Arquitetura de Computadores

LEI – 2021/22

DI-FCT/NOVA

Semana 2 - Representação de Informação

Sumário

- Representação de informação
- Representação de inteiros
  - Aritmética com números inteiros com sinal.
- Representação de reais

Bibliografia:

Bryant, O'Hallaron Computer Systems: A Programmer's Perspective, 2nd ou 3rd Ed, Cap 2

João M. Fernandes, Essentials of computing systems, Coleção Educação | Ciências, Engenharia e Tecnologia, Cap 2. secções 2.1 a 2.4 (https://doi.org/10.21814/uminho.ed.33)

Representação da informação

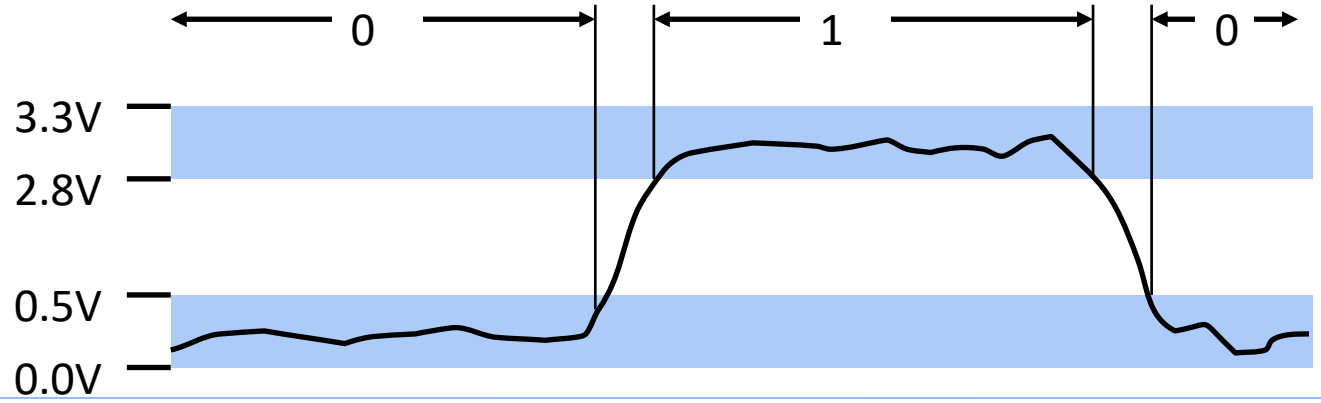
- Dispositivos eletrónicos existem em dois estados diferentes
  - Permitem representar um Bit
    - A um dos estados associa-se um 0 e ao outro um 1
  - Memória organizada em bytes (8 bits)
    - Códigos máquina
    - Dados
  - CPUs suportam múltiplos tamanhos de dados
    - Sempre um número inteiro de byte
    - Neste curso assume-se que o tamanho “normal” de transferência é 4 bytes (32 bits)

Porque é que os computadores não usam base 10?

- Representação de números em base 10
- Implementação electónica
  - Difícil de armazenar
    - ENIAC (1º computador digital) usava 10 válvulas / dígito
  - Difícil de transmitir
  - Complicado implementar operações aritméticas e lógicas

Representação Binária

- Representação de números em Base 2
  - Representa 1521310 como 11101101101101_2
  - Representa 1.2010 como $1.0011001100110011[0011]\ldots_2$
  - Representa 1.5213×10^4 como $1.1101101101101_2 \times 2^{13}$
- Implementação Electrónica
  - Fácil de armazenar com elementos que são estáveis em dois valores
  - Transmissão fiável em fios com ruído



Memória Organizada por Bytes

- Programas emitem endereços
  - A memória é conceptualmente um vector de bytes muito grande
  - Implementada por uma hieraquia de memória de vários tipos

- Compilador + Sistema Operativo controlam a memória reservada a um programa
  - Onde as várias partes do programa são carregadas
  - Há um espaço único por programa
  - Há várias zonas (código, dados, …)

Codificação de Valores em Bytes

- Byte = 8 bits
  - Binário 00000000_2 a 11111111_2
    - 1011010 é escrito em C como

      `0b1011010`
  - Decimal: 0_{10} a 255_{10}
  - Hexadecimal 00_{16} a FF_{16}
    - Representação em Base 16
    - Usa caracteres '0' a '9' e 'A' a 'F'
    - $FA1D37B_{16}$ é escrito em C como

      `0xFA1D37B` ou `0xfa1d37b`

Hex	Decimal	Binary
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

Representação de informação em binário

- Quantos valores diferentes podem ser representados com uma sequência de n bits?

- 2 bits :
  - 2^2 = 4 valores distintos

00 ➔ 2^2

01

10

11

Representação de informação em binário

- Quantos valores diferentes podem ser representados com uma sequência de **n** bits?

- **3** bits :
  - 2^3 = 8 valores distintos

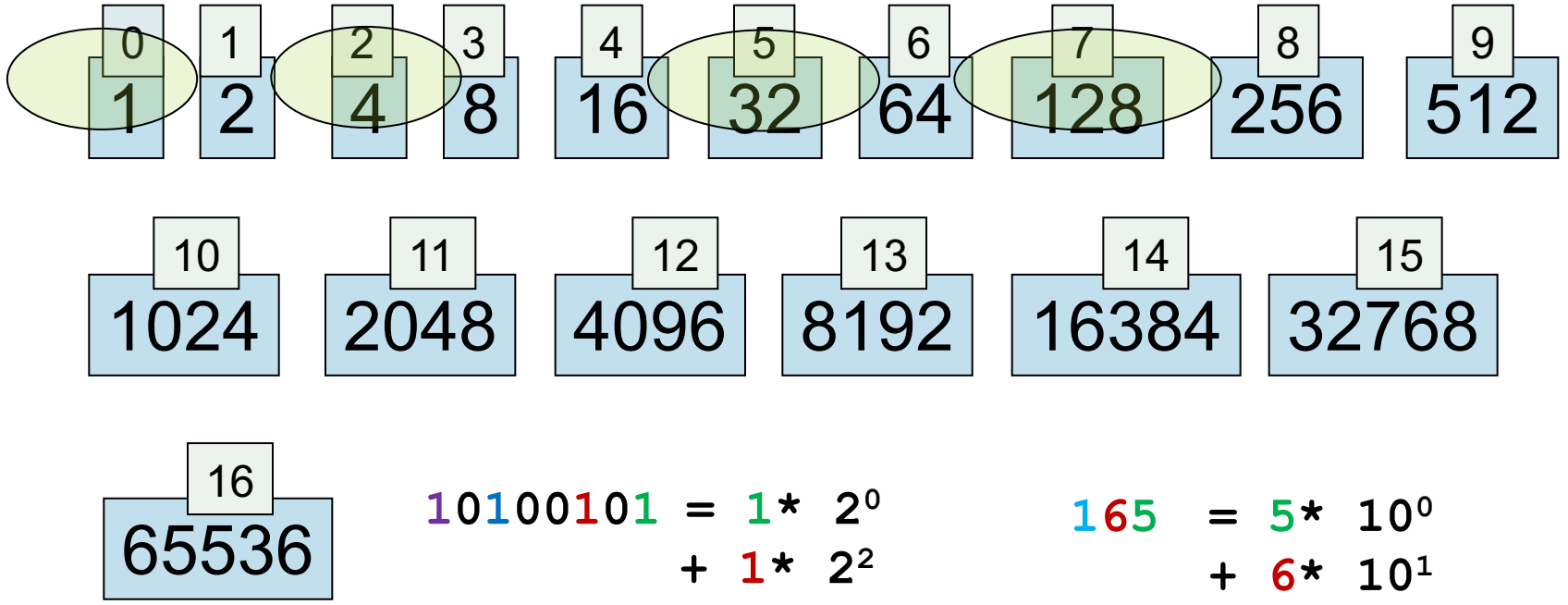
000 → 2^3
001
010
011
100
101
110
111

Representação de informação em binário

- Quantos valores diferentes podem ser representados com uma sequência de **n** bits?

