Universidad Nacional Autónoma de México

Facultad de Ingeniería

(0840) Sistemas Operativos

Presentación:
“Aritmética Binaria”

Elaborado por: Torres Mayén Gerardo

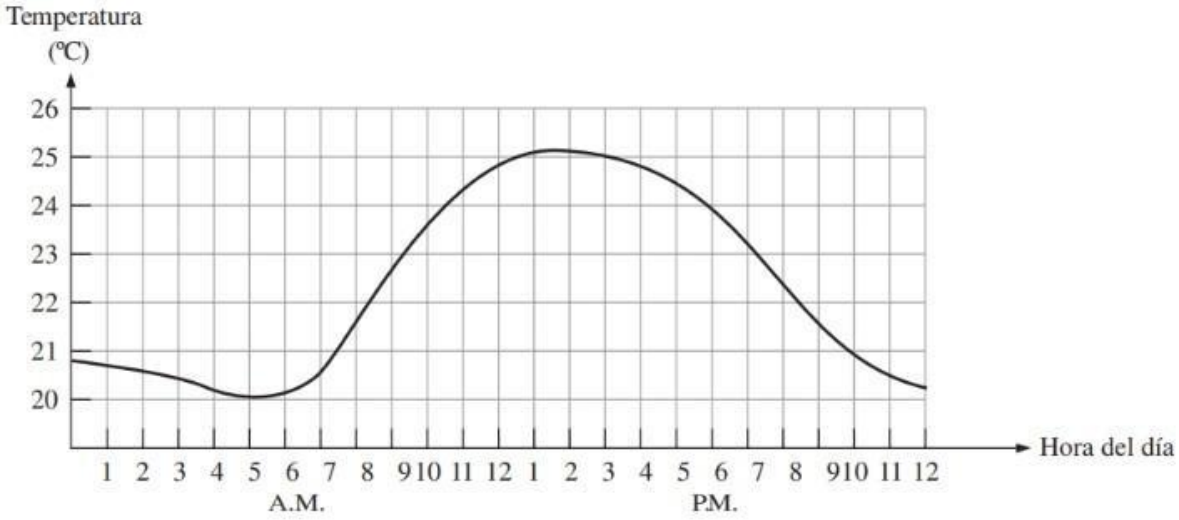
Fecha de entrega: 11/03/2023

Aritmética Binaria

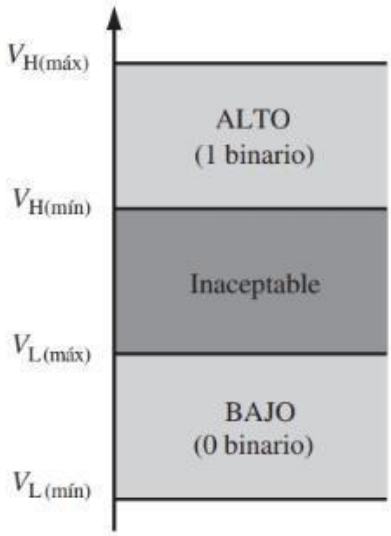
¿Por qué las computadoras utilizan el sistema binario para operar?

¿Qué es un sistema analógico?

Es un conjunto de **circuitos electrónicos** que utiliza señales variables y continuas en el tiempo o en el espacio para representar información. En un sistema analógico, las magnitudes físicas, como voltaje, corriente, temperatura, presión, etc. pueden tener un rango infinito de valores dentro de un intervalo determinado.



- ¿Qué es un sistema digital?



Es un conjunto de **circuitos electrónicos** capaz de **operar** datos **codificados** mediante señales **discretas** que representan **dígitos**.

En casi todos los sistemas digitales actuales, las señales emplean sólo dos valores discretos que asociamos a 0 o 1, así pues, los números en un sistema digital están formados por **dígitos** binarios que reciben el nombre de **bits** (***bi**nary digi**ts***) y se dice que están **codificados** en base 2.La electrónica digital utiliza magnitudes con valores discretos y la electrónica analógica emplea magnitudes con valores continuos.

- ¿Por qué las computadoras son sistemas digitales?

Las computadoras trabajan con el sistema binario porque su diseño se basa en circuitos electrónicos que pueden tener dos estados distintos: encendido o apagado, representados como 1 y 0 respectivamente.

Esto se debe a que los dispositivos electrónicos más simples, como los transistores, funcionan de manera binaria, respondiendo a la presencia o ausencia de corriente eléctrica. El sistema binario es eficiente y confiable para representar y procesar información digitalmente, permitiendo a las computadoras realizar cálculos y manipular datos de manera precisa y rápida mediante operaciones lógicas y aritméticas “simples” de implementar en dispositivos electrónicos.

La principal ventaja de esta implementación es que los datos digitales pueden ser procesados y transmitidos de forma más fiable y eficiente que los datos analógicos.

Otra ventaja de los sistemas digitales sobre los analógicos es su habilidad para almacenar con facilidad grandes cantidades de información de forma compacta y confiable durante periodos cortos o largos. Esta capacidad de memoria es lo que hace a los sistemas digitales tan versátiles y adaptables a muchas situaciones.

Por último, los sistemas digitales con casi inmunes al ruido. El "ruido en una señal" se refiere a cualquier perturbación no deseada o aleatoria que se superpone a la señal original que se está transmitiendo, procesando o recibiendo. Este ruido puede manifestarse como interferencias electromagnéticas, errores de medición, entre otros.

