Bits, bytes, y enteros

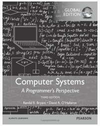
Organización del computador - FIUBA

2.do cuatrimestre de 2023

Última modificación: Mon Aug 28 11:38:20 2023 -0300

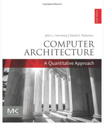
Créditos

Para armar las presentaciones del curso utilizamos:



R. E. Bryant and D. R. O'Hallaron, *Computer systems: a programmer's perspective*, Third edition, Global edition. Boston Columbus Hoboken Indianapolis New York San Francisco Cape Town: Pearson, 2015.

D. A. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/software interface*, RISC-V edition. Cambridge, Massachusetts: Morgan Kaufmann Publishers, an imprint of Elsevier, 2017.



J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. 2017.

El contenido de los slides está basado en las presentaciones de Patricio Moreno y de Organización del Computador I - FCEN.

Tabla de contenidos

Tabla de contenidos

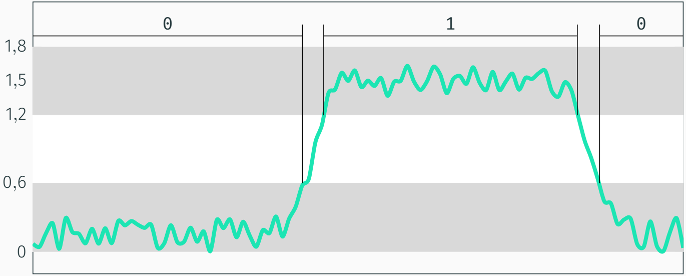
Tabla de contenidos

Todo está compuesto por bits

- Cada bit es `0` o es `1`
- Al codificar/interpretar los bits de distintas maneras...
  - las computadoras determinan qué hacer (instrucciones)
  - representan y manipulan numeros, caracteres, cadenas, etc.
- ¿por qué bits? Por la electrónica subyacente
  - son fáciles de almacenar en elementos biestables
  - se transmiten de manera confiable

Todo está compuesto por bits

- Cada bit es `0` o es `1`
- Al codificar/interpretar los bits de distintas maneras...
  - las computadoras determinan qué hacer (instrucciones)
  - representan y manipulan numeros, caracteres, cadenas, etc.
- ¿por qué bits? Por la electrónica subyacente
  - son fáciles de almacenar en elementos biestables
  - se transmiten de manera confiable



Álgebra de Boole

Desarrollada por George Boole en el siglo 19

- Representación algebraica de la lógica
- Codifica los valores de verdad “Verdadero” como 1 y “Falso” como 0

AND

A&B = 1 cuando A = 1 y B = 1

&	0	1
0	0	0
1	0	1

OR

A | B = 1 cuando A = 1 o B = 1

|	0	1
0	0	1
1	1	1

NOT

~A = 1 cuando A = 0

~	
0	1
1	0

EXclusive-OR (XOR)

A ^ B = 1 cuando A = 1 o B = 1, pero no ambos

^	0	1
0	0	1
1	1	0

Álgebras de Boole en general

Operan con vectores de bits

```
  01101001       01101001       01101001
& 01010101     | 01010101     ^ 01010101     ~ 01010101
----------     ----------     ----------     ----------
  01000001       01111101       00111100       10101010
```

Aplican todas las propiedades del Álgebra Booleana

- $\langle \{0, 1\}, |, \&, \ , 0, 1 \rangle$ forman un álgebra
- OR es la operación “suma”
- AND es la operación “producto”
- NOT es la operación “complemento”
- 0 es la identidad para la “suma”
- 1 es la identidad para el “producto”

Tabla de contenidos

Operaciones con bits en C

Operaciones & (AND), | (OR), ~ (NOT), ^ (XOR)

- Aplican a cualquier dato “entero”
  - `long`, `int`, `short`, `char`, `unsigned`
- Vectores de bits

Ejemplos

- $\sim 01000001_2 \Rightarrow 10111110_2$
- $\sim 00000000_2 \Rightarrow 11111111_2$
- $01101001_2 \& 01010101_2 \Rightarrow 01000001_2$
- $01101001_2 | 01010101_2 \Rightarrow 01111101_2$

Contraste: operaciones lógicas en C

Operaciones && (AND), || (OR), ! (NOT)

- 0 es “Falso”
- Cualquier cosa distintas de cero es “Verdadera”
- Retornan 0 ó 1
- Evaluación mínima: **¡short-circuits!**

Ejemplos:

- `!41` ⇒ `00`
- `!00` ⇒ `01`
- `!!41` ⇒ `01`

- `69 && 55` ⇒ `1`
- `69 && 0` ⇒ `0`
- `69 || 55` ⇒ `1`
- `p && *p` (evita el acceso a punteros nulos)

Desplazamientos: `shift`

Left Shift: u << k

- Desplaza los bits de u a la izquierda k posiciones
  - Descarta los k bits de la izquierda
- Completa con ceros a la derecha

Argumento u	01100010
<< 3	00010000
Log. >> 2	00011000
Arit. >> 2	00011000

Right Shift: u >> k

- Desplaza los bits de u a la derecha
  - Descarta los k bits de la derecha
- Desplazamiento lógico
  - Completa con ceros a izquierda
- Desplazamiento aritmético
  - Replica el msb k veces a izquierda

Argumento u	10100010
<< 3	00010000
Log. >> 2	00101000
Arit. >> 2	11101000

Si $k < 0$ ó $k \geq$ *word-size* $\Rightarrow$ **Undefined Behaviour**

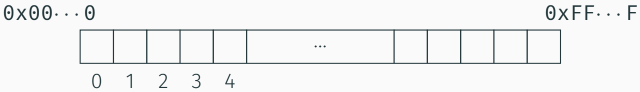
Tabla de contenidos

Organización de la memoria: de a bytes

- Los programas acceden a los *datos* usando *direcciones de memoria*
  - conceptualmente, la ven como un arreglo de **bytes**
    - no lo es, pero sirve pensarlo así

Organización de la memoria: de a bytes

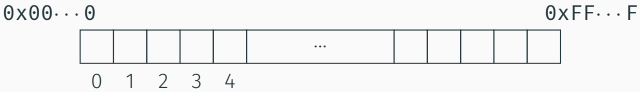
- Los programas acceden a los *datos* usando *direcciones de memoria*
  - conceptualmente, la ven como un arreglo de **bytes**
    - no lo es, pero sirve pensarlo así



  - una *dirección de memoria* es como un índice en ese arreglo

Organización de la memoria: de a bytes

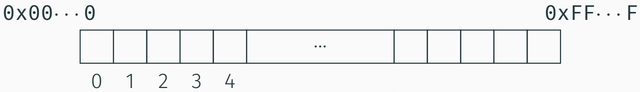
- Los programas acceden a los *datos* usando *direcciones de memoria*
  - conceptualmente, la ven como un arreglo de **bytes**
    - no lo es, pero sirve pensarlo así



  - una *dirección de memoria* es como un índice en ese arreglo
    - En C, un **puntero** es una variable que guarda *direcciones de memoria*

Organización de la memoria: de a bytes

- Los programas acceden a los *datos* usando *direcciones de memoria*
    - conceptualmente, la ven como un arreglo de **bytes**
        - no lo es, pero sirve pensarlo así



- una *dirección de memoria* es como un índice en ese arreglo
    - En C, un **puntero** es una variable que guarda *direcciones de memoria*

```
p = 0xFF...D;
```

Organización de la memoria: de a bytes

- Los programas acceden a los *datos* usando *direcciones de memoria*
  - conceptualmente, la ven como un arreglo de **bytes**
    - no lo es, pero sirve pensarlo así

  - una *dirección de memoria* es como un índice en ese arreglo
    - En C, un **puntero** es una variable que guarda *direcciones de memoria*

```
p = 0xFF...D;
```

Organización de la memoria: de a bytes

- Los programas acceden a los *datos* usando *direcciones de memoria*
  - conceptualmente, la ven como un arreglo de **bytes**
    - no lo es, pero sirve pensarlo así

  - una *dirección de memoria* es como un índice en ese arreglo
    - En C, un **puntero** es una variable que guarda *direcciones de memoria*

```
p = 0xFF...D;
```

- Cada *proceso* tiene su espacio de direcciones privado
  - piensen un *proceso* como un programa en ejecución
  - un programa puede modificar sus datos, pero no los de otro
    - no siempre fue así :-S

Tamaño de palabras

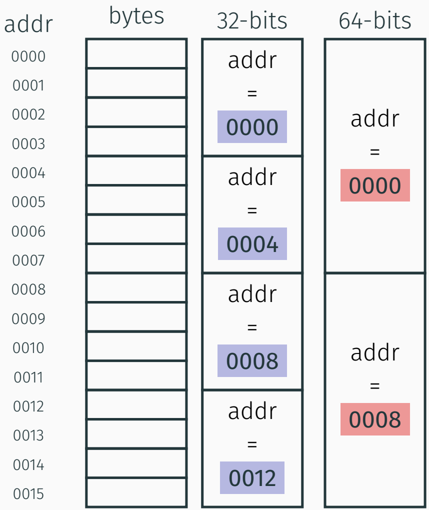
Todo sistema tiene un tamaño de palabra (*word size*)