- **n** bits :
  - 2^n valores distintos

Potências de dois



$10100101 = 1*2^0 + 1*2^2 + 1*2^5 + 1*2^7 = 165$

$165 = 5*10^0 + 6*10^1 + 1*10^2 = 165$

Representação de informação em binário

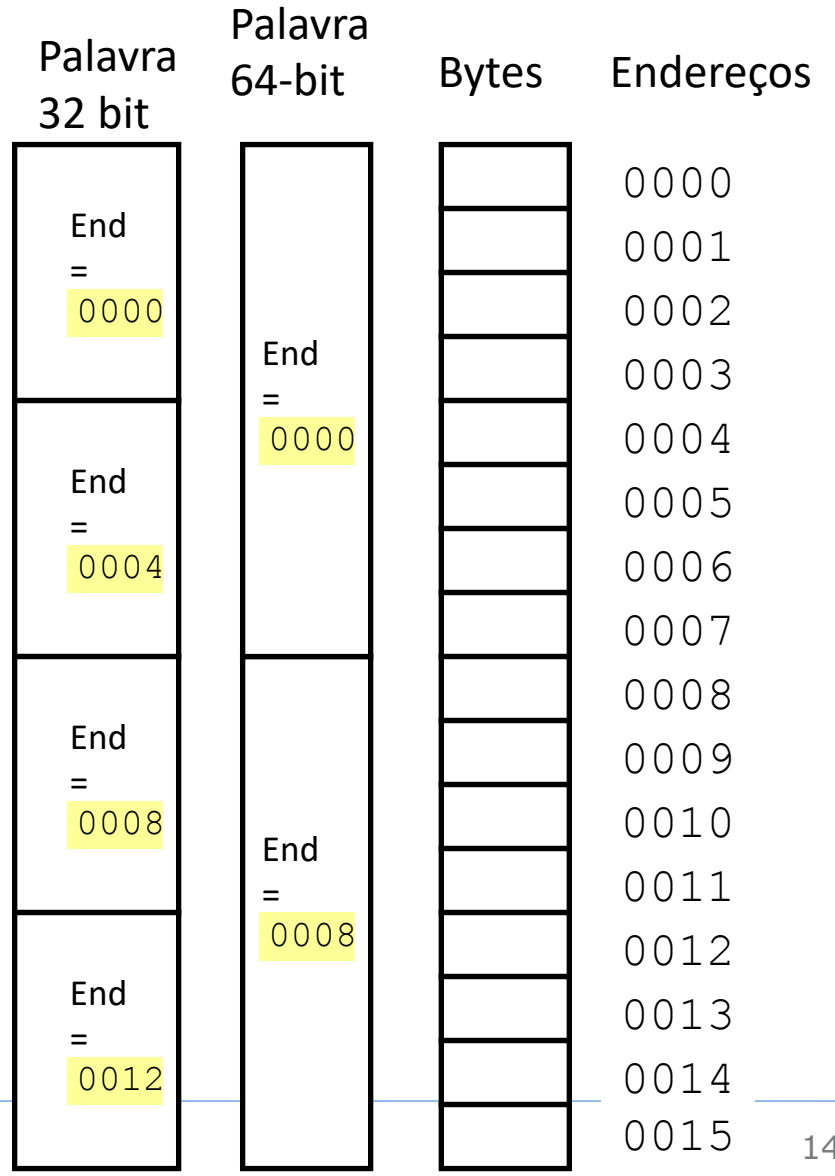
• 1 bit de informação	dois valores distintos
• 1 byte de informação = 8 bits	$2^8 = 256$ valores distintos
• 2 bytes de informação = 16 bits	$2^{16} = 65.536$ valores distintos
• 4 bytes de informação = 32 bits	$2^{32} = 4.294.967.296$ valores distintos

Palavras Máquina

- Cada CPU tem um "**Tamanho de Palavra**"
  - Tamanho usado para os dados inteiros
    - Incluindo endereços
  - Máquina usada neste curso tem 32 bits (4 bytes)
    - Endereços limitados a 4GB
  - Muitos sistemas modernos com 64 bits (8 bytes)
    - Endereça potencialmente $\approx 1.8 \times 10^{19}$ bytes
  - CPUs suportam múltiplos tamanhos de dados
    - Fracções ou múltiplos do tamanho da palavra
    - Sempre um número inteiro de bytes

Organização de Memória Orientada para a Palavra

- Endereços especificam a localização de Bytes
  - Endereço do 1º byte da palavra
  - Endereços de palavras sucessivas diferem de 4 (32-bit) ou 8 (64-bit)



Representações de Dados

- Tamanho de dados do C (em Bytes)
  - char 1
  - short 2
  - int 2 ou 4
  - long int 4 ou 8
  - long long int 8

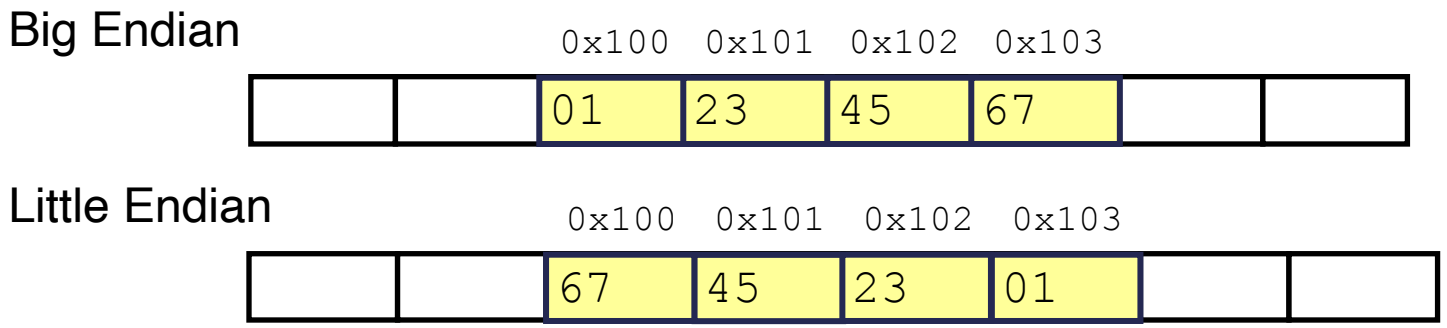
  - float 4
  - double 8
  - long double 10 ou 16

Ordenação dos Bytes

- Como ordenar uma palavra com múltiplos bytes em memória?
- Convenções
  - CPUs “Big Endian”
    - byte menos significativo tem o maior endereço
  - CPUs “Little Endian”
    - byte menos significativo tem o menor endereço

Ordenação de Bytes

- CPUs “Big Endian”
  - byte menos significativo tem o maior endereço
- CPUs “Little Endian”
  - byte menos significativo tem o menor endereço
- Exemplo
  - Variável **x** tem o valor com 4-bytes 0x01234567
  - O **endereço de x** é 0x100



Representação de Inteiros

```
int A = 15213;
int B = -15213;
long C = 15213;
long long D = 15213;
```

Decimal:	15213
Binary:	0011 1011 0110 1101
Hex:	3 B 6 D

IA32 A

E:	6D
E+1:	3B
E+2:	00
E+3:	00

IA32 B

E:	93
E+1:	C4
E+2:	FF
E+3:	FF

IA32 C

E:	6D
E+1:	3B
E+2:	00
E+3:	00

IA32 D

6D
3B
00
00
00
00
00
00

Representação em complemento para 2 (a seguir)

Representação de texto

- Texto não formatado (caracteres sem atributos: fonte, cor, …)
- O número de símbolos é finito, pode-se atribuir uma sequência de bits diferente a cada um

  Codificação (character set)

- Uma tabela que lista os caracteres e o código que lhe corresponde

- Há várias normas que são aceites pelos fabricantes de hardware e software

Codificação ASCII

- **ASCII** significa American Standard Code for Information Interchange
- ASCII inicialmente 7 bits, o que permite 128 caracteres diferentes; **estendido** para 8 bits o que permite 256 caracteres diferentes
- 1 carácter codificado em ASCII 8 bits ocupa um byte

Codificação ASCII de 7 bits

Left Digit(s) \ *Right Digit*	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
	ASCII									
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
1	LF	VT	FF	CR	SO	SI	DLE	DC1	DC2	DC3
2	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
3	RS	US	□	!	“	#	$	%	&	’
4	(	)	*	+	,	–	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	DEL		

Codificação Unicode

- **Extended ASCII** não é suficiente para representar os símbolos usados nas línguas

- A codificação **Unicode** usa 16 bits por carácter

  Quantos caracteres é que esta codificação suporta?

