

Representação de Dados Inteiros com sinal

Representação de Inteiros

- Com n bits, podemos ter 2^n valores distintos
- Considerando só inteiros não-negativos (*unsigned*) a faixa de valores é $[0, 2^n - 1]$
- Considerando inteiros quaisquer ($i < 0$, $i \geq 0$), também tem-se apenas 2^n possíveis valores
- Opção 1: (bit mais significativo como sinal).
 - representação por sinal e magnitude
- $n = 4 \rightarrow 15$ valores possíveis

0000 ... 0111	0 a 7 decimal
1001 ... 1111	-1 a -7 decimal
1000	zero negativo?

Complemento a 2

- Representação mais usual para inteiros com sinal
 - interpretar o bit mais significativo com **peso negativo**
 - intervalo de valores: $[-2^{n-1}, 2^{n-1}-1]$
 - com 8 bits: $[-2^7, 2^7-1] \rightarrow [-128, 127]$

$x_{n-1}, x_{n-2}, \dots, x_3, x_2, x_1, x_0$

Unsigned

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

Complemento a 2

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

- Exemplos:
 - $0101 = 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5$
 - $1011 = -1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = -8 + 2 + 1 = -5$

Faixa de Valores

- Unsigned

- $UMin = 0$

000...0

- $UMax = 2^w - 1$

111...1

- Complemento a 2

- $TMin = -2^{w-1}$

100...0

- $TMax = 2^{w-1} - 1$

011...1

intervalo é assimétrico!

$\frac{1}{2}$ para os negativos

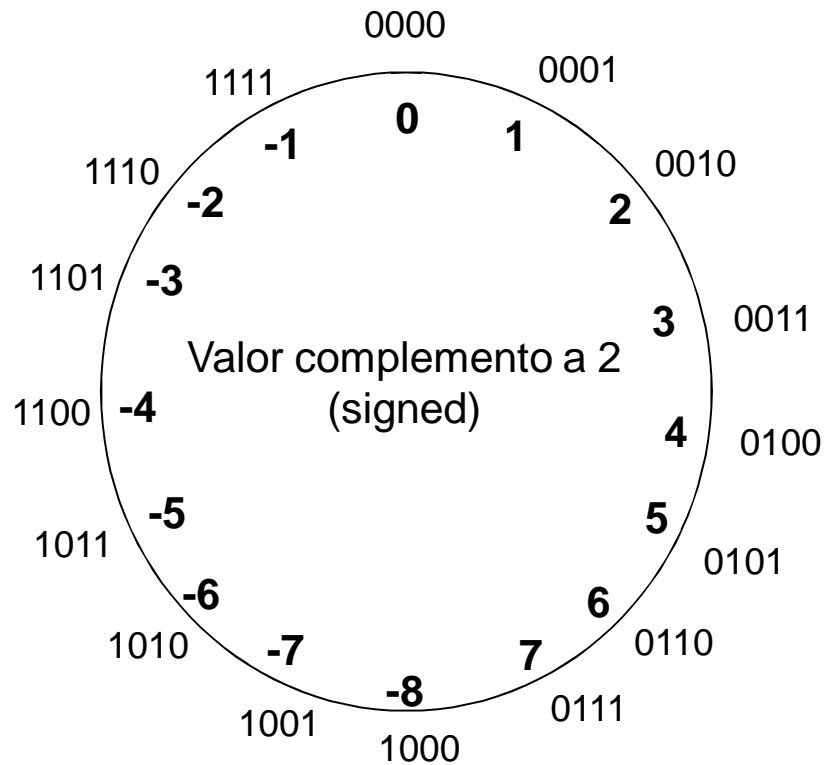
$\frac{1}{2}$ para positivos + 0

Exemplos 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

Modelo Circular

$n = 4$



binário	Compl-2	binário	Compl-2
0000	0	1000	-8
0001	1	1001	-7
0010	2	1010	-6
...
0111	7	1111	-1

$$-8 + 4 + 2 + 1 = -8 + 7$$

bit mais significativo tem "peso" -8!