El ruido puede degradar la calidad de la señal y dificultar su interpretación o análisis. En muchos casos, el objetivo es minimizar o eliminar el ruido para obtener una representación más clara y precisa de la señal original.

Sistemas numéricos posicionales

Un sistema numérico posicional es un conjunto de caracteres y reglas que nos permiten construir números.

Reglas del sistema numérico posicional

1. Tiene una base.
2. Los caracteres deben ser únicos (no se cuenta como carácter a una combinación de otros caracteres del sistema).
3. Cada carácter tiene una posición.
4. En una posición solo puede haber 1 carácter.

5. Existe un símbolo que permite ubicar las posiciones de los caracteres de un número, éste es el carácter ".".
6. Los caracteres que decidamos utilizar definirán la base, de tal manera que, si permitimos n caracteres (sin contar “.”), n será la base del sistema.
7. Las posiciones inician hacia la izquierda del "." en la posición cero y hacia la derecha en -1.
8. Se va sumando de a 1 a la posición para obtener una nueva posición de la izquierda.
9. Se va restando de a 1 a la posición para obtener una nueva posición de la derecha.
10. El peso está definido como el cálculo de la base elevado a su posición.
11. El valor de un número en el sistema se calcula como la suma de productos del carácter en cada posición por el peso de la posición.
12. Se pueden construir infinitos números en el sistema, siempre que se cumplan las anteriores características y la definición del sistema.

Sistema binario

El número 2 es el menor de los números que se puede tomar como base de un sistema de numeración. En el sistema binario sólo aparecen dos símbolos (‘0’ y ‘1’).

Ventajas del sistema:

- Pocos símbolos.
- Las operaciones aritméticas se vuelven muy sencillas.

Desventajas del sistema:

- Necesitamos mucho más símbolos para representar cualquier número.
- Impráctico para leer.

Para saber el valor de un número binario es necesario sumar en base 10 los productos de cada carácter por el peso de su posición. Es decir:

$B = \{0,1\}; b_n \in B; m,n \in N$

$N = (b_n\, b_{n-1} \ldots b_0 \,.\, b_{-1} \ldots b_{-m})_2$

$N = [\, (b_n \times 2^n) + (b_{n-1} \times 2^{n-1}) + \ldots + (b_0 \times 2^0) + (b_{-1} \times 2^{-1}) + \ldots + (b_{-m} \times 2^{-m}) \,]_d$

$N = \sum_{i=0}^{n}(b_i \times 2^i) + \sum_{j=1}^{m}(b_j \times 2^{-j})$

Ejemplo:

Calcular el valor de $N_0 = (10.1)_2$

$= [(1 \times 2^1) + (0 \times 2^1) + (1 \times 2^{-1})]_d = [2 + ½]_d$

$(10.1)_2 = (2.5)_d$

Cambio de base

Para conocer la representación de un número entero escrito en base 10 en base 2 hay que realiza el siguiente algoritmo:

1. Divide el número decimal por 2, registra el cociente y el residuo.

2. Toma el cociente obtenido en el paso anterior y divídelo nuevamente por 2, registra el nuevo cociente y residuo.

3. Continúa dividiendo el último cociente obtenido hasta que el cociente sea 0.

4. La secuencia de residuos que has registrado, leída de abajo hacia arriba, es el número binario equivalente al número decimal inicial.

Por ejemplo, para convertir el número decimal 13 a binario:

1. 13 / 2 = 6, residuo 1 (bit **1**).
2. 6 / 2 = 3, residuo 0 (bit **0**).
3. 3 / 2 = 1, residuo 1 (bit **1**).
4. 1 / 2 = 0, residuo 1 (bit **1**).

Al leer los residuos de abajo hacia arriba, obtenemos: 1101, que es el equivalente binario de 13 en base 10.

$(13)_d = (1101)_2$

Para conocer la representación de un número con decimales escrito en base 10 en base 2 hay que realiza el siguiente algoritmo:

1. Hacer el cambio de base para la parte entera del número

2. Multiplica la parte decimal por 2.

3. Toma la parte entera del resultado como el primer bit del número binario.

4. Repite el proceso multiplicando la parte decimal del nuevo número por 2 y tomando la parte entera del resultado como el siguiente bit del número binario.

5. Continúa este proceso hasta que la parte decimal se convierta en 0 o hasta que obtengas la precisión deseada.

6. La secuencia de bits que has obtenido representará la parte decimal en binario.

Por ejemplo, para convertir el número decimal 13.12 a binario:

- Utilizamos nuestro resultado previo:
  $13_d = 1101_2$

- Aplicamos el algoritmo hasta los 10 bits:
  1. 0.12 * 2 = 0.24 (bit **0**).
  2. 0.24 * 2 = 0.48 (bit **0**).
  3. 0.48 * 2 = 0.96 (bit **0**).
  4. 0.96 * 2 = 1.92 (bit **1**).
  5. 0.92 * 2 = 1.84 (bit **1**).
  6. 0.84 * 2 = 1.68 (bit **1**).
  7. 0.68 * 2 = 1.36 (bit **1**).
  8. 0.36 * 2 = 0.72 (bit **0**).
  9. 0.72 * 2 = 1.44 (bit **1**).
  10. 0.44 * 2 = 0.88 (bit **0**).