- O Unicode inclui o ASCII: os 1ºs 256 caracteres do Unicode correspondem ao extended ASCII (de 8 bits) 00 00 a 00 FF

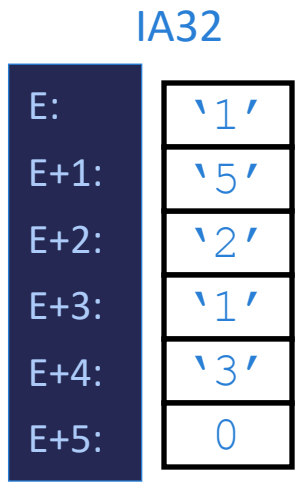
Exemplos da codifcação Unicode

Code (Hex)	Character	Source
0041	A	English (Latin)
042F	Я	Russian (Cyrillic)
0E09	ฉ	Thai
13EA	Ꮺ	Cherokee
211E	℞	Letterlike Symbols
21CC	⇌	Arrows
282F	⠯	Braille
345F	㑟	Chinese/Japanese/ Korean (Common)

Representação de Strings

- Strings em C
  - Representadas por um vector de caracteres
  - Cada carácter codificado no format ASCII
    - O Código ASCII do carácter c é normalmente representado por ‘c’
  - Strings terminadas pelo valor 0, que é o código do carácter ‘\0’ ou NUL

```
char S[6] = "15213";
```



- Compatibilidade
  - Como os dados têm só um byte não há diferenças na ordenação
  - Ficheiros de texto independentes da plataforma (salvo por diferenças na terminação das linhas)

Codificação ASCII de 7 bits

Left Digit(s) \ *Right Digit*	*ASCII* 0	1	2	3	4	5	6	7	8	9
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
1	LF	VT	FF	CR	SO	SI	DLE	DC1	DC2	DC3
2	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
3	RS	US	□	!	“	#	$	%	&	´
4	(	)	*	+	,	–	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	DEL		

Código ASCII de ‘1’ é 49

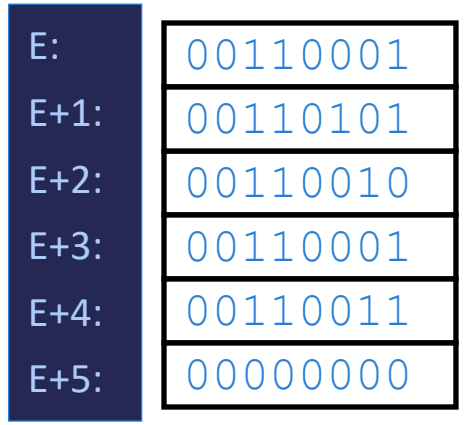
Representação de Strings

- Strings em C
  - Representadas por um vector de caracteres
  - Cada carácter codificado no format ASCII
  - Strings terminadas por um NULL (0)
    - Carácter final = 0

- O código ASCII de ‘1’ é $49_{10} = 00110001_{2}$ (a 8 bits)

```
char S[6] = "15213";
```

IA32



Representação do Código Máquina

- Um programa é uma sequência de instruções máquina simples
  - Operação aritmética
  - Ler ou escrever memória
  - Saltos
- Instruções codificadas como bytes
  - Todas do mesmo tamanho
  - Tamanho variável (IA-32)
- As codificações das instruções são diferentes de CPU para CPU
  - Código não é compatível entre CPUs diferentes

- **Os programas também são sequências de bytes!**

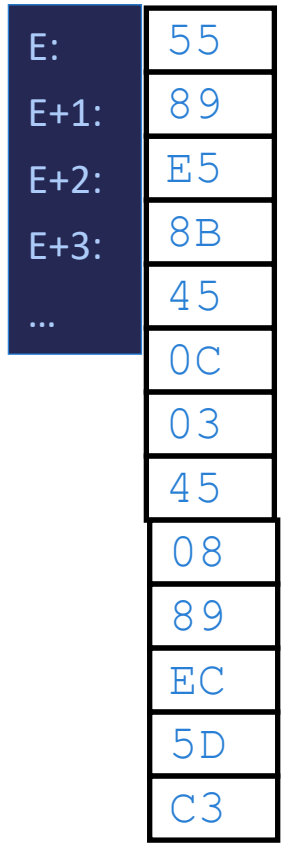
Representação de Instruções

```
int sum(int x, int y){
    return x+y;
}
```

IA32 usa 7 instruções com comprimentos de 1, 2 ou 3 bytes

As instruções são as mesmas no Windows/IA32 e Linux/IA32 mas os programas de ambas as plataformas não são “binary compatible”

IA32 `sum`



Máquinas diferentes usam instruções diferentes e codificações diferentes !

Representação de Números Inteiros

Representação de inteiros

- Com n bits, podem-se representar 2^n valores diferentes.

- Se representarmos números inteiros sem sinal, os valores são:

  de 0 a $2^n - 1$.

  com **32 bits**: de 0 a 4.294.967.295

- Se representarmos números inteiros com sinal, os valores são:

  de -2^{n-1} a $2^{n-1} - 1$.

  com **32 bits**: de -2.147.483.648 a +2.147.483.647

Codificação de inteiros em w bits

Sem sinal

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Com sinal, Complemento para 2

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Bit de Sinal

```
short int x =  15213;
short int y = -15213;
```

- short em C ocupa 2 bytes

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

- Bit de sinal
  - Em complemento para 2, o bit mais significativo é o sinal
    - 0 para não negativo
    - 1 para negativo

Exemplo

```
x =      15213: 00111011 01101101
y =     -15213: 11000100 10010011
```

Weight	15213		-15213	
1	1	1	1	1
2	0	0	1	2
4	1	4	0	0
8	1	8	0	0
16	0	0	1	16
32	1	32	0	0
64	1	64	0	0
128	0	0	1	128
256	1	256	0	0
512	1	512	0	0
1024	0	0	1	1024
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
Sum		**15213**		**-15213**

Valores representáveis

- Sem sinal
  - UMin = 0
    - 000…0
  - UMax = $2^w - 1$
    - 111…1
- Valores em complemento para 2
  - TMin = -2^{w-1}
    - 100…0
  - TMax = $2^{w-1} - 1$
    - 011…1

Valores para $W = 16$

	Decimal	Hex	Binary
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

Valores para vários tamanhos de palavra

	W			
	8	16	32	64
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

Observações

- |TMin | = TMax + 1
  - Assimétrico
- Umax = 2 * TMax + 1

Programação em C

```
#include <limits.h>
```

Ver livro K&R (App. B11)

Declara, por exemplo, `ULONG_MAX`, `LONG_MAX`, `LONG_MIN`

Valores dependem da plataforma (32b/64b)

Programa para ver os valores de ULONG_MAX, …

int a = 16;

int* p = **&**a;

