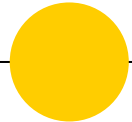


Sistemas Embebidos

1. Representación Numérica



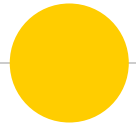
Juan Ignacio Causse
juan.causse@ort.edu.ar



Contenido de la unidad

Unidad 1: Representación numérica y alfabética en binario.

- Sistema de numeración en base dos.
- Abreviación en hexadecimal de binarios.
- Números enteros.
 - Representaciones no signadas.
 - Representaciones signadas.
 - Signo Magnitud (con bit de signo).
 - Complemento a la base (complemento a dos).
 - Aritmética binaria con enteros signados con complemento a dos.
 - Detección de overflow.
- Números fraccionarios.
 - Representación de números fraccionarios con punto fijo y bit de signo.
 - Estándar IEEE 754 de 32 bits (punto flotante).
 - Conversión de números fraccionarios en base decimal desde y hacia IEEE 754 de 32 bits.
 - Casos especiales del estándar IEEE 754.
 - Comparación de números en punto flotante.
- Código ASCII.
 - C-Strings (cadenas de caracteres) null-terminated.
 - Conversiones mayúsculas ↔ minúsculas y de valores numéricos a dígitos ASCII.



Numeración binaria



Numeración binaria

Sistema de numeración pesado: cada dígito recibe un **peso según su posición** en el número representado. Dicho peso equivale a la **base** (la cantidad de símbolos del sistema) **elevada a una potencia entera**.

Al dígito de más a la derecha de la parte entera del número se lo denomina **dígito menos significativo** y tiene peso $(base)^0 = 1$.

Luego **hacia la izquierda los exponentes aumentan**, y **hacia la derecha** del punto decimal (parte fraccionaria), **disminuyen**, teniendo exponentes negativos.



Numeración binaria

Ejemplo en octal (base 8) con el número 1325.72_8 :

Dígito:	1	3	2	5	.	7	2
Exponente:	3	2	1	0		-1	-2

Para conocer el valor en decimal del número, basta hacer:

$$1 \times 8^3 + 3 \times 8^2 + 2 \times 8^1 + 5 \times 8^0 + 7 \times 8^{-1} + 2 \times 8^{-2} = 725.90625$$



Numeración binaria

En binario, la base es 2 (símbolos posibles: 0 y 1), y a un dígito binario se lo llama **bit** (del inglés: *binary digit*).

Además:

- Un conjunto de 8 bits se llama **byte**.
- Un conjunto de 4 bits se llama **nibble**.
- Al bit de menor peso de un conjunto se lo nota **LSb** (*less significative bit*).
- Al bit de mayor peso de un conjunto se lo nota **MSb** (*most significative bit*).
- Al byte de menor peso de un conjunto se lo nota **LSB** (*less significative byte*).
- Al byte de mayor peso de un conjunto se lo nota **MSB** (*most significative byte*).



Numeración binaria

Ejemplo en binario (base 2) con el número 10011010_2 :

	MSb							LSb
Dígito:	1	0	0	1	1	0	1	0
Exponente:	7	6	5	4	3	2	1	0

Para conocer el valor en decimal del número, basta hacer:

$$1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 =$$

$$1 \times 2^7 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^1 = 128 + 16 + 8 + 2 = 154$$



Abreviación hexadecimal

La base hexadecimal tiene 16 símbolos:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Donde: $A_H = 10_{10}$, $B_H = 11_{10}$, $C_H = 12_{10}$, $D_H = 13_{10}$, $E_H = 14_{10}$, $F_H = 15_{10}$.

Entonces, podemos **abreviar un número en binario tomándolo de a nibbles**, y **reemplazando cada nibble por su equivalente en base hexadecimal**. Con 4 bits, la cantidad de combinaciones que podemos formar es 16 ($16 = 2^4$).