- Resultado:
  $(13.12)_d = (1101.0001111010)_2$

Números negativos

En el sistema binario hay 3 formas de representar números negativos:

1. Magnitud verdadera

Es la forma más simple de representar números negativos en la que el bit más significativo (el bit de signo) indica si el número es positivo o negativo.
En la representación de magnitud verdadera, los números se representan de la misma manera que los números positivos, pero se utiliza un bit adicional para indicar el signo. Por ejemplo, en un sistema de 8 bits, donde el bit más significativo es el bit de signo, los números se representan de la siguiente manera:

- 0XXXXXXX: Representa números positivos, donde X puede ser 0 o 1.
- 1XXXXXXX: Representa números negativos, donde X puede ser 0 o 1.

Ventaja:

- Lectura directa del número representado .

Desventaja:

- El '0' tiene dos representaciones, lo que puede causar inconvenientes en ciertos algoritmos.

2. Complemento a 2

Se utiliza comúnmente en sistemas digitales debido a su simplicidad y eficiencia en operaciones aritméticas. Para obtener el complemento a 2 de un número primero hay que calcular el complemento a 1.Para obtener el complemento a 1 se invierten todos los bits del número original. Es decir, se cambian todos los 0 por 1 y todos los 1 por 0.
Una vez que se ha obtenido el complemento a 1, se le suma 1 al resultado. Esta suma se realiza bit a bit, comenzando desde el bit menos significativo (LSB).

N	0	1	1	0
C_N^1	1	0	0	1
			+	1
C_N^2	1	0	1	0

Ventaja:
Permite realizar restas con un algoritmo casi igual
al de la suma.

Desventaja:
Para leer el resultado obtenido en una operación, a veces debemos obtener el complemento de nuevo.

3. Sesgo o exceso

En el formato de sesgo, se asigna un valor de sesgo a los números representados, lo que desplaza el rango de representación de números negativos hacia valores positivos. En 8 bits (con un sesgo de 127) se vería de la siguiente manera:

[- 127, -1]	[+0, +127]
[1 111 1111, 1 000 0001]	[0 000 0000, 0 111 1111]
[0,126]	[127, 254]
[0000 0000, 0111 1110]	[0111 1111, 1111 1110]

Así, nuestro “0” ahora es el $127_d = (1000\ 0000)_2$, los números positivos son aquellos que están después del 127. Un número es negativo si está antes del 127.

- Para pasar un número en magnitud verdadera a sesgo debemos restar la magnitud verdadera al sesgo si es negativo, y si es positivo sólo debemos sumar sesgo y número normalmente.
- Para leer la magnitud verdadera de un número representado en exceso debemos restar el sesgo.

Ventaja:

- El cero tiene una representación
- “Ganamos” un número extra al eliminar una representación del cero.

Desventaja:

- La lectura de un número siempre es indirecta.

Suma y resta de binarios

Utilizamos la siguiente tabla para saber cómo resolver cada caso que puede aparecer. El acarreo ocurre cuando sumamos una columna de dígitos y el resultado supera la base del sistema numérico, así que anotamos el bit más significativo de nuestro resultado arriba de la siguiente columna que operaremos y el otro bit lo anotamos en el resultado de la columna actual. Siempre escribimos en el número de bits del número que más bits tiene, y siempre asumimos que: A > 0 ^ B > 0.

						Bit resultante	**¿Produce un acarreo?**
		0	+	0	=	0	No
		0	+	1	=	1	No
		1	+	0	=	1	No
		1	+	1	=	0	Sí
1	+	1	+	1	=	1	Sí

1. **Caso C = A + B:**
   Realizamos la suma considerando siempre el tamaño del resultado que obtendremos para evitar desbordamientos.

   Ejemplo:

   Calcular $(10111)_2 + (1001101)_2$

	Acarreo		1	1	1	1	1	
+	**A**	0	0	1	0	1	1	1
	B	1	0	0	1	1	0	1
=	**C**	1	1	0	0	1	0	0

   $(10111)_2 + (1001101)_2 = (1100100)_2$

2. **Caso C = A - B**

En este caso no nos preocupa el espacio**,** pues el resultado (C) siempre es menor que A y B.

1. Anotamos el número A tal cual, sólo agregamos un bit de signo (si no lo tiene) a la izquierda. Dicho bit vale 0 pues A es positivo.
2. Anotamos el complemento a 2 de B y agregamos un bit de signo (si no lo tiene) a la izquierda. Dicho bit vale 1 pues el término con B es negativo.
3. Realizamos la suma normalmente.
4. En el último bit que sumamos puede ocurrir un desbordamiento, lo ignoramos tranquilamente.
5. Si el resultado de la suma es positivo, el bit de signo quedará en 0 y el resultado se leerá directamente.
6. Si el resultado es negativo, el bit de signo quedará en 1 y para leer la magnitud verdadera deberemos obtener el complemento a 2 del número obtenido ignorando el bit de signo.

Calcular $(1001101)_2 - (1100100)_2$:

De nuestro resultado previo anticipamos la respuesta del ejercicio:

$(10111)_2 + (1001101)_2 = (1100100)_2$

$(10111)_2 - (10111)_2 + (1001101)_2 - (1100100)_2 = (1100100)_2 - (1100100)_2 - (10111)_2$

$(1001101)_2$ - $(1100100)_2$ = - $(10111)_2$

1. Escribimos A directamente:
   01001101

2. Escribimos B en complemento a 2 y con bit de signo negativo

B = 1100100
$C_B^2 = 0011011 + 1$
$D = 10011100$

3. Sumamos

		Signo							
	Acarreo			1	1	1			
+	**A**	0	1	0	0	1	1	0	1
	D	1	0	0	1	1	1	0	0
=	**E**	1	1	1	0	1	0	0	1

4. En E el bit de signo es 1, para obtener leer la magnitud del resultado hay que obtener su complemento a 2 ignorando el bit de signo.

