



Universidade de Brasília

Departamento de Ciência da Computação

Aula 8

Representação Numérica de Inteiros

“42

The Answer to the Ultimate Question of Life,
the Universe, and Everything”

Douglas Adams

http://en.wikipedia.org/wiki/The_Hitchhiker's_Guide_to_the_Galaxy

Sistemas Numéricos Posicionais

- Base decimal (base 10):

Símbolos: 0,1,2,3,4,5,6,7,8,9

□ Ex.: $124 = 1 \times 10^2 + 2 \times 10^1 + 4 \times 10^0 = 124_{10}$

- Base binária (base 2) :

Símbolos: 0,1

□ Ex.: $124 = 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 1111100_2$

- Base octal (base 8):

Símbolos: 0,1,2,3,4,5,6,7

□ Ex.: $124 = 1 \times 8^2 + 7 \times 8^1 + 4 \times 8^0 = 174_8$

- Base hexadecimal (base 16):

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

□ Ex.: $124 = 7 \times 16^1 + 12 \times 16^0 = 7C_{16}$



Generalizando

- Um número X_B de N dígitos na base B pode ser convertido em X_D na base D pela sua definição.

$$X_B = (d_{N-1}d_{N-2}d_{N-3} \dots d_2d_1d_0)$$

$$X_D = \left(\sum_{i=0}^{N-1} d_i \times B^i \right)_D$$

Esta definição e pode ser usada para realizar a conversão de um número em base qualquer para qualquer base, bastando que as operações aritméticas sejam feitas na base de destino.



Conversão Decimal, Binário, Hexadecimal

00	0000	0
01	0001	1
02	0010	2
03	0011	3
04	0100	4
05	0101	5
06	0110	6
07	0111	7
08	1000	8
09	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Ex.: $1010\ 1100\ 0101_2 = AC5_{16}$

$10\ 1111_2 = 2F_{16}$

$3F9_{16} = 11\ 1111\ 1001_2$

Obs.: $0xFF = FF_{16}$

Decimal: Bom para Humanos (pq?)

Binário: Bom para Computadores (pq?)

Hexa: Bom para quem?



Representação Numérica Computacional

- **Bits são apenas Bits!!** Sem nenhum significado inerente

- Ex.: o que é: 10100101 ????

Pode representar um número, caractere, instrução, cor, sinal de voz, música, temperatura, posição, taxa juros, \$,

O que podemos representar com N bits?

Apenas 2^N coisas!

Logo: Bom para coisas limitadas (contáveis)

Ex.: 26 Letras: 5 bits é suficiente

Caracteres ASCII: 7 bits (A,a,!) : ASCII estendido (8 bits)

Caracteres UNICODE 13.0 (2020): 143.859 caracteres

UTF-8 UTF-16 UTF-32 (*Unicode Transformation Format*)



Limitações:

- Se os bits representarem números:
Convenções definem a relação entre bits e números.
- Complicadores:
Números são infinitos!
Diferentes tipos: Naturais(\mathbb{N}), Inteiros(\mathbb{Z}), Reais(\mathbb{R}), Complexos(\mathbb{C})
Como representar os símbolos '-' e ',' ? Ex.: -2,5



Representação de Números Inteiros(\mathbb{Z})

Sinal e magnitude Complemento de um Complemento de dois

0000 = +0

0001 = +1

0010 = +2

0011 = +3

0100 = +4

0101 = +5

0110 = +6

0111 = +7

1000 = -0

1001 = -1

1010 = -2

1011 = -3

1100 = -4

1101 = -5

1110 = -6

1111 = -7

0000 = +0

0001 = +1

0010 = +2

0011 = +3

0100 = +4

0101 = +5

0110 = +6

0111 = +7

1000 = -7

1001 = -6

1010 = -5

1011 = -4

1100 = -3

1101 = -2

1110 = -1

1111 = -0

0000 = +0

0001 = +1

0010 = +2

0011 = +3

0100 = +4

0101 = +5

0110 = +6

0111 = +7

1000 = -8

1001 = -7

1010 = -6

1011 = -5

1100 = -4

1101 = -3

1110 = -2

1111 = -1

Qual a melhor representação?



Comparação

■ Sinal e Magnitude:

- ☺ Fácil de entender. Fácil de negar. Simetria na representação.
- ☹ Circuitos aritméticos complexos.
- ☹ Existe +0 e -0.

■ Complemento de 1:

- ☺ Fácil de negar. Simetria na representação
- ☹ Circuitos aritméticos complexos.
- ☹ Existe +0 e -0.

