Fundamentos de Arquitetura de Computadores

Tiago Alves

Faculdade UnB Gama
Universidade de Brasília

Circuitos digitais lidam com *bits*. O significado desses bits é dado pelo circuito, projeto ou aplicação, mas eles **sempre** lida com bits.

Já vimos que, ao manipularmos valores de grandezas na física e matemática, usamos um sistema numérico posicional:

$$\begin{aligned} 1734_{10} &= 1 \cdot 1000 + 7 \cdot 100 + 3 \cdot 10 + 4 \cdot 1 \\ &= 1 \cdot 10^3 + 7 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0 \end{aligned}$$

De forma geral, o valor atribuído a um número $d_1 d_0{,}d_{-1} d_{-2}$ é:

$$v = d_1 \cdot 10^1 + d_0 \cdot 10^0 + d_{-1} \cdot 10^{-1} + d_{-2} \cdot 10^{-2}$$

Ou, de forma compacta:

$$v = \sum_{i=-n}^{p-1} d_i \cdot \mathbf{r}^i$$

Já vimos que, para números binários:

$$v = \sum_{i=-n}^{p-1} b_i \cdot \mathbf{2}^i$$

E:

$$\begin{aligned}
\mathbf{10011}_2 = &\quad \mathbf{1}\cdot 16 + \mathbf{0}\cdot 8 + \mathbf{0}\cdot 4 + \mathbf{1}\cdot 2 + \mathbf{1}\cdot 1 && = 19_{10} \\
\mathbf{100010}_2 = &\quad \mathbf{1}\cdot 32 + \mathbf{0}\cdot 16 + \mathbf{0}\cdot 8 + \mathbf{0}\cdot 4 + \mathbf{1}\cdot 2 + \mathbf{0}\cdot 1 && = 34_{10} \\
\mathbf{101,001}_2 = &\quad \mathbf{1}\cdot 4 + \mathbf{0}\cdot 2 + \mathbf{1}\cdot 1 + \mathbf{0}\cdot \tfrac{1}{2} + \mathbf{0}\cdot \tfrac{1}{4} + \mathbf{1}\cdot \tfrac{1}{8} && = 5,125_{10}
\end{aligned}$$

Podemos usar qualquer base b (também referenciada como *radix*) para nossa representação númerica, desde que $b \in \{x \in \mathbb{Z} \mid x > 1\}$. Por exemplo, podemos usar base 3, 7, 42, 64, etc..

Os antigos babilônios utilizavam base 60 e, por isso, temos 60 minutos em uma hora e 360 graus em um círculo.

De todas as bases, além das bases 2 e 10, duas delas são mais importantes: as bases 8 (octal) e 16 (hexadecimal).

Sistema Octal e Hexadecimal

No sistema octal os algarismos (símbolos) possíveis são: [0] ou $\mathbf{0}$, [1] ou $\mathbf{1}$, [2] ou $\mathbf{2}$, [3] ou $\mathbf{3}$, [4] ou $\mathbf{4}$, [5] ou $\mathbf{5}$, [6] ou $\mathbf{6}$ e [7] ou $\mathbf{7}$, e o valor atribuído ao número é:

$$v = \sum_{i=-n}^{p-1} o_i \cdot \mathbf{8}^i$$

No sistema hexadecimal os algarismos possíveis são: [0] ou $\mathbf{0}$, [1] ou $\mathbf{1}$, [2] ou $\mathbf{2}$, [3] ou $\mathbf{3}$, [4] ou $\mathbf{4}$, [5] ou $\mathbf{5}$, [6] ou $\mathbf{6}$, [7] ou $\mathbf{7}$, [8] ou $\mathbf{8}$, [9] ou $\mathbf{9}$, [10] ou $\mathbf{A}$, [11] ou $\mathbf{B}$, [12] ou $\mathbf{C}$, [13] ou $\mathbf{D}$, [14] ou $\mathbf{E}$ e [15] ou $\mathbf{F}$, e o valor atribuído ao número é:

$$v = \sum_{i=-n}^{p-1} h_i \cdot \mathbf{16}^i$$

UnB Gama

Por que usar os sistemas octal ou hexadecimal?