Representação de números negativos complemento a 2

Idéia central:

$$(2^n + x) \bmod 2^n$$

Se $x \geq 0$ $\text{rep}_2(x) = x$

Se $x < 0$ então $\text{rep}_2(x) = 2^n + x$

Exemplos:

$$\text{rep}_2(-2) = 2^4 + (-2) = 14 = [1110]$$

$$\text{rep}_2(-8) = 2^4 + (-8) = 8 = [1000]$$

$$\text{rep}_2(-1) = 2^4 + (-1) = 15 = [1111]$$

$$[1111] = 15 = 2^4 + x \rightarrow x = 15 - 16 = -1$$

$$[1000] = 8 = 2^4 + x \rightarrow x = 8 - 16 = -8$$

binário	Compl-2	binário	Compl-2
0000	0	1111	-1
0001	1	1110	-2
0010	2	1101	-3
...	..	1001	-7
0111	7	1000	-8

Uma outra forma...

1. Encontrar a representação de $(-x)$
2. Inverter bit a bit
3. Somar 1

$$\begin{array}{r} -5 \rightarrow 5 \rightarrow 0000\ 0101 \rightarrow 1111\ 1010 \\ + 1 \\ 1111\ 1011 \end{array}$$

1111 1011
- 1
1111 1010 \rightarrow 0000 0101 \rightarrow 5 \rightarrow -5

Representação binária de um inteiro negativo

- Por que funciona (para $x < 0$)?

$$2^n + x = (2^n - 1) - (-x) + 1$$

Complemento bit-a-bit
de $(-x)$

$$\begin{array}{l} 1 - 1 = 0 \\ 1 - 0 = 1 \end{array}$$

$$\begin{array}{r} 1111 \ 1111 \\ - 0000 \ 0101 \\ \hline 1111 \ 1010 \end{array}$$

Faixa de Valores

- Maior e menor inteiros que podemos representar com *signed/unsigned*

	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

- C não requer que a representação seja em complemento a 2, mas a maioria das máquinas o faz
- Não é boa prática (para portabilidade) a faixa de valores
 - <limits.h> define constantes para os tipos de dados inteiros
 - INT_MAX, INT_MIN, UINT_MAX

Soma e Subtração

- Uma enorme vantagem da representação de inteiros em complemento a 2 é que todas as somas e subtrações empregam o **mesmo algoritmo de adição**
 - subtração = soma do complemento!
 - o algoritmo para achar o complemento é trivial
- A aritmética módulo 2^n garante que o resultado da soma é correto mesmo com sinais diferentes (a menos de overflow)
- Exemplo para $n=3$
$$1-2 = (1 \bmod 8) + (-2 \bmod 8) = -1 \bmod 8$$
$$[001] - [010] = [001] + [110] = [111]$$

Soma complemento a 2

(exemplos para 4 bits)

- $2 + 3 = 0010 + 0011 = 0101 = 5$
- $7 - 1 = 7 + (-1) = 0111 + 1111 = 0110 = 6$
- $(-3) + 6 = 1101 + 0110 = 0011 = 3$
- $(-1) + (-1) = 1111 + 1111 = 1110 = (-2)$

Signed e Unsigned em C

- Na conversão entre tipos *signed/unsigned* de mesmo tamanho, o padrão de bits não é afetado: apenas **a interpretação desse padrão muda**.
 - `int x;`
 - `unsigned u = (unsigned) x;`
 - `unsigned t = -1; /* ffffffff interpretado com valor sem sinal */`
- Constantes são valores “signed” por default

Exemplos de conversão

```
short int x = -12345; /* 1100 1111 1100 0111 */  
                /* 0xCFC7 como unsigned = 53191 */  
  
unsigned short ux = (unsigned short) x; /* 53191 */  
x = (short int) ux; /* -12345 */
```

- Em expressões com combinações signed/unsigned, é feita conversão do valor signed para unsigned
 - pode afetar operações relacionais!