Dígito:	1	1	1	1
Peso:	8	4	2	1



Abreviación hexadecimal

Por ejemplo, si tenemos el binario:

11010001110100101110101001

Lo separamos por nibbles, **comenzando desde el menos significativo**. Si algún nibble quedara incompleto, **se lo completa con ceros**:

0011 0100 0111 0100 1011 1010 1001

Y finalmente se hace la conversión a hexadecimal:

0011	0100	0111	0100	1011	1010	1001
3	4	7	4	B	A	9

Suele utilizarse el prefijo 0x para indicar que el número está expresado en base hexadecimal



Ejercicio 1

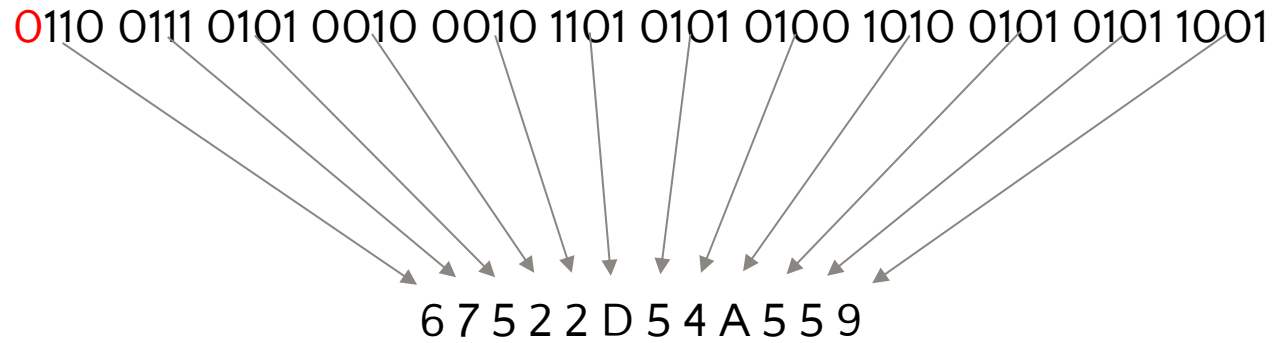


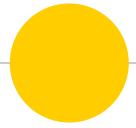
Convertir el siguiente número en binario a hexadecimal:

11001110101001000101101010101001010010101011001



Solución






Números enteros



Números enteros

Imaginemos que queremos representar el número 87, expresado en base decimal, en binario. Esto se logra realizando **divisiones enteras sucesivas por dos**, hasta que no sea posible seguir dividiendo:

$$\begin{array}{l} 87 / 2 = 43, \text{ resto } 1 \\ 43 / 2 = 21, \text{ resto } 1 \\ 21 / 2 = 10, \text{ resto } 1 \\ 10 / 2 = 5, \text{ resto } 0 \\ 5 / 2 = 2, \text{ resto } 1 \\ 2 / 2 = 1, \text{ resto } 0 \end{array}$$



$$87_{10} = 1010111_2$$

Finalmente, formamos el número tomando como **MSb** el último cociente, y luego obteniendo los **bits sucesivos desde el último resto hasta el primero**.

En la diapositiva 8 vimos cómo hacer la conversión inversa.



Números enteros no signados

En matemática, cuando necesitamos representar un número negativo, utilizamos el signo **—**. Ahora bien, las computadoras trabajan con señales digitales, donde un pulso alto representa un 1, y un pulso bajo un 0, por lo tanto, **contamos únicamente con ceros y unos** para representar cualquier información.

Con lo que vimos hasta ahora, solo podemos representar **números positivos**, donde el **mínimo representable es el cero** (000...0), y el **máximo representable depende de la cantidad de bits utilizados** (111...1).

Cuando trabajamos con un número binario **no signado**, estamos trabajando con **enteros positivos**, como hasta ahora.



Números enteros signados

Las dos maneras más comunes de **representar números negativos** son las siguientes:

- Mediante la utilización de un bit dedicado al signo (**signo-magnitud**).
- Utilizando el **complemento a la base**, en este caso, complemento a dos.



