascal.

Otras instrucciones

Algo poco elegante en el ejemplo del FOR es hacer que la CPU entre en un bucle infinito para detener la ejecución del programa. Existe una manera más elegante de hacer esto y es pedirle gentilmente a la CPU que detenga la ejecución de instrucciones mediante la instrucción HLT (HALT, *detener* en inglés).
Existe otra instrucción que no hace nada. Si, es correcto, no hace nada, simplemente ocupa memoria y es decodificada de la misma manera que cualquier otra instrucción, pero el efecto al momento de ejecutarla es no hacer nada. Dicha instrucción es NOP.

Cabe mencionar que el SX88 cuenta con otras instrucciones que no se detallan en este apunte, ya que escapan a los contenidos propuestos para este curso. Sin embargo, el lector interesado puede recurrir al manual del simulador MSX88, en donde se enumeran todas las instrucciones que dicho simulador soporta.

Modo de direccionamiento indirecto

Existe un modo de direccionamiento que aún no se mencionó y que facilita el recorrido de tablas. Estamos hablando del modo de direccionamiento indirecto vía el registro BX y se ilustra el uso de éste en el siguiente ejemplo:

```
ORG 1000h
      tabla     DB 1, 2, 3, 4, 5                    ; (1)
      fin_tabla DB ?                                ; (2)
      resultado DB 0                                ; (3)

ORG 2000h
      MOV BX, OFFSET tabla                          ; (4)
      MOV CL, OFFSET fin_tabla – OFFSET tabla       ; (5)

Loop: MOV AL, [BX]                                  ; (6)
      INC BX                                        ; (7)
      XOR resultado, AL                             ; (8)
      DEC CL                                        ; (9)
      JNZ Loop                                      ; (10)

      HLT                                           ; (11)
END
```

En este ejemplo se introducen varias cosas además del modo de direccionamiento indirecto.
El ejemplo comienza definiendo una tabla (1) y dos variables más (2) y (3).
Luego, en (4), comienza inicializando BX con "*OFFSET tabla*". Esto indica que se debe cargar en BX la dirección de *tabla*, no el contenido de dicha variable.

En (5) se asigna a CL la diferencia entre la dirección de *tabla* y la dirección de *fin_tabla*. Si meditamos un segundo sobre esto, veremos que lo que se logra es calcular cuantos bytes hay entre el comienzo y el final de la tabla. De esta manera, obtenemos la cantidad de datos que contiene la tabla.
En (6) vemos que en el MOV aparece BX entre corchetes. Esto significa que se debe asignar en AL el contenido de la celda de memoria cuya dirección es el valor contenido en BX. Así, como BX se había inicializado con la dirección del comienzo de la tabla, esto causa que se cargue en AL el contenido de la primer entrada de la tabla.
En (7) se incrementa BX, con lo que ahora apunta al siguiente byte de la tabla.
En (8) se calcula una operación XOR con el contenido de la variable *resultado* y el byte que se acaba de obtener en AL, dejando el resultado de esa operación en la misma variable.
(9) y (10) se encargan de iterar saltando a la etiqueta Loop mientras CL sea distinto de 0, cosa que ocurrirá mientras no se llegue al final de la tabla. Cuando esto ocurra, no se producirá el salto condicional y la ejecución seguirá en la próxima instrucción a continuación de esta. Dicha instrucción (11) es un HLT que detiene la CPU.

Ejemplos

A continuación veremos una serie de ejemplos que ilustran el uso de las instrucciones que se vieron a lo largo del apunte y también como codificar las estructuras de control que son comunes en los lenguajes de alto nivel pero que no existen en assembly.

Selección: IF THEN ELSE

No existe una instrucción en assembly que sea capaz de hacer lo que hace la estructura IF-THEN-ELSE de pascal. Sin embargo, es posible emularla mediante la combinación de instrucciones CMP y saltos condicionales e incondicionales. Por ejemplo, intentemos simular el siguiente código de pascal:

```
IF AL = 4 THEN
BEGIN
  BL = 1;
  CL = CL + 1;
END;
```

La idea es comenzar calculando la condición del IF, en este caso, comparar AL con 4. Eso se logra con una instrucción CMP:

```
CMP AL, 4
```

Esta instrucción alterará los flags y en particular, nos interesa ver al flag Z, ya que si dicho flag está en 1, implica que al resta AL con 4, el resultado dio 0, por lo que AL tiene que valer 4. Entonces, si esa condición es verdadera, deberíamos ejecutar las instrucciones que están dentro del THEN. Si no, deberíamos evitar ejecutarlas. Una solución simplista es usar saltos, del siguiente modo:

```
        CMP AL, 4     ; (1)
        JZ Then       ; (2)
        JMP Fin_IF    ; (3)

Then:   MOV BL, 1     ; (4)
        INC CL        ; (5)

Fin_IF: HLT           ; (6)
```

Analizando el código, vemos lo siguiente:
Si la comparación en (1) establece el flag Z en 1, el salto en (2) se produce, haciendo que la ejecución continúe en la etiqueta *Then*. Ahí, se ejecutan las instrucciones (4) y (5) que hacen lo que se encuentra en el THEN del IF y continúa la ejecución en la instrucción apuntada por la etiqueta *Fin_IF*.
Si el flag Z quedó en 0 en (1), el salto en (2) no se produce, por lo que la ejecución continúa en la próxima instrucción, el JMP en (3), que saltea las instrucciones y continúa la ejecución en la instrucción apuntada por la etiqueta *Fin_IF*, que señala el final del IF.
En el final del IF, se ejecuta un HLT para terminar la ejecución del programa.

El ejemplo anterior se puede mejorar un poco más. El par JZ/JMP en (2) y (3) pueden reemplazarse por un "`JNZ Fin_IF`", ya que si no se cumple la condición, se omite la parte del THEN y continúa la ejecución al final del IF. Estas "optimizaciones" contribuyen a un código más compacto y, en consecuencia, más eficiente. Sin embargo, como toda optimización, atenta contra la claridad del código.

```
        CMP AL, 4     ; (1)
        JNZ Fin_IF    ; (2)

Then:   MOV BL, 1     ; (3)
        INC CL        ; (4)

Fin_IF: HLT           ; (5)
```

El ejemplo fue planteado de esa manera, con instrucciones aparentemente redundantes, porque así como está, es más sencillo extenderlo para codificar una estructura IF-THEN-ELSE completa. Por ejemplo, si queremos simular el siguiente código de pascal:

```
      IF AL = 4 THEN
        BEGIN
          BL = 1;
          CL = CL + 1;
        END
      ELSE
        BEGIN
          BL = 2;
          CL = CL – 1;
        END;
```

Como ahora tenemos las dos alternativas, el código equivalente en assembly podría ser así:

```
        CMP AL, 4     ; (1)
        JZ Then       ; (2)
        JMP Else      ; (3)

Then:   MOV BL, 1     ; (4)
        INC CL        ; (5)
        JMP Fin_IF    ; (6)

Else:   MOV BL, 2     ; (7)
        DEC CL        ; (8)

Fin_IF: HLT           ; (9)
```

La cuestión ahora es que en (2) y (3) se decide que parte del IF se ejecutará, el THEN o el ELSE. Además, es necesario un salto en (6) para que una vez finalizada la ejecución del THEN, se sigua con la próxima instrucción luego del fin del IF. El resto es equivalente al ejemplo anterior.

Es posible eliminar el salto JMP (3) intercambiando las partes del THEN y del ELSE, como se ilustra a continuación:

```
        CMP AL, 4     ; (1)
        JZ Then       ; (2)

Else:   MOV BL, 2     ; (3)
        DEC CL        ; (4)
        JMP Fin_IF    ; (5)

Then:   MOV BL, 1     ; (6)
        INC CL        ; (7)

Fin_IF: HLT           ; (8)
```

Este último ejemplo muestra la forma más compacta de generar la estructura IF-THEN-ELSE mediante instrucciones de assembly.

