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 >=0), em n bits também teremos apenas 2^n possíveis valores
  - 0, valores negativos e valores positivos
- Como representar esses valores?
  - Há diferentes formas de representação

Sinal e Magnitude

- A idéia é usar o bit mais significativo como sinal
  - "1" → valor negativo

- Com 4 bits (1 + 3) → sinal + 8 valores possíveis (0..7)

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
- O bit mais significativo também distingue valores negativos e não negativos

$$X_{n-1}\ X_{n-2},\ \ldots\ldots\ldots\ldots\ldots X_3\ X_2\ X_1\ X_0$$

- representação baseada em *aritmética módulo* 2^n

motivação - complemento a 2

- Os mesmos n bits agora devem ser usados para representar "alguns números não negativos" e "alguns números positivos"
  - continuamos tendo 2^n padrões diferentes
  - usamos alguns desses padrões para representar negativos
    - quais padrões?
    - $x < 0$ representado por $y = x$ módulo 2^n

- representação baseada em *aritmética módulo* 2^n
  - $x = y \bmod k$ se $|x-y| = m*k$ para algum $m \varepsilon I$
  - exemplos:
    - 5 = --3 modulo 8
    - 5 = -11 modulo 8
    - 255 = -1 modulo 256

equivalência mod 2^n

- a relação de equivalência mod 2^n define uma partição dos inteiros em classes de equivalência
  - exemplo com n=3 (8 classes)

    {..., -16, -8, 0, 8, 16, ...}

    {..., -15, -7, 1, 9, 17, ...}

    {..., -14, -6, 2, 10, 18, ...}

    {..., -13, -5, 3, 11, 19, ...}

    {..., -12, -4, 4, 12, 20, ...}

    {..., -11, -3, 5, 13, 21, ...}

    {..., -10, -2, 6, 14, 22, ...}

    {..., -9, -1, 7, 15, 23, ...}
- em complemento a 2, cada padrão representa um número de sua classe de equivalência

equivalência mod 2^n

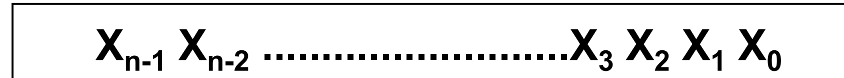
- a relação de equivalência mod 2^n define uma partição dos inteiros em classes de equivalência
  - exemplo com n=3

classe	padrão
{..., -16, -8, 0, 8, 16, ...}	000
{..., -15, -7, 1, 9, 17, ...}	001
{..., -14, -6, 2, 10, 18, ...}	010
{..., -13, -5, 3, 11, 19, ...}	011
{..., -12, -4, 4, 12, 20, ...}	100
{..., -11, -3, 5, 13, 21, ...}	101
{..., -10, -2, 6, 14, 22, ...}	110
{..., -9, -1, 7, 15, 23, ...}	111

- em complemento a 2, cada padrão representa um número de sua classe de equivalência

Intervalo de Valores

- Com **n** bits:



  - o menor valor representado é $\mathbf{-2^{n-1}}$
  - o maior valor representado é $\mathbf{2^{n-1} - 1}$ (n-1 bits setados)
  - uma única representação para 0 !

- Intervalo de valores: $[-2^{n-1},\ 2^{n-1}-1]$

– com 8 bits: $[-2^7,\ 2^7-1] \rightarrow [-128,\ 127]$

– com 16 bits: $[-2^{15},\ 2^{15}-1]$
  $\rightarrow [-32768,\ 32767]$

intervalo é assimétrico!
½ para os negativos
½ para positivos + 0

  - uma vantagem da escolha de "mais um negativo" é que o bit mais significativo indica o sinal do número!

Faixa de Valores com e sem sinal

- Unsigned
  - ***UMin*= 0**

    00…00
  - ***UMax*** = $2^n - 1$

    11…11
- Complemento a 2
  - ***TMin*** = -2^{n-1}

    100…0
  - ***TMax*** = $2^{n-1} - 1$

    011…1

Exemplos para n = 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

Faixa de Valores em C

- 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) assumir a faixa de valores
  – <limits.h> define constantes para os tipos de dados inteiros
  – INT_MAX, INT_MIN, UINT_MAX
  – SHRT_MIN, SHRT_MAX, USHRT_MAX
  – UCHAR_MAX, SCHAR_MIN, SCHAR_MAX
    – CHAR_MIN e CHAR_MAX: char tem sinal?

