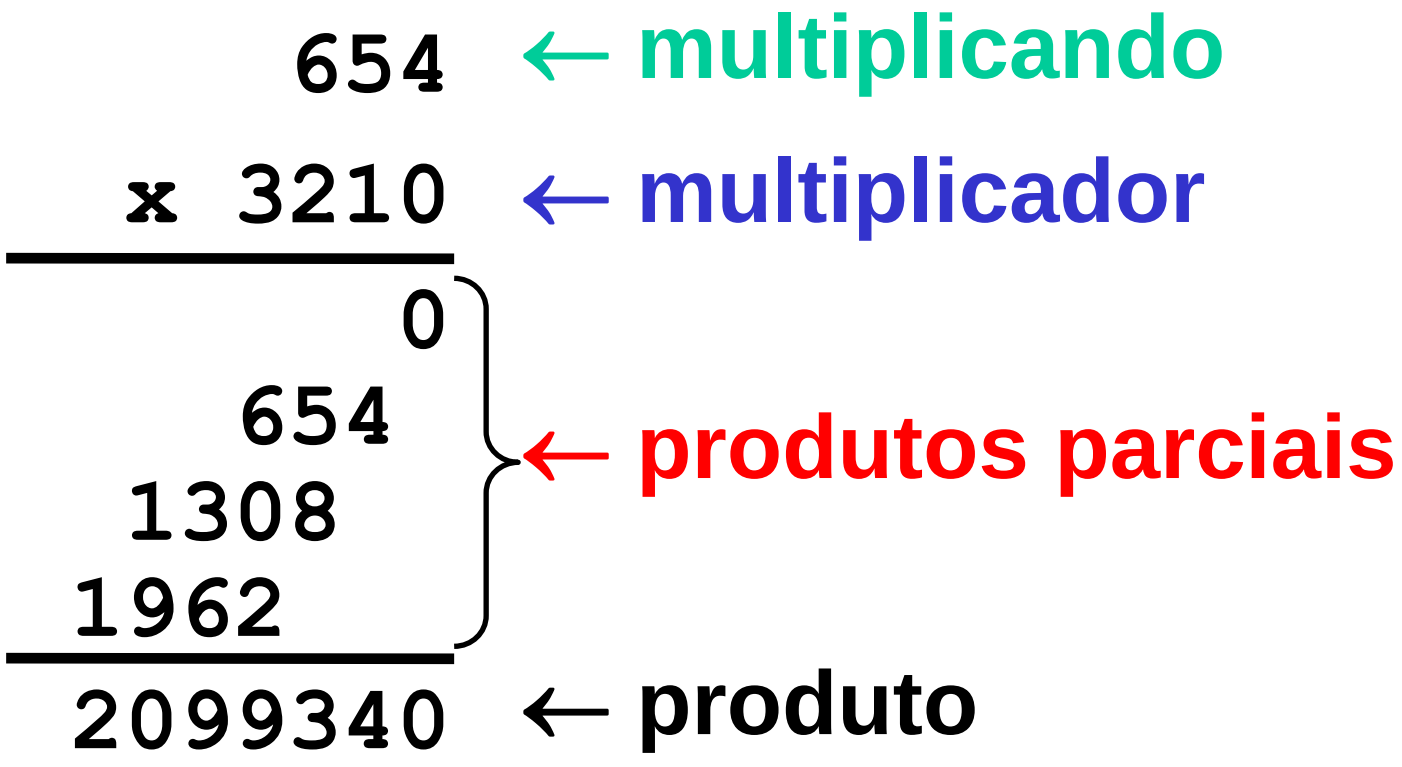
Multiplicação em binário

Uma aplicação para o Ahmes

Pensando bem ... como se multiplica ?



(3 dígitos x 4 dígitos = 7 dígitos !)

Pensando bem ... como se multiplica ?

```
     654
   x 3210
---------
       0 = 0 x    1 x 654
    6540 = 1 x   10 x 654
  130800 = 2 x  100 x 654
 1962000 = 3 x 1000 x 654
---------
 2099340
```

Em binário, as regras são as mesmas ! (muda somente a base)

Exemplo: multiplicação de int. positivos

```
      111010
    x 110101
------------
      111010
     000000
    111010
   000000
  111010
 111010
------------
110000000010
```

(6 dígitos x 6 dígitos = 12 dígitos !)

