INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus Guarulhos

Estruturas de Dados 2

03 – Complemento de 1, Complemento de 2 e Operadores *bit_a_bit*

Antonio Angelo de Souza Tartaglia
angelot@ifsp.edu.br

Operações binárias

- Representação de números negativos – operações de adição e subtração
  - A adição é a mais básica das operações aritméticas, e quase a “única” coisa que os computadores fazem. Portanto, se computadores podem somar valores, também podem subtrair, multiplicar, dividir, calcular pagamentos de boletos, calcular a rota em um GPS, guiar foguetes até Marte, etc.
  - A adição em números binários é muito semelhante à adição de números decimais. Quando somamos 245 e 673, dividimos o problema em etapas mais simples. Cada etapa requer que apenas se some um par de dígitos decimais. Assim, em 245 + 673, o problema começaria a ser resolvido somando-se 5 + 3, depois 4 + 7 (vai 1), e por último 2 + 6 + (vem 1).

$$
\begin{array}{r}
{\color{red}1} \\
245 \\
+\ \ 673 \\
\hline
918
\end{array}
$$

Operações binárias

- Representação de números negativos – operações de adição e subtração

  - A grande vantagem da representação de números em formato binário sobre os decimais está em sua simplicidade:

+	0	1
0	0	1
1	1	10

0 + 0 é igual a 0.
0 + 1 é igual a 1.
1 + 0 é igual a 1.
0 + 1 é igual a 1 e "vai 1".

  - Podemos reescrever a tabela de adição com zeros à esquerda , para que cada resultado seja representado por um valor de 2 *bits*:

+	0	1
0	00	01
1	01	10

Vista dessa forma, a adição de um par de números binários resulta em 2 *bits*, que são chamados *bit* de soma e *bit* de "vai um"

Operações binárias

- Representação de números negativos – operações de adição e subtração

  - Como na adição decimal, somamos dois números binários coluna por coluna, começando pelo *bit* menos significativo, na coluna mais à direita:

```
 1    1
  01100101
+ 10110110
----------
 100011011
```

- Observe que, quando somamos a 3ª coluna a partir da direita, 1 é transportado para a próxima coluna. Isso acontece novamente na 6ª, 7ª e na 8ª colunas, contando a partir da direita.

Operações binárias

- Representação de números negativos – operações de adição e subtração

  - Sabemos que em C números inteiros ocupam 4 *bytes* ou 32 *bits* de memória, mas vamos ser razoáveis aqui, para uma melhor compreensão. Trabalharemos somente com valores de 1 *byte*, ou 8 *bits* (mais precisamente), afim de facilitar a compreensão e diminuir os tamanhos à representar.

  - Portanto, trabalharemos com números em representação binária, que possam variar de `0000 0000` a `1111 1111`.

  - Esses valores variam de `00h` a `FFh` em representação hexadecimal, ou de `0` a `255` em representação decimal.

Operações binárias

- Representação de números negativos – operações de adição e subtração

  - Uma adição marcha consistentemente da coluna de dígitos mais à direita, para a esquerda. Cada "vai 1" (*carry*) de uma coluna, é transportado e somado à próxima coluna. No entanto, não há "transporte" na subtração. Em vez disso, "tomamos emprestado" (*borrow*), e isso envolve um mecanismo diferente, um tipo de ida e volta um tanto confuso.

  - Por exemplo, um problema típico de subtração que usa a técnica de tomar emprestado:

$$\begin{array}{rr} & 253 \\ - & 176 \\ \hline & 77 \end{array}$$

Olhando para a coluna mais a direita, vemos que 6 é maior do que 3, então é necessário pedir emprestado da próxima coluna à esquerda, e essa coluna também precisa de um empréstimo. Então, realizando essa cálculo corretamente teremos um resultado de 77.