Iteración: FOR, WHILE, REPEAT-UNTIL

Ya vimos como se codifica en assembly una estructura IF-THEN-ELSE. Ahora veremos como implementar un FOR, un WHILE o un REPEAT-UNTIL. En el caso del primero, ya algo se comentó cuando se describieron los saltos condicionales. La idea simplemente es realizar un salto condicional al inicio del código a repetir mientras no se haya alcanzado el límite de iteraciones. Por ejemplo:

```
AL := 0;
FOR CL := 1 TO 10 DO
  AL := AL + AL;
```

Se puede implementar mediante el siguiente esquema:

```
        MOV AL, 0
        MOV CL, 1
Iterar: CMP CL, 10
        JZ Fin
        ADD AL, AL
        INC CL
        JMP Iterar
Fin:    HLT
```

Si quisiéramos hacer un *downto* en lugar de un *to*, simplemente se realizan los siguientes cambios en el código:

```
        MOV AL, 0
        MOV CL, 10
Iterar: CMP CL, 1
        JZ Fin
        ADD AL, AL
        DEC CL
        JMP Iterar
Fin:    HLT
```

Implementar un WHILE es exactamente igual al esquema anterior, solo que en lugar de evaluar si el contador llegó al valor límite, lo que se hace es evaluar la condición del WHILE. Si quisiéramos ver un REPEAT-UNTIL, la diferencia es que en éste, la condición se evalúa luego de ejecutar el código a repetir al menos una vez. Esto lo logramos simplemente cambiando de lugar algunas cosas del ejemplo anterior:

```
        MOV AL, 0
        MOV CL, 10
Iterar: ADD AL, AL
        DEC CL
        CMP CL, 1
        JNZ Iterar
Fin:    HLT
```

Resumiendo, todas las estructuras de iteración se resuelven de la misma manera. Lo que varía es donde y que condición se evalúa.

Arreglos y tablas

Anteriormente se mostró como definir arreglos y tablas y de que manera se inicializan. También se mencionó el modo de direccionamiento indirecto mediante el registro BX, con lo que se facilitaba recorrer la tabla definida. Veremos a continuación unos ejemplos concretos.

Supongamos que queremos encontrar el máximo número almacenado en una tabla de words. No se sabe si los números están o no en orden, pero si que son todos números positivos (BSS). Si planteamos la solución en PASCAL, obtendríamos algo como esto:

```
const
  tabla: array[1..10] of Word = {5, 2, 10, 4, 5, 0, 4, 8, 1, 9};

var
  max: Word;

begin
  max := 0;
  for i := 1 to 10 do
    if tabla[i] > max then
      max := tabla[i];
end.
```

El programa inicializa la variable *max* con el mínimo valor posible y recorre la tabla comparando dicho valor con cada elemento de la tabla. Si alguno de esos elementos resulta ser mayor que el máximo actualmente almacenado en la variable *max*, se lo asigna a la variable y sigue recorriendo. Al finalizar la iteración, en *max* queda almacenado el máximo valor de la tabla.
Una posible implementación del programa anterior en assembly sería:

```
ORG 1000h

       tabla dw 5, 2, 10, 4, 5, 0, 4, 8, 1, 9;
       max   dw 0

ORG 2000h

       MOV BX, OFFSET tabla        ; (1)
       MOV CL, 0                   ; (2)
       MOV AX, max                 ; (3)
Loop:  CMP [BX], AX                ; (4)
       JC Menor                    ; (5)
       MOV AX, [BX]                ; (6)
Menor: ADD BX, 2                   ; (7)
       INC CL                      ; (8)
       CMP CL, 10                  ; (9)
       JNZ Loop                    ; (10)
       MOV max, AX                 ; (11)
       HLT                         ; (12)
END
```

La instrucción (1) se encarga de cargar en BX la dirección del comienzo de la tabla. En (2) se inicializa en 0 el contador que va a usarse para saber cuantas iteraciones van haciendo. En (3) se carga en AX el valor almacenado en la variable *max*, que inicialmente es el mínimo posible. En (4) se compara el máximo actual, que se encuentra en AX, con el número de la tabla apuntado actualmente por BX. Si el número es mayor o igual que AX, al restarlos no habrá *borrow*. Si el número en la tabla es menor que AX, al restarlos se producirá un *borrow*, que se indicará en el flag C. Por eso, si el flag C queda en 1, en (5) salta a la etiqueta *Menor*, por lo que la ejecución continua en la instrucción (7). Si el flag C queda en 0, continua ejecutando (6), que lo que hace es reemplazar el máximo actual guardado en AX por el nuevo máximo encontrado. En ambos casos, la ejecución continúa en la instrucción (7), que se encarga de incrementar BX para que apunte al próximo número de la tabla. Como cada entrada de la tabla es de 16 bits (o dos bytes), es necesario incrementar BX en 2 para que apunte a la siguiente palabra. En (8) se incrementa el contador y en (9) se verifica que no se haya llegado al final de la iteración. Si el contador no llegó a 10, en (10) se produce el salto a la etiqueta *Loop* de manera de continuar con la iteración. Si, en cambio, el contador CL llegó a 10, el salto no se produce y continúa la ejecución en (11), instrucción que se encarga de asignar el máximo almacenado en AX a la variable *max*. Por último, el programa termina su ejecución con un HLT en (12).

Organización de Computadoras 2003

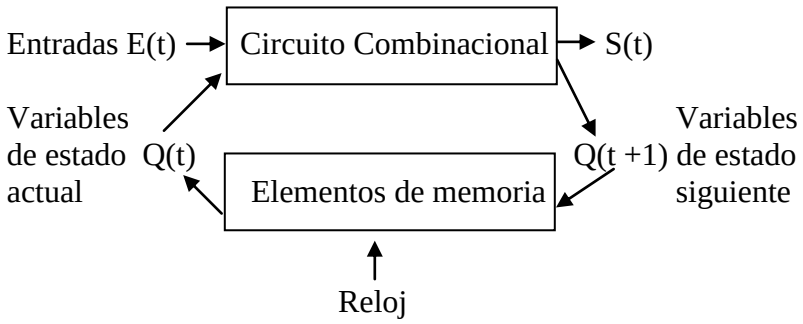
Apunte 5: Circuitos Lógicos Secuenciales

Introducción:

En el desarrollo de los sistemas digitales es fundamental el almacenamiento de la información, esta característica la permiten los **Circuitos Lógicos Secuenciales**.
Para poder entender el tema se necesitan ciertos conceptos básicos sobre el funcionamiento de los Circuitos Lógicos Combinacionales, respuesta de las compuertas lógicas ante sus entradas. Un circuito lógico combinacional se puede asociar rápidamente al estudio de compuertas lógicas. En este tipo de circuitos sus salidas dependen exclusivamente de sus entradas actuales.
Los circuitos lógicos secuenciales se asocian al estudio de dispositivos de almacenamiento en general y en donde una de sus características principales es que sus salidas dependen de las entradas actuales, de entradas en tiempos anteriores y de una señal externa de reloj.