Signo-Magnitud

Se utiliza un bit, **el MSb**, para representar el signo: **0 si el número es positivo, 1 si el número es negativo**.

Anteriormente, la sencillez propia de las representaciones no signadas hacía que pudiéramos representar números con cualquier cantidad de bits.

Ahora, **resulta necesario definir una cantidad de bits fija para nuestras representaciones**. Esto, por supuesto, establece **límites a los números que son representables**.

En microcontroladores y microprocesadores, la **arquitectura** es la que **define la cantidad de bits** que se utilizan. La arquitectura de los procesadores comerciales **actuales** (Intel, AMD) suele ser de **64 bits**.



Signo-Magnitud

Sin perder generalidad, simplificaremos los cálculos considerando una arquitectura de 8 bits.

En este caso, el bit 7 (**MSb**) representa el signo, mientras que **los demás bits** (6 al 0) **representan el valor absoluto del número**.

Por ejemplo, el número -37 se representa como:

S	64	32	16	8	4	2	1
1	0	1	0	0	1	0	1



Signo-Magnitud

En Signo-Magnitud de n bits podemos, entonces, **representar números enteros en el intervalo $[-2^{n-1}+1 ; 2^{n-1}-1]$** . Para 8 bits, el intervalo es $[-127 ; 127]$.

Este sistema, si bien puede parecer adecuado tiene **dos grandes problemas**:

- **Doble representación del cero** (0000 0000 y 1000 0000).
- **Las operaciones aritméticas son más complejas**: para sumar dos números, por ejemplo se necesita primero verificar que ambos sean positivos o ambos sean negativos, y en caso de que sean de signos distintos, se debe comparar sus valores absolutos para restar el menor al mayor de ellos.



Ejercicio 2



Representar los números 58 y -107 en sistema Signo-Magnitud de 8 bits.



Solución

S	64	32	16	8	4	2	1	HEX
0	0	1	1	1	0	1	0	0x3A

S	64	32	16	8	4	2	1	HEX
1	1	1	0	1	0	1	1	0xEB



Ejercicio 3



Indicar, sin hacer la conversión, el signo de los siguientes números expresados en Signo-Magnitud en base hexadecimal. Luego, realizar la conversión a decimal.

- ☐ 0xF0
- ☐ 0xB9
- ☐ 0x5D



Solución

- ⦿ Negativo, -112
- ⦿ Negativo, -57
- ⦿ Positivo, 93



Complemento a la Base

La representación signada utilizando el complemento a la base permite solucionar los dos problemas que existían en el sistema Signo-Magnitud:

- **El cero se representa de forma unívoca:** $000\dots 0$ ($0x00\dots 0$).
- Como veremos más adelante, el sistema de Complemento a la Base **permite realizar operaciones algebraicas fácilmente**, sin la necesidad de realizar conversiones ni comparaciones.

En este sistema, **el primer dígito sigue teniendo dos posibles valores, independientemente de en qué base se trabaje: 0 en caso de que el número sea positivo, y 1 si fuese negativo**. Ahora bien, exceptuando esta coincidencia, el sistema de Complemento a la Base tiene un funcionamiento completamente distinto al de Signo-Magnitud.



Complemento a dos

Cuando trabajamos en binario, como la base es dos, se suele llamar al sistema de Complemento a la Base simplemente como “**complemento a dos**”.

Este sistema permite **representar números enteros en el intervalo $[-2^{n-1} ; 2^{n-1}-1]$** . Para 8 bits, el intervalo es $[-128 ; 127]$.

Debemos notar que, debido a que no existe la doble representación del cero, en el intervalo negativo podemos representar un número más que en Signo-Magnitud.



Complemento a dos

Para obtener el complemento a dos de un número, primero **expresamos el valor absoluto de dicho número en binario**, como si se tratase de un número no signado. Por lo tanto, **el MSb valdrá, inicialmente, cero**.