Operações binárias

- Representação de números negativos – operações de adição e subtração

  - Vamos usar agora uma pequena técnica que nos permite realizar a mesma subtração sem a necessidade de "pedir emprestado". Pode parecer um truque no início, mas é um primeiro passo crucial para entender como os números negativos são armazenados nos computadores.

  - Podemos representar a subtração como uma adição à um número negativo:

$$- 176 + 253$$

Operações binárias

- Representação de números negativos – operações de adição e subtração
  - Vamos colocar mais alguns números nesta expressão – um positivo e um negativo – para que estejamos somando uma série de 4 números:

`1000 - 176 + 253 - 1000`

  - Somando `1000` e em seguida subtraindo `1000`, não fará nenhuma diferença para o resultado. Sabemos que `1000` é `999 + 1`, então em vez de começar com `1000`, podemos começar com `999` e então somar `1` mais tarde. Assim, a sequencia fica ainda maior, mas continua equivalente:

`999 - 176 + 253 + 1 - 1000`

Operações binárias

- Representação de números negativos – operações de adição e subtração
  - Trabalhando da esquerda para a direita, O primeiro passo é uma subtração: 999 – 176. Nessa subtração não é necessário o empréstimo de nenhum dígito, então efetuar esse cálculo é fácil:

$$999 - 176 = 823$$

  - Subtrair um número de uma sequência de 9s, resulta em um número chamado de complemento de nove. O complemento de nove de 176 é 823, e vice e versa, o complemento de nove de 823 é 176. O interessante é o seguinte:

> Não importa qual seja o número, **calcular o complemento de nove nunca requer um empréstimo**.

Operações binárias

- Representação de números negativos – operações de adição e subtração
  - Os próximos dois passos envolvem apenas adição. Somando 253 à 823 teremos como resultado 1076:

$$1076 + 1 - 1000$$

  - E, em seguida somamos 1 e subtraímos 1000:

$$1076 + 1 - 1000 = 77$$

  - Que é a mesma resposta de antes, mas realizada **sem um único empréstimo** desagradável.

Operações binárias

- Representação de números negativos – operações de adição e subtração
  - Essa técnica também é usada para números binários, e na verdade, nesta notação se torna mais simples do que com números decimais.
  - O problema de subtração original foi:

```
  253
- 176
-----
  ???
```

  - Quando esses números são convertidos para binário, o problema se torna:

```
  1111 1101
- 1011 0000
-----------
  ???? ????
```

Operações binárias

- Representação de números negativos – operações de adição e subtração
  - Executando os mesmos passos que vimos para os decimais, mudamos os números para que o problema se torne um número negativo somado a um número positivo:

```
-10110000 + 11111101
   -176   +    253
```

  - Em seguida somamos 11111111 (que é 255 em decimal) no início e depois somar 00000001 (1 em decimal) e subtrair 100000000 (256 em decimal):

```
11111111 - 10110000 + 11111101 + 00000001 - 100000000
  255    -    176   +    253   +     1    -    256
```

Operações binárias

- Representação de números negativos – operações de adição e subtração

$$\underbrace{11111111 - 10110000}_{01001111} + 11111101 + 00000001 - 100000000$$

$$01001111 + 11111101 + 00000001 - 100000000$$

  - Em binário, essa primeira subtração não requer transporte porque o número está sendo subtraído de 11111111
  - Quando um número decimal é subtraído de uma cadeia de 9s, o resultado é chamado de complemento de 9. Com números binários, subtrair algo de uma sequência de 1s é chamado complemento de um.

Operações binárias

- Representação de números negativos – operações de adição e subtração
  - No entanto, observe que realmente não é necessário executar uma subtração para calcular o complemento de um. Observe esses dois números:

```
10110000
01001111
```

  - O complemento de `10110000` é `01001111`, e o complemento de `01001111` é `10110000`.
  - Os *bits* são apenas invertidos: cada *bit* `0` se torna `1` e cada *bit* `1` se torna `0`. Por esse motivo, complemento de um muitas vezes é chamado de inverso.