Em binário, as regras são as mesmas ! (muda somente a base)

Exemplo: multiplicação de int. positivos

```
          111010
        x 110101
--------------------
          111010 = 1 x      1 x 111010
         0000000 = 0 x     10 x 111010
        11101000 = 1 x    100 x 111010
       000000000 = 0 x   1000 x 111010
      111010000  = 1 x  10000 x 111010
     1110100000  = 1 x 100000 x 111010
--------------------
  110000000010
```

Mas, como os dígitos do multiplicador só podem ser 0 ou 1, só existem 2 valores possíveis para os produtos parciais:

```
        111010
      x 110101
--------------
        111010   ← bit multiplicador = 1
       000000    ← bit multiplicador = 0
      111010
     000000
    111010
   111010
--------------
 110000000010
```

Os produtos parciais podem ser acumulados à medida em que são calculados; o resultado obtido é o mesmo

← quando o bit do multiplicador é 0, não é necessário somar nada

← multiplicando é deslocado para a esquerda em 1 bit após fazer cada adição

Depois de cada adição, o 'vai um' deve ser incorporado ao produto parcial como seu bit mais significativo (lembrar que na primeira linha o multiplicando foi somado a um "produto parcial inicial" igual a 0)

Algoritmo básico – generalizado para n digitos

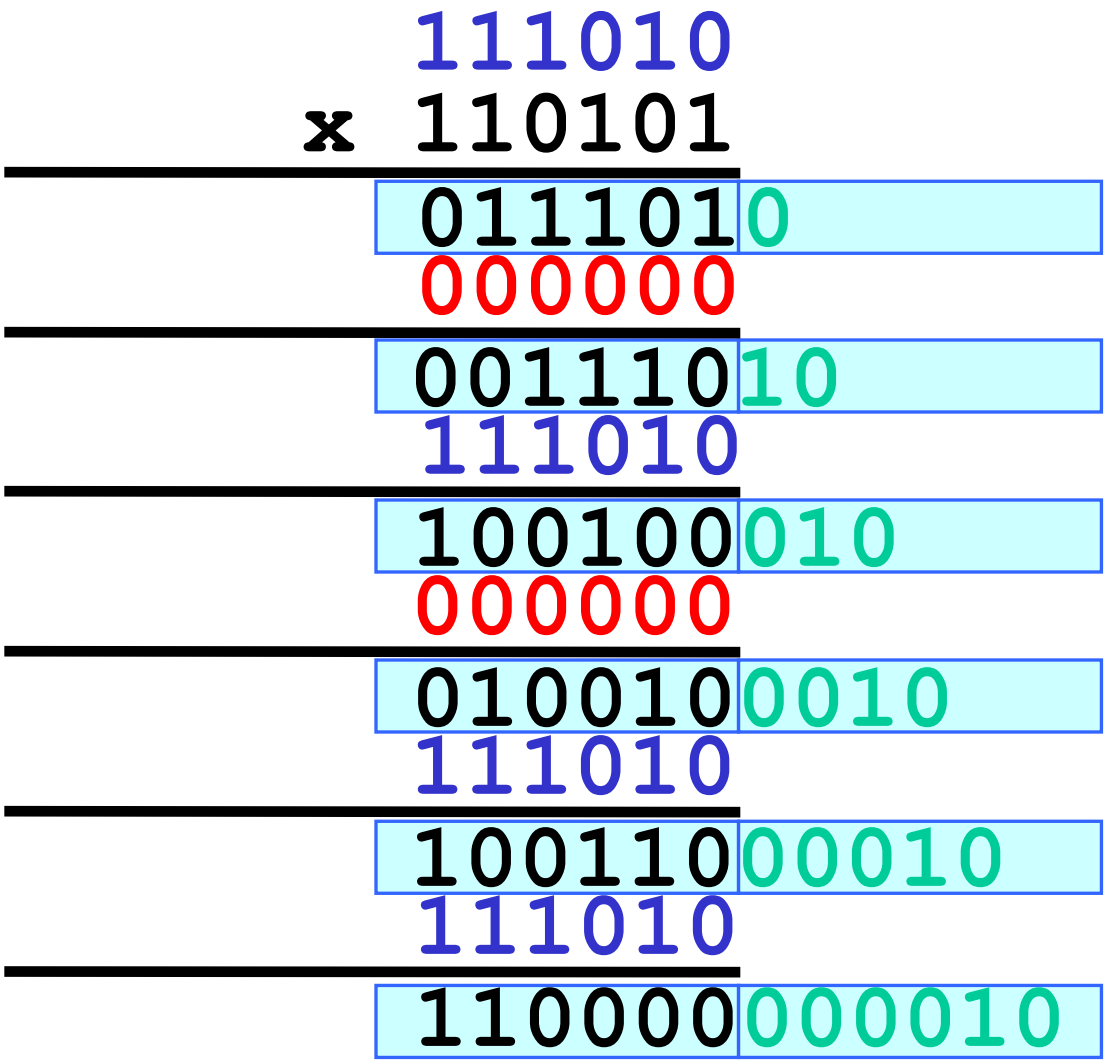
1. Início: i ← 0, produto ← 0
2. Se o bit de ordem ‘i’ do multiplicador for zero, ir para 4 (otimização ...)
3. Somar o multiplicando ao produto (produto ← produto + multiplicando)
4. Deslocar o multiplicando para a esquerda em 1 bit (multiplicando ← multiplicando x 2)
5. Incrementar ‘i’ de uma unidade (i ← i + 1)
6. Se ‘i’ for menor que n, ir para 2
7. Terminar

