Sistemas de Numeración

OdC - 2024

Sistemas de numeración

Nombre	Base	Símbolos	Ejemplo
Binario	2	0, 1	$(100110)_2$
Decimal	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9	$(38)_{10}$
Hexadecimal	16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F	$(26)_{16}$

Conversión hexadecimal - binario [1/2]

Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
0_{hex}	0000_{two}	4_{hex}	0100_{two}	8_{hex}	1000_{two}	c_{hex}	1100_{two}
1_{hex}	0001_{two}	5_{hex}	0101_{two}	9_{hex}	1001_{two}	d_{hex}	1101_{two}
2_{hex}	0010_{two}	6_{hex}	0110_{two}	a_{hex}	1010_{two}	e_{hex}	1110_{two}
3_{hex}	0011_{two}	7_{hex}	0111_{two}	b_{hex}	1011_{two}	f_{hex}	1111_{two}

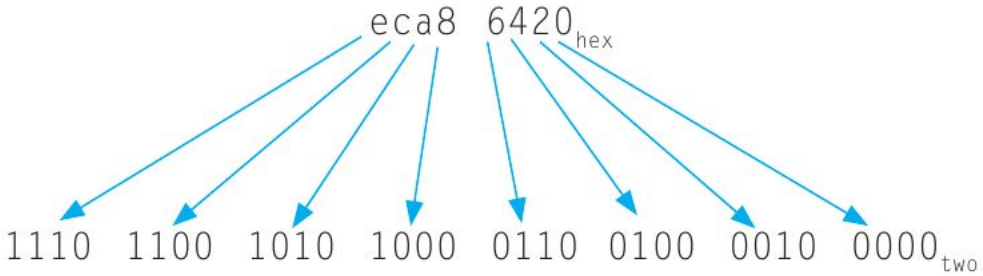
FIGURE 2.4 The hexadecimal–binary conversion table. Just replace one hexadecimal digit by the corresponding four binary digits, and vice versa. If the length of the binary number is not a multiple of 4, go from right to left.

- Convertir de **hexadecimal a binario** implica reemplazar cada uno de los dígitos hexadecimales por los cuatro dígitos binarios correspondientes.

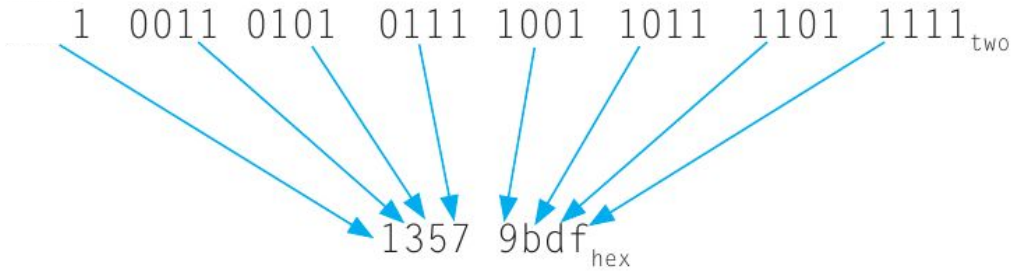
- Para convertir de **binario a hexadecimal**, primero debe subdividirse en grupos de 4 dígitos binarios (de derecha a izquierda) y reemplazar cada uno de estos grupos por un dígito hexadecimal.

Conversión hexadecimal - binario [2/2]

Ejemplo 1: Convertir el número $(ECA86420)_{16}$ a binario de 32 bits.