Operações binárias

- Representação de números negativos – operações de adição e subtração

  - O problema agora é:

$$01001111 + 11111101 + 00000001 - 100000000$$
$$79 + 253 + 1 - 256$$

  - Somando os dois primeiros números:

$$101001100 + 00000001 - 100000000$$
$$332 + 1 - 256$$

  - O resultado é um número de 9 *bits*, mas tudo bem, já resolveremos isso.

Operações binárias

- Representação de números negativos – operações de adição e subtração
  - Somar 1 é trivial:

```
101001100 + 00000001 - 100000000
   332    +     1    -    256
```

```
101001101 - 100000000
   333    -    256
```

Operações binárias

- Representação de números negativos – operações de adição e subtração

  - Agora tudo que resta é subtrair o equivalente binário de 256, que simplesmente descarta o *bit* mais a esquerda:

101001101 – 100000000

333 – 256

  - E dessa forma, obtemos o mesmo valor que obtivemos no inicio quando o cálculo foi realizado com valores representados em notação de base decimal:

01001101

77

Operações binárias

- O complemento de 2
  - Vimos como valores decimais são representados em notação binária, e como efetuar somas e subtrações em números binários .

```
0000 0000 → 0
0000 0001 → 1
0000 0010 → 2
  ......   ..
1111 1111 → 255
```

  - Mas como representar os números binários negativos no computador, se só temos posições de memória que aceitam *bits* `0` e `1`?
  - Não existe o conceito de sinal à frente do número binário. Para que funcione, o *bit* mais à esquerda (o mais significativo) é usado para informar o sinal (*signed*) do valor.

Operações binárias

- O complemento de 2

  - Dessa forma o valor armazenado em `1` *byte* é positivo se este *bit* mais significativo for `0`, e negativo se for `1`:

```
0000 0000 →   0
0000 0001 →   1
1000 0001 →  -1
```

  - Por esse motivo, não é mais possível representar 256 valores (com 8 *bits*) como fizemos anteriormente (*unsigned*), e ficamos com a faixa de valores restringida de `0` à `127`.

Operações binárias

- O complemento de 2

  - Só que há um problema:

```
0000 0000 → +0
1000 0000 → -0
```

  - Usando esse formato, por exemplo, se fosse necessário somar +1 com -1:

```
  0000 0001 →  +1
+ 1000 0001 →  -1
-----------------
  1000 0010 →  -2
```