Si el número que queremos representar es positivo, queda así, ya que **la representación en Complemento a la Base para números positivos coincide con la representación de dicho número en un sistema no signado**.

Por ejemplo, el número 82 se representa como 0101 0010.

El número 127, el mayor en 8 bits, se representa como 0111 1111.



Complemento a dos

Si el número que queremos representar es **negativo**, el siguiente paso es obtener el complemento del binario. Para ello, **hallamos el llamado “complemento a uno” del número, que consiste en invertir todos sus bits, y luego le sumamos 1**. Así, habremos obtenido el complemento a dos de dicho número negativo.

Por ejemplo, para representar el -82:

No signado	0	1	0	1	0	0	1	0
Complemento a 1	1	0	1	0	1	1	0	1
Complemento a 2	1	0	1	0	1	1	1	0



Complemento a dos

Para poder **obtener el número representado en complemento a dos** de K bits, aplicamos la fórmula:

$$-(2^{K-1}) \times b_{K-1} + 2^{K-2} \times b_{K-2} + \dots + 2^2 \times b_2 + 2^1 \times b_1 + 2^0 \times b_0$$

Donde b_i representa al bit en la posición i.

Al calcular **el complemento a dos de un número** negativo (que ya estaba **expresado como complemento**), se **obtiene el valor absoluto** del mismo.



Ejercicio 4

Expresar los números -1 , $+127$, -64 , -8 , y -128 en binario utilizando el sistema de complementos, con 8 bits.



Solución

- -1: 0000 0001 → 1111 1110 → 1111 1111
- 127: 0111 1111
- -64: 0100 0000 → 1011 1111 → 1100 0000
- -8: 0000 1000 → 1111 0111 → 1111 1000
- -128: 1000 0000



Ejercicio 5



Indicar qué números representan los siguientes bytes sabiendo que están expresados en binario complemento a 2 de 8 bits:

- ☐ 0011 1011
- ☐ 0110 0001
- ☐ 1010 0101
- ☐ 1101 0100
- ☐ 1111 0000



Solución

- ☐ 59
- ☐ 97
- ☐ -91
- ☐ -44
- ☐ -16



Aritmética en binario con complemento a la base

El método para sumar números en binario es el mismo que usamos para sumar números en base decimal, el que aprendemos en la primaria.

Las principales **diferencias** radican en que:

- Al contar únicamente con dos símbolos, $1_2 + 1_2 = 10_2$, por lo que **coloco un cero en esa columna y “me llevo uno”**.
- Para realizar una **resta**, debemos **sumar el opuesto del sustraendo**. Ejemplos: hacer $2-3$ es igual a hacer $2+(-3)$, $-4-5 = -4+(-5)$. Recordar que **el opuesto de un número en binario es el complemento a dos** del mismo.
- Siempre debemos **tener en cuenta la cantidad de bits** con la que estamos trabajando. **Si el resultado fuese mayor que el máximo o menor que el mínimo representable con dicha cantidad de bits**, ocurrirá un error conocido como **overflow**.



Aritmética en binario con complemento a la base

$$\begin{array}{r} 7 \\ + 18 \\ \hline 25 \end{array}$$

$$\begin{array}{r} 11 \\ 0000 \ 0111 \\ + 0001 \ 0010 \\ \hline 0001 \ 1001 \end{array}$$



Aritmética en binario con complemento a la base

$$\begin{array}{r} 64 \\ + -65 \\ \hline -1 \end{array}$$

$$\begin{array}{r} 0100 \ 0000 \\ + 1011 \ 1111 \\ \hline 1111 \ 1111 \end{array}$$



● Aritmética en binario con complemento a la base

$$\begin{array}{r} -17 \\ + 22 \\ \hline 5 \end{array}$$

								Carry Out	Carry In	
								1 1 1 1	1 1	
								1 1 1 0	1 1 1 1	
								+ 0 0 0 1	0 1 1 0	
								<hr/>		
								1	0 0 0 0	0 1 0 1
								Carry		Dígito 4

Notar que, en la última columna, nos llevamos 1. Al ser el **último bit** en la suma, se dice que ocurrió un **acarreo** o **carry**.