& → endereço de

* → apontador (variável que contém um endreço)

T* → apontador para um posição de memória com tipo T (mais detalhes numa próxima aula)

```c
#include <stdio.h>
#include <limits.h>

int main() {
   int a = 16;
   int* p = &a;
   char c1 = 65;
   printf("value of c1 = %c\n", c1);
   printf("value of c1 = %d\n", c1);

   printf("Size of variable a : %ld\n",(unsigned long)sizeof(a));
   printf("Size of int data type : %ld\n",(unsigned long)sizeof(int));
   printf("Size of long int data type : %ld\n",(unsigned long)sizeof(long int));
   printf("Size of long long int data type : %ld\n",(unsigned long)sizeof(long long int));
   printf("Size of char data type : %ld\n",(unsigned long)sizeof(char));
   printf("Size of float data type : %ld\n",(unsigned long)sizeof(float));
   printf("Size of double data type : %ld\n",(unsigned long)sizeof(double));
   printf("Size of pointer : %ld\n",(unsigned long)sizeof(p));
   printf("Value of pointer p: %p\n",p);

   printf("The number of bits in a byte %d\n", CHAR_BIT);

   printf("The minimum value of SIGNED CHAR = %d\n", SCHAR_MIN);
   printf("The maximum value of SIGNED CHAR = %d\n", SCHAR_MAX);
   printf("The maximum value of UNSIGNED CHAR = %d\n", UCHAR_MAX);

   printf("The minimum value of SHORT INT = %d\n", SHRT_MIN);
   printf("The maximum value of SHORT INT = %d\n", SHRT_MAX);

   printf("The minimum value of INT = %d\n", INT_MIN);
   printf("The maximum value of INT = %d\n", INT_MAX);

   printf("The minimum value of CHAR = %d\n", CHAR_MIN);
   printf("The maximum value of CHAR = %d\n", CHAR_MAX);

   printf("The minimum value of LONG = %ld\n", LONG_MIN);
   printf("The maximum value of LONG = %ld\n", LONG_MAX);

   return(0);
}
```

Resultados Windows 10, gcc 9.2.0
CPU em modo 32 bit

```
C:\Users\pm\Desktop>gcc -m32 -Wall -o limits-32.exe limits.c

C:\Users\pm\Desktop>.\limits-32.exe
value of c1 = A
value of c1 = 65
Size of variable a : 4
Size of int data type : 4
Size of long int data type : 4
Size of long long int data type : 8
Size of char data type : 1
Size of float data type : 4
Size of double data type : 8
Size of pointer : 4
Value of pointer p: 0064FEC4
The number of bits in a byte 8
The minimum value of SIGNED CHAR = -128
The maximum value of SIGNED CHAR = 127
The maximum value of UNSIGNED CHAR = 255
The minimum value of SHORT INT = -32768
The maximum value of SHORT INT = 32767
The minimum value of INT = -2147483648
The maximum value of INT = 2147483647
The minimum value of CHAR = -128
The maximum value of CHAR = 127
The minimum value of LONG = -2147483648
The maximum value of LONG = 2147483647
```

Resultados Windows 10, gcc 9.2.0 CPU em modo 64 bit

```
C:\Users\pm\Desktop>gcc -Wall -o limits-64.exe limits.c

C:\Users\pm\Desktop>.\limits-64.exe
value of c1 = A
value of c1 = 65
Size of variable a : 4
Size of int data type : 4
Size of long int data type : 4
Size of long long int data type : 8
Size of char data type : 1
Size of float data type : 4
Size of double data type : 8
Size of pointer : 8
Value of pointer p: 000000000065FE10
The number of bits in a byte 8
The minimum value of SIGNED CHAR = -128
The maximum value of SIGNED CHAR = 127
The maximum value of UNSIGNED CHAR = 255
The minimum value of SHORT INT = -32768
The maximum value of SHORT INT = 32767
The minimum value of INT = -2147483648
The maximum value of INT = 2147483647
The minimum value of CHAR = -128
The maximum value of CHAR = 127
The minimum value of LONG = -2147483648
The maximum value of LONG = 2147483647
```

Complemento para 2 = Negação bit a bit + 1

- A seguinte propriedade é verdadeira

```
~x + 1 == -x
```

$\sim x + x == 1111\ldots11_{\mathbf{2}} == -1$

x	1	0	0	1	1	1	0	1
+ ~x	0	1	1	0	0	0	1	0
-1	1	1	1	1	1	1	1	1

```
~x + 1 = ~x + 1 + (x - x)
       = (~x + x) + 1 - x
       = -1 + 1 -x
       = -x
```

Exemplos

x = 15213

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
~x	-15214	C4 92	11000100 10010010
~x+1	-15213	C4 93	11000100 10010011
y	-15213	C4 93	11000100 10010011

0

	Decimal	Hex	Binary
0	0	00 00	00000000 00000000
~0	-1	FF FF	11111111 11111111
~0+1	0	00 00	00000000 00000000

Representação de inteiros com sinal

- O bit mais significativo é o sinal
- A representação usa complemento para 2
- O complemento para 2 (em base 2) de um número X com n dígitos é 2^n-X
- Exemplo:
  - complemento para 2 de 0011_2 é $2^4 - 0011_2$
  - $10000_2 - 0011_2 = 1101_2$

$$\begin{array}{r} 1\,0\,0\,0\,0 \\ -\;0\,0\,1\,1 \\ \hline 1\,1\,0\,1 \end{array}$$

Representação de inteiros com sinal

- Na prática é mais fácil, fazer o complemento para 1 e depois somar 1
- Exemplos:

$23_{10} = 00010111_2$

$-23_{10} = -(00010111_2) =$

$= 11101000_2 + 1 =$

$= 11101001_2$

Inicial:	0 0 0 1 0 1 1 1
Compl. para 1	1 1 1 0 1 0 0 0
Somando 1	1 1 1 0 1 0 0 1

$-9_{10} = -(00001001_2) = 11110110_2 + 1 = 11110111_2$

$--9_{10} = -(11110111_2) = 000010000_2 + 1 = 00001001_2 = 9_{10}$

Inteiros com e sem sinal

- Para números positivos a representação é igual

Valor	Sem sinal	Com sinal
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

Conversão (Cast)

- C permite conversões de *Signed* para *Unsigned*

```
short int           x =  15213;
unsigned short int ux = (unsigned short) x;
short int           y  = -15213;
unsigned short int uy = (unsigned short) y;
```

- Valor resultante
  - Não há mudanças na representação
  - Os valores não negativos não são alterados
    - *ux = 15213*
  - Os valores negativos tornam-se (grandes) valores positivos
    - *uy = 50323*

Relação entre Signed e Unsigned

Weight	-15213		50323	
1	1	1	1	1
2	1	2	1	2
4	0	0	0	0
8	0	0	0	0
16	1	16	1	16
32	0	0	0	0
64	0	0	0	0
128	1	128	1	128
256	0	0	0	0
512	0	0	0	0
1024	1	1024	1	1024
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*
Sum		**-15213**		**50323**

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

- uy = y + 2 * 32768 = y + 65536
- -15213 + 65536 = 50323

Signed vs. Unsigned em C

- Constantes
  - Por omissão, são considerados como tendo sinal
  - Unsigned se têm o sufixo“U”

```
4294967259U
```