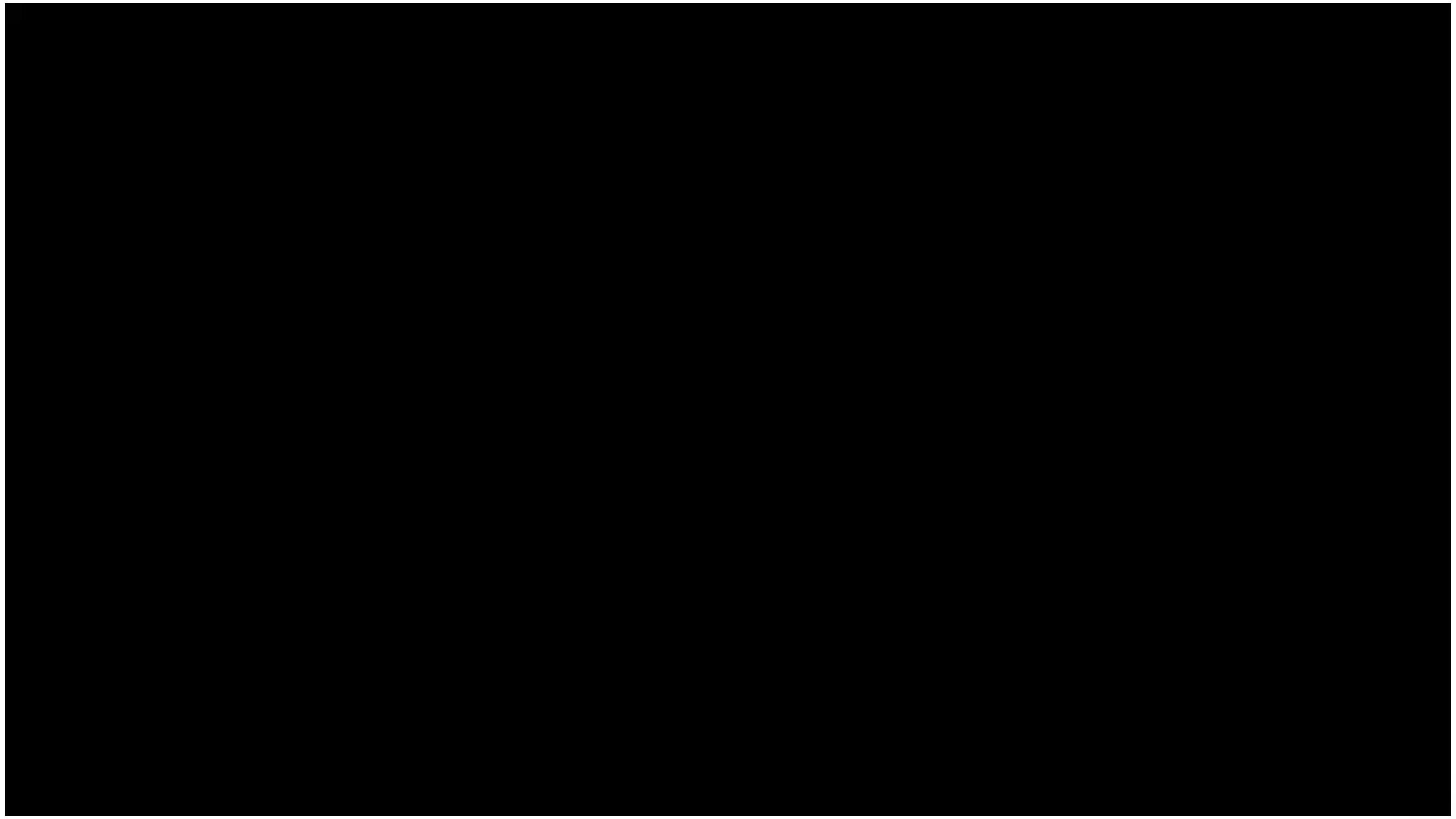
  - Cálculos utilizando este formato, não funcionam, e ainda ficaríamos com duas representações para o valor 0.

Operações binárias

- O complemento de 2
  - A partir dessa ideia, várias construções foram realizadas ao longo do tempo. As mais comuns são o complemento de 1 e o complemento de 2, sendo que esta última, foi a que realmente resolveu o problema de representação de números negativos em notação binária.
  - No complemento de 1, inverte-se todos os *bits* do valor, o que é `0` vira `1` e o que é `1` vira `0`, mas somente isso não resolve o problema, pois ficaríamos com 2 representações para o valor `0`.
  - Esse problema é resolvido com o método complemento de 2, que é realizado executando-se o método complemento de 1 e somando-se `1` ao resultado da inversão dos *bits*.

Operações binárias

- O complemento de 2
  - Para entender de uma forma mais fácil, temos o método do odômetro, dispositivo que registra a quilometragem de veículos:



Operações binárias

- O complemento de 2
  - Em um carro zero quilômetro, seu odômetro estaria com todos os seus dígitos em em 0, por exemplo:

00000000

  - Agora imagine que o veículo foi utilizado e a cada quilômetro o odômetro foi incrementado em 1:

00000001 → 00000002 → 00000003 → 00000004

  - Até

99999997 → 99999998 → 99999999 → 00000000

Operações binárias

- O complemento de 2

99999999	⇨	00000000

- O odômetro então “zera”, porque não é possível representar o valor 100000000 (9 dígitos). Ou seja, o estouro de base “sai” da possibilidade de representação (que são somente 8 posições), pois não há digito, ou espaço, correspondente para representa-lo.