Además, cada dígito tiene su **Carry In** y su **Carry Out**.



Negación

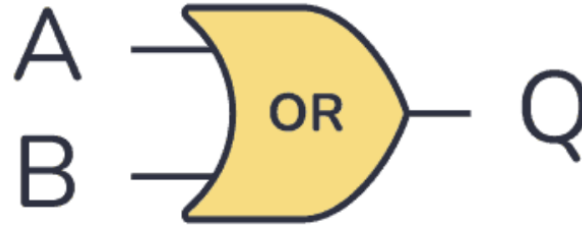


A	Q
0	1
1	0

NOT: $Q = \neg A$



Suma o disyunción

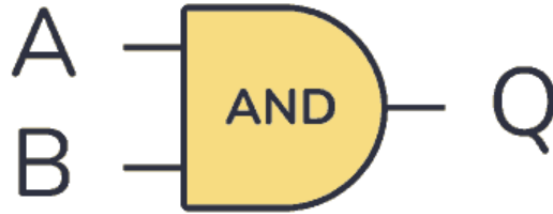


A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

OR: $Q = A + B$



Producto o conjunción



A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

AND: $Q = A \times B$



Suma exclusiva o disyunción exclusiva

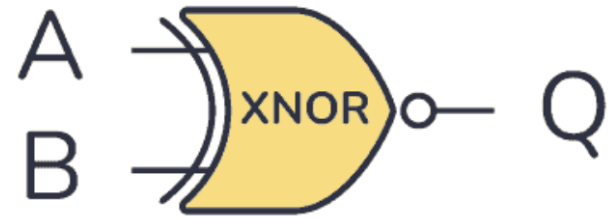
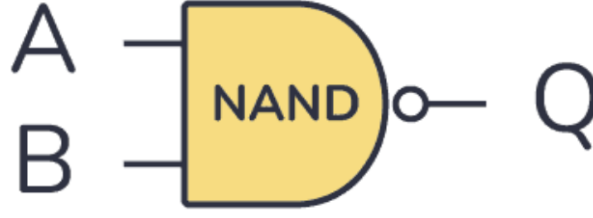
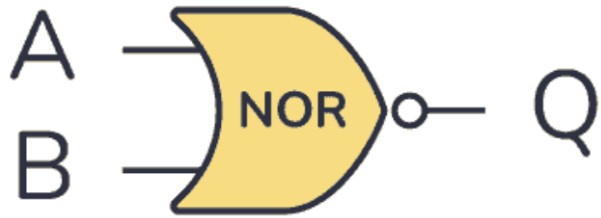


A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

XOR: $Q = A \oplus B$



Compuertas lógicas negadas



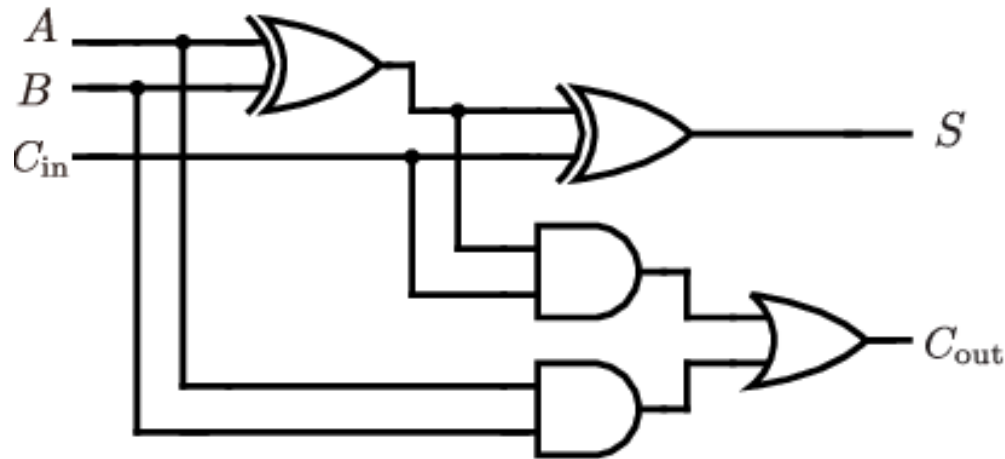
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