Conceptos Generales de Latches y flip-flops:

Los circuitos biestables son aquellos que poseen dos estados estables que se pueden mantener por tiempo indefinido, lo que nos permite tener almacenado un dato en un dispositivo por el tiempo que se desee.
Las salidas del circuito, además de ser función de las entradas son función de la información almacenada en elementos de memoria del circuito, en el momento que se producen las entradas. Están formados por un circuito combinacional y un bloque de elementos de memoria:



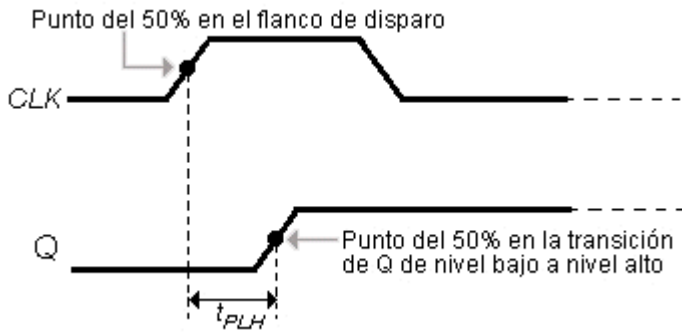
La señal del reloj indica a los elementos de memoria cuando deben cambiar su estado.
Existen dos tipos de biestables muy importantes: el latch y el flip-flop. Estos circuitos están compuestos por compuertas lógicas y lazos de retroalimentación y son considerados los circuitos básicos que constituyen los sistemas digitales.
El latch es un circuito biestable asíncrono, es decir que sus salidas cambian en la medida en que sus entradas cambien. El flip-flop es un dispositivo secuencial sincrónico que toma

muestras de sus entradas y determina una salida sólo en los tiempos determinados por el reloj (CLK).
Además, se pueden tener flip-flops Master-Slave y flip-flops disparados por flanco. Los flip-flops Master-Slave están conformados por dos latches con habilitación en cascada, es decir que la salida de un latch es la entrada del otro, mientras que el flip-flop disparado por flanco posee un dispositivo para determinar cuando hay una pendiente, ya sea de subida o de bajada, en el reloj que habilita el flip-flop.

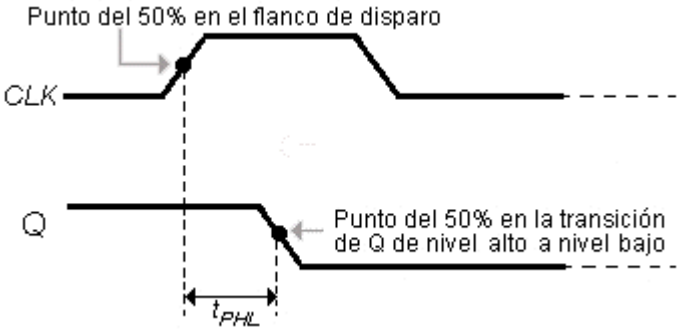
Características de operación:

Las características de operación son de gran importancia porque permiten entender y manipular de forma correcta los flip-flops. Estas características dan al usuario la posibilidad de prever y corregir errores (ejemplos: Retardos de propagación, tiempo de Set-up, Frecuencia máxima del reloj, Ancho de pulso, Disipación de potencia).

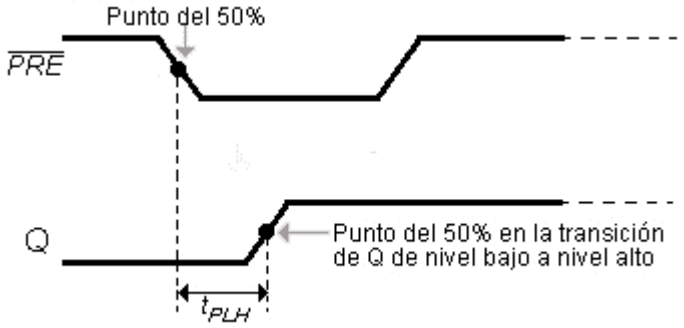
Retardos de propagación:
Es el tiempo requerido para que se produzca un cambio válido a la salida, debido a la aplicación de una señal en la entrada.
Existen cuatro categorías de retardos de propagación en los flip-flops:
1. Retardo entre el reloj y la salida (t_{PLH})



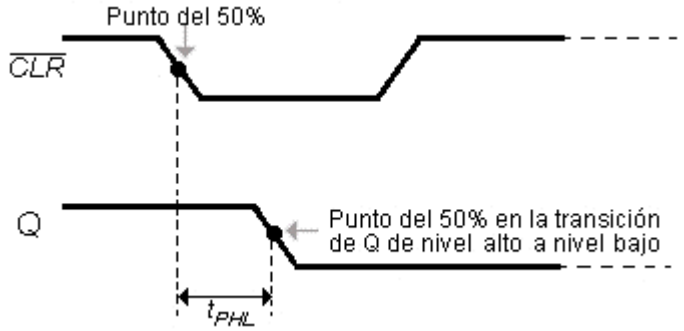
2.Retardo entre el reloj y la salida (t_{PHL})



3. Retardo entre la señal de iniciación y la salida (t_{PLH}):

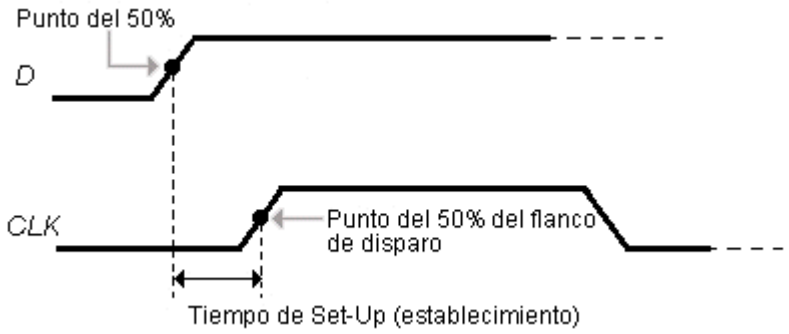


4. Retardo entre la señal de borrado y la salida (t_{PHL}):



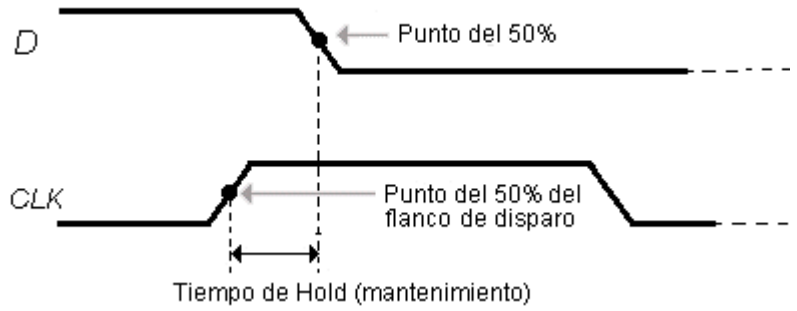
Tiempos de SetUp:

Es el intervalo mínimo que deben mantener fijas las entradas de un flip-flop antes que aparezca el flanco de disparo del impulso de reloj.
Ej en flip-flop D:



Tiempo de Hold:

Es el intervalo mínimo que deben mantener fijas las entradas de un flip-flop despues que aparezca el borde de disparo del impulso de reloj.
Ej en flip-flop D:



Frecuencia máxima del flip-flop:

Es la máxima velocidad a la que puede trabajar el flip-flop.

Ancho de Pulso:

Es el ancho mínimo del impulso. Está asociado con las entradas de reloj, iniciación "Preset" y borrado "Clear".

Disipación de potencia:

Se define como la potencia total consumida por el dispositivo.