- Tamaño nominal de los datos enteros
  - incluyendo las direcciones de memoria
- Máquinas de 32 bits:
  - *word size*: 4 bytes
  - *address space*: limitado a 4 GiB (2^{32} *bytes*)
- Máquinas de 64 bits:
  - *word size*: 8 bytes
  - *address space*: “limitado” a 16 EiB (exbibyte) (2^{64} *bytes*)
  - ```
              4294967296
    18446744073709551616
    ```
- ¿Y los datos de otros tamaños?
    ```

Organización de la memoria: de a palabras

Las direcciones

- indican las posiciones de **bytes**
- en particular, del primer byte de una palabra
- palabras sucesivas difieren en 4 u 8



Tamaños de datos

Tipo (C)	32-bit	64-bit	x86-64
`char`	1	1	1
`short`	2	2	2
`int`	4	4	4
`long`	4	8	8
`float`	4	4	4
`double`	8	8	8
`long double`	–	–	10/16
`int *`	4	8	8
`char *`	4	8	8
`cualquier puntero`	4	8	8

Tabla de contenidos

Byte ordering

¿Cómo se pueden ordenar los bytes de datos multi-byte en memoria?

Supongamos

- La variable `kiwi` almacena el valor de 4-bytes: `0x01234567`
- La posición de memoria dada por `&kiwi` es: `0x100`
- El byte más significativo es `0x01`, el menos significativo es `0x67`

Big Endian:

			0x100	0x101	0x102	0x103			
...			01	23	45	67			...

Little Endian:

			0x100	0x101	0x102	0x103			
...			67	45	23	01			...

Byte ordering

¿Cómo se ordenan los bytes de datos multi-byte en memoria?

¡Por convención!

Byte ordering

¿Cómo se ordenan los bytes de datos multi-byte en memoria?

¡Por convención!

Convenciones

- *Big Endian*: las viejas Sun SPARC y iMac PowerPC (hoy son bi-).
  - El byte **menos** significativo tiene la dirección más alta.

Byte ordering

¿Cómo se ordenan los bytes de datos multi-byte en memoria?

¡Por convención!

Convenciones

- *Big Endian*: las viejas Sun SPARC y iMac PowerPC (hoy son bi-).
    - El byte **menos** significativo tiene la dirección más **alta**.
- *Little Endian*: x86, ADM64/x86-64, los ARM que corren Android, iOS y Windows.
    - El byte **menos** significativo tiene la dirección más **baja**.

Tabla de contenidos

Tabla de contenidos

Representaciones numéricas

Sistemas de numeración

- Posicionales
  - Decimal: 2953253037_{10} = 2953253037_{10}

$$2953253037_{10} = 2 \cdot 10^9 + 9 \cdot 10^8 + 5 \cdot 10^7 + 3 \cdot 10^6 + 2 \cdot 10^5 + 5 \cdot 10^4 + 3 \cdot 10^3 + 3 \cdot 10^1 + 7 \cdot 10^0$$

Representaciones numéricas

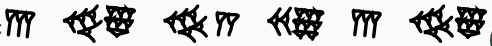
Sistemas de numeración

- Posicionales
  - Decimal: $2953253037_{10} = 2953253037_{10}$
  - Hexadecimal alias *hexa*: $2953253037_{10} = \texttt{b00710ad}_{16}$

$$\texttt{b00710ad}_{16} = b \cdot 16^7 + 7 \cdot 16^4 + 1 \cdot 16^3 + a \cdot 16^1 + d \cdot 16^0$$

Representaciones numéricas

Sistemas de numeración

- Posicionales
  - Decimal: 2953253037_{10} = 2953253037_{10}
  - Hexadecimal alias *hexa*: 2953253037_{10} = `b00710ad`$_{16}$
  - Sexagesimal: 2953253037_{10} = $_{60}$

Representaciones numéricas

Sistemas de numeración

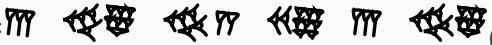
- Posicionales
  - Decimal: 2953253037_{10} = 2953253037_{10}
  - Hexadecimal alias *hexa*: 2953253037_{10} = $\texttt{b00710ad}_{16}$
  - Sexagesimal: 2953253037_{10} =  $_{60}$
  - Binario: 2953253037_{10} = $10110000000001110001000010101101_2$

$$10110000000001110001000010101101_2 = 1{\cdot}2^{31}+0{\cdot}2^{30}+1{\cdot}2^{29}+1{\cdot}2^{28}+0{\cdot}2^{27}+0{\cdot}2^{26}+0{\cdot}2^{25}+0{\cdot}2^{24}+0{\cdot}2^{23}+0{\cdot}2^{22}+$$

sigue hasta $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$ un tanto dificil de leer para más de 4 bits

Representaciones numéricas

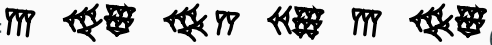
Sistemas de numeración

- Posicionales
  - Decimal: $2953253037_{10} = 2953253037_{10}$
  - Hexadecimal alias *hexa*: $2953253037_{10} =$ `b00710ad`$_{16}$
  - Sexagesimal: $2953253037_{10} =$  $_{60}$
  - Binario: $2953253037_{10} = 10110000000001110001000010101101_2$

$$N_b = \sum_{i=-\infty}^{\infty} c_i \cdot b^i$$

Representaciones numéricas

Sistemas de numeración

- Posicionales
  - Decimal: 2953253037_{10} = 2953253037_{10}
  - Hexadecimal alias *hexa*: 2953253037_{10} = $\texttt{b00710ad}_{16}$
  - Sexagesimal: 2953253037_{10} =  $_{60}$
  - Binario: 2953253037_{10} = $10110000000001110001000010101101_2$

$$N_b = \sum_{i=-\infty}^{\infty} c_i \cdot b^i$$

- No posicionales
  - Romano: $\texttt{MMXIX}$  = 2019_{10}

Ejemplos: representaciones en binario

Representación de números en Base 2

- El número 13548250_{10} se representa como $110011101011101011011010_2$

$$N_2 = \sum_{i=0}^{m-1} c_i \cdot 2^i$$

Ejemplos: representaciones en binario

Representación de números en Base 2

$$N_2 = \sum_{i=0}^{m-1} c_i \cdot 2^i$$

- El número 13548250_{10} se representa como $1100111010111010110110 10_2$
- El número $1{,}20_{10}$ se representa como $1.001100110011[0011]..._2$

$$1{,}20_{10} = 1\cdot 2^0 + 0\cdot 2^{-1} + 0\cdot 2^{-2} + 1\cdot 2^{-3} + 1\cdot 2^{-4} + 0\cdot 2^{-5} + 0\cdot 2^{-6} + 1\cdot 2^{-7} + 1\cdot 2^{-8} + 0\cdot 2^{-9} +$$

$$+ 0 \cdot 2^{-10} + 1 \cdot 2^{-11} + 1 \cdot 2^{-12} + 0 \cdot 2^{-13} + 0 \cdot 2^{-14} + 1 \cdot 2^{-15} + 1 \cdot 2^{-16} + [0011]\ldots_2$$

Ejemplos: representaciones en binario

Representación de números en Base 2

- El número 13548250_{10} se representa como $1100111010111010110111010_2$
- El número $1{,}20_{10}$ se representa como $1.001100110011[0011]..._2$

$$N_2 = \sum_{i=0}^{m-1} c_i \cdot 2^i$$

$$N_2 = \sum_{i=-n}^{m-n-1} c_i \cdot 2^i$$

$$1{,}20_{10} = 1 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} + 0 \cdot 2^{-5} + 0 \cdot 2^{-6} + 1 \cdot 2^{-7} + 1 \cdot 2^{-8} + 0 \cdot 2^{-9} +$$
$$+ 0 \cdot 2^{-10} + 1 \cdot 2^{-11} + 1 \cdot 2^{-12} + 0 \cdot 2^{-13} + 0 \cdot 2^{-14} + 1 \cdot 2^{-15} + 1 \cdot 2^{-16} + [0011] \ldots_2$$

Ejemplos: representaciones en binario

Representación de números en Base 2

$$N_2 = \sum_{i=0}^{m-1} c_i \cdot 2^i$$

$$N_2 = \sum_{i=-n}^{m-n-1} c_i \cdot 2^i$$

- El número 13548250_{10} se representa como $110011101011101011011010_2$
- El número $1{,}20_{10}$ se representa como $1.001100110011[0011]..._2$

$$1{,}20_{10} = 1 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} + 0 \cdot 2^{-5} + 0 \cdot 2^{-6} + 1 \cdot 2^{-7} + 1 \cdot 2^{-8} + 0 \cdot 2^{-9} + \\ + 0 \cdot 2^{-10} + 1 \cdot 2^{-11} + 1 \cdot 2^{-12} + 0 \cdot 2^{-13} + 0 \cdot 2^{-14} + 1 \cdot 2^{-15} + 1 \cdot 2^{-16} + [0011] \ldots_2$$

- El número $1{,}3548250 \times 10^7$ se representa como $1.10011101011101011011010_2 \times 2^{23}$

Encoding Bytes

1 Byte = 8 bits

- Binario: $\texttt{00000000}_2$ a $\texttt{11111111}_2$
- Decimal: 0_{10} a 255_{10}
- Hexa: 00_{16} a FF_{16}
    - del 0 al 9 ¿y después? ¿113 es 1-13 u 11-3?

Encoding Bytes

1 Byte = 8 bits

- Binario: `00000000`$_2$ a `11111111`$_2$
- Decimal: 0_{10} a 255_{10}
- Hexa: 00_{16} a FF_{16}
  - del 0 al 9 ¿y después? ¿113 es 1-13 u 11-3?

hexa	decimal	binario
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Encoding Bytes

1 Byte = 8 bits

- Binario: 00000000_2 a 11111111_2
- Decimal: 0_{10} a 255_{10}
- Hexa: 00_{16} a FF_{16}
  - del 0 al 9 ¿y después? ¿113 es 1-13 u 11-3?
  - Se utilizan los caracteres `0` a `9` y `A` a `F`

hexa	decimal	binario
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Encoding Bytes

1 Byte = 8 bits

- Binario: `00000000`$_2$ a `11111111`$_2$
- Decimal: 0_{10} a 255_{10}
- Hexa: 00_{16} a FF_{16}
    - del 0 al 9 ¿y después? ¿113 es 1-13 u 11-3?
    - Se utilizan los caracteres `0` a `9` y `A` a `F`
- En C, un entero en hexa se escribe de la siguiente manera:
    - $\text{b00710ad}_{16} \rightarrow$ `0xb00710ad`
    - o bien: `0xB00710AD`

hexa	decimal	binario
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Encoding Bytes

1 Byte = 8 bits

- Binario: `00000000`$_2$ a `11111111`$_2$
- Decimal: 0_{10} a 255_{10}
- Hexa: 00_{16} a FF_{16}
  - del 0 al 9 ¿y después? ¿113 es 1-13 u 11-3?
  - Se utilizan los caracteres `0` a `9` y `A` a `F`
- En C, un entero en hexa se escribe de la siguiente manera:
  - $\text{b00710ad}_{16} \rightarrow$ `0xb00710ad`
  - o bien: `0xB00710AD`

¿para qué me sirve?

hexa	decimal	binario
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Tabla de contenidos

Codificación de enteros

No signado

$$B2U_w(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

`short`: 2 bytes