A	B	Q
0	0	1
0	1	0
1	0	0
1	1	1



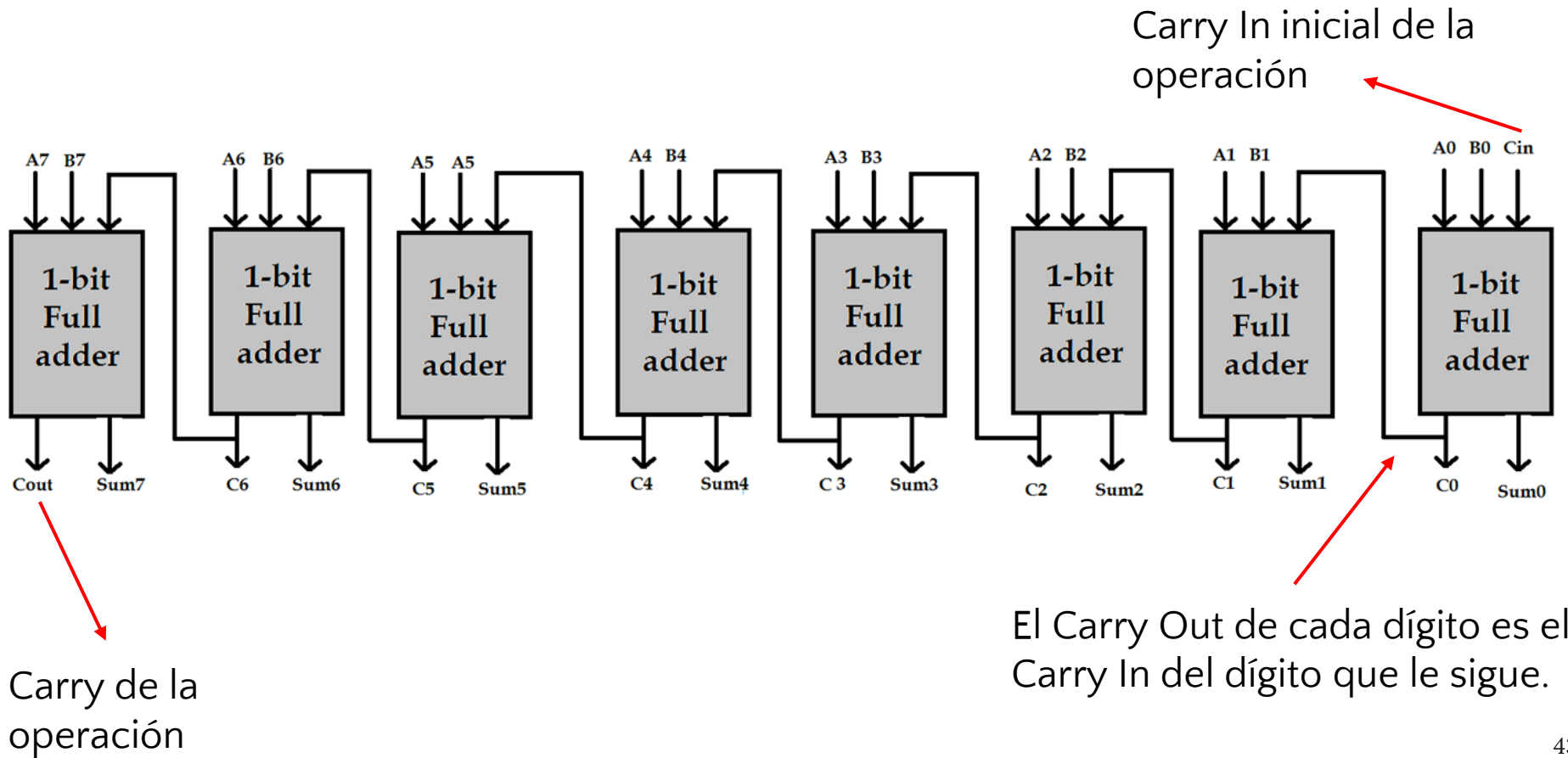
1-bit Full-Adder



Inputs			Outputs	
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



8-bit Full-Adder





Detección de Overflow

Para detectar si hubo overflow en una suma de n bits, podemos hacer la operación $CIn_{n-1} \oplus COut_{n-1}$. Si el resultado es 1, hubo overflow, si el resultado es 0, no hubo overflow.

$$\begin{array}{r} 124 \\ + 3 \\ \hline 127 \end{array}$$

$$\begin{array}{r} 0111 \ 1100 \\ + 0000 \ 0011 \\ \hline 0111 \ 1111 \end{array}$$

$$CIn_7 = 0$$

$$COut_7 = 0$$

$0 \oplus 0 = 0$: No hubo overflow

$$\begin{array}{r} 124 \\ + 4 \\ \hline 128 \end{array}$$

$$\begin{array}{r} 1111 \ 1 \\ 0111 \ 1100 \\ + 0000 \ 0100 \\ \hline 1000 \ 0000 \end{array}$$

$$CIn_7 = 1$$

$$COut_7 = 0$$

$1 \oplus 0 = 1$: Hubo overflow



Detección de Overflow

Recordar que se utiliza una **compuerta XOR**. Por lo tanto, en esta operación vista en un ejemplo anterior, no hay overflow.

$$\begin{array}{r} -17 \\ + 22 \\ \hline 5 \end{array}$$

$$\begin{array}{r} 1111 11 \\ 1110 1111 \\ + 0001 0110 \\ \hline 1 0000 0101 \end{array}$$

$$CIn_7 = 1$$

$$COut_7 = 1$$

$$1 \oplus 1 = 0: \text{No hubo overflow}$$



Ejercicio 6