Porque a base é uma potência de 2! Logo, é fácil converter de octal / hexadecimal para binário e vice-versa.

Ex: Agrupamos 3 bits para formar um número octal:

$$\mathbf{100011001110}_2 = \quad \mathbf{100\ 011\ 001\ 110}_2 \quad = 4316_8$$

$$3715_8 = \quad \mathbf{011\ 111\ 001\ 101}_2 \quad = \mathbf{011111001101}_2$$

Ex: Agrupamos 4 bits para formar um número hexadecimal:

$$\mathbf{100011001110}_2 = \quad \mathbf{1000\ 1100\ 1110}_2 \quad = 8CE_{16}$$

$$DBA9_{16} = \quad \mathbf{1101\ 1011\ 1010\ 1001}_2 \quad = \mathbf{1101101110101001}_2$$

Decimal	Binário	Octal	Hexadecimal
0	**0**	0	0
1	**1**	1	1
2	**10**	2	2
3	**11**	3	3
4	**100**	4	4
5	**101**	5	5
6	**110**	6	6
7	**111**	7	7
8	**1000**	10	8
9	**1001**	11	9
10	**1010**	12	A
11	**1011**	13	B
12	**1100**	14	C
13	**1101**	15	D
14	**1110**	16	E
15	**1111**	17	F

O sistema octal não é mais tão utilizado, mas o hexadecimal é muito utilizado costumamos agrupar dados em bytes (8 bits) (Por que agrupamos em bytes?).

Um dígito hexadecimal pode ser referido como *nibble*.

Um byte precisa de 2 dígitos hexadecimais (ou 2 nibbles). Logo, n bytes precisam de $2n$ nibbles.

Adição de Números Binários

A adição de números binários funciona da mesma maneira que para números decimais, mas temos apenas duas alternativas para o *vai-um* (ou *carry*): **0** ou **1**.

$$\begin{array}{r} X \\ Y \\ \hline X+Y \end{array} \qquad \begin{array}{r} 190 \\ +141 \\ \hline 331 \end{array} \qquad \begin{array}{rr} \textit{carry} & 101111000 \\ & 10111110 \\ + & 10001101 \\ \hline & 101001011 \end{array}$$

A subtração de números binários também funciona da mesma maneira que para números decimais, utilizando o *borrow* (ou *empresta*).

X	229
Y	$-$ 46
$X - Y$	183

borrow	001111100
	11100101
$-$	00101110
	10110111

Operações Aritméticas

Operações de interesse ao nosso estudo:

- Somar
- Subtrair
- Multiplicar
- Dividir
- Comparar

Sistemas numéricos e suas conveniências:

- Decimal: Bom para Humanos (pq?)
- Binário: Bom para Computadores
- Hexa: Bom para quem?

Exemplos básicos:

- Decimal: $13 + 7 = 20$
- Binário: $1101 + 111 = 10100$
- Hexa: $1D + 7 = 24$

$$A + 3 + 10 = ??$$

Operações Aritméticas

Se os bits representarem número:

- Convenções definem a relação entre bits e números.

Complicadores:

- Números são infinitos!
- Tipos: Naturais, Inteiros, Reais, Racionais, Complexos...
- Qual N (cardinalidade) necessário?????

Q: Como representamos os Números Inteiros (com sinal)?

Representação de Números Negativos

Existem várias formas de representar números negativos, e elas podem ser utilizadas para qualquer base. Algumas delas são:

- Signed Magnitude Representation
- Complement Number System
- Excess Representation

Vamos focar no caso binário de cada uma delas.