El Flip-Flop S-R (Set-Reset):

Es un circuito biestable conformado por un detector de transición de impulsos que está encargado de detectar cuándo se tiene un flanco de subida o de bajada del reloj (CLK), dos compuertas NAND encargadas de enviar estos pulsos a las compuertas OR. En estas compuertas OR, una de las salidas está conectada a la entrada de la otra compuerta, logrando una retroalimentación:

Diagrama lógico del flip-flop S-R:

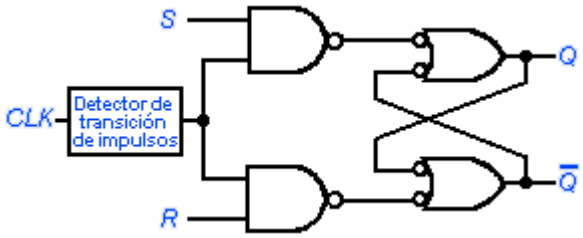
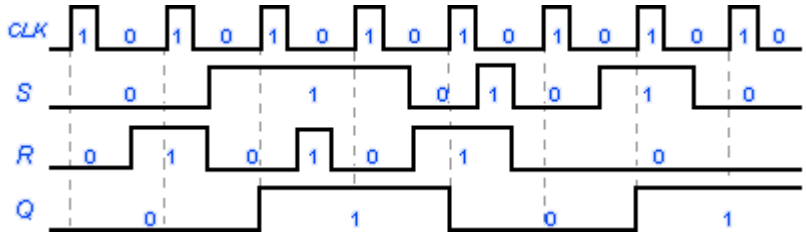


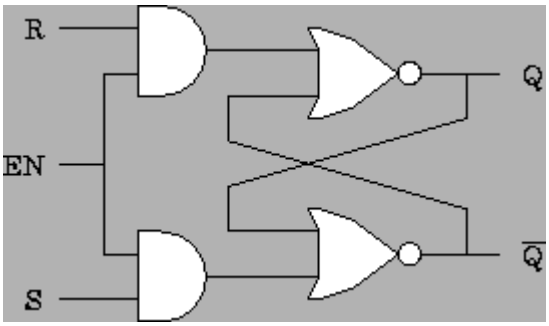
Tabla de verdad del flip-flop S-R:

Entradas			Salidas	
CLK	S	R	Q	$\overline{Q}$
X	0	0	Q_0	$\overline{Q}_0$
↑	0	1	0	1
↑	1	0	1	0
↑	1	1	?	?

Diagrama de tiempos del flip-flop S-R:



Utilizando las equivalencias lógicas podemos definir al flip-flop S-R mediante 2 compuertas NOR:



El flip-flop D:

Está compuesto por dos compuertas NAND encargadas de enviar la señal de habilitación a las compuertas OR (al igual que el flip-flop SR se puede construir con otras compuertas lógicas). La salida de una compuerta OR se transforma en la entrada de la otra (retroalimentación). Se puede observar la similitud con el flip-flop SR, solamente difieren en una entrada de habilitación y en que la entrada de Reset es igual a la de Set negada

Diagrama lógico del flip-flop D:

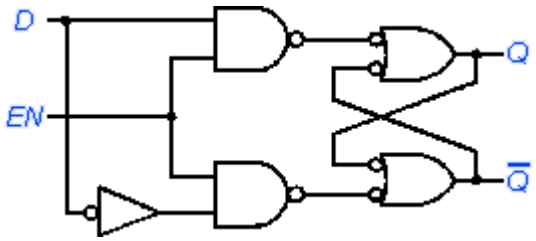
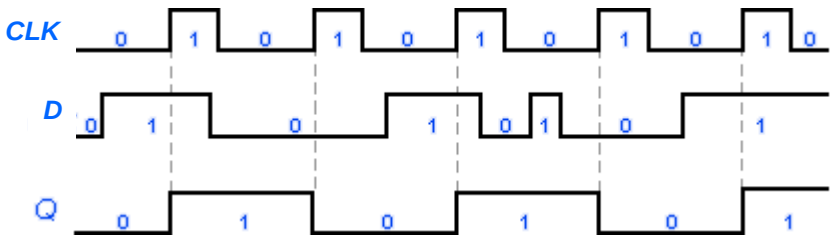


Tabla de verdad del Flip-Flop D:

Entradas		Salidas	
CLK	D	Q	$\overline{Q}$
0	X	Q_0	$\overline{Q}_0$
1	X	Q_0	$\overline{Q}_0$
↑	0	0	1
↑	1	1	0

Diagrama de Tiempos del flip-flop D:



El biestable S-R presenta problemas cuando se activan simultáneamente las dos entradas S y R. Podemos diseñar un biestable similar que no presente estos problemas a partir de un biestable D (el resultado es el flip-flop J-K):

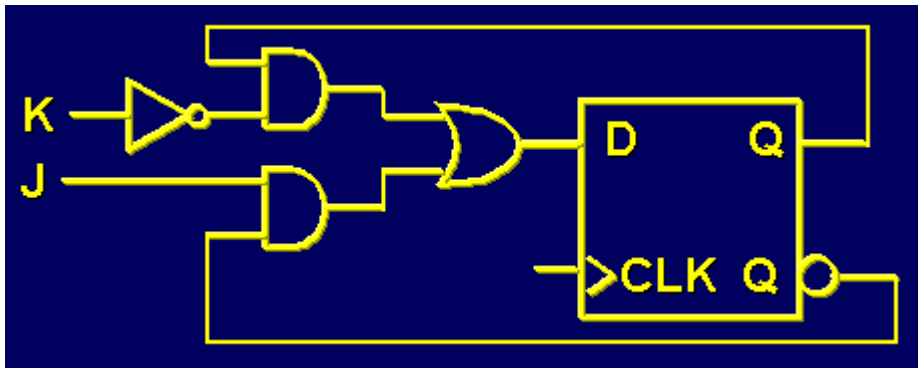
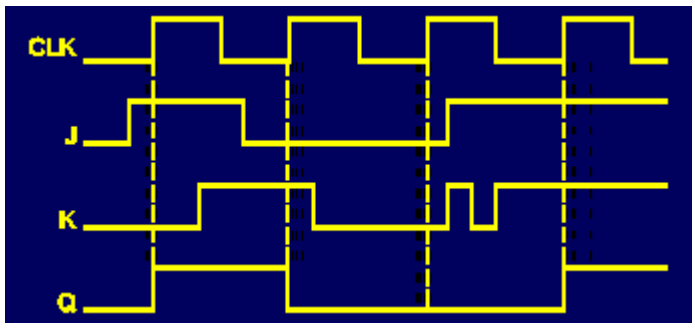


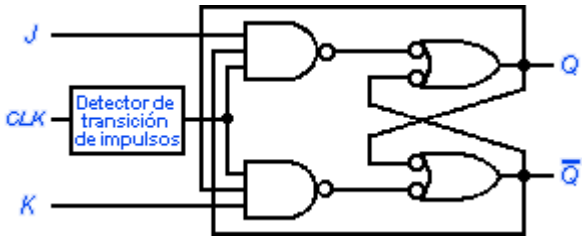
Tabla de verdad del biestable J-K:

CLK	J	K	Q(t+1)	/Q(t+1)
0	X	X	Q(t)	/Q(t)
1	X	X	Q(t)	/Q(t)
↑	0	0	Q(t)	/Q(t)
↑	0	1	0	1
↑	1	0	1	0
↑	1	1	/Q(t)	Q(t)

Cronograma del biestable J-K (activado por flanco de subida):



Por medio de equivalencias lógicas se puede obtener el siguiente diagrama lógico para el flip flop J-K:



En este caso, para lograr un valor estable cuando se activan ambas entradas se hace una retroalimentación de Q y /Q con las compuertas de la entrada.