```c
short int x =  16162;
short int y = -16162;
```

Complemento a 2

$$B2T_w(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

	Decimal	Hexa	Binario
x	16162	3F 22	00111111 00100010
y	-16162	C0 DE	11000000 11011110

Bit de signo
En complemento a dos, el bit más significativo (MSB) indica el signo

- 0 para positivo
- 1 para negativo

Codificación de enteros

No signado

$$B2U_w(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

`short`: 2 bytes

```c
short int x =  16162;
short int y = -16162;
```

Complemento a 2

$$B2T_w(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

	Decimal	Hexa	Binario
x	16162	3F 22	00111111 00100010
y	-16162	C0 DE	11000000 11011110

Bit de signo
En complemento a dos, el bit más significativo (MSB) indica el signo

- 0 para positivo
- 1 para negativo

Codificación de enteros

No signado

$$B2U_w(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

short: 2 bytes

```
short int x =  16162;
short int y = -16162;
```

Complemento a 2

$$B2T_w(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

	Decimal	Hexa	Binario
x	16162	3F 22	$\boxed{0}$0111111 00100010
y	-16162	C0 DE	11000000 11011110

Bit de signo
En complemento a dos, el bit más significativo (MSB) indica el signo

- 0 para positivo
- 1 para negativo

Codificación de enteros

No signado

$$B2U_w(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

`short`: 2 bytes

```c
short int x =  16162;
short int y = -16162;
```

Complemento a 2

$$B2T_w(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

	Decimal	Hexa	Binario
x	16162	3F 22	$\boxed{0}$0111111 00100010
y	-16162	C0 DE	$\boxed{1}$1000000 11011110

Bit de signo

En complemento a dos, el bit más significativo (MSB) indica el signo

- 0 para positivo
- 1 para negativo

Ejemplos $B2U_w(X)$ y $B2T_w(X)$

```
x =  16162  : 00111111 00100010
y = -16162  : 11000000 11011110
```

Peso (2^i)	16162		-16162	
1	0	0	0	0
2	1	2	1	2
4	0	0	1	4
8	0	0	1	8
16	0	0	1	16
32	1	32	0	0
64	0	0	1	64
128	0	0	1	128
256	1	256	0	0
512	1	512	0	0
1024	1	1024	0	0
2048	1	2048	0	0
4096	1	4096	0	0
8192	1	8192	0	0
16384	0	0	1	16384
-32768	0	0	1	-32768
Suma:		16162		-16162

Rangos numéricos

Límites: no signados

$$\text{UMin} = 0 \qquad (000\ldots0)$$

$$\text{UMax} = 2^w - 1 \qquad (111\ldots1)$$

Límites: complemento a dos

$$\text{TMin} = -2^{w-1} \qquad (100\ldots0)$$

$$\text{TMax} = 2^{w-1} - 1 \qquad (011\ldots1)$$

Valores para $w = 16$

	Decimal	Hexa	Binario
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

C Puzzles

- 32-bit *word-size*, complemento a dos
- Argumentar si es cierto, o dar un contraejemplo en las siguientes expresiones

```
foo();
bar();
 ux = x;
 uy = y;
```

- `x < 0` ⇒ `((x*2) < 0)`
- `ux >= 0`
- `x & 7 == 7` ⇒ `(x<<30) < 0`
- `ux > -1`
- `x > y` ⇒ `-x < -y`
- `x * x >= 0`
- `x > 0 && y > 0` ⇒ `x + y > 0`
- `x >= 0` ⇒ `-x <= 0`
- `x <= 0` ⇒ `-x >= 0`

Valores para distintos tamaño de palabra

	tamaño de palabra (w)			
	8	16	32	64
UMax	255	65535	4294967296	18 446 744 073 709 551 615
TMax	127	32767	2147483647	9 223 372 036 854 775 807
TMin	-128	-32768	-2147483648	-9 223 372 036 854 775 808

Observaciones

- $|\text{TMin}| = \text{TMax} + 1$
  - Rango asimétrico
- $\text{UMax} = 2 \cdot \text{TMax} + 1$

Programación en C

- `#include <limits.h>`
- Declara constantes
  - `ULONG_MAX`,
  - `LONG_MAX`,
  - `LONG_MIN`, etc.
- Los valores son dependientes de la plataforma

Relación entre signado y no signado

X	$B2U(X)$	$B2T(X)$
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

- **Equivalencia**
  - Mismo encoding para valores no negativos
- **Unicidad**
  - Cada patrón representa un único entero
  - A cada entero representable le corresponde un único patrón
- $\Rightarrow$ **se pueden invertir los mapeos**
  - $U2B(X) = B2U^{-1}(X)$
    - Patrón de bits para un entero sin signo
  - $T2B(X) = B2T^{-1}(X)$
    - Patrón de bits para un entero en complemento a dos

Conversión de signado a no signado

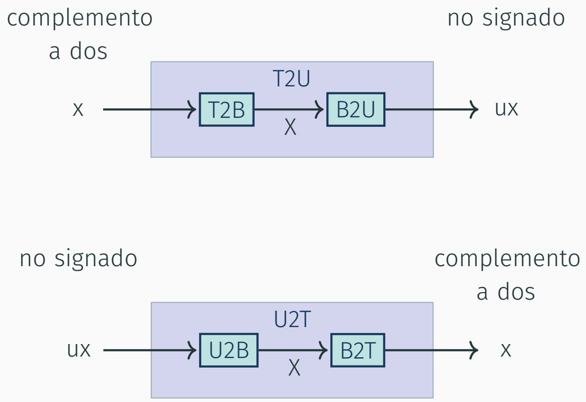
C permite convertir enteros signados a no signados