$-C_E^2 = C$
$-C_E^2 = 0010110 + 1$
$C = -10111$

3. **Caso C = -A + B**

En este caso no nos preocupa el espacio**,** pues el resultado (C) siempre es menor que A y B.

1. Anotamos el complemento a 2 de A y agregamos un bit de signo (si no lo tiene) a la izquierda. Dicho bit vale 1 pues el término con A es negativo. Como B es positivo lo anotamos tal cual añadiendo el bit de signo (si es que no lo tiene)

2. El resto de los pasos son idénticos al caso anterior.

Ejemplo:

Calcular -$(1001101)_2$ +$(1100100)_2$:

De nuestro resultado previo anticipamos la respuesta del ejercicio

(-1) * [$(1001101)_2$ - $(1100100)_2$] = (-1) * [- $(10111)_2$]

$-(1001101)_2 + (1100100)_2 = (10111)_2$

1. Escribimos B directamente:
   01100100

2. Escribimos A en complemento a 2 y con bit de signo negativo
   A = 1001101
   $C_A^2 = 0110010 + 1$
   $D = \ 10110011$

3. Sumamos

		Signo							
	Acarreo	1	1						
+	B	0	1	1	0	0	1	0	0
	D	1	0	1	1	0	0	1	1
=	**E**	0	0	0	1	0	1	1	1

4. $E = C$
   $C = 10111$

4. **Caso C = -A - B**

En este caso sí nos preocupa el espacio**,** pues el resultado (C) puede no caber en el espacio que tengamos asignado.

1. Anotamos el complemento a 2 de A y agregamos un bit de signo (si no lo tiene) a la izquierda. Dicho bit vale 1 pues el término con A es negativo.
2. Anotamos el complemento a 2 de B y agregamos un bit de signo (si no lo tiene) a la izquierda. Dicho bit vale 1 pues el término con B es negativo.
3. Realizamos la suma normalmente.
4. En el último bit que sumamos puede ocurrir un desbordamiento, lo ignoramos tranquilamente.
5. El bit de signo quedará en 1 y para leer la magnitud verdadera deberemos obtener el complemento a 2 del número obtenido ignorando el bit de signo.

Ejemplo:

Calcular $-(10111)_2 - (1001101)_2$

Aprovechamos nuestro resultado para anticipar la respuesta:

$(-1)[(10111)_2 + (1001101)_2] = (-1)\ [\ (1100100)_2]$

$-(10111)_2 - (1001101)_2 = -(1100100)_2$

1. Escribimos A en complemento a 2 y con bit de signo negativo

A = 0010111
$C_A^2 = 1101000 + 1$
$D = \ 11101001$

2. Escribimos B en complemento a 2 y con bit de signo negativo

B = 1001101
$C_B^2 = 0110010 + 1$
$F = \ 10110011$

3. Sumamos

		Signo							
	Acarreo	1	1				1	1	
+	D	1	1	1	0	1	0	0	1
	F	1	0	1	1	0	0	1	1
=	**E**	1	0	0	1	1	1	0	0

4. En E el bit de signo es 1, para obtener leer la magnitud del resultado hay que obtener su complemento a 2 ignorando el bit de signo

$-C_E^2 = C$
$-C_E^2 = 1100011 + 1$
$C = -1100100$

Notación científica normalizada

Un número normalizado en notación científica es expresado como una potencia de 10 multiplicada por un coeficiente con decimales:

$$A = M \times 10^n$$

Donde M es un número menor a 1 en el que el primer dígito después del punto es distinto de cero y recibe el nombre de **mantisa**. n indica cuántos dígitos se debe desplazar el punto decimal hacia la izquierda o la derecha para leer el valor del número y recibe el nombre de **exponente**.
Recordemos que el punto se recorre

Ejemplo:

Normalizar los siguientes números en notación científica:

125098.345

$0.125098345 \times 10^{6}$

0.0000125098345

$0.125098345 \times 10^{-4}$

Notación binaria normalizada

Los número binarios también pueden ser representados en notación normalizada de la siguiente manera:

$$A = M \times (10_2)^n$$

n es el exponente (expresado en binario) y M es la mantisa (también en binario), que debe ser menor a 1.

Para pasar de un decimal a binario normalizado:

1. Pasamos la parte entera a binario
2. Pasamos la parte decimal a binario
3. Normalizamos

Para pasar de un binario normalizado a decimal y conocer su valor se realiza el siguiente algoritmo:

1. Desnormalizar el número
2. Sustituir en la expresión: $\sum_{i=0}^{n}(b_i \times 2^i) + \sum_{j=1}^{m}(b_j \times 2^{-j})$
3. Realizar el cálculo

Ejemplo:

Calcular el valor de N_0

$N_0 = (0.101 \times 10^{-10})_2$

$N_0 = (0.00101)_2$

n = 0, m = 5

$= [(0 \times 2^0) + (0 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3}) + (0 \times 2^{-4}) + (1 \times 2^{-5})]_d = [1/8+1/32]_d$

$(0.101 \times 10^{-10})_2 = (0.00101)_2 = [0.15625]_d$

Representación en memoria de un número decimal. (estándar IEEE 754)

Un número de punto flotante es una manera de representar números reales en la computadora. Este método de representación permite expresar números con una

cantidad fija de dígitos significativos (llamada mantisa) y un exponente que indica la escala del número.