Operações binárias

- O complemento de 2
  - Agora vamos supor que um veículo zero quilômetro, foi utilizado, e assim se deslocou por 2 quilômetros:

`00000001` ⇨ `00000002`

  - E que fosse possível, utilizando a marcha à ré, retroceder os dígitos do odômetro:

`00000002` ⇨ `00000001` ⇨ `00000000`

  - E que ai, continuássemos andando para trás:

`00000000` ⇨ `99999999` ⇨ `99999998` ⇨ `99999997`

Operações binárias

- O complemento de 2
  - Então podemos imaginar dessa forma:

00000000	⇨	99999999	⇨	99999998	⇨	99999997
0		-1		-2		-3

  - O complemento de 2 é exatamente isso.

Operações binárias

- O complemento de 2
  - Agora, representando essa ideia em notação binária:

```
0000 0010 →   2
0000 0001 →   1
0000 0000 →   0
1111 1111 →  -1
1111 1110 →  -2
1111 1101 →  -3
1111 1100 →  -4
```

Operações binárias

- O complemento de 2
  - Assim, para valores positivos, podemos incrementar até `0111 1111`, que corresponde ao valor `+127` em notação decimal, porque esta é a última representação possível em que o *bit* mais significativo (mais a esquerda) é `0`, ou seja, este *bit* define que o valor que vem a seguir nos próximos *bits*, é positivo.
  - Dessa forma, o *bit* mais significativo é utilizado para definir o sinal (*signed*) dos valores em notação binária, e somente os 7 *bits* restantes são destinados ao armazenamento do número. Com 7 *bits* só é possível armazenar números até `127`, neste caso em que estamos trabalhando com 8 *bits*.
  - De forma análoga, o maior número negativo possível de se representar em notação binária é `1000 0000`, que corresponde a `-128`.

Operações binárias

- O complemento de 2

  - Então para representações em notação binária, para valores de 8 *bits*, temos que o máximo valor positivo é `+127` e o máximo valor negativo `-128`. Essa diferença se deve ao fato de o valor `0` ser tratado como positivo, e assim teremos apenas uma representação para o valor `0`.

  - Com o complemento de 2, os cálculos funcionam corretamente:

```
 1
   0000 0001  →   1
 + 1111 1111  →  -1
 -------------------
   0000 0000  →   0
```

Operações binárias

- O complemento de 2
  - Outro exemplo: Em notação binária vamos executar o cálculo `5 – 3`. Sabemos que `5 – 3` é igual a `5 + (-3)` então convertemos o subtraendo (que é o valor 3) para o formato complemento de 2, invertendo todos os seus *bits* e somando 1 à inversão:

```
~ 0000 0011  →   3
  1111 1100
+ 0000 0001  →   1
  1111 1101  →  -3
```

Operador de negação (inversor), para manipulação de *bits.*

Operações binárias

- O complemento de 2

  - E ai então executamos a soma com o complemento de 2 do subtraendo (3), e executamos a soma:

```
  0000 0101  →   5
+ 1111 1101  →  -3
-----------------
  0000 0010  →   2
```

  - Ao nível do processador, temos instruções que são para valores com sinal (*signed*) e sem sinal (*unsigned*). O processador sempre executa uma soma. Assim, se o cálculo for uma subtração, internamente um dos valores é convertido para complemento de 2 e então uma soma é executada. Baseado nesse paralelo com o odômetro, fica mais fácil de entender como números negativos são representados em notação binária.