UnB Gama

Signed Magnitude Representation

- Essa é a forma mais simples, onde a representação consiste de uma magnitude e de um sinal, indicando se o número é positivo ou negativo.
- Em geral, usamos o MSB para o sinal, onde $\mathbf{0}$ representa o sinal positivo $+$ e $\mathbf{1}$ representa o sinal negativo $-$.

Ex:

$$\mathbf{01010101}_2 = +85_{10} \qquad \mathbf{11010101}_2 = -85_{10}$$

$$\mathbf{01111111}_2 = +127_{10} \qquad \mathbf{11111111}_2 = -127_{10}$$

Signed Magnitude Representation

- Representa uma quantidade igual de números positivos e negativos.
- Com n bits, o intervalo é $-\left(2^{n-1}-1\right)$ a $\left(2^{n-1}-1\right)$.
- Tem duas representações para o zero: $+0$ e -0!

$$v = -1^{s} \cdot \left(\sum_{i=0}^{n-1} b_i \cdot \mathbf{2}^{i} \right)$$

Pró: Arranjo intuitivo dos dígitos.

Contra: Como fazer a adição? Lembre-se: temos dois zeros!

UnB Gama

No sistema de Representação de Magnitude com Sinal, negativamos um número invertendo o bit de sinal. No sistema de Complemento, negativamos o número pegando seu complemento (a ser definido pelo sistema).

Este sistema pode ser utilizado para qualquer base, com o nome de *radix complement* (complemento de base) e *diminished radix complement* (complemento de base reduzido). Para o caso binário, estes são, respectivamente, o complemento de 2 e o complemento de 1.

Complemento de 1

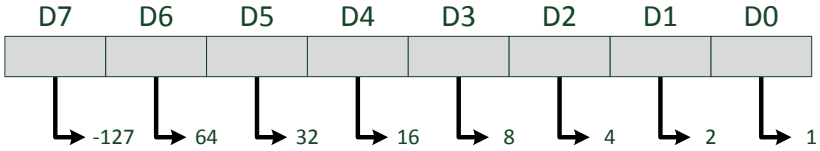
Neste sistema, o complemento de um número de n bits é obtido subtraindo esse número de $r^{n-1} - 1$. Isto pode ser obtido facilmente complementando os dígitos individualmente.

Neste sistema, o peso do primeiro bit é *negativo*:

$$v = -b_{n-1} \cdot \left(2^{n-1} - 1\right) + \sum_{i=0}^{n-2} b_i \cdot \mathbf{2}^i$$

- Note que temos (novamente!) duas representações para o zero: $+0$ $(00...00)$ e -0 $(11...11)$.
- Um número negativo sempre tem MSB igual a 1, mas um número com MSB igual a 1 pode não ser negativo!
- Com n bits, representamos números de $-2^{n-1} + 1$ até $2^{n-1} - 1$ (com $n = 8$, representamos números de -127 a 127).

Para $n = 8$ bits, separa-se um bit para sinal e codifica-se o restante do número nos outros 7 bits:



$$17_{10} = \mathbf{00010001}$$

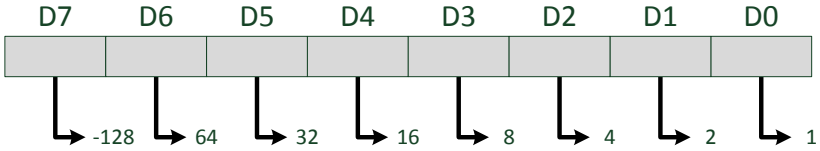
$$-17_{10} = \mathbf{11101110}$$

Neste sistema, o complemento de um número de n bits é obtido subtraindo esse número r^{n-1}. Isto pode ser obtido facilmente complementando os dígitos individualmente (complemento de 1) *e depois adicionando* 1.