Realizar, si es posible, las siguientes operaciones en binario complemento a la base de 8 bits. Indicar si hubo carry y si hubo overflow.

- ⦿ $-98 + 127$
- ⦿ $128 + 34$
- ⦿ $126 + 34$
- ⦿ $-152 - 37$
- ⦿ $-120 - 35$



Solución

Los ítems 2 y 4 no pueden realizarse ya que 128 y -152 no pueden expresarse en 8 bits signado.

1

$$\begin{array}{r} -98 \\ + 127 \\ \hline 29 \end{array}$$

$$\begin{array}{r} 1111 \ 11 \\ 1001 \ 1110 \\ + 0111 \ 1111 \\ \hline \end{array}$$

1 0001 1101

3

$$\begin{array}{r} 126 \\ + 34 \\ \hline 160 \end{array}$$

$$\begin{array}{r} 1111 \ 11 \\ 0111 \ 1110 \\ + 0010 \ 0010 \\ \hline \end{array}$$

1010 0000

5

$$\begin{array}{r} -120 \\ + -35 \\ \hline -155 \end{array}$$

$$\begin{array}{r} 11 \\ 1000 \ 1000 \\ + 1101 \ 1101 \\ \hline \end{array}$$

1 0110 0101

- 1) Carry: 1, Overflow: 0
- 3) Carry: 0, Overflow: 1
- 5) Carry: 1, Overflow: 1



Ejercicio 7



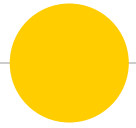
Indicar los operandos de la siguiente operación, si tiene carry, e indicar la validez del resultado.

$$\begin{array}{r} 1111 \ 1111 \\ + 1000 \ 0000 \\ \hline 1 \ 0111 \ 1111 \end{array}$$



Solución

La operación que se hace es $-1 + (-128)$, tiene carry, y tiene overflow, ya que el resultado es inválido. Ver que el último bit tiene $C_{In} = 0$ y $C_{Out} = 1$, y $0 \oplus 1 = 1$.