Tabela *Hash*

- Operadores *bit-a-bit*

- Ao contrário de muitas outras linguagens, C suporta um completo conjunto de operadores *bit-a-bit*. Uma vez que a linguagem foi projetada para substituir a linguagem *assembly,* na maioria das esferas de programação, era importante que ela tivesse a habilidade de suportar muitas das operações que naturalmente são realizadas pela linguagem *assembly*.

- Operação *bit-a-bit*, refere-se a testar, atribuir ou deslocar os *bits* efetivos em um *byte* (palavra), que correspondem aos tipos de dados `char` e `int` e suas variantes do padrão C. Essas operações são aplicadas aos *bits* individuais dos operandos.

- Operações *bit-a-bit* **não podem ser realizadas nos tipos `float`, `double`, `long double`, `void` ou outros tipos mais complexos**.

Tabela *Hash*

- A tabela abaixo, lista os operadores que se aplicam às operações *bit-a-bit*.

- Considere duas variáveis inteiras de o *bits* sem sinal:
  - A = 0001 0001 (17 em decimal)
  - B = 0110 0011 (99 em decimal)

```
0001 0001
0110 0011
0000 0001
```

```
0001 0001
0110 0011
0111 0011
```

```
0001 0001
0110 0011
0111 0010
```

Operador	Descrição	Exemplo	Resultado
&	AND bit-a-bit	A & B	0000 0001 (1)
|	OR bit-a-bit	A | B	0111 0011 (15)
^	XOR bit-a-bit	A ^ B	0111 0010 (114)
~	complemento de 1	~A	1110 1110 (238)
<<	desloca à esquerda N bits	A << 2	0100 0100 (68)
>>	desloca à direita N bits	A >> 2	0000 0100 (4)

```
0001 0001
1110 1110
```

```
0001 0001
0100 0100
```

```
0001 0001
0000 0100
```

Tabela *Hash*

- Operadores *bit-a-bit*
- As operações *bit-a-bit* AND, OR e NOT (complemento de um) são governadas pela mesma tabela verdade de seus equivalentes lógicos, exceto por trabalharem *bit-a-bit*.

A	B	S
0	0	0
0	1	0
1	0	0
1	1	1

Operação lógica
E – AND - **&**

A	B	S
0	0	0
0	1	1
1	0	1
1	1	1

Operação lógica
OU – OR - **|**

A	S
0	1
1	0

Operação lógica
NÃO – NOT - **~**

- A operação OR exclusivo (XOR - ^), tem a tabela verdade neste formato:

A	B	S
0	0	0
0	1	1
1	0	1
1	1	0

Tabela *Hash*

- Operadores *bit-a-bit*

- Como é possível observar na última tabela, o resultado de um OU exclusivo (XOR), é verdadeiro apenas, e, se exatamente apenas, um dos operandos for verdadeiro, caso contrário, o resultado será falso.

- Operadores *bit-a-bit* encontram aplicações mais frequentemente em "*drivers*" de dispositivos – como em roteadores, rotinas para manuseio de arquivos em disco e rotinas de impressão – porque as operações *bit-a-bit* mascaram certos *bits*, como o *bit* de paridade.

O *bit* de paridade, neste caso, confirma se o restante dos *bits* em um *byte* não se modificaram. É geralmente o *bit* mais significativo em cada *byte*.

Tabela *Hash*

- Operadores *bit-a-bit*
- O operador E – AND: Imagine-o como uma maneira de desligar *bits*, isto é, qualquer *bit* que é zero, em qualquer operando, faz com que o *bit* correspondente no resultado seja desligado.

- A paridade é indicada pelo oitavo *bit*, que é colocado em `0`. Executando-se uma operação AND com um *byte* em que os bits de `1` a `7` contém o valor binário `1` e o oitavo bit é zero. Por exemplo, a expressão `ch & 127`, significa executar uma operação AND *bit-a-bit* de `ch` com os *bits* que compõe o valor `127`. O resultado é que o oitavo bit de `ch` ficará zerado. Vamos assumir que `ch` tenha recebido o caractere “A” e que o bit de paridade tenha sido ativado:

bit de paridade

Executa AND bit-a-bit, que resulta em “A” sem paridade

```
  1100 0001
& 0111 1111
  ---------
  0100 0001
```

`ch` contém “A” com a paridade ligada

127 em binário.