- Precisión simple: Utiliza 32 bits para representar un número de punto flotante, con 1 bit para el signo, 8 bits para el exponente (codificado en exceso a 127) y 23 bits para la mantisa.
- Doble precisión: Utiliza 64 bits para representar un número de punto flotante, con 1 bit para el signo, 11 bits para el exponente (codificado en exceso a 1023) y 52 bits para la mantisa.

En el estándar IEEE-754 existen 5 tipos de valores que se pueden representar:

1. El cero
2. Valores normales
3. Valores denormales
4. $\pm\infty$
5. Not a number (NaN)

Números normales

Los valores válidos posibles para el exponente de un número **normal** son los siguientes:

[- 126, -1]	[+0, +127]
[1 111 1111, 1 000 0001]	[0 000 0000, 0 111 1111]
[1,126]	[127, 254]
[0000 0001, 0111 1110]	[0111 1111, 1111 1110]

El intervalo de valores **normales** posibles en 32 bits en el estándar IEEE-754 es:

$$[-(2-2^{-23})*2^{127}, -2^{-126}] \cup [+2^{-126}, +(2-2^{-23})*2^{127}]$$

Para representar un número normal utilizamos la notación binaria normalizada y guardamos en la mantisa todos los bits excepto el primero, de lo contrario necesitaríamos 24 bits para llegar al número máximo de la mantisa ($2-2^{-23}$). Al bit que no escribimos le llamamos **bit oculto**. Para el exponente primero lo representamos en sesgo a 127 y después guardamos todos sus bits.

Represente $(-0.1010111011101 \times 10^{-10})_2$ como un decimal de precisión simple:

S = 1
M = 1010111011101
m = 1010 1110 1110 1000 0000 0000

E = -10
e = 0111 1111 – 10 = 0111 1101 = 125_d

-0.101×10^{-10} = 1 0111 1101 010 1110 1110 1000 0000 0000

El cero

Para diferenciar un número normal del cero, el cero sólo aparece si el exponente y la mantisa son todos ceros. Evidentemente no consideramos el bit oculto y tiene dos representaciones posibles:

0 0000 0000 000 0000 0000 0000 0000 0000
1 0000 0000 000 0000 0000 0000 0000 0000

El infinito

Este valor puede aparecer si un criterio de redondeo se aplica sobre un número normal muy grande positivo o negativo. Para que se lea un infinito, la mantisa debe ser todo 0 y el exponente debe ser 255 (nunca se alcanza si el número es normal).

$+\infty$ = 0 1111 1111 000 0000 0000 0000 0000 0000
$-\infty$ = 1 1111 1111 000 0000 0000 0000 0000 0000

NaN

En español: no es un número. Se usa para expresar un resultado “imposible” de calcular, como el caso de raíces negativas, indeterminaciones, etc. Aparece como un número con mantisa distinta a 0 pero con exponente igual a 255.

Los números denormales

En los números denormales, la mantisa no tiene bit oculto ni se lee de forma normalizada (de ahí el nombre), sino que se trata de un número con un valor menor al mínimo normalizado. Aparece cuando la mantisa es distinta de 0 y el exponente vale todo 0, pero se interpreta como -126.

Los números denormales son útiles porque permiten que los sistemas de punto flotante mantengan una transición suave de valores muy pequeños a cero, en lugar de tener un salto brusco. Sin embargo, los cálculos que involucran números denormales pueden ser más lentos en hardware debido a la necesidad de manejar casos especiales. Además, la precisión de los cálculos con números denormales es menor que con números normales.

Algoritmo de Booth para multiplicar

Este algoritmo fue desarrollado por Andrew Donald Booth con la finalidad de aumentar la velocidad de los algoritmos básicos a la hora de hacer cálculos. Por

ejemplo, para multiplicar 1010_2 (10_d) x 1010_2 (10_d) podríamos sumar 10 veces el número 1010_2 o aplicar el algoritmo de multiplicación larga que aprendimos en la primaria. Sin embargo, en el primer caso hacemos una extensa cantidad de operaciones que conllevan un tiempo y energía, y en el segundo necesitamos guardar cada uno de los resultados parciales para luego sumarlos, esto implica mucha memoria y tiempo de lectura/escritura. En el algoritmo de Booth los pasos necesarios para llegar al resultado es el número de bits que se utilicen en la operación, también llamados bits de trabajo.

Mínimo número de bits de trabajo = Bits mínimos necesarios del número mayor + 1

En la práctica se trabaja con 2^n bits de trabajo.

Ventajas:

- Sólo necesitamos guardar 3 números para trabajar:
  1. Multiplicador (donde guardamos el resultado de las operaciones intermedias y el resultado final)
  2. Multiplicando
  3. Complemento a 2 del multiplicando
- Complejidad computacional lineal respecto al número de bits de los números a operar
- El algoritmo sólo necesita soporte de hardware para sumas
- Pocos casos base

“Desventaja”:

- Poco intuitivo

División rápida mediante el método de Newton para ecuaciones algebraica

La idea fundamental de este método es multiplicar por el inverso multiplicativo (recíproco) del denominador en lugar de llevar a cabo una costosa operación de división.