Ejemplo 2: Convertir el número $(10011010101111001101111011111)_2$ a hexadecimal



Conversión decimal - binario

Conversión de binario a decimal
El número decimal se obtiene de la sumatoria de todos los dígitos binarios multiplicado por el factor de 2^N, donde N en la posición del dígito binario en la secuencia, comenzando en 0 y creciendo de derecha a izquierda.

$$b_N*2^N + \ldots + b_3*2^3 + b_2*2^2 + b_1*2^1 + b_0*2^0 = num_{10}$$

Ejemplo 1-a: $(100101)_2 = 37$

$$1x2^5 + 0x2^4 + 0x2^3 + 1x2^2 + 0x2^1 + 1x2^0 =$$
$$32 + 0 + 0 + 4 + 0 + 1 = (37)_{10}$$

Conversión decimal - binario

Conversión de binario a decimal con parte fraccionaria

Parte entera: El número decimal se obtiene de la sumatoria de todos los dígitos binarios multiplicado por el factor de 2^N, donde N en la posición del dígito binario en la secuencia, comenzando en 0 y creciendo de derecha a izquierda.

Parte fraccionaria: Se obtiene de la sumatoria de todos los dígitos binarios multiplicado por el factor de 2^M, donde M en la posición del dígito binario en la secuencia, comenzando en -1 y cuyo módulo crece de izquierda a derecha .

$$b_N*2^N + \ldots + b_3*2^3 + b_2*2^2 + b_1*2^1 + b_0*2^0 + b_{-1}*2^{-1} + b_{-2}*2^{-2} + b_{-3}*2^{-3} + \ldots + b_{-M}*2^{-M} = num_{10}$$

Ejemplo 1-b: $(100101,0101)_2 = 37,3125$

Parte entera:

$1x2^5 + 0x2^4 + 0x2^3 + 1x2^2 + 0x2^1 + 1x2^0 =$

$32 + 0 + 0 + 4 + 0 + 1 = (37)_{10}$

Parte fraccionaria:

$0x2^{-1} + 1x2^{-2} + 0x2^{-3} + 1x2^{-4} =$

$0x(½) + 1x(¼) + 0x(⅛) + 1x(1/16) =$

$0 + 0,25 + 0 + 0,0625 = 0,3125$

Conversión decimal - binario

$$b_N*2^N + \dots + b_3*2^3 + b_2*2^2 + b_1*2^1 + b_0*2^0 + b_{-1}*2^{-1} + b_{-2}*2^{-2} + b_{-3}*2^{-3} + \dots + b_{-M}*2^{-M} = num_{10}$$

Ejercicio 2: $(10110011011011,11000010000)_2 = 11483,7578125$

Parte entera:

$$1x2^{13} + 0x2^{12} + 1x2^{11} + 1x2^{10} + 0x2^9 + 0x2^8 + 1x2^7 + 1x2^6 + 0x2^5 + 1x2^4 + 1x2^3 + 0x2^2 + 1x2^1 + 1x2^0 =$$
$$8192 + 0 + 2048 + 1024 + 0 + 0 + 128 + 64 + 0 + 16 + 8 + 0 + 2 + 1 = (11483)_{10}$$

Parte fraccionaria:

$$1x2^{-1} + 1x2^{-2} + 0x2^{-3} + 0x2^{-4} + 0x2^{-5} + 0x2^{-6} + 1x2^{-7} + 0x2^{-8} + 0x2^{-9} + 0x2^{-10} + 0x2^{-11} =$$
$$0,5 + 0,25 + 0 + 0 + 0 + 0 + 0,0078125 + 0 + 0 + 0 + 0 = 0,7578125$$

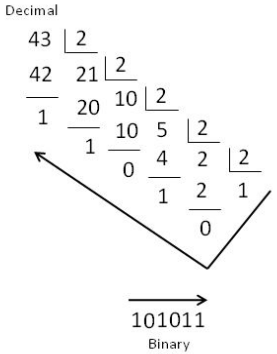
Conversión decimal - binario

Conversión de decimal a binario

Parte entera: Se divide la parte entera del número por dos, luego al resultado se lo vuelve a dividir por dos y así hasta que el resultado sea 0 o 1. Mediante los restos obtenidos de las divisiones consecutivas, se conforma el número en binario, siendo el primero en obtenerse el menos significativo.