Cada vez que um produto parcial é calculado, fica definido um dos dígitos menos significativos do produto final e este não é mais afetado pelas somas seguintes:

```
      111010
    x 110101
------------
     0111010
     000000
------------
    00111010
    111010
------------
   100100010
   000000
------------
  0100100010
   111010
------------
 10011000010
  111010
------------
110000000010
```

← lembre-se: aqui já foi feita uma adição; o produto parcial inicial era zero

Logo, deslocando as somas parciais um dígito para a direita a cada etapa, também poderíamos representar o processo de cálculo assim:



- duas “variáveis”
- 6 dígitos cada uma
- 6 etapas
- a cada etapa, deslocar para a direita

Algoritmo adaptado para uso em computador (P e p são as duas metades do produto, M = multiplicando e m = multiplicador, c = carry)

1. Início: i $\leftarrow$ 0 , P $\leftarrow$ 0 , p $\leftarrow$ 0
2. Se o bit de ordem ‘i’ de m for zero, fazer c $\leftarrow$ 0 e ir para 4
3. Somar M a P (P $\leftarrow$ P + M); c $\leftarrow$ ‘vai um’ da soma
4. Deslocar os 2n bits do produto para a direita e inserir c como bit mais significativo do produto
   (P p) $\leftarrow$ deslocamento p/direita de (c P p)
5. Incrementar ‘i’ de uma unidade (i $\leftarrow$ i + 1)
6. Se ‘i’ for menor que n, ir para 2
7. Terminar

Algoritmo melhorado para uso em computador (P e p são as duas metades do produto, M = multiplicando e m = multiplicador, c = carry)

1. Início: i ← ‘n’ , P ← 0
2. Deslocar m para a direita junto com o carry
   (m c ← deslocamento p/direita de m)
3. Se c = 0, ir para 5
4. Somar M a P (P ← P + M); c ← ‘vai um’ da soma
5. Deslocar os 2n bits do produto para a direita e inserir c como bit mais significativo do produto
   (P p) ← deslocamento p/direita de (c P p)
6. Decrementar ‘i’ de uma unidade (i ← i - 1)
7. Se ‘i’ não for zero, ir para 2
8. Terminar. Resultado em (P p)

Algoritmo adaptado para uso no Ahmes (P e p são as duas metades do produto, M = multiplicando e m = multiplicador, c = carry)

1. Início: i ← ‘n’ , P ← 0
2. Deslocar m para a direita junto com o carry
   m c ← SHR (m)
3. Se c = 0, ir para 5
4. Somar M a P (P ← P + M); c ← ‘vai um’ da soma
5. Deslocar os 2n bits do produto para a direita e inserir c como bit mais significativo do produto
   (P c) ← ROR (c P) e (p c) ← ROR (c p)
6. Decrementar ‘i’ de uma unidade (i ← i - 1)
7. Se ‘i’ não for zero, ir para 2
8. Terminar. Resultado em (P p)