```
short int x = 16162;
unsigned short int ux = (unsigned short) x;
short int y = -16162;
unsigned short int uy = (unsigned short) y;
```

Resultado

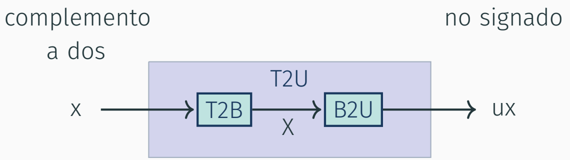
- **NO** cambia la representación *binaria*
- Los valores no negativos se mantienen igual
    - `ux = 16162`
- Los valores negativos cambian a valores (muy) grandes
    - `uy = 49374`

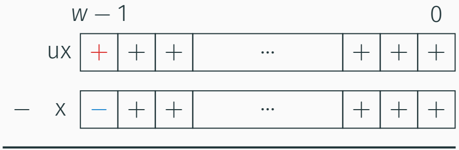
Conversión de signado a no signado



No cambia el patrón de bits: mantener los bits y reinterpretar

Relación entre signado y no signado





$$+2^{w-1} - -2^{w-1} = 2 \cdot 2^{w-1} = 2^w$$

$$\boxed{ux = \begin{cases} x & x \geq 0 \\ x + 2^w & x < 0 \end{cases}}$$

Un peso negativo grande se convierte en un peso positivo grande

Relación entre signado y no signado

Peso (2^i)	-16162		49374	
1	0	0	0	0
2	1	2	1	2
4	1	4	1	4
8	1	8	1	8
16	1	16	1	16
32	0	0	0	0
64	1	64	1	64
128	1	128	1	128
256	0	0	0	0
512	0	0	0	0
1024	0	0	0	0
2048	0	0	0	0
4096	0	0	0	0
8192	0	0	0	0
16384	1	16384	1	16384
±32768	1	-32768	1	32768
Suma:		-16162		49374

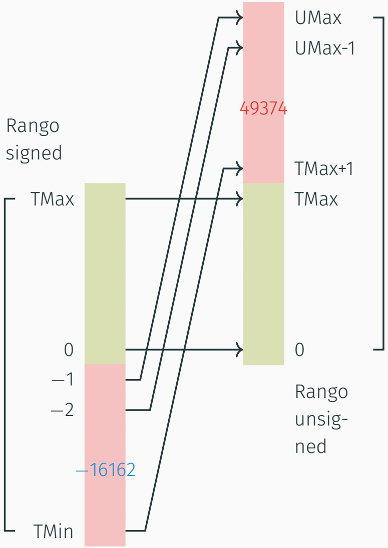


Tabla de contenidos

En C, ¿cuándo es cuál?

- Constantes
  - por omisión, son signadas
  - `U` es el sufijo `unsigned`
    `0U, 4294967259U`
- Casting
  - Explícito

```c
int tx, ty;
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```

  - Implícito: asignaciones, evaluaciones, llamadas

```c
tx = ux;
uy = ty;
```

En C, ¿cuándo es cuál?

- Evaluación de expresiones
  - si hay mezcla de signados y no signados en una expresión los valores con signo se *castean implícitamente* a `unsigned`
  - Ejemplos para $W = 32$: $TMIN = -2147483648$, $TMAX = 2147483647$

Constante$_1$	Constante$_2$	Relación	Evaluación
`0`	`0U`	`==`	
`-1`	`0`	`<`	
`-1`	`0U`	`>`	
`2147483647`	`-2147483647-1`	`>`	
`2147483647U`	`-2147483647-1`	`<`	
`-1`	`-2`	`>`	
`(unsigned)-1`	`-2`	`>`	
`2147483647`	`2147483648U`	`<`	
`2147483647`	`(int)2147483648U`	`>`	

En C, ¿cuándo es cuál?

- Evaluación de expresiones
  - si hay mezcla de signados y no signados en una expresión los valores con signo se *castean implícitamente* a `unsigned`
  - Ejemplos para $W = 32$: $TMIN = -2147483648$, $TMAX = 2147483647$

Constante$_1$	Constante$_2$	Relación	Evaluación
`0`	`0U`	`==`	`unsigned`
`-1`	`0`	`<`	
`-1`	`0U`	`>`	
`2147483647`	`-2147483647-1`	`>`	
`2147483647U`	`-2147483647-1`	`<`	
`-1`	`-2`	`>`	
`(unsigned)-1`	`-2`	`>`	
`2147483647`	`2147483648U`	`<`	
`2147483647`	`(int)2147483648U`	`>`	

En C, ¿cuándo es cuál?

- Evaluación de expresiones
  - si hay mezcla de signados y no signados en una expresión los valores con signo se *castean implícitamente* a `unsigned`
  - Ejemplos para $W = 32$: $TMIN = -2147483648$, $TMAX = 2147483647$

Constante_1	Constante_2	Relación	Evaluación
`0`	`0U`	`==`	`unsigned`
`-1`	`0`	`<`	`signed`
`-1`	`0U`	`>`	
`2147483647`	`-2147483647-1`	`>`	
`2147483647U`	`-2147483647-1`	`<`	
`-1`	`-2`	`>`	
`(unsigned)-1`	`-2`	`>`	
`2147483647`	`2147483648U`	`<`	
`2147483647`	`(int)2147483648U`	`>`	

En C, ¿cuándo es cuál?

- Evaluación de expresiones
  - si hay mezcla de signados y no signados en una expresión los valores con signo se *castean implícitamente* a `unsigned`
  - Ejemplos para $W = 32$: $TMIN = -2147483648$, $TMAX = 2147483647$

Constante$_1$	Constante$_2$	Relación	Evaluación
`0`	`0U`	`==`	`unsigned`
`-1`	`0`	`<`	`signed`
`-1`	`0U`	`>`	`unsigned`
`2147483647`	`-2147483647-1`	`>`	
`2147483647U`	`-2147483647-1`	`<`	
`-1`	`-2`	`>`	
`(unsigned)-1`	`-2`	`>`	
`2147483647`	`2147483648U`	`<`	
`2147483647`	`(int)2147483648U`	`>`	

En C, ¿cuándo es cuál?

- Evaluación de expresiones
  - si hay mezcla de signados y no signados en una expresión los valores con signo se *castean implícitamente* a `unsigned`
  - Ejemplos para $W = 32$: $TMIN = -2147483648$, $TMAX = 2147483647$

Constante_1	Constante_2	Relación	Evaluación
`0`	`0U`	`==`	`unsigned`
`-1`	`0`	`<`	`signed`
`-1`	`0U`	`>`	`unsigned`
`2147483647`	`-2147483647-1`	`>`	`signed`
`2147483647U`	`-2147483647-1`	`<`	
`-1`	`-2`	`>`	
`(unsigned)-1`	`-2`	`>`	
`2147483647`	`2147483648U`	`<`	
`2147483647`	`(int)2147483648U`	`>`	

En C, ¿cuándo es cuál?

- Evaluación de expresiones
  - si hay mezcla de signados y no signados en una expresión los valores con signo se *castean implícitamente* a `unsigned`
  - Ejemplos para $W = 32$: $TMIN = -2147483648$, $TMAX = 2147483647$

Constante$_1$	Constante$_2$	Relación	Evaluación
`0`	`0U`	`==`	`unsigned`
`-1`	`0`	`<`	`signed`
`-1`	`0U`	`>`	`unsigned`
`2147483647`	`-2147483647-1`	`>`	`signed`
`2147483647U`	`-2147483647-1`	`<`	`unsigned`
`-1`	`-2`	`>`	
`(unsigned)-1`	`-2`	`>`	
`2147483647`	`2147483648U`	`<`	
`2147483647`	`(int)2147483648U`	`>`	

En C, ¿cuándo es cuál?

- Evaluación de expresiones
  - si hay mezcla de signados y no signados en una expresión los valores con signo se *castean implícitamente* a `unsigned`
  - Ejemplos para $W = 32$: $TMIN = -2147483648$, $TMAX = 2147483647$

Constante_1	Constante_2	Relación	Evaluación
`0`	`0U`	`==`	`unsigned`
`-1`	`0`	`<`	`signed`
`-1`	`0U`	`>`	`unsigned`
`2147483647`	`-2147483647-1`	`>`	`signed`
`2147483647U`	`-2147483647-1`	`<`	`unsigned`
`-1`	`-2`	`>`	`signed`
`(unsigned)-1`	`-2`	`>`	
`2147483647`	`2147483648U`	`<`	
`2147483647`	`(int)2147483648U`	`>`	

En C, ¿cuándo es cuál?

- Evaluación de expresiones
  - si hay mezcla de signados y no signados en una expresión los valores con signo se *castean implícitamente* a `unsigned`
  - Ejemplos para $W = 32$: $TMIN = -2147483648$, $TMAX = 2147483647$

Constante$_1$	Constante$_2$	Relación	Evaluación
`0`	`0U`	`==`	`unsigned`
`-1`	`0`	`<`	`signed`
`-1`	`0U`	`>`	`unsigned`
`2147483647`	`-2147483647-1`	`>`	`signed`
`2147483647U`	`-2147483647-1`	`<`	`unsigned`
`-1`	`-2`	`>`	`signed`
`(unsigned)-1`	`-2`	`>`	`unsigned`
`2147483647`	`2147483648U`	`<`	
`2147483647`	`(int)2147483648U`	`>`	

En C, ¿cuándo es cuál?

- Evaluación de expresiones
  - si hay mezcla de signados y no signados en una expresión los valores con signo se *castean implícitamente* a `unsigned`
  - Ejemplos para $W = 32$: $TMIN = -2147483648$, $TMAX = 2147483647$

Constante_1	Constante_2	Relación	Evaluación
`0`	`0U`	`==`	`unsigned`
`-1`	`0`	`<`	`signed`
`-1`	`0U`	`>`	`unsigned`
`2147483647`	`-2147483647-1`	`>`	`signed`
`2147483647U`	`-2147483647-1`	`<`	`unsigned`
`-1`	`-2`	`>`	`signed`
`(unsigned)-1`	`-2`	`>`	`unsigned`
`2147483647`	`2147483648U`	`<`	`unsigned`
`2147483647`	`(int)2147483648U`	`>`	

En C, ¿cuándo es cuál?