Tabela *Hash*

- Operadores *bit-a-bit*
- O operador OU – OR: pode ser empregado para ligar *bits* individuais, como por exemplo a operação a seguir, 178 | 9 (178 OU 9):

```
  1011 0010   (178)
| 0000 1001     (9)
-------------------
  1011 1011   (187)
```

- Observe que com a utilização do operador OR, na realidade opera-se uma soma!

Tabela *Hash*

- Operadores *bit-a-bit*
- Um OU exclusivo - XOR: ativa um *bit* se, e somente se, os *bits* comparados forem diferentes. Por exemplo `127 ^ 120`:

```
  0111 1111   (127)
^ 0111 1000   (120)
-----------------------
  0000 0111     (7)
```

- Lembre-se de que os operadores lógicos e relacionais (**&&**, `||`, `!`), sempre produzem um resultado que é `0` ou `1` de acordo com as expressões avaliadas, enquanto as operações similares *bit-a-bit* (**&**, `|`, **~**) produzem quaisquer valores arbitrários, de acordo com a operação específica. Em outras palavras, operações *bit-a-bit* podem produzir valores diferentes de `0` e `1`, mas os operadores lógicos sempre produzem `0` e `1`.

Tabela *Hash*

- Operadores *bit-a-bit*

- Os operadores de deslocamento “ **>>** “ e “ **<<** “: Fazem com que todos os *bits* sejam deslocados para a direita ou esquerda, respectivamente. Um uso bastante interessante desses operadores, é a multiplicação ou divisão de um número inteiro de forma bastante rápida.
- Por exemplo, deslocar um valor um *bit* à esquerda efetivamente multiplica-o por 2. Dois deslocamentos à esquerda, multiplica-o por 4, e assim por diante:

```
<<, 1   0001 1100     (28)
        ------------------
        0011 1000     (56)
```

- Assim como o deslocamento de um *bit* a direita efetivamente divide-o por 2, dois *bits* à direta divide por 4, e assim por diante:

```
>>, 1   0001 1100     (28)
        ------------------
        0000 1110     (14)
```

Tabela *Hash*

- Operadores *bit-a-bit*
- Conforme os *bits* são deslocados para uma extremidade, zeros são adicionados na outra.
- O deslocamento, não é uma rotação, ou seja, os *bits* que saem por uma extremidade, não voltam para a outra. Os *bits* deslocados são perdidos e zeros são colocados em seu lugar.

unsigned char x;	x a cada execução	valor de x
x = 7;	`0000 0111`	7
x = x << 1;	`0000 1110`	14
x = x << 3;	`0111 0000`	112
x = x << 2;	`1100 0000`	192
x = x >> 1;	`0110 0000`	96
x = x >> 2;	`0001 1000`	24

Como cada deslocamento à esquerda, multiplica por 2, observe que se perdeu informação após a instrução `x = x << 2;`

Como cada deslocamento à direita, divide por 2, observe que divisões subsequentes não trazem de volta os *bits* anteriormente perdidos.

Tabela *Hash*

- Operadores *bit-a-bit*

- Devido à maneira como os números negativos são representados dentro do computador, deve-se tomar cuidado ao utilizar o deslocamento para multiplicação ou divisão. Um valor 1 colocado na posição do *bit* mais significativo, fará com que o computador assuma que está tratando com um número negativo.

```c
//Arquivo hashTable.c - Chave: método da divisão
int chaveDivisao(int chave, int TABLE_SIZE){
    return (chave & 0x7FFFFFFF) % TABLE_SIZE;
}
```

- Por esse motivo é executada uma operação AND *bit-a-bit*. E assim, elimina-se a possibilidade de um *overflow* com a obtenção de um número negativo.