Implementação no Ahmes

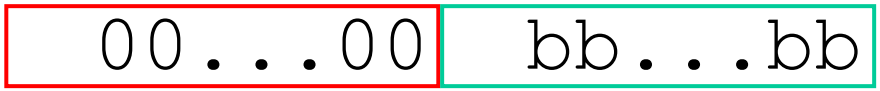
```
  0    32 136    LDA 136
  2    16 132    STA 132
  4    32 134    LDA 134
  6    16 130    STA 130
  8    32 129    LDA 129
 10    16 133    STA 133
 12    32 133    LDA 133
 14   224        SHR
 15    16 133    STA 133
 17   180  25    JNC 25
 19    32 128    LDA 130
 21    48 130    ADD 128
 23    16 130    STA 130
 25    32 130    LDA 130
 27   226        ROR
 28    16 130    STA 130
 30    32 131    LDA 131
 32   226        ROR
 33    16 131    STA 131
 35    32 132    LDA 132
 37   112 135    SUB 135
 39    16 132    STA 132
 41   164  12    JNZ 12
 43   240        HLT

128     0        M
129     0        m
130     0        P
131     0        p
132     0        i
133     0        m'
134     0        =0
135     1        =1
136     8        =8
```

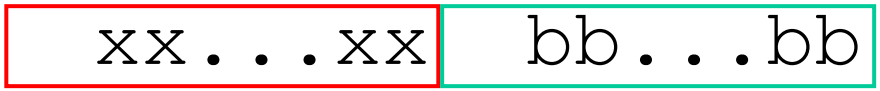
Truncando o produto (P p) para n bits

(multiplicação de valores positivos !)

Inteiros positivos

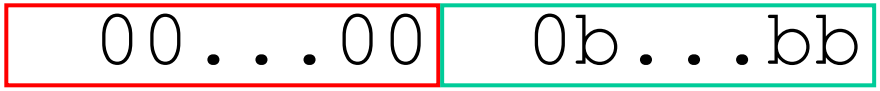


pode ser truncado

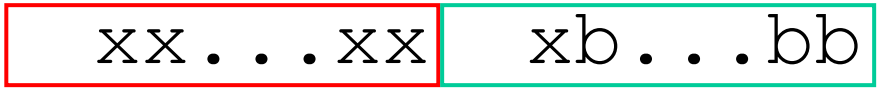


não pode ser truncado se algum x ≠ 0

Complemento de 2



pode ser truncado



não pode ser truncado se algum x ≠ 0

O que pode ser melhorado (1) ?

- Depois do teste de carry (JNC), os dois ramos iniciam com a instrução (LDA 130). Como LDA não afeta os códigos de condição B, C e V, podemos colocá-la antes do teste do carry, “economizando” uma instrução:

```
15  STA 133
17  JNC 25
19  LDA 130
21  ADD 128
23  STA 130
25  LDA 130
27  ROR
28  STA 130
30  LDA 131
32  ROR
33  STA 131
...
```

```
15  STA 133
17  LDA 130
19  JNC 25
21  ADD 128
23  STA 130
25  ROR
26  STA 130
28  LDA 131
30  ROR
31  STA 131
...
```

O que pode ser melhorado (2) ?

- Com a modificação feita, a instrução STA no na palavra 23 não é mais necessária, pois em seguida o AC é girado para a direita e só armazenado novamente na palavra 130; nova “economia”:

```
15  STA 133        15  STA 133        15  STA 133
17  JNC 25         17  LDA 130        17  LDA 130
19  LDA 130        19  JNC 25         19  JNC 23
21  ADD 128        21  ADD 128        21  ADD 128
23  STA 130        23  STA 130        23  ROR
25  LDA 130        25  ROR            24  STA 130
27  ROR            26  STA 130        26  LDA 131
28  STA 130        28  LDA 131        28  ROR
30  LDA 131        30  ROR            29  STA 131
32  ROR            31  STA 131        ...
33  STA 131  ...
...
```

O que pode ser melhorado (3) ?

- **Considerando os deslocamentos do multiplicador (m) e dos resultados parciais (P p) para a direita, durante o processo de multiplicação, pode-se usar a mesma variável para guardar 'm' e 'p':**

Mas, para economizar esta palavra, o algoritmo fica novamente mais longo e precisamos de mais uma constante. Quanto ao desempenho, este melhora.