Parte fraccionaria: Se multiplica la parte fraccionaria del número por dos, si el resultado es mayor a 1 el coeficiente obtenido es 1, se le resta uno y se vuelve a multiplicar, si es menor a 1 el coeficiente es 0 y simplemente se vuelve a multiplicar. Este procedimiento debe continuar hasta que el resultado sea cero.

Ejemplo: $(43{,}6875)_{10} = (101011{,}1011)_2$



	Entero		Fracción	Coeficiente
$0.6875 \times 2 =$	1	+	0.3750	$a_{-1} = 1$
$0.3750 \times 2 =$	0	+	0.7500	$a_{-2} = 0$
$0.7500 \times 2 =$	1	+	0.5000	$a_{-3} = 1$
$0.5000 \times 2 =$	1	+	0.0000	$a_{-4} = 1$

Por tanto, la respuesta es $(0.6875)_{10} = (0.a_{-1}a_{-2}a_{-3}a_{-4})_2 = (0.1011)_2$

Complemento a dos

Procedimiento para calcular el complemento a dos de un número:

1 - Agregar ceros hasta completar el registro y verificar que el número se puede representar.
2 - Negar bit a bit todos los dígitos del número
3 - Sumar 1 al resultado.

Ejemplo (en 8 bits): $(101101)_2$

Conversión de Decimales negativos - Binario complemento a dos [1/2]

Procedimiento para convertir decimales negativos a binario en complemento a dos:

1) Convertir el número decimal a binario (sin considerar, momentáneamente, que es negativo).
2) Completar con ceros a la izquierda la cantidad de bits en que se esté trabajando.
3) Aplicar el proceso de complemento a dos

Procedimiento para convertir binario en complemento a dos a decimal:

1) Aplicar el proceso de complemento a dos
2) Convertir el número binario a decimal y agregar el signo negativo

Rango: -2^{n-1} a $2^{n-1} - 1$

En los número negativos en complemento a dos, el bit más significativo (MSB) siempre es un 1 mientras que en los positivos es un cero.
En caso de necesitar extender el bit de signo, para los números negativos deben agregarse unos a la izquierda para no modificar el valor del mismo

Binario (Complemento a 2)	Decimal
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0
1111	−1
1110	−2
1101	−3
1100	−4
1011	−5
1010	−6

Binario (Complemento a 2)	Decimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	**−8**
1001	**−7**
1010	**−6**
1011	**−5**
1100	**-4**
1101	**-3**
1110	**-2**
1111	**-1**

Conversión de Decimales negativos - Binario complemento a dos [1/2]

Decimales negativos a binario en complemento a dos

Ejemplo en 16 bits: $(-1245)_{10}$

1) Convertir el número decimal a binario

$$(-1245)_{10} = (-10011011101)_{2}$$

2) Completar con ceros a la izquierda la cantidad de bits en que se esté trabajando.

$$(-1245)_{10} = (-0000010011011101)_{2}$$

3) Aplicar el proceso de complemento a dos

$$(-1245)_{10} = (1111101100100011)_{2}$$

Procedimiento para convertir binario en complemento a dos a decimal:

1) Aplicar el proceso de complemento a dos

$$(1111101100100011)_{2} \Rightarrow (-0000010011011101)_{2}$$

2) Convertir el número binario a decimal y agregar el signo negativo

$$(-0000010011011101)_{2} \Rightarrow (-1245)_{10}$$

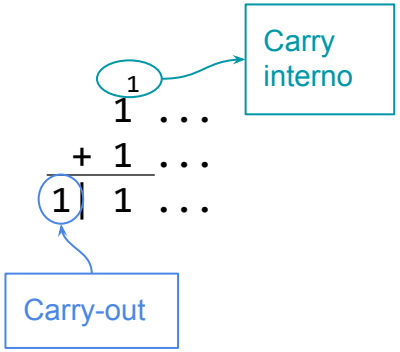
Suma binaria

Al sumar dos números se pueden dar las siguientes posibilidades:

```
  0
+ 0
---
  0
```

```
  1
+ 0
---
  1
```

```
  1
+ 1
---
1| 0
```