Neste sistema, o peso do primeiro bit é *negativo*:

$$v = -b_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot \mathbf{2}^i$$

Para $n = 8$ bits, separa-se um bit para sinal e codifica-se o restante do número nos outros 7 bits:



$$17_{10} = \mathbf{00010001}$$

$$-17_{10} = \mathbf{11101111}$$

- **Vantagem:** só existe uma representação para o zero! $(00..00)$.
- Se o MSB é 1, o número é negativo e vice-versa.
- Com n bits, representamos números de -2^{n-1} até $2^{n-1}-1$ (com $n=8$, representamos números de -128 a 127).
- É o sistema mais usado: a soma e a subtração são fáceis!

Excess Representations

No sistema de representação por excesso-B, um número de m bits com valor absoluto (*unsigned*) é M (com $0 \leq M < 2^m$) e representa o inteiro com sinal com valor $M - B$, onde B é chamado o *bias* do sistema.

Este sistema é muito utilizado em notação de ponto flutuante.

Representação de números de 4 bits

Decimal	Complemento de 2	Complemento de 1	Signed Representation	Excess $B = 8$
-8	**1000**	-	-	**0000**
-7	**1001**	**1000**	**1111**	**0001**
-6	**1010**	**1001**	**1110**	**0010**
-5	**1011**	**1010**	**1101**	**0011**
-4	**1100**	**1011**	**1100**	**0100**
-3	**1101**	**1100**	**1011**	**0101**
-2	**1110**	**1101**	**1010**	**0110**
-1	**1111**	**1110**	**1001**	**0111**
0	**0000**	**1111** ou **0000**	**1000** ou **0000**	**1000**
1	**0001**	**0001**	**0001**	**1001**
2	**0010**	**0010**	**0010**	**1010**
3	**0011**	**0011**	**0011**	**1011**
4	**0100**	**0100**	**0100**	**1100**
5	**0101**	**0101**	**0101**	**1101**
6	**0110**	**0110**	**0110**	**1110**
7	**0111**	**0111**	**0111**	**1111**

UnB Gama

Padding de números em complemento de 2

Em números sem sinal, para converter de n para m bits (onde $m > n$), basta adicionar $m - n$ dígitos **0**s ao número (caso $m < n$, basta truncar o número - mas isso leva a erros de arredondamento caso algum dos bits descartados seja diferente de 0).

Para números em complemento de 2, se $m > n$ devemos adicionar $m - n$ cópias do bit de sinal à esquerda do número (isto é, nós preenchemos um número positivo com **0**s e um número negativo com **1**s até que a quantidade de dígitos seja contemplada). Esta operação é chamada de *sign extension*.

Se $m < n$, descartamos $n - m$ bits à esquerda (isto é, truncamos o número para m bits). Porém, o resultado só é válido se todos os bits descartados forem iguais ao bit do sinal (isto é, 0 se o número for positivo e 1 se for negativo).

Adição e Subtração em Complemento de 2

- Em complemento de 2, o próximo número é sempre 1 a mais do que o número anterior!
- Como a adição é uma extensão da contagem, podemos somar números em complemento de 2 simplesmente somando em binário e ignorando o *carry* no último bit.
- O resultado sempre será correto desde que o intervalo não seja excedido!

$$\begin{array}{rr} & +3 \\ + & +4 \\ \hline & +7 \end{array} \qquad \begin{array}{rr} & 0011 \\ + & 0100 \\ \hline & 0111 \end{array}$$

$$\begin{array}{rr} & -2 \\ + & -6 \\ \hline & -8 \end{array} \qquad \begin{array}{rr} & 1110 \\ + & 1010 \\ \hline 1 & 1000 \end{array}$$

$$\begin{array}{rr} & +6 \\ + & -3 \\ \hline & +3 \end{array} \qquad \begin{array}{rr} & 0110 \\ + & 1101 \\ \hline 1 & 0011 \end{array}$$