Partamos de la división que queremos computar:

$$C = \frac{A}{B}$$

Sabemos que el resultado lo podemos escribir como C/1:

$$\frac{A}{B} = \frac{C}{1}$$

También sabemos que multiplicar por uno, no afecta el resultado

$$\frac{A}{B} * \frac{x}{x} = \frac{C}{1}$$

Igualamos numerador y denominador

$$\{A * x = C \;\; B * x = 1$$

Despejamos “x”

$$x = \frac{1}{B}$$

$$C = A * \frac{1}{B}$$

Nuestro objetivo era encontrar la función cuya raíz es el número por el cual cambiaremos una división por una multiplicación.

$$B = \frac{1}{x}$$

$$f(x) = \frac{1}{x} - B = 0$$

Derivamos y sustituimos en la expresión del método de Newton

$$f'(x) = -\frac{1}{x^2}$$

$$x_{i+1} = x_i - \frac{\frac{1}{x_i} - B}{-\frac{1}{(x_i)^2}}$$

Simplificamos hasta que desaparezcan todas las divisiones:

$$x_{i+1} = x_i - \frac{\frac{1}{x_i} - B}{-\frac{1}{(x_i)^2}} \left[\frac{-(x_i)^2}{-(x_i)^2}\right]$$

$$x_{i+1} = x_i - \left[\frac{1}{x_i} - B\right][-(x_i)^2]$$

$$x_{i+1} = x_i + [x_i - B(x_i)^2]$$

$$x_{i+1} = 2x_i - B(x_i)^2$$

$$x_{i+1} = x_i * (2 - B * x_i)$$

Con esta expresión iterativa obtendremos el recíproco de B, por lo que el resultado de A / B será igual a multiplicar A por el resultado que nos arrojen las iteraciones del método de Newton.

Ventajas:

- El número de iteraciones que hagamos no depende del tamaño de los números que trabajamos sino de la precisión que deseemos.

- Convergencia muy rápida
- Para empezar el cálculo del recíproco sólo necesitamos el denominador y un valor semilla

“Desventaja”:

- Necesitamos una semilla.
- Si no elegimos una semilla adecuada, podría ocurrir una divergencia.
- El algoritmo necesita soporte de hardware para multiplicaciones.

En una versión modificada del método ni siquiera es necesaria una semilla.

Sistema hexadecimal

El problema del sistema binario es que necesita muchos dígitos para representar un número cualquiera en comparación con cualquier otra base.

Ejemplo:

202d sólo necesita tres dígitos en base 10 y ocho dígitos en base 2 -> 11001010b.

El sistema hexadecimal presenta la ventaja de ser compacto, a diferencia del sistema binario.

El sistema hexadecimal tiene varias aplicaciones en el ámbito de la informática y la tecnología, gracias a sus propiedades y características. Aquí hay algunas aplicaciones comunes del sistema hexadecimal:

- **Representación de direcciones de memoria**: En programación y sistemas informáticos, las direcciones de memoria se representan a menudo en hexadecimal. Esto facilita la identificación y manipulación de direcciones de memoria de una manera compacta y legible.

- **Representación de colores**: En el campo del diseño gráfico, la representación de colores en formato hexadecimal es muy común. Los colores se pueden especificar usando códigos hexadecimales de seis dígitos que representan la cantidad de rojo, verde y azul (RGB) en un color específico.

- **Direcciones de red y MAC**: En redes informáticas, las direcciones IP y las direcciones MAC de los dispositivos a menudo se representan en formato

hexadecimal. Esto es especialmente útil para configurar y diagnosticar problemas de red.

- **Representación de datos binarios:** El sistema hexadecimal se utiliza a menudo para representar datos binarios de una manera más compacta y legible. Por ejemplo, cuando se trabaja con archivos binarios, es común ver datos representados en formato hexadecimal.

- **Programación y depuración:** Los programadores a menudo utilizan el sistema hexadecimal para manipular y depurar datos en el nivel de la máquina. Esto es particularmente útil al trabajar con lenguajes de bajo nivel como el ensamblador.

- **Criptografía:** En el campo de la criptografía, el sistema hexadecimal se utiliza para representar claves de cifrado y firmas digitales de una manera más legible y manejable.

Dado que 16 (base hexadecimal) es una potencia de 2 (base binaria), las conversiones entre estos dos sistemas son relativamente simples. Un dígito hexadecimal es equivalente a cuatro dígitos binarios (porque 2^4 = 16). Esto permite que la conversión de binario a hexadecimal y viceversa sea directa y eficiente.

- Para convertir un número binario a hexadecimal, agrupas los dígitos binarios en grupos de cuatro, desde la derecha hacia la izquierda, y luego conviertes cada grupo de cuatro dígitos binarios en su equivalente hexadecimal.

- Para convertir de hexadecimal a binario, simplemente haces el proceso inverso: conviertes cada dígito hexadecimal en su equivalente binario de cuatro dígitos.

Para realizar la conversión de base dos a base dieciséis, basta con sustituir cada dígito hexadecimal en su posición por su representación binaria.