Suma y resta binaria

Considerar que los registros A y B, contienen los valores 0x1A y 0x3F. Encontrar el contenido de C si a) se suman los registros, b) se restan. Asumir que se está trabajando con valores signados.

a) Suma

```
  00011010     26
+ 00111111   + 63
```

b) Resta

```
  00011010     26
- 00111111   - 63
             - 37
```

Complemento a dos

```
  00011010       26
+ 11000001   + (-63)
  11011011     - 37
```

Complemento a dos:
1) Completar a la cantidad de bits necesarios.
2) Negar todos los bits:
00111111 => 11000000
3)Sumar 1:

```
    11000000
  +        1
    11000001
```

Suma y resta binaria

Ejercicio 6: Suponga que los registros A y B del microprocesador del ejercicio 4 (registros de 8 bits) contienen los valores **0x4B** y **0x24** respectivamente.

a) ¿Qué valor contiene el registro C después de ejecutar la operación C = A + B?
¿El resultado que se guarda en C es el esperado?

b) ¿Qué valor contiene el registro C después de ejecutar la operación C = A - B?
¿El resultado que se guarda en C es el esperado?

c) En base al análisis de las operaciones anteriores, ¿cuál es la ventaja de la representación de números negativos mediante su complemento a 2, por sobre la representación binaria regular + un bit de signo?

Suma y resta binaria

a) Suma

```
    1
  10101011
+ 00100100
----------
  11001111
```

b) Resta

El complemento a dos de $(00100100)_2$ es $(11011100)_2$

```
    1 1 1 1
  10101011
+ 11011100
----------
1|10000111
```

Suma y resta binaria

¿Los resultados obtenidos son los esperados?

Si Consideramos que los números son **no signados**:

$10101011_2 \Rightarrow 171_{10}$

$00100100_2 \Rightarrow 36_{10}$

a)

El resultado esperado es $(171)_{10} + (36)_{10} = (207)_{10}$

El resultado obtenido es $(11001111)_2 = (207)_{10}$

b)

El resultado esperado es $(171)_{10} - (36)_{10} = (135)_{10}$

El resultado obtenido es $(10000111)_2 = (135)_{10}$

Suma y resta binaria

¿Los resultados obtenidos son los esperados?

Si Consideramos que los números son **signados**:

$10101011_2 \Rightarrow -85_{10}$

$00100100_2 \Rightarrow 36_{10}$

a)

El resultado esperado es $(-85)_{10} + (36)_{10} = (-49)_{10}$

El resultado obtenido es $(11001111)_2 = (-49)_{10}$

b)

El resultado esperado es $(-85)_{10} - (36)_{10} = (-121)_{10}$

El resultado obtenido es $(10000111)_2 = (-121)_{10}$

Suma y resta binaria

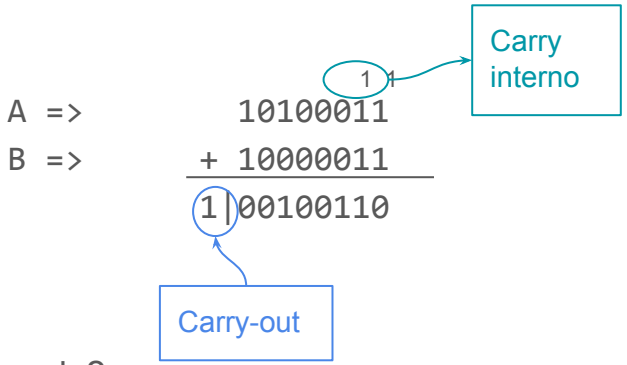
c) En base al análisis de las operaciones anteriores, ¿cuál es la ventaja de la representación de números negativos mediante su complemento a 2, por sobre la representación binaria regular + un bit de signo?