El Flip-Flop T (Toggle):

Mantiene su estado o lo cambia dependiendo del valor de T cada vez que se activa.
Se puede implementar utilizando un biestable J-K

Diagrama lógico del flip-flop T:

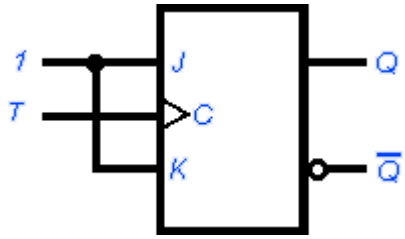
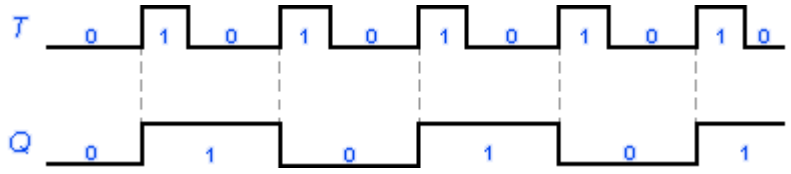


Tabla de verdad del flip-flop T:

Entradas T	Salidas Q	Salidas $\overline{Q}$
↑	$\overline{Q}_0$	Q_0
—	Q_0	$\overline{Q}_0$

Diagrama de tiempos del flip-flop T:

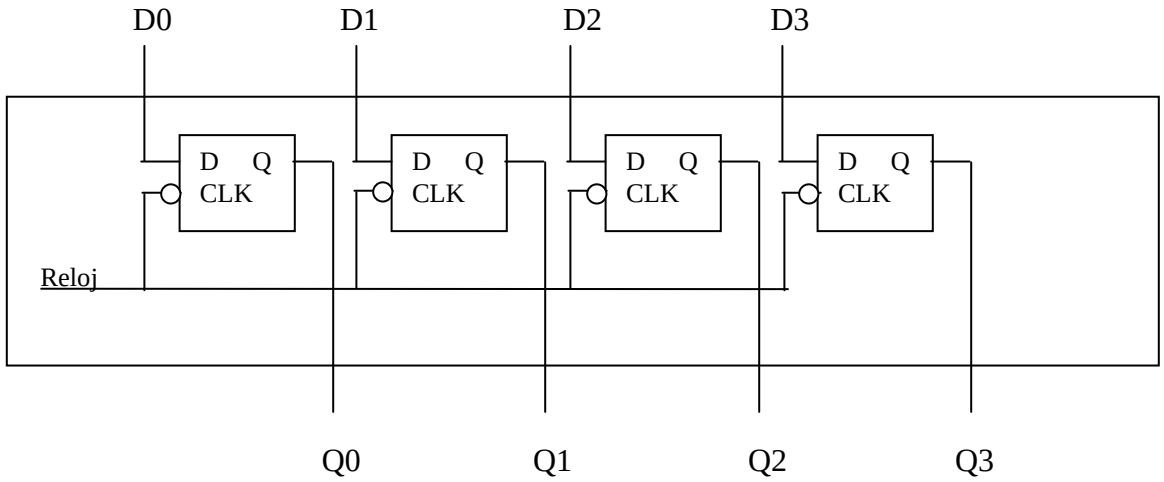


Registros:

Se forman a partir de biestables de tipo D conectados en cascada. Un registro con N biestables es capaz de almacenar N bits. Son circuitos sincrónicos y todos los biestables están gobernados por la misma señal de reloj.
Podemos definir dos tipos de registros (de Almacenamiento y de Desplazamiento)

Registros de Almacenamiento:

Ej: Registro de 4 bits



D=(D0,D1,D2,D3) es el dato a escribir.
Q=(Q0,Q1,Q2,Q3) es el dato leído.

Registros de Desplazamiento:

Son circuitos sincrónicos que cuando se activan, se desplazan los bits de sus biestables "hacia derecha" o "hacia izquierda".
Se clasifican de la siguiente manera:
Entrada Serie Salida Paralelo
Entrada Serie Salida Serie
Entrada Paralelo Salida Paralelo
Entrada Paralelo Salida Serie
Registros Universales.

Entrada/Salida Serie: Entra/Sale un bit en cada pulso de reloj.
Entrada/Salida paralelo: Entran/Salen todos los bits del dato en el mismo pulso de reloj

Ej: Registro de Desplazamiento de 4 bits (Entrada Serie)

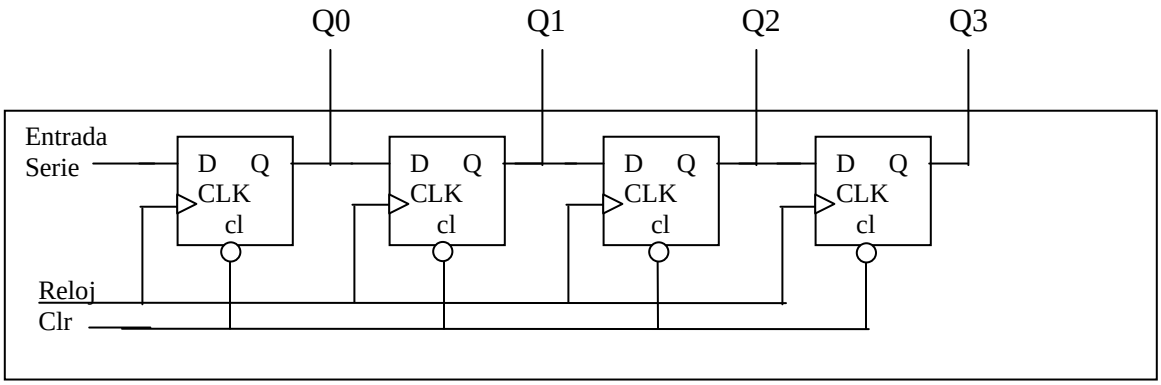


Tabla de funcionamiento:

Entradas			Salidas			
/Clear	Reloj	EntradaSerie	Q0	Q1	Q2	Q3
1	↑	0	0	Q0´	Q1´	Q2´
1	↑	1	1	Q0´	Q1´	Q2´
1	0	X	Q0´	Q1´	Q2´	Q3´
0	X	X	0	0	0	0

Contadores:

Un contador de N bits se implementa utilizando N biestables de tipo T.
Los contadores se pueden clasificar en:
-Asíncronos y Síncronos
-Ascendentes y Descendentes
-Módulo N.

Contadores Asíncronos: Sólo utilizan biestables sin ninguna puerta lógica adicional.
La entrada de reloj al contador sólo se conecta al primero de los biestables (el de menor peso).
La salida de estado de cada biestable (Q) o la complementaria (/Q) se conecta con el inmediato posterior.
Las entradas de datos de los biestables (J-K o T) se conectan a un "1" fijo.

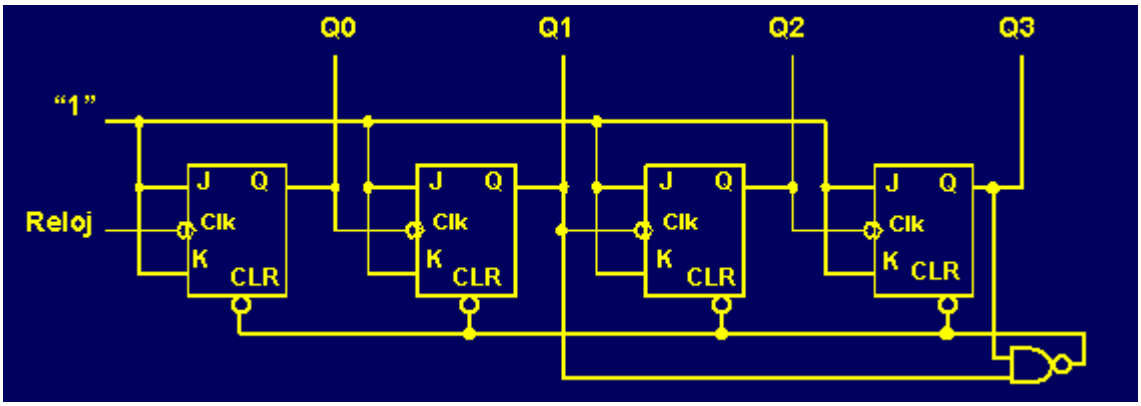
Contadores Sincrónicos: La señal externa del reloj está conectada a todos los biestables, por lo tanto, se activan todos de manera simultánea.
La entradas de reloja al contador se conecta a las entradas de reloj de todos los biestables.
La entrada de datos (J-K o T) del biestable de menor peso se conecta a un "1" fijo.
Se precisan puertas adicionales para implementar la lógica que indique cuando deben voltear su estado los biestables.

Contadores módulo N: El módulo de un contador es el número de cuentas distintas que realiza dicho contador.
Para implementar un contador de módulo N, se elige un contador con n bits (ascendente o descendente), siendo $2^{n-1} < N < 2^n$ y se eliminan las cuentas sobrantes, añadiendo lógica combinacional. Por ejemplo, para implementar un contador asíncrono módulo diez ascendente, que cuente los diez dígitos decimales se necesita un contador ascendente de 4 bits, ya que $2^3 < 4 < 2^4$, y se añade la lógica combinacional requerida.