- *Casting (com as consequências descritas atrás)*
  - Explícito

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

  - *Casting* ocorre através de atribuições e em invocações de funções

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

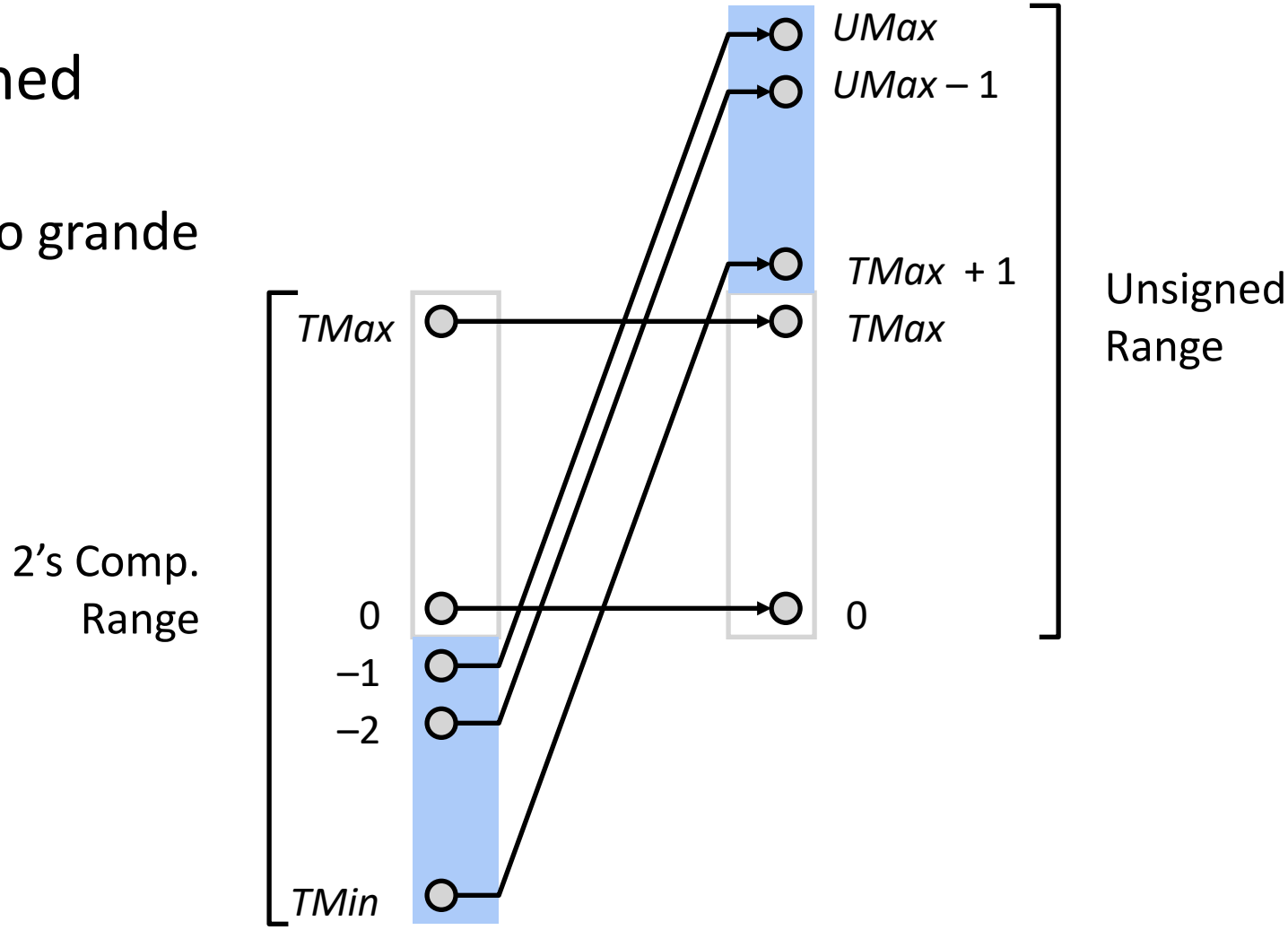
Surpresas associadas ao Casting

- Expression Evaluation
  - Se se misturam *unsigned* e *signed* numa expressão, os valores *signed* são *cast implicitamente* para *unsigned*
  - Incluindo comparações <, >, ==, <=, >=
  - Exemplos para 32 bits

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

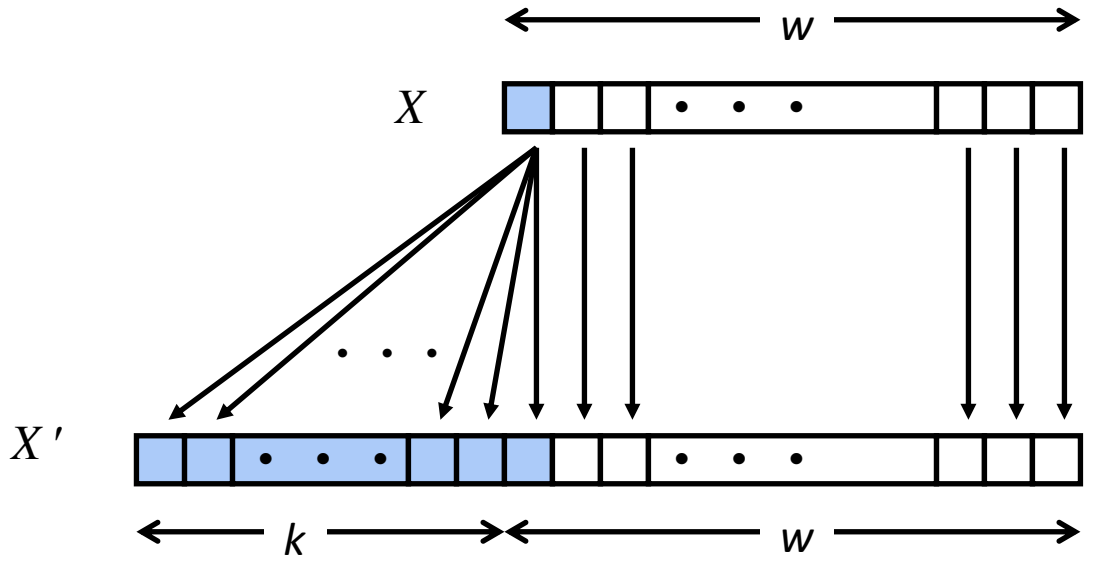
Explicação das surpresas

- 2's Comp. → Unsigned
  - Inversão da ordem
  - Negativo → Positivo grande



Extensão de sinal

- Problema:
  - Dado um inteiro *x* com sinal representado em ***w*-bits**, converter para um inteiro em ***w+k*-bits** com o **mesmo valor**.
- Regra:
  - Fazer *k* cópias do bit de sinal :
  - $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ cópias do MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$