El complemento a dos tiene las siguientes ventajas:

- Sin importar si el valor almacenado en el registro es signado o no, el proceso de suma y resta es exactamente igual.
- El proceso para convertir un número a complemento a dos es computacionalmente sencillo (se requiere un negador bit a bit y un sumador)
- Es posible realizar la resta con el mismo proceso de suma, solo hay que aplicar el complemento a dos del segundo operando antes de realizar la suma.

Carry

Si ahora los registros de 8 bits contienen los siguientes valores: A = $(A3)_{16}$ y B = $(83)_{16}$



¿El resultado que se guarda en C es el esperado?

El resultado esperado es $(160)_{10} + (131)_{10} = (291)_{10}$
El resultado obtenido es $(00100011)_{2} = (35)_{10}$

Se denomina **Carry** (carry-out) de una operación al carry de la suma o resta de los bits más significativos. Este valor **no** se almacena en el registro resultado porque se necesitaria mas bits de los disponibles.

Overflow

Considere que ahora A y B ahora son números signados y que A = $(63)_{16}$ y B = $(43)_{16}$

```
A =>        01100011      99
B =>      + 01000011      67
          ------------  -----
          0|10100110     166
```

¿El resultado que se guarda en C es el esperado?

El resultado esperado es $(99)_{10} + (67)_{10} = (166)_{10}$
El resultado obtenido es $(10100110)_{2} = (-91)_{10}$

Existe overflow cuando se realizan operaciones signadas y el signo del resultado es incorrecto.

Carry no siempre es Overflow

Supongamos el siguiente ejemplo de números signados de 8 bits:

```
  1 1 1 1 1
  11111110        -2
+ 11111111    + -1
----------    -----
1|11111101       -3
```

¿El resultado que se guarda en C es el esperado?

El resultado esperado es $(-2)_{10} + (-1)_{10} = (-3)_{10}$
El resultado obtenido, sin considerar el carry, es $(11111101)_2 = (-3)_{10}$

Es decir, el resultado obtenido es correcto, pero se obtuvo carry en la operación.

Carry vs overflow

Nos interesa el carry cuando operamos con números no signados. Existe carry cuando el resultado no puede representarse con la cantidad de bits disponibles.

En cambio, nos interesa el overflow cuando se realizan operaciones signadas y el signo del resultado es incorrecto:

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

FIGURE 3.2 Overflow conditions for addition and subtraction.

Punto flotante (estándar IEEE754)

La representación de los números en punto flotante son una forma de notación científica, y por lo tanto, la posición del punto (coma en español) decimal está determinado por el exponente al que se eleva la base.
Por ejemplo:

- En base diez: $35,2 \times 10^{-5}$
- En base dos: $- 1101001 \times 2^{3}$

A partir de esto, se pueden identificar tres partes: **signo**, **exponente** y **fracción (o parte significativa)**. El estándar IEEE754 establece las características de estos tres elementos, como cantidad de bits, método de almacenar valores negativos, etc.

Dentro del estándar existen diferentes variaciones, en la materia utilizamos la denominada de “precisión simple” que utiliza 32 bits.

En primer lugar, a la notación científica se le realizan algunas modificaciones, quedando de la siguiente forma:

$$(-1)^{BS} \times (1 + \text{fracción}) \times 2^{\text{exponente} - \text{bias}}$$

Donde se resaltan las tres partes antes mencionadas. Los 32 bits disponibles se pueden dividir en estas partes con la siguiente distribución:

- Bit 31: Bit de Signo (BS).
- Bit 30 - 23: Exponente.
- Bit 22 - 0: Fracción.

Punto flotante (estándar IEEE754)

Bit de Signo
Establece el signo del número. Cuando es cero implica que el número positivo, en el caso en que sea 1, es negativo.

Exponente

Indica cuántos “lugares” se debe desplazar hacia la derecha o hacia la izquierda la coma binaria de la parte significativa. El exponente de un número puede ser tanto positivo como negativo.