Ejemplo: Contador asíncrono módulo 10:

Paso 1: elegir un Contador ascendente de 4 bits (de 0 a 15)
Paso 2: Detectar el 10 (1010 en binario) con una compuerta NAND
Paso 3: Reset de todos los biestables cuando ocurra la detección

Circuito:



Apunte 6: Conceptos sobre Parámetros Eléctricos

Contenido:

La Fuente de Alimentación:

Cuando se piensa en la configuración de una computadora comúnmente se presta principal atención al procesador, la memoria RAM, la placa base, el disco duro y los dispositivos ópticos, pero rara vez se piensa en la fuente de alimentación. Esto es una falta grave debido a la importancia que tiene la fuente dentro de los componentes que conforman una computadora. Su valor radica en que es la encargada de suministrar energía a todo el sistema, energía sin la cual todos los otros componentes no podrían funcionar.

Una característica fundamental que se tiene en cuenta a la hora de seleccionar una fuente es la potencia que ésta suministra y la calidad de la misma. Las necesidades de potencia pueden ser muy variables, dependiendo de qué componentes integren el sistema que se desee servir. Este parámetro ha ido variando con los años, ya que si antes bastaba una fuente de 250 o 300 watts, esa potencia hoy en día es insuficiente, estableciéndose el mínimo requerido en torno a los 450 watts.

Cuando se observa las características de una fuente de alimentación, ¿qué quiere decir que ésta suministra 400 watts? ¿ó 500 watts?¿qué significa que en el pin de 12 volts se suministra una corriente de 16 amperes? ¿Hay alguna relación entre estos parámetros? Para responder estas preguntas se deberá adquirir el conocimiento básico sobre los conceptos físicos que reflejan el comportamiento de la electricidad.

Como se sabe, la electricidad es el fenómeno físico que “le da vida” a las computadoras. Sin ella no se podría representar la información binaria dentro de la memoria como prendido o apagado, el disco rígido no podría girar a tantas revoluciones por minuto, así como tampoco funcionarían los miles de componentes electrónicos que se encuentran en la placa base ni en el microprocesador. Aun conociendo la importancia de la electricidad, rara vez los usuarios se han detenido a pensar cuáles son los fenómenos físicos que la explican. A continuación se verán los conceptos eléctricos básicos, de modo que al finalizar el apunte el lector haya adquirido el conocimiento acerca de qué son y cómo se relacionan entre sí.

Cargas Eléctricas:

Se considera un experimento simple en el que interviene la atracción eléctrica. Una barra de plástico se frota con un trozo de piel que se sostiene de una cuerda que puede girar libremente. Si se aproxima a esta barra una segunda de plástico, frotada también con piel, se podrá observar que las barras se repelen entre sí. El mismo resultado se obtiene si se repite el mismo experimento con dos barras de vidrio que han sido frotadas con seda. Sin embargo, si se utiliza una barra de plástico frotada con piel y una varilla de vidrio frotada con seda, se podrá observar que las barras se atraen entre sí. Se dice entonces que los cuerpos están cargados estáticamente.

Para explicar cómo se origina la electricidad estática, se ha de considerar que la materia está compuesta de átomos, y los átomos de partículas cargadas, de modo de que queda conformada por un núcleo compuesto por protones con carga positiva y de neutrones carentes de carga eléctrica, rodeado de una nube

de electrones que tienen carga negativa. Normalmente, la materia es neutra, tiene el mismo número de cargas positivas (protones) y negativas (electrones).

En el caso de las barra de vidrio, cuando se frotan con un paño de seda, se transfieren electrones del vidrio a la seda y por lo tanto ésta adquiere un número en exceso de electrones y el vidrio queda con un déficit de estas partículas. Mientras tanto, cuando se frota una barra de plástico con un trozo de piel, ésta transfiere electrones a la barra, haciendo que ésta quede con más electrones que protones. El exceso de electrones da lugar a materiales cargados "negativamente" y el déficit de estos, a materiales cargados "positivamente".

Si se realiza nuevamente el experimento, se podría concluir fácilmente que al acercar dos barras de plástico, previamente frotadas, estas se repelen debido a que poseen la misma carga (negativa), al igual que ocurre con dos barras de vidrio (cargadas positivamente). Por el contrario, si se acercan una barra de vidrio a una segunda de plástico estas se atraen debido a que poseen cargas diferentes.

De lo anterior se concluye que:

- La materia contiene dos tipos de cargas eléctricas denominadas positivas y negativas.
- Los objetos no cargados poseen cantidades iguales de cada tipo de carga.
- Cuando un cuerpo se frota la carga se transfiere de un cuerpo al otro, uno de los cuerpos adquiere un exceso de carga positiva y el otro, un exceso de carga negativa.
- En cualquier proceso que ocurra en un sistema aislado, la carga total o neta no cambia.
- Los objetos cargados con cargas del mismo signo, se repelen.
- Los objetos cargados con cargas de distinto signo, se atraen.

Los electrones son las partículas más importantes en los mecanismos de la conducción eléctrica, ya que disponen de carga y movilidad para desplazarse por los materiales. La diferencia entre dos materiales vendrá dada, entre otras cosas, por la cantidad y movilidad de los electrones que la componen. Para poder realizar cálculos en donde intervengan las cargas eléctricas es necesario cuantificar su magnitud. La unidad de carga eléctrica es el Coulomb [C] y es equivalente a la carga que suman $6{,}28 \times 10^{18}$ electrones.

Potencial Eléctrico:

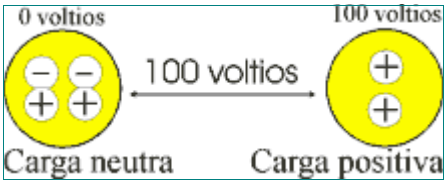
Si una carga positiva y otra negativa se separan a una cierta distancia y luego se las deja en libertad, éstas se atraerán una hacia la otra, debido a la fuerza presente entre ambas, la cual es explicada por la Ley de Coulomb. En estas condiciones se dice que ambas cargas adquirieron energía potencial eléctrica al separarlas. Esto se evidencia en el hecho que al dejarlas en libertad se aceleran una hacia la otra transformando la energía potencial en cinética (velocidad). Lo mismo ocurre si dos cuerpos se cargan, por cualquier medio, con polaridades distintas y se los interconecta mediante un conductor eléctrico. En efecto, entre ambos cuerpos existirá una diferencia de potencial en virtud de la diferencia de cargas eléctricas. Los electrones del cuerpo cargado con exceso de electrones (carga negativa) serán atraídos por los protones del cuerpo cargado positivamente intentando la neutralización de las cargas eléctricas.

De lo anterior puede observarse que a la diferencia de cargas eléctricas se la puede evaluar en función de la diferencia de potencial que producen.

Un "agente externo" deberá realizar un trabajo para quitar electrones de un cuerpo (dejándolo cargado positivamente) y colocarlos en otro cuerpo (cargándolo negativamente). Se define al potencial eléctrico, o mejor dicho a la "diferencia de potencial eléctrico" como el cociente entre el trabajo realizado por el agente para separar las cargas dividido la totalidad de cargas separadas. Matemáticamente:

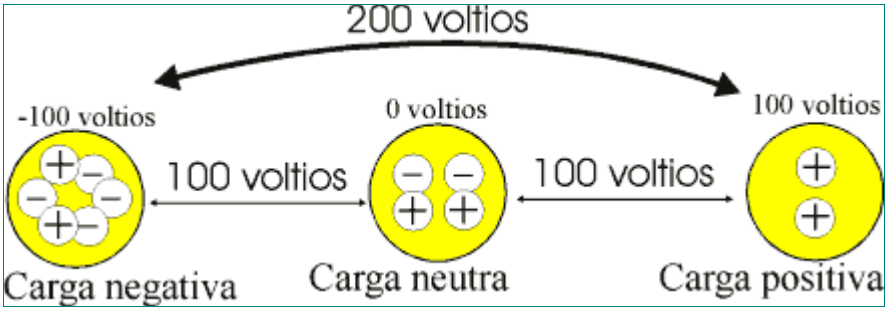
$$U = \frac{W}{Q}$$

donde U es la diferencia de potencial cuya unidad de medida es el Volt [V] , W es el trabajo realizado en Joule [J] y Q la cantidad de carga separada en Coulomb [C].