Compartilhamento de 1 palavra por p e m’

```
 0   32 136   LDA 136
 2   16 132   STA 132
 4   32 134   LDA 134
 6   16 130   STA 130
 8   32 129   LDA 129
10   16 131   STA 131
12   32 131   LDA 131
14  224       SHR
15   16 131   STA 131
17   32 130   LDA 130
19  180  23    JNC 23
21   48 128   ADD 128
23  226       ROR
24   16 130   STA 130
26  180  34    JNC 34
28   32 131   LDA 131
30   64 137   OR  137
```

```
32   16 131   STA 131
34   32 132   LDA 132
36  112 135   SUB 135
38   16 132   STA 132
40  164  12    JNZ 12
42  240       HLT

128   0       M
129   0       m
130   0       P
131   0       p e m’
132   0       i
134   0       =0
135   1       =1
136   8       =8
137 128       =128
```

Multiplicação de números negativos (em complemento de 2)

Inteiro positivo

$$10101 = 1.2^4+0.2^3+1.2^2+0.2^1+1.2^0$$
$$= 16 + 0 + 4 + 0 + 1$$
$$= 21$$

Complemento de 2

$$10101 = -1.2^4+0.2^3+1.2^2+0.2^1+1.2^0$$
$$= -16 + 0 + 4 + 0 + 1$$
$$= -11$$

Conseqüência: na última iteração, se o bit mais significativo do multiplicador for um, o multiplicando deve ser subtraído do resultado parcial.

Multiplicação de números negativos (em complemento de 2)

```
    111010        -6
  x 110101      x -11
-----------
   1111010
   0000000
-----------
  11111010
  11101000
-----------
 111100010
 000000000
-----------
1111100010
1110100000
-----------
11110000010
-11101000000
-----------
00000001000010      = +66
```

Depois de cada adição, o novo bit mais significativo será:

- igual ao **sinal anterior** se a soma/subtração não provocou estouro (ou se não houve soma/subtração), ou
- o complemento do sinal anterior se a soma/subtração provocou estouro (isto corrige o estouro)

(no Ahmes, a ocorrência de estouro liga o código de condição V ... ☺)

Multiplicação de números negativos (em complemento de 2)

```
      011010        +26
    x 110101      x -11
-------------
   0011010
   0000000
-------------
  00011010
  01101000
-------------
  01000010
  00000000
-------------
  00100010
  01101000
-------------
 000000100010
-011010000000
-------------
 111011100010      = -286
```

Depois de cada adição, o novo bit mais significativo será:

- igual ao sinal anterior se a soma/subtração não provocou estouro (ou se não houve soma/subtração), ou
- o complemento do sinal anterior se a soma/subtração provocou estouro (isto corrige o estouro)

(no Ahmes, a ocorrência de estouro liga o código de condição V ... ☺)

Multiplicação de números negativos (em complemento de 2)

Portanto, existem quatro situações a considerar:

- resultado positivo, sem estouro (N=0, V=0)
  → o novo bit deve ser **0** (SHR P)
- resultado positivo, com estouro (N=0, V=1)
  → o novo bit deve ser **1** (SHR P, OR #128)
- resultado negativo, sem estouro (N=1, V=0)
  → o novo bit deve ser **1** (idem ao anterior)
- resultado negativo, com estouro (N=1, V=1)
  → o novo bit deve ser **0** (SHR P)

Obs: após isto, ainda é preciso fazer um ROR para acertar os bits menos significativos do produto (p) !

Soluções alternativas

- Para o Ahmes

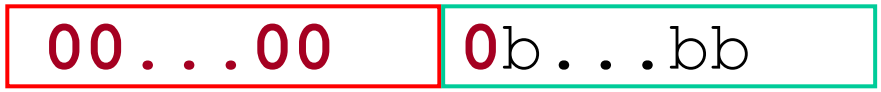
  Converter os fatores em positivos, multiplicar usando o algoritmo para inteiros positivos e depois acertar o sinal do produto de acordo com os sinais dos fatores

- Genérica

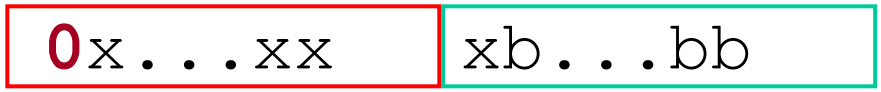
  Método de Booth

Truncando o produto (P p) para n bits
(multiplicação de valores em complemento de 2)

Complemento de 2, positivo



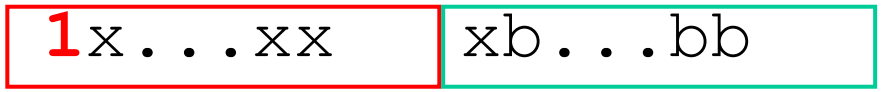
pode ser truncado



não pode ser truncado se algum x ≠ **0**

Complemento de 2, negativo

pode ser truncado



não pode ser truncado se algum x ≠ **1**