En la estandarización se decidió agregar un *bias* que me permite tener valores positivos y negativos sin dificultar aún más el proceso de conversión y la lógica necesaria para realizar operaciones aritméticas.

En el caso de los flotantes de 32 bits, el *bias* es 127. En la siguiente tabla se muestran algunos ejemplos.

Exponente	Exponente + Bias
...	...
-3	124
-2	125
-1	126
0	127
1	128
2	129
3	130
...	....

Fracción
La parte fraccionaria del número debe ser **normalizada**. Esto implica mover la coma tantos lugares como sea necesario para que el número resultante quede expresado como un uno seguido por la parte fraccionaria:

$$1,xxxxxxxxx \ 2^{yyy}$$

Es importante notar que se debe adaptar el exponente para que el valor no se modifique.

Punto flotante (estándar IEEE754)

Procedimiento decimal => flotante (IEEE 754 de 32 bits):

1) Encontrar el bit de signo
2) Pasar el número a binario y normalizar
3) Sumar el *bias* al exponente y convertirlo a binario
4) Encontrar la parte fraccionaria
5) Conformar el número

Procedimiento flotante (IEEE 754 de 32 bits) => decimal:

1) Dividir los el conjunto de bits en las tres partes respetando el formato
2) Encontrar el bit de signo
3) Encontrar el exponente
4) Desnormalizar el número y pasar a decimal

Punto flotante (estándar IEEE754)

Ejemplo 1: Convertir de decimal a flotante IEEE 754 de 32 bits el número 263,3

1) Encontrar el bit de signo: Como 263.3 es positivo, el bit de signo es 0

2) Pasar el número a binario y normalizar

$$263,3 = (100000111,01\overline{001} \times 2^{0})_{2} \text{(Sin normalizar)}$$

$$263,3 = (1,00000111010011001\overline{1001}... \times 2^{8})_{2} \text{ (Normalizado)}$$

(**Aclaración:** el número elegido es períodico, por lo tanto, se agregaran todos los decimales que permitan la cantidad de bits)

Debido a la normalización, siempre el primer bit será un uno. Entonces, no es necesario almacenarlo. De esta forma ganamos un bit más de precisión.

3) Sumar el *bias* al exponente y convertirlo a binario

$$127 + 8 = 135_{10} = (10000111)_{2}$$

4) Encontrar la parte fraccionaria

La parte fraccionaria se compone por los 23 bits que siguen a la coma decimal del número normalizado. En caso de que el número necesite más bits de precisión, deberá ser truncado. En este caso, queda:

$$(00000111010011001100110)_{2}$$

5) Conformar el número

Respetando el formato, se une el bit de signo con el exponente y la parte fraccionaria:

$$(01000011100000111010011001100110)_{2}$$

Punto flotante (estándar IEEE754)

Convertir el número obtenido en el punto anterior de flotante IEEE 754 de 32 bits a decimal.

1) Dividir los el conjunto de bits en las tres partes respetando el formato
El bit más significativo es el bit de signo, los bits 30 a 23 son el exponente y del 22 al 0 la parte fraccionaria.

$$0 \quad 10000111 \quad 00000111010011001100110_2$$

2) Encontrar el bit de signo
Como el bit de signo es 0, el número es positivo.

3) Encontrar el exponente
Esto implica pasar a decimal la parte del exponente, y luego restarle el *bias.*

$$10000111_2 = 135_{10}$$
$$135 - 127 = 8$$

4) Desnormalizar el número y pasar a decimal
Esto implica agregar el uno que se elimina al normalizar y luego desplazar la coma tantos lugares (y en el sentido) como indique el exponente

Parte fraccionaria: $00000111010011001100110_2 \Rightarrow (1,00000111010011001100110 \times 2^8)_2$

Desnormalización: $100000111,010011001100110_2 = 263,299987793$