Exemplo de extensão de sinal

```
short int x =  15213;
int      ix = (int) x;
short int y = -15213;
int      iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

- Conversão de inteiro com menos bits para mais bits
- C faz automaticamente extensão de sinal

Operações aritméticas com um número fixo de bits

- Se somarmos 2.147.483.647 (intmax) com 10 deveria dar

  2.147.483.657,

  que não é representável com 32 bits (com sinal), porque é maior do que intmax (2.147.483.647).

- O mesmo problema ocorre quando o resultado de uma operação for menor do que -2.147.483.648 (intmin)

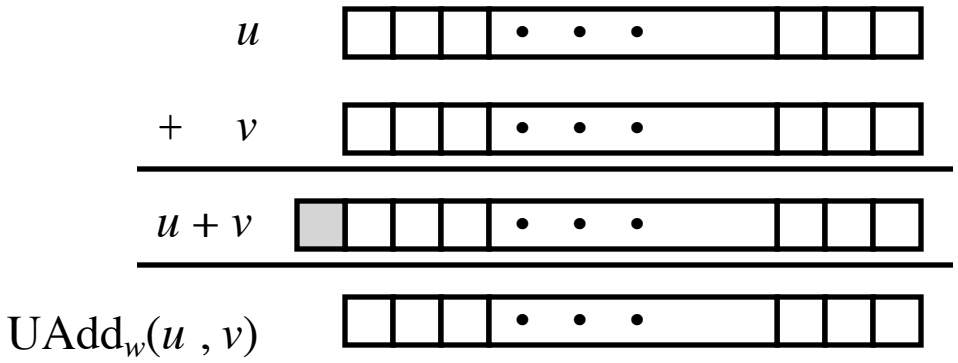
- A aritmética dos computadores reserva-nos algumas surpresas !

Soma sem sinal

Operandos: *w* bits

Resultado real: *w*+1 bits

Ignorar Transporte (Carry): *w* bits



- Soma habitual
  - Ignora carry
- Implementa Aritmética Modular
  - s = $UAdd_w(u , v) = (u + v) \mod 2^w$

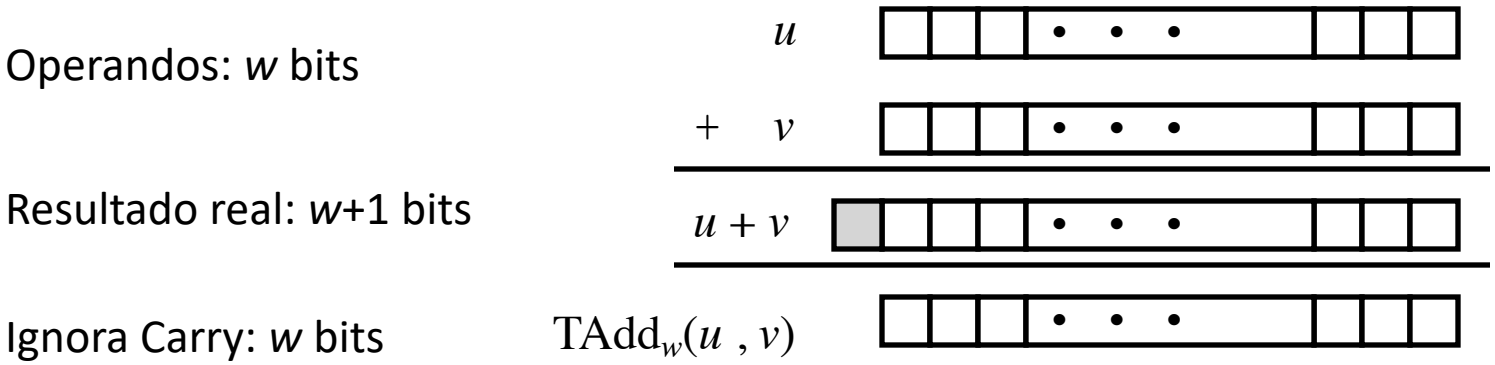
$$UAdd_w(u,v) = \begin{cases} u+v & u+v < 2^w \\ u+v-2^w & u+v \geq 2^w \end{cases}$$

Soma e subtracção

- X – Y = X + (-Y)
- 9 - 23 = 9 + (-23) = -14

$$
\begin{array}{rl}
 & 0\,0\,0\,0\,1\,0\,0\,1\ (9) \\
+ & 1\,1\,1\,0\,1\,0\,0\,1\ (-23) \\
\hline
 & 1\ 1\,1\,1\,0\,0\,1\,0\ (-14)
\end{array}
$$

Soma em Complemento para 2



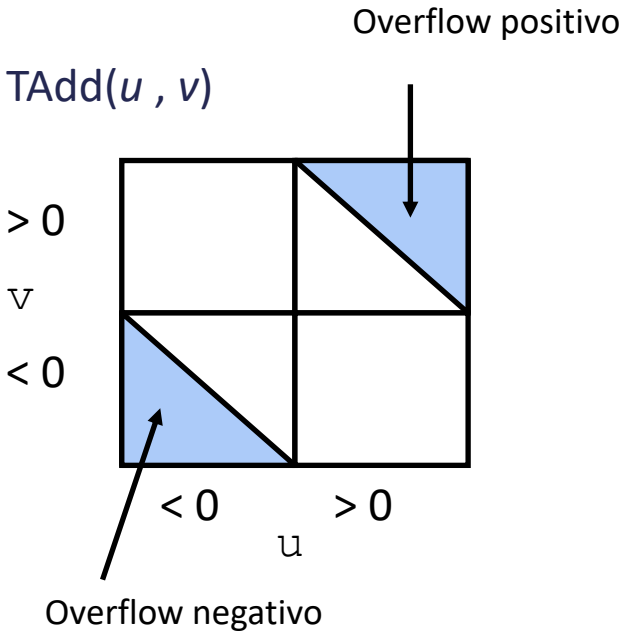
- A soma com e sem sinal dão os mesmos bits como resultado
  - Se fizermos em C:

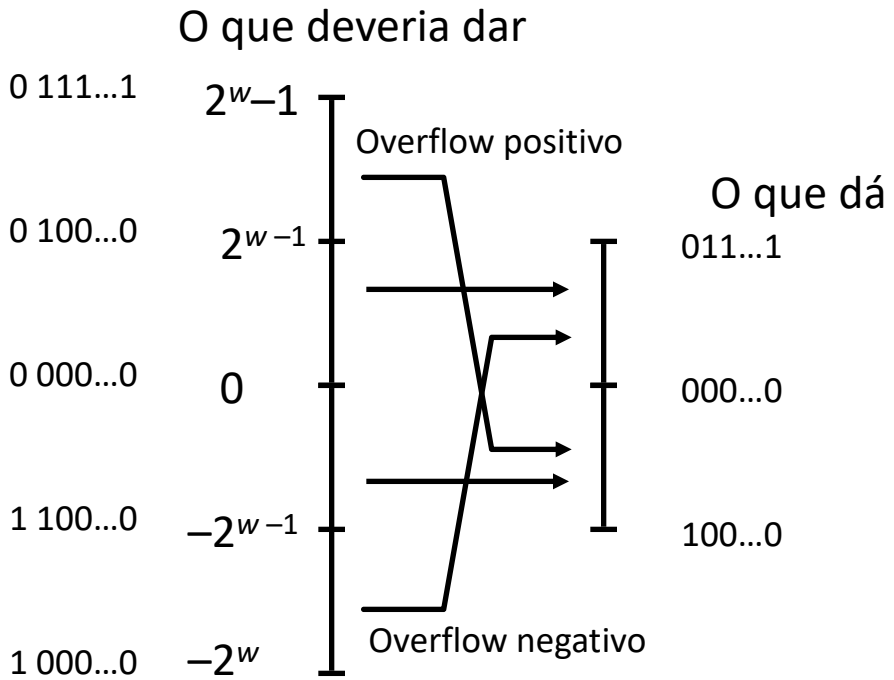
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
 t = u + v
```

  - Os dois são iguais

Caracterização da soma com sinal

- Funcionalidade
  - Seriam requeridos *w*+1 bits
  - MS bit ignorado
  - Os w bits estão em comp. para w2





$$TAdd_w(u,v) = \begin{cases} u+v+2^{w-1} & u+v < TMin_w \quad \text{(NegOver)} \\ u+v & TMin_w \le u+v \le TMax_w \\ u+v-2^{w-1} & TMax_w < u+v \quad \text{(PosOver)} \end{cases}$$