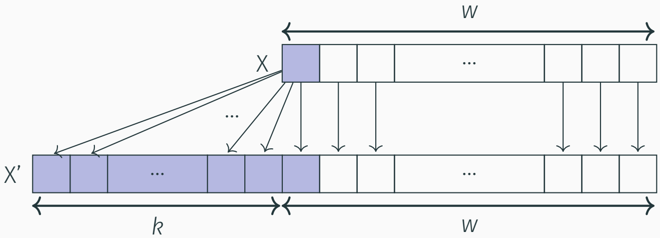
- Evaluación de expresiones
  - si hay mezcla de signados y no signados en una expresión los valores con signo se *castean implícitamente* a `unsigned`
  - Ejemplos para $W = 32$: $TMIN = -2147483648$, $TMAX = 2147483647$

Constante$_1$	Constante$_2$	Relación	Evaluación
`0`	`0U`	`==`	`unsigned`
`-1`	`0`	`<`	`signed`
`-1`	`0U`	`>`	`unsigned`
`2147483647`	`-2147483647-1`	`>`	`signed`
`2147483647U`	`-2147483647-1`	`<`	`unsigned`
`-1`	`-2`	`>`	`signed`
`(unsigned)-1`	`-2`	`>`	`unsigned`
`2147483647`	`2147483648U`	`<`	`unsigned`
`2147483647`	`(int)2147483648U`	`>`	`signed`

Extensión de signo

- **Queremos:**
  - Dado un entero *signado* de w-bits
  - Convertirlo a un entero de $w + k$-bits con el mismo valor
- **Regla:**
  - Propagar el bit de signo a los nuevos k bits (MSB)

$$X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copias del MSB}}, \underbrace{x_{w-1}, x_{w-2}, \ldots, x_1, x_0}_{\text{entero original}}$$



Extensión de signo: ejemplo

```c
short int x =  16162;
int      ix = (int) x;
short int y = -16162;
int      iy = (int) y;
```

	Decimal	Hex	Binario
x	16162	3F 22	00111111 00100010
ix	16162	00 00 3F 22	00000000 00000000 00111111 00100010
y	-16162	C0 DE	11000000 11011110
iy	-16162	FF FF C0 DE	11111111 11111111 11000000 11011110

Se aumenta la cantidad de bits en la representación: expansión ó *up casting*.