Tabla de Equivalencias 2-16		
Decimal	Binario	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

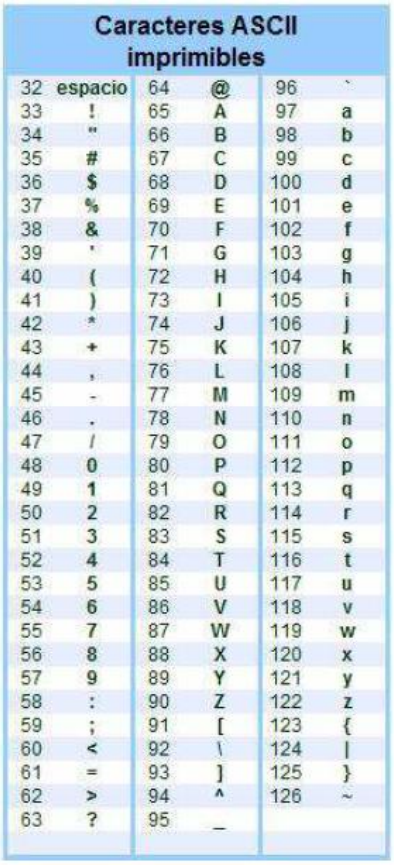
Ejemplo:
1234h = 0001 0010 0011 0100_2
$1111\ 0000\ 1010\ 1011_2$ = F 0 A Bh

Impresión de un número en ASCII

Hasta ahora todos los formatos vistos son para realizar cálculos, pero y para visualizar la información, ¿cómo lo resolvemos?
Para uniformar la representación de caracteres, los fabricantes de microcomputadoras adoptaron el código ASCII, un código uniforme que facilita la transferencia de información entre los diferentes dispositivos de la computadora.

Está dividido en 4 grupos de 32 caracteres cada uno. Los primeros 32, (0h al Fh) forman un set especial de caracteres no imprimibles llamados caracteres de control. Se le llaman así porque ejecutan operaciones de control, de despliegue e impresión. Ejemplo de éstos es el retorno de carro, el cual posiciona el cursor a la izquierda de la línea actual de caracteres, “line feed” (el cual mueve el cursor debajo de una línea, en el dispositivo de salida).
Fue creado en 1963 por el Comité Estadounidense de Estándares o “ASA”, este organismo cambio su nombre en 1969 por “Instituto Estadounidense de Estándares Nacionales” o “ANSI”, como se lo conoce desde entonces.
En un primer momento sólo incluía las letras mayúsculas, pero en 1967 se agregaron las letras minúsculas y algunos caracteres de control, formando así lo que se conoce como US-ASCII, es decir los códigos del 0 al 127. Así con este conjunto de sólo 128 caracteres fue publicado en 1967 como estándar, conteniendo todos los necesarios

para escribir en idioma inglés. En 1986, se modificó el estándar para agregar nuevos caracteres latinos, necesarios para la escritura de textos en otros idiomas, como por ejemplo el español, así fue como se agregaron los caracteres que van del ASCII 128 al 255.Este ASCII extendido necesita de 8 bits para su representación.



Caracteres ASCII imprimibles					
32	espacio	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Partamos de un número entero con 3 dígitos decimales, el 13 por ejemplo.
255d = 1111 1111_2 = F F h
Luego sabemos que los caracteres que lo componen son el ‘2’, el ‘5’ y otro ‘5’, que escritos en ASCII se verían así:
[32h] [35h] [35h] = [0011 0010] [0011 0101] [0011 0101].
Entonces, debemos construir un algoritmo que nos permita pasar de un número binario puro a una representación en ASCII de caracteres decimales, que también está escrita en binario. Como paso intermedio utilizaremos una representación binaria de un número decimal, para ello guardaremos el valor de cada dígito en binario en grupos de 8 bits para que sea compatible con ASCII.

El 255d se vería así:
[0000 0010] [0000 0101] [0000 0101].

Para llegar a dicho número primero debemos determinar cuántos dígitos decimales necesitaremos y, por lo tanto, cuántos grupos de 8 bytes necesitaremos.

[Mínimo, Máximo]	Número mínimo de dígitos decimales	Divisor
[0h,9h]	1	1h
[Ah,63h]	2	Ah
[64h,3E7h]	3	64h
[3E8h,270Fh]	4	3E8h
[2710h,1869Fh]	5	2710h

Nuestro 255d necesita 3 dígitos decimales. El siguiente paso es dividir el número por el divisor que le corresponde al número de dígitos, en este caso es el 64h = 100d. La división entera que realizaremos es FFh / 64h = 02h y el residuo es el

siguiente número que dividiremos FFh mod 64h = 37h. El 02h es el primer dígito de nuestro resultado, por lo que guardamos en 8 bits para que sea compatible 02h = [0000 0010]. El 37h cae en la segunda fila por lo que hay que dividirlo entre Ah : 37h / Ah = 05h y el residuo es el siguiente número que dividiremos 37h mod 05h = 5h. El 05h es el segundo dígito de nuestro resultado y lo guardamos igual. Dividimos 05h / 1h = 5h y como 5 mod 1 = 0, detenemos el algoritmo. Nuestro tercer resultado también fue 5 y lo guardamos igual.
Resumido se ve así:

FFh / 64h = 2h -> [0000 0010] = [02]h

FFh mod 64h = 37h

37h / Ah = 5h -> [0000 0101] = [05]h

37h mod 5h = 5h

5h / 1h = 5h -> [0000 0101] = [05]h

5h mod 1h = 0 FIN DEL ALGORITMO

Así pasamos de [FF]h = [1111 1111] a [0000 0010] [0000 0101] [0000 0101] = [02][05][05]. Pero para que esté codificado en ASCII debe ser [32] [35] [35], entonces sólo debemos sumar 30h a cada dígito, y es todo, ya podremos visualizar un entero en ASCII.

Sistema octal.

El sistema octal es un sistema numérico posicional que utiliza una base de 8 dígitos, que son 0, 1, 2, 3, 4, 5, 6 y 7.También es muy utilizado para leer binarios pues resulta compacto y legible.

Binario	Octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Dado que 8 es una potencia de 2 (base binaria), las conversiones entre estos dos sistemas son relativamente simples. Un dígito octal es equivalente a 3 dígitos binarios (porque 2^3 = 8). Esto permite que la conversión de binario a octal y viceversa sea directa y eficiente.