Recapitulando: Representação de números de 4 bits

Decimal	Complemento de 2	Complemento de 1	Signed Representation	Excess $B = 8$
-8	**1000**	-	-	**0000**
-7	**1001**	**1000**	**1111**	**0001**
-6	**1010**	**1001**	**1110**	**0010**
-5	**1011**	**1010**	**1101**	**0011**
-4	**1100**	**1011**	**1100**	**0100**
-3	**1101**	**1100**	**1011**	**0101**
-2	**1110**	**1101**	**1010**	**0110**
-1	**1111**	**1110**	**1001**	**0111**
0	**0000**	**1111** ou **0000**	**1000** ou **0000**	**1000**
1	**0001**	**0001**	**0001**	**1001**
2	**0010**	**0010**	**0010**	**1010**
3	**0011**	**0011**	**0011**	**1011**
4	**0100**	**0100**	**0100**	**1100**
5	**0101**	**0101**	**0101**	**1101**
6	**0110**	**0110**	**0110**	**1110**
7	**0111**	**0111**	**0111**	**1111**

UnB Gama

- Para fazer $A - B$ podemos fazer $A + (-B)$!
- Ou seja, basta tirar o complemento de 2 do subtraendo e somar normalmente.

$+4$	0100	0100
$-\ +3$	$-\ 0011$	$+\ 1101$
$+1$		$1\ 0001$

$+3$	0011	0011
$-\ -4$	$-\ 1100$	$+\ 0100$
$+7$		0111

Overflow

Um resultado que excede o intervalo do sistema causa um *overflow*.

A adição de dois números com sinal diferente nunca causa *overflow* (por que?) mas a adição de números de mesmo sinal pode causar *overflow*.

Na adição, ocorre *overflow* se o sinal dos adendos é igual mas o sinal do resultado é diferente! Outra forma de detectar esse *overflow* é se o *carry* entrando no MSB e o *carry* saindo do MSB forem diferentes.

Na subtração, observamos o sinal do minuendo e do subtraendo *complementado* (isto é, observamos se houve *overflow* na adição!). Também podemos observar o *carry* entrando e saindo do MSB (se forem diferentes, houve *overflow*).

Note que um *carry* ou *borrow* no bit mais significativo não necessariamente indica o *overflow* em complemento de 2!

UnB Gama

Complemento de 2 e Números Sem Sinal

Como a adição e subtração de números em complemento de 2 e números sem sinal funcionam da mesma forma, muitas vezes o mesmo circuito é utilizado para as duas operações.

Porém, os resultados devem ser interpretados de forma diferente caso o sistema esteja usando números em complemento de 2 (de -8 a 7, por exemplo) e números sem sinal (de 0 a 15).

Em especial, o *overflow* que ocorre entre -8 (1000) e 7 (0111) em complemento de 2 ocorre entre 0 (0000) e 15 (1111) em números sem sinal.

Para detectar o *overflow* na adição ou subtração de um número sem sinal, basta verificar se ouve *carry* ou *borrow* no MSB.

UnB Gama

Multiplicação em binário

A multiplicação em binário se dá da mesma forma que a multiplicação em decimal: adicionando uma lista de multiplicandos deslocados computados de acordo com os dígitos do multiplicador.

Na multiplicação em binário esse processo é ainda mais fácil, pois ou o bit do multiplicador é 0 (e o multiplicando deslocado é 0) ou é 1 (e o multiplicando deslocado é igual ao multiplicando).

$$\begin{array}{r} 11 \\ \times\ 13 \\ \hline 33 \\ 11 \\ \hline 143 \end{array}$$

$$\begin{array}{rl} 1011 & \text{multiplicando} \\ \times\ 1101 & \text{multiplicador} \\ \hline 1011 & \\ 0000 & \text{multiplicando} \\ 1011 & \text{deslocados} \\ 1011 & \\ \hline 10001111 & \text{produto} \end{array}$$