Representación alfabética



Representación alfabética

Para **representar caracteres alfabéticos**, se necesitó crear un **código** que permitiera **asociar los distintos caracteres** que se utilizan a ciertas secuencias de bits, **números enteros en binario**.

El más utilizado actualmente es UTF-8, que permite la codificación de todos los caracteres del estándar Unicode.

Debido a su **soporte en el lenguaje C**, en esta materia veremos el **código ASCII** (American Standard Code for Information Interchange). Este código representa **cada carácter** con de **8 bits**, de los cuales **utiliza los 7 menos significativos, y el MSb es cero**.

Existe además el código ASCII extendido, que agrega más caracteres utilizando el octavo bit.



Código ASCII

El código ASCII consta de:

- ⦿ **Caracteres de control** (no imprimibles): Caracteres de **uso interno de la computadora**, que se utilizan para el **control de dispositivos**, para **controlar la impresión de texto**, o para indicar **valores nulos (NULL - 0x00 - \0)**.
- ⦿ **Caracteres imprimibles**: Son los **caracteres del alfabeto inglés**, los que podemos visualizar (incluyendo el espacio). Incluyen además los **números, signos de puntuación**, etc.
No encontraremos en ASCII las vocales acentuadas, la ñ, etc.



Tabla de caracteres ASCII

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]



Strings en C (cadenas de texto)

En C, **no tenemos String como tipo de dato** como en otros lenguajes (ej: Java, Python, C++). En su ausencia, **utilizamos** los denominados C-Strings, **Cadenas de Texto**.

Consisten en **una serie de bytes, contiguos en memoria**, donde **cada byte almacena un carácter ASCII**.

Por convención, **se agrega al final** el carácter de control **“NUL”** o **“NULL” (0000 0000)**. Este carácter especial recibe el nombre de **“Null terminator”**, y es necesario ya que de no utilizarse no habría una forma de **indicar el final de una cadena** (tener el número de caracteres por separado complejiza el String).

En el código, representamos al terminador como `'\0'`.



Ejercicio 11



En la siguiente tabla se tiene la representación en memoria de un String de C. Cada casillero representa una posición distinta en la memoria y todas son contiguas. Convertir a texto el mensaje que comienza en el primer casillero. Recordar que la cadena termina cuando se encuentra un terminador.

43	20	65	73	20	67	65	6E	69	61	6C	00	1A	98	FF
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----



Solución

“C es genial”, 0x00 es el terminador, y las posiciones de memoria que le siguen no son parte de la cadena.



Ejercicio 12



Observando la tabla ASCII, encontrar:

- Una fórmula que, dado un número del 0 al 9, permita encontrar el valor numérico del carácter ASCII que le corresponde.
- Una fórmula que, dado un valor numérico que representa una letra minúscula en ASCII, nos devuelva el valor numérico que representa a la misma letra pero en mayúscula.
- Una fórmula que, dado un valor numérico que representa una letra mayúscula en ASCII, nos devuelva el valor numérico que representa a la misma letra pero en minúscula.



Solución

- Conversión de número a ASCII:

$c = n + 48$ (donde c es el carácter ASCII y n el número).

- Conversión de minúscula a mayúscula:

$M = (m - 97) + 65$ (M = mayúscula, m = minúscula).

- Conversión de mayúscula a minúscula:

$m = (M - 65) + 97$ (M = mayúscula, m = minúscula).