Deteção do Overflow em Complemento para 2

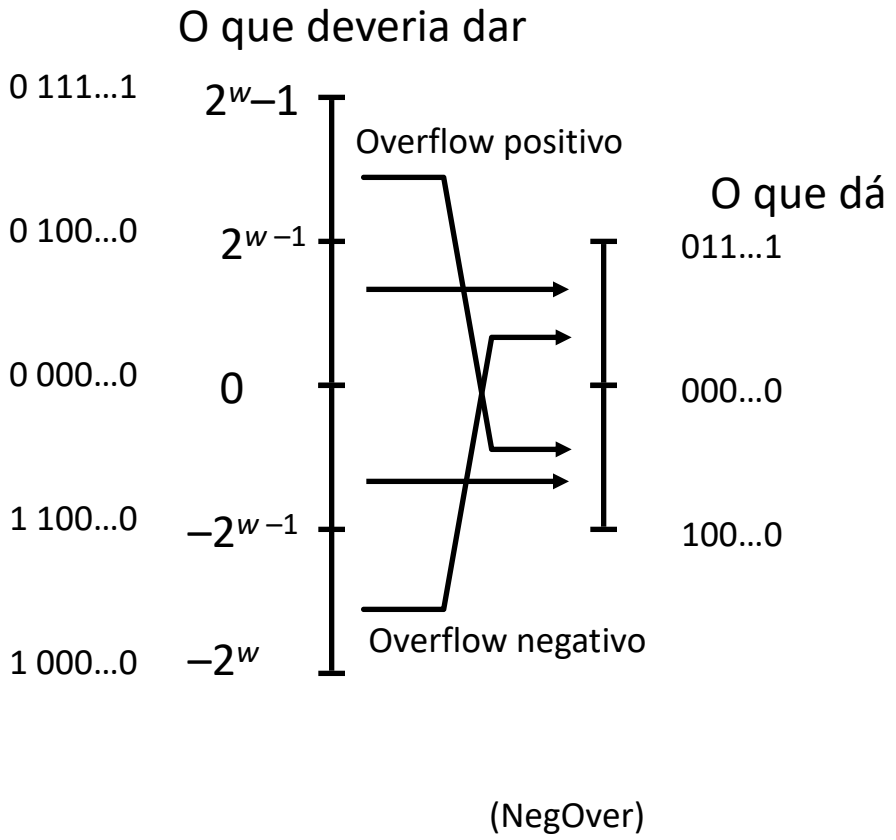
- Problema
  - Dados $s = \text{Soma}_w(u , v)$
  - Determinar se o resultado está correto
  - Exemplo

```
int s, u, v;
s = u + v;
```

- Solução
  - Overflow se e só se:

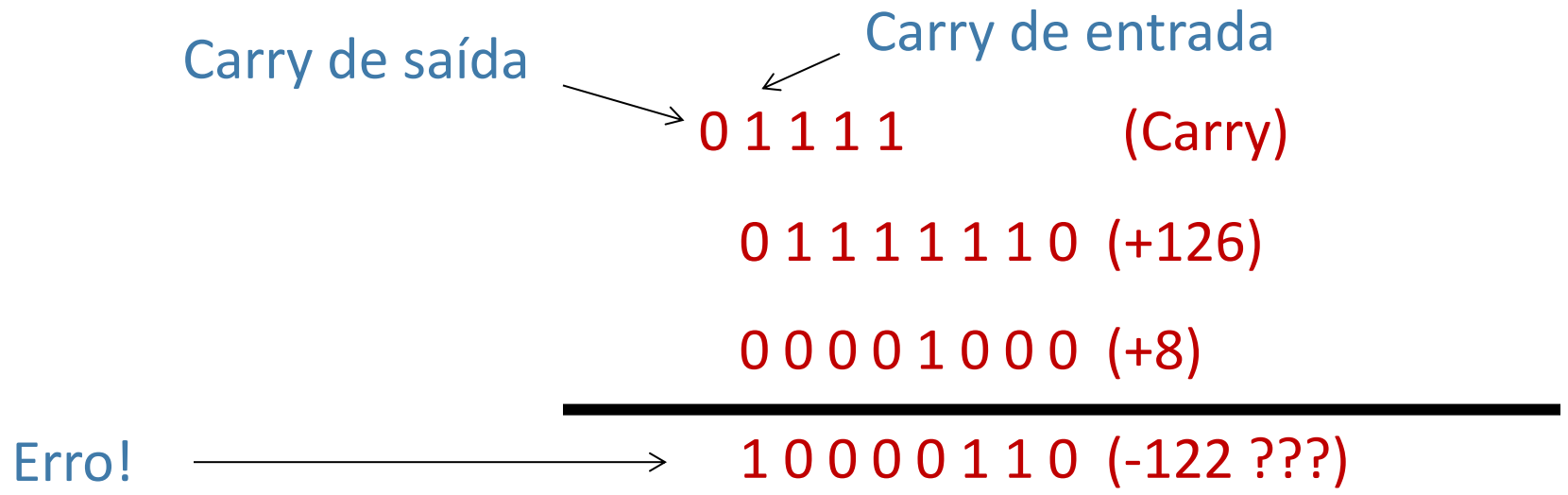
    $u, v < 0, s \geq 0$ (Overflow negativo)

    $u, v \geq 0, s < 0$ (Overflow positivo)

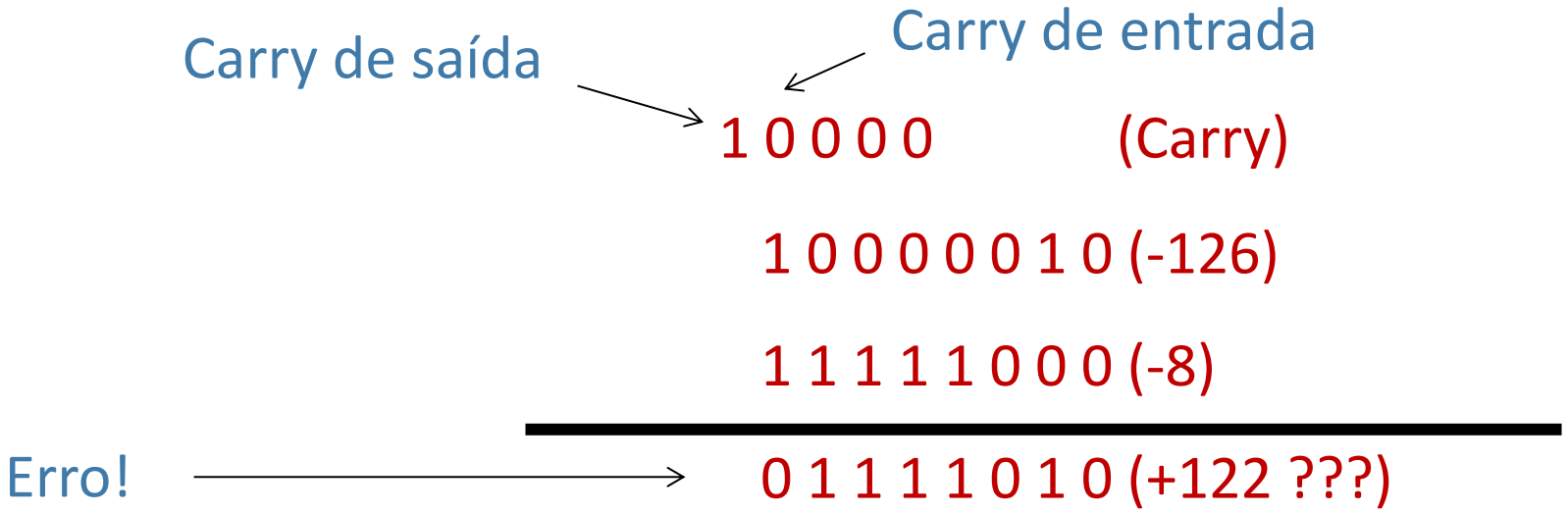


Overflow

- Ocorre quando se somam dois valores positivos e o resultado é negativo ou se somam dois valores negativos e o resultado é positivo
- É fácil detectar "overflow" – ele acontece se o transporte (carry) "de entrada" no bit mais significativo é **diferente** do de saída



Overflow



- Outra forma de detectar o overflow é verificar se o carry de saída e o bit mais significativo são diferentes
- Os CPUs têm uma flag a assinalar se houve *overflow* no último uso da ALU
  - ALU - unidade lógica e aritmética que executa as operações lógicas e aritméticas

Representação de números reais.
A norma IEEE Floating Point Standard.