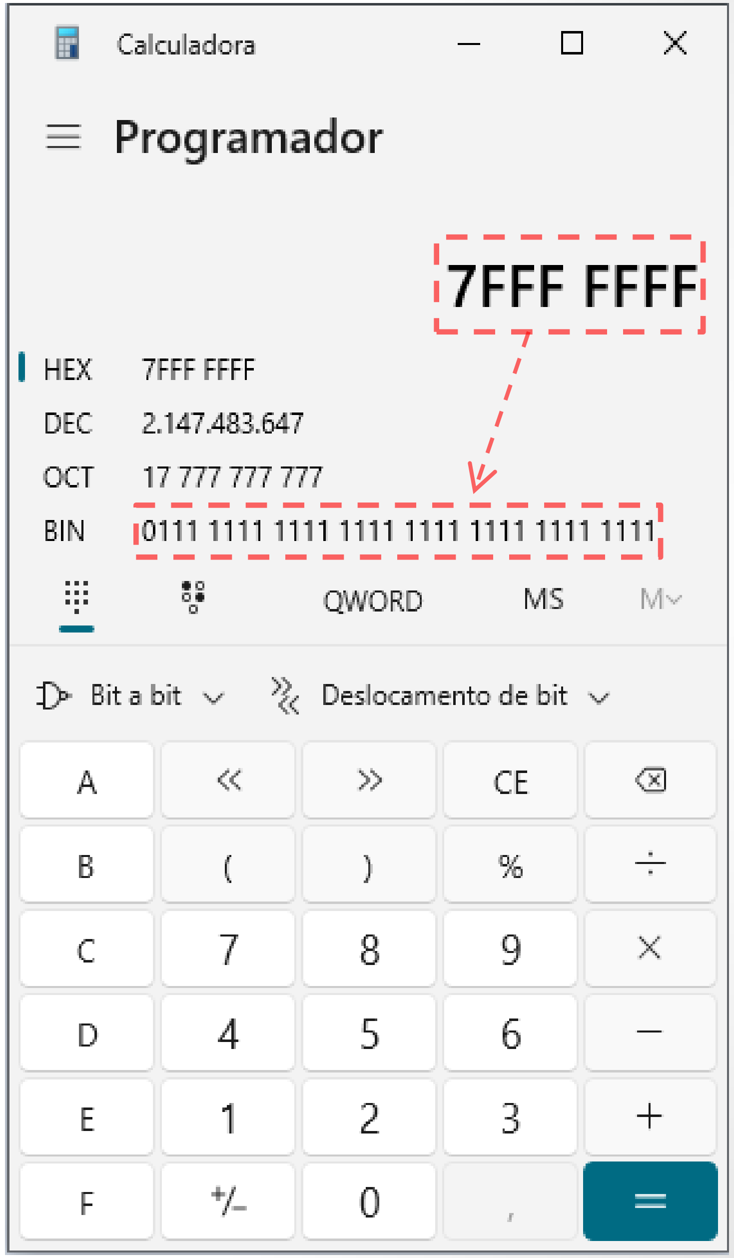


Tabela *Hash*

- Operadores *bit-a-bit*

- O operador NÃO – NOT, ou operador **complemento de 1**, serve basicamente para inverter os *bits*, 0s e 1s, que compõe o número:

```
~ 0001 1100      (28)
  ----------------------
  1110 0011     (227)
```

- Precedência dos operadores *bia-a-bit*:

<table>
<tr><th>Precedência</th><th>operador</th></tr>
<tr><td>maior</td><td>~</td></tr>
<tr><td rowspan="3"></td><td><< e >></td></tr>
<tr><td>&</td></tr>
<tr><td>^</td></tr>
<tr><td>menor</td><td>|</td></tr>
</table>