Ejemplo de dos cuerpos, uno cargado positivamente con +100 V, y el otro descargado (0 V.); entre ambos hay una diferencia de potencial de 100 V.

Ejemplo de dos cuerpos, uno cargado negativamente con -100 V, y el otro descargado (0 V.); entre ambos hay una diferencia de potencial de 100 V.



Ejemplo de tres cuerpos, uno descargado (0 V), otro cargado positivamente con +100 V, y un tercero cargado negativamente con -100. Entre la carga neutra y cualquiera de las cargas negativa o positiva hay una diferencia de potencial de 100 V, pero entre las cargas positiva y negativa hay una diferencia de potencial de 200 V.

Si a los dos cuerpos cargados mencionados anteriormente se los interconecta con un conductor eléctrico se establecerá entre ellos un flujo de electrones del cuerpo cargado negativamente al cargado positivamente. Este flujo se mantendrá hasta que la diferencia de cargas eléctricas entre los cuerpos quede neutralizada. Este flujo de electrones no es otra cosa que una corriente eléctrica.

Existen dispositivos que establecen en forma permanente una diferencia de potencial entre sus terminales (cuyo valor puede ser constante o variar periódicamente), tales como las baterías, generadores, alternadores, etc. Estos dispositivos se denominan de forma genérica “fuentes de alimentación” o “fuentes de tensión”.

Fuentes de tensión:

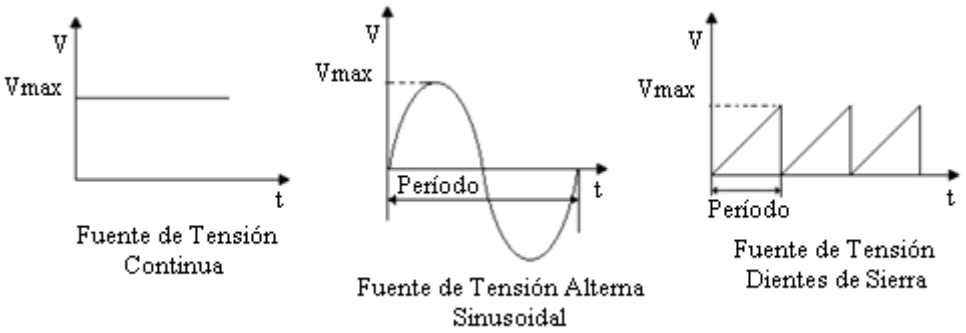
Imponen el valor de la tensión entre dos puntos del circuito, impulsando el flujo de electrones por el mismo. Las fuentes de tensión pueden ser constantes o variables en el tiempo según una ley preestablecida.

Cabe aclarar que la duración de los períodos es del orden de los milisegundos.

Ejemplos:

En la izquierda de la siguiente figura, se representa una fuente cuya tensión es invariable en el tiempo. A éstas fuentes se las denomina de tensión continua y la podemos encontrar en las pilas, baterías, dinamos, etc. La fuente de alimentación de las computadoras suministra una tensión de este tipo.

En el medio de la figura se representa una fuente en donde la tensión varía periódicamente al transcurrir el tiempo. A estas fuentes se las denomina de tensión alterna y son las que suministran los alternadores de las usinas generadoras de energía eléctrica.



Este tipo de tensión es la que se encuentra en los "toma-corriente" de nuestros hogares.
La tensión representada en la derecha de la figura es también periódica pero con una forma de onda denominada diente de sierra. Este tipo de fuentes tienen utilidad en algunos equipos eléctricos para uso industrial, médico, etc.

Corriente Eléctrica:

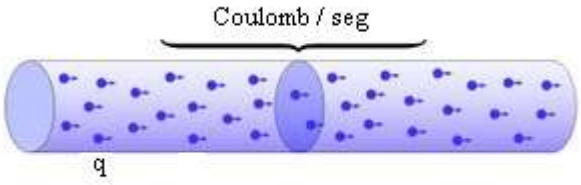
Cuando se aplica una diferencia de potencial o tensión entre los extremos de un conductor de cierta longitud, se establece un movimiento ordenado de los electrones desde el polo negativo al positivo de la fuente de tensión constituyéndose así una corriente eléctrica.

Se toma como sentido de la corriente el del flujo de cargas positivas. Esta convención fue establecida antes de que se conociera que los electrones libres, negativamente cargados, son las partículas que realmente se mueven produciendo la llamada corriente eléctrica. El movimiento de los electrones cargados negativamente en una dirección es equivalente al flujo de las cargas positivas en sentido opuesto. A los efectos de los cálculos es indistinto el sentido de circulación de las cargas eléctricas.

La corriente eléctrica se define como el flujo de cargas eléctricas que, por unidad de tiempo, atraviesan un área transversal. Su unidad de medida es el Ampere [A]:

$$1[A] = \frac{1[C]}{1[seg]}$$

Si por el área transversal de un conductor circula 1 (un) Coulomb por segundo, se dice que la intensidad de corriente es de 1 (un) Ampere.

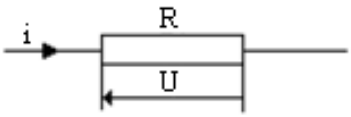


Resistencia:

Se define como resistencia eléctrica a la oposición ofrecida por un material al paso de la corriente eléctrica. Los resistores transforman la energía eléctrica en calor o en energía mecánica. El paso de cargas eléctricas (corriente eléctrica) a través de un resistor provoca una disminución de la energía potencial eléctrica de las cargas, generando una caída de tensión entre los dos puntos de conexión al circuito de dicha resistencia. El calor o energía transformada por el resistor es igual a la disminución de la energía potencial de las cargas que lo atravesaron.

La unidad de resistencia eléctrica es el Ohm (Ω).

Simbología del resistor o resistencia:



La punta de la flecha de **U** (tensión) indica el punto de mayor potencial, mientras que **i** el correspondiente sentido de la corriente.

Ley de Ohm:

Los tres parámetros eléctricos que se han explicado hasta ahora (tensión o diferencia de potencial, corriente y resistencia) se relacionan a través de la Ley de Ohm, la cual es fundamental para la resolución de los circuitos eléctricos. Esta ley establece que:

"La intensidad de corriente que circula por un circuito eléctrico es directamente proporcional a la tensión e inversamente proporcional a la resistencia"

Matemáticamente:

$$I = \frac{U}{R}$$

donde ***I*** es la intensidad de corriente en Amperes [A], ***U*** es la diferencia de potencial medida en Volts [V] y ***R*** es la resistencia expresada en Ohms [Ω].

Trabajo y Potencia:

El trabajo eléctrico que realizan las cargas eléctricas al atravesar un componente es directamente proporcional a la caída de potencial que experimentan. Si un resistor (o cualquier otro componente del circuito) experimenta una caída de tensión U cuando por el mismo circula, durante un determinado tiempo t, una corriente I, el trabajo W realizado por las cargas eléctricas es:

$$W = U \; x \; I \; x t$$

donde ***U*** está en Volts, ***I*** en Amperes , ***t*** en segundos y ***W*** en Joule.

El trabajo realizado por las cargas eléctricas que circulan a través del resistor se manifiesta como un aumento de temperatura de dicho resistor. Si en su lugar se encontrara un motor eléctrico el trabajo se manifestaría como energía mecánica en el eje del motor.