Multiplicação em binário

No entanto, em vez de listar todos os multiplicandos deslocados e então realizar a adição, em um sistema digital é mais conveniente adicionar cada multiplicando em um produto parcial.

$$\begin{array}{rr} & 11 \\ \times & 13 \\ \hline & 33 \\ & 11 \\ \hline & 143 \end{array}$$

$$\begin{array}{rr} & 1011 \\ \times & 1101 \\ \hline & 0000 \\ & 1011 \\ \hline & 01011 \\ & 0000\downarrow \\ \hline & 001011 \\ & 1011\downarrow\downarrow \\ \hline & 0110111 \\ & 1011\downarrow\downarrow\downarrow \\ \hline & 10001111 \end{array}$$

multiplicando
multiplicador
produto parcial
multiplicando deslocado
produto parcial
multiplicando deslocado
produto parcial
multiplicando deslocado
produto parcial
multiplicando deslocado
produto

UnB Gama

Multiplicação em binário

Em geral, a multiplicação de um número de n bits por um número de m bits requer $n + m$ bits (para que não haja *overflow*).

O algoritmo de deslocar e adicionar (*shift and add*) requer m produtos parciais e adições. Um circuito para realizar essa multiplicação pode ser realizado com um registrador de deslocamento, um somador e uma lógica de controle.

A multiplicação de números em complemento de 2 requer um pouco mais de cuidado.

Divisão em binário

A divisão em binário é baseado no algoritmo de deslocar e subtrair.

217	11
11 ↓	1
107	
99	19
8	

11011001	1011	dividendo e divisor
1011	1	divisor deslocado e quociente
00101		dividendo reduzido
0000	10	
1010		
0000	100	
10100		
1011	1001	
10011		
1000	10011	resto e quociente

A notação mais usada para representar números fracionários é a notação em ponto flutuante padrão IEEE 754.

$$v = -1^s \cdot m \cdot b^e$$

Onde:

- s representa o sinal (0 para positivo, 1 para negativo)
- m representa a mantissa (*significand*)
- b representa a base. No padrão IEEE 754 a base pode ser 10 ou 2, mas em sistemas computacionais utiliza-se sempre a base 2 (por que?)
- e representa o expoente

Existem dois tipos principais de números:

	Single Precision (float)	*Double Precision* (double)
Sinal	1 bit	1 bit
Expoente	8 bits	11 bits
Mantissa	23 bits	52 bits
Total	32 bits	64 bits

De acordo com o valor do Expoente, os números são classificados em:

- Normalizados
- Não Normalizados
- Especiais

Um número é considerado normalizado se $E \neq 00..00$ **e** $E \neq 11..11$.

Para codificar o expoente, usa-se um *Excess* code com BIAS $= 2^{m-1} - 1$ (onde m é o número de bits do expoente).

	BIAS	Intervalo
Single Precision	127	-126 a 127
Double Precision	1023	-1022 a 1023

Para codificar a mantissa, assume-se que $M = 1.XXXXXXXX_2$

- O 1 é assumido, logo apenas os valores após o ponto decimal são codificados.
- Logo, se todos os bits da mantissa forem zero, $M = 1.0$.

Exemplo: 15213_{10}.

$$\begin{aligned} 15213_{10} &= 11101101101101_2 \\ &= 1.1101101101101_2 \cdot 2^{13} \end{aligned}$$

	Valor	Bits
Sinal	Positivo	0
Expoente	$13+\mathbf{127}$	10001100
Mantissa	$\mathbf{1}.1101101101101_2$	11011011011010000000000

Codificação de Números Não Normalizados

Um número é considerado não normalizado se $E = 00..00$ (logo, o expoente é zero).

O expoente é considerado como $-BIAS + 1$ (ou seja, -2^{m-1}).