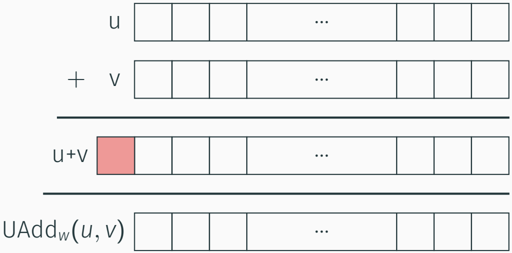
Tabla de contenidos

Suma de enteros no signados

Operandos de w bits

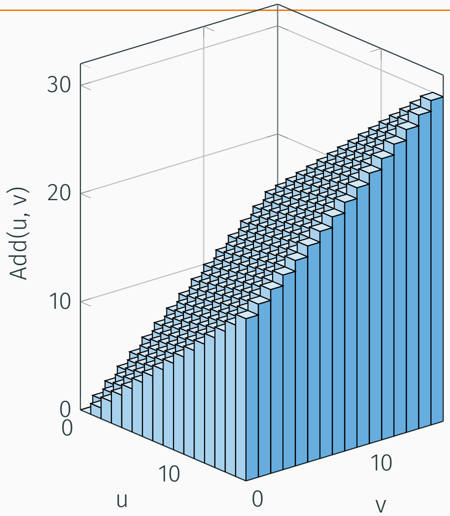
Suma verdadera de $w + 1$ bits

Descarta *carry*: w bits

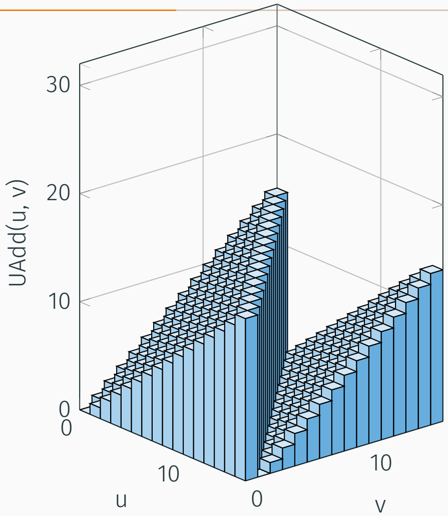


$$\mathrm{UAdd}_w(u, v) = u + v \mod 2^w$$

Suma de enteros no signados

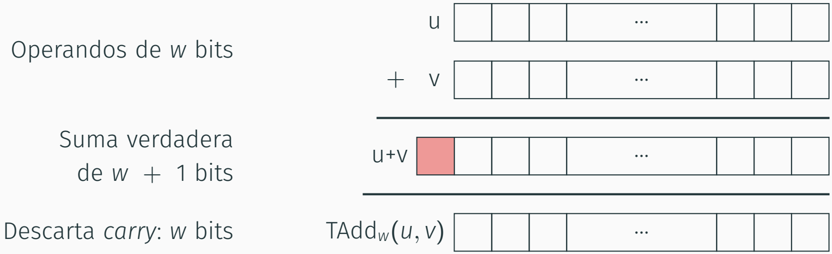


Suma teórica de enteros



Suma real de enteros

Suma de enteros en complemento a dos



- A nivel binario: comportamiento idéntico a la suma unsigned
  - ```c
    int s, t, u, v;
    s = (int) ((unsigned) u + (unsigned) v);
    t = u + v;
    ```
  - da `t == s`
    ```

Suma de enteros signados

Ejemplo con $w = 4 \Longrightarrow$

- Requiere $w + 1$ bits
- Descarta el MSB
- El resultado es en complemento a dos
- *wraps around*
  - Si la suma $\geq 2^{w-1}$
    - Se vuelve negativa
    - No más de una vez
  - Si la suma $< -2^{w-1}$
    - Se vuelve positiva
    - No más de una vez

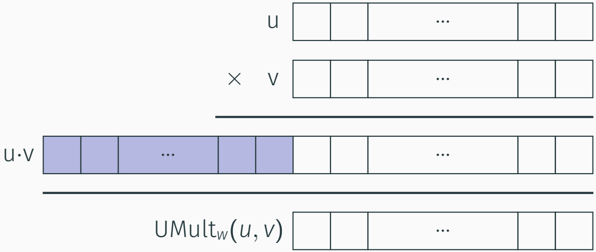
u		v		suma		TAdd$_4$(u,v)		
bin	dec	bin	dec	bin	dec	msb	4-bit	dec
0111	7	0111	7	01110	14	0	1110	-2
0111	7	0110	6	01101	13	0	1101	-3
0111	7	0101	5	01100	12	0	1100	-4
0110	6	0100	4	01010	10	0	1010	-6
0011	3	0110	6	01001	9	0	1001	-7
0110	6	0010	2	01000	8	0	1000	-8
0100	4	0011	3	00111	7	0	0111	7
0000	0	0110	6	00110	6	0	0110	6
0101	5	0000	0	00101	5	0	0101	5
1101	-3	0110	6	00011	3	0	0011	3
1111	-1	0010	2	00001	1	0	0001	1
1001	-7	0111	7	00000	0	0	0000	0
0011	3	1100	-4	11111	-1	1	1111	-1
1000	-8	0101	5	11101	-3	1	1101	-3
0100	4	1000	-8	11100	-4	1	1100	-4
1001	-7	0010	2	11011	-5	1	1011	-5
1000	-8	0001	1	11001	-7	1	1001	-7
1010	-6	1110	-2	11000	-8	1	1000	-8
1000	-8	1111	-1	10111	-9	1	0111	7
1010	-6	1100	-4	10110	-10	1	0110	6
1001	-7	1100	-4	10101	-11	1	0101	5
1000	-8	1100	-4	10100	-12	1	0100	4
1000	-8	1001	-7	10001	-15	1	0001	1
1000	-8	1000	-8	10000	-16	1	0000	0

Multiplicación de enteros

Operandos de w bits

Resultado de $2w$ bits

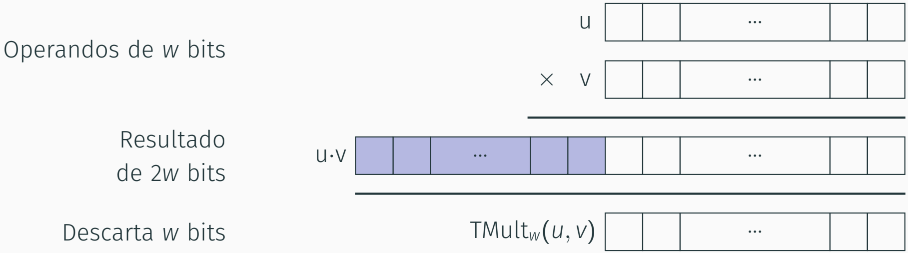
Descarta w bits



- **Multiplicación estándar**
  - ignora los w bits superiores
- **Implementa aritmética modular**
  - como ocurre con la suma:

$$\text{UMult}_w(u, v) = u \cdot v \mod 2^w$$

Multiplicación de enteros



- **Multiplicación estándar**
  - ignora los w bits superiores
  - los bits dentro de los w superiores pueden diferir
  - los bits inferiores son iguales a $\mathrm{UMult}_w(u, v)$

Multiplicar por potencias de 2

- Representación numérica

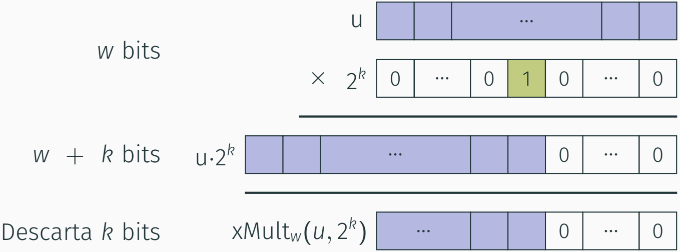
$$N_2 = \sum_{i=0}^{w-1} b_i \cdot 2^i$$

- Multiplicar por 2^k
  - requiere k bits más

$$\begin{aligned} N_2 \cdot 2^k &= \left(\sum_{i=0}^{w-1} b_i \cdot 2^i \right) \cdot 2^k \\ &= \sum_{i=0}^{w-1} b_i \cdot 2^{i+k} \\ &= \sum_{i=k}^{w+k-1} b_i \cdot 2^i + \sum_{j=0}^{k-1} 0 \cdot 2^j \end{aligned}$$

  - ignora los k bits superiores

Multiplicar por potencias de 2: shift



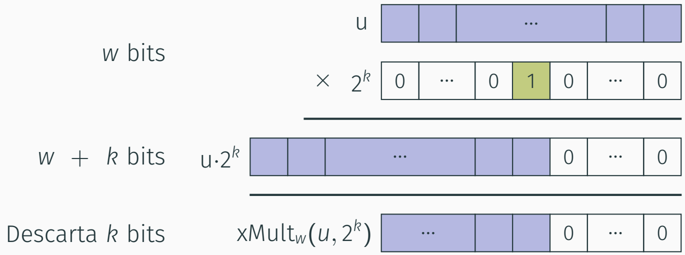
Operación

- $u \cdot 2^k$ se puede hacer con desplazamientos: $u << k$

Ejemplos

- `u << 3 == u * 8`
- `(u << 5) - (u << 3) == u * 24`
- `64 - 1 << 3 * 2 - 1 ==`
- `INT_MAX << 1 == ??`

Multiplicar por potencias de 2: shift



Operación

- $u \cdot 2^k$ se puede hacer con desplazamientos: $u << k$

Ejemplos

- `u << 3 == u * 8`
- `(u << 5) - (u << 3) == u * 24`
- `64 - 1 << 3 * 2 - 1 == 2016`
- `INT_MAX << 1 == ??`

Dividir por potencias de 2: shift

Operación

- $\lfloor u/2^k \rfloor$ se puede hacer con desplazamientos: $u >> k$

Ejemplos

	Teórica	Real	Hex	Binario
x	16162	16162	3F 22	00111111 00100010
x >> 1	8081.0	8081	1F 91	00011111 10010001
x >> 2	4040.50	4040	0F C8	00001111 11001000
x >> 8	63.133	63	00 3F	00000000 00111111
y	-16162	-16162	C0 DE	11000000 11011110
y >> 1	-8081.0	-8081	E0 6F	11100000 01101111
y >> 2	-4040.50	-4041	F0 37	11110000 00110111
y >> 8	-63.133	-64	FF C0	11111111 11000000

¿Cuándo usar `unsigned`?

- *No usar* sólo porque un número es no negativo
  - sin entender las consecuencias
  - es fácil cometer errores

```c
unsigned i;
for (i = cnt-2; i >= 0; i--)
    a[i] += a[i+1];
```

  - algunos muy sutiles

```c
#define DELTA sizeof(int)
int i;
for (i = CNT; i-DELTA >= 0; i-= DELTA) ...
```

- *Usar* cuando se opera con aritmética modular
  - Por ejemplo: aritmética de precisión arbitaria
- *Usar* cuando se utilizan los bits para representar información en grupos

Contar hacia abajo usando `unsigned`

- Manera correcta de usar **`unsigned`** en un ciclo

```c
unsigned i;
for (i = cnt - 2; i < cnt; --i)
    a[i] += a[i+1];
```

- el estándar de C garantiza que la suma **`unsigned`** es modular: $0 - 1 \rightarrow$ UMax
- Versión mejorada

```c
size_t i;
for (i = cnt - 2; i < cnt; --i)
    a[i] += a[i+1];
```

- Versión mejorada y concisa

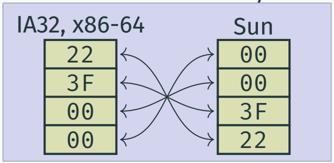
```c
while (cnt--)
    a[cnt] += a[cnt + 1]; /* ?`cuál es el problema? */
```

Tabla de contenidos

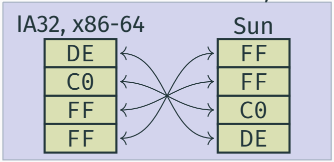
Representación en memoria: enteros

Decimal:	16162			
Binario:	0011	1111	0010	0010
Hex:	3	F	2	2

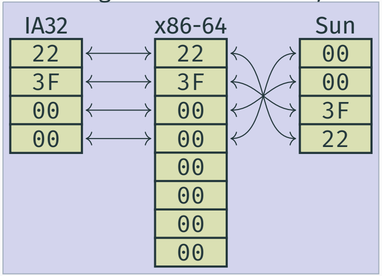
`int x = 16162;`



`int x = -16162;`

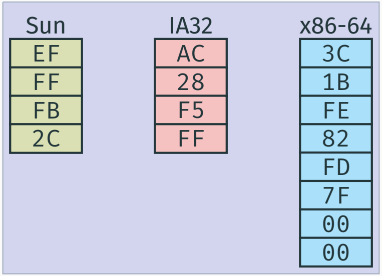


`long int x = 16162;`



Representación en memoria: punteros

```c
int i = -16162;
int *p = &i;
```



Representación en memoria: cadenas

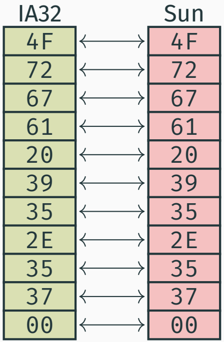
Cadenas en C

- Arreglo de `char`s
- Encoding ASCII
  - Set de caracteres estándar de 7-bit
  - El caracter `'0'` tiene el código `0x30`
    - Dígito `i`: código `0x30 + i`
  - El caracter `'a'` tiene el código `0x61`
  - El caracter `'A'` tiene el código `0x41`
- null-terminated (último caracter = 0)

Compatibilidad

- No hay problema con el endianness

```c
char s[] = "Orga 95.57";
```



Representación en memoria: programas

Cada programa se almacena como una secuencia de instrucciones

- *Cada instrucción es una operación*
  - una operación aritmética
  - leer o escribir la memoria
  - un salto condicional
- *Cada instrucción se codifica con una secuencia de bytes*
  - Reduced Instruction Set Computer (RISC)
    - Instrucciones simples y más veloces
    - Requiere más instrucciones para una operación compleja
  - Complex Instruction Set Computer (CISC)
    - Instrucciones complejas y más lentas
    - Requiere menos instrucciones para una operación compleja
- El tipo de instrucciones y su codificación varía de máquina a máquina
  - Código binario no compatible

Representación en memoria: programas

```c
int multstore (int x, int y, int *r) {
    *r = x * y;
    return *r;
}
```

```
$ $CC -std=c99 -o $ARCH-multsore -c -Og multstore.c
```

intel/gcc	arm/gcc	mac/gcc
89	FB	E0
F8	01	00
0F	F0	00
AF	00	91
C6	60	E5
89	10	82
02	47	00
C3	70	00
		E1
		2F
		FF
		1E

Examinando datos en C

Código para imprimir los datos

```
typedef unsigned char *byte_pointer;

void show_bytes(byte_pointer start, size_t len) {
    size_t i;

    for (i = 0; i < len; i++)
        printf("%p\t0x%02x\n",start+i, start[i]);
    printf("\n");
}
```

Especificadores de `printf`:

imprimir puntero: %p

imprimir valor hexadecimal: %x

Examinando datos en C

Código para imprimir los datos

```
typedef unsigned char *byte_pointer;

void show_bytes(byte_pointer start, size_t len) {
    size_t i;

    for (i = 0; i < len; i++)
        printf("%p\t0x%02x\n",start+i, start[i]);
    printf("\n");
}
```

Especificadores de `printf`:

imprimir puntero:	%p
imprimir valor hexadecimal:	%x

Castear *byte_pointer* a **`unsigned char *`** nos permite trabajar los datos como un arreglo de bytes

Ejemplo de ejecución

Código