Se define la potencia ***P*** como el trabajo realizado en la unidad de tiempo, es decir:

$$P = \frac{W}{t} = \frac{U \; x \; I \; x t}{t} = U \; x \; I$$

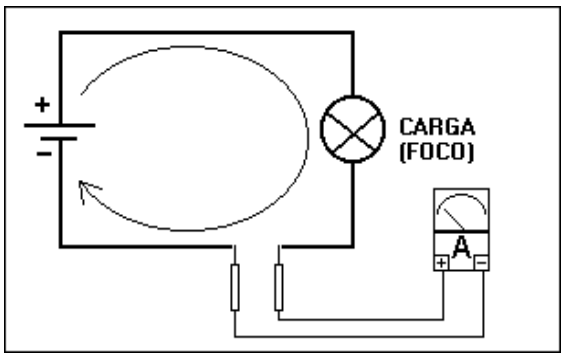
donde la potencia ***P*** está expresada en Watt.

Medición de los Parámetros:

Corriente:

Para medir la intensidad de corriente en un circuito eléctrico se utiliza un instrumento de medición denominado *amperímetro*, que puede ser analógico o digital. Este instrumento indica en amperes la cantidad de electrones que pasan por segundo en un punto del conductor.

El uso del amperímetro es similar al de un medidor del caudal de agua a través de una tubería; en ese caso el medidor se sitúa en medio del tubo, indicando así cuantos litros pasan a través de él por unidad de tiempo. En un circuito eléctrico debe suceder algo similar, por lo que el conductor tiene que ser cortado en algún punto para insertar el amperímetro en el camino de la corriente (esta forma de ubicación del medidor se denomina 'en serie').

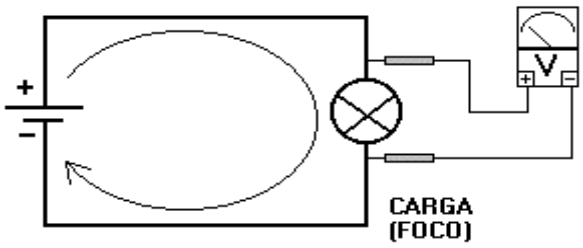


Amperímetro conectado 'en serie' en un circuito constituido por una fuente de alimentación y una carga (en este caso una lámpara eléctrica o foco).

Tensión:

En el caso de la medición de la tensión eléctrica interesa conocer la diferencia de potencial entre los

extremos de un elemento por el que circula la corriente eléctrica del circuito. Por lo tanto, el *voltímetro* debe colocarse en los extremos del componente para medir la diferencia de potencial existente (esta forma de ubicación del medidor se denomina 'en paralelo').

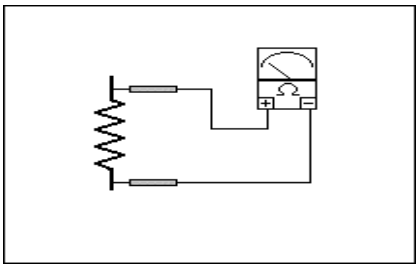


Voltímetro conectado 'en paralelo' a la carga (foco) cuya diferencia de potencial se desea saber.

El voltímetro tiene polaridad. Para medir correctamente la tensión hay que colocar los terminales negativo y positivo del aparato a los terminales negativo y positivo de la carga. Si los terminales están invertidos, la medida obtenida será de valor negativo.

Resistencia:

Para medir una resistencia, ésta se tiene que desconectar del circuito. El dispositivo usado para la medición se denomina *Ohmetro*. El instrumento utiliza una pila o batería interna para la alimentación de corriente eléctrica para funcionar. Su conexión debe realizarse en paralelo, tal como lo indica la figura:



Ohmetro conectado 'en paralelo' a la resistencia cuya oposición al paso de la corriente eléctrica se desea conocer.

Consumos:

Una computadora utiliza internamente tensiones de 12 V, 5 V, 3,3 V e inferiores reguladas directamente por la placa base para hacer funcionar todos los elementos que se conectan a ella. A continuación comentamos consumos y tensiones (aproximadas) utilizadas por algunos componentes.

- Placa base: en principio la placa utiliza todas las tensiones disponibles, ya sea directamente o bien transformándolas, para su propia alimentación o para alimentar otros componentes a través de ella. Para su consumo (chipset, BIOS, etc) suele utilizar 5 V y 3,3 V. La entrada de 5 V está siempre suministrándole este valor (aunque el ordenador esté apagado) para poder realizar acciones tales como el mismo encendido de la PC. El consumo medio de una placa base es entre 15 W y 20 W, a los que por supuesto hay que sumar los consumos de los elementos integrados (sonido, tarjeta de red, gráfica, etc.).
- Procesadores: la mayoría de los procesadores actuales trabajan a tensiones entre 1,8 V y 1,4 V, suministrados a través de la placa base. Procesadores anteriores pueden trabajar a tensiones diferentes, que van desde los 1,5 V hasta 5 V, dependiendo del modelo. El consumo se sitúa entre los 65 W y los 115 W, dependiendo del modelo de procesador y la tecnología que utilice (están en desarrollo procesadores con 45 W de consumo). Son habituales los de 90 W de consumo aunque cada vez son más frecuentes los de 65 W. Consumos superiores típicamente corresponden a procesadores en sistemas servidores (tipo Intel Xenon o AMD Opteron).
- Memorias: los módulos de memoria suelen trabajar entre 1,5 V y 2 V.

- Tarjetas Gráficas: las tarjetas gráficas suelen necesitar entre 3,3 V y 5 V para la transmisión de señal, y dependiendo del tipo de refrigeración que lleve, 5 V ó 12 V. La potencia que necesita depende de la gráfica, llegando a ser superior a los 115 W y en algunos casos, llegando a necesitar alimentación directa de la fuente de alimentación.
- Disqueteras: una disquetera utiliza 5 V para el procesamiento de datos y transmisión de señal y 12 V para motores, suministrados directamente de la fuente de alimentación. Su consumo es mayor a los 20 W.
- Discos Rígidos: Un disco rígido (ya sea IDE o SATA) utiliza 5 V para procesamiento de datos y transmisión de señal y 12 V para los motores, ambas tensiones suministrados directamente de la fuente de alimentación. Su consumo está entre los 20 W y 45 W.
- Unidades Lectoras y Grabadoras (CD/DVD): Una unidad lectora o grabadora de CD/DVD utiliza 5 V para procesamiento de datos y transmisión de señal y 12 V para motores, ambas tensiones suministradas directamente de la fuente de alimentación. Su consumo está entre los 25 W y 40 W.
- Ventiladores: Los ventiladores de la PC (disipador del procesador, externos, caja...) suelen trabajar a 12 V o 5 V, suministrados en unos casos a través de la placa base y en otros directamente de la fuente de alimentación. Los consumos son muy bajos (entre 5 W y 10 W).

Se ha de tener en cuenta que estos consumos (watts) no son fijos, ya que dependen de muchos factores. Por ejemplo, si se tienen dos discos rígidos de 40 W cada uno y dos unidades de CD/DVD de 40 W cada una NO significa que se tenga un consumo estable de 160 W, ya que no es normal que estén trabajando a la vez los dos discos rígidos y las dos lectoras. En este caso la única constante sería el consumo de los motores de giro de los discos rígidos. Tampoco es estable el consumo del procesador ni de la tarjeta gráfica, ya que dependerá del trabajo que esté realizando en ese momento.

Bibliografía utilizada:

- Libros:
  - Física para la Ciencia y Tecnología, Volumen 1, TIPLER-MOSCA, 5° edición.
- Páginas Internet:
  - http://www.viasatelital.com/proyectos_electronicos/corriente_voltaje_resistencia.htm
  - http://www.natureduca.com/fis_elec_cvr01.php
  - http://www.ing.unlp.edu.ar/aeron/catedras/archivos/electrotecnia_Apunte.pdf