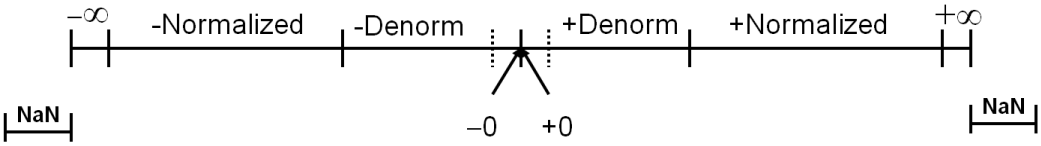
Para codificar a mantissa, assume-se que $M = 0.XXXXXXXX_2$

- O 0 é assumido, logo apenas os valores após o ponto decimal são codificados.
- Logo, se todos os bits da mantissa forem zero, $M = 0.0$.
- É utilizado para representar número próximos de zero.
- A precisão vai diminuindo a medida que os números ficam menores (*gradual underflow*).

Um número é considerado especial se $E = 11..11$.

Se $M = 000..000$, representa o infinito positivo e negativo. Em geral, ocorre por uma divisão por zero ou porque houve um *overflow*.

Se $M \neq 000.000$, indica um *Not A Number*, isto é, um valor que não tem representação numérica. O padrão define códigos especiais para indicar o porque isso não é um número.


−∞
-Normalized
-Denorm
+Denorm
+Normalized
+∞
NaN
−0
+0
NaN

MIPS e representação numérica

Números de 32 bits com sinal: Complemento de 2!

$$
\begin{array}{ll}
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} & = 0_{ten} \\
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} & = 1_{ten} \\
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} & = 2_{ten} \\
\ldots & \ldots \\
0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} & = 2{,}147{,}483{,}645_{ten} \\
0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} & = 2{,}147{,}483{,}646_{ten} \\
0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} & = 2{,}147{,}483{,}647_{ten} \\
1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} & = -2{,}147{,}483{,}648_{ten} \\
1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} & = -2{,}147{,}483{,}647_{ten} \\
1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} & = -2{,}147{,}483{,}646_{ten} \\
\ldots & \ldots \\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} & = -3_{ten} \\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} & = -2_{ten} \\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} & = -1_{ten}
\end{array}
$$

Operações em Complemento de 2

Processo básico: Negar (calcular o oposto de) um número em complemento de dois:

- Inverta todos os bits e some 1

Extensão de Sinal:

- Converter números de n bits em números com mais de n bits.

  Ex.:o campo imediato de 16 bits do MIPS é convertido em 32 bits para aritmética

  Copie o bit mais significativo (o "bit de sinal") para os outros bits

  $0010 \leftarrow 00000010 = 2$

  $1010 \leftarrow 11111010 = -6$

Extensão de sinal (`lbu` versus `lb`, load byte):

Carrega 1 byte da memória para um registrador.

Obs.: `lhu` e `lh` (load half word, 16 bits).

$$x + \bar{x} = 111...111_2 = -1$$
$$x + \bar{x} + 1 = 0$$
$$\bar{x} + 1 = -x$$

Comparação de números com e sem sinal

Suponha que:

- $s0 armazene o número $11111111111111111111111111111111_2$
- $s1 armazene o número $00000000000000000000000000000001_2$

Quais os valores de $t0 e $t1 dado os comandos abaixo?

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

Logo: `$t0=1` e `$t1=0` pq?

Verificação de limite: com atalho!

Exemplo: Considere a verificação de um índice i que aponte para um elemento válido de `v[dim]`.

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

sltu $t0, $a1, $t2   # $t0=0 se $a1 (i)>=$t2 (dim) ou $a1 (i)<0
beq $t0, $zero, indice_fora_limite
```

Obs.: Tipos em C (32 bits)

- unsigned char e char (8 bits): intervalos de 0...255 e -128...127
- unsigned short e short (16 bits): intervalos de 0...65535 e -32768...32767
- unsigned int e int (32 bits): intervalos de $0...2^{32}-1$ e $-2^{31}...2^{31}-1$.