```c
unsigned int manzana = 2953253037;
puts("unsigned int manzana = 2953253037;")
show_bytes((byte_pointer) &manzana, sizeof(unsigned int));
```

Resultado (GNU/Linux x86-64)

```
unsigned int manzana = 2953253037;
0x7ffc49e50c24  0xad
0x7ffc49e50c25  0x10
0x7ffc49e50c26  0x07
0x7ffc49e50c27  0xb0
```

Ejemplo de ejecución

Código

```c
unsigned int manzana = 2953253037;
puts("unsigned int manzana = 2953253037;")
show_bytes((byte_pointer) &manzana, sizeof(unsigned int));
```

Resultado (GNU/Linux x86-64)

```
unsigned int manzana = 2953253037;
0x7ffc49e50c24  0xad
0x7ffc49e50c25  0x10
0x7ffc49e50c26  0x07
0x7ffc49e50c27  0xb0
```

¿En qué *endianness* está guardado?

Descifrando código desensamblado

Desensamblado

- Texto que representa código de máquina binario
- Generado por un programa que lee el código de máquina

Ejemplo

```
Address      Instruction Code          Assembly
8048365:     5b                      pop  %ebx
8048366:     81 c3 ab 12 00 00       $0x12ab,%ebx
804836c:     83 bb 28 00 00 00 00    $0x0,0x28(%ebx)
```

Descifrando valores

Valor:	0x12ab
Pad a 32-bits:	0x000012ab
Separar en bytes:	00 00 12 ab
Invertir:	ab 12 00 00

Descifrando código desensamblado

Desensamblado

- Texto que representa código de máquina binario
- Generado por un programa que lee el código de máquina

Ejemplo

```
Address      Instruction Code          Assembly
8048365:     5b                        pop  %ebx
8048366:     81 c3 ab 12 00 00         $0x12ab,%ebx
804836c:     83 bb 28 00 00 00 00      $0x0,0x28(%ebx)
```

Descifrando valores

Valor:	`0x12ab`
Pad a 32-bits:	`0x000012ab`
Separar en bytes:	`00 00 12 ab`
Invertir:	`ab 12 00 00`

¿En qué *endianness* está guardado?

C Puzzles

- 32-bit *word-size*, complemento a dos
- Argumentar si es cierto, o dar un contraejemplo en las siguientes expresiones (x, y: int, ux, uy: unsigned)

```
1. x < 0               ⇒  ((x*2) < 0)      Falso:
2. ux >= 0
3. x & 7 == 7          ⇒  (x<<30) < 0
4. ux > -1
5. x > y               ⇒  -x < -y
6. x * x >= 0
7. x > 0 && y > 0      ⇒  x + y > 0
8. x >= 0              ⇒  -x <= 0
9. x <= 0              ⇒  -x >= 0
```

C Puzzles

- 32-bit *word-size*, complemento a dos
- Argumentar si es cierto, o dar un contraejemplo en las siguientes expresiones (x, y: int, ux, uy: unsigned)

```
1. x < 0              ⇒   ((x*2) < 0)      Falso:   TMin
2. ux >= 0
3. x & 7 == 7         ⇒   (x<<30) < 0
4. ux > -1
5. x > y              ⇒   -x < -y
6. x * x >= 0
7. x > 0 && y > 0     ⇒   x + y > 0
8. x >= 0             ⇒   -x <= 0
9. x <= 0             ⇒   -x >= 0
```

C Puzzles

- 32-bit *word-size*, complemento a dos
- Argumentar si es cierto, o dar un contraejemplo en las siguientes expresiones (x, y: int, ux, uy: unsigned)

```
1. x < 0             ⇒   ((x*2) < 0)
2. ux >= 0
3. x & 7 == 7        ⇒   (x<<30) < 0
4. ux > -1
5. x > y             ⇒   -x < -y
6. x * x >= 0
7. x > 0 && y > 0    ⇒   x + y > 0
8. x >= 0            ⇒   -x <= 0
9. x <= 0            ⇒   -x >= 0
```

Falso: TMin

Verdadero:

C Puzzles

- 32-bit *word-size*, complemento a dos
- Argumentar si es cierto, o dar un contraejemplo en las siguientes expresiones (x, y: int, ux, uy: unsigned)