Representação de números negativos em complemento a 2

- **Idéia central: é uma representação mod** 2^n

- Se x>= 0 $rep_2(x) = x$
- Se x<0 então $rep_2(x) = 2^n + x$
  - (menor positivo na classe de equivalência de x)

  -1 mod 16 = (-1 + 16) mod 16

Exemplos para n = 4:

$rep_2(-2) = 2^4 + (-2) = 14 = [1110]$
$rep_2(-8) = 2^4 + (-8) = 8 = [1000]$
$rep_2(-1) = 2^4 + (-1) = 15 = [1111]$

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

Mecanismo trabalhoso se o número de bits é muito grande...

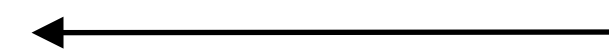
Uma outra forma...

Para encontrar a representação de (-x)

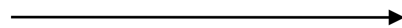
1. inverter ”x” bit a bit
2. somar 1

-5 → 5 → 0000 0101 → 1111 1010
+ 1
1111 1011

para achar a representação em comp2



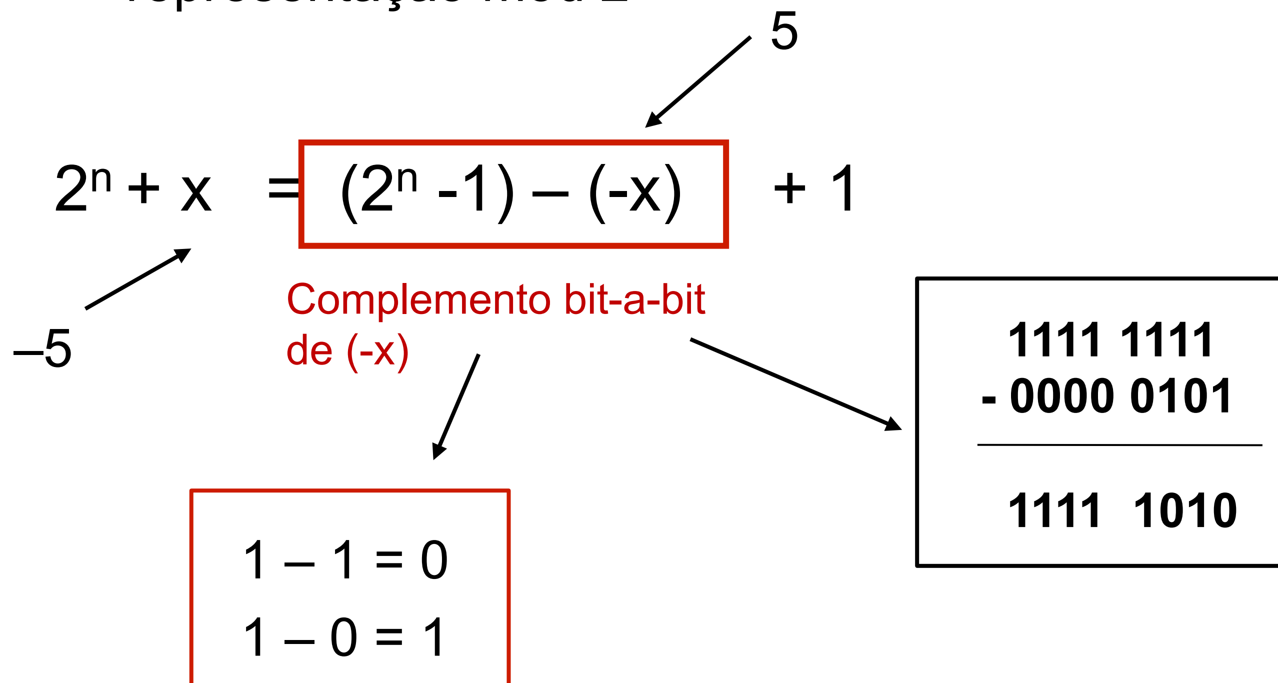
para achar o valor de uma representação



1111 1011 → 0000 0100
+ 1
0000 0101 → 5 → -5

Representação binária de um inteiro negativo

- Por que funciona (para $x < 0$)?
  - representação mod 2^n



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) (mod 8) = (1+(-2)) mod 8) =

  (1 mod 8) + (-2 mod 8) = 1 + 6 = 7 = rep[-1 mod 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 = -1; /* na memória: 1111......1111 */
  - unsigned u = (unsigned) x;   /* na memória: 1111......1111 */

```
short int x =  -12345; /* 1100 1111 1100 0111  */
                  /*como unsigned = 53191 */

unsigned short ux = (unsigned short) x; /* 53191 */
x = (short int) ux; /* -12345 */
```