■ Complemento de 2:

- ☺ Circuitos aritméticos mais simples. 1 único Zero.
- ☹ Representação assimétrica (+3,-4) (maior faixa dinâmica! ☺)
- ☹ Negação um pouco mais complexa.

Representação Numérica: Inteiros (\mathbb{Z})

- Como representar números negativos sem usar o símbolo '-' ?
- Complemento de X_B da base B com N dígitos

$$X_B + (-X)_B = (B^N)_B$$

Ex.: $X_{10}=4$ $(-X)_{10}=?$

obs.: $10000 = 9999+1$

- Complemento de 10 com 4 dígitos

$$0004 + (-X) = (10^4)_{10} \quad -X = 10000 - 4$$

$$(-X)_{10} = 9996$$

- Complemento de 2 com 4 bits:

$$0100 + (-X) = (2^4)_2 \quad -X = 10000 - 0100$$

$$(-X)_2 = 1100$$

- Complemento de 3 com 4 dígitos

$$0011 + (-X) = (3^4)_3 \quad -X = 10000 - 0011$$

$$(-X)_3 = 2212$$

- Complemento de 8 com 4 dígitos

$$0004 + (-X) = (8^4)_8 \quad -X = 10000 - 4$$

$$(-X)_8 = 7774$$

- Complemento de 16 com 4 dígitos

$$0004 + (-X) = (16^4)_{16} \quad -X = 10000 - 4$$

$$(-X)_{16} = \text{FFFC}$$



Representação Numérica de Inteiros (\mathbb{Z})

- Binário sem sinal (\mathbb{N}) em N bits:
$$X = \sum_{i=0}^{N-1} b_i 2^i$$

- Binário complemento de 2 em N bits

□ **Interpretação:**
$$X = -b_{N-1} 2^{N-1} + \sum_{i=0}^{N-2} b_i 2^i$$

- **Negação:** truque = inverter e somar 1

Ex.: 5 = 0101

$$-5 = 1010 + 1 = 1011 = -2^3 + 2^1 + 2^0$$

$$X + \bar{X} = 111 \dots 111 = -1$$

$$-X = \bar{X} + 1$$

- **Extensão de Sinal** : repetir o MSB

Ex.: 5 = 0000 0101

-5 = 1111 1011



ISA RV32I

Números de 32 bits com sinal:

0000 0000 0000 0000 0000 0000 0000 0000	$_{bin}$	=	0_{dec}
0000 0000 0000 0000 0000 0000 0000 0001	$_{bin}$	=	1_{dec}
0000 0000 0000 0000 0000 0000 0000 0010	$_{bin}$	=	2_{dec}
...	...		
0111 1111 1111 1111 1111 1111 1111 1101	$_{bin}$	=	$2.147.483.645_{dec}$
0111 1111 1111 1111 1111 1111 1111 1110	$_{bin}$	=	$2.147.483.646_{dec}$
0111 1111 1111 1111 1111 1111 1111 1111	$_{bin}$	=	$2.147.483.647_{dec}$
1000 0000 0000 0000 0000 0000 0000 0000	$_{bin}$	=	$-2.147.483.648_{dec}$
1000 0000 0000 0000 0000 0000 0000 0001	$_{bin}$	=	$-2.147.483.647_{dec}$
1000 0000 0000 0000 0000 0000 0000 0010	$_{bin}$	=	$-2.147.483.646_{dec}$
...	...		
1111 1111 1111 1111 1111 1111 1111 1101	$_{bin}$	=	-3_{dec}
1111 1111 1111 1111 1111 1111 1111 1110	$_{bin}$	=	-2_{dec}
1111 1111 1111 1111 1111 1111 1111 1111	$_{bin}$	=	-1_{dec}

Operações em complemento de dois

■ Extensão de Sinal:

Converter números de n bits em números com mais de n bits:

Copiar o bit mais significativo para os outros bits

0010 → 0000 0010 = 2 (infinitos zeros)

1010 → 1111 1010 = -6 (infinitos uns)

Ex.: O campo imediato de 12 bits do RV32I é convertido em 32 bits para efetuar as operações aritméticas

```
addi t0, t0, -32      lw t0, 32(t0)
```

Ex.: Carregar um byte (8 bits) da memória para um registrador (32 bits)

```
lb t0, 32(t1)          lbu t0, 32(t1)
```

Ex.: Carregar uma half word (16 bits) da memória para um registrador (32 bits)

```
lh t0, 32(t1)          lhu t0, 32(t1)
```