```
1. x < 0              ⇒  ((x*2) < 0)
2. ux >= 0
3. x & 7 == 7         ⇒  (x<<30) < 0
4. ux > -1
5. x > y              ⇒  -x < -y
6. x * x >= 0
7. x > 0 && y > 0     ⇒  x + y > 0
8. x >= 0             ⇒  -x <= 0
9. x <= 0             ⇒  -x >= 0
```

Falso: TMin

Verdadero: 0 = UMin

C Puzzles

- 32-bit *word-size*, complemento a dos
- Argumentar si es cierto, o dar un contraejemplo en las siguientes expresiones (x, y: int, ux, uy: unsigned)

1. `x < 0` ⇒ `((x*2) < 0)` **Falso:** TMin
2. `ux >= 0` **Verdadero:** 0 = UMin
3. `x & 7 == 7` ⇒ `(x<<30) < 0` **Verdadero:**
4. `ux > -1`
5. `x > y` ⇒ `-x < -y`
6. `x * x >= 0`
7. `x > 0 && y > 0` ⇒ `x + y > 0`
8. `x >= 0` ⇒ `-x <= 0`
9. `x <= 0` ⇒ `-x >= 0`

C Puzzles

- 32-bit *word-size*, complemento a dos
- Argumentar si es cierto, o dar un contraejemplo en las siguientes expresiones (x, y: int, ux, uy: unsigned)

1. `x < 0` ⇒ `((x*2) < 0)` **Falso:** TMin
2. `ux >= 0` **Verdadero:** 0 = UMin
3. `x & 7 == 7` ⇒ `(x<<30) < 0` **Verdadero:** $x_1 = 1$
4. `ux > -1`
5. `x > y` ⇒ `-x < -y`
6. `x * x >= 0`
7. `x > 0 && y > 0` ⇒ `x + y > 0`
8. `x >= 0` ⇒ `-x <= 0`
9. `x <= 0` ⇒ `-x >= 0`

C Puzzles

- 32-bit *word-size*, complemento a dos
- Argumentar si es cierto, o dar un contraejemplo en las siguientes expresiones (x, y: int, ux, uy: unsigned)

1. `x < 0`	$\Rightarrow$	`((x*2) < 0)`	**Falso:**	TMin
2. `ux >= 0`			**Verdadero:**	0 = UMin
3. `x & 7 == 7`	$\Rightarrow$	`(x<<30) < 0`	**Verdadero:**	$x_1 = 1$
4. `ux > -1`			**Falso:**	
5. `x > y`	$\Rightarrow$	`-x < -y`		
6. `x * x >= 0`				
7. `x > 0 && y > 0`	$\Rightarrow$	`x + y > 0`		
8. `x >= 0`	$\Rightarrow$	`-x <= 0`		
9. `x <= 0`	$\Rightarrow$	`-x >= 0`		

C Puzzles

- 32-bit *word-size*, complemento a dos
- Argumentar si es cierto, o dar un contraejemplo en las siguientes expresiones (x, y: int, ux, uy: unsigned)

1. `x < 0`	⇒	`((x*2) < 0)`	**Falso:**	TMin
2. `ux >= 0`			**Verdadero:**	0 = UMin
3. `x & 7 == 7`	⇒	`(x<<30) < 0`	**Verdadero:**	$x_1 = 1$
4. `ux > -1`			**Falso:**	0
5. `x > y`	⇒	`-x < -y`		
6. `x * x >= 0`				
7. `x > 0 && y > 0`	⇒	`x + y > 0`		
8. `x >= 0`	⇒	`-x <= 0`		
9. `x <= 0`	⇒	`-x >= 0`		

C Puzzles

- 32-bit *word-size*, complemento a dos
- Argumentar si es cierto, o dar un contraejemplo en las siguientes expresiones (x, y: int, ux, uy: unsigned)

1. `x < 0`	$\Rightarrow$	`((x*2) < 0)`	**Falso:**	TMin
2. `ux >= 0`			**Verdadero:**	0 = UMin
3. `x & 7 == 7`	$\Rightarrow$	`(x<<30) < 0`	**Verdadero:**	$x_1 = 1$
4. `ux > -1`			**Falso:**	0
5. `x > y`	$\Rightarrow$	`-x < -y`	**Falso:**	
6. `x * x >= 0`				
7. `x > 0 && y > 0`	$\Rightarrow$	`x + y > 0`		
8. `x >= 0`	$\Rightarrow$	`-x <= 0`		
9. `x <= 0`	$\Rightarrow$	`-x >= 0`		

C Puzzles

- 32-bit *word-size*, complemento a dos
- Argumentar si es cierto, o dar un contraejemplo en las siguientes expresiones (x, y: int, ux, uy: unsigned)

1. `x < 0`	⇒	`((x*2) < 0)`	**Falso:**	TMin
2. `ux >= 0`			**Verdadero:**	0 = UMin
3. `x & 7 == 7`	⇒	`(x<<30) < 0`	**Verdadero:**	$x_1 = 1$
4. `ux > -1`			**Falso:**	0
5. `x > y`	⇒	`-x < -y`	**Falso:**	-1, TMin
6. `x * x >= 0`				
7. `x > 0 && y > 0`	⇒	`x + y > 0`		
8. `x >= 0`	⇒	`-x <= 0`		
9. `x <= 0`	⇒	`-x >= 0`		

C Puzzles

- 32-bit *word-size*, complemento a dos
- Argumentar si es cierto, o dar un contraejemplo en las siguientes expresiones (x, y: int, ux, uy: unsigned)

1.	`x < 0`	⇒	`((x*2) < 0)`	**Falso:** TMin
2.	`ux >= 0`			**Verdadero:** 0 = UMin
3.	`x & 7 == 7`	⇒	`(x<<30) < 0`	**Verdadero:** $x_1 = 1$
4.	`ux > -1`			**Falso:** 0
5.	`x > y`	⇒	`-x < -y`	**Falso:** -1, TMin
6.	`x * x >= 0`			**Falso:**
7.	`x > 0 && y > 0`	⇒	`x + y > 0`	
8.	`x >= 0`	⇒	`-x <= 0`	
9.	`x <= 0`	⇒	`-x >= 0`	

C Puzzles

- 32-bit *word-size*, complemento a dos
- Argumentar si es cierto, o dar un contraejemplo en las siguientes expresiones (x, y: int, ux, uy: unsigned)

1. `x < 0`	$\Rightarrow$	`((x*2) < 0)`	**Falso:**	TMin
2. `ux >= 0`			**Verdadero:**	0 = UMin
3. `x & 7 == 7`	$\Rightarrow$	`(x<<30) < 0`	**Verdadero:**	$x_1 = 1$
4. `ux > -1`			**Falso:**	0
5. `x > y`	$\Rightarrow$	`-x < -y`	**Falso:**	-1, TMin
6. `x * x >= 0`			**Falso:**	49374
7. `x > 0 && y > 0`	$\Rightarrow$	`x + y > 0`		
8. `x >= 0`	$\Rightarrow$	`-x <= 0`		
9. `x <= 0`	$\Rightarrow$	`-x >= 0`		

C Puzzles

- 32-bit *word-size*, complemento a dos
- Argumentar si es cierto, o dar un contraejemplo en las siguientes expresiones (x, y: int, ux, uy: unsigned)

1. `x < 0`	⇒	`((x*2) < 0)`	**Falso:**	TMin
2. `ux >= 0`			**Verdadero:**	0 = UMin
3. `x & 7 == 7`	⇒	`(x<<30) < 0`	**Verdadero:**	$x_1 = 1$
4. `ux > -1`			**Falso:**	0
5. `x > y`	⇒	`-x < -y`	**Falso:**	-1, TMin
6. `x * x >= 0`			**Falso:**	49374
7. `x > 0 && y > 0`	⇒	`x + y > 0`	**Falso:**	
8. `x >= 0`	⇒	`-x <= 0`		
9. `x <= 0`	⇒	`-x >= 0`		

C Puzzles

- 32-bit *word-size*, complemento a dos
- Argumentar si es cierto, o dar un contraejemplo en las siguientes expresiones (x, y: int, ux, uy: unsigned)

1. `x < 0`	$\Rightarrow$	`((x*2) < 0)`	**Falso:**	TMin
2. `ux >= 0`			**Verdadero:**	0 = UMin
3. `x & 7 == 7`	$\Rightarrow$	`(x<<30) < 0`	**Verdadero:**	$x_1 = 1$
4. `ux > -1`			**Falso:**	0
5. `x > y`	$\Rightarrow$	`-x < -y`	**Falso:**	-1, TMin
6. `x * x >= 0`			**Falso:**	49374
7. `x > 0 && y > 0`	$\Rightarrow$	`x + y > 0`	**Falso:**	TMax, TMax
8. `x >= 0`	$\Rightarrow$	`-x <= 0`		
9. `x <= 0`	$\Rightarrow$	`-x >= 0`		

C Puzzles

- 32-bit *word-size*, complemento a dos
- Argumentar si es cierto, o dar un contraejemplo en las siguientes expresiones (x, y: int, ux, uy: unsigned)

1. `x < 0`	⇒	`((x*2) < 0)`	**Falso:**	TMin
2. `ux >= 0`			**Verdadero:**	0 = UMin
3. `x & 7 == 7`	⇒	`(x<<30) < 0`	**Verdadero:**	$x_1 = 1$
4. `ux > -1`			**Falso:**	0
5. `x > y`	⇒	`-x < -y`	**Falso:**	-1, TMin
6. `x * x >= 0`			**Falso:**	49374
7. `x > 0 && y > 0`	⇒	`x + y > 0`	**Falso:**	TMax, TMax
8. `x >= 0`	⇒	`-x <= 0`	**Verdadero:**	
9. `x <= 0`	⇒	`-x >= 0`		

C Puzzles

- 32-bit *word-size*, complemento a dos
- Argumentar si es cierto, o dar un contraejemplo en las siguientes expresiones (x, y: int, ux, uy: unsigned)

1. `x < 0`	⇒	`((x*2) < 0)`	**Falso:**	TMin
2. `ux >= 0`			**Verdadero:**	0 = UMin
3. `x & 7 == 7`	⇒	`(x<<30) < 0`	**Verdadero:**	$x_1 = 1$
4. `ux > -1`			**Falso:**	0
5. `x > y`	⇒	`-x < -y`	**Falso:**	-1, TMin
6. `x * x >= 0`			**Falso:**	49374
7. `x > 0 && y > 0`	⇒	`x + y > 0`	**Falso:**	TMax, TMax
8. `x >= 0`	⇒	`-x <= 0`	**Verdadero:**	-TMax < 0
9. `x <= 0`	⇒	`-x >= 0`		

C Puzzles

- 32-bit *word-size*, complemento a dos
- Argumentar si es cierto, o dar un contraejemplo en las siguientes expresiones (x, y: int, ux, uy: unsigned)

1. `x < 0`	$\Rightarrow$	`((x*2) < 0)`	**Falso:**	TMin
2. `ux >= 0`			**Verdadero:**	0 = UMin
3. `x & 7 == 7`	$\Rightarrow$	`(x<<30) < 0`	**Verdadero:**	$x_1 = 1$
4. `ux > -1`			**Falso:**	0
5. `x > y`	$\Rightarrow$	`-x < -y`	**Falso:**	-1, TMin
6. `x * x >= 0`			**Falso:**	49374
7. `x > 0 && y > 0`	$\Rightarrow$	`x + y > 0`	**Falso:**	TMax, TMax
8. `x >= 0`	$\Rightarrow$	`-x <= 0`	**Verdadero:**	-TMax < 0
9. `x <= 0`	$\Rightarrow$	`-x >= 0`	**Falso:**	

C Puzzles

- 32-bit *word-size*, complemento a dos
- Argumentar si es cierto, o dar un contraejemplo en las siguientes expresiones (x, y: int, ux, uy: unsigned)

1. `x < 0`	$\Rightarrow$	`((x*2) < 0)`	**Falso:**	TMin
2. `ux >= 0`			**Verdadero:**	0 = UMin
3. `x & 7 == 7`	$\Rightarrow$	`(x<<30) < 0`	**Verdadero:**	$x_1 = 1$
4. `ux > -1`			**Falso:**	0
5. `x > y`	$\Rightarrow$	`-x < -y`	**Falso:**	-1, TMin
6. `x * x >= 0`			**Falso:**	49374
7. `x > 0 && y > 0`	$\Rightarrow$	`x + y > 0`	**Falso:**	TMax, TMax
8. `x >= 0`	$\Rightarrow$	`-x <= 0`	**Verdadero:**	-TMax < 0
9. `x <= 0`	$\Rightarrow$	`-x >= 0`	**Falso:**	TMin

Tabla de contenidos

Code Security

```c
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

- ¿Hay alguna vulnerabilidad a la vista?
- **Hay *legiones* de personas buscando vulnerabilidades en los programas**.. y no siempre para reportarlas y mejorar el software.

Code Security: uso típico

```c
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```c
#define MSIZE 528
void getstuff(void) {
  char buf[MSIZE];
  copy_from_kernel(buf, MSIZE);
  printf("%s\n", buf);
}
```

Code Security: uso malicioso

```c
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```c
#define MSIZE 528
void getstuff(void) {
  char buf[MSIZE];
  copy_from_kernel(buf, -MSIZE);
  printf("%s\n", buf);
}
```

```c
/* Declaration of library function
     memcpy */
void *
memcpy(void *dest,
             void *src,
             size_t n)
```

Licencia del estilo de beamer

Obtén el código de este estilo y la presentación demo en

`github.com/pamoreno/mtheme`

El estilo *en sí* está licenciado bajo la Creative Commons Attribution-ShareAlike 4.0 International License. El estilo es una modificación del creado por Matthias Vogelgesang, disponible en

`github.com/matze/mtheme`