- Compilador usa as instruções adequadas conforme o tipo da variável

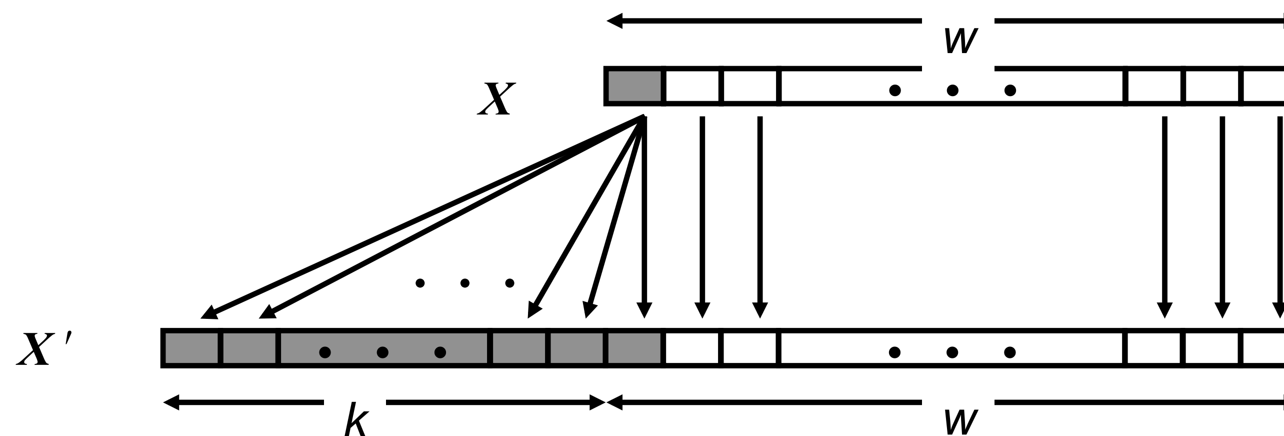
Operadores relacionais

- 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 do tipo dos operandos
- Mas em expressões que envolvem inteiros (**int**) signed e unsigned, 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
  - conversão que altera o tamanho: 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: extensão de 3 para 5 bits

- $rep_2(-3)$ em 3 bits = $2^3 + (-3)$ = 101
- $rep_2(-3)$ em 5 bits = $2^5 + (-3)$ = 11101
- por que basta completar com 1 à esquerda?

[$rep_2(-3)$ em 5 bits] - [$rep_2(-3)$ em 3 bits] =

$[2^5 + (-3)] - [2^3 + (-3)] = 2^5 - 2^3 =$

100000 - 1000 = 11000

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

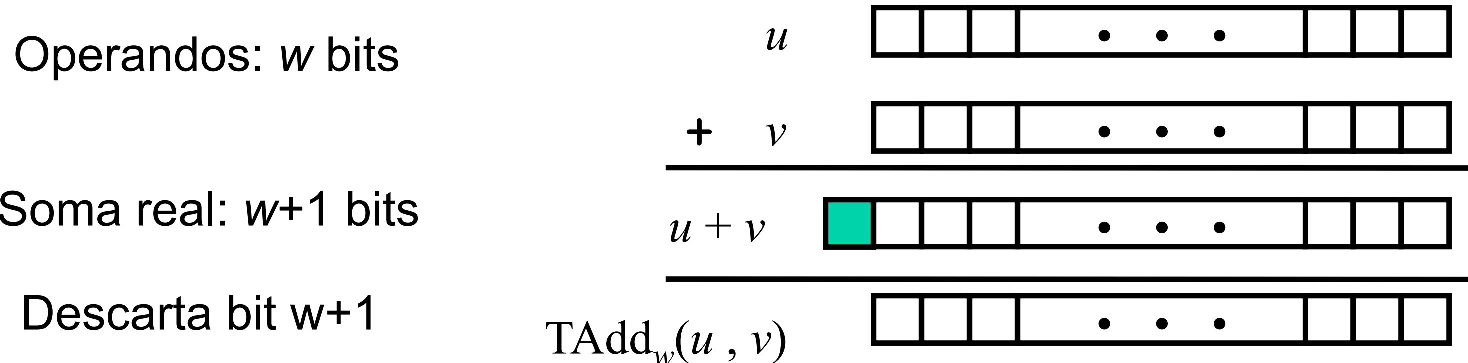
Truncamento de Inteiros

- 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 mod 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



- Para signed:
  - Se x,y tem sinais diferentes, nunca ocorre overflow!
  - Se x> 0 e y>0 o último bit dos 2 é 0, e ocorre overflow se houver carry do penúltimo para o último bit (resultado negativo!) e não há carry p/ fora
  - Se x<0 e y<0, overflow ocorre quando não há carry do penúltimo para o último (resultado positivo!), 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]
-1 [1111]	-5 [1011]	-6 [11010]	-6 [1010]
2 [0010]	5 [0101]	7 [00111]	7 [0111]
5 [0101]	5 [0101]	10 [01010]	-6 [1010]

← Overflow negativo

← Sem overflow (2 carries)

← 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