Ejemplo:
111 111 110 = 7 7 6
El sistema hexadecimal tiene varias aplicaciones en el ámbito de la informática y la tecnología, gracias a sus propiedades y características. Aquí hay algunas aplicaciones comunes del sistema octal:

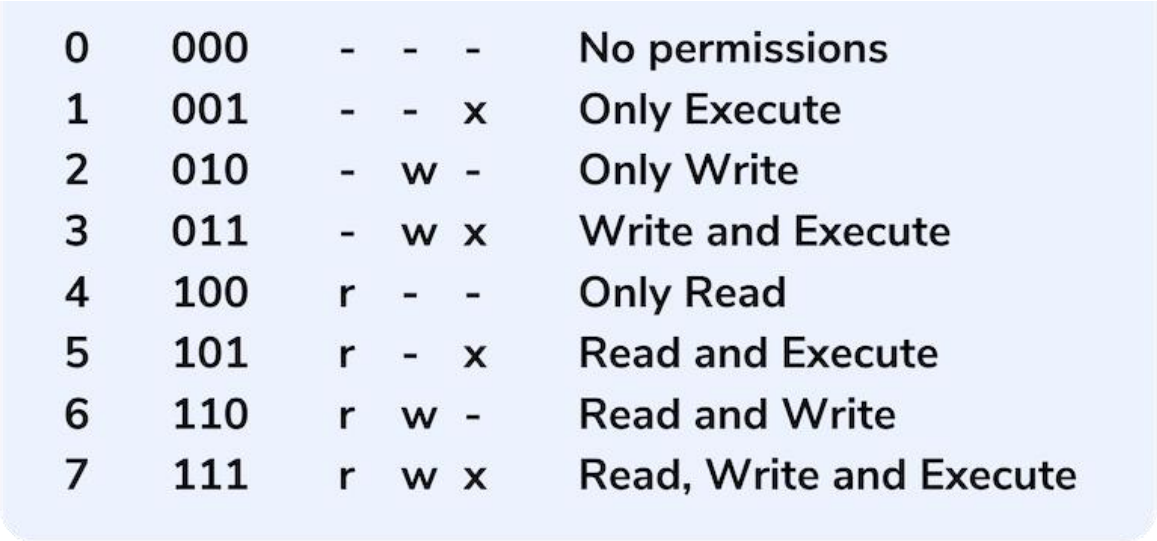
- **Representación de datos binarios**: Aunque el sistema octal es menos común que el sistema hexadecimal para representar datos binarios, aún se utiliza en algunas aplicaciones donde la compacidad y la legibilidad son importantes, especialmente en entornos de programación de bajo nivel.
- **Depuración de sistemas embebidos**: En sistemas embebidos y microcontroladores, donde los recursos son limitados, el sistema octal puede ser útil para representar y depurar datos y direcciones de memoria.
- **Sistemas de permisos en sistemas Unix y Linux**: En los sistemas Unix y Linux, los permisos de archivos y directorios se representan utilizando números octales. Cada dígito en el número octal representa los permisos para diferentes usuarios y grupos en el sistema de archivos.
- En sistemas Unix y Unix-like, como Linux, la cadena de permisos es una representación visual de los permisos de acceso de un archivo o directorio. Esta cadena consiste en diez caracteres que representan diferentes aspectos de los permisos y atributos del archivo o directorio. La cadena de permisos tiene el siguiente formato:

  - rwx rwx rwx

  Donde:

  El primer carácter indica el tipo de archivo (por ejemplo, '-' para archivos regulares, 'd' para directorios).

  Los siguientes tres caracteres representan los permisos del propietario del archivo. Los tres caracteres siguientes representan los permisos del grupo al que pertenece el archivo. Los últimos tres caracteres representan los permisos para otros usuarios.



0	000	- - -	No permissions
1	001	- - x	Only Execute
2	010	- w -	Only Write
3	011	- w x	Write and Execute
4	100	r - -	Only Read
5	101	r - x	Read and Execute
6	110	r w -	Read and Write
7	111	r w x	Read, Write and Execute

Bibliografía:

- Fleta, G., Sierra, D., Manuel, J. (2012). Sistemas Operativos Monopuestos. España: Macmillan education.

- Parhami, B. (2007). Arquitectura de computadoras. McGraw- Hill.

- Tocci, Ronald, J. y Widmer, N. S. (2017). Sistemas digitales. Principios y aplicaciones. Pearsons educación.

- Jimenez, J. A. (2015). Matemáticas para la computación. (3ª Edición). México: Alfaomega.

- Organización de computadoras. (2019) Módulo 5. Norma IEEE-754. Universidad Nacional del Sur. Disponible en: https://cs.uns.edu.ar/~ldm/mypage/data/oc/apuntes/2019-modulo5.pdf.

- The Institute of Electrical and Electronics Engineers. (2008). IEEE Standard for Floating-Point Arithmetic. New York: Institute of Electrical and Electronics Engineers.

- Douglas J. Faires y Richard L.(2002) Análisis numérico. (7ª ed.) México:Thomson learning.

- Williams, T. E. y Horowitz, M. (1986). SRT Division Diagrams and Their Usage in Designing Custom Integrated Circuits for Division. Stanford University.

- McBrien, S. (2023, January 12). Linux file permissions explained. Enable Sysadmin. https://www.redhat.com/sysadmin/linux-file-permissions-explained