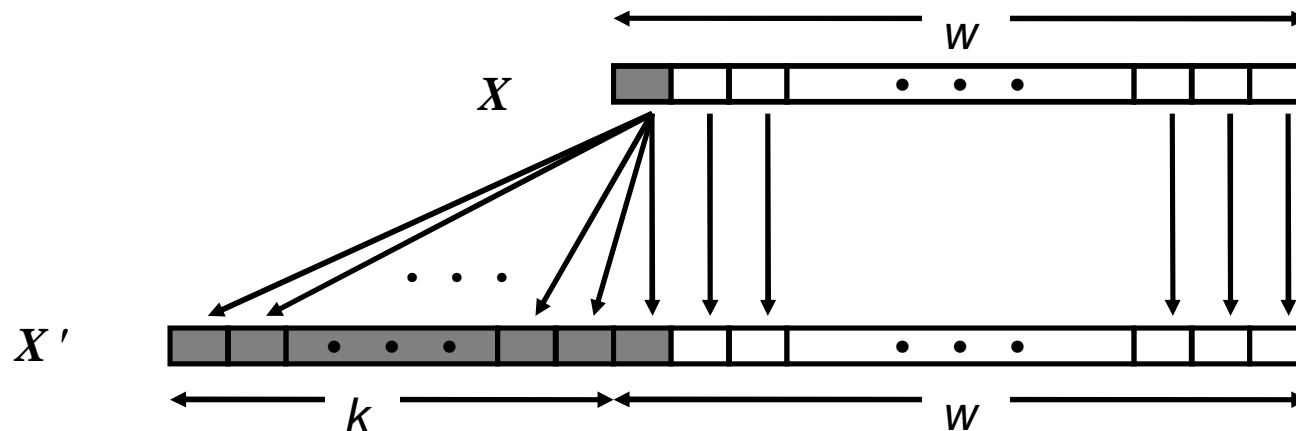
Operadores relacionais

- Operadores de comparação (<, ==, >, etc.) levam em conta se operandos são unsigned ou compl 2.
- Em assembler existem instruções específicas para cada caso
 - o compilador C gera o código com as instruções corretas, dependendo da declaração dos operandos
- Mas em expressões que envolvem os dois tipos, os valores são tratados como unsigned !
- Exemplo:

```
int a[2] = {-1, 0};  
if (a[0] < a[1]) → true  
unsigned int z=0;  
if (a[0] < z) → false !!!!!
```

Extensão de Representação

- Ocorre quando aumentamos o número de bits usados na representação
 - char para short/int, short para int
- Como converter um número em w bits para $w+k$ bits mantendo o mesmo valor?
 - unsigned : adicionar zeros à esquerda (zero extension)
 - signed: k cópias do bit de sinal (sign extension)



Extensão de Representação

- Exemplo: $w = 3$, $k = 2$ (extensão de 3 para 5 bits)

- $101 = -1 * 2^2 + 1 = -3$

- $11101 = -1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 1 = -16 + 8 + 4 + 1 = -3$

$$(-1 * 2^2 * 4) + (1 * 2^2 * 2) + (1 * 2^2) = -1 * 2^2$$

Conversão em C

```
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

Truncamento de Inteiros

- Ocorre quando reduzimos o número de bits usados na representação
 - int para short/char, short para char
- Quando truncamos um inteiro simplesmente removemos os bits mais significativos, o que pode alterar o valor do inteiro!
 - Quando x é unsigned, (short) x = $x \bmod 16$
 - Quando x é signed (Compl-2), os bits menos significativos são simplesmente interpretados como complemento a dois (o que pode alterar o sinal!)
- Exemplo (truncamento 8 bits para 4 bits):

$[00011001] = 25$
 $[1001] = -7$

Overflow em Compl.-2

- Quando o resultado $(x+y)$ não é representável em n bits

Operandos: w bits


u 

+ v 

Soma real: $w+1$ bits

$u + v$ 

Descarta bit $w+1$

$\text{TAdd}_w(u, v)$ 

- Para signed:
 - Se x, y tem sinais diferentes, nunca ocorre!
 - Se $x > 0$ e $y > 0$ overflow ocorre se houver carry do penúltimo para o último bit (resultado negativo!) e sem carry p/ fora
 - Se $x < 0$ e $y < 0$, overflow ocorre quando não há carry do penúltimo para o último, mas sempre haverá do último para fora)
 - Portanto, quando os dois carries são diferentes, o hardware marca a condição de overflow !

Exemplos

Para 4 bits (resultado real em 5 bits):

x	y	x + y	resultado
-8 [1000]	-5 [1011]	-13 [10011]	3 [0011]
-8 [1000]	-8 [1000]	-16 [10000]	0 [0000]
-8 [1000]	5 [0101]	-3 [11101]	-3 [1101]
2 [0010]	5 [0101]	7 [00111]	7 [0111]
5 [0101]	5 [0101]	10 [01010]	-6 [1010]

← Overflow negativo (carry para fora)

← Overflow negativo (carry para fora)

← Sem overflow

← Sem overflow

← Overflow positivo (carry do penúltimo para o último bit)

Obs: para unsigned, overflow é indicado por outra condição:
quando houve carry para fora