Operações em complemento de dois

■ Comparação de números:

Suponha que:

s0 armazene o número

1111 1111 1111 1111 1111 1111 1111 1111₂

s1 armazene o número

0000 0000 0000 0000 0000 0000 0000 0001₂

Quais os valores de t0 e t1 dadas as instruções abaixo?

```
slt    t0, s0, s1    #comparação com sinal
sltu   t1, s0, s1    #comparação sem sinal
```

Logo: t0=1 e t1=0



Exemplo: Considere a verificação de um índice i que aponte para um elemento válido de um vetor $v[\text{dim}]$.

```
if ( i < 0 || i >= dim )
    goto indice_fora_limite;
```

```
# associando s0=i e s1=dim
blt s0,zero,indice_fora_limite
bge s0,s1,indice_fora_limite
```

```
bgeu s0,s1,indice_fora_do_limite
```

■ *Obs.: Tipos em C (processador de 64 bits)*

8 bits: unsigned char: 0 ... 255	e	char: -128 ... 127
16 bits: unsigned short: 0 ... 65535	e	short: -32768 ... 32767
32 bits: unsigned int: 0 ... $2^{32}-1$	e	int: $-2^{31} ... 2^{31}-1$
64 bits: unsigned long long int: 0 ... $2^{64}-1$	e	long long int: $-2^{63} ... 2^{63}-1$

Windows: unsigned long int: $0 ... 2^{32}-1$ e long int: $-2^{31} ... 2^{31}-1$

Linux: unsigned long int: $0 ... 2^{64}-1$ e long int: $-2^{63} ... 2^{63}-1$

Operações Aritméticas: Adição e subtração

- Exatamente como base decimal (emprestar/vai 1s) **descartando o transbordo**

Ex.:

$\begin{array}{r} 0011 \\ + 0010 \\ \hline 0101 \end{array}$	$\begin{array}{r} 0111 \\ + 1010 \\ \hline \text{✗}0001 \end{array}$	$\begin{array}{r} 1100 \\ + 1111 \\ \hline \text{✗}1011 \end{array}$	$\begin{array}{r} 0111 \\ - 0110 \\ \hline 0001 \end{array}$	$\begin{array}{r} 0110 \\ - 0101 \\ \hline 0001 \end{array}$	$\begin{array}{r} 0010 \\ - 0100 \\ \hline 1110 \end{array}$
--	--	--	--	--	--

- Facilidade de operações do complemento de dois
subtração pode ser feita usando adição de números negativos
- Overflow:** resultado muito grande para a word finita do computador
Somar dois números de n bits pode produzir um número de n+1 bits.

Ex.:

$\begin{array}{r} 0101 = +5 \\ + 0100 = +4 \\ \hline 1001 \neq 9 \end{array}$	$\begin{array}{r} 1001 = -7 \\ + 1010 = -6 \\ \hline \text{✗}0011 \neq -13 \end{array}$	$\begin{array}{r} 0110 = 6 \\ - 1000 = -8 \\ \hline 1110 \neq 14 \end{array}$
---	---	---

Note que o termo **overflow** não significa que um carry simplesmente “transbordou”
(n de bits do resultado > n bits das parcelas)

Mas sim que o resultado não é representável na faixa dinâmica de n bits!!!

Detectando overflow

- Nenhum overflow quando: - somar operandos com sinais diferentes
- subtrair operandos com sinais iguais
- **O overflow ocorre quando uma inconsistência matemática é gerada:**

Dado dois números positivos $A > 0$ e $B > 0$

- ✓ somar dois positivos produz um negativo: $A + B < 0$
- ✓ somar dois negativos produz um positivo: $(-A) + (-B) > 0$
- ✓ subtrair um negativo de um positivo e obtenha um negativo: $A - (-B) < 0$
- ✓ subtrair um positivo de um negativo e obtenha um positivo: $(-A) - B > 0$
- ✓ Forma prática: *Carry In* do último dígito diferente do *Carry Out*



Efeitos do overflow

- Na ISA RISC-V: Overflow não é detectado na aritmética inteira!
Motivação: Simplificação do hardware

- Porém: C vs FORTRAN

C: Não detecta overflow.

FORTRAN: Detecta overflow.

Logo, se for necessário detectar overflow, testes devem ser implementados no software.

Ex.: Para adição de números sem sinal

```
addu t0, t1, t2
```

```
bltu t0, t1, overflow
```

- No x86 e no ARM, overflow aciona uma *flag* indicativa.
- No MIPS, overflow causa uma exceção.