Representação de Números Reais

- Os inteiros têm uma gama limitada de valores
  - não consegue representar números muito grandes e muito pequenos
- Os computadores modernos permitem representar números reais no formato de vírgula flutuante
  - $32\ 767 = 3.2767 \times 10^4$
- Hardware especializado para operar com este tipo de dados

Representação em vírgula flutuante (FP)

- Três partes
  - Sinal (um bit)
  - Expoente (em potências de 2)
  - Mantissa (parte fraccionária)

- Exemplo didáctico com 14 bits: 1 sinal, 5 expoente, 8 mantissa.
- Suponhamos o número $17.0 = 0.17*10^2$

  Em binário: $17_{10} = 10001_2*2^0 = 1000.1_2*2^1 = 100.01_2*2^2 = 10.001_2*2^3 = 1.0001_2*2^4 = 0.10001_2*2^5$

Sinal, expoente, mantissa

- $0.10001_2 * 2^5$

  0 0 0 1 0 1 1 0 0 0 1 0 0 0

  sinal expoente mantissa

- Consegue-se representar um número muito maior do que nos inteiros
- $65536 = 10\,000\,000\,000\,000\,000_2 = 0.1_2 * 2^{17}$

  0 1 0 0 0 1 1 0 0 0 0 0 0 0

  sinal expoente mantissa

Deslocamento do expoente

- Não há expoentes negativos. Como representar

  $0.25 = 1 * 2^{-2}$?

- Podíamos usar uma representação em complemento para 2; mas é mais eficiente usar um **expoente deslocado (biased)**
- A ideia é que o expoente seja sempre positivo.
  - Para tal, soma-se um deslocamento que faz com que o intervalo de valores representáveis comece em -1
- Valores do expoente todo a 0 ou todo a 1 (valor -1) são reservados para casos especiais (0 ou infinito)
- No nosso exemplo didático reservamos 5 bits para o expoente
  - Gama de valores com sinal representável: [-16, 15]
  - Usamos um deslocamento de 15 (expoente está em **excesso 15**) → [-1, 30]
  - Um número maior que 15 corresponde a um expoente positivo.

Deslocamento do expoente

- $17 = 0.10001_2 * 2^5$ com expoente não deslocado

  0 0 0 1 0 1 1 0 0 0 1 0 0 0

  sinal expoente mantissa

- $0.10001_2 * 2^5$ com expoente deslocado:
  - expoente = 5 + 15 = 20

  0 1 0 1 0 0 1 0 0 0 1 0 0 0

  sinal expoente mantissa

- Para representar $0.25 = 1.0_2 * 2^{-2}$
  - expoente = -2 + 15 = 13

  0 0 1 1 1 0 1 0 0 0 0 0 0 0

  sinal expoente mantissa

Normalização

- Não há uma representação única para um número
  - 0 10101 10001000
  - 0 10110 01000100
  - 0 10111 00100010
  - 0 11000 00010001

São todas equivalentes

- A convenção é de que o bit mais significativo da mantissa deve ser 1 (normalização)
- Como o bit é sempre 1 escusa de ser representado e ganha-se um bit na mantissa

Normalização

- **Exemplo**: 0.03125 em notação de vírgula flutuante normalizada:

$$0.03125_{10} = 2^{-5} = 1.0_2 * 2^{-5} = 0.00001_2 x 2^0 = 0.0001_2 x 2^{1} = \ldots = 0.\ 1_2 x 2^{-4}$$

- A versão normalizada é $1.0_2 * 2^{-5}$
  - Sinal: + → $0 = 0_2$ → 0 (representação com 1 bit)
  - Expoente: -5 → -5 + 15 = 10 = 1010_2 → 01010 (representação com 5 bits)
  - Mantissa: 1.0 → guardamos apenas a parte fracionária: 0_2
    → 00000000 (representação com 8 bits)
- O resultdo é:

0 0 1 0 1 0 0 0 0 0 0 0 0 0 0

Soma em vírgula flutuante

- **Soma**: é preciso colocar os números com o mesmo expoente, somar e normalizar

  0 10010 01001000 +

  0 10000 10011010

Depois de alinhar fica:

  1 0 1 . 0 0 1 0 0 0 +

  1 . 1 0 0 1 1 0 1 0

---

  1 1 0 . 1 0 1 1 1 0 1 0

- Normalizando e truncando os bits que naoi cabem na mantissa, obtemos:

  0 10010 10101110

Multiplicação em vírgula flutuante

- **Regras normais**: multiplicação das mantissas, soma dos expoentes e normalização

  $2^{-3} \times 2^{4} = 2^{1}$

- **Exemplo**:

  0 10010 10010000 $1.10010000 \times 2^{3} = 12.5$

  0 10000 10011010 $1.10011010 \times 2 = 3.203125$

- O produto das mantissas dá 10.100000001010 e a soma dos expoentes dá 4.
- Normalizando, temos $1.0100000001010 \times 2^{5}$
  - Sinal: + → $0 = 0_2$ → 0 (representação com 1 bit)
  - Expoente: 5 → 5 + 15 = 20 = 10100_2 → 10100 (representação com 5 bits)
  - Mantissa: 1.0100000001010 → 01000000 (representação com 8 bits)
- Resultado:

  0 1 0 1 0 0 0 1 0 0 0 0 0 0 0

  sinal **expoente** **mantissa**

Erros de representação de números reais

- O conjunto dos reais é “contínuo”: há um número infinito de reais e com um número limitado de bits de representação só podemos representar um número finito de valores.
- A representação é assim uma aproximação
  - Quanto mais bits tivermos melhor a aproximação.
- Não se consegue representar valores demasiado grandes ou demasiado pequenos
- Na nossa representação a 14 conseguimos representar os valores

  $-1.11111111_2 \times 2^{-15}$ a $1.11111111_2 \times 2^{15}$
- Não consegue representar 2^{-19} ou 2^{128} por exemplo

Erros de representação de números reais

- A representação de 128.5 é $1000\ 0000.1_2$
- Mas os 9 bits não cabem nos 8 bits da mantissa
  - o número representado é 128.0
- O erro relativo é (128.5 – 128)/128.5 = 0.4 %
- Os erros podem acumular-se
  - Os algoritmos devem tentar minimizá-los

Propagação de erros (iteração)

Multiplicador	Multiplicando	Produto a 14 bits	Produto real	Erro %
1000.001 16.125	0.11101000 0.90625	1110.1001 14.5625	14.7784	1.46
1110.1001 14.5625	0.11101000 0.90625	1101.0011 13.1885	13.4483	1.94
1101.0011 13.1885	0.11101000 0.90625	1011.1111 11.9375	12.2380	2.46
1011.1111 11.9375	0.11101000 0.90625	1010.1101 10.8125	11.1366	2.91
1010.1101 10.8125	0.11101000 0.90625	1001.1100 9.75	10.1343	3.79
1001.1100 9.75	0.11101000 0.90625	1000.1101 8.8125	8.3922	4.44

A norma IEEE-754 Floating Point Standard (1985)

- Precisão simples 32 bits
  - Sinal 1 bit
  - Expoente 8 bits (deslocamento 127)
  - Mantissa (significand) 23 bits
- Precisão dupla 64 bits
  - Sinal 1 bit
  - Expoente 11 bits (deslocamento 1023)
  - Mantissa (significand) 52 bits
- Adoptado quase universalmente

Precisão simples

- Expoente = 255 e mantissa (significando) = 0
  - + infinito
  - - infinito
- Expoente = 255 e mantissa (significando) dif. 0
  - NaN (Not a Number)
- Dois valores para 0
  - Sinal = 0; expoente = 0, mantissa = 0
  - Sinal = 1; expoente = 0, mantissa = 0
- Também para os 64 bits

Precisão dupla

- NaN e infinito indicados quando o expoente é 2